UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE INGENIERÍA ELÉCTRICA

ARQUITECTURA DE DOMINIO ESPECIFICO PARA REDES NEURONALES RECURRENTES UTILIZANDO LA ISA DE RISC-V

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL ELÉCTRICO

TOMÁS ARTURO LETELIER ZAMORA

PROFESOR GUÍA:
FRANCISCO RIVERA SERRANO

MIEMBROS DE LA COMISIÓN:
ANDRÉS CABA RUTTE
MARCOS DIAZ QUEZADA

SANTIAGO DE CHILE
2022

Resumen

Las redes Neuronales Artificiales son ampliamente utilizadas en la actualidad debido a su capacidad de resolver problemas a partir de los datos disponibles de este. Estas, sin embargo, conllevan una necesidad de una gran capacidad computacional debido al gran número de cálculos matemáticos requeridos para el funcionamiento de estas redes.

A partir de esto, se busca explorar la reciente *ISA open-source*: *RISC-V*, realizando una aplicación de dominio especifico orientado a redes neuronales sobre este tipo de procesador. Para esto, en este trabajo de título se realiza la implementación de una red neuronal utilizando esta *ISA*, específicamente, una red neuronal recurrente *LSTM*. El objetivo consiste, entonces, en realizar mediciones de tiempo, respecto al entrenamiento y ejecución de la implementación de esta red, y comparar estos tiempos a mediciones obtenidas utilizando los procesadores más comunes: *CPU* y *GPU*.

Entre los resultados obtenidos se logra realizar una implementación funcional de una red *LSTM* en *RISC-V*, donde los tiempos obtenidos son comparables a aquellos obtenidos por una *CPU*, sin embargo el procesador *GPU* sigue siendo considerablemente más eficiente, al menos respecto a la implementación realizada, la cual puede ser mejorara aún más utilizando instrucciones atómicas de *RISC-V* o, incluso, a través de la implementación de instrucciones personalizadas.

Tabla de Contenido

Índice de Ilustraciones

Capítulo 1

Introducción

1.1. Motivación

Actualmente, nos encontramos en una era donde la informática y la automatización están cada vez más presentes en nuestro alrededor y en nuestra vida, desde el actual desarrollo de automóviles con piloto automático hasta la gran automatización de procesos en distintas empresas, incluso, empezando a llegar a los hogares de las personas con el creciente interés por los *smart home* [3]. Muchas de estas nuevas tecnologías utilizan, usualmente, alguna forma de inteligencia computacional, ya sea desde programas relativamente simples hasta las más recientemente usadas redes neuronales.

Las Redes Neuronales Artificiales son ampliamente utilizadas en la actualidad, debido a su capacidad de resolver problemas no lineales, así como encontrar soluciones a través de entrenamiento, en donde estas redes son capaces de ser moldeadas para obtener el funcionamiento deseado. Sin embargo, para lograr un entrenamiento efectivo, se requiere no solo una gran cantidad de datos, sino que también una enorme capacidad computacional, debido al gran número de operaciones matemáticas que se realizan para ajustarse apropiadamente al caso a entrenar. Esto se vuelve aún más crítico al trabajar con redes neuronales de alta complejidad o "profundidad", las cuales son capaces de resolver problemas de mayor dificultad pero toman tiempos exponencialmente mayores de entrenamiento. Es por eso que, en el área de redes neuronales artificiales, encontrar las formas más eficientes de trabajar con estas es una problemática recurrente.

Existen variados tipos y arquitecturas de redes neuronales, sin embargo, para esta memoria se trabaja, específicamente, sobre Redes Neuronales Recurrentes (RNN). Este tipo de red es particularmente útil debido a su capacidad de memoria de corto plazo y secuencialidad en el tiempo, siendo así ampliamente utilizada para problemas de información secuencial, como, por ejemplo, reconocimiento de lenguaje natural [4][5][6]. Particularmente, el trabajo estará orientado a redes *Long-Short Term Memory* ($LSTM$) [1], la cuales son una versión mejorada de RNN capaz de mantener memoria a largo plazo de mejor manera, siendo estas las más comúnmente usadas en la actualidad.

Una de las soluciones más comunes para optimizar las velocidades de entrenamiento y

ejecución de redes neuronales es el uso de *GPU* como procesador principal [7] debido a su capacidad de paralelismo, siendo capaz de realizar un gran número de cálculos matemáticos de forma simultánea. Sin embargo, estos procesadores están diseñados con el objetivo de ser utilizados, principalmente, para el cálculo de gráficos computacionales por lo que, aunque generan una notable disminución de tiempos, quedan ciertas necesidades de las redes neuronales que no son optimizadas por estos procesadores.

Por esto surgen las Arquitecturas de Dominio Específico (*DSA*), las cuales consisten en arquitecturas de hardware o software diseñadas para un dominio o una tarea en específico, con lo que son capaces de realizar aquella tarea para la cual están diseñados a mayor eficiencia en comparación a hardware más general, a cambio de perder la capacidad de poder realizar otras tareas.

En esto aparece *RISC-V*. Este es un tipo de procesador relativamente reciente con una *ISA open-source*, la cual posee herramientas que permiten tanto aprovechar de manera más completa el procesador mismo, como crear extensiones personalizadas para optimizar aun más el funcionamiento de alguna aplicación.

1.2. Objetivos del trabajo de titulo

1.2.1. Objetivo general

El tema de esta memoria consiste en utilizar un procesador *RISC-V* para realizar una aplicación de dominio especifico, en donde se implementará directamente el funcionamiento de entrenamiento y ejecución de una red *LSTM* sobre un procesador. El objetivo de esto es explorar este tipo de procesadores y la *ISA* respectiva para aplicaciones de inteligencia artificial, y comparar su funcionamiento respecto a una *CPU* y una *GPU*, siendo estos los tipos de procesador más comúnmente utilizados para este tipo de aplicaciones.

1.2.2. Objetivos específicos

Para lograr lo planteado anteriormente, se buscan realizar los objetivos específicos listados a continuación, los cuales son completados de forma cronológica:

1. Diseñar una red *LSTM* en *Python* para ser utilizada como punto de comparación
2. Implementar el funcionamiento de la ejecución de la red *LSTM* anterior utilizando la *ISA RISC-V*
3. Implementar en *RISC-V*, a partir de lo anterior, el entrenamiento de la red *LSTM*
4. Obtener comparaciones de funcionamiento y tiempos entre la implementación en *Python* sobre una *CPU* y una *GPU* y las implementaciones realizadas en *RISC-V*

1.3. Estructura del informe

Este informe se encuentra dividido en 4 secciones. Primero, se presenta el marco teórico, en donde se entra a hablar sobre los conceptos y conocimientos necesarios como para entender el trabajo realizado, así como de las herramientas utilizadas para este trabajo. Posteriormente, se presenta el Estado del arte, donde brevemente se mencionan *papers* importantes para la realización de este trabajo. Luego, la gran parte del informe es dedicada a la sección de metodología y desarrollo, donde se describe el trabajo realizado, el procedimiento para realizar este y los códigos desarrollados para este trabajo. Posteriormente, se entra en la sección de resultados, donde se realiza un análisis sobre las mediciones tanto de funcionamiento como de tiempos obtenidos para las distintas implementaciones. Finalmente, se pasa a las conclusiones, donde se comenta sobre los resultados obtenidos, así como se mencionan posibles cambios o mejoras para trabajos futuros.

Se presenta, además de todo lo anterior, un Anexo el cual contiene una versión abreviada de los códigos implementados, seguido de un listado con las instrucciones disponibles en la *ISA RISC-V*.

Capítulo 2

Marco teórico

2.1. Redes Neuronales

2.1.1. Redes Neuronales Artificiales

Una red neuronal artificial se compone de unidades simples de procesamiento, o nodos, y las conexiones entre estos. Todas las conexiones entre nodos poseen un "peso", el cual determina qué tanto afecta un nodo al otro. Además de esto, se define un conjunto de nodos como nodos de entrada (*input*) y otro como de salida (*output*).

Al asignarle un valor, o activación, a cada nodo de entrada, este valor es propagado a través de las conexiones entre los nodos de la red donde se realiza un mapeo funcional del conjunto de valores asignados como entrada, a otro conjunto asignado a los nodos de salida. Este mapeo es guardado tanto en los pesos de las conexiones como en las funciones de activación de cada nodo, los cuales son ajustados mediante algún método de entrenamiento, generalmente, entregando valores de entrada y comparando los valores de salida obtenidos de la red con los valores de salida esperados [8].

Existen variados tipos de redes neuronales, donde distintas arquitecturas definen la forma en que se ordenan los nodos y las conexiones entre estos. El caso más estándar, una red neuronal *"Feed-Forward"* (Figura 2.1), se estructura con una capa con nodos de entrada, un número determinado de capas ocultas intermedias sucesivas y, finalmente, una capa de salida, teniendo los nodos de cada capa "conectados" con los nodos de la capa anterior y la siguiente.

De esta forma, la información entregada a los nodos de entrada es propagada a través de los nodos de cada capa, hasta llegar a los nodos de salida, cuyos valores corresponden al mapeo respectivo.

Para poder entrenar una red neuronal se requiere un gran número de datos correspondientes al funcionamiento esperado de la red: datos correspondientes al tipo de información que recibirá esta red y los resultados esperados que debiese entregar a partir de estos. Estos

datos son entonces separados en un conjunto de entrenamiento, validación y prueba.

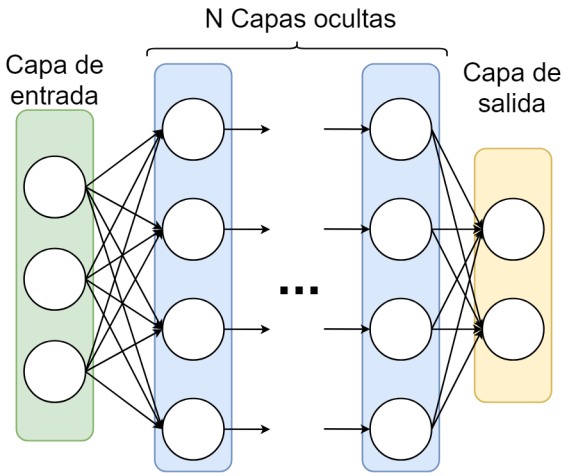

Figura 2.1: Ejemplo de la estructura típica de una red neuronal artificial *Feed-Forward*.

A partir del conjunto de entrenamiento, se alimenta la red con estos, uno por uno, obteniendo resultados de la red, los cuales son comparados con los resultados *esperados* a través de una función de error predefinida, donde a partir de la diferencia de estos dos se calcula, para cada parámetro de la red (pesos y *biases*), qué tanto influyen en el error del resultado, lo cual consiste en el *gradiente*. A partir de estos gradientes se ajustan los valores de los parámetros de la red, utilizando algún algoritmo predefinido. Este proceso es realizado con los datos de entrenamiento múltiples veces hasta que la red se considere suficientemente entrenada. Este punto se define en base al algoritmo de entrenamiento a utilizar, aunque este suele ser aquel donde se empieza a perder precisión de predicción respecto a los datos del conjunto de validación (los cuales al alimentarlos a la red no se modifican los pesos, solo se comparan los resultados entregados y esperados).

2.1.2. Redes Neuronales Recurrentes

Para esta memoria el foco fueron las Redes Neuronales Recurrentes o *RNN* (*Recurrent Neural Network*). La principal particularidad de este tipo de redes es que solo poseen una capa oculta, sin embargo, esta es retroalimentada a sí misma. Además de esto, la información con la que se trabaja en este tipo de redes consiste en agrupaciones de datos secuenciales (o secuencias de datos, cada uno con sus distintas características), a diferencia de conjuntos de datos independientes

De esta forma, las *RNN* poseen memoria respecto a la información que reciben, lo que les permite contextualizar los datos que se les entregan, pues para una secuencia de datos la salida de la red dependerá tanto de la información de cada dato como de los datos anteriores. Una forma de visualizar una *RNN* se muestra en la Figura 2.2, donde a partir de una secuencia de datos se puede imaginar la red como replicándose a sí misma por tantos datos se posean en la secuencia entregada, en donde, para todos los casos, los valores de U, V y W son los mismos.

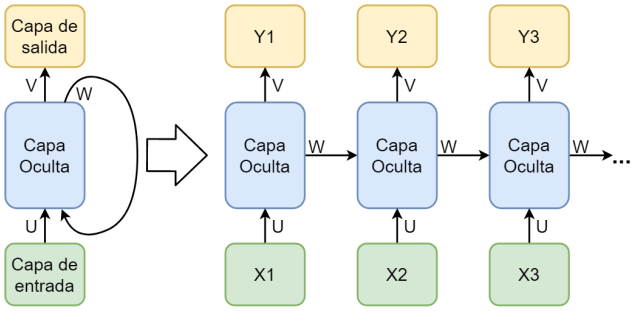

Figura 2.2: Ejemplo de la estructura de una red neuronal recurrente simple.

Como cualquier red neuronal, las *RNN* poseen un significante costo computacional relacionado a la complejidad de la red, debido al gran número de operaciones matemáticas que realiza, en especial, a la hora de entrenarla [6].

2.1.3. Redes *Long-Short Term Memory*

Existen distintas formas de implementar una *RNN*. Actualmente, la más utilizada corresponde a redes *Long-Short Term Memory* (*LSTM*), representada en la Figura 2.3. Estas son una versión mejorada del concepto base de *RNN*, mostrado previamente, las cuales presentan un "canal" de memoria extra denominado C, cuyo propósito es atacar el problema del *desvanecimiento del gradiente*, donde la información pierde importancia o "impacto" a través del tiempo, lo cual ocurre si la secuencia de datos es relativamente larga.

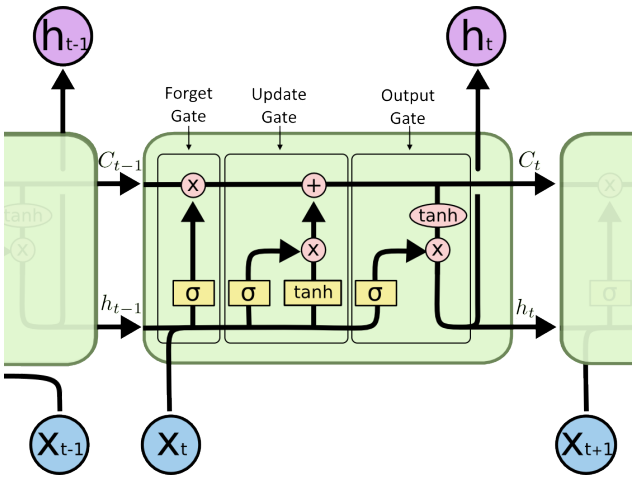

Figura 2.3: Representación visual de una capa oculta "tiempo" t de una red *LSTM*. Obtenida originalmente de [1].

Dada entonces una secuencia de datos, en una *LSTM* se realizan 3 pasos para cada dato de la secuencia, utilizando la información h_{t-1} del dato anterior: primero se decide qué información *"olvidar"* o "recordar" de la memoria del canal C, luego se *actualiza* la información de la memoria con los datos y, finalmente, se decide qué información se *utiliza en la salida*.

A partir de esto, se suelen nombrar las tres partes fundamentales de esta red como *Forget*

Gate, *Update Gate* y *Output Gate*. Luego, las ecuaciones que describen una *LSTM* son las mostradas a continuación, donde $\otimes$ corresponde a multiplicación punto a punto y las constantes W y b corresponden a los distintos pesos y *bias* de la red:

$$\text{Forget:} \quad f_t = \sigma(W_{xf} \cdot x_t + b_{xf} + W_{hf} \cdot h_{t-1} + b_{hf}) \tag{2.1}$$

$$\text{Update:} \quad \mathrm{i}_t = \sigma(W_{x\mathrm{i}} \cdot x_t + b_{x\mathrm{i}} + W_{h\mathrm{i}} \cdot h_{t-1} + b_{h\mathrm{i}}) \tag{2.2}$$

$$g_t = tanh(W_{xc} \cdot x_t + b_{xc} + W_{hc} \cdot h_{t-1} + b_{hc}) \tag{2.3}$$

$$\text{Output:} \quad o_t = \sigma(W_{xo} \cdot x_t + b_{xo} + W_{ho} \cdot h_{t-1} + b_{ho}) \tag{2.4}$$

$$C_t = f_t \otimes C_{t-1} + \mathrm{i}_t \otimes g_t \tag{2.5}$$

$$h_t = o_t \otimes tanh(C_t) \tag{2.6}$$

2.2. *Assembler* y *RISC-V*

2.2.1. *Assembler*

Assembler consiste en un lenguaje de programación de muy bajo nivel, cuyas instrucciones consisten directamente en las acciones que un procesador debiese realizar. Estas instrucciones dependen de la *ISA* del procesador sobre el cual se trabaja. Esto último sera explicado en breve.

Para el caso de programas realizados en lenguajes de mayor nivel, como *Python* o *Java*, estos deben ser traducidos por un intérprete a lenguaje de máquina, en donde el intérprete toma las decisiones de cual resultaría la traducción más óptima a lenguaje de máquina a partir del código presentado. A diferencia de esto, implementaciones realizadas en *Assembler* son traducidas prácticamente 1:1 al lenguaje de máquina correspondiente, lo cual permite tener mucho mayor control sobre el funcionamiento puro de un procesador, a cambio de necesitar explicitar exhaustivamente todas las acciones a realizar por parte del procesador en el código a ejecutar.

2.2.2. Programación en *Assembler*

Para la programación en lenguajes *Assembler* es útil tener en cuenta ciertos conceptos básicos.

Registros y Memoria

Un procesador, además de poseer memoria, posee lo que se llaman "registros". En *Assembler* uno no puede realizar operaciones entre datos en memoria, uno debe cargar la información necesaria en registros para poder realizar las operaciones/instrucciones deseadas, para luego guardar el resultado nuevamente en memoria.

El acto de cargar y descargar información también requiere de instrucciones a programar, lo cual puede extender el tiempo de ejecución total en caso de realizar un gran número de accesos en memoria. Es por esto que es recomendable realizar la mayor cantidad de operaciones entre cada uno de estos accesos, cargando y guardando información solo cuando esto sea realmente necesario y mantener valores "intermedios" de una operación solo en registros.

Secciones .data y .text

Los lenguajes *Assembler* poseen en general distintas directivas, dependiendo de la *ISA*, sin embargo la gran mayoría tienen en común las directivas `.section`, `.text` y `.data`. La primera, `.section`, indica que lo escrito debajo de esta directiva pertenece a una sección, la cual es definida por las otras dos directivas.

Una sección definida por `.section .text` indica que lo que se encuentre en esta sección será considerado código *Assembler*. En pocas palabras, uno realiza la programación del código bajo esta sección del archivo.

Por el otro lado se encuentra la sección `.section .data`, la cual permite definir *Variables y Constantes*. En esta sección es posible definir punteros y el dato o secuencia de datos a la cual apunta, permitiendo así predefinir constantes en memoria, o alojar un espacio de memoria conocido como variables, pues es posible cargar nuevos valores en memoria sin alterar el puntero.

Un ejemplo implementado en *RISC-V* de ambas secciones se presenta en el fragmento de código *Listing* 2.1.

Listing 2.1: Ejemplo de código en RISC-V.

```
1   .section .data
2   //se define primero el nombre de un puntero, y luego a que datos apunta
3   numero:
4       .word 55; //la directiva .word define un valor entero de 32 bits
5
6   listaDeNumeros:
7       .word 111, 222, 333; //se define la secuencia de valores
8
9   .section .text
10  main:
11      la a0, numero // 'numero' nos entrega la direccion del valor 55
12      lw t0, 0(a0) //<---- t0 = 55
13
14      la a1, listaDeNumeros // 'listaDeNumeros' nos entrega la direccion del
15                            // primer valor de la secuencia
16      lw t1, 0(a1) // t1 = 111
17      lw t2, 4(a1) // agregando un desfase podemos acceder al siguiente n mero -> t2 =
            222
18      lw t3, 8(a1) // t3 = 333
```

2.2.3. ISA

Aunque existe una convención, no todos los procesadores funcionan exactamente de la misma forma y, por lo tanto, tienden a tener distintos "lenguajes" que describan su funcionamiento. Dicho lenguaje de un procesador es denominado *ISA*.

Más precisamente, una *ISA* o *Instruction Set Architecture* corresponde al conjunto de instrucciones *Assembler* que posee y utiliza un procesador, el cual es utilizado para "traducir" al momento de compilar un lenguaje de mayor nivel (como *Python* o *Java*) a las acciones equivalentes apropiadas en lenguaje de máquina. Una *ISA* define así las distintas acciones que puede realizar el procesador correspondiente.

2.2.4. RISC-V

RISC-V es un procesador relativamente reciente con una *ISA open-source*, la cual se tiene 3 extensiones basicas: *RV32I*, *RV32E* y *RV64I*. Estas extensiones poseen las instrucciones correspondientes para el manejo de números enteros, donde *RV32I* y *RV32E* utilizan (y por tanto requieren de) respectivamente, 32 y 16 registros de 32 bits, incluyendo el registro de constante cero x0, donde *RV32E* es usado particularmente en procesadores más pequeños.

Por el otro lado, la extension *RV64I* posee las mismas instrucciones de manejo de números enteros, los cuales utilizan en este caso 32 registros de 64 bits, teniendo además la habilidad de trabajar valores como 32 bits. Existe, además, la extensión *RV128I*, la cual fue creada con tal de anticiparse a la futura existencia de procesadores con registros de 128 bits, de tal forma de no perder compatibilidad en el futuro [9].

Como se indicó, *RISC-V* se caracteriza por tener una *ISA* modular, donde esta se divide en distintas extensiones, las cuales pueden ser o no usadas dependiendo de las necesidades o características del procesador a utilizar. Las extensiones que dispone un procesador son definidas por la arquitectura física de este. Las extensiones estándares son las indicadas en la tabla de la Figura 2.4.

La *ISA* a utilizar es nombrada según su base, seguido de las letras correspondientes a las extensiones utilizadas. Por ejemplo, *RV32IMF* se referiría a una *ISA* la cual utiliza la base de 32 registros de 32 bits junto a las instrucciones de multiplicación, división y punto flotante.

Además de las extensiones previas, la *ISA* también permite crear extensiones extra, las cuales posean instrucciones personalizadas. Debido a la estructura modular de *RISC-V*, estas instrucciones personalizadas no arruinan la compatibilidad de software que no esté consciente o no utilice dichas instrucciones, siendo entonces solo consideradas por aquel software que si las utilicen [10].

2.3. Herramientas

2.3.1. Google Collaboratory

Google Colaboratory, o *Google Colab*, es una herramienta gratuita de Google para programación en *Python*. Esta funciona a través de cuadernos de *Jupyter*, con la ventaja de poder

Extensión	Descripción breve
I/E	Es la extensión base de RISC-V. Posee operaciones basicas de numeros enteros y trabaja sobre 32 y 16 registros respectivamente
M	Operaciones de multiplicación y división de enteros
C	Instrucciones "Compactas" utilizadas para operaciones de 16 bits y a menor costo de memoria
F	Instrucciones para el manejo de Punto Flotante de simple precisión
D	Instrucciones para el manejo de Punto Flotante de doble precisión
A	Instrucciones para operaciones en memoria atómica

Figura 2.4: Extensiones estándares de *RISC-V*.

utilizar recursos computacionales (*CPU* y *GPU*) provistos por Google, permitiendo tener un buen ambiente de programación, independiente del equipo utilizado. Además de esto, los cuadernos de *Google Colab* funcionan de forma similar a otros tipos de documentos de Google, como *Google Docs*, donde estos cuadernos pueden ser fácilmente compartidos entre personas y ser guardados en la nube de *Google Drive*. Colab contiene además una gran cantidad de librerías comúnmente usadas ya preinstaladas, como *Numpy*, *Pandas*, *SKLearn*, *matplotlib*, etc, lo cual permite comenzar a programar en cuadernos nuevos de forma más rápida.

Para este trabajo, todos los códigos de *Python* necesarios serán programados utilizando *Google Colab*, en donde se realizarán tanto las implementaciones de la red como la medición de tiempos en *CPU* y *GPU*.

2.3.2. *Freedom Studio*

Freedom Studio [11] es una herramienta de desarrollo basada en la *IDE Eclipse*. Creada por la empresa *SiFive*, *Freedom Studio* fue creada con el propósito de facilitar la programación sobre procesadores *RISC-V*, entregando a priori emuladores de procesadores de la misma empresa.

Esta herramienta incluye un compilador de *RISC-V*, así como un simulador y *debugger GNU* de *RISC-V*, donde *Freedom Studio* automatiza el proceso de compilación y *debugging*, facilitando el desarrollo de código. Además de esto, permite fácilmente acceder a los estados de los registros y memoria, lo que permite ver los cambios de estos a medida que se ejecutan las distintas instrucciones una a una, así como exportar los valores de memoria a un archivo de texto, lo cual fue utilizado para comprobar el funcionamiento de las implementaciones.

2.3.3. *Gem5*

Un problema de *Freedom Studio* es que en el simulador *GNU*, su simulador por defecto, los códigos son ejecutados a la mayor velocidad posible por el computador o dispositivo donde se está trabajando, no obteniéndose entonces un rendimiento real (a menos que se posea el procesador correspondiente en forma física). Para remediar este problema es que se utiliza el simulador *Gem5*.

Gem5 es un simulador de plataforma modular orientado al estudio de arquitectura computacional [12]. Este tiene la particularidad de ser un simulador a base de ciclos, capaz de simular de forma más fidedigna un procesador, pues toma en consideración las características y limitaciones físicas de este al momento de realizar simulaciones. Esto nos permite obtener mediciones de tiempo más cercanas a la realidad al ejecutar los distintos códigos desarrollados.

Además de poder especificar los distintos parámetros necesarios para simular, como tamaños de memoria, velocidad de reloj, etc, *Gem5* tiene a disposición distintos tipos de *CPU* a seleccionar [13] [14]. Entre estas, nos interesan los siguientes tres modelos:

- **TimingSimpleCPU**: Un submodelo de SimpleCPU, el cual trabaja sin pipeline y simula tiempos accesos en memoria. Esta *CPU* tiene la particularidad de ser considerablemente más rápida al realizar simulaciones.
- **MinorCPU**: Un tipo de *CPU* más cercano a la realidad. Esta *CPU* trabaja con un pipeline *in-order* (ejecuta instrucciones en el orden especificado en el programa) y posee la capacidad de realizar múltiples instrucciones en un ciclo de reloj.
- **DerivO3CPU**: Un tipo de *CPU* similar al anterior, con la diferencia de trabajar sobre un pipeline *out-of-order* (ejecuta las instrucciones a medida que los recursos necesarios estén disponibles).

Capítulo 3

Estado del arte

3.1. Algoritmo de actualización de gradientes: Adam

Adam [15] (Adaptive moment estimation) es un algoritmo de optimización de parámetros. Este algoritmo combina las ventajas de dos algoritmos anteriores *AdaGrad* y *RMSEProp*, en donde el algoritmo utiliza el promedio de los gradientes y el cuadrado de estos, asi como *learning rates* por parámetro, en donde estos últimos son modificados en cada epoca de entrenamiento de tal forma de ir "desacelerando" la modificación de los parámetros.

Dado un parámetro θ en particular y su gradiente $\frac{\mathrm{d}L}{\mathrm{d}\theta}$, el algoritmo Adam esta dado por las ecuaciones 3.1 a 3.5, donde lr corresponde al hiperparámetro *learning rate*. β_1 y β_2 son hiperparámetros que valen usualmente 0,9 y 0,999, respectivamente, y t corresponde a la época de entrenamiento.

$$P_\theta = P_\theta \cdot \beta_1 + (1 - \beta_1) \cdot \frac{\mathrm{d}L}{\mathrm{d}\theta} \tag{3.1}$$

$$S_\theta = S_\theta \cdot \beta_2 + (1 - \beta_2) \cdot (\frac{\mathrm{d}L}{\mathrm{d}\theta})^2 \tag{3.2}$$

$$\hat{P}_\theta = \frac{P_\theta}{1 - \beta_1^t} \tag{3.3}$$

$$\hat{S}_\theta = \frac{S_\theta}{1 - \beta_2^t} \tag{3.4}$$

$$\theta = \theta - lr \cdot \frac{\hat{P}_\theta}{\sqrt{\hat{S}_\theta} + \varepsilon} \tag{3.5}$$

3.2. *RISC-V* para *LSTM*

Existen pocos casos de uso de *RISC-V* orientados específicamente a redes *LSTM*. Sin embargo, en un caso [16] se aprovechan las distintas extensiones de la *ISA*, así como la capacidad

de crear instrucciones personalizadas, para generar una *DSA* optimizada para aplicaciones de *RNN*, utilizando particularmente *LSTM* como arquitectura de *RNN*. Se realizan, además de esto, comparaciones con otros tipos de redes neuronales, como redes convolucionales y redes *fully connected*.

Particularmente, se fijan en las operaciones de tangente hiperbólica y sigmoide, siendo estas las funciones de activación principales en redes neuronales, con lo que implementan instrucciones personalizadas de *RISC-V* capaces de realizar un cálculo de estas de forma más eficiente. Además de esto, desarrollan instrucciones personalizadas enfocadas en "fusionar" aquellas instrucciones más utilizadas, con tal de reducir aún más el número de instrucciones necesarias.

Cabe mencionar que, en este *paper* mencionado, se utiliza la habilidad de crear instrucciones personalizadas en *RISC-V*, lo cual no será realizado en este trabajo de memoria pues se realizará una implementación de *LSTM* utilizando solamente las extensiones ya disponibles de la *ISA* (Figura 2.4). Además de esto, en este *paper* solo se realizan comparaciones de eficiencia sobre un procesador *RISC-V* "limpio" versus un procesador con las instrucciones agregadas. Para este trabajo se busca realizar comparaciones entre *RISC-V* y otros procesadores (*CPU* y *GPU*).

3.3. Aproximación de funciones trigonométricas *tanh* y *sigmoide*

Las funciones trigonométricas de tangente hiperbólica y sigmoide son funciones esenciales para casi cualquier tipo de red neuronal. Aunque el enfoque de este trabajo no abarca la realización de instrucciones personalizadas para estas funciones, como fue realizado en el *paper* mencionado previamente, se fija más en la implementación de buenas aproximaciones de estas funciones las cuales posean un rendimiento relativamente eficiente.

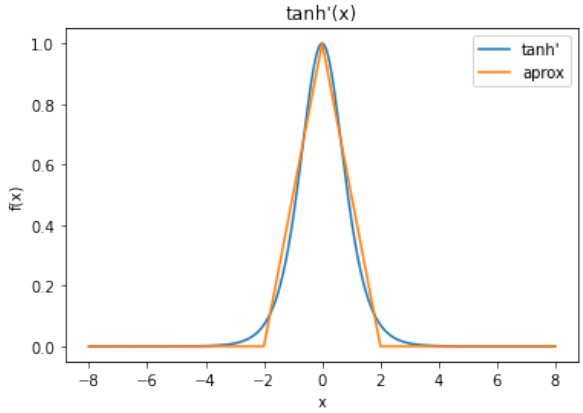

Figura 3.1: Aproximación de $tanh'$ a partir de la ecuación del triángulo isósceles.

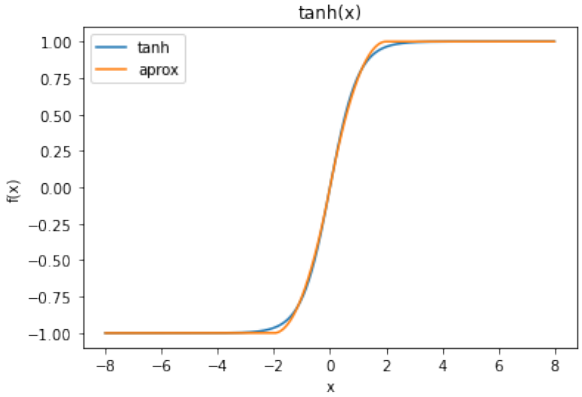

Figura 3.2: Aproximación de la función *tanh* obtenida.

Existen varias simplificaciones y aproximaciones de estas funciones, sin embargo, para este trabajo se utiliza el procedimiento descrito en el *paper* [17], el cual propone un método

para obtener una aproximación de segundo orden de la función tangente hiperbólica (*tanh*). Esto lo logra tomando la derivada de *tanh* y ajustando la ecuación del triángulo isósceles sobre esta (Figura 3.1), donde luego la integral resulta en una función de segundo orden la cual se ajusta a la función real con una alta (y un tanto inesperada) precisión (Figura 3.2).

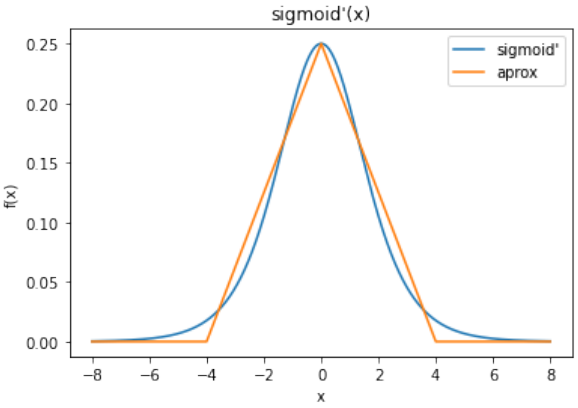

Figura 3.3: Aproximación de sigmoide' a partir de la ecuación del triángulo isósceles.

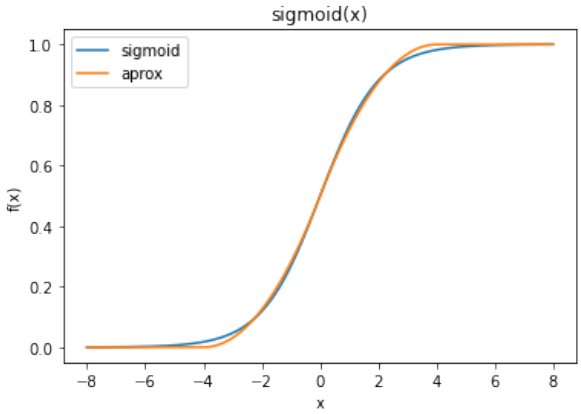

Figura 3.4: Aproximación de la función sigmoide obtenida.

Aunque no se menciona directamente en el *paper* anterior una simplificación de la función sigmoide, se menciona que el método utilizado es aplicable a este tipo de función, por lo que se realiza el mismo proceso manualmente (Figuras 3.3 y 3.4), obteniéndose así entonces las siguientes ecuaciones *tanh* y sigmoide utilizadas:

$$tanh(x) = \begin{cases} x - 0{,}25 \times sign(x) \times x^2 & , |x| < 2; \\ sign(x) & , \text{de lo contrario} \end{cases} \tag{3.6}$$

$$sigmoid(x) = \begin{cases} 0 & , x \leq -4 \\ 0{,}5 + 0{,}25 \times x - 0{,}03125 \times sign(x) \times x^2 & , |x| < 4; \\ 1 & , x \geq 4 \end{cases} \tag{3.7}$$

3.4. Predicción de valores de la bolsa

Como inspiración tanto para la red neuronal a diseñar como el *Dataset* a usar, se utiliza un caso de uso previo de redes *LSTM* [2]. En este *paper* se utiliza tanto una red *LSTM* y como una variación denominada *Bi-LSTM* (Figura 3.5), la cual consiste en dos redes *LSTM* en direcciones opuestas, para realizar predicciones sobre el valor y tendencia de la bolsa de valores. En este trabajo se utilizaron los valores históricos de las acciones de Google entre 2004 - 2019, donde de las 4170 muestras, 88 % se utilizó para el entrenamiento, mientras que el restante 22 % fue usado para pruebas. A partir de esto se obtienen predicciones con error *RMSE* 0.0032 y 0.0004 con la red *LSTM* y *Bi-LSTM* respectivamente.

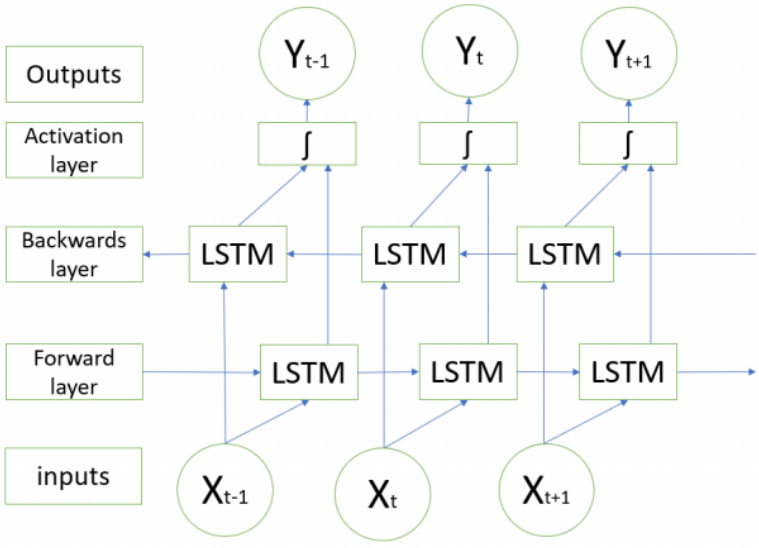

Figura 3.5: Red *Bi-LSTM*. Obtenido de [2].

Capítulo 4

Metodología y Desarrollo

En este capitulo se presenta el trabajo realizado en esta memoria. Este trabajo se divide, de forma cronológica, en los siguientes pasos:

1. Se implementa una red neuronal *LSTM* en *Python* y se prueba su funcionamiento para comprobar que el diseño de esta red sea aceptable.
2. Se pasa a trabajar en *RISC-V*, donde se decide qué placa de *RISC-V* utilizar para el trabajo.
3. Se realiza la implementación en *RISC-V* de las funciones trigonométricas necesarias para el funcionamiento de la red.
4. Con lo anterior listo, se realiza la implementación del funcionamiento (ejecución con parámetros pre-entrenados en *Python*) de la red *LSTM* en *RISC-V*.
5. Se realizan mediciones de funcionamiento en comparación a la implementación en *Python*, para comprobar que la implementación en *RISC-V* fue correcta.
6. Se obtienen las ecuaciones diferenciales necesarias para poder realizar el ajuste de pesos en el *backpropagation* del entrenamiento de la red.
7. A partir de lo anterior, se implementa el entrenamiento de la red en *RISC-V*, utilizando el código anterior como base.
8. Nuevamente se realizan mediciones de funcionamiento en comparación a la implementación en *Python*, para comprobar que la implementación en *RISC-V* fue correcta.
9. Se realizan mediciones de tiempo de la red de *Python* sobre una *CPU* y una *GPU*.
10. Se realizan simulaciones sobre las implementaciones de *RISC-V* para obtener mediciones de tiempo.

Las siguientes secciones de este capitulo corresponden, entonces, a la metodología y desarrollo de lo anterior, presentado en el orden en que fueron realizados, comenzando por el diseño de la red neuronal a utilizar.

4.1. La red neuronal

Para poder realizar comparaciones entre implementaciones de una red neuronal, primero se debe tener una red neuronal para comparar. Para esto se diseña una red neuronal en *Python* basándose en el *paper* de predicción de la bolsa de valores [2] donde se utilizan redes *LSTM* y *Bi-LSTM*. La red es diseñada en *Python* utilizando la herramienta Google Collaboratory, con el propósito principal de probar el funcionamiento de la red diseñada con tal de obtener una red *LSTM* suficientemente buena para realizar la implementación posterior y comparaciones de tiempo y funcionamiento. Se busca crear una red *LSTM* que sea funcional pero no necesariamente compleja, la cual sea capaz de predecir los valores de cierre de cada día a partir de la información de los 10 días anteriores.

4.2. Implementación en *Python*

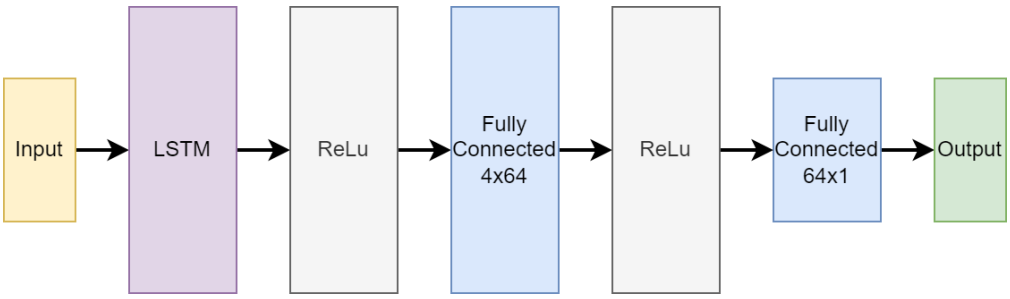

Figura 4.1: Diagrama de la red diseñada.

Como fue mencionado previamente, la implementación en *Python* fue realizada en *Google Collaboratory* habilitando el uso de una *GPU*, donde se realizaron las pruebas de funcionamiento, así como las mediciones de tiempo correspondientes (más detalles del Hardware utilizado, así como las mediciones, es descrito en la sección de mediciones de tiempo). Se hace uso de la librería de *machine learning PyTorch* para la implementación de la red neuronal, la cual permite, fácilmente, definir que procesador utilizar (*CPU* o *GPU*) al entrenar y ejecutar estas redes. El *notebook* de *Google Colab* utilizado se encuentra en [18].

La arquitectura de la red final diseñada consiste en una red *LSTM* con 4 *features* en sus estado interno C y sus salidas intermedias h. Para esta red, sin embargo, solo se utiliza la última salida $h_{t=10}$, a la cual le sigue una "capa" función *ReLu*, una capa densa *Fully Conected* (FC) de 64 neuronas, una segunda capa *ReLu* seguido de una capa de salida *FC* de una neurona (Figura 4.1).

Los datos a utilizar, similarmente a los utilizados en el *paper* mencionado anteriormente [2], corresponden al historial de la bolsa de valores de Google (GOOG) desde 19/08/2004 hasta 04/10/2019, donde se encuentra información de valores de apertura, máximo, mínimo, cierre, cierre ajustado y volumen de transacciones de cada día entre las fechas indicadas [19]. Los datos son separados en dos *datasets* nuevos, donde el primero (datos de entrada) consiste en la información de todos los días excepto el último, excluyendo además el volumen de transacciones, mientras que el segundo ("labels") posee solamente los valores de cierre de

cada día, empezando desde el décimo día. Se realiza un preprocesamiento de los datos con tal de estandarizar estos, aplicando "MinMax Scaling" a los datos del primer *dataset* y *Standard Scaling* al segundo. Se agrupan, posteriormente, los datos del conjunto de entrada, de tal forma de tener para cada valor de cierre a predecir, tener en los datos de entrada un grupo consistente de la información de los 10 días anteriores. Finalmente, los datos son separados en un conjunto de entrenamiento y prueba, tomando el 88 % de los datos para el entrenamiento y el resto para prueba.

Respecto al entrenamiento de la red, normalmente lo correcto sería entrenar utilizando un conjunto de validación, deteniendo el entrenamiento cuando el error de la red sobre dicho conjunto comience a subir. Esto se realiza para evitar el sobreajuste de la red con los datos de entrenamiento. Sin embargo, se opta por definir una cantidad de épocas fijas, con el objetivo de tener mediciones de tiempo de entrenamiento consistentes. La red es entonces entrenada por 220 épocas, con un radio de entrenamiento (*learning rate*) de 0.005 utilizando un criterio de error de *Mean-Squared Error* (*MSE*) y utilizando el algoritmo de optimización "Adam". Para todos los entrenamientos, los pesos son inicializados aleatoriamente, utilizando una "semilla" o *seed* fija, la cual garantiza que los pesos iniciales (y por tanto finales) siempre serán los mismos.

Una vez entrenada la red se vuelven a pasar todos los datos por la red y las predicciones pueden ser observadas en la Figura 4.2, donde los datos de entrenamiento corresponden a aquellos a la izquierda de la línea roja divisora, teniendo los datos de prueba a la derecha de esta.

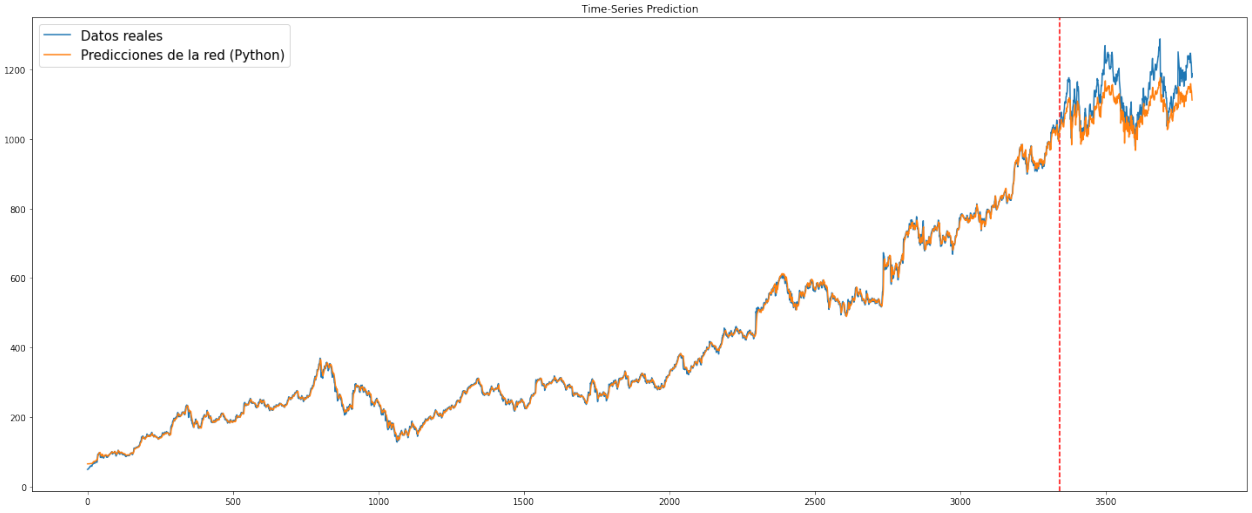

Figura 4.2: Predicciones de la red implementada en *Python* VS. datos reales.

Como se puede observar, las predicciones se ajustan a los valores reales, aunque no perfectamente. Sin embargo, dado que el objetivo de esta memoria no consiste en diseño de una red precisa, sino que en emular el funcionamiento de una red y realizar comparaciones de eficiencia respecto a tiempos, se considera que la implementación realizada es, aunque relativamente inexacta respecto a su precisión, suficientemente apropiada para este trabajo.

4.3. Herramientas para *RISC-V*

Para el trabajo sobre *RISC-V* se utilizaron principalmente las dos herramientas previamente mencionadas: *Freedom Studio* y *Gem5*. *Freedom Studio* es utilizado para todo lo que involucra desarrollo del código en *RISC-V Assembler*, debido a la facilidad que este software entrega para lo que es simulación, compilación y herramientas de *debug* de código, así como visualización de memoria y registros. Por el otro lado, *Gem5* es utilizado para simulación "real" de los códigos desarrollados, principalmente para obtener mediciones de tiempo de ejecución de estos.

4.3.1. "Hardware" seleccionado

Las mediciones son realizadas solamente a través de simulaciones. Para obtener resultados más auténticos se utiliza como referencia la placa de SiFive "HiFive Unleashed"[20] la cual, aunque fue descontinuada, es de gran interés debido al *SoC* (*System on a Chip*) que esta posee: el *Core* de SiFive **U-54**. Este *SoC* nos interesa debido a que se encuentra entre los pocos *Core*s incluidos para la simulación en *Freedom Studio*. Además de esto, este procesador posee todas las extensiones base de *RISC-V*, entre las cuales se encuentra la extensión que contiene instrucciones de punto flotante, las cuales son fundamentales para implementar una red neuronal de cualquier tipo. Más detalles de la placa utilizada son mencionados en la sección de mediciones de tiempo.

4.4. *RISC-V*: Trabajo realizado

Para este trabajo se realiza primero la implementación de las funciones trigonométricas tangente hiperbólica y sigmoide, definidas previamente en el estado del arte. Posterior a esto, se realizan dos implementaciones en *RISC-V* de la red neuronal definida previamente: una exclusivamente para la ejecución de la red y otra para el entrenamiento de esta.

Una de las principales consideraciones que se tienen en mente al momento de diseñar el código en *RISC-V* consiste en reducir el número de instrucciones necesarias lo más posible con tal de obtener un código más rápido. Es por esto que se busca, por ejemplo, reducir el número de accesos a memoria lo más posible, o aplicar *Loop Unrolling* donde sea conveniente.

4.4.1. Loop Unrolling

En lenguajes de mayor nivel, la forma correcta de implementar código, en donde este se repite con variaciones menores, consistiría en programar un *loop*/bucle que recorra dicho código, con el propósito de evitar duplicación innecesaria de este. Sin embargo, en especial para lenguajes de bajo nivel, realizar estas variaciones menores y realizar *loop*s de código puede llegar a ser costoso respecto a número de instrucciones necesarias.

Para las implementaciones del código se utiliza entonces una técnica conocida como *Loop Unrolling* o "Desenrollado de bucle(s)" [21]. Esta técnica consiste en escribir manualmente cada una de las repeticiones de código de un *loop*, a diferencia de escribirlo una vez y ejecutar este múltiples veces. Esto permite ahorrarse aquellas instrucciones encargadas de controlar el *loop*, haciendo que el código resultante sea más rápido, a cambio de tener un código más difícil de leer, en especial cuando se tienen bucles con un gran número de instrucciones o repeticiones.

Esta técnica es utilizable sobre bucles de repeticiones no variables, donde se sabe exactamente cuáles y cuantas instrucciones serian ejecutadas dentro del bucle. A continuación se muestra un ejemplo sencillo de esta técnica:

```
int main(void)
{
    //El siguiente c digo muestra en consola los n meros del 0 al 4
    for (int i=0; i<5; i++)
        printf("%d\n",i);

    //El siguiente c digo realiza la misma tarea, omitiendo
    //aquellas instrucciones usadas para funcionamiento del bucle for()
    printf("0\n");
    printf("1\n");
    printf("2\n");
    printf("3\n");
    printf("4\n");
}
```

4.5. Implementación de Funciones trigonométricas en *RISC-V*

Para facilitar la implementación y el uso de estas funciones, estas son implementadas como "metodos" usuales de programación, los cuales son llamados utilizando la instrucción `jal a0, tanh_fa6` o `jal a0, sig_fa6`, en donde se aplica la función sobre el valor guardado en el registro `fa6` y se guarda el resultado en `fa7`. La implementación de ambas funciones es similar, donde en ambos casos se realiza un primer *check* para ver si el número en `fa6` se encuentra en el rango correcto, en donde, en caso de estarlo, se realiza el cálculo de la función y, de lo contrario, se fija como resultado 1 o 0 según corresponda. Estas funciones se encuentran en las dos implementaciones de la red.

En el fragmento de código *Listing* 4.1 se presenta la implementación de *tanh*. Ambas implementaciones (*tanh* y sigmoide), junto con los código completos del trabajo, se encuentran en el anexo.

Listing 4.1: Implementación de la función tangente hiperbólica

```
tanh_fa6:
//apply tanh(x) where x = fa6
//if -2 < x < 2  -> tanh = x - 0.25*x  *sign(x)
//else tanh=sign(x)
//tanh result stored in fa7
//must be called with jal a0, tanh_fa6


```

```
    //---- load address of floats to be used
    la t1, tanh_floats

    //---- load 2 in ft10 and check if abs(x)<2
    flw ft10, 0(t1) //ft10 = 2
    fabs.s ft11, fa6 //ft11 = abs(x)
    flt.s t2, ft11, ft10 //t2 == 1 if |x|<2
    beqz t2, tanh_is_not_in_curve // t2 ==0 -> skip to return sign(x)

    //---- else: do x - 0.25*x  *sign(x)
    //x
    fmul.s fa7, fa6, fa6
    //ans*sign(x)
    fsgnj.s fa7, fa7, fa6 //sign inject fa7(ans) with fa6(x) sign
    //-(ans*0.25)+x
    flw ft10, 4(t1); // ft10 = 0.25
    fnmsub.s fa7, fa7, ft10, fa6

    //---- finish
    jr a0

//---- x < -2 or 2 < x
tanh_is_not_in_curve: //return sign
    fsgnj.s fa7, fs11, fa6

jr a0
//---- end of tanh
```

4.6. Implementación de la ejecución de la red neuronal en *RISC-V*

La primera implementación consiste, exclusivamente, en el funcionamiento de la "ejecución" de la red. Esto es, la obtención de predicciones de la red neuronal a partir de la información de entrada. Esta implementación utiliza los parámetros (pesos y *biases*) preentrenados de la red neuronal implementada en *Python* y debe ser capaz de, a partir de la misma información de entrada, realizar una predicción similar.

El trabajo realizado para esta primera implementación se puede resumir en las siguientes tareas:

1. Definición de estructura y carga de datos.
2. Implementación del funcionamiento de la red *LSTM*, utilizando las ecuaciones 2.1 a 2.6.
3. A partir del resultado de $t = 10$ de lo anterior, implementar en funcionamiento las funciones *ReLu* y *Fully connected* correspondientes.
4. Ejecución y extracción de predicciones de la red para mediciones de funcionamiento.

4.6.1. Datos

Primero, para poder cargar los datos de entrada, así como los pesos y *biases* de la red, estos son explicitados directamente en la sección *.data*, debido a que no se poseen métodos

para leer archivos externos (como, por ejemplo, un .csv con los datos). Además de esto, se predefinen en esta sección espacios de memoria a utilizar como "variables": los canales C y H de la red *LSTM* a través del tiempo, los valores finales de las redes *Fully Connected* (FC) y los valores de salida.

4.6.2. LSTM

Una vez hecho esto, se implementa el funcionamiento de la red *LSTM*. Para esto se implementó primero la multiplicación de matrices de las ecuaciones 2.1 a 2.4. Considerando que los resultados de estas ecuaciones son vectores de dimensión 4, y que las ecuaciones 2.5 y 2.6 consisten solamente en sumas, multiplicaciones punto-a-punto y la función punto-a-punto tanh, podemos ver que para un valor de los vectores C_t y h_t solo afecta en el cálculo de este los valores de i_t, f_t, g_t, o_t y C_{t-1} en la misma posición. Esto es:

$$C_t[\mathrm{i}] = f(f_t[\mathrm{i}], C_{t-1}[\mathrm{i}], \mathrm{i}_t[\mathrm{i}], g_t[\mathrm{i}]) \quad , \mathrm{i}\, \varepsilon [0,3] \tag{4.1}$$

$$h_t[\mathrm{i}] = f(o_t[\mathrm{i}], C_t[\mathrm{i}]) \quad , \mathrm{i}\, \varepsilon [0,3] \tag{4.2}$$

Esto implica que es posible calcular valores de C_t y h_t sin necesidad de calcular completamente i_t, f_t, g_t, o_t, solamente calculando los primeros números de cada vector, luego los segundos... etc. Hacer esto nos permite guardar los valores de i_t, f_t, g_t, o_t en registros y calcular directamente uno a uno los valores de C_t y h_t, ahorrando así varias instrucciones de acceso a memoria.

Es así como se desarrolla entonces el código encargado de realizar los cálculos del valor los vectores i_t, f_t, g_t, o_t para su "posición" i, los cuales son calculados utilizando el fragmento de código *Listing* 4.2. Este código corresponde solo al cálculo del primer valor de i_t, en donde se cargan los valores de los pesos y *biases*, y se multiplica respecto a los valores de entradas y h_{t-1}. Para el caso de $t = 0$, se tiene que $h_{t-1} = 0$ por lo que se omiten las instrucciones de multiplicación de h_{t-1} con el peso correspondiente.

Listing 4.2: Fragmento del código responsable del cálculo de vectores i, f, g y o para un instante t.

```
1     //matrix multiplication for i, f, g & o
2     //ht vector stored as = [fs0 ... fs3]
3     //x vector stored in x  = [fs4 ... fs8]
4     //Weights and biases must be the following (with proper offsets)
5     // s1, weights_floats_input
6     // s2, weights_floats_hidden
7     // s3, biases_floats_input
8     // s4, biases_floats_hidden
9     //if t=0 flag a7 is 0, else a7=1
10    //(skips Wh*h multiplications as h=0 when t=0)
11
12    //---- fa1 -> i
13    //Wxi * x
14    flw ft0, 0(s1)
15    fmul.s fa6, ft0, fs4
16
17    flw ft0, 4(s1)
18    fmadd.s fa6, ft0, fs5 ,fa6
19
20    flw ft0, 8(s1)
21    fmadd.s fa6, ft0, fs6 ,fa6
```

```
22
23      flw ft0, 12(s1)
24      fmadd.s fa6, ft0, fs7 ,fa6
25
26      flw ft0, 16(s1)
27      fmadd.s fa6, ft0, fs8 ,fa6
28
29      //ans + bx
30      flw ft0, 0(s3)
31      fadd.s fa6, ft0, fa6
32
33      //t=0? (a7=0?) if so, skip Wh*h (h is 0)
34      beqz a7, mat_mul_skip_h_i
35
36      //ans + (Wh * h)
37      flw ft0, 0(s2)
38      fmadd.s fa6, ft0, fs0, fa6
39
40      flw ft0, 4(s2)
41      fmadd.s fa6, ft0, fs1, fa6
42
43      flw ft0, 8(s2)
44      fmadd.s fa6, ft0, fs2, fa6
45
46      flw ft0, 12(s2)
47      fmadd.s fa6, ft0, fs3, fa6
48
49      mat_mul_skip_h_i:
50      //ans + bh
51      flw ft0, 0(s4)
52      fadd.s fa6, ft0, fa6
53
54      jal a0, sig_fa6 //apply sigmoid
55      fmv.s fa1, fa7
```

Este fragmento corresponde, además, a la primera instancia de *Loop Unrolling* realizada en el código en donde se implementa el mismo código para los vectores f_t, g_t, o_t, con la diferencia de que para $t = 0$ se omite el cálculo de f_t, pues este solo es utilizado en la ecuación 2.5, donde $C_{t-1} = 0$ para $t = 0$. EL realizar *Loop Unrolling* permite ahorrar las instrucciones que ajusten las direcciones $s1$, $s2$, $s3$ y $s4$ cada vez que se quiera calcular los valores de los vectores anteriores.

Habiendo calculado entonces los primeros valores de cada uno de los vectores i_t, f_t, g_t, o_t, se calcula los primeros valores de C_t y h_t (ecuaciones 4.1 y 4.2 para $\mathrm{i} = 0$), omitiendo la multiplicación de $C_{t-1} * f$ en caso de que $t = 0$. Esto se realiza en el fragmento de código *Listing* 4.3.

Listing 4.3: Fragmento del código responsable del cálculo Ct y ht para un instante t.

```
1       //fa1 -> i
2       //fa2 -> f
3       //fa3 -> g
4       //fa4 -> o
5       //calculate ct = f * ct-1 + i * g
6       // ct = fa2 * 0(s6) + fa1 * fa3
7
8       //i * g
9       fmul.s fa6, fa1, fa3
10
11      // if a7=0 -> ct-1 = 0 -> ct = fa1 * fa3
12      beqz a7, skip_ct1_mul
13
14      //else: ct-1 * f + ans
15      flw ft1, 0(s6)
16      fmadd.s fa6, ft1, fa2, fa6
17
```

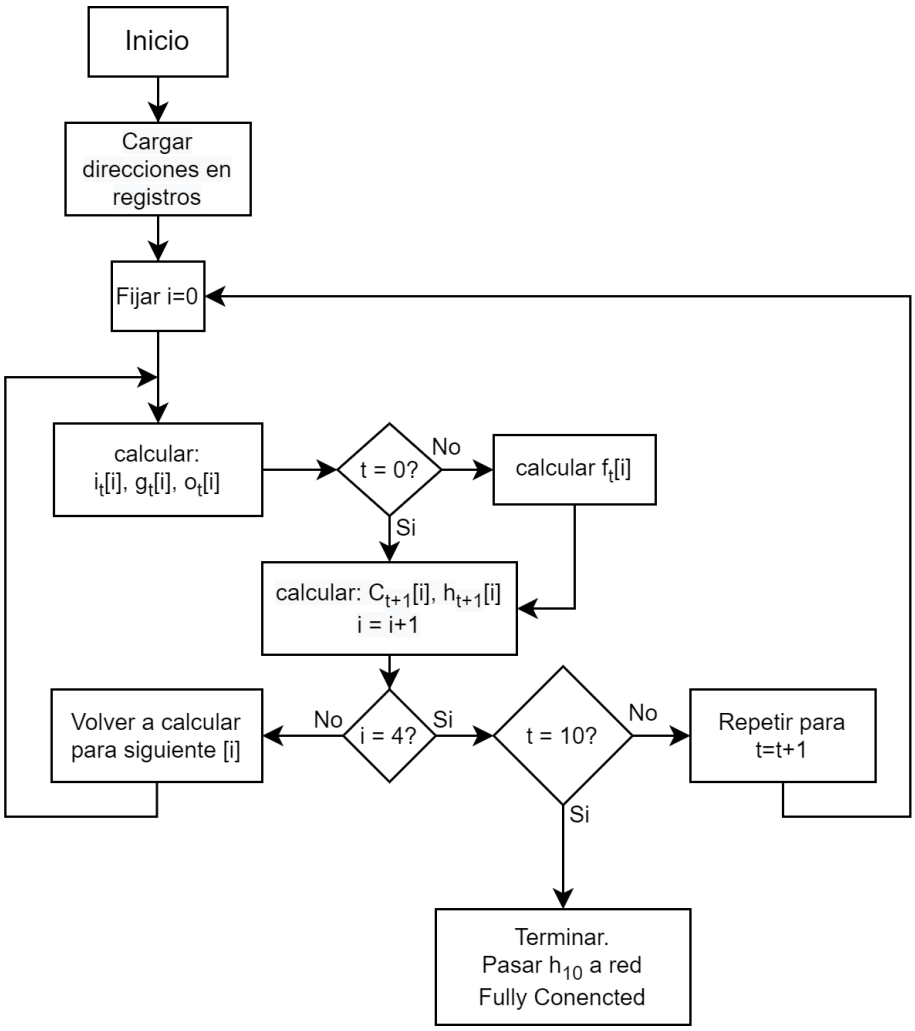

Figura 4.3: Diagrama de flujo simplificado del funcionamiento de la implementación de la red *LSTM* en *RISC-V*.

```
18      skip_ct1_mul:
19      //store ct
20      fsw fa6, 0(s6)
21
22      //ht+t = o * tanh(ct)
23      jal a0, tanh_fa6
24      //fa7 = tanh(ct) * o
25      fmul.s fa7, fa7, fa4
26      //store ht+1
27      fsw fa7, 0(s5)
```

Finalmente, se guardan los valores calculados de C_t[i] y h_t[i], se ajustan los *offsets* de los punteros para trabajar sobre i = i + 1 y se repite el cálculo de 2.1 a 2.6, hasta que se calculen los cuatro valores de C_t y h_t. Teniendo los vectores completos, se pasa a realizar otro reajuste de punteros, esta vez con el propósito de calcular el paso de tiempo siguiente $(t + 1)$, con lo que se ajusta con tal de utilizar el siguiente conjunto de entradas (de un total de 10), y se reinicia el proceso nuevamente con i = 0. Esto entonces se repite hasta haber calculado h_t para $t = 10$, donde se termina el *loop* y se alimentan las capas *FC* de la red. El funcionamiento de lo anterior queda resumido en el diagrama de la Figura 4.3.

4.6.3. Fully Connected

$$FC_{64} = W_{64} \cdot ReLu(h_{t=10}) + b_{64} \tag{4.3}$$

$$\hat{y} = W_{out} \cdot ReLu[FC_{64}] + b_{out} \tag{4.4}$$

Teniendo los 4 valores del vector $h_{t=10}$, se pasa al cálculo correspondiente a las dos capas *FC* de la red, así como las funciones *ReLu* correspondientes. Específicamente, el orden a implementar consiste en: *ReLu*, *FC* de 64 neuronas, *ReLu* y *FC* de 1 neurona (donde el resultado de esta última consistiría en el *output* de la red). Esto queda representado en las ecuaciones 4.3 y 4.4. La implementación realizada de lo anterior se puede resumir en el pseudocódigo *Listing* 4.4.

Listing 4.4: Pseudocódigo de la implementación de las capas Fully Connected de la red.

```
1
2     FC64 = vector de largo 64, que representa la capa de 64 neuronas
3     W64 = matriz de pesos de dimensi n 64x4, donde cada fila corresponde a los pesos de una
          neurona de la capa de 64 neuronas
4     B64 = vector que contiene los biases de las neuronas de la capa de 64 neuronas
5
6     FCout = valor de la capa de salida de una neurona
7     Wout = vector de pesos de la capa de salida, de largo 64
8     Bout = bias de la capa de salida
9
10    Ht10 = vector de largo 4
11
12    ReLu(x) -> Equivale a 0 si x es menor o igual a 0, de lo contrario vale x
13
14    //Comienza el codigo:
15    FCout = Bout
16    Para i de (0 a 63):
17        FC64[i] = B64[i]
18        Para k de (0 a 3):
19            FC64[i] = FC64[i] + W64[i][k] * ReLu(Ht10[k])
20        FCout = FCout + Wout[i] * ReLu(FC64[i])
```

Al momento de implementar el pseudocódigo *Listing* 4.4, la forma en que se implementa la función *ReLu* en el código consiste en revisar si el valor es menor o igual a cero (lo cual es posible con una sola instrucción) y, en caso de serlo, se omite completamente la multiplicación correspondiente, lo cual es funcionalmente idéntico a fijar el valor a cero y multiplicar pero, además, permite ahorrar las instrucciones multiplicación y carga desde memoria del valor del peso correspondiente.

Entonces, primero se realiza un *check* respecto a los valores de $h_{t=10}$, correspondiente a la primera *ReLu*, donde se fija una "bandera" (*flag*) para cada uno de estos valores, que indica si son o no mayor a cero, el cual se utiliza para omitir las multiplicaciones correspondientes. Además, para omitir cargas y guardados en memoria innecesarios, se calcula el valor de la capa de salida, inmediatamente, a medida que se van calculando los valores de las neuronas de la capa de 64. Esto es, se calcula el valor resultante de una neurona de la capa *FC* de 64 y se pondera inmediatamente con el peso correspondiente a la capa de salida, realizando antes de esta última ponderación el mismo *check* hecho para la implementación de la función *ReLu* sobre $h_{t=10}$. Con esto, se obtiene el fragmento de código *Listing* 4.5, donde al final se muestran las instrucciones correspondientes al cálculo para una de las 64 neuronas, el cual

es repetido manualmente (*Loop Unrolling*) 64 veces con el propósito de ahorrar instrucciones.

Listing 4.5: Fragmento de código responsable del cálculo de las función *ReLu* y la capas *FC*.

```
    //----- apply ReLu flag -----
    //relu flag is used to skip the multiplication,
    //instead of setting the number to zero (same result)
    //flag is 1 if number is =< 0 -> should skip multiplication
    fmv.s.x ft3, x0 //set zero

    //flags:
    //fs0 -> t0
    //fs2 -> t1
    //fs2 -> t2
    //fs3 -> t3
    fle.s t0, fs0, ft3
    fle.s t1, fs1, ft3
    fle.s t2, fs2, ft3
    fle.s t3, fs3, ft3

    //addresses for fc_1 weights, biases and results
    la a1, fc_1_weight
    la a2, fc_1_bias
    la a3, fc_1_values

    //adresses for fc weights and bias
    la a4, fc_weight_and_bias

    //fa0 -> final output
    //start with loading fc bias
    flw fa0, 256(a4)

    //Following code loop is unraveled to save 4*64 instructions
    //it feeds ht into each fc_1 neuron, which then feeds into fc

    //--------------------------------
    //------    "LOOP" START   -------
    //--------------------------------

    //0
    flw ft0, 0(a2)

    bnez t0, skip_0_0
        flw ft1, 0(a1)
        fmadd.s ft0, ft1, fs0, ft0
    skip_0_0:

    bnez t1, skip_1_0
        flw ft1, 4(a1)
        fmadd.s ft0, ft1, fs1, ft0
    skip_1_0:

    bnez t2, skip_2_0
        flw ft1, 8(a1)
        fmadd.s ft0, ft1, fs2, ft0
    skip_2_0:

    bnez t3, skip_3_0
        flw ft1, 12(a1)
        fmadd.s ft0, ft1, fs3, ft0
    skip_3_0:

    //relu, then feed to fc
    fle.s t4, ft0, ft3//relu flag
    bnez t4, skip_fc_0//if ft0<=0 then skip fmadd.s
        flw ft1, 0(a4)
        fmadd.s fa0, ft0, ft1, fa0
    skip_fc_0:
```

4.6.4. *Output*

Para terminar, el resultado de la capa de salida es guardado en memoria, se reinician los desfases de punteros necesarios para realizar el mismo proceso con el siguiente conjunto de 10 días y se reinicia el código (fragmento de código *Listing* 4.6). Esto ocurre hasta que la red sea alimentada con los 3797 conjuntos de entradas y genere el mismo número de salidas, todas almacenadas en memoria en direcciones conocidas. Estas son posteriormente exportadas a un archivo de texto, con el propósito de comprobar el funcionamiento de la implementación, así como realizar comparaciones con la implementación en *Python*.

Listing 4.6: Fragmento de código responsable de guardar en memoria el resultado final y reiniciar con los desfases necesarios.

```
    //--------------------------------
    //------     "LOOP" END     ------
    //--------------------------------

    fsw fa0, 0(a5)


    //---- FORWARD PASS END ----
    //restart, with input and output offset
    //input(s0): go back 8 rows = -160
    addi s0, s0, -160

    //output(a5) offset -> 1 float  = +4
    addi a5, a5, 4

    //"add" -1 to the FeedForward counter (s11)
    //loop feed forward 3797 times
    //repeat until counter reaches 0
    addi s11, s11, -1

    //if s11 != 0 -> repeat
    bnez s11, forward_pass_full_restart


call exit
```

4.7. Implementación del entrenamiento de la red neuronal en *RISC-V*

Una vez que se termina la implementación del código encargado de la ejecución de la red y se comprueba su apropiado funcionamiento (lo cual es realizado en la sección de resultados), se pasa a realizar la implementación del entrenamiento de la red. Este utiliza como base la implementación previa, en donde se le agrega sobre esta el funcionamiento necesario para poder realizar el cálculo de gradientes de cada parámetro a través de *backpropagation*, así como el posterior ajuste de estos parámetros.

El trabajo realizado para esta segunda implementación puede ser resumido por las siguientes tareas:

1. Obtención de las ecuaciones diferenciales para *backpropagation*.

2. Modificación del código base para guardar en memoria datos extra necesarios.
3. Cálculo de gradientes (*backpropagation*) para las capas F.C.
4. Cálculo de gradientes para la red *LSTM*
5. Actualización de parámetros de la red utilizando el algoritmo Adam.
6. Entrenamiento de la red y carga de parámetros entrenados sobre la primera implementación para probar funcionamiento.

4.7.1. Ecuaciones Diferenciales

Antes de poder empezar a trabajar en el código correspondiente a la implementación del entrenamiento, es necesario obtener las ecuaciones de gradientes para poder realizar *backpropagation*. Normalmente, al trabajar con programas o librerías de lenguajes de alto nivel como en *Python*, uno no necesita explicitar estas ecuaciones, pues gran parte de estas son creadas al definir la estructura de la red, necesitando solamente explicitar la función de error a utilizar. En nuestro caso no solo debemos programar estas ecuaciones, sino que además obtenerlas a partir de la estructura de la red utilizada.

$$(MSE)\ L(\hat{y}, y) = \frac{1}{N}\sum_{\mathrm{i}=1}^{N}(y - \hat{y})^2 \tag{4.5}$$

Dada la ecuación de error 4.5, el gradiente de cada parámetro θ estaría dado por la derivada parcial de la ecuación 4.6. Las ecuaciones de gradiente de cada parámetro se obtienen utilizando la regla de la cadena para derivadas parciales, utilizando tanto la ecuación de error 4.5 como las ecuaciones que definen la estructura de nuestra red neuronal, siendo estas las ecuaciones de 2.1 a 2.6, 4.3 y 4.4.

$$\mathrm{grad}(\theta) = \frac{\mathrm{d}L}{\mathrm{d}\theta} \tag{4.6}$$

Respecto a las derivadas parciales que involucran funciones trigonométricas, se utilizan las derivadas de las funciones no aproximadas, utilizando entonces como derivadas las ecuaciones 4.7 y 4.8.

$$\frac{\mathrm{d}}{\mathrm{d}x}tanh(x) = 1 - tanh^2(x) \tag{4.7}$$

$$\frac{\mathrm{d}}{\mathrm{d}x}\sigma(x) = \sigma(x)(1 - \sigma(x)) \tag{4.8}$$

Además de esto, es necesario separar las ecuaciones 2.1, 2.2 2.3 y 2.4 en dos partes, con tal de poder obtener las derivadas parciales necesarias para calcular los gradientes de los parámetros de estas ecuaciones. De esta forma, para el cálculo de los gradientes, se utilizan las ecuaciones 4.9 a 4.16

$$f_t = \sigma(a_{f_t}) \tag{4.9}$$

$$\mathrm{i}_t = \sigma(a_{\mathrm{i}_t}) \tag{4.10}$$

$$g_t = tanh(a_{g_t}) \tag{4.11}$$

$$o_t = \sigma(a_{o_t}) \tag{4.12}$$

$$a_{f_t} = W_{xf} \cdot x_t + b_{xf} + W_{hf} \cdot h_{t-1} + b_{hf} \tag{4.13}$$

$$a_{\mathrm{i}_t} = W_{x\mathrm{i}} \cdot x_t + b_{x\mathrm{i}} + W_{h\mathrm{i}} \cdot h_{t-1} + b_{h\mathrm{i}} \tag{4.14}$$

$$a_{g_t} = W_{xc} \cdot x_t + b_{xc} + W_{hc} \cdot h_{t-1} + b_{hc} \tag{4.15}$$

$$a_{o_t} = W_{xo} \cdot x_t + b_{xo} + W_{ho} \cdot h_{t-1} + b_{ho} \tag{4.16}$$

Finalmente, definimos la notación de la función escalón, la cual resulta ser la derivada de la función *ReLu*, como la presentada en la ecuación 4.17

$$\frac{\mathrm{d}L}{\mathrm{d}x}\mathrm{ReLu(x)} = u(x) = \begin{cases} 1 & , x > 0 \\ 0 & , x <= 0 \end{cases} \tag{4.17}$$

A partir de esto, recordando que como salida de la red *LSTM* solo se utiliza la última salida (eso es, $h_{t=10}$), se obtienen manualmente las derivadas parciales necesarias para calcular los gradientes de la red, las cuales consisten en las ecuaciones 4.18 a 4.30, donde las ecuaciones de 4.18 a 4.23 corresponden a las ecuaciones de las capas *Fully Connected*, mientras que las ecuaciones de 4.24 a 4.31 corresponden a las de la red *LSTM*.

$$\frac{\mathrm{d}L}{\mathrm{d}FC_{out}} = 2(\hat{y} - y) \tag{4.18}$$

$$\frac{\mathrm{d}L}{\mathrm{d}b_{out}} = \frac{\mathrm{d}L}{\mathrm{d}FC_{out}} \tag{4.19}$$

$$\frac{\mathrm{d}L}{\mathrm{d}W_{out}} = \frac{\mathrm{d}L}{\mathrm{d}FC_{out}} \cdot (\mathrm{ReLu}(FC_{64}))^T \tag{4.20}$$

$$\frac{\mathrm{d}L}{\mathrm{d}FC_{64}} = (W_{out}^T \cdot \frac{\mathrm{d}L}{\mathrm{d}FC_{out}}) \otimes u(FC_{64}) \tag{4.21}$$

$$\frac{\mathrm{d}L}{\mathrm{d}b_{64}} = \frac{\mathrm{d}L}{\mathrm{d}FC_{64}} \tag{4.22}$$

$$\frac{\mathrm{d}L}{\mathrm{d}W_{64}} = \frac{\mathrm{d}L}{\mathrm{d}FC_{64}} \cdot (\mathrm{ReLu}(h_{t=10}))^T \tag{4.23}$$

$$\frac{\mathrm{d}L}{\mathrm{d}h_{t=10}} = (W_{64}^T \cdot \frac{\mathrm{d}L}{\mathrm{d}FC_{64}}) \otimes u(h_{t=10}) \tag{4.24}$$

$$\begin{aligned} \frac{\mathrm{d}L}{\mathrm{d}C_t} &= \frac{\mathrm{d}L}{\mathrm{d}h_t} \otimes o_t(1 - tanh^2(C_t)) \\ &= \frac{\mathrm{d}L}{\mathrm{d}h_t} \otimes o_t - h_t \otimes tanh^2(C_t) \end{aligned} \tag{4.25}$$

$$\begin{aligned} \frac{\mathrm{d}L}{\mathrm{d}a_{f_t}} &= \frac{\mathrm{d}L}{\mathrm{d}C_t} \otimes f_t \otimes (1 - f_t) \\ &= \frac{\mathrm{d}L}{\mathrm{d}C_t} \otimes f_t - f_t^2 \end{aligned} \tag{4.26}$$

$$\begin{aligned} \frac{\mathrm{d}L}{\mathrm{d}a_{\mathrm{i}_t}} &= \frac{\mathrm{d}L}{\mathrm{d}C_t} \otimes g_t \otimes \mathrm{i}_t \otimes (1 - \mathrm{i}_t) \\ &= \frac{\mathrm{d}L}{\mathrm{d}C_t} \otimes g_t \otimes (\mathrm{i}_t - \mathrm{i}_t^2) \end{aligned} \tag{4.27}$$

$$\frac{\mathrm{d}L}{\mathrm{d}a_{g_t}} = \frac{\mathrm{d}L}{\mathrm{d}C_t} \otimes \mathrm{i}_t \otimes (1 - g_t^2) \tag{4.28}$$

$$\frac{\mathrm{d}L}{\mathrm{d}a_{o_t}} = \frac{\mathrm{d}L}{\mathrm{d}h_t} \otimes tanh(C_t) \otimes o_t \otimes (1 - o_t) \tag{4.29}$$

$$\begin{aligned} &= \frac{\mathrm{d}L}{\mathrm{d}h_t} \otimes h_t \otimes (1 - o_t) \end{aligned} \tag{4.30}$$

$$\frac{\mathrm{d}L}{\mathrm{d}h_{t-1}} = W_{f_t}^T \cdot \frac{\mathrm{d}L}{\mathrm{d}a_{f_t}} + W_{\mathrm{i}_t}^T \cdot \frac{\mathrm{d}L}{\mathrm{d}a_{\mathrm{i}_t}} + W_{g_t}^T \cdot \frac{\mathrm{d}L}{\mathrm{d}a_{g_t}} + W_{o_t}^T \cdot \frac{\mathrm{d}L}{\mathrm{d}a_{o_t}} \tag{4.31}$$

Finalmente, para cada uno de los parámetros de la red *LSTM* (pesos y *biases* de las ecuaciones de f_t, i_t, g_t y o_t) se tiene que las ecuaciones de gradiente son las mismas, mientras se utilice la derivada parcial correspondiente (por ejemplo para W_{xf} y b_f, utilizar el gradiente $\mathrm{d}L/\mathrm{d}a_{f_t}$, Para $W_{x\mathrm{i}}$ y b_{i} usar $\mathrm{d}L/\mathrm{d}a_{\mathrm{i}_t}$, etc.). Luego, para evitar redundancia, sea $\theta = \{f, \mathrm{i}, g, o\}$, se tiene que las ecuaciones de gradiente de los parámetros de la red *LSTM* son las ecuaciones 4.32 a 4.34.

$$\frac{\mathrm{d}L}{\mathrm{d}b_{x\theta}} = \frac{\mathrm{d}L}{\mathrm{d}b_{h\theta}} = \frac{1}{10}\sum_{t=1}^{10}\frac{\mathrm{d}L}{\mathrm{d}a_{\theta_t}} \tag{4.32}$$

$$\frac{\mathrm{d}L}{\mathrm{d}W_{x\theta}} = \frac{1}{10}\sum_{t=1}^{10}\frac{\mathrm{d}L}{\mathrm{d}a_{\theta_t}} \cdot x_t^T \tag{4.33}$$

$$\frac{\mathrm{d}L}{\mathrm{d}W_{h\theta}} = \frac{1}{10}\sum_{t=1}^{10}\frac{\mathrm{d}L}{\mathrm{d}a_{\theta_t}} \cdot h_{t-1}^T \tag{4.34}$$

Todos los gradientes anteriores son calculados para cada conjunto de *inputs* a la red, con lo que antes de ajustar los parámetros, se deben calcular todos los gradientes para todo el *dataset* completo de 3342 datos de entrenamiento, donde luego se promedian los gradientes calculados (cada parámetro promedia sus propios 3342 gradientes). La forma de implementar este promedio consiste en, simplemente, ir sumando los gradientes de cada parámetro para, finalmente, dividir los gradientes acumulados de cada uno de estos por 3342 al realizar la actualización de gradientes. Para el caso de gradientes de la red *LSTM*, se divide además por 10 debido a que en cada predicción hay 10 gradientes para cada parámetro de la red, debido a los 10 instantes de tiempo.

4.7.2. Modificaciones al código base

Como se menciona previamente, la implementación del entrenamiento de la red utiliza como código base la implementación anterior, pues para calcular los gradientes a través de *backpropagation* es necesario saber que valores se obtienen en cada parte de la red. Dichos valores, al no ser necesarios previamente, no son guardados en memoria, con lo que es necesario modificar el código de la ejecución de la red de tal forma que se guarde información extra.

Se modifica entonces el código para que, para cada una de las predicciones realizadas, sean guardadas en memoria los siguientes valores:

- vectores h_t para t de 1 a 9 (previamente solo se guardaba $t = 10$).
- vectores C_t para t de 1 a 10.
- vectores resultantes de $\tanh(C_t)$ para t de 1 a 10, con tal de evitar realizar el mismo cálculo múltiples veces.
- vectores f_t, i_t, g_t y o_t para t de 1 a 10.
- resultados de la capa Fully Connected de 64 neuronas (vector FC_{64})
- resultados de $ReLu(h_{t=10})$ y $ReLu(FC_{64})$
- resultados de $u(h_{t=10})$ y $u(FC_{64})$

Para el caso de los últimos (resultados de *ReLu*), cabe recordar la forma en que se "implementa" la función *ReLu*. Esta no es implementada directamente, sino que en su lugar se realiza un *check* de si el número correspondiente es mayor a cero o no (equivalente a función escalón $u(\mathrm{x})$), donde el resultado de esto es utilizado para omitir los cálculos de aquellos valores negativos (pues esto requiere de menos instrucciones que realizar *ReLu* y multiplicar posteriormente por cero cada vez). Luego, no necesitamos realmente guardar los valores de $ReLu(h_{t=10})$ y $ReLu(FC_{64})$, sino que basta con guardar en memoria $h_{t=10}$, FC_{64}, $u(h_{t=10})$ y $u(FC_{64})$, los cuales deben ser guardados de todas formas pues son necesarios por si solos.

Finalmente, el último cambio al código base consiste en los datos guardados para cada parámetro, en donde para cada "conjunto de datos" (o para cada puntero) se agregan espacios para guardar los gradientes de cada parámetro, así como los valores de P y S (algoritmo Adam) de cada uno.

4.7.3. Cálculo de gradientes: *Fully Connected*

Como indica el nombre, el cálculo de gradientes a través de *backpropagation* ocurre "hacia atrás" por la red, por lo que primero es necesario implementar el cálculo de gradiente de las últimas partes de la red.

Se implementa primero a través del código *Listing* 4.7 las ecuaciones 4.18 y 4.19, las cuales corresponden al mismo valor, y se inicializan los registros correspondientes a $\mathrm{d}L/\mathrm{d}h_{10}$ para empezar a sumar gradientes sobre este.

Listing 4.7: Fragmento de código inicial de backpropagation.

```
1   //expected result -> ft0
2   //predicted value -> fa0
3   flw ft0, 0(s10)
4
5   //offset s10 for next one
6   addi s10, s10, 4
7   //dL/dFCout = 2*(predicted - expected)
8   //saved in fs10
9   fsub.s fs10, fa0, ft0
10  fadd.s fs10, fs10, fs10
11
12  //dL/dbout = dL/dFCout
13  flw ft0, 1028(a4)
14  fadd.s ft0, ft0, fs10
15  fsw ft0, 1028(a4)
16
17  //dL/dh10 = [fs4, ..., fs7]
18  //set to x0 to start
19  fmv.s.x fs4, x0
20  fmv.s.x fs5, x0
21  fmv.s.x fs6, x0
22  fmv.s.x fs7, x0
```

Después de esto, se pasa a implementar el cálculo de las ecuaciones 4.23 a 4.24. Para esto, se realiza un procedimiento similar al pseudocódigo *Listing* 4.4. Al momento de calcular los gradientes para ir sumándolos al "acumulado", se realiza un *check* previo que indica si los cálculos a realizar entregarían cero o no (debido a que se sabe de antemano las multiplicaciones involucradas). En caso de que se sepa que ciertos gradientes den cero, estos son omitidos completamente (lo cual ahorra instrucciones que simplemente sumarían +cero al acumulado).

Además de esto, con el propósito de utilizar la menor cantidad de accesos en memoria, la implementación del cálculo de 4.23 a 4.24 queda descrito por el pseudocódigo *Listing* 4.8.

Listing 4.8: Pseudocódigo de la implementación del cálculo de gradientes para las capas *FC* y ht=10.

```
1
2     for every i flag in vector fc_1_reluflags (64 in total), check flag
3     if flag is 1, skip everything & move to the next [i]
4     if flag is 0, use fc_1_values in position [i]
5       dL/dWout [i] += fc_1_values [i] * fs10
6       dL/dfc1 [i] = fc_weight_and_bias[i] * fs10
7       dL/db1[i] += dL/dfc1 [i]
8       for every [k] in [t0, ..., t3] (h10 relu flags), check flag. if 1-> skip k
9           dL/dW1 [i][k] += dL/dfc1 [i] * h10[k]
10          dL/dh10 [k] += dL/dfc1 [i] * fc_1_weight[i][k] -> save to [fs4, ..., fs7]
11
12    Para vectores, cada [i] es un desfase extra de 4
13    Para matrices, cada columna [k] es un desfase de +4 y cada fila [i] es (+rowLength*4)
          (+(4*4)=+16 en este caso)
```

Para la implementación se vuelve a aplicar *Loop Unrolling*, con lo que el código *Listing* 4.9 obtenido es repetido 64 veces para obtener todos los gradientes necesarios.

Listing 4.9: Fragmento de código responsable del cálculo de gradientes para las capas *FC* y ht=10.

```
1     //i=0
2     //fc1 relu flag
3     lw t4, 0(s9)
4     //if 1-> skip all
5     bnez t4, skip_backprop_i0
6         //else use fc_1_values in position [i] -> ft0 = fc_1[i]
7         flw ft0, 0(a3)
8
9         //dL/dWout [i] += fc_1_values [i] * fs10
10        //gradient offset is 260 from fc_weight_and_bias
11        flw ft1, 260(a4)
12        fmadd.s ft1, ft0, fs10, ft1
13        fsw ft1, 260(a4)
14
15        //  get dL/dfc1 [i] = fc_weight_and_bias[i] * fs10 -> ft1 = dL/dfc1[i]
16        flw ft1, 0(a4)
17        fmul.s ft1, ft1, fs10
18        //  dL/db1[i] += dL/dfc1 [i]
19        flw ft2, 256(a2)
20        fadd.s ft2, ft2, ft1
21        fsw ft2, 256(a2)
22
23        //h10 = [fs0, ..., fs3]
24        //h10 relu flags = [t0, ..., t3]
25        //  for every [k] in [t0, ..., t3] (h10 relu flags), check flag. if 1-> skip k
26
27        //k=0
28        bnez t0, skip_backprop_k00
29            //dL/dW1 [i][k] += dL/dfc1 [i] * h10[k]
30            //W1 = fc_1_weights; gradients have a base offset of 1024
31            flw ft2, 1024(a1)//=dL/dW1[i][k]
32            fmadd.s ft2, ft1, fs0, ft2
33            fsw ft2, 1024(a1)
34
35            //dL/dh10 [k] += dL/dfc1[i] * fc_1_weight[i][k] -> dL/dh10[0] = fs4
36            flw ft2, 0(a1)//=fc_1_weight[i][k]
37            fmadd.s fs4, ft1, ft2, fs4
38
39        skip_backprop_k00:
40
41        //k=1
42        bnez t1, skip_backprop_k01
```

```
            //dL/dW1 [i][k] += dL/dfc1 [i] * h10[k]
            flw ft2, 1028(a1)//=dL/dW1[i][k]
            fmadd.s ft2, ft1, fs1, ft2
            fsw ft2, 1028(a1)

            //dL/dh10 [k] += dL/dfc1[i] * fc_1_weight[i][k] -> dL/dh10[1] = fs5
            flw ft2, 4(a1)//=fc_1_weight[i][k]
            fmadd.s fs5, ft1, ft2, fs5

        skip_backprop_k01:

        //k=2
        bnez t2, skip_backprop_k02
            flw ft2, 1032(a1)//=dL/dW1[i][k]
            fmadd.s ft2, ft1, fs2, ft2
            fsw ft2, 1032(a1)

            //dL/dh10 [k] += dL/dfc1[i] * fc_1_weight[i][k] -> dL/dh10[1] = fs5
            flw ft2, 8(a1)//=fc_1_weight[i][k]
            fmadd.s fs6, ft1, ft2, fs6
        skip_backprop_k02:

        //k=3
        bnez t3, skip_backprop_k03
            flw ft2, 1036(a1)//=dL/dW1[i][k]
            fmadd.s ft2, ft1, fs3, ft2
            fsw ft2, 1036(a1)

            //dL/dh10 [k] += dL/dfc1[i] * fc_1_weight[i][k] -> dL/dh10[1] = fs5
            flw ft2, 12(a1)//=fc_1_weight[i][k]
            fmadd.s fs7, ft1, ft2, fs7
        skip_backprop_k03:
    skip_backprop_i0:
```

4.7.4. Cálculo de gradientes: *LSTM*

Una vez implementado lo anterior, particularmente el cálculo de la ecuación 4.24, se pasa a implementar los cálculos de gradiente para la red *LSTM*. Esto es, implementar el cálculo de las ecuaciones 4.25 a 4.31 y posteriormente los gradientes 4.32 a 4.34. Para esta parte la implementación realizada es considerablemente más sencilla, debido a tener suficientes registros como para calcular y guardar los valores de las ecuaciones 4.26 a 4.30, así como calcular 4.25, 4.31 y de 4.32 a 4.34, todo esto sin realizar accesos en memoria.

Luego, empezando desde $t = 10$, se calcula inicialmente $\frac{\mathrm{d}L}{\mathrm{d}C_t}$ (ecuación 4.25) y $\frac{\mathrm{d}L}{\mathrm{d}a_{o_t}}$ (4.30), los cuales dependen de h_t, o_t, $\tanh(C_t)$ y $\frac{\mathrm{d}L}{\mathrm{d}h_t}$. Posteriormente, se calcula $\frac{\mathrm{d}L}{\mathrm{d}a_{i_t}}$ (4.27) y $\frac{\mathrm{d}L}{\mathrm{d}a_{g_t}}$ (4.28), los cuales dependen del resultado de 4.25, así como de g_t y i_t. Finalmente, se calcula $\frac{\mathrm{d}L}{\mathrm{d}a_{f_t}}$ (4.26) utilizando los valores de $\frac{\mathrm{d}L}{\mathrm{d}C_t}$, C_{t-1} y f_t. Todo esto es realizado, en el orden mencionado, a través del fragmento de código *Listing* 4.10.

Listing 4.10: Fragmento de código encargado de la primera parte del cálculo de gradientes.

```

    //Backprop at LSTM
    //for Vt={i,f,g,o}: Vt = f(a_vt) ;f() respective trig. function
    //---- first get dL/dCt and dL/daot

    flw ft4, 0(s7)//tanh(C10) -> ft4 - ft7
    flw ft5, 4(s7)
    flw ft6, 8(s7)
    flw ft7, 12(s7)
```

```
10
11
12      flw ft8, 480(s8)//ot -> ft8 - ft11
13      flw ft9, 484(s8)
14      flw ft10, 488(s8)
15      flw ft11, 492(s8)
16
17
18      //dL/dCt = dL/dht * (ot - ht*tanh(Ct)) -> [ft0, ..., ft3]
19      fnmsub.s ft0, fs0, ft4, ft8 //-(ht*tanh(Ct)) + ot
20      fmul.s ft0, ft0, fs4 //ans * dL/dht
21
22      fnmsub.s ft1, fs1, ft5, ft9
23      fmul.s ft1, ft1, fs5
24
25      fnmsub.s ft2, fs2, ft6, ft10
26      fmul.s ft2, ft2, fs6
27
28      fnmsub.s ft3, fs3, ft7, ft11
29      fmul.s ft3, ft3, fs7
30
31      //dL/daot = dL/dht * (ht - ht*ot) -> [fa0, ..., fa3]
32      fnmsub.s fa0, ft8, fs0, fs0 //-(ot*ht) + ht
33      fmul.s fa0, fa0, fs4 //ans * dL/dht
34
35      fnmsub.s fa1, ft9, fs1, fs1
36      fmul.s fa1, fa1, fs5
37
38      fnmsub.s fa2, ft10, fs2, fs2
39      fmul.s fa2, fa2, fs6
40
41      fnmsub.s fa3, ft11, fs3, fs3
42      fmul.s fa3, fa3, fs7
43
44      //---- now get dL/dait and dL/dagt
45
46      flw ft4, 0(s8)//it
47      flw ft5, 4(s8)
48      flw ft6, 8(s8)
49      flw ft7, 12(s8)
50
51      flw ft8, 320(s8)//gt
52      flw ft9, 324(s8)
53      flw ft10, 328(s8)
54      flw ft11, 332(s8)
55
56      //dL/dait = dL/dCt * gt * (it - it*it) -> [fa4, ..., fa7]
57      fnmsub.s fa4, ft4, ft4, ft4//-(it * it) + it
58      fmul.s fa4, fa4, ft8//ans * gt
59      fmul.s fa4, fa4, ft0//ans * dL/dCt
60
61      fnmsub.s fa5, ft5, ft5, ft5
62      fmul.s fa5, fa5, ft9
63      fmul.s fa5, fa5, ft1
64
65      fnmsub.s fa6, ft6, ft6, ft6
66      fmul.s fa6, fa6, ft10
67      fmul.s fa6, fa6, ft2
68
69      fnmsub.s fa7, ft7, ft7, ft7
70      fmul.s fa7, fa7, ft11
71      fmul.s fa7, fa7, ft3
72
73      //dL/dagt = dL/dCt * it * (1.0 - gt*gt) -> [ft4, ..., fs7]
74      fnmsub.s fs0, ft8, ft8, fs11//-(gt*gt)+1.0
75      fmul.s ft4, fs0, ft4//ans * it
76      fmul.s ft4, ft4, ft0//ans * dL/dCt
77
78      fnmsub.s fs1, ft9, ft9, fs11
79      fmul.s ft5, fs1, ft5
80      fmul.s ft5, ft5, ft1
```

```
81
82      fnmsub.s fs2, ft10, ft10, fs11
83      fmul.s ft6, fs2, ft6
84      fmul.s ft6, ft6, ft2
85
86      fnmsub.s fs3, ft11, ft11, fs11
87      fmul.s ft7, fs3, ft7
88      fmul.s ft7, ft7, ft3
89
90      //---- get dL/daft
91      //substract 1 to time counter now. If its 0, Ct-1 = C0 = 0 -> skip Ct-1 related
            instructions
92      addi a7, a7, -1
93
94      flw ft8, 160(s8) //ft
95      flw ft9, 164(s8)
96      flw ft10, 168(s8)
97      flw ft11, 172(s8)
98
99      //dL/daft = dL/dCt * Ct-1 * (ft - ft*ft) -> [ft0, ..., ft3]
100     fnmsub.s fs0, ft8, ft8, ft8 //-(ft*ft) + ft
101     fmul.s ft0, fs0, ft0//ans*dL/dCt
102     beqz a7, skip_daft_0//if a7!=0 -> ans=ans*Ct-1
103         flw ft8, -16(s6)
104         fmul.s ft0, ft0, ft8
105     skip_daft_0:
106
107     fnmsub.s fs1, ft9, ft9, ft9
108     fmul.s ft1, fs1, ft1
109     beqz a7, skip_daft_1
110         flw ft9, -12(s6)
111         fmul.s ft1, ft1, ft9
112     skip_daft_1:
113
114     fnmsub.s fs2, ft10, ft10, ft10
115     fmul.s ft2, fs2, ft2
116     beqz a7, skip_daft_2
117         flw ft10, -8(s6)
118         fmul.s ft2, ft2, ft10
119     skip_daft_2:
120
121     fnmsub.s fs3, ft11, ft11, ft11
122     fmul.s ft3, fs3, ft3
123     beqz a7, skip_daft_3
124         flw ft11, -4(s6)
125         fmul.s ft3, ft3, ft11
126     skip_daft_3:
127
128     //dL/daot -> [fa0, ..., fa3]
129     //dL/dait -> [fa4, ..., fa7]
130     //dL/dagt -> [ft4, ..., ft7]
131     //dL/daft -> [ft0, ..., ft3]
```

Finalmente, se pasa a calcular los gradientes de los parámetros a partir de la implementación de las ecuaciones 4.32, 4.33 y 4.34. Para el cálculo de los gradientes de los *bias* (4.32), simplemente se carga el valor actual acumulado del gradiente, se le suma a este valor la derivada correspondiente y se vuelve a guardar en memoria. Para los pesos, primero son calculados los gradientes de los pesos que multiplican a h_{t-1} (4.34), en donde se cargan los valores de h_{t-1} para realizar el cálculo del gradiente, el cual es agregado al gradiente acumulado en cada caso, para posteriormente guardar este valor. Finalmente, se realiza el mismo proceso para los pesos que multiplican a x_{t-1} (4.33), con la diferencia que por supuesto se cargan esta vez los valores de x_{t-1} para los cálculos de gradiente. Debido a que lo anterior consiste en una serie de repeticiones de cargar desde memoria, realizar una sola instrucción de suma o suma+multiplicación y guardar en memoria, no se muestra un fragmento de código,

sin embargo la implementación de esto puede ser encontrada en el código *Listing* 6.2, entre las líneas 1423 a 2128. De la misma forma, la implementación del cálculo de $\frac{\mathrm{d}L}{\mathrm{d}h_{t-1}}$ (ecuación 4.31) consiste en una implementación directa de multiplicación de matrices, la cual puede ser observada en el código *Listing* 6.2 entre las líneas 2131 y 2285

Todo lo anterior es repetido nuevamente para $t-1$, hasta calcular los gradientes en $t=1$, donde se termina el cálculo de gradientes y se pasa a la actualización de parámetros a partir de estos, utilizando el algoritmo "Adam".

4.7.5. Actualización de parámetros

Debido a la enorme cantidad de parámetros a actualizar y, por tanto, la enorme cantidad de veces que se debe utilizar el algoritmo Adam, este es implementado de forma similar a las funciones trigonométricas, en donde es implementado como un método el cual puede ser llamado para realizar el algoritmo sobre los valores cargados en los registros correspondientes. El código *Listing* 4.11 corresponde a la implementación del algoritmo, en donde se aplican las ecuaciones 3.1 a 3.5.

Listing 4.11: Implementación del algoritmo Adam.

```
1
2  adam_crit:
3  //apply adam, where fa0=PARAM (0) , fa1=GRADIENT (dL/d0), fa2=P , fa3=S
4  //betas_and_others values must be preloaded
5  //fs0=learning rate , fs1=beta1 , fs2=beta2 , fs3=1-beta1^t , fs4=1-beta2^t , fs5=
       gradient_divisor
6  //fs6=epsilon
7  //returns adjusted param in fa0, new P in fa2 and new S in fa3
8  //must be called with jal a0, adam_crit
9
10     //---- divides gradient fa1 by fs5
11     fdiv.s fa1, fa1, fs5
12
13     //---- P = P * beta1 + (1-beta1)* dL/d0
14     //       = P * beta1 + (dL/d0 - beta1*dL/d0)
15     //-(beta1*dL/d0) + dL/d0
16     fnmsub.s ft1, fs1, fa1, fa1
17     //P = P * beta1 + ans
18     fmadd.s fa2, fa2, fs1, ft1
19
20     //---- S = S * beta2 + (1-beta2)* dL/d0*dL/d0
21     //       = S * beta2 + ((dL/d0*dL/d0)-beta2 * (dL/d0*dL/d0))
22     //dL/d0*dL/d0
23     fmul.s fa1, fa1, fa1
24     //-(ans * beta2) + ans
25     fnmsub.s ft1, fa1, fs2, fa1
26     //S = S * beta2 + ans
27     fmadd.s fa3, fa3, fs2, ft1
28
29     //---- PP = P/(1- beta1^(t))
30     fdiv.s ft2, fa2, fs3
31
32     //---- SS = S/(1- beta2^(t))
33     fdiv.s ft3, fa3, fs4
34
35     //---- 0 = 0 - lr*PP/(sqrt(SS)+epsilon)
36     //sqrt(SS)
37     fsqrt.s ft3, ft3
38     //ans + epsilon
39     fadd.s ft3, ft3, fs6
40     //PP / ans
```

```
41      fdiv.s ft2, ft2, ft3
42      //0 =- (lr*ans) + 0
43      fnmsub.s fa0, fs0, ft2, fa0
44
45      //---- finish
46      jr a0
```

Luego, para cada uno de los parámetros de la red se carga en el registro `fa0` el valor del parámetro, en `fa1` su gradiente acumulado y en `fa2` y `fa3` el valor de P y S, respectivamente, del parámetro. Además de eso se carga previamente en `fs0` el *learning rate* (0,005), en `fs1` y `fs2` los valores de $\beta_1 = 0{,}9$ y $\beta_2 = 0{,}999$, respectivamente, en `fs3` y `fs4` los valores acumulados de $1 - \beta_1^t$ y $1 - \beta_2^t$ (con t = época de entrenamiento actual), en `fs5` el valor a dividir los gradientes para calcular el promedio de los gradientes acumulados (3342 para los parámetros de las capas *FC*, 33420 para los de la red *LSTM*) y, finalmente, en `fs4` un épsilon correspondiente al valor de punto flotante más pequeño posible, para evitar divisiones por cero. Esto es entonces realizado por el fragmento de código *Listing* 4.12, en donde las últimas 9 líneas de código son repetidas para cada uno de los parámetros de la red.

Listing 4.12: Fragmento de código responsable de la actualización de parámetros utilizando el algoritmo Adam.

```
1
2       //-------------------------------------------------------------
3       //------        ADAM        -----------------------------------
4       //-------------------------------------------------------------
5       // 0 = parameter
6       //P = P * beta1 + (1-beta1)* dL/d0
7       //S = S * beta2 + (1-beta2)* dL/d0*dL/d0
8       //PP = P/(1- beta1^(t))
9       //SS = S/(1- beta2^(t))
10      //0 = 0 - lr*PP/sqrt(SS)
11
12      //load betas and learning rate
13      la t0, betas_and_others
14      //learning rate
15      flw fs0, 16(t0)
16      //betas
17      flw fs1, 0(t0)//beta1
18      flw fs2, 4(t0)//beta2
19
20      flw fs3, 8(t0)//beta1^(t)
21      fmul.s fs3, fs3, fs1
22      fsw fs3, 8(t0)//store for t+1
23      fsub.s fs3, fs11, fs3 //1-beta1^t
24
25      flw fs4, 12(t0)//beta2^(t)
26      fmul.s fs4, fs4, fs2
27      fsw fs4, 12(t0)//store for t+1
28      fsub.s fs4, fs11, fs4 //1-beta2^t
29
30      //other floats used for division
31      flw fs5, 20(t0)//3342.0
32      //flw fs6, 20(t0)//10.0
33
34      //load an epsilon to avoid division by zero
35      flw fs6, 28(t0)
36
37      //ft0 = zero, to set gradients after adam
38      fmv.s.x ft0, x0
39
40
41      //-------------------------------------------------------------fc
42      //a4 -> fc_weight_and_bias
43      //bias
44      flw fa0, 256(a4)
```

```
45      flw fa1, 1028(a4)
46      flw fa2, 1032(a4)
47      flw fa3, 1036(a4)
48      jal a0, adam_crit
49      fsw fa0, 256(a4)
50      fsw ft0, 1028(a4)//set gradient to x0
51      fsw fa2, 1032(a4)
52      fsw fa3, 1036(a4)
53
54      //REPEAT FOR ALL PARAMETERS
```

4.7.6. Entrenamiento y obtencion de resultados

Una vez actualizado todos los parámetros para una época, se vuelve a repetir el proceso completo por 220 épocas, ajustando desfases de punteros cada vez. Terminadas estas 220 épocas, se extraen los valores de los parámetros entrenados de memoria y se vuelve a trabajar sobre el primer código de la implementación de la ejecución de la red. Sobre este, se modifican los valores de los pesos y *biases* para esta vez ejecutar la red con los parámetros entrenados en *RISC-V* y así obtener predicciones y comprobar la capacidad de entrenamiento de la implementación realizada.

4.8. Comparaciones de funcionamiento

A medida que se finalizan las distintas implementaciones, se realizan pruebas de funcionamiento, en donde se obtienen las predicciones de cada una de las redes y se comparan entre ellas para comprobar el buen funcionamiento de cada implementación. Específicamente, primero se realiza un análisis del funcionamiento de la red en *Python*, para luego comparar el funcionamiento de ambas implementaciones de *RISC-V* en función a esta.

Respecto a *RISC-V*, en ambos casos los resultados a comparar son obtenidos del código correspondiente a solo la ejecución de la red, en donde para probar el funcionamiento de este se utilizan parámetros (esto es, los pesos y *biases* correspondientes) pre-entrenados por la red en *Python*. Luego, una vez asegurado un comportamiento apropiado de la ejecución de la red en *RISC-V* y, posteriormente, habiendo implementado en un segundo código el entrenamiento de esta, se comprueba el funcionamiento de este último realizando el entrenamiento la red en *RISC-V* y cargando los parámetros obtenidos sobre la primera implementación, con tal de obtener predicciones con estos nuevos parámetros.

4.9. Mediciones de tiempo

Una vez comprobado el funcionamiento de los códigos implementados, se pasa a realizar unas últimas mediciones de tiempo, con el propósito de comparar la eficiencia de estas implementaciones.

4.9.1. *Python*: Mediciones en *CPU* y *GPU*

Metodología

Para las mediciones de tiempos en *CPU* y *GPU*, se utiliza la implementación realizada en *Python*, midiendo directamente desde *Google Colab* el tiempo de ejecución y entrenamiento de la red implementada. Para esto, se entrena la desde cero 10 veces y, posteriormente, se ejecuta la red utilizando el conjunto completo de datos 10 veces, comparando el tiempo de inicio versus el tiempo final para realizar el cálculo de tiempo. A partir de esto se obtiene un tiempo mínimo, máximo y promedio para ambos casos.

Parámetros de *CPU* y *GPU*

Aunque *Google Colab* no siempre entrega los mismos recursos entre distintas "sesiones de uso", estos recursos se mantienen constantes en una misma sesión, y es entonces posible saber exactamente cual hardware está siendo utilizado. Esto se logra a través de los comandos `!cat /proc/cpuinfo` para la *CPU* y `!nvidia-smi -q` para la *GPU*

La *CPU* utilizada es un procesador Intel(R) Xeon. Esta trabaja a una frecuencia de 2.3GHz, posee un cache de 46MB y un solo núcleo, el cual corre un máximo de 2 *Threads*.

La *GPU* Utilizada es un procesador NVIDIA Tesla K80. Esta trabaja a una frecuencia de aproximadamente 2.5GHz, 24GB de memoria GDDR5 VRAM y 4992 núcleos[22].

4.9.2. RISC-V

Metodología

Como se mencionó previamente, para las mediciones de tiempo de las implementaciones en *RISC-V* se utiliza el simulador *Gem5*. Los parámetros utilizados para la medición se ajustan a la placa de SiFive *HiFive Unleashed*, la cual posee un *SoC* U-54, sobre el cual se diseñaron los programas. Esta placa posee una memoria de 8MB, un cache L1 con 16KB de memoria de instrucciones y 16KB para datos y un cache L2 de 128KB.

Parámetros de Medición

La simulación en *Gem5* fue realizada utilizando las configuraciones del modo SE por defecto utilizando una sola *CPU*, explicitando las características previas y fijando la frecuencia de relojes del sistema y *CPU* en 2.5GHz, con el propósito de tener una frecuencia comparable a las mediciones de *CPU* y *GPU*. Se realizan simulaciones utilizando los tres distintos tipos de *CPU* mencionados en el marco teórico: *TimingSimpleCPU*, *MinorCPU* y *DerivO3CPU*. Aunque el modelo de *MinorCPU* es más cercano a nuestro punto de referencia respecto

al hardware especificado anteriormente, se realizan mediciones sobre estos 3 modelos para obtener resultados más completos

A partir de esto el programa entrega, entre otros datos, el tiempo de simulación, el cual consiste en el tiempo que demoró el programa en ejecutarse bajo los parámetros especificados. Este tiempo sea el utilizado para las comparaciones.

Capítulo 5

Resultados y Análisis

A continuación se presentan los resultados obtenidos del trabajo, comenzando primero con la implementación en *Python*, la cual es utilizada como punto de referencia, para luego entrar en los resultados obtenidos por las implementaciones de la red en *RISC-V* y comparar estos con los resultados previos en *Python*.

5.1. Resultados de *Python*

5.1.1. Funcionamiento

Como fue mencionado previamente, se realiza una implementación de la red a utilizar en *Python* no solo para obtener los tiempos, sino que para verificar que la estructura de la red utilizada sea suficientemente buena. Esto implica que la estructura de la red sea capaz de, a través de su entrenamiento, poder realizar perdiciones que se ajusten a los datos reales con un cierto grado mínimo de certeza.

Figura 5.1: Predicciones de la red implementada en *Python* vs. datos reales.

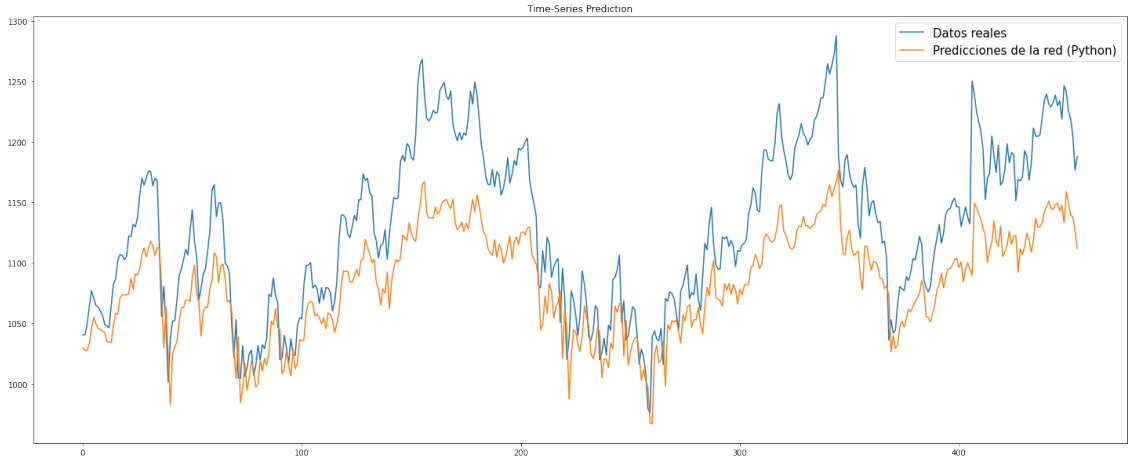

Figura 5.2: Predicciones de la red implementada en *Python* vs. datos reales, para los datos del conjunto validación.

Como se mencionaba previamente en la metodología, se puede observar en los resultados obtenidos en las Figuras 5.1 y 5.2, las predicciones realizadas por la red logran ajustarse con un cierto grado a la forma de los datos reales. Se obtiene además un error *RMSE* de 0.04348 en el conjunto de validación.

Aunque las predicciones tienden a tener una forma similar a los datos reales, el error identificable en la Figura 5.2 no es menor, además de existir un desfase temporal en donde las predicciones , finalmente, indican resultados ya ocurridos, con lo que esta red no se podría considerar una buena implementación para su uso en la realidad.

Sin embargo, el objetivo de esta memoria no consiste en obtener el mejor modelo predictivo para los datos utilizados (predicción de la bolsa de valores), sino que realizar comparaciones de funcionamiento al realizar la misma implementación en *RISC-V*. Es por esto que, aunque el modelo no se ajusta perfectamente a la realidad, ya sea por un entrenamiento no ideal (se entrena por 220 épocas fijas para obtener medidas consistentes a costa de sobre-ajustarse a los datos de entrenamiento) o, simplemente, por la dificultad de realizar predicciones sobre este tipo de información (es extremadamente difícil definir un patrón de la bolsa de valores sin considerar factores externos al historial valórico de esta), se considera que el comportamiento obtenido de la red es suficientemente satisfactorio como para poder distinguir un ajuste sobre los datos de entrenamiento (con lo que es posible afirmar si la misma implementación en *RISC-V* posee un funcionamiento correcto) y poder realizar medidas de tiempo sobre esta red.

5.1.2. Tiempos Medidos

Los tiempos obtenidos a partir de las mediciones en *Python* pueden observarse en la tabla 5.1. Como es de esperarse, la utilización de una *GPU* acelera considerablemente el funcionamiento de la red neuronal, tanto en entrenamiento como ejecución.

	CPU		GPU	
Tiempo	Entrenamiento	Ejecución	Entrenamiento	Ejecución
Promedio	5.54273152 [s]	0.02914178 [s]	1.82004334 [s]	0.00268528 [s]
Minimo	5.44781661 [s]	0.02470827 [s]	1.80727624 [s]	0.00162029 [s]
Máximo	5.80476546 [s]	0.03463578 [s]	1.83167386 [s]	0.00499391 [s]

Tabla 5.1: Resultados de mediciones de tiempo para la implementación en *Python*.

	RISC-V	
CPU	Entrenamiento	Ejecución
SimpleCPU	19.08436743 [s]	0.04687732 [s]
MinorCPU	10.82745946 [s]	0.02224435 [s]
DerivO3CPU	-	0.00658411 [s]

Tabla 5.2: Resultados de mediciones de tiempo para la implementaciones en *RISC-V*.

5.2. Resultados *RISC-V*

5.2.1. Funcionamiento

Como se mencionó previamente, se realizan dos implementaciones de la red, donde la primera consiste solamente en la ejecución de la red con *pesos* y *biases* pre-entrenados en *Python*, mientras que la segunda corresponde al entrenamiento de la red.

Ejecución con parámetros pre-entrenados

Para comprobar el funcionamiento de la ejecución de la red, se compara las predicciones de esta respecto a los resultados en *Python* y los datos reales, utilizando los mismos parámetros de la implementación entrenada en *Python*. Con esto se obtienen las Figuras 5.3 y 5.4, donde la nueva línea verde corresponde a la predicciones realizadas en *RISC-V*.

Como se puede observar, las predicciones obtenidas por la red implementada en *RISC-V* son notablemente similares a la red de *Python* tanto en el conjunto de entrenamiento como validación, con la diferencia de tener un desfase aún mayor que la red en *Python* respecto a los datos reales, evidenciado aun más por el error *RMSE* de 0.06461 obtenido (en comparación al 0.04348 de *Python*). Este desfase se puede atribuir a las funciones trigonométricas debido a que, con lo que concierne a funcionamiento, estas funciones son lo único que intencionalmente difiere entre la implementación en *Python* y *RISC-V*. A pesar de este mayor error presente, se puede observar como de todas formas la implementación en *RISC-V* logra ajustar sus predicciones a la forma de los datos reales, con lo que se comprueba su funcionamiento esperado.

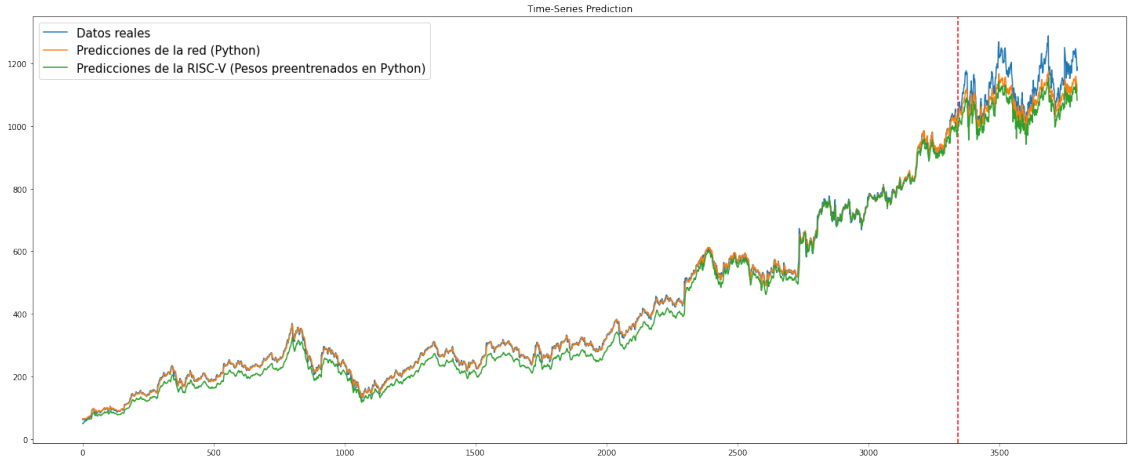

Figura 5.3: Predicciones de la red implementada en *RISC-V* con pesos pre-entrenados, en comparación con los datos reales y la implementación en *Python*.

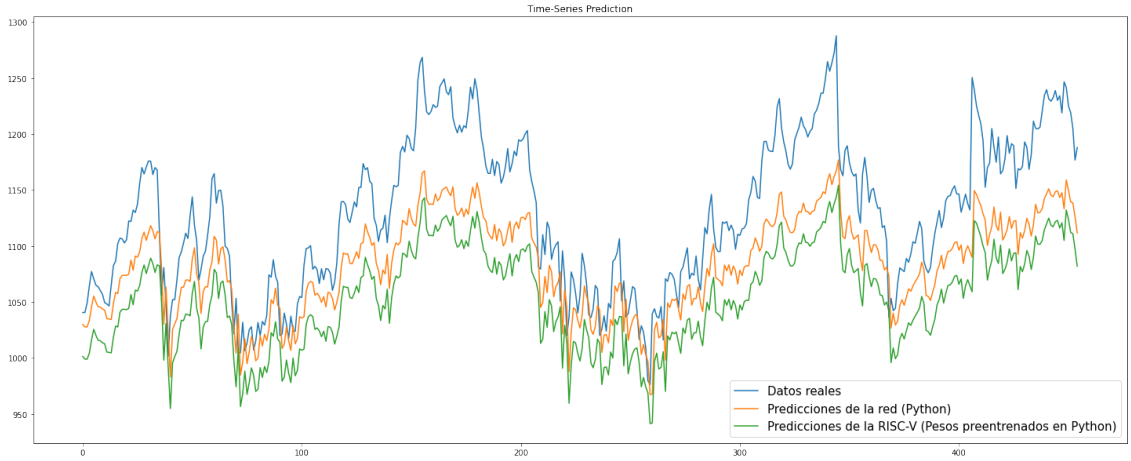

Figura 5.4: Predicciones de la red implementada en *RISC-V* con pesos pre-entrenados, en comparación con los datos reales y la implementación en *Python*, para los datos del conjunto de validación.

Parámetros entrenados en *RISC-V*

Como se menciona en la metodología, para los siguientes resultados se vuelve a ejecutar la red anterior, esta vez habiendo realizado el entrenamiento de sus parámetros en *RISC-V*. A partir de esto se obtienen los resultados de las tablas 5.5 y 5.6, con las nuevas predicciones representadas por la línea roja.

Los resultados obtenidos logran comprobar que es posible entrenar de forma apropiada una red neuronal en *RISC-V*. Es más, debido a que el entrenamiento es realizado tomando en consideración las formas aproximadas de las funciones trigonométricas, este es capaz de compensar por su inexactitud, logrando ajustar los parámetros de la red de forma incluso ligeramente mejor que el entrenamiento en *Python*, obteniéndose un error *RMSE* de validación de 0.03451, aunque esta última diferencia respecto a *Python* podría variar dependiendo de los valores iniciales de los parámetros de la red antes de entrenar. A partir de esto se puede

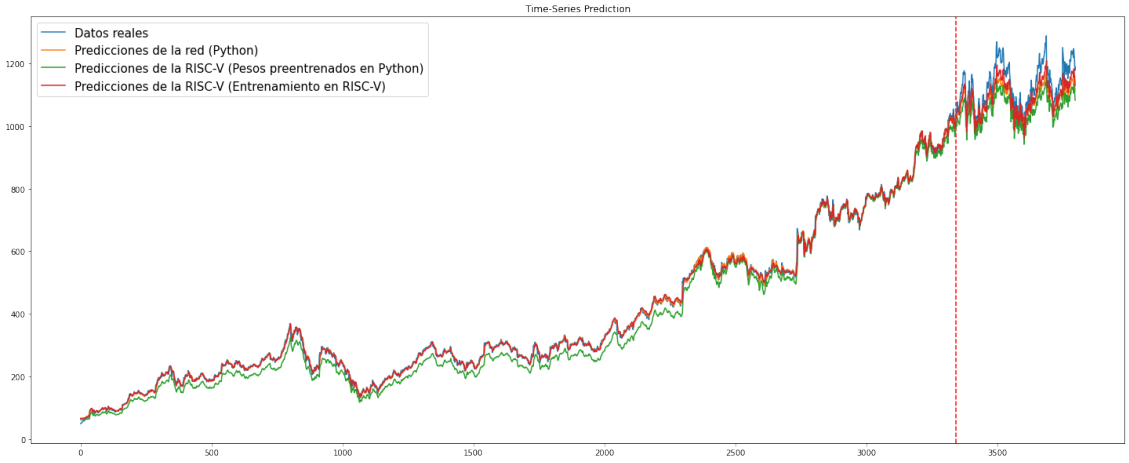

Figura 5.5: Predicciones de la red implementada en *RISC-V* con pesos pre-entrenados, en comparación con los datos reales y la implementación en *Python*.

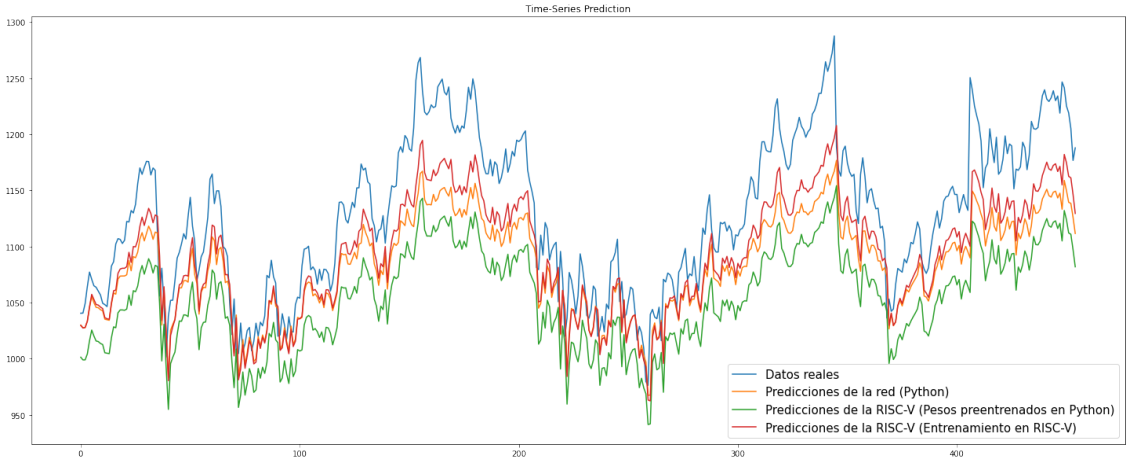

Figura 5.6: Predicciones de la red implementada en *RISC-V* con pesos pre-entrenados, en comparación con los datos reales y la implementación en *Python*, para los datos del conjunto de validación.

formar entonces la tabla de errores 5.3.

Implementación	Error RMSE
Python	0.04348
RISC-V (entrenado en Python)	0.06461
RISC-V (entrenado en RISC-V)	0.03451

Tabla 5.3: Errores *RMSE* obtenidos sobre el conjunto de validación para las distintas implementaciones.

5.2.2. Tiempos Medidos

Habiendo comprobado el funcionamiento de ambas implementaciones, se realizan las simulaciones en *Gem5* mencionadas previamente, obteniéndose los resultados de la tabla 5.2. Lamentablemente, por problemas en el simulador, no fue posible medir el tiempo para la implementación del entrenamiento para una *CPU* del tipo *Out-of-order*.

Como es de esperar, los tiempos obtenidos para *SimpleCPU*, la cual no trabaja con *pipeline*, son notablemente más lentos incluso que una *CPU* para la red en *Python*.

Por el otro lado, la simulación en *MinorCPU* (*pipeline in-order*) posee tiempos más comparables con una *CPU* en *Python*, donde para *RISC-V*, aunque en el entrenamiento sigue siendo más lento, se obtiene un tiempo ligeramente más rápido en la ejecución. Finalmente, para la simulación de *DerivO3CPU* (*pipeline out-of-order*) se logra obtener un tiempo considerablemente más rápido, debido a la capacidad de este tipo de procesador de utilizar recursos a medida que estos se encuentran disponibles, de cierta forma acercándose un poco a lo que podría ser una implementación con múltiples *threads/cores*. Aunque no se lograron obtener medidas para este último modelo respecto al entrenamiento, se estima que el tiempo hubiera sido comparable, e incluso más rápido a los tiempos obtenidos para una *CPU* en *Python*.

A pesar de lo anterior, se puede observar como el uso de una *GPU* sigue siendo considerablemente más rápido tanto en entrenamiento como ejecución de la red neuronal, llegando a ser hasta casi 10 veces más rápida, tanto en entrenamiento como ejecución, en comparación con *MinorCPU* y en promedio aproximadamente 3 veces más rápida para la ejecución de la red en comparación con *DerivO3CPU*.

Capítulo 6

Conclusiones

Para este trabajo se implementó de forma satisfactoria una red neuronal recurrente *LSTM* utilizando la *ISA* de *RISC-V*, en donde a través de la implementación de su entrenamiento y ejecución, permite funcionar de forma similar a una implementación realizada en el lenguaje de mayor nivel *Python*, pudiendo ser posible entrenar esta red a partir de un conjunto de datos de valores de la bolsa para posteriormente obtener predicciones de esta.

Respecto a la eficiencia de la implementaciones realizadas en función a la velocidad de estas, la implementación de la red en *RISC-V* logra obtener tiempos cercanos a la ejecución en una *CPU*. Sin embargo la utilización del procesador *GPU* sigue siendo considerablemente más eficiente. Este resultado se debe principalmente a que: las implementaciones realizadas en *RISC-V* son procesadas en un solo procesador, debido a que no se utilizan instrucciones atómicas ni *multi-threading* para las implementaciones, pues el manejo de esto quedaba fuera del alcance de este trabajo.

A partir de los resultados obtenidos se considera como una decisión factible la futura exploración de la *ISA* de *RISC-V* para el área de inteligencia artificial, al ser capaz de ser utilizada para el funcionamiento de redes neuronales.

6.1. Trabajo Futuro

Una opción de trabajo futuro consiste en la implementación de instrucciones atómicas que posee la *ISA*. La integración de estas instrucciones, así como el uso de hardware capaz de procesar múltiples *threads*, puede llegar a obtener resultados los cuales lograrían competir con aquellos obtenibles en una *GPU*.

Además de esto, como trabajo futuro está la opción de aprovechar la capacidad particular de *RISC-V* de crear instrucciones personalizadas, las cuales han demostrado [16] reducir considerablemente el número de instrucciones necesarias al momento de trabajar con redes neuronales, reduciendo así los tiempos de estos.

Bibliografía

[1] Christopher Olah. Understanding lstm networks. URL: http://colah.github.io/posts/2015-08-Understanding-LSTMs/.

[2] Md. Arif Istiake Sunny, Mirza Mohd Shahriar Maswood, and Abdullah G. Alharbi. Deep learning-based stock price prediction using lstm and bi-directional lstm model. In *2020 2nd Novel Intelligent and Leading Emerging Sciences Conference (NILES)*, pages 87–92, 2020. doi:10.1109/NILES50944.2020.9257950.

[3] Arne Holst. Topic: Smart home, Apr 2021. URL: https://www.statista.com/topics/2430/smart-homes/#:~:text=Thenumberofsmarthomes,settodoubleby2024.

[4] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 6645–6649, 2013. doi:10.1109/ICASSP.2013.6638947.

[5] Haşim Sak, Andrew Senior, and Françoise Beaufays. Long short-term memory based recurrent neural network architectures for large vocabulary speech recognition, Feb 2014. URL: https://arxiv.org/abs/1402.1128.

[6] Jeremy Appleyard, Tomas Kocisky, and Phil Blunsom. Optimizing performance of recurrent neural networks on gpus, Apr 2016. URL: https://arxiv.org/abs/1604.01946.

[7] Jason Dsouza. What is a gpu and do you need one in deep learning?, Apr 2020. URL: https://towardsdatascience.com/what-is-a-gpu-and-do-you-need-one-in-deep-learning-718b9597aa0d.

[8] Nipa Chowdhury and Mohammod Abul kashem. A comparative analysis of feed-forward neural network recurrent neural network to detect intrusion. In *2008 International Conference on Electrical and Computer Engineering*, pages 488–492, 2008. doi:10.1109/ICECE.2008.4769258.

[9] Shell Waterman. Design of the risc-v instruction set architecture, Sep 2016. URL: https://escholarship.org/uc/item/7zj0b3m7.

[10] Codasip. Creating domain-specific processors using custom risc-v isa instructions, Sep 2020. URL: https://codasip.com/2020/09/23/creating-domain-specific-processors-using-custom-risc-v-isa-instructions/.

[11] Freedom Studio by SiFive [Online]. URL: https://www.sifive.com/software.

[12] Gem5 official website [Online]. URL: https://www.gem5.org/.

[13] José Manuel Giralt Álvarez. Desarrollo de técnicas de procesamiento paralelo a nivel de lenguaje assembler para el procesador risc-v. URL: https://repositorio.uchile.cl/handle/2250/181127.

[14] Gem5 wiki: Documentación. URL: http://www.m5sim.org/Documentation.

[15] Sashank J. Reddi, Satyen Kale, and Sanjiv Kumar. On the convergence of adam and beyond. In *International Conference on Learning Representations*, 2018. URL: https://openreview.net/forum?id=ryQu7f-RZ.

[16] Renzo Andri, Tomas Henriksson, and Luca Benini. Extending the risc-v isa for efficient rnn-based 5g radio resource management. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2020. doi:10.1109/DAC18072.2020.9218496.

[17] Che-Wei Lin and Jeen-Shing Wang. A digital circuit design of hyperbolic tangent sigmoid function for neural networks. In *2008 IEEE International Symposium on Circuits and Systems*, pages 856–859, 2008. doi:10.1109/ISCAS.2008.4541553.

[18] Tomás Letelier. Notebook de google collaboratory utilizado para el trabajo [online]. URL: https://colab.research.google.com/drive/1qcFZK38pLHJAFhZVfIffCCRBHzvMVMiQ?usp=sharing.

[19] "Yahoo! Finance". Google stocks historical data [online]. URL: https://finance.yahoo.com/quote/GOOG/history?p=GOOG.

[20] HiFive Unleashed by SiFive [Online]. URL: https://www.sifive.com/boards/hifive-unleashed.

[21] Wikipedia. "loop unrolling" [online]. URL: https://en.wikipedia.org/wiki/Loop_unrolling.

[22] NVIDIA. Procesador tesla k80 [online]. URL: https://www.nvidia.com/es-la/data-center/tesla-k80/.

[23] RISC-V. Isa reference card [online]. URL: https://www.cl.cam.ac.uk/teaching/1617/ECAD+Arch/files/docs/RISCVGreenCardv8-20151013.pdf.

[24] Tomás Letelier. Github con los codigos completos [online]. URL: https://github.com/Tomz295/Trabajo_Memoria_LSTM_Riscv.

ANEXO

Código de Ejecución de la red

Se presenta en el *Listing* 6.1 a continuación el código correspondiente a la ejecución de la red. Ciertas partes han sido cortadas/abreviadas para una mejor lectura. El código completo se encuentra en [24].

Listing 6.1: Código abreviado de la ejecución de la red

```
.section .data
//Data to feed the neural net
//float format was too heavy to copy-paste, had to be copied as HEX
InputXData:
.word 0xbfa7dcd8,0xbfa6a4eb,0xbfa77abb,0xbfa6547d,[...];



//floats of all weights for the input (Wx)
weights_floats_input:
.float -0.35827997, 0.05938131, 0.18730448, -0.013169811, -0.23521963;
.float -0.46681443, 0.08964378, 0.20164162, -0.31895328, -0.23556578;
.float 0.3040431, 0.17712256, -0.3461838, 0.36348435, 0.47444913;
.float -0.17302854, 0.18428648, 0.29201257, 0.09435762, -0.00930144;

.float 0.44754723, -0.08066094, 0.4581912, 0.52598375, -0.42307752;
.float 0.2155123, 0.26821846, 0.42700484, -0.054282337, 0.31792504;
.float -0.2386188, -0.15138388, -0.069781214, 0.0038194975, 0.11908612;
.float 0.25664034, 0.104205206, 0.1354457, 0.15202136, -0.16998766;

.float 0.13092172, 0.23487923, -0.39323586, 0.29078865, 0.29866523;
.float -0.44483444, 0.05683437, 0.16578467, 0.39044386, 0.24891873;
.float 0.12589498, -0.52437997, 0.1811092, 0.3858903, 0.41356215;
.float 0.39401585, -0.29017258, 0.26738435, -0.31785884, 0.19933777;

.float 0.036730576, -0.28680432, -0.3026414, 0.1497047, 0.5197461;
.float -0.230515, -0.4806044, -0.28860497, 0.17460811, -0.10117014;
.float -0.26617596, 0.14041893, 0.0045193485, 0.5754174, -0.022015592;
.float -0.13543461, 0.29685494, -0.22504038, 0.019144073, 0.06714319;

//floats of all weights for the hidden state (Wh)
weights_floats_hidden:
.float 0.41882938, 0.10439411, 0.0028533696, -0.19168808;
.float 0.15462692, 0.12448547, 0.044238944, -0.45048884;
.float -0.4734179, 0.09983343, -0.40681022, 0.3505679;
.float -0.0017518946, 0.12710464, 0.41552475, 0.123120986;

.float 0.17878428, 0.17363349, -0.40339258, 0.18231148;
.float -0.60326785, 0.50398374, -0.27074966, 0.2935585;
.float 0.028193017, 0.06583899, -0.13050707, -0.3397992;
.float -0.3340697, -0.14149238, 0.24763599, 0.19448255;

.float 0.23519517, -0.43452188, -0.09191109, 0.33462086;
.float -0.25800848, -0.37493372, 0.31577832, 0.2242885;
```

```
.float -0.26518077, -0.25358906, -0.5422076, 0.012269103;
.float 0.34050235, -0.3401385, 0.12278287, 0.303648;

.float 0.4314578, 0.36833993, 0.3307672, -0.20069674;
.float 0.0196411, 0.34530383, -0.2386732, 0.28658998;
.float 0.10797291, -0.5087089, -0.26668712, -0.052001294;
.float -0.45437226, 0.4973123, -0.14873752, -0.10845318;

//floats of all weights for the input (bx)
biases_floats_input:
.float 0.44384044, -0.14330973, 0.35945147, -0.47605035;

.float -0.12206475, -0.3062925, 0.27499866, 0.04957947;

.float 0.4390355, -0.50898594, -0.13655037, 0.1227533;

.float 0.2787063, 0.15745397, -0.09274792, 0.40270627;

//floats of all weights for the hidden state (bh)
biases_floats_hidden:
.float -0.180783, 0.35506147, -0.48006466, 0.39342412;

.float 0.4363031, -0.41448334, 0.37773272, 0.08724673;

.float 0.0033176043, -0.29198974, 0.16630256, 0.50002426;

.float 0.11912934, 0.4607911, 0.28142205, 0.21240686;

//first Fully Connected Layer weights and biases
//4 columns (numbers) per row
fc_1_weight:
.float -0.14141054, 0.16883655, -0.39145327, [...];
//1 column per row
fc_1_bias:
.float 0.4468142, 0.017939115, -0.49321836, [...];

//second Fully Connected Layer weights and bias
fc_weight_and_bias:
//64 weights
.float -0.08772384, -0.062091853, -0.090731055, [...];
//bias located at address+256 (4*64)
.float 0.030288255;




//FC_1 layer values. 64 floats
fc_1_values:
.float 0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,[...],0.0;\\total of 64 "0.0"s


floats_ht:
.float 0.0, 0.0, 0.0, 0.0;//t=1
.float 0.0, 0.0, 0.0, 0.0;//t=2
.float 0.0, 0.0, 0.0, 0.0;//t=3
.float 0.0, 0.0, 0.0, 0.0;//t=4
.float 0.0, 0.0, 0.0, 0.0;//t=5
.float 0.0, 0.0, 0.0, 0.0;//t=6
.float 0.0, 0.0, 0.0, 0.0;//t=7
.float 0.0, 0.0, 0.0, 0.0;//t=8
.float 0.0, 0.0, 0.0, 0.0;//t=9
.float 0.0, 0.0, 0.0, 0.0;//t=10

floats_ct:
.float 0.0, 0.0, 0.0, 0.0;

float_final_output:
.float 0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,[...],0.0;\\total of 3797 "0.0"s
//numer one, used in fs11
float_one:
    .float 1.0;
```

```
116
117 //numbers used in tanh
118 tanh_floats:
119     .float 2.0, 0.25;
120
121 //numbers used in sigmoid
122 sig_floats:
123     .float 4.0, 0.5, 0.25, 0.03125;
124
125
126 .section .text
127
128 .global main
129
130 //register use:
131 /*
132 s0      -> Input data pointer
133 s1 - s4 -> Weights and biases pointers
134 s5 & s6 -> ht & ct pointers
135 s7 - s10-> FREE
136 s11     -> FeedForward loop counter
137
138 a0      -> jal/ret adress
139 a1 - a3 -> FC_1 address (only used when calculating FC)
140 a4      -> FREE
141 a5      -> Results/Outputs Pointer
142 a6      -> LSTM [i] offset counter
143 a7      -> LSTM time step counter
144
145
146
147
148 */
149 #run_code:
150 main:
151
152     //pointer to the output of the network (a5)
153     la a5, float_final_output
154
155     //load float 1.0 into register fs11 for future use
156     la t0, float_one;
157     flw fs11, 0(t0); //fs11 = 1.0
158
159     // a0 function return adress
160
161     //input data pointer
162     la s0, InputXData
163 /*
164     //Weights and biases
165     la s1, weights_floats_input
166     la s2, weights_floats_hidden
167     la s3, biases_floats_input
168     la s4, biases_floats_hidden
169 */
170 /*
171     //ht & ct
172     la s5, floats_ht
173     la s6, floats_ct
174 */
175
176     //Feed Forward counter
177     //loop forward_pass_full_restart 3797 times
178     //exits on s11==0
179     li s11, 3797
180
181
182     forward_pass_full_restart:
183
184     //ht & ct
185     la s5, floats_ht
186     la s6, floats_ct
```

```
187
188     //reset ct with zeros
189     fmv.s.x ft0, x0
190
191     //ct
192     fsw ft0, 0(s6)
193     fsw ft0, 4(s6)
194     fsw ft0, 8(s6)
195     fsw ft0, 12(s6)
196
197
198     //timestep counter
199     //a7==0 indicates t=0 (ht = 0 & ct=0)
200     li a7, 0
201
202
203
204     forward_pass_loop:
205     //forward pass for a single time step
206     //ht is not needed the first pass
207
208
209     //reset offset/counter
210     //a6 counts [i] loops (from 4 to 0)
211     li a6, 4
212
213     //Reset weights and biases pointers
214     la s1, weights_floats_input
215     la s2, weights_floats_hidden
216     la s3, biases_floats_input
217     la s4, biases_floats_hidden
218
219
220     //Load x into float registers
221     //x  = [fs4 ... fs8]
222     flw fs4, 0(s0)
223     flw fs5, 4(s0)
224     flw fs6, 8(s0)
225     flw fs7, 12(s0)
226     flw fs8, 16(s0)
227
228
229
230     forward_pass_matrix_mul:
231     //loops while a6 > 0
232
233     //---- fa1 -> i
234
235     //matrix multiplication for i, f, g & o
236     //ht vector stored as = [fs0 ... fs3]
237     //x vector stored in x  = [fs4 ... fs8]
238     //Weights and biases must be the following (with proper offsets)
239     // s1, weights_floats_input
240     // s2, weights_floats_hidden
241     // s3, biases_floats_input
242     // s4, biases_floats_hidden
243     //if t=0 flag a7 is 0, else a7=1
244     //(skips Wh*h multiplications as h=0 when t=0)
245     //result is saved at fa6 to be used in tanh or sig functions
246
247     //Wxi * x
248     flw ft0, 0(s1)
249     fmul.s fa6, ft0, fs4
250
251     flw ft0, 4(s1)
252     fmadd.s fa6, ft0, fs5 ,fa6
253
254     flw ft0, 8(s1)
255     fmadd.s fa6, ft0, fs6 ,fa6
256
257     flw ft0, 12(s1)
```

```
    fmadd.s fa6, ft0, fs7 ,fa6

    flw ft0, 16(s1)
    fmadd.s fa6, ft0, fs8 ,fa6

    //ans + bx
    flw ft0, 0(s3)
    fadd.s fa6, ft0, fa6

    //t=0? (a7=0?) if so, skip Wh*h (h is 0)
    beqz a7, mat_mul_skip_h_i

    //ans + (Wh * h)
    flw ft0, 0(s2)
    fmadd.s fa6, ft0, fs0, fa6

    flw ft0, 4(s2)
    fmadd.s fa6, ft0, fs1, fa6

    flw ft0, 8(s2)
    fmadd.s fa6, ft0, fs2, fa6

    flw ft0, 12(s2)
    fmadd.s fa6, ft0, fs3, fa6

    mat_mul_skip_h_i:
    //ans + bh
    flw ft0, 0(s4)
    fadd.s fa6, ft0, fa6

    jal a0, sig_fa6 //apply sigmoid
    fmv.s fa1, fa7

    //a7=0? if so, skip f
    beqz a7, skip_f



    //---- fa2 -> f
    //Wxi * x
    flw ft0, 80(s1)
    fmul.s fa6, ft0, fs4

    flw ft0, 84(s1)
    fmadd.s fa6, ft0, fs5 ,fa6

    flw ft0, 88(s1)
    fmadd.s fa6, ft0, fs6 ,fa6

    flw ft0, 92(s1)
    fmadd.s fa6, ft0, fs7 ,fa6

    flw ft0, 96(s1)
    fmadd.s fa6, ft0, fs8 ,fa6

    //ans + bx
    flw ft0, 16(s3)
    fadd.s fa6, ft0, fa6

    //t=0? (a7=0?) if so, skip Wh*h (h is 0)
    beqz a7, mat_mul_skip_h_f

    //ans + (Wh * h)
    flw ft0, 64(s2)
    fmadd.s fa6, ft0, fs0, fa6

    flw ft0, 68(s2)
    fmadd.s fa6, ft0, fs1, fa6

    flw ft0, 72(s2)
    fmadd.s fa6, ft0, fs2, fa6
```

```
329
330     flw ft0, 76(s2)
331     fmadd.s fa6, ft0, fs3, fa6
332
333     mat_mul_skip_h_f:
334     //ans + bh
335     flw ft0, 16(s4)
336     fadd.s fa6, ft0, fa6
337
338
339     jal a0, sig_fa6 //apply sigmoid
340     fmv.s fa2, fa7
341
342
343     skip_f:
344     //---- fa3 -> g
345     //Wxi * x
346     flw ft0, 160(s1)
347     fmul.s fa6, ft0, fs4
348
349     flw ft0, 164(s1)
350     fmadd.s fa6, ft0, fs5 ,fa6
351
352     flw ft0, 168(s1)
353     fmadd.s fa6, ft0, fs6 ,fa6
354
355     flw ft0, 172(s1)
356     fmadd.s fa6, ft0, fs7 ,fa6
357
358     flw ft0, 176(s1)
359     fmadd.s fa6, ft0, fs8 ,fa6
360
361     //ans + bx
362     flw ft0, 32(s3)
363     fadd.s fa6, ft0, fa6
364
365     //t=0? (a7=0?) if so, skip Wh*h (h is 0)
366     beqz a7, mat_mul_skip_h_g
367
368     //ans + (Wh * h)
369     flw ft0, 128(s2)
370     fmadd.s fa6, ft0, fs0, fa6
371
372     flw ft0, 132(s2)
373     fmadd.s fa6, ft0, fs1, fa6
374
375     flw ft0, 136(s2)
376     fmadd.s fa6, ft0, fs2, fa6
377
378     flw ft0, 140(s2)
379     fmadd.s fa6, ft0, fs3, fa6
380
381     mat_mul_skip_h_g:
382     //ans + bh
383     flw ft0, 32(s4)
384     fadd.s fa6, ft0, fa6
385
386     jal a0, tanh_fa6 //apply tanh
387     fmv.s fa3, fa7
388
389
390
391     //---- fa4 -> o
392     //Wxi * x
393     flw ft0, 240(s1)
394     fmul.s fa6, ft0, fs4
395
396     flw ft0, 244(s1)
397     fmadd.s fa6, ft0, fs5 ,fa6
398
399     flw ft0, 248(s1)
```

```
400     fmadd.s fa6, ft0, fs6 ,fa6
401
402     flw ft0, 252(s1)
403     fmadd.s fa6, ft0, fs7 ,fa6
404
405     flw ft0, 256(s1)
406     fmadd.s fa6, ft0, fs8 ,fa6
407
408     //ans + bx
409     flw ft0, 48(s3)
410     fadd.s fa6, ft0, fa6
411
412     //t=0? (a7=0?) if so, skip Wh*h (h is 0)
413     beqz a7, mat_mul_skip_h_o
414
415     //ans + (Wh * h)
416     flw ft0, 192(s2)
417     fmadd.s fa6, ft0, fs0, fa6
418
419     flw ft0, 196(s2)
420     fmadd.s fa6, ft0, fs1, fa6
421
422     flw ft0, 200(s2)
423     fmadd.s fa6, ft0, fs2, fa6
424
425     flw ft0, 204(s2)
426     fmadd.s fa6, ft0, fs3, fa6
427
428     mat_mul_skip_h_o:
429     //ans + bh
430     flw ft0, 48(s4)
431     fadd.s fa6, ft0, fa6
432
433
434     jal a0, sig_fa6 //apply sigmoid
435     fmv.s fa4, fa7
436
437     //fa1 -> i
438     //fa2 -> f
439     //fa3 -> g
440     //fa4 -> o
441     //calculate ct = f * ct-1 + i * g
442     // ct = fa2 * 0(s6) + fa1 * fa3
443
444     //i * g
445     fmul.s fa6, fa1, fa3
446
447     // if a7=0 -> ct-1 = 0 -> ct = fa1 * fa3
448     beqz a7, skip_ct1_mul
449
450     //else: ct-1 * f + ans
451     flw ft1, 0(s6)
452     fmadd.s fa6, ft1, fa2, fa6
453
454     skip_ct1_mul:
455     //store ct
456     fsw fa6, 0(s6)
457
458     //ht+t = o * tanh(ct)
459     jal a0, tanh_fa6
460     //fa7 = tanh(ct) * o
461     fmul.s fa7, fa7, fa4
462     //store ht+1
463     fsw fa7, 0(s5)
464
465
466     //---- repeat for [i+1] ----
467     addi a6, a6, -1
468     beqz a6, end_index_loop
469
470     //reset and offset wieghts to the next row
```

```
    addi s1, s1, 20
    addi s2, s2, 16
    //reset and offset biases to the next column
    addi s3, s3, 4
    addi s4, s4, 4

    //offset ht+1 and ct to next number
    addi s5, s5, 4
    addi s6, s6, 4

    j forward_pass_matrix_mul


    end_index_loop:
    //time step finished, move to next time step
    //unless already done with 10 time steps -> check a7
    addi a7, a7, 1
    slti t0, a7, 10 //if a7 < 10 -> t0 = 1 -> not done yet
    beqz t0, end_forward_pass_loop //t0 = 0 -> done with time steps, move on to FC

    //else: start over with XData offset of 5 numbers (+20)
    addi s0, s0, 20


    //remove ht & ct offset
    addi s5, s5, -12
    addi s6, s6, -12

    //Load/Reload ht into float registers
    //ht = [fs0 ... fs3]
    flw fs0, 0(s5)
    flw fs1, 4(s5)
    flw fs2, 8(s5)
    flw fs3, 12(s5)


    //and offset ht+1 to the next timestep "set"
    addi s5, s5, 16 //(4 floats)


    j forward_pass_loop //repeat one time step

    end_forward_pass_loop:
    //done with data on the LSTM
    //do Fully Connected next

    //remove ht offset and load floats
    addi s5, s5, -12
    //ht = [fs0 ... fs3]
    flw fs0, 0(s5)
    flw fs1, 4(s5)
    flw fs2, 8(s5)
    flw fs3, 12(s5)


    //----- apply ReLu flag -----
    //relu flag is used to skip the multiplication,
    //instead of setting the number to zero (same result)
    //flag is 1 if number is =< 0 -> should skip multiplication
    fmv.s.x ft3, x0 //set zero

    //flags:
    //fs0 -> t0
    //fs2 -> t1
    //fs2 -> t2
    //fs3 -> t3
    fle.s t0, fs0, ft3
    fle.s t1, fs1, ft3
    fle.s t2, fs2, ft3
    fle.s t3, fs3, ft3

```

```
    //addresses for fc_1 weights, biases and results
    la a1, fc_1_weight
    la a2, fc_1_bias
    la a3, fc_1_values

    //adresses for fc weights and bias
    la a4, fc_weight_and_bias

    //fa0 -> final output
    flw fa0, 256(a4)

    //Following code loop is unraveled to save 4*64 instructions
    //it feeds ht into each fc_1 neuron, which then feeds into fc

    //---------------------------------
    //------    "LOOP" START   -------
    //---------------------------------

    //0
    flw ft0, 0(a2)

    bnez t0, skip_0_0
        flw ft1, 0(a1)
        fmadd.s ft0, ft1, fs0, ft0
    skip_0_0:

    bnez t1, skip_1_0
        flw ft1, 4(a1)
        fmadd.s ft0, ft1, fs1, ft0
    skip_1_0:

    bnez t2, skip_2_0
        flw ft1, 8(a1)
        fmadd.s ft0, ft1, fs2, ft0
    skip_2_0:

    bnez t3, skip_3_0
        flw ft1, 12(a1)
        fmadd.s ft0, ft1, fs3, ft0
    skip_3_0:

    fsw ft0, 0(a3)

    //relu, then feed to fc
    fle.s t4, ft0, ft3//relu flag
    bnez t4, skip_fc_0//if ft0<=0 then skip fmadd.s
        flw ft1, 0(a4)
        fmadd.s fa0, ft0, ft1, fa0
    skip_fc_0:

    //1
    flw ft0, 4(a2)
    bnez t0, skip_0_1
        flw ft1, 16(a1)
        fmadd.s ft0, ft1, fs0, ft0
    skip_0_1:
    bnez t1, skip_1_1
        flw ft1, 20(a1)
        fmadd.s ft0, ft1, fs1, ft0
    skip_1_1:
    bnez t2, skip_2_1
        flw ft1, 24(a1)
        fmadd.s ft0, ft1, fs2, ft0
    skip_2_1:
    bnez t3, skip_3_1
        flw ft1, 28(a1)
        fmadd.s ft0, ft1, fs3, ft0
    skip_3_1:
    fsw ft0, 4(a3)
    fle.s t4, ft0, ft3
    bnez t4, skip_fc_1
```

```
        flw ft1, 4(a4)
        fmadd.s fa0, ft0, ft1, fa0
    skip_fc_1:


    //---------------------------------------
    //THIS CODE REPEATS A TOTAL OF 64 TIMES
    //REMOVED DUPLICATES 2 TO 61 FOR READABILITY
    //---------------------------------------

    //62
    flw ft0, 248(a2)
    bnez t0, skip_0_62
        flw ft1, 992(a1)
        fmadd.s ft0, ft1, fs0, ft0
    skip_0_62:
    bnez t1, skip_1_62
        flw ft1, 996(a1)
        fmadd.s ft0, ft1, fs1, ft0
    skip_1_62:
    bnez t2, skip_2_62
        flw ft1, 1000(a1)
        fmadd.s ft0, ft1, fs2, ft0
    skip_2_62:
    bnez t3, skip_3_62
        flw ft1, 1004(a1)
        fmadd.s ft0, ft1, fs3, ft0
    skip_3_62:
    fsw ft0, 248(a3)
    fle.s t4, ft0, ft3
    bnez t4, skip_fc_62
        flw ft1, 248(a4)
        fmadd.s fa0, ft0, ft1, fa0
    skip_fc_62:

    //63
    flw ft0, 252(a2)
    bnez t0, skip_0_63
        flw ft1, 1008(a1)
        fmadd.s ft0, ft1, fs0, ft0
    skip_0_63:
    bnez t1, skip_1_63
        flw ft1, 1012(a1)
        fmadd.s ft0, ft1, fs1, ft0
    skip_1_63:
    bnez t2, skip_2_63
        flw ft1, 1016(a1)
        fmadd.s ft0, ft1, fs2, ft0
    skip_2_63:
    bnez t3, skip_3_63
        flw ft1, 1020(a1)
        fmadd.s ft0, ft1, fs3, ft0
    skip_3_63:
    fsw ft0, 252(a3)
    fle.s t4, ft0, ft3
    bnez t4, skip_fc_63
        flw ft1, 252(a4)
        fmadd.s fa0, ft0, ft1, fa0
    skip_fc_63:



    //---------------------------------
    //------     "LOOP" END     ------
    //---------------------------------

    fsw fa0, 0(a5)


    //---- FORWARD PASS END ----
    //restart, with input and output offset
```

```
    //input(s0): go back 8 rows = -160
    addi s0, s0, -160

    //output(a5) offset -> 1 float  = +4
    addi a5, a5, 4

    //"add" -1 to the FeedForward counter (s11)
    //loop feed forward 3797 times
    //repeat until counter reaches 0
    addi s11, s11, -1

    //if s11 != 0 -> repeat
    bnez s11, forward_pass_full_restart


call exit



tanh_fa6:
//apply tanh(x) where x = fa6
//if -2 < x < 2  -> tanh = x - 0.25*x  *sign(x)
//else tanh=sign(x)
//tanh result stored in fa7
//must be called with jal a0, tanh_fa6


    //---- load address of floats to be used
    la t1, tanh_floats

    //---- load 2 in ft10 and check if abs(x)<2
    flw ft10, 0(t1) //ft10 = 2
    fabs.s ft11, fa6 //ft11 = abs(x)
    flt.s t2, ft11, ft10 //t2 == 1 if |x|<2
    beqz t2, tanh_is_not_in_curve // t2 ==0 -> skip to return sign(x)

    //---- else: do x - 0.25*x  *sign(x)
    //x
    fmul.s fa7, fa6, fa6
    //ans*sign(x)
    fsgnj.s fa7, fa7, fa6 //sign inject fa7(ans) with fa6(x) sign
    //-(ans*0.25)+x
    flw ft10, 4(t1); // ft10 = 0.25
    fnmsub.s fa7, fa7, ft10, fa6

    //---- finish
    jr a0

//---- x < -2 or 2 < x
tanh_is_not_in_curve: //return sign
    fsgnj.s fa7, fs11, fa6

jr a0
//---- end of tanh




sig_fa6:
//apply sigmoid(x) where x = fa6
//if x <= -4     -> sig = 0
//  -4 < x < 4  -> sig = 0.5 + 0.25*x - 0.03125*x  *sign(x)
//   4 <= x      -> sig = 1
//sig result stored in fa7
//must be called with jal a0, sig_fa6


    //---- load address of floats to be used
```

```
    la t1, sig_floats

    //---- load 4 and check if abs(x)<4
    flw ft10, 0(t1) //ft10 = 4
    fabs.s ft11, fa6 //ft11 = abs(x)
    flt.s t2, ft11, ft10 //t2 == 1 if |x|<4
    beqz t2, sig_is_not_in_curve // t2 ==0 -> skip to return 0 or 1

    //---- sig = 0.5 + 0.25*x - 0.03125*x  *sign(x)
    //x
    fmul.s fa7, fa6, fa6
    //ans*sign(x)
    fsgnj.s fa7, fa7, fa6 //sign inject fa7(ans) with fa6(x) sign
    //-(ans*0.03125)+0.5
    flw ft10, 4(t1) // ft10 = 0.5
    flw ft11, 12(t1) // ft11 = 0.03125
    fnmsub.s fa7, fa7, ft11, ft10 //fa7 = -(fa7 * ft11) + ft10

    //(0.25*x)+ans
    flw ft10, 8(t1) // ft10 = 0.25
    fmadd.s fa7, ft10, fa6, fa7 //fa7 = (fa7 * ft10) + fa7

    //---- finish
    jr a0

//---- x < -4 or 4 < x
sig_is_not_in_curve:
    // 4 <= x?
    fle.s t2, ft10, fa6 //t2==1 if 4<=x , else (x < -4) t2==0
    fcvt.s.w fa7, t2    //fa7 = float(t2)

jr a0
//---- end of sigmoid
```

Código de Entrenamiento de la red

Se presenta en el *Listing* 6.2 a continuación el código correspondiente al entrenamiento de la red. Ciertas partes han sido cortadas/abreviadas para una mejor lectura. El código completo se encuentra en [24].

Listing 6.2: Código abreviado del entrenamiento de la red

```
.section .data
//Data to feed the neural net
//float format was too heavy to copy-paste, had to be copied as HEX
InputXData:
    .word 0xbfa7dcd8,0xbfa6a4eb,0xbfa77abb,[...];
//Training data
OutputYData:
    .word 0x0,0x3a25a235,0x3a7197af,0x3a72a5cc,[...];


//floats of all weights for the input (Wx)
weights_floats_input:
.float -0.4216829, -0.004383266, 0.123098135, -0.077561855, -0.29961163;
.float -0.47134632, 0.08512348, 0.1966759, -0.323861, -0.24047363;
.float 0.20856214, 0.08093691, -0.44261116, 0.26686895, 0.37783372;
.float -0.2566322, 0.10052627, 0.20792532, 0.01020056, -0.09345847;

.float 0.38641912, -0.14216578, 0.39630258, 0.46388924, -0.48517215;
.float 0.23581558, 0.2884431, 0.44685048, -0.034435213, 0.33777213;
.float -0.37306458, -0.28682595, -0.20564377, -0.13230413, -0.017037451;
.float 0.17520684, 0.022608936, 0.053577006, 0.07007998, -0.25192904;

.float 0.103973866, 0.20773745, -0.4210474, 0.26291317, 0.27078974;
```

```
.float -0.425669, 0.07602745, 0.18534493, 0.40995592, 0.26843077;
.float 0.2135818, -0.4413525, 0.26417184, 0.46630788, 0.49397957;
.float 0.2994073, -0.38495362, 0.17222542, -0.4131, 0.10409653;

.float -0.07575834, -0.39979917, -0.41603988, 0.03611195, 0.40615332;
.float -0.2240789, -0.4741977, -0.28268385, 0.1805619, -0.09521639;
.float -0.45738965, -0.051279545, -0.18734187, 0.3835225, -0.21391016;
.float -0.23530155, 0.19664788, -0.3255785, -0.08149612, -0.033497036;

//Gradient: offset 320
.float 0.0, 0.0, 0.0, 0.0, 0.0;//i
.float 0.0, 0.0, 0.0, 0.0, 0.0;
.float 0.0, 0.0, 0.0, 0.0, 0.0;
.float 0.0, 0.0, 0.0, 0.0, 0.0;

.float 0.0, 0.0, 0.0, 0.0, 0.0;//f
.float 0.0, 0.0, 0.0, 0.0, 0.0;
.float 0.0, 0.0, 0.0, 0.0, 0.0;
.float 0.0, 0.0, 0.0, 0.0, 0.0;

.float 0.0, 0.0, 0.0, 0.0, 0.0;//g
.float 0.0, 0.0, 0.0, 0.0, 0.0;
.float 0.0, 0.0, 0.0, 0.0, 0.0;
.float 0.0, 0.0, 0.0, 0.0, 0.0;

.float 0.0, 0.0, 0.0, 0.0, 0.0;//o
.float 0.0, 0.0, 0.0, 0.0, 0.0;
.float 0.0, 0.0, 0.0, 0.0, 0.0;
.float 0.0, 0.0, 0.0, 0.0, 0.0;

//P : off 640
.float 0.0, 0.0, 0.0, 0.0, 0.0;
.float 0.0, 0.0, 0.0, 0.0, 0.0;
.float 0.0, 0.0, 0.0, 0.0, 0.0;
.float 0.0, 0.0, 0.0, 0.0, 0.0;

.float 0.0, 0.0, 0.0, 0.0, 0.0;
.float 0.0, 0.0, 0.0, 0.0, 0.0;
.float 0.0, 0.0, 0.0, 0.0, 0.0;
.float 0.0, 0.0, 0.0, 0.0, 0.0;

.float 0.0, 0.0, 0.0, 0.0, 0.0;
.float 0.0, 0.0, 0.0, 0.0, 0.0;
.float 0.0, 0.0, 0.0, 0.0, 0.0;
.float 0.0, 0.0, 0.0, 0.0, 0.0;

.float 0.0, 0.0, 0.0, 0.0, 0.0;
.float 0.0, 0.0, 0.0, 0.0, 0.0;
.float 0.0, 0.0, 0.0, 0.0, 0.0;
.float 0.0, 0.0, 0.0, 0.0, 0.0;

//S : off 960
.float 0.0, 0.0, 0.0, 0.0, 0.0;
.float 0.0, 0.0, 0.0, 0.0, 0.0;
.float 0.0, 0.0, 0.0, 0.0, 0.0;
.float 0.0, 0.0, 0.0, 0.0, 0.0;

.float 0.0, 0.0, 0.0, 0.0, 0.0;
.float 0.0, 0.0, 0.0, 0.0, 0.0;
.float 0.0, 0.0, 0.0, 0.0, 0.0;
.float 0.0, 0.0, 0.0, 0.0, 0.0;

.float 0.0, 0.0, 0.0, 0.0, 0.0;
.float 0.0, 0.0, 0.0, 0.0, 0.0;
.float 0.0, 0.0, 0.0, 0.0, 0.0;
.float 0.0, 0.0, 0.0, 0.0, 0.0;

.float 0.0, 0.0, 0.0, 0.0, 0.0;
.float 0.0, 0.0, 0.0, 0.0, 0.0;
.float 0.0, 0.0, 0.0, 0.0, 0.0;
.float 0.0, 0.0, 0.0, 0.0, 0.0;
```

```
95
96
97  //floats of all weights for the hidden state (Wh)
98  weights_floats_hidden:
99  .float 0.4575506, 0.05386263, -0.035822153, -0.1387741;
100 .float 0.2426635, 0.09454799, 0.022639275, -0.40208167;
101 .float -0.47809297, 0.043361187, -0.429963, 0.34989363;
102 .float 0.040919125, 0.052719116, 0.34607226, 0.1788984;
103
104 .float 0.25464368, 0.1096496, -0.4446842, 0.25877202;
105 .float -0.4922228, 0.46996975, -0.27924186, 0.3575132;
106 .float 0.016857564, 0.015506566, -0.16837955, -0.34664214;
107 .float -0.29062146, -0.2048012, 0.18073153, 0.25138396;
108
109 .float 0.29553604, -0.46133327, -0.120453596, 0.36323273;
110 .float -0.29439515, -0.3556685, 0.31927258, 0.21197033;
111 .float -0.22039115, -0.25451106, -0.44844902, 0.05314392;
112 .float 0.289571, -0.40340388, 0.048012376, 0.310561;
113
114 .float 0.37911284, 0.3122326, 0.25405425, -0.21814841;
115 .float 0.11495173, 0.31893253, -0.25266486, 0.33620614;
116 .float 0.0020601153, -0.49355453, -0.3919518, -0.1567089;
117 .float -0.46437746, 0.43377686, -0.2236054, -0.07727009;
118
119 //Gradient off: 256
120 .float 0.0, 0.0, 0.0, 0.0;
121 .float 0.0, 0.0, 0.0, 0.0;
122 .float 0.0, 0.0, 0.0, 0.0;
123 .float 0.0, 0.0, 0.0, 0.0;
124
125 .float 0.0, 0.0, 0.0, 0.0;
126 .float 0.0, 0.0, 0.0, 0.0;
127 .float 0.0, 0.0, 0.0, 0.0;
128 .float 0.0, 0.0, 0.0, 0.0;
129
130 .float 0.0, 0.0, 0.0, 0.0;
131 .float 0.0, 0.0, 0.0, 0.0;
132 .float 0.0, 0.0, 0.0, 0.0;
133 .float 0.0, 0.0, 0.0, 0.0;
134
135 .float 0.0, 0.0, 0.0, 0.0;
136 .float 0.0, 0.0, 0.0, 0.0;
137 .float 0.0, 0.0, 0.0, 0.0;
138 .float 0.0, 0.0, 0.0, 0.0;
139 //P off: 512
140 .float 0.0, 0.0, 0.0, 0.0;
141 .float 0.0, 0.0, 0.0, 0.0;
142 .float 0.0, 0.0, 0.0, 0.0;
143 .float 0.0, 0.0, 0.0, 0.0;
144
145 .float 0.0, 0.0, 0.0, 0.0;
146 .float 0.0, 0.0, 0.0, 0.0;
147 .float 0.0, 0.0, 0.0, 0.0;
148 .float 0.0, 0.0, 0.0, 0.0;
149
150 .float 0.0, 0.0, 0.0, 0.0;
151 .float 0.0, 0.0, 0.0, 0.0;
152 .float 0.0, 0.0, 0.0, 0.0;
153 .float 0.0, 0.0, 0.0, 0.0;
154
155 .float 0.0, 0.0, 0.0, 0.0;
156 .float 0.0, 0.0, 0.0, 0.0;
157 .float 0.0, 0.0, 0.0, 0.0;
158 .float 0.0, 0.0, 0.0, 0.0;
159
160 //S off: 768
161 .float 0.0, 0.0, 0.0, 0.0;
162 .float 0.0, 0.0, 0.0, 0.0;
163 .float 0.0, 0.0, 0.0, 0.0;
164 .float 0.0, 0.0, 0.0, 0.0;
165
```

```
.float 0.0, 0.0, 0.0, 0.0;
.float 0.0, 0.0, 0.0, 0.0;
.float 0.0, 0.0, 0.0, 0.0;
.float 0.0, 0.0, 0.0, 0.0;

.float 0.0, 0.0, 0.0, 0.0;
.float 0.0, 0.0, 0.0, 0.0;
.float 0.0, 0.0, 0.0, 0.0;
.float 0.0, 0.0, 0.0, 0.0;

.float 0.0, 0.0, 0.0, 0.0;
.float 0.0, 0.0, 0.0, 0.0;
.float 0.0, 0.0, 0.0, 0.0;
.float 0.0, 0.0, 0.0, 0.0;

//floats of all weights for the input (bx)
biases_floats_input:
.float 0.45528716, -0.12105113, 0.3534127, -0.4324776;

.float -0.06819719, -0.26856756, 0.26856917, 0.09046912;

.float 0.43491727, -0.4721396, -0.107459486, 0.10790658;

.float 0.23291057, 0.17885023, -0.19004881, 0.4123158;

//Gradient off: 64

.float 0.0, 0.0, 0.0, 0.0;
.float 0.0, 0.0, 0.0, 0.0;
.float 0.0, 0.0, 0.0, 0.0;
.float 0.0, 0.0, 0.0, 0.0;

//P off:128
.float 0.0, 0.0, 0.0, 0.0;
.float 0.0, 0.0, 0.0, 0.0;
.float 0.0, 0.0, 0.0, 0.0;
.float 0.0, 0.0, 0.0, 0.0;
//S off:192
.float 0.0, 0.0, 0.0, 0.0;
.float 0.0, 0.0, 0.0, 0.0;
.float 0.0, 0.0, 0.0, 0.0;
.float 0.0, 0.0, 0.0, 0.0;

//floats of all weights for the hidden state (bh)
biases_floats_hidden:
.float -0.16933638, 0.37731993, -0.48610342, 0.43699688;

.float 0.49017084, -0.3767584, 0.37130326, 0.12813634;

.float -0.0008007884, -0.25514352, 0.1953935, 0.48517787;

.float 0.07333362, 0.4821878, 0.18412137, 0.2220164;

//Gradient, P & S same as biases_floats_input

//first Fully Connected Layer weights and biases
//4 columns (numbers) per row (4x64 values)
fc_1_weight:
.float -0.094638824, 0.2098822, -0.3115564, [...], -0.15274096;
//Gradient
//offset: 1024

.float 0.0, 0.0, 0.0, [...], 0.0;
.float 0.0, 0.0, 0.0, [...], 0.0;
.float 0.0, 0.0, 0.0, [...], 0.0;
.float 0.0, 0.0, 0.0, [...], 0.0;
//P
//offset: 2048
.float 0.0, 0.0, 0.0, [...], 0.0;
.float 0.0, 0.0, 0.0, [...], 0.0;
.float 0.0, 0.0, 0.0, [...], 0.0;
```

```
.float 0.0, 0.0, 0.0, [...], 0.0;
//S
//offset: 3072
.float 0.0, 0.0, 0.0, [...], 0.0;
.float 0.0, 0.0, 0.0, [...], 0.0;
.float 0.0, 0.0, 0.0, [...], 0.0;
.float 0.0, 0.0, 0.0, [...], 0.0;



//1 column per row
fc_1_bias:
.float 0.45598888, 0.028622866, -0.49321836, [...], -0.18397611;
//Gradient
//offset: 256
.float 0.0, 0.0, 0.0, [...], 0.0;
//P
//offset: 512
.float 0.0, 0.0, 0.0, [...], 0.0;
//S
//offset: 768
.float 0.0, 0.0, 0.0, [...], 0.0;


//second Fully Connected Layer weights and bias
fc_weight_and_bias:
//64 weights
.float -0.07416618, -0.099058956, -0.090731055, [...], -0.11582455;
//bias located at address+256 (4*64)
.float 0.025282487;

//Gradient 64 floats
//starts at offset 260
.float 0.0, 0.0, 0.0, [...], 0.0;
//P 64 floats
//starts at offset 516
.float 0.0, 0.0, 0.0, [...], 0.0;
//S 64 floats
//starts at offset 772
.float 0.0, 0.0, 0.0, [...], 0.0;

//Gradientbias
//offset 1028
.float 0.0;
//Pbias
//offset 1032
.float 0.0;
//Sbias
//offset 1036
.float 0.0;




//FC_1 layer values. 64 floats
fc_1_values:
.float 0.0, 0.0, 0.0, [...], 0.0;

fc_1_reluflags: //64 ints = 64 flags
.word 0,0,0,[...], 0;

floats_ht:
.float 0.0, 0.0, 0.0, 0.0;//t=1
.float 0.0, 0.0, 0.0, 0.0;//t=2
.float 0.0, 0.0, 0.0, 0.0;//t=3
.float 0.0, 0.0, 0.0, 0.0;//t=4
.float 0.0, 0.0, 0.0, 0.0;//t=5
.float 0.0, 0.0, 0.0, 0.0;//t=6
.float 0.0, 0.0, 0.0, 0.0;//t=7
.float 0.0, 0.0, 0.0, 0.0;//t=8
```

```
.float 0.0, 0.0, 0.0, 0.0;//t=9
.float 0.0, 0.0, 0.0, 0.0;//t=10

//160 bits offset, save ReLu flags here
.word 0,0,0,0 //

floats_ct:
.float 0.0, 0.0, 0.0, 0.0;//t=1
.float 0.0, 0.0, 0.0, 0.0;//t=2
.float 0.0, 0.0, 0.0, 0.0;//t=3
.float 0.0, 0.0, 0.0, 0.0;//t=4
.float 0.0, 0.0, 0.0, 0.0;//t=5
.float 0.0, 0.0, 0.0, 0.0;//t=6
.float 0.0, 0.0, 0.0, 0.0;//t=7
.float 0.0, 0.0, 0.0, 0.0;//t=8
.float 0.0, 0.0, 0.0, 0.0;//t=9
.float 0.0, 0.0, 0.0, 0.0;//t=10

floats_tanhct:
.float 0.0, 0.0, 0.0, 0.0;//t=1 off 0
.float 0.0, 0.0, 0.0, 0.0;//t=2 off 16
.float 0.0, 0.0, 0.0, 0.0;//t=3 off 32
.float 0.0, 0.0, 0.0, 0.0;//t=4 off 48
.float 0.0, 0.0, 0.0, 0.0;//t=5 off 64
.float 0.0, 0.0, 0.0, 0.0;//t=6 off 80
.float 0.0, 0.0, 0.0, 0.0;//t=7 off 96
.float 0.0, 0.0, 0.0, 0.0;//t=8 off 112
.float 0.0, 0.0, 0.0, 0.0;//t=9 off 128
.float 0.0, 0.0, 0.0, 0.0;//t=10 off 144

floats_ifgo:
//i
.float 0.0, 0.0, 0.0, 0.0;//t=1
.float 0.0, 0.0, 0.0, 0.0;//t=2
.float 0.0, 0.0, 0.0, 0.0;//t=3
.float 0.0, 0.0, 0.0, 0.0;//t=4
.float 0.0, 0.0, 0.0, 0.0;//t=5
.float 0.0, 0.0, 0.0, 0.0;//t=6
.float 0.0, 0.0, 0.0, 0.0;//t=7
.float 0.0, 0.0, 0.0, 0.0;//t=8
.float 0.0, 0.0, 0.0, 0.0;//t=9
.float 0.0, 0.0, 0.0, 0.0;//t=10
//f
.float 0.0, 0.0, 0.0, 0.0;//t=1
.float 0.0, 0.0, 0.0, 0.0;//t=2
.float 0.0, 0.0, 0.0, 0.0;//t=3
.float 0.0, 0.0, 0.0, 0.0;//t=4
.float 0.0, 0.0, 0.0, 0.0;//t=5
.float 0.0, 0.0, 0.0, 0.0;//t=6
.float 0.0, 0.0, 0.0, 0.0;//t=7
.float 0.0, 0.0, 0.0, 0.0;//t=8
.float 0.0, 0.0, 0.0, 0.0;//t=9
.float 0.0, 0.0, 0.0, 0.0;//t=10
//g
.float 0.0, 0.0, 0.0, 0.0;//t=1
.float 0.0, 0.0, 0.0, 0.0;//t=2
.float 0.0, 0.0, 0.0, 0.0;//t=3
.float 0.0, 0.0, 0.0, 0.0;//t=4
.float 0.0, 0.0, 0.0, 0.0;//t=5
.float 0.0, 0.0, 0.0, 0.0;//t=6
.float 0.0, 0.0, 0.0, 0.0;//t=7
.float 0.0, 0.0, 0.0, 0.0;//t=8
.float 0.0, 0.0, 0.0, 0.0;//t=9
.float 0.0, 0.0, 0.0, 0.0;//t=10
//o
.float 0.0, 0.0, 0.0, 0.0;//t=1
.float 0.0, 0.0, 0.0, 0.0;//t=2
.float 0.0, 0.0, 0.0, 0.0;//t=3
.float 0.0, 0.0, 0.0, 0.0;//t=4
.float 0.0, 0.0, 0.0, 0.0;//t=5
.float 0.0, 0.0, 0.0, 0.0;//t=6
```

```
.float 0.0, 0.0, 0.0, 0.0;//t=7
.float 0.0, 0.0, 0.0, 0.0;//t=8
.float 0.0, 0.0, 0.0, 0.0;//t=9
.float 0.0, 0.0, 0.0, 0.0;//t=10



float_final_output:
.float 0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0, [...];


//numer one, used in fs11
float_one:
    .float 1.0;

//numbers used in tanh
tanh_floats:
    .float 2.0, 0.25;

//numbers used in sigmoid
sig_floats:
    .float 4.0, 0.5, 0.25, 0.03125;

betas_and_others:
    //beta1 & beta2
    .float 0.9, 0.999;

    //cumulative beta^(t)
    .float 1.0, 1.0;

    //learning rate
    .float 0.005;

    //numbers to divide by
    .float 3342.0, 10.0;

    //epsilon
    .word 0x00800000




.section .text

.global main

//register use:
/*
s0      -> Input data pointer
s1 - s4 -> Weights and biases pointers
s5      -> ht (4x10 floats) & ht_reluflags (4 ints) pointer
s6      -> ct pointer
s7      -> pointer for tanh(ct) results
s8      -> pointer to store i, f, g and o
s9      -> FC_1 ReLu flags
s10     -> Output data pointer
s11     -> FeedForward loop counter

a0      -> jal/ret adress
a1 - a2 -> FC_1 parameters address
a3      -> FC_1 results
a4      -> FC parameters adress
a5      -> Results/Outputs Pointer
a6      -> LSTM [i] offset counter
a7      -> LSTM time step counter

fs0 - fs8 -> used
fs9       -> FREE
```

```
fs10       -> dJ/dy^
fs11       -> = 1.0

fa0 - fa4 -> used
fs5        -> FREE
fs6 - fa7 -> tanh/sig


*/
#run_code:
main:
    //epoch counter. Do everthing 220 times, then exit
    li t6, 220
    full_epoch_restart:
    //pointer to the output of the network (a5)
    la a5, float_final_output

    //load float 1.0 into register fs11 for future use
    la t0, float_one;
    flw fs11, 0(t0); //fs11 = 1.0

    // a0 function return adress

    //input data pointer
    la s0, InputXData

    //output data pointer
    la s10, OutputYData

    //** CHANGE: Save ALL relu flags with the same offset as a3
    la s9, fc_1_reluflags

    //adresses for fc weights and bias
    la a4, fc_weight_and_bias


    //Feed Forward counter
    //loop forward_pass_full_restart 3797 times
    //** CHANGED TO 3342 FOR TRAINING ONLY
    //exits on s11==0
    //li s11, 3797
    li s11, 3342
/*
    //ht & ct
    la s5, floats_ht
    la s6, floats_ct

    //** CHANGE: new adress: s7 -> stores values for tanh(ct)
    la s7, floats_tanhct

    //** CHANGE: new adress: s8 -> stores values for i, f, g and o
    // all of them on the same address, with 160 bits appart (each is 4 values, 10 times)
    la s8, floats_ifgo*/

    //Set weights and biases pointers
    la s1, weights_floats_input
    la s2, weights_floats_hidden
    la s3, biases_floats_input
    la s4, biases_floats_hidden

    forward_pass_full_restart:

    //ht & ct
    la s5, floats_ht
    la s6, floats_ct

    //** CHANGE: new adress: s7 -> stores values for tanh(ct)
    la s7, floats_tanhct

    //** CHANGE: new adress: s8 -> stores values for i, f, g and o
    // all of them on the same address, with 160 bits appart (each is 4 values, 10 times)
```

```
    la s8, floats_ifgo


    //reset ct with zeros
    //** CHANGE: dont
    //fmv.s.x ft0, x0

    //ct
    //fsw ft0, 0(s6)
    //fsw ft0, 4(s6)
    //fsw ft0, 8(s6)
    //fsw ft0, 12(s6)


    //timestep counter
    //a7==0 indicates t=0 (ht = 0 & ct=0)
    li a7, 0



    forward_pass_loop:
    //forward pass for a single time step
    //ht is not needed the first pass


    //reset offset/counter
    //a6 counts [i] loops (from 4 to 0)
    li a6, 4
/*
    //Reset weights and biases pointers
    la s1, weights_floats_input
    la s2, weights_floats_hidden
    la s3, biases_floats_input
    la s4, biases_floats_hidden
*/

    //Load x into float registers
    //x  = [fs4 ... fs8]
    flw fs4, 0(s0)
    flw fs5, 4(s0)
    flw fs6, 8(s0)
    flw fs7, 12(s0)
    flw fs8, 16(s0)



    forward_pass_matrix_mul:
    //loops while a6 > 0

    //---- fa1 -> i

    //matrix multiplication for i, f, g & o
    //ht vector stored as = [fs0 ... fs3]
    //x vector stored in x  = [fs4 ... fs8]
    //Weights and biases must be the following (with proper offsets)
    // s1, weights_floats_input
    // s2, weights_floats_hidden
    // s3, biases_floats_input
    // s4, biases_floats_hidden
    //if t=0 flag a7 is 0, else a7=1
    //(skips Wh*h multiplications as h=0 when t=0)
    //result is saved at fa6 to be used in tanh or sig functions

    //Wxi * x
    flw ft0, 0(s1)
    fmul.s fa6, ft0, fs4

    flw ft0, 4(s1)
    fmadd.s fa6, ft0, fs5 ,fa6

    flw ft0, 8(s1)
```

```
    fmadd.s fa6, ft0, fs6 ,fa6

    flw ft0, 12(s1)
    fmadd.s fa6, ft0, fs7 ,fa6

    flw ft0, 16(s1)
    fmadd.s fa6, ft0, fs8 ,fa6

    //ans + bx
    flw ft0, 0(s3)
    fadd.s fa6, ft0, fa6

    //t=0? (a7=0?) if so, skip Wh*h (h is 0)
    beqz a7, mat_mul_skip_h_i

    //ans + (Wh * h)
    flw ft0, 0(s2)
    fmadd.s fa6, ft0, fs0, fa6

    flw ft0, 4(s2)
    fmadd.s fa6, ft0, fs1, fa6

    flw ft0, 8(s2)
    fmadd.s fa6, ft0, fs2, fa6

    flw ft0, 12(s2)
    fmadd.s fa6, ft0, fs3, fa6

    mat_mul_skip_h_i:
    //ans + bh
    flw ft0, 0(s4)
    fadd.s fa6, ft0, fa6

    jal a0, sig_fa6 //apply sigmoid
    fmv.s fa1, fa7

    //a7=0? if so, skip f
    //** CHANGE: no longer skip f for t=0 (needed for gradient)
    //beqz a7, skip_f



    //---- fa2 -> f
    //Wxi * x
    flw ft0, 80(s1)
    fmul.s fa6, ft0, fs4

    flw ft0, 84(s1)
    fmadd.s fa6, ft0, fs5 ,fa6

    flw ft0, 88(s1)
    fmadd.s fa6, ft0, fs6 ,fa6

    flw ft0, 92(s1)
    fmadd.s fa6, ft0, fs7 ,fa6

    flw ft0, 96(s1)
    fmadd.s fa6, ft0, fs8 ,fa6

    //ans + bx
    flw ft0, 16(s3)
    fadd.s fa6, ft0, fa6

    //t=0? (a7=0?) if so, skip Wh*h (h is 0)
    beqz a7, mat_mul_skip_h_f

    //ans + (Wh * h)
    flw ft0, 64(s2)
    fmadd.s fa6, ft0, fs0, fa6

    flw ft0, 68(s2)
```

```
663     fmadd.s fa6, ft0, fs1, fa6
664
665     flw ft0, 72(s2)
666     fmadd.s fa6, ft0, fs2, fa6
667
668     flw ft0, 76(s2)
669     fmadd.s fa6, ft0, fs3, fa6
670
671     mat_mul_skip_h_f:
672     //ans + bh
673     flw ft0, 16(s4)
674     fadd.s fa6, ft0, fa6
675
676
677     jal a0, sig_fa6 //apply sigmoid
678     fmv.s fa2, fa7
679
680     //** CHANGE: f is no longer skipped at t=0
681     //skip_f:
682     //---- fa3 -> g
683     //Wxi * x
684     flw ft0, 160(s1)
685     fmul.s fa6, ft0, fs4
686
687     flw ft0, 164(s1)
688     fmadd.s fa6, ft0, fs5 ,fa6
689
690     flw ft0, 168(s1)
691     fmadd.s fa6, ft0, fs6 ,fa6
692
693     flw ft0, 172(s1)
694     fmadd.s fa6, ft0, fs7 ,fa6
695
696     flw ft0, 176(s1)
697     fmadd.s fa6, ft0, fs8 ,fa6
698
699     //ans + bx
700     flw ft0, 32(s3)
701     fadd.s fa6, ft0, fa6
702
703     //t=0? (a7=0?) if so, skip Wh*h (h is 0)
704     beqz a7, mat_mul_skip_h_g
705
706     //ans + (Wh * h)
707     flw ft0, 128(s2)
708     fmadd.s fa6, ft0, fs0, fa6
709
710     flw ft0, 132(s2)
711     fmadd.s fa6, ft0, fs1, fa6
712
713     flw ft0, 136(s2)
714     fmadd.s fa6, ft0, fs2, fa6
715
716     flw ft0, 140(s2)
717     fmadd.s fa6, ft0, fs3, fa6
718
719     mat_mul_skip_h_g:
720     //ans + bh
721     flw ft0, 32(s4)
722     fadd.s fa6, ft0, fa6
723
724     jal a0, tanh_fa6 //apply tanh
725     fmv.s fa3, fa7
726
727
728
729     //---- fa4 -> o
730     //Wxi * x
731     flw ft0, 240(s1)
732     fmul.s fa6, ft0, fs4
733
```

```
    flw ft0, 244(s1)
    fmadd.s fa6, ft0, fs5 ,fa6

    flw ft0, 248(s1)
    fmadd.s fa6, ft0, fs6 ,fa6

    flw ft0, 252(s1)
    fmadd.s fa6, ft0, fs7 ,fa6

    flw ft0, 256(s1)
    fmadd.s fa6, ft0, fs8 ,fa6

    //ans + bx
    flw ft0, 48(s3)
    fadd.s fa6, ft0, fa6

    //t=0? (a7=0?) if so, skip Wh*h (h is 0)
    beqz a7, mat_mul_skip_h_o

    //ans + (Wh * h)
    flw ft0, 192(s2)
    fmadd.s fa6, ft0, fs0, fa6

    flw ft0, 196(s2)
    fmadd.s fa6, ft0, fs1, fa6

    flw ft0, 200(s2)
    fmadd.s fa6, ft0, fs2, fa6

    flw ft0, 204(s2)
    fmadd.s fa6, ft0, fs3, fa6

    mat_mul_skip_h_o:
    //ans + bh
    flw ft0, 48(s4)
    fadd.s fa6, ft0, fa6


    jal a0, sig_fa6 //apply sigmoid
    fmv.s fa4, fa7

    //fa1 -> i
    //fa2 -> f
    //fa3 -> g
    //fa4 -> o
    //** CHANGE: values are now stored in memory, at s8, each having an offset of 160
    // index and time offsets are added separately
    fsw fa1, 0(s8)
    fsw fa2, 160(s8)
    fsw fa3, 320(s8)
    fsw fa4, 480(s8)

    //calculate ct = f * ct-1 + i * g
    // ct = fa2 * 0(s6) + fa1 * fa3

    //i * g
    fmul.s fa6, fa1, fa3

    // if a7=0 -> ct-1 = 0 -> ct = fa1 * fa3
    beqz a7, skip_ct1_mul

    //else: ct-1 * f + ans
    //** CHANGE: ct-1 is now s6-16
    flw ft1, -16(s6)
    fmadd.s fa6, ft1, fa2, fa6

    skip_ct1_mul:
    //store ct
    fsw fa6, 0(s6)

    //ht+t = o * tanh(ct)
```

```
    jal a0, tanh_fa6
    //** CHANGE: store tanh(ct) (fa7) into s7
    fsw fa7, 0(s7)
    //fa7 = tanh(ct) * o
    fmul.s fa7, fa7, fa4
    //store ht+1
    fsw fa7, 0(s5)


    //---- repeat for [i+1] ----
    addi a6, a6, -1
    beqz a6, end_index_loop

    //reset and offset wieghts to the next row
    addi s1, s1, 20
    addi s2, s2, 16
    //reset and offset biases to the next column
    addi s3, s3, 4
    addi s4, s4, 4

    //offset ht+1 and ct to next number
    addi s5, s5, 4
    addi s6, s6, 4

    //** CHANGE: offset s7 and s8 as well (tanhct results and i,f,g,o)
    addi s7, s7, 4
    addi s8, s8, 4

    j forward_pass_matrix_mul


    end_index_loop:

    //Reset weights and biases pointers
    la s1, weights_floats_input
    la s2, weights_floats_hidden
    la s3, biases_floats_input
    la s4, biases_floats_hidden

    //time step finished, move to next time step
    //unless already done with 10 time steps -> check a7
    addi a7, a7, 1
    slti t0, a7, 10 //if a7 < 10 -> t0 = 1 -> not done yet
    beqz t0, end_forward_pass_loop //t0 = 0 -> done with time steps, move on to FC

    //else: start over with XData offset of 5 numbers (+20)
    addi s0, s0, 20


    //remove ht & ct offset
    //** CHANGE: dont remove ct offset
    addi s5, s5, -12
    //addi s6, s6, -12

    //** CHANGE: add an extra 4 for next row to ct
    addi s6, s6, 4

    //** CHANGE: add 4 to s7 & s8 as well
    addi s7, s7, 4
    addi s8, s8, 4

    //Load/Reload ht into float registers
    //ht = [fs0 ... fs3]
    flw fs0, 0(s5)
    flw fs1, 4(s5)
    flw fs2, 8(s5)
    flw fs3, 12(s5)


    //and offset ht+1 to the next timestep "set"
    addi s5, s5, 16 //(4 floats)
```

```
876
877
878     j forward_pass_loop //repeat one time step
879
880     end_forward_pass_loop:
881     //done with data on the LSTM
882     //do Fully Connected next
883
884     //remove ht offset and load floats
885     addi s5, s5, -12
886     //ht10 = [fs0 ... fs3]
887     flw fs0, 0(s5)
888     flw fs1, 4(s5)
889     flw fs2, 8(s5)
890     flw fs3, 12(s5)
891
892
893     //----- apply ReLu flag -----
894     //relu flag is used to skip the multiplication, instead of setting the number to zero (
            same result)
895     //flag is 1 if number is =< 0 -> should skip multiplication
896     fmv.s.x ft3, x0 //set zero
897
898     //flags:
899     //fs0 -> t0
900     //fs2 -> t1
901     //fs2 -> t2
902     //fs3 -> t3
903     fle.s t0, fs0, ft3
904     fle.s t1, fs1, ft3
905     fle.s t2, fs2, ft3
906     fle.s t3, fs3, ft3
907
908     //** CHANGE:save ReLu flags on s5+16 - s5+28
909     sw t0, 16(s5)
910     sw t1, 20(s5)
911     sw t2, 24(s5)
912     sw t3, 28(s5)
913
914
915     //addresses for fc_1 weights, biases and results
916     la a1, fc_1_weight
917     la a2, fc_1_bias
918     la a3, fc_1_values
919
920     //** CHANGE: Save ALL relu flags with the same offset as a3
921     //done at the start of the program
922     //la s9, fc_1_reluflags
923
924     //** CHANGE: adress loaded at the start of the program
925     //adresses for fc weights and bias
926     //la a4, fc_weight_and_bias
927
928     //fa0 -> final output
929     flw fa0, 256(a4)
930
931     //Following code loop is unraveled to save 4*64 instructions
932     //it feeds ht into each fc_1 neuron, which then feeds into fc
933
934     //-------------------------------------------------------------
935     //------    "LOOP" START   -----------------------------------
936     //-------------------------------------------------------------
937
938     //0
939     flw ft0, 0(a2)
940
941     bnez t0, skip_0_0
942         flw ft1, 0(a1)
943         fmadd.s ft0, ft1, fs0, ft0
944     skip_0_0:
945
```

```
    bnez t1, skip_1_0
        flw ft1, 4(a1)
        fmadd.s ft0, ft1, fs1, ft0
    skip_1_0:

    bnez t2, skip_2_0
        flw ft1, 8(a1)
        fmadd.s ft0, ft1, fs2, ft0
    skip_2_0:

    bnez t3, skip_3_0
        flw ft1, 12(a1)
        fmadd.s ft0, ft1, fs3, ft0
    skip_3_0:

    fsw ft0, 0(a3)

    //relu, then feed to fc
    fle.s t4, ft0, ft3//relu flag = (0 if ft0 > 0) -> do mul; else is 1 -> skip mul
    //** CHANGE: save relu flag into s9 with same offset as previous a3
    sw t4, 0(s9)
    bnez t4, skip_fc_0//if ft0<=0 then skip fmadd.s
        flw ft1, 0(a4)
        fmadd.s fa0, ft0, ft1, fa0
    skip_fc_0:

    //1
    flw ft0, 4(a2)
    bnez t0, skip_0_1
        flw ft1, 16(a1)
        fmadd.s ft0, ft1, fs0, ft0
    skip_0_1:
    bnez t1, skip_1_1
        flw ft1, 20(a1)
        fmadd.s ft0, ft1, fs1, ft0
    skip_1_1:
    bnez t2, skip_2_1
        flw ft1, 24(a1)
        fmadd.s ft0, ft1, fs2, ft0
    skip_2_1:
    bnez t3, skip_3_1
        flw ft1, 28(a1)
        fmadd.s ft0, ft1, fs3, ft0
    skip_3_1:
    fsw ft0, 4(a3)
    fle.s t4, ft0, ft3
    sw t4, 4(s9)
    bnez t4, skip_fc_1
        flw ft1, 4(a4)
        fmadd.s fa0, ft0, ft1, fa0
    skip_fc_1:

    //----------------------------------------
    //THIS CODE REPEATS A TOTAL OF 64 TIMES
    //REMOVED DUPLICATES FOR READABILITY
    //----------------------------------------

    //63
    flw ft0, 252(a2)
    bnez t0, skip_0_63
        flw ft1, 1008(a1)
        fmadd.s ft0, ft1, fs0, ft0
    skip_0_63:
    bnez t1, skip_1_63
        flw ft1, 1012(a1)
        fmadd.s ft0, ft1, fs1, ft0
    skip_1_63:
    bnez t2, skip_2_63
        flw ft1, 1016(a1)
        fmadd.s ft0, ft1, fs2, ft0
    skip_2_63:
```

```
    bnez t3, skip_3_63
        flw ft1, 1020(a1)
        fmadd.s ft0, ft1, fs3, ft0
    skip_3_63:
    fsw ft0, 252(a3)
    fle.s t4, ft0, ft3
    sw t4, 252(s9)
    bnez t4, skip_fc_63
        flw ft1, 252(a4)
        fmadd.s fa0, ft0, ft1, fa0
    skip_fc_63:



    //-----------------------------------------------------------------
    //------     "LOOP" END     ---------------------------------------
    //-----------------------------------------------------------------

    fsw fa0, 0(a5)
    //j skip_to_adam_for_test

    //-----------------------------------------------------------------
    //------   BACKPROPAGATION  ---------------------------------------
    //-----------------------------------------------------------------

    //Important Variables so far:
    //fs11 = 1.0
    //h10 = [fs0, ..., fs3]
    //h10 relu flags = [t0, ..., t3]
    //a1, fc_1_weight
    //a2, fc_1_bias
    //a3, fc_1_values
    //s9, fc_1_reluflags
    //a4, fc_weight_and_bias
    //ReLu flags are 1 if value was less than 0, and are 0 otherwise



    //expected result -> ft0
    //predicted value -> fa0
    flw ft0, 0(s10)

    //offset s10 for next one
    addi s10, s10, 4
    //dL/dFCout = 2*(predicted - expected)
    //saved in fs10
    fsub.s fs10, fa0, ft0
    fadd.s fs10, fs10, fs10

    //dL/dbout = dL/dFCout
    flw ft0, 1028(a4)
    fadd.s ft0, ft0, fs10
    fsw ft0, 1028(a4)

    //dL/dh10 = [fs4, ..., fs7]
    //set to x0 to start
    fmv.s.x fs4, x0
    fmv.s.x fs5, x0
    fmv.s.x fs6, x0
    fmv.s.x fs7, x0

    //in order to avoid as many flw and fsw as possible, as well as skipping
    //multiplications by 0, the following is done:

    //for every i flag in vector fc_1_reluflags (64 in total), check flag
    //if flag is 1, skip everything & move to the next [i]
    //if flag is 0, use fc_1_values in position [i]
    //  do dL/dWout [i] += fc_1_values [i] * fs10
    //  get dL/dfc1 [i] = fc_weight_and_bias[i] * fs10
    //  dL/db1[i] += dL/dfc1 [i]
    //  for every [k] in [t0, ..., t3] (h10 relu flags), check flag. if 1-> skip k
```

```
1088        //      do dL/dW1 [i][k] += dL/dfc1 [i] * h10[k]
1089        //      get dL/dh10 [k] += dL/dfc1 [i] * fc_1_weight[i][k] -> save into [fs4, ..., fs7]
1090
1091        //for vectors, each [i] is an extra offset of 4
1092        //for matrices, each cloumn [k] is +4, each row[i] is +rowLength*4 (+(4*4)=+16 in this
              case)
1093        //this is once again loop unrolled to avoid addi instructions that would mess with the
1094        //adresses
1095
1096        //Loop unrolling:this is repeated 64 times in total, for i = [0, ..., 63]
1097
1098
1099        //-----------------------------------------------------------------
1100        //------    "LOOP" START   ---------------------------------------
1101        //-----------------------------------------------------------------
1102
1103        //i=0
1104        //fc1 relu flag
1105        lw t4, 0(s9)
1106        //if 1-> skip all
1107        bnez t4, skip_backprop_i0
1108            //else use fc_1_values in position [i] -> ft0 = fc_1[i]
1109            flw ft0, 0(a3)
1110
1111            //dL/dWout [i] += fc_1_values [i] * fs10
1112            //gradient offset is 260 from fc_weight_and_bias
1113            flw ft1, 260(a4)
1114            fmadd.s ft1, ft0, fs10, ft1
1115            fsw ft1, 260(a4)
1116
1117            //  get dL/dfc1 [i] = fc_weight_and_bias[i] * fs10 -> ft1 = dL/dfc1[i]
1118            flw ft1, 0(a4)
1119            fmul.s ft1, ft1, fs10
1120            //  dL/db1[i] += dL/dfc1 [i]
1121            flw ft2, 256(a2)
1122            fadd.s ft2, ft2, ft1
1123            fsw ft2, 256(a2)
1124
1125            //h10 = [fs0, ..., fs3]
1126            //h10 relu flags = [t0, ..., t3]
1127            //  for every [k] in [t0, ..., t3] (h10 relu flags), check flag. if 1-> skip k
1128
1129            //k=0
1130            bnez t0, skip_backprop_k00
1131                //dL/dW1 [i][k] += dL/dfc1 [i] * h10[k]
1132                //W1 = fc_1_weights; gradients have a base offset of 1024
1133                flw ft2, 1024(a1)//=dL/dW1[i][k]
1134                fmadd.s ft2, ft1, fs0, ft2
1135                fsw ft2, 1024(a1)
1136
1137                //dL/dh10 [k] += dL/dfc1[i] * fc_1_weight[i][k] -> dL/dh10[0] = fs4
1138                flw ft2, 0(a1)//=fc_1_weight[i][k]
1139                fmadd.s fs4, ft1, ft2, fs4
1140
1141            skip_backprop_k00:
1142
1143            //k=1
1144            bnez t1, skip_backprop_k01
1145                //dL/dW1 [i][k] += dL/dfc1 [i] * h10[k]
1146                flw ft2, 1028(a1)//=dL/dW1[i][k]
1147                fmadd.s ft2, ft1, fs1, ft2
1148                fsw ft2, 1028(a1)
1149
1150                //dL/dh10 [k] += dL/dfc1[i] * fc_1_weight[i][k] -> dL/dh10[1] = fs5
1151                flw ft2, 4(a1)//=fc_1_weight[i][k]
1152                fmadd.s fs5, ft1, ft2, fs5
1153
1154            skip_backprop_k01:
1155
1156            //k=2
1157            bnez t2, skip_backprop_k02
```

```
            flw ft2, 1032(a1)//=dL/dW1[i][k]
            fmadd.s ft2, ft1, fs2, ft2
            fsw ft2, 1032(a1)

            //dL/dh10 [k] += dL/dfc1[i] * fc_1_weight[i][k] -> dL/dh10[1] = fs5
            flw ft2, 8(a1)//=fc_1_weight[i][k]
            fmadd.s fs6, ft1, ft2, fs6
        skip_backprop_k02:

        //k=3
        bnez t3, skip_backprop_k03
            flw ft2, 1036(a1)//=dL/dW1[i][k]
            fmadd.s ft2, ft1, fs3, ft2
            fsw ft2, 1036(a1)

            //dL/dh10 [k] += dL/dfc1[i] * fc_1_weight[i][k] -> dL/dh10[1] = fs5
            flw ft2, 12(a1)//=fc_1_weight[i][k]
            fmadd.s fs7, ft1, ft2, fs7
        skip_backprop_k03:
    skip_backprop_i0:

    //---------------------------------------
    //THIS CODE REPEATS A TOTAL OF 64 TIMES
    //REMOVED DUPLICATES FOR READABILITY
    //---------------------------------------

    //i=63
    lw t4, 252(s9)
    bnez t4, skip_backprop_i63
        flw ft0, 252(a3)

        flw ft1, 512(a4)
        fmadd.s ft1, ft0, fs10, ft1
        fsw ft1, 512(a4)

        flw ft1, 252(a4)
        fmul.s ft1, ft1, fs10
        flw ft2, 508(a2)
        fadd.s ft2, ft2, ft1
        fsw ft2, 508(a2)

        //k=0
        bnez t0, skip_backprop_k630
            flw ft2, 2032(a1)
            fmadd.s ft2, ft1, fs0, ft2
            fsw ft2, 2032(a1)

            flw ft2, 1008(a1)
            fmadd.s fs4, ft1, ft2, fs4
        skip_backprop_k630:

        //k=1
        bnez t1, skip_backprop_k631
            flw ft2, 2036(a1)
            fmadd.s ft2, ft1, fs1, ft2
            fsw ft2, 2036(a1)

            flw ft2, 1012(a1)
            fmadd.s fs5, ft1, ft2, fs5
        skip_backprop_k631:

        //k=2
        bnez t2, skip_backprop_k632
            flw ft2, 2040(a1)
            fmadd.s ft2, ft1, fs2, ft2
            fsw ft2, 2040(a1)

            flw ft2, 1016(a1)
            fmadd.s fs6, ft1, ft2, fs6
        skip_backprop_k632:
```

```
        //k=3
        bnez t3, skip_backprop_k633
            flw ft2, 2044(a1)
            fmadd.s ft2, ft1, fs3, ft2
            fsw ft2, 2044(a1)

            flw ft2, 1020(a1)
            fmadd.s fs7, ft1, ft2, fs7
        skip_backprop_k633:
    skip_backprop_i63:


    //---------------------------------------------------------------
    //------     "LOOP" END     -------------------------------------
    //---------------------------------------------------------------

    //Reminder
    //fs11 = 1.0
    //h10 = [fs0, ..., fs3]
    //dL/dh10 = [fs4, ..., fs7]
    //s5 = floats_ht
    //s6 = floats_ct
    //s7 = floats_tanhct
    //0(s8)   = it
    //160(s8) = ft
    //320(s8) = gt
    //480(s8) = ot
    //s1 = weights_floats_input
    //s2 = weights_floats_hidden
    //s3 = biases_floats_input
    //s4 = biases_floats_hidden

    //ht & ct
    la s5, floats_ht
    la s6, floats_ct

    //reset s7 (tanh(ct))
    la s7, floats_tanhct

    // reset i, f, g and o adress
    // all of them on the same address, with 160 bits appart (each is 4 values, 10 times)
    la s8, floats_ifgo

    //offset so all are at t=10 -> offset by 144
    addi s5, s5, 144 //ht
    addi s6, s6, 144 //ct
    addi s7, s7, 144 //tanhct
    addi s8, s8, 144 //ifgo

    li a7, 10 //time step

    begin_backprop_lstm:
    //Backprop at LSTM
    //for Vt={i,f,g,o}: Vt = f(a_vt) ;f() respective trig. function
    //---- first get dL/dCt and dL/daot

    flw ft4, 0(s7)//tanh(C10) -> ft4 - ft7
    flw ft5, 4(s7)
    flw ft6, 8(s7)
    flw ft7, 12(s7)


    flw ft8, 480(s8)//ot -> ft8 - ft11
    flw ft9, 484(s8)
    flw ft10, 488(s8)
    flw ft11, 492(s8)


    //dL/dCt = dL/dht * (ot - ht*tanh(Ct)) -> [ft0, ..., ft3]
    fnmsub.s ft0, fs0, ft4, ft8 //-(ht*tanh(Ct)) + ot
    fmul.s ft0, ft0, fs4 //ans * dL/dht
```

```
1300
1301      fnmsub.s ft1, fs1, ft5, ft9
1302      fmul.s ft1, ft1, fs5
1303
1304      fnmsub.s ft2, fs2, ft6, ft10
1305      fmul.s ft2, ft2, fs6
1306
1307      fnmsub.s ft3, fs3, ft7, ft11
1308      fmul.s ft3, ft3, fs7
1309
1310      //dL/daot = dL/dht * (ht - ht*ot) -> [fa0, ..., fa3]
1311      fnmsub.s fa0, ft8, fs0, fs0 //-(ot*ht) + ht
1312      fmul.s fa0, fa0, fs4 //ans * dL/dht
1313
1314      fnmsub.s fa1, ft9, fs1, fs1
1315      fmul.s fa1, fa1, fs5
1316
1317      fnmsub.s fa2, ft10, fs2, fs2
1318      fmul.s fa2, fa2, fs6
1319
1320      fnmsub.s fa3, ft11, fs3, fs3
1321      fmul.s fa3, fa3, fs7
1322
1323      //---- now get dL/dait and dL/dagt
1324
1325      flw ft4, 0(s8)//it
1326      flw ft5, 4(s8)
1327      flw ft6, 8(s8)
1328      flw ft7, 12(s8)
1329
1330      flw ft8, 320(s8)//gt
1331      flw ft9, 324(s8)
1332      flw ft10, 328(s8)
1333      flw ft11, 332(s8)
1334
1335      //dL/dait = dL/dCt * gt * (it - it*it) -> [fa4, ..., fa7]
1336      fnmsub.s fa4, ft4, ft4, ft4//-(it * it) + it
1337      fmul.s fa4, fa4, ft8//ans * gt
1338      fmul.s fa4, fa4, ft0//ans * dL/dCt
1339
1340      fnmsub.s fa5, ft5, ft5, ft5
1341      fmul.s fa5, fa5, ft9
1342      fmul.s fa5, fa5, ft1
1343
1344      fnmsub.s fa6, ft6, ft6, ft6
1345      fmul.s fa6, fa6, ft10
1346      fmul.s fa6, fa6, ft2
1347
1348      fnmsub.s fa7, ft7, ft7, ft7
1349      fmul.s fa7, fa7, ft11
1350      fmul.s fa7, fa7, ft3
1351
1352      //dL/dagt = dL/dCt * it * (1.0 - gt*gt) -> [ft4, ..., fs7]
1353      fnmsub.s fs0, ft8, ft8, fs11//-(gt*gt)+1.0
1354      fmul.s ft4, fs0, ft4//ans * it
1355      fmul.s ft4, ft4, ft0//ans * dL/dCt
1356
1357      fnmsub.s fs1, ft9, ft9, fs11
1358      fmul.s ft5, fs1, ft5
1359      fmul.s ft5, ft5, ft1
1360
1361      fnmsub.s fs2, ft10, ft10, fs11
1362      fmul.s ft6, fs2, ft6
1363      fmul.s ft6, ft6, ft2
1364
1365      fnmsub.s fs3, ft11, ft11, fs11
1366      fmul.s ft7, fs3, ft7
1367      fmul.s ft7, ft7, ft3
1368
1369      //---- get dL/daft
```

```
    //substract 1 to time counter now. If its 0, Ct-1 = C0 = 0 -> skip Ct-1 related
        instructions
    addi a7, a7, -1

    flw ft8, 160(s8) //ft
    flw ft9, 164(s8)
    flw ft10, 168(s8)
    flw ft11, 172(s8)

    //dL/daft = dL/dCt * Ct-1 * (ft - ft*ft) -> [ft0, ..., ft3]
    fnmsub.s fs0, ft8, ft8, ft8 //-(ft*ft) + ft
    fmul.s ft0, fs0, ft0//ans*dL/dCt
    beqz a7, skip_daft_0//if a7!=0 -> ans=ans*Ct-1
        flw ft8, -16(s6)
        fmul.s ft0, ft0, ft8
    skip_daft_0:

    fnmsub.s fs1, ft9, ft9, ft9
    fmul.s ft1, fs1, ft1
    beqz a7, skip_daft_1
        flw ft9, -12(s6)
        fmul.s ft1, ft1, ft9
    skip_daft_1:

    fnmsub.s fs2, ft10, ft10, ft10
    fmul.s ft2, fs2, ft2
    beqz a7, skip_daft_2
        flw ft10, -8(s6)
        fmul.s ft2, ft2, ft10
    skip_daft_2:

    fnmsub.s fs3, ft11, ft11, ft11
    fmul.s ft3, fs3, ft3
    beqz a7, skip_daft_3
        flw ft11, -4(s6)
        fmul.s ft3, ft3, ft11
    skip_daft_3:


    //dL/daot -> [fa0, ..., fa3]
    //dL/dait -> [fa4, ..., fa7]
    //dL/dagt -> [ft4, ..., ft7]
    //dL/daft -> [ft0, ..., ft3]
    //ht      = [fs0, ..., fs3]
    //fs11 = 1.0
    //free
    //[fs4, ..., fs7, fs8, fs9, fs10]
    //[ft8, ..., ft11]


    //BIASES
    //s3 = biases_floats_input for ifgo. gradient at offset 64
    //biases_floats_hidden has the same values of biases_floats_input
    //i
    flw ft8, 64(s3)
    fadd.s ft8, fa4, ft8
    fsw ft8, 64(s3)

    flw ft8, 68(s3)
    fadd.s ft8, fa5, ft8
    fsw ft8, 68(s3)

    flw ft8, 72(s3)
    fadd.s ft8, fa6, ft8
    fsw ft8, 72(s3)

    flw ft8, 76(s3)
    fadd.s ft8, fa7, ft8
    fsw ft8, 76(s3)

    //f
```

```
    flw ft8, 80(s3)
    fadd.s ft8, ft0, ft8
    fsw ft8, 80(s3)

    flw ft8, 84(s3)
    fadd.s ft8, ft1, ft8
    fsw ft8, 84(s3)

    flw ft8, 88(s3)
    fadd.s ft8, ft2, ft8
    fsw ft8, 88(s3)

    flw ft8, 92(s3)
    fadd.s ft8, ft3, ft8
    fsw ft8, 92(s3)

    //g
    flw ft8, 96(s3)
    fadd.s ft8, ft4, ft8
    fsw ft8, 96(s3)

    flw ft8, 100(s3)
    fadd.s ft8, ft5, ft8
    fsw ft8, 100(s3)

    flw ft8, 104(s3)
    fadd.s ft8, ft6, ft8
    fsw ft8, 104(s3)

    flw ft8, 108(s3)
    fadd.s ft8, ft7, ft8
    fsw ft8, 108(s3)

    //o
    flw ft8, 112(s3)
    fadd.s ft8, fa0, ft8
    fsw ft8, 112(s3)

    flw ft8, 116(s3)
    fadd.s ft8, fa1, ft8
    fsw ft8, 116(s3)

    flw ft8, 120(s3)
    fadd.s ft8, fa2, ft8
    fsw ft8, 120(s3)

    flw ft8, 124(s3)
    fadd.s ft8, fa3, ft8
    fsw ft8, 124(s3)


    //WEIGHTS
    //s2 = weights_floats_hidden (gradients at offset +256)
    //add offset and load ht-1
    addi s5, s5, -16
    flw fs0, 0(s5)
    flw fs1, 4(s5)
    flw fs2, 8(s5)
    flw fs3, 12(s5)
    //ht-1 -> [fs0, ..., fs3]
    //dL/daot -> [fa0, ..., fa3]
    //dL/dait -> [fa4, ..., fa7]
    //dL/dagt -> [ft4, ..., ft7]
    //dL/daft -> [ft0, ..., ft3]
    //i
    // dL/dW = dL/da() * ht-1
    flw ft8, 256(s2)
    fmadd.s ft8, fa4, fs0, ft8
    fsw ft8, 256(s2)

    flw ft8, 260(s2)
```

```
    fmadd.s ft8, fa4, fs1, ft8
    fsw ft8, 260(s2)

    flw ft8, 264(s2)
    fmadd.s ft8, fa4, fs2, ft8
    fsw ft8, 264(s2)

    flw ft8, 268(s2)
    fmadd.s ft8, fa4, fs3, ft8
    fsw ft8, 268(s2)


    flw ft8, 272(s2)
    fmadd.s ft8, fa5, fs0, ft8
    fsw ft8, 272(s2)

    flw ft8, 276(s2)
    fmadd.s ft8, fa5, fs1, ft8
    fsw ft8, 276(s2)

    flw ft8, 280(s2)
    fmadd.s ft8, fa5, fs2, ft8
    fsw ft8, 280(s2)

    flw ft8, 284(s2)
    fmadd.s ft8, fa5, fs3, ft8
    fsw ft8, 284(s2)


    flw ft8, 288(s2)
    fmadd.s ft8, fa6, fs0, ft8
    fsw ft8, 288(s2)

    flw ft8, 292(s2)
    fmadd.s ft8, fa6, fs1, ft8
    fsw ft8, 292(s2)

    flw ft8, 296(s2)
    fmadd.s ft8, fa6, fs2, ft8
    fsw ft8, 296(s2)

    flw ft8, 300(s2)
    fmadd.s ft8, fa6, fs3, ft8
    fsw ft8, 300(s2)


    flw ft8, 304(s2)
    fmadd.s ft8, fa7, fs0, ft8
    fsw ft8, 304(s2)

    flw ft8, 308(s2)
    fmadd.s ft8, fa7, fs1, ft8
    fsw ft8, 308(s2)

    flw ft8, 312(s2)
    fmadd.s ft8, fa7, fs2, ft8
    fsw ft8, 312(s2)

    flw ft8, 316(s2)
    fmadd.s ft8, fa7, fs3, ft8
    fsw ft8, 316(s2)

    //f
    //dL/daft -> [ft0, ..., ft3]
    flw ft8, 320(s2)
    fmadd.s ft8, ft0, fs0, ft8
    fsw ft8, 320(s2)

    flw ft8, 324(s2)
    fmadd.s ft8, ft0, fs1, ft8
    fsw ft8, 324(s2)
```

```
1582
1583      flw ft8, 328(s2)
1584      fmadd.s ft8, ft0, fs2, ft8
1585      fsw ft8, 328(s2)
1586
1587      flw ft8, 332(s2)
1588      fmadd.s ft8, ft0, fs3, ft8
1589      fsw ft8, 332(s2)
1590
1591
1592      flw ft8, 336(s2)
1593      fmadd.s ft8, ft1, fs0, ft8
1594      fsw ft8, 336(s2)
1595
1596      flw ft8, 340(s2)
1597      fmadd.s ft8, ft1, fs1, ft8
1598      fsw ft8, 340(s2)
1599
1600      flw ft8, 344(s2)
1601      fmadd.s ft8, ft1, fs2, ft8
1602      fsw ft8, 344(s2)
1603
1604      flw ft8, 348(s2)
1605      fmadd.s ft8, ft1, fs3, ft8
1606      fsw ft8, 348(s2)
1607
1608
1609      flw ft8, 352(s2)
1610      fmadd.s ft8, ft2, fs0, ft8
1611      fsw ft8, 352(s2)
1612
1613      flw ft8, 356(s2)
1614      fmadd.s ft8, ft2, fs1, ft8
1615      fsw ft8, 356(s2)
1616
1617      flw ft8, 360(s2)
1618      fmadd.s ft8, ft2, fs2, ft8
1619      fsw ft8, 360(s2)
1620
1621      flw ft8, 364(s2)
1622      fmadd.s ft8, ft2, fs3, ft8
1623      fsw ft8, 364(s2)
1624
1625
1626      flw ft8, 368(s2)
1627      fmadd.s ft8, ft3, fs0, ft8
1628      fsw ft8, 368(s2)
1629
1630      flw ft8, 372(s2)
1631      fmadd.s ft8, ft3, fs1, ft8
1632      fsw ft8, 372(s2)
1633
1634      flw ft8, 376(s2)
1635      fmadd.s ft8, ft3, fs2, ft8
1636      fsw ft8, 376(s2)
1637
1638      flw ft8, 380(s2)
1639      fmadd.s ft8, ft3, fs3, ft8
1640      fsw ft8, 380(s2)
1641
1642      //g
1643      //dL/dagt -> [ft4, ..., ft7]
1644
1645      flw ft8, 384(s2)
1646      fmadd.s ft8, ft4, fs0, ft8
1647      fsw ft8, 384(s2)
1648
1649      flw ft8, 388(s2)
1650      fmadd.s ft8, ft4, fs1, ft8
1651      fsw ft8, 388(s2)
1652
```

```
    flw ft8, 392(s2)
    fmadd.s ft8, ft4, fs2, ft8
    fsw ft8, 392(s2)

    flw ft8, 396(s2)
    fmadd.s ft8, ft4, fs3, ft8
    fsw ft8, 396(s2)


    flw ft8, 400(s2)
    fmadd.s ft8, ft5, fs0, ft8
    fsw ft8, 400(s2)

    flw ft8, 404(s2)
    fmadd.s ft8, ft5, fs1, ft8
    fsw ft8, 404(s2)

    flw ft8, 408(s2)
    fmadd.s ft8, ft5, fs2, ft8
    fsw ft8, 408(s2)

    flw ft8, 412(s2)
    fmadd.s ft8, ft5, fs3, ft8
    fsw ft8, 412(s2)


    flw ft8, 416(s2)
    fmadd.s ft8, ft6, fs0, ft8
    fsw ft8, 416(s2)

    flw ft8, 420(s2)
    fmadd.s ft8, ft6, fs1, ft8
    fsw ft8, 420(s2)

    flw ft8, 424(s2)
    fmadd.s ft8, ft6, fs2, ft8
    fsw ft8, 424(s2)

    flw ft8, 428(s2)
    fmadd.s ft8, ft6, fs3, ft8
    fsw ft8, 428(s2)


    flw ft8, 432(s2)
    fmadd.s ft8, ft7, fs0, ft8
    fsw ft8, 432(s2)

    flw ft8, 436(s2)
    fmadd.s ft8, ft7, fs1, ft8
    fsw ft8, 436(s2)

    flw ft8, 440(s2)
    fmadd.s ft8, ft7, fs2, ft8
    fsw ft8, 440(s2)

    flw ft8, 444(s2)
    fmadd.s ft8, ft7, fs3, ft8
    fsw ft8, 444(s2)

    //o
    //dL/daot -> [fa0, ..., fa3]
    flw ft8, 448(s2)
    fmadd.s ft8, fa0, fs0, ft8
    fsw ft8, 448(s2)

    flw ft8, 452(s2)
    fmadd.s ft8, fa0, fs1, ft8
    fsw ft8, 452(s2)

    flw ft8, 456(s2)
    fmadd.s ft8, fa0, fs2, ft8
```

```
    fsw ft8, 456(s2)

    flw ft8, 460(s2)
    fmadd.s ft8, fa0, fs3, ft8
    fsw ft8, 460(s2)


    flw ft8, 464(s2)
    fmadd.s ft8, fa1, fs0, ft8
    fsw ft8, 464(s2)

    flw ft8, 468(s2)
    fmadd.s ft8, fa1, fs1, ft8
    fsw ft8, 468(s2)

    flw ft8, 472(s2)
    fmadd.s ft8, fa1, fs2, ft8
    fsw ft8, 472(s2)

    flw ft8, 476(s2)
    fmadd.s ft8, fa1, fs3, ft8
    fsw ft8, 476(s2)


    flw ft8, 480(s2)
    fmadd.s ft8, fa2, fs0, ft8
    fsw ft8, 480(s2)

    flw ft8, 484(s2)
    fmadd.s ft8, fa2, fs1, ft8
    fsw ft8, 484(s2)

    flw ft8, 488(s2)
    fmadd.s ft8, fa2, fs2, ft8
    fsw ft8, 488(s2)

    flw ft8, 492(s2)
    fmadd.s ft8, fa2, fs3, ft8
    fsw ft8, 492(s2)


    flw ft8, 496(s2)
    fmadd.s ft8, fa3, fs0, ft8
    fsw ft8, 496(s2)

    flw ft8, 500(s2)
    fmadd.s ft8, fa3, fs1, ft8
    fsw ft8, 500(s2)

    flw ft8, 504(s2)
    fmadd.s ft8, fa3, fs2, ft8
    fsw ft8, 504(s2)

    flw ft8, 508(s2)
    fmadd.s ft8, fa3, fs3, ft8
    fsw ft8, 508(s2)


    //s1 = weights_floats_input
    //load xt (s0) -> [fs4, fs5, fs6, fs7, fs8]
    flw fs4, 0(s0)
    flw fs5, 4(s0)
    flw fs6, 8(s0)
    flw fs7, 12(s0)
    flw fs8, 16(s0)

    //i
    //dL/dait -> [fa4, ..., fa7]
    flw ft8, 320(s1)
    fmadd.s ft8, fa4, fs4, ft8
    fsw ft8, 320(s1)
```

```
1795
1796      flw ft8, 324(s1)
1797      fmadd.s ft8, fa4, fs5, ft8
1798      fsw ft8, 324(s1)
1799
1800      flw ft8, 328(s1)
1801      fmadd.s ft8, fa4, fs6, ft8
1802      fsw ft8, 328(s1)
1803
1804      flw ft8, 332(s1)
1805      fmadd.s ft8, fa4, fs7, ft8
1806      fsw ft8, 332(s1)
1807
1808      flw ft8, 336(s1)
1809      fmadd.s ft8, fa4, fs8, ft8
1810      fsw ft8, 336(s1)
1811
1812
1813      flw ft8, 340(s1)
1814      fmadd.s ft8, fa5, fs4, ft8
1815      fsw ft8, 340(s1)
1816
1817      flw ft8, 344(s1)
1818      fmadd.s ft8, fa5, fs5, ft8
1819      fsw ft8, 344(s1)
1820
1821      flw ft8, 348(s1)
1822      fmadd.s ft8, fa5, fs6, ft8
1823      fsw ft8, 348(s1)
1824
1825      flw ft8, 352(s1)
1826      fmadd.s ft8, fa5, fs7, ft8
1827      fsw ft8, 352(s1)
1828
1829      flw ft8, 356(s1)
1830      fmadd.s ft8, fa5, fs8, ft8
1831      fsw ft8, 356(s1)
1832
1833
1834      flw ft8, 360(s1)
1835      fmadd.s ft8, fa6, fs4, ft8
1836      fsw ft8, 360(s1)
1837
1838      flw ft8, 364(s1)
1839      fmadd.s ft8, fa6, fs5, ft8
1840      fsw ft8, 364(s1)
1841
1842      flw ft8, 368(s1)
1843      fmadd.s ft8, fa6, fs6, ft8
1844      fsw ft8, 368(s1)
1845
1846      flw ft8, 372(s1)
1847      fmadd.s ft8, fa6, fs7, ft8
1848      fsw ft8, 372(s1)
1849
1850      flw ft8, 376(s1)
1851      fmadd.s ft8, fa6, fs8, ft8
1852      fsw ft8, 376(s1)
1853
1854
1855      flw ft8, 380(s1)
1856      fmadd.s ft8, fa7, fs4, ft8
1857      fsw ft8, 380(s1)
1858
1859      flw ft8, 384(s1)
1860      fmadd.s ft8, fa7, fs5, ft8
1861      fsw ft8, 384(s1)
1862
1863      flw ft8, 388(s1)
1864      fmadd.s ft8, fa7, fs6, ft8
1865      fsw ft8, 388(s1)
```

```
1866 |
1867 |     flw ft8, 392(s1)
1868 |     fmadd.s ft8, fa7, fs7, ft8
1869 |     fsw ft8, 392(s1)
1870 |
1871 |     flw ft8, 396(s1)
1872 |     fmadd.s ft8, fa7, fs8, ft8
1873 |     fsw ft8, 396(s1)
1874 |
1875 |     //f
1876 |     //dL/daft -> [ft0, ..., ft3] 400
1877 |     flw ft8, 400(s1)
1878 |     fmadd.s ft8, ft0, fs4, ft8
1879 |     fsw ft8, 400(s1)
1880 |
1881 |     flw ft8, 404(s1)
1882 |     fmadd.s ft8, ft0, fs5, ft8
1883 |     fsw ft8, 404(s1)
1884 |
1885 |     flw ft8, 408(s1)
1886 |     fmadd.s ft8, ft0, fs6, ft8
1887 |     fsw ft8, 408(s1)
1888 |
1889 |     flw ft8, 412(s1)
1890 |     fmadd.s ft8, ft0, fs7, ft8
1891 |     fsw ft8, 412(s1)
1892 |
1893 |     flw ft8, 416(s1)
1894 |     fmadd.s ft8, ft0, fs8, ft8
1895 |     fsw ft8, 416(s1)
1896 |
1897 |
1898 |     flw ft8, 420(s1)
1899 |     fmadd.s ft8, ft1, fs4, ft8
1900 |     fsw ft8, 420(s1)
1901 |
1902 |     flw ft8, 424(s1)
1903 |     fmadd.s ft8, ft1, fs5, ft8
1904 |     fsw ft8, 424(s1)
1905 |
1906 |     flw ft8, 428(s1)
1907 |     fmadd.s ft8, ft1, fs6, ft8
1908 |     fsw ft8, 428(s1)
1909 |
1910 |     flw ft8, 432(s1)
1911 |     fmadd.s ft8, ft1, fs7, ft8
1912 |     fsw ft8, 432(s1)
1913 |
1914 |     flw ft8, 436(s1)
1915 |     fmadd.s ft8, ft1, fs8, ft8
1916 |     fsw ft8, 436(s1)
1917 |
1918 |
1919 |     flw ft8, 440(s1)
1920 |     fmadd.s ft8, ft2, fs4, ft8
1921 |     fsw ft8, 440(s1)
1922 |
1923 |     flw ft8, 444(s1)
1924 |     fmadd.s ft8, ft2, fs5, ft8
1925 |     fsw ft8, 444(s1)
1926 |
1927 |     flw ft8, 448(s1)
1928 |     fmadd.s ft8, ft2, fs6, ft8
1929 |     fsw ft8, 448(s1)
1930 |
1931 |     flw ft8, 452(s1)
1932 |     fmadd.s ft8, ft2, fs7, ft8
1933 |     fsw ft8, 452(s1)
1934 |
1935 |     flw ft8, 456(s1)
1936 |     fmadd.s ft8, ft2, fs8, ft8
```

```
    fsw ft8, 456(s1)


    flw ft8, 460(s1)
    fmadd.s ft8, ft3, fs4, ft8
    fsw ft8, 460(s1)

    flw ft8, 464(s1)
    fmadd.s ft8, ft3, fs5, ft8
    fsw ft8, 464(s1)

    flw ft8, 468(s1)
    fmadd.s ft8, ft3, fs6, ft8
    fsw ft8, 468(s1)

    flw ft8, 472(s1)
    fmadd.s ft8, ft3, fs7, ft8
    fsw ft8, 472(s1)

    flw ft8, 476(s1)
    fmadd.s ft8, ft3, fs8, ft8
    fsw ft8, 476(s1)

    //g
    //dL/dagt -> [ft4, ..., ft7]
    flw ft8, 480(s1)
    fmadd.s ft8, ft4, fs4, ft8
    fsw ft8, 480(s1)

    flw ft8, 484(s1)
    fmadd.s ft8, ft4, fs5, ft8
    fsw ft8, 484(s1)

    flw ft8, 488(s1)
    fmadd.s ft8, ft4, fs6, ft8
    fsw ft8, 488(s1)

    flw ft8, 492(s1)
    fmadd.s ft8, ft4, fs7, ft8
    fsw ft8, 492(s1)

    flw ft8, 496(s1)
    fmadd.s ft8, ft4, fs8, ft8
    fsw ft8, 496(s1)


    flw ft8, 500(s1)
    fmadd.s ft8, ft5, fs4, ft8
    fsw ft8, 500(s1)

    flw ft8, 504(s1)
    fmadd.s ft8, ft5, fs5, ft8
    fsw ft8, 504(s1)

    flw ft8, 508(s1)
    fmadd.s ft8, ft5, fs6, ft8
    fsw ft8, 508(s1)

    flw ft8, 512(s1)
    fmadd.s ft8, ft5, fs7, ft8
    fsw ft8, 512(s1)

    flw ft8, 516(s1)
    fmadd.s ft8, ft5, fs8, ft8
    fsw ft8, 516(s1)


    flw ft8, 520(s1)
    fmadd.s ft8, ft6, fs4, ft8
    fsw ft8, 520(s1)
```

```
    flw ft8, 524(s1)
    fmadd.s ft8, ft6, fs5, ft8
    fsw ft8, 524(s1)

    flw ft8, 528(s1)
    fmadd.s ft8, ft6, fs6, ft8
    fsw ft8, 528(s1)

    flw ft8, 532(s1)
    fmadd.s ft8, ft6, fs7, ft8
    fsw ft8, 532(s1)

    flw ft8, 536(s1)
    fmadd.s ft8, ft6, fs8, ft8
    fsw ft8, 536(s1)


    flw ft8, 540(s1)
    fmadd.s ft8, ft7, fs4, ft8
    fsw ft8, 540(s1)

    flw ft8, 544(s1)
    fmadd.s ft8, ft7, fs5, ft8
    fsw ft8, 544(s1)

    flw ft8, 548(s1)
    fmadd.s ft8, ft7, fs6, ft8
    fsw ft8, 548(s1)

    flw ft8, 552(s1)
    fmadd.s ft8, ft7, fs7, ft8
    fsw ft8, 552(s1)

    flw ft8, 556(s1)
    fmadd.s ft8, ft7, fs8, ft8
    fsw ft8, 556(s1)

    //o
    //dL/daot -> [fa0, ..., fa3]
    flw ft8, 560(s1)
    fmadd.s ft8, fa0, fs4, ft8
    fsw ft8, 560(s1)

    flw ft8, 564(s1)
    fmadd.s ft8, fa0, fs5, ft8
    fsw ft8, 564(s1)

    flw ft8, 568(s1)
    fmadd.s ft8, fa0, fs6, ft8
    fsw ft8, 568(s1)

    flw ft8, 572(s1)
    fmadd.s ft8, fa0, fs7, ft8
    fsw ft8, 572(s1)

    flw ft8, 576(s1)
    fmadd.s ft8, fa0, fs8, ft8
    fsw ft8, 576(s1)


    flw ft8, 580(s1)
    fmadd.s ft8, fa1, fs4, ft8
    fsw ft8, 580(s1)

    flw ft8, 584(s1)
    fmadd.s ft8, fa1, fs5, ft8
    fsw ft8, 584(s1)

    flw ft8, 588(s1)
    fmadd.s ft8, fa1, fs6, ft8
    fsw ft8, 588(s1)
```

```
2079
2080        flw ft8, 592(s1)
2081        fmadd.s ft8, fa1, fs7, ft8
2082        fsw ft8, 592(s1)
2083
2084        flw ft8, 596(s1)
2085        fmadd.s ft8, fa1, fs8, ft8
2086        fsw ft8, 596(s1)
2087
2088
2089        flw ft8, 600(s1)
2090        fmadd.s ft8, fa2, fs4, ft8
2091        fsw ft8, 600(s1)
2092
2093        flw ft8, 604(s1)
2094        fmadd.s ft8, fa2, fs5, ft8
2095        fsw ft8, 604(s1)
2096
2097        flw ft8, 608(s1)
2098        fmadd.s ft8, fa2, fs6, ft8
2099        fsw ft8, 608(s1)
2100
2101        flw ft8, 612(s1)
2102        fmadd.s ft8, fa2, fs7, ft8
2103        fsw ft8, 612(s1)
2104
2105        flw ft8, 616(s1)
2106        fmadd.s ft8, fa2, fs8, ft8
2107        fsw ft8, 616(s1)
2108
2109
2110        flw ft8, 620(s1)
2111        fmadd.s ft8, fa3, fs4, ft8
2112        fsw ft8, 620(s1)
2113
2114        flw ft8, 624(s1)
2115        fmadd.s ft8, fa3, fs5, ft8
2116        fsw ft8, 624(s1)
2117
2118        flw ft8, 628(s1)
2119        fmadd.s ft8, fa3, fs6, ft8
2120        fsw ft8, 628(s1)
2121
2122        flw ft8, 632(s1)
2123        fmadd.s ft8, fa3, fs7, ft8
2124        fsw ft8, 632(s1)
2125
2126        flw ft8, 636(s1)
2127        fmadd.s ft8, fa3, fs8, ft8
2128        fsw ft8, 636(s1)
2129
2130
2131        //get dL/dht-1 -> [fs4, ..., fs7]
2132
2133        //dL/dait -> [fa4, ..., fa7]
2134        //dL/daft -> [ft0, ..., ft3]
2135        //dL/dagt -> [ft4, ..., ft7]
2136        //dL/daot -> [fa0, ..., fa3]
2137        //s2 = weights_floats_hidden
2138
2139        //i
2140        flw ft8, 0(s2)
2141        fmul.s fs4, fa4, ft8
2142        flw ft8, 4(s2)
2143        fmul.s fs5, fa4, ft8
2144        flw ft8, 8(s2)
2145        fmul.s fs6, fa4, ft8
2146        flw ft8, 12(s2)
2147        fmul.s fs7, fa4, ft8
2148
2149        flw ft8, 16(s2)
```

```
    fmadd.s fs4, fa5, ft8, fs4
    flw ft8, 20(s2)
    fmadd.s fs5, fa5, ft8, fs5
    flw ft8, 24(s2)
    fmadd.s fs6, fa5, ft8, fs6
    flw ft8, 28(s2)
    fmadd.s fs7, fa5, ft8, fs7

    flw ft8, 32(s2)
    fmadd.s fs4, fa6, ft8, fs4
    flw ft8, 36(s2)
    fmadd.s fs5, fa6, ft8, fs5
    flw ft8, 40(s2)
    fmadd.s fs6, fa6, ft8, fs6
    flw ft8, 44(s2)
    fmadd.s fs7, fa6, ft8, fs7

    flw ft8, 48(s2)
    fmadd.s fs4, fa7, ft8, fs4
    flw ft8, 52(s2)
    fmadd.s fs5, fa7, ft8, fs5
    flw ft8, 56(s2)
    fmadd.s fs6, fa7, ft8, fs6
    flw ft8, 60(s2)
    fmadd.s fs7, fa7, ft8, fs7

    //f
    flw ft8, 64(s2)
    fmadd.s fs4, ft0, ft8, fs4
    flw ft8, 68(s2)
    fmadd.s fs5, ft0, ft8, fs5
    flw ft8, 72(s2)
    fmadd.s fs6, ft0, ft8, fs6
    flw ft8, 76(s2)
    fmadd.s fs7, ft0, ft8, fs7

    flw ft8, 80(s2)
    fmadd.s fs4, ft1, ft8, fs4
    flw ft8, 84(s2)
    fmadd.s fs5, ft1, ft8, fs5
    flw ft8, 88(s2)
    fmadd.s fs6, ft1, ft8, fs6
    flw ft8, 92(s2)
    fmadd.s fs7, ft1, ft8, fs7

    flw ft8, 96(s2)
    fmadd.s fs4, ft2, ft8, fs4
    flw ft8, 100(s2)
    fmadd.s fs5, ft2, ft8, fs5
    flw ft8, 104(s2)
    fmadd.s fs6, ft2, ft8, fs6
    flw ft8, 108(s2)
    fmadd.s fs7, ft2, ft8, fs7

    flw ft8, 112(s2)
    fmadd.s fs4, ft3, ft8, fs4
    flw ft8, 116(s2)
    fmadd.s fs5, ft3, ft8, fs5
    flw ft8, 120(s2)
    fmadd.s fs6, ft3, ft8, fs6
    flw ft8, 124(s2)
    fmadd.s fs7, ft3, ft8, fs7

    //g
    flw ft8, 128(s2)
    fmadd.s fs4, ft4, ft8, fs4
    flw ft8, 132(s2)
    fmadd.s fs5, ft4, ft8, fs5
    flw ft8, 136(s2)
    fmadd.s fs6, ft4, ft8, fs6
    flw ft8, 140(s2)
```

```
    fmadd.s fs7, ft4, ft8, fs7

    flw ft8, 144(s2)
    fmadd.s fs4, ft5, ft8, fs4
    flw ft8, 148(s2)
    fmadd.s fs5, ft5, ft8, fs5
    flw ft8, 152(s2)
    fmadd.s fs6, ft5, ft8, fs6
    flw ft8, 156(s2)
    fmadd.s fs7, ft5, ft8, fs7

    flw ft8, 160(s2)
    fmadd.s fs4, ft6, ft8, fs4
    flw ft8, 164(s2)
    fmadd.s fs5, ft6, ft8, fs5
    flw ft8, 168(s2)
    fmadd.s fs6, ft6, ft8, fs6
    flw ft8, 172(s2)
    fmadd.s fs7, ft6, ft8, fs7

    flw ft8, 176(s2)
    fmadd.s fs4, ft7, ft8, fs4
    flw ft8, 180(s2)
    fmadd.s fs5, ft7, ft8, fs5
    flw ft8, 184(s2)
    fmadd.s fs6, ft7, ft8, fs6
    flw ft8, 188(s2)
    fmadd.s fs7, ft7, ft8, fs7

    //o
    flw ft8, 192(s2)
    fmadd.s fs4, fa0, ft8, fs4
    flw ft8, 196(s2)
    fmadd.s fs5, fa0, ft8, fs5
    flw ft8, 200(s2)
    fmadd.s fs6, fa0, ft8, fs6
    flw ft8, 204(s2)
    fmadd.s fs7, fa0, ft8, fs7

    flw ft8, 208(s2)
    fmadd.s fs4, fa1, ft8, fs4
    flw ft8, 212(s2)
    fmadd.s fs5, fa1, ft8, fs5
    flw ft8, 216(s2)
    fmadd.s fs6, fa1, ft8, fs6
    flw ft8, 220(s2)
    fmadd.s fs7, fa1, ft8, fs7

    flw ft8, 224(s2)
    fmadd.s fs4, fa2, ft8, fs4
    flw ft8, 228(s2)
    fmadd.s fs5, fa2, ft8, fs5
    flw ft8, 232(s2)
    fmadd.s fs6, fa2, ft8, fs6
    flw ft8, 236(s2)
    fmadd.s fs7, fa2, ft8, fs7

    flw ft8, 240(s2)
    fmadd.s fs4, fa3, ft8, fs4
    flw ft8, 244(s2)
    fmadd.s fs5, fa3, ft8, fs5
    flw ft8, 248(s2)
    fmadd.s fs6, fa3, ft8, fs6
    flw ft8, 252(s2)
    fmadd.s fs7, fa3, ft8, fs7


    //get dL/dht-1 -> [fs4, ..., fs7]
    //ht-1 -> [fs0, ..., fs3]

    //offset pointers to t-1
```

```
    //addi s5, s5, -16 //ht already at t-1
    addi s6, s6, -16 //ct
    addi s7, s7, -16 //tanhct
    addi s8, s8, -16 //ifgo



    //TODO: LOOP LSTM BACKPROP WITH BNEZ A7
    //TODO: OFFSET S0 BY -1 ROW EACH TIME, THEN 1 ROW BEFORE RESTARTING AND REMOVE -160
        OFFSET INSTRUCTION AT THE END
    //offset inputs by 1 timestep (1 row = -20)
    addi s0, s0, -20
    bnez a7, begin_backprop_lstm


    //---- FORWARD PASS END ----
    //restart, with input and output offset
    //input(s0): go back 8 rows = -160
    //** CHANGE: s0 is at -20 from its initial offset -> move 2 rows forward
    //addi s0, s0, -160
    addi s0, s0, 40

    //output(a5) offset -> 1 float  = +4
    addi a5, a5, 4

    //"add" -1 to the FeedForward counter (s11)
    //loop feed forward 3797 times
    //repeat until counter reaches 0
    addi s11, s11, -1

    //if s11 != 0 -> repeat
    bnez s11, forward_pass_full_restart


    //divide parameters by train size. Divide lstm params by an extra 10
    //ADAM
    //set all gradients to 0 at the end (S & P stay)
    //skip_to_adam_for_test:
    //-----------------------------------------------------------------
    //------        ADAM        -------------------------------------
    //-----------------------------------------------------------------
    // 0 = parameter
    //P = P * beta1 + (1-beta1)* dL/d0
    //S = S * beta2 + (1-beta2)* dL/d0*dL/d0
    //PP = P/(1- beta1^(t))
    //SS = S/(1- beta2^(t))
    //0 = 0 - lr*PP/sqrt(SS)

    //load betas and learning rate
    la t0, betas_and_others
    //learning rate
    flw fs0, 16(t0)
    //betas
    flw fs1, 0(t0)//beta1
    flw fs2, 4(t0)//beta2

    flw fs3, 8(t0)//beta1^(t)
    fmul.s fs3, fs3, fs1
    fsw fs3, 8(t0)//store for t+1
    fsub.s fs3, fs11, fs3 //1-beta1^t

    flw fs4, 12(t0)//beta2^(t)
    fmul.s fs4, fs4, fs2
    fsw fs4, 12(t0)//store for t+1
    fsub.s fs4, fs11, fs4 //1-beta2^t

    //other floats used for division
    flw fs5, 20(t0)//3342.0
    //flw fs6, 20(t0)//10.0

    //load an epsilon to avoid division by zero
```

```
    flw fs6, 28(t0)

    //ft0 = zero, to set gradients after adam
    fmv.s.x ft0, x0


    //-------------------------------------------------------------fc
    //a4 -> fc_weight_and_bias
    //bias
    flw fa0, 256(a4)
    flw fa1, 1028(a4)
    flw fa2, 1032(a4)
    flw fa3, 1036(a4)
    jal a0, adam_crit
    fsw fa0, 256(a4)
    fsw ft0, 1028(a4)//set gradient to x0
    fsw fa2, 1032(a4)
    fsw fa3, 1036(a4)

    //weights
    //0
    flw fa0, 0(a4)
    flw fa1, 260(a4)
    flw fa2, 516(a4)
    flw fa3, 772(a4)
    jal a0, adam_crit
    fsw fa0, 0(a4)
    fsw ft0, 260(a4)
    fsw fa2, 516(a4)
    fsw fa3, 772(a4)

    //1
    flw fa0, 4(a4)
    flw fa1, 264(a4)
    flw fa2, 520(a4)
    flw fa3, 776(a4)
    jal a0, adam_crit
    fsw fa0, 4(a4)
    fsw ft0, 264(a4)
    fsw fa2, 520(a4)
    fsw fa3, 776(a4)

    //---------------------------------------
    //THIS CODE REPEATS A TOTAL OF 64 TIMES
    //REMOVED DUPLICATES FOR READABILITY
    //---------------------------------------

    //63
    flw fa0, 252(a4)
    flw fa1, 512(a4)
    flw fa2, 768(a4)
    flw fa3, 1024(a4)
    jal a0, adam_crit
    fsw fa0, 252(a4)
    fsw ft0, 512(a4)
    fsw fa2, 768(a4)
    fsw fa3, 1024(a4)

    //-------------------------------------------------------------fc1
    //-----------------------------------------------------------bias
    //a2 -> fc_1_bias

    //0
    flw fa0, 0(a2)
    flw fa1, 256(a2)
    flw fa2, 512(a2)
    flw fa3, 768(a2)
    jal a0, adam_crit
    fsw fa0, 0(a2)
    fsw ft0, 256(a2)
    fsw fa2, 512(a2)
```

```
    fsw fa3, 768(a2)

    //1
    flw fa0, 4(a2)
    flw fa1, 260(a2)
    flw fa2, 516(a2)
    flw fa3, 772(a2)
    jal a0, adam_crit
    fsw fa0, 4(a2)
    fsw ft0, 260(a2)
    fsw fa2, 516(a2)
    fsw fa3, 772(a2)

    //---------------------------------------
    //THIS CODE REPEATS A TOTAL OF 64 TIMES
    //REMOVED DUPLICATES FOR READABILITY
    //---------------------------------------

    //63
    flw fa0, 252(a2)
    flw fa1, 508(a2)
    flw fa2, 764(a2)
    flw fa3, 1020(a2)
    jal a0, adam_crit
    fsw fa0, 252(a2)
    fsw ft0, 508(a2)
    fsw fa2, 764(a2)
    fsw fa3, 1020(a2)


    //-----------------------------------------------------------------weights
    //a1 -> fc_1_weight
    //0
    //cannot use offsets bigger than 2047
    addi t5, a1, 1024
    addi t5, t5, 1024
    flw fa0, 0(a1)
    flw fa1, 1024(a1)
    flw fa2, 0(t5)
    flw fa3, 1024(t5)
    jal a0, adam_crit
    fsw fa0, 0(a1)
    fsw ft0, 1024(a1)
    fsw fa2, 0(t5)
    fsw fa3, 1024(t5)

    //1
    flw fa0, 4(a1)
    flw fa1, 1028(a1)
    flw fa2, 4(t5)
    flw fa3, 1028(t5)
    jal a0, adam_crit
    fsw fa0, 4(a1)
    fsw ft0, 1028(a1)
    fsw fa2, 4(t5)
    fsw fa3, 1028(t5)

    //---------------------------------------
    //THIS CODE REPEATS A TOTAL OF 256 TIMES
    //REMOVED DUPLICATES FOR READABILITY
    //---------------------------------------

    //255
    flw fa0, 1020(a1)
    flw fa1, 2044(a1)
    flw fa2, 1020(t5)
    flw fa3, 2044(t5)
    jal a0, adam_crit
    fsw fa0, 1020(a1)
    fsw ft0, 2044(a1)
    fsw fa2, 1020(t5)
```

```
    fsw fa3, 2044(t5)


    //-----------------------------------------------------------------LSTM
    //multiply fs5 by 10 (because of gradient averaging)
    flw ft1, 24(t0)//10.0
    fmul.s fs5, fs5, ft1
    //WEIGHTS
    //INPUT
    //s1 = weights_floats_input
    //0
    flw fa0, 0(s1)
    flw fa1, 320(s1)
    flw fa2, 640(s1)
    flw fa3, 960(s1)
    jal a0, adam_crit
    fsw fa0, 0(s1)
    fsw ft0, 320(s1)
    fsw fa2, 640(s1)
    fsw fa3, 960(s1)

    //1
    flw fa0, 4(s1)
    flw fa1, 324(s1)
    flw fa2, 644(s1)
    flw fa3, 964(s1)
    jal a0, adam_crit
    fsw fa0, 4(s1)
    fsw ft0, 324(s1)
    fsw fa2, 644(s1)
    fsw fa3, 964(s1)

    //---------------------------------------
    //THIS CODE REPEATS A TOTAL OF 80 TIMES
    //REMOVED DUPLICATES FOR READABILITY
    //---------------------------------------

    //79
    flw fa0, 316(s1)
    flw fa1, 636(s1)
    flw fa2, 956(s1)
    flw fa3, 1276(s1)
    jal a0, adam_crit
    fsw fa0, 316(s1)
    fsw ft0, 636(s1)
    fsw fa2, 956(s1)
    fsw fa3, 1276(s1)


    //-----------------------------------------------------------------HIDDEN
    //s2 = weights_floats_hidden
    //0
    flw fa0, 0(s2)
    flw fa1, 256(s2)
    flw fa2, 512(s2)
    flw fa3, 768(s2)
    jal a0, adam_crit
    fsw fa0, 0(s2)
    fsw ft0, 256(s2)
    fsw fa2, 512(s2)
    fsw fa3, 768(s2)

    //1
    flw fa0, 4(s2)
    flw fa1, 260(s2)
    flw fa2, 516(s2)
    flw fa3, 772(s2)
    jal a0, adam_crit
    fsw fa0, 4(s2)
    fsw ft0, 260(s2)
    fsw fa2, 516(s2)
```

```
    fsw fa3, 772(s2)

    //---------------------------------------
    //THIS CODE REPEATS A TOTAL OF 64 TIMES
    //REMOVED DUPLICATES FOR READABILITY
    //---------------------------------------

    //63
    flw fa0, 252(s2)
    flw fa1, 508(s2)
    flw fa2, 764(s2)
    flw fa3, 1020(s2)
    jal a0, adam_crit
    fsw fa0, 252(s2)
    fsw ft0, 508(s2)
    fsw fa2, 764(s2)
    fsw fa3, 1020(s2)


    //-----------------------------------------------------------------BIASES
    //s3 = biases_floats_input
    //s4 = biases_floats_hidden
    //this one is different because s3 and s4 share the same gradients, P and S
    //0
    flw fa0, 0(s3)
    flw fa1, 64(s3)
    flw fa2, 128(s3)
    flw fa3, 192(s3)
    flw fa4, 0(s4)
    jal a0, adam_crit
    fnmsub.s fa4, fs0, ft2, fa4
    fsw fa0, 0(s3)
    fsw ft0, 64(s3)
    fsw fa2, 128(s3)
    fsw fa3, 192(s3)
    fsw fa4, 0(s4)

    //---------------------------------------
    //THIS CODE REPEATS A TOTAL OF 16 TIMES
    //REMOVED DUPLICATES FOR READABILITY
    //---------------------------------------

    //15
    flw fa0, 60(s3)
    flw fa1, 124(s3)
    flw fa2, 188(s3)
    flw fa3, 252(s3)
    flw fa4, 60(s4)
    jal a0, adam_crit
    fnmsub.s fa4, fs0, ft2, fa4
    fsw fa0, 60(s3)
    fsw ft0, 124(s3)
    fsw fa2, 188(s3)
    fsw fa3, 252(s3)
    fsw fa4, 60(s4)


    //-------------------------------------------------------------------------
    //-------------------------------------------------------------------------
    //-------------------------------------------------------------------------
    //END

    //epoch check
    addi t6, t6, -1
    bnez t6, full_epoch_restart

call exit

adam_crit:
//apply adam, where fa0=PARAM (0) , fa1=GRADIENT (dL/d0), fa2=P , fa3=S
//betas_and_others values must be preloaded
```

```
2646 //fs0=learning rate , fs1=beta1 , fs2=beta2 , fs3=1-beta1^t , fs4=1-beta2^t , fs5=
         gradient_divisor
2647 //fs6=epsilon
2648 //returns adjusted param in fa0, new P in fa2 and new S in fa3
2649 //must be called with jal a0, adam_crit
2650
2651     //---- divides gradient fa1 by fs5
2652     fdiv.s fa1, fa1, fs5
2653
2654     //---- P = P * beta1 + (1-beta1)* dL/d0
2655     //       = P * beta1 + (dL/d0 - beta1*dL/d0)
2656     //-(beta1*dL/d0) + dL/d0
2657     fnmsub.s ft1, fs1, fa1, fa1
2658     //P = P * beta1 + ans
2659     fmadd.s fa2, fa2, fs1, ft1
2660
2661     //---- S = S * beta2 + (1-beta2)* dL/d0*dL/d0
2662     //       = S * beta2 + ((dL/d0*dL/d0)-beta2 * (dL/d0*dL/d0))
2663     //dL/d0*dL/d0
2664     fmul.s fa1, fa1, fa1
2665     //-(ans * beta2) + ans
2666     fnmsub.s ft1, fa1, fs2, fa1
2667     //S = S * beta2 + ans
2668     fmadd.s fa3, fa3, fs2, ft1
2669
2670     //---- PP = P/(1- beta1^(t))
2671     fdiv.s ft2, fa2, fs3
2672
2673     //---- SS = S/(1- beta2^(t))
2674     fdiv.s ft3, fa3, fs4
2675
2676     //---- 0 = 0 - lr*PP/(sqrt(SS)+epsilon)
2677     //sqrt(SS)
2678     fsqrt.s ft3, ft3
2679     //ans + epsilon
2680     fadd.s ft3, ft3, fs6
2681     //PP / ans
2682     fdiv.s ft2, ft2, ft3
2683     //0 =- (lr*ans) + 0
2684     fnmsub.s fa0, fs0, ft2, fa0
2685
2686     //---- finish
2687     jr a0
2688
2689
2690 tanh_fa6:
2691 //apply tanh(x) where x = fa6
2692 //if -2 < x < 2  -> tanh = x - 0.25*x  *sign(x)
2693 //else tanh=sign(x)
2694 //tanh result stored in fa7
2695 //must be called with jal a0, tanh_fa6
2696
2697
2698     //---- load address of floats to be used
2699     la t1, tanh_floats
2700
2701     //---- load 2 in ft10 and check if abs(x)<2
2702     flw ft10, 0(t1) //ft10 = 2
2703     fabs.s ft11, fa6 //ft11 = abs(x)
2704     flt.s t2, ft11, ft10 //t2 == 1 if |x|<2
2705     beqz t2, tanh_is_not_in_curve // t2 ==0 -> skip to return sign(x)
2706
2707     //---- else: do x - 0.25*x  *sign(x)
2708     //x
2709     fmul.s fa7, fa6, fa6
2710     //ans*sign(x)
2711     fsgnj.s fa7, fa7, fa6 //sign inject fa7(ans) with fa6(x) sign
2712     //-(ans*0.25)+x
2713     flw ft10, 4(t1); // ft10 = 0.25
2714     fnmsub.s fa7, fa7, ft10, fa6
2715
```

```
    //---- finish
    jr a0

//---- x < -2 or 2 < x
tanh_is_not_in_curve: //return sign
    fsgnj.s fa7, fs11, fa6

jr a0
//---- end of tanh

sig_fa6:
//apply sigmoid(x) where x = fa6
//if x <= -4      -> sig = 0
//   -4 < x < 4   -> sig = 0.5 + 0.25*x - 0.03125*x  *sign(x)
//    4 <= x      -> sig = 1
//sig result stored in fa7
//must be called with jal a0, sig_fa6


    //---- load address of floats to be used
    la t1, sig_floats

    //---- load 4 and check if abs(x)<4
    flw ft10, 0(t1) //ft10 = 4
    fabs.s ft11, fa6 //ft11 = abs(x)
    flt.s t2, ft11, ft10 //t2 == 1 if |x|<4
    beqz t2, sig_is_not_in_curve // t2 ==0 -> skip to return 0 or 1

    //---- sig = 0.5 + 0.25*x - 0.03125*x  *sign(x)
    //x
    fmul.s fa7, fa6, fa6
    //ans*sign(x)
    fsgnj.s fa7, fa7, fa6 //sign inject fa7(ans) with fa6(x) sign
    //-(ans*0.03125)+0.5
    flw ft10, 4(t1) // ft10 = 0.5
    flw ft11, 12(t1) // ft11 = 0.03125
    fnmsub.s fa7, fa7, ft11, ft10 //fa7 = -(fa7 * ft11) + ft10

    //(0.25*x)+ans
    flw ft10, 8(t1) // ft10 = 0.25
    fmadd.s fa7, ft10, fa6, fa7 //fa7 = (fa7 * ft10) + fa7

    //---- finish
    jr a0

//---- x < -4 or 4 < x
sig_is_not_in_curve:
    // 4 <= x?
    fle.s t2, ft10, fa6 //t2==1 if 4<=x , else (x < -4) t2==0
    fcvt.s.w fa7, t2    //fa7 = float(t2)

jr a0
//---- end of sigmoid
```

RISC-V ISA

A continuación se muestran las instrucciones de la *ISA RISC-V*. Obtenido en [23].

Free & Open RISC-V Reference Card ①

Base Integer Instructions: RV32I, RV64I, and RV128I

Category	Name	Fmt	RV32I Base	+RV{64,128}
Loads	Load Byte	I	`LB rd,rs1,imm`	
	Load Halfword	I	`LH rd,rs1,imm`	
	Load Word	I	`LW rd,rs1,imm`	`L{D|Q} rd,rs1,imm`
	Load Byte Unsigned	I	`LBU rd,rs1,imm`	
	Load Half Unsigned	I	`LHU rd,rs1,imm`	`L{W|D}U rd,rs1,imm`
Stores	Store Byte	S	`SB rs1,rs2,imm`	
	Store Halfword	S	`SH rs1,rs2,imm`	
	Store Word	S	`SW rs1,rs2,imm`	`S{D|Q} rs1,rs2,imm`
Shifts	Shift Left	R	`SLL rd,rs1,rs2`	`SLL{W|D} rd,rs1,rs2`
	Shift Left Immediate	I	`SLLI rd,rs1,shamt`	`SLLI{W|D} rd,rs1,shamt`
	Shift Right	R	`SRL rd,rs1,rs2`	`SRL{W|D} rd,rs1,rs2`
	Shift Right Immediate	I	`SRLI rd,rs1,shamt`	`SRLI{W|D} rd,rs1,shamt`
	Shift Right Arithmetic	R	`SRA rd,rs1,rs2`	`SRA{W|D} rd,rs1,rs2`
	Shift Right Arith Imm	I	`SRAI rd,rs1,shamt`	`SRAI{W|D} rd,rs1,shamt`
Arithmetic	ADD	R	`ADD rd,rs1,rs2`	`ADD{W|D} rd,rs1,rs2`
	ADD Immediate	I	`ADDI rd,rs1,imm`	`ADDI{W|D} rd,rs1,imm`
	SUBtract	R	`SUB rd,rs1,rs2`	`SUB{W|D} rd,rs1,rs2`
	Load Upper Imm	U	`LUI rd,imm`	
	Add Upper Imm to PC	U	`AUIPC rd,imm`	
Logical	XOR	R	`XOR rd,rs1,rs2`	
	XOR Immediate	I	`XORI rd,rs1,imm`	
	OR	R	`OR rd,rs1,rs2`	
	OR Immediate	I	`ORI rd,rs1,imm`	
	AND	R	`AND rd,rs1,rs2`	
	AND Immediate	I	`ANDI rd,rs1,imm`	
Compare	Set <	R	`SLT rd,rs1,rs2`	
	Set < Immediate	I	`SLTI rd,rs1,imm`	
	Set < Unsigned	R	`SLTU rd,rs1,rs2`	
	Set < Imm Unsigned	I	`SLTIU rd,rs1,imm`	
Branches	Branch =	SB	`BEQ rs1,rs2,imm`	
	Branch ≠	SB	`BNE rs1,rs2,imm`	
	Branch <	SB	`BLT rs1,rs2,imm`	
	Branch ≥	SB	`BGE rs1,rs2,imm`	
	Branch < Unsigned	SB	`BLTU rs1,rs2,imm`	
	Branch ≥ Unsigned	SB	`BGEU rs1,rs2,imm`	
Jump & Link	J&L	UJ	`JAL rd,imm`	
	Jump & Link Register	UJ	`JALR rd,rs1,imm`	
Synch	Synch thread	I	`FENCE`	
	Synch Instr & Data	I	`FENCE.I`	
System	System CALL	I	`SCALL`	
	System BREAK	I	`SBREAK`	
Counters	ReaD CYCLE	I	`RDCYCLE rd`	
	ReaD CYCLE upper Half	I	`RDCYCLEH rd`	
	ReaD TIME	I	`RDTIME rd`	
	ReaD TIME upper Half	I	`RDTIMEH rd`	
	ReaD INSTR RETired	I	`RDINSTRET rd`	
	ReaD INSTR upper Half	I	`RDINSTRETH rd`	

RV Privileged Instructions

Category	Name	RV mnemonic
CSR Access	Atomic R/W	`CSRRW rd,csr,rs1`
	Atomic Read & Set Bit	`CSRRS rd,csr,rs1`
	Atomic Read & Clear Bit	`CSRRC rd,csr,rs1`
	Atomic R/W Imm	`CSRRWI rd,csr,imm`
	Atomic Read & Set Bit Imm	`CSRRSI rd,csr,imm`
	Atomic Read & Clear Bit Imm	`CSRRCI rd,csr,imm`
Change Level	Env. Call	`ECALL`
	Environment Breakpoint	`EBREAK`
	Environment Return	`ERET`
Trap Redirect	to Supervisor	`MRTS`
	Redirect Trap to Hypervisor	`MRTH`
	Hypervisor Trap to Supervisor	`HRTS`
Interrupt	Wait for Interrupt	`WFI`
MMU	Supervisor FENCE	`SFENCE.VM rs1`

Optional Compressed (16-bit) Instruction Extension: RVC

Category	Name	Fmt	RVC	RVI equivalent
Loads	Load Word	CL	`C.LW rd',rs1',imm`	`LW rd',rs1',imm*4`
	Load Word SP	CI	`C.LWSP rd,imm`	`LW rd,sp,imm*4`
	Load Double	CL	`C.LD rd',rs1',imm`	`LD rd',rs1',imm*8`
	Load Double SP	CI	`C.LDSP rd,imm`	`LD rd,sp,imm*8`
	Load Quad	CL	`C.LQ rd',rs1',imm`	`LQ rd',rs1',imm*16`
	Load Quad SP	CI	`C.LQSP rd,imm`	`LQ rd,sp,imm*16`
Stores	Store Word	CS	`C.SW rs1',rs2',imm`	`SW rs1',rs2',imm*4`
	Store Word SP	CSS	`C.SWSP rs2,imm`	`SW rs2,sp,imm*4`
	Store Double	CS	`C.SD rs1',rs2',imm`	`SD rs1',rs2',imm*8`
	Store Double SP	CSS	`C.SDSP rs2,imm`	`SD rs2,sp,imm*8`
	Store Quad	CS	`C.SQ rs1',rs2',imm`	`SQ rs1',rs2',imm*16`
	Store Quad SP	CSS	`C.SQSP rs2,imm`	`SQ rs2,sp,imm*16`
Arithmetic	ADD	CR	`C.ADD rd,rs1`	`ADD rd,rd,rs1`
	ADD Word	CR	`C.ADDW rd,rs1`	`ADDW rd,rd,imm`
	ADD Immediate	CI	`C.ADDI rd,imm`	`ADDI rd,rd,imm`
	ADD Word Imm	CI	`C.ADDIW rd,imm`	`ADDIW rd,rd,imm`
	ADD SP Imm * 16	CI	`C.ADDI16SP x0,imm`	`ADDI sp,sp,imm*16`
	ADD SP Imm * 4	CIW	`C.ADDI4SPN rd',imm`	`ADDI rd',sp,imm*4`
	Load Immediate	CI	`C.LI rd,imm`	`ADDI rd,x0,imm`
	Load Upper Imm	CI	`C.LUI rd,imm`	`LUI rd,imm`
	MoVe	CR	`C.MV rd,rs1`	`ADD rd,rs1,x0`
	SUB	CR	`C.SUB rd,rs1`	`SUB rd,rd,rs1`
Shifts	Shift Left Imm	CI	`C.SLLI rd,imm`	`SLLI rd,rd,imm`
Branches	Branch=0	CB	`C.BEQZ rs1',imm`	`BEQ rs1',x0,imm`
	Branch≠0	CB	`C.BNEZ rs1',imm`	`BNE rs1',x0,imm`
Jump	Jump	CJ	`C.J imm`	`JAL x0,imm`
	Jump Register	CR	`C.JR rd,rs1`	`JALR x0,rs1,0`
Jump & Link	J&L	CJ	`C.JAL imm`	`JAL ra,imm`
	Jump & Link Register	CR	`C.JALR rs1`	`JALR ra,rs1,0`
System	Env. BREAK	CI	`C.EBREAK`	`EBREAK`

32-bit Instruction Formats

	31 30 25	24 21	20	19 15	14 12	11 8	7	6 0
R	funct7	rs2		rs1	funct3	rd		opcode
I	imm[11:0]			rs1	funct3	rd		opcode
S	imm[11:5]	rs2		rs1	funct3	imm[4:0]		opcode
SB	imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode
U	imm[31:12]					rd		opcode
UJ	imm[20]	imm[10:1]	imm[11]	imm[19:12]		rd		opcode

16-bit (RVC) Instruction Formats

	15 14 13	12	11 10	9 8 7	6 5	4 3 2	1 0
CR	funct4		rd/rs1		rs2		op
CI	funct3	imm	rd/rs1		imm		op
CSS	funct3	imm			rs2		op
CIW	funct3	imm				rd′	op
CL	funct3	imm		rs1′	imm	rd′	op
CS	funct3	imm		rs1′	imm	rs2′	op
CB	funct3	offset		rs1′	offset		op
CJ	funct3	jump target					op

RISC-V Integer Base (RV32I/64I/128I), privileged, and optional compressed extension (RVC). Registers x1-x31 and the pc are 32 bits wide in RV32I, 64 in RV64I, and 128 in RV128I (x0=0). RV64I/128I add 10 instructions for the wider formats. The RVI base of <50 classic integer RISC instructions is required. Every 16-bit RVC instruction matches an existing 32-bit RVI instruction. See risc.org.

Free & Open RISC-V Reference Card (riscv.org) ②

Optional Multiply-Divide Instruction Extension: RVM

Category	Name	Fmt	RV32M (Multiply-Divide)		+RV{64,128}	
Multiply	MULtiply	R	MUL	rd,rs1,rs2	MUL{W|D}	rd,rs1,rs2
	MULtiply upper Half	R	MULH	rd,rs1,rs2		
	MULtiply Half Sign/Uns	R	MULHSU	rd,rs1,rs2		
	MULtiply upper Half Uns	R	MULHU	rd,rs1,rs2		
Divide	DIVide	R	DIV	rd,rs1,rs2	DIV{W|D}	rd,rs1,rs2
	DIVide Unsigned	R	DIVU	rd,rs1,rs2		
Remainder	REMainder	R	REM	rd,rs1,rs2	REM{W|D}	rd,rs1,rs2
	REMainder Unsigned	R	REMU	rd,rs1,rs2	REMU{W|D}	rd,rs1,rs2

Optional Atomic Instruction Extension: RVA

Category	Name	Fmt	RV32A (Atomic)		+RV{64,128}	
Load	Load Reserved	R	LR.W	rd,rs1	LR.{D|Q}	rd,rs1
Store	Store Conditional	R	SC.W	rd,rs1,rs2	SC.{D|Q}	rd,rs1,rs2
Swap	SWAP	R	AMOSWAP.W	rd,rs1,rs2	AMOSWAP.{D|Q}	rd,rs1,rs2
Add	ADD	R	AMOADD.W	rd,rs1,rs2	AMOADD.{D|Q}	rd,rs1,rs2
Logical	XOR	R	AMOXOR.W	rd,rs1,rs2	AMOXOR.{D|Q}	rd,rs1,rs2
	AND	R	AMOAND.W	rd,rs1,rs2	AMOAND.{D|Q}	rd,rs1,rs2
	OR	R	AMOOR.W	rd,rs1,rs2	AMOOR.{D|Q}	rd,rs1,rs2
Min/Max	MINimum	R	AMOMIN.W	rd,rs1,rs2	AMOMIN.{D|Q}	rd,rs1,rs2
	MAXimum	R	AMOMAX.W	rd,rs1,rs2	AMOMAX.{D|Q}	rd,rs1,rs2
	MINimum Unsigned	R	AMOMINU.W	rd,rs1,rs2	AMOMINU.{D|Q}	rd,rs1,rs2
	MAXimum Unsigned	R	AMOMAXU.W	rd,rs1,rs2	AMOMAXU.{D|Q}	rd,rs1,rs2

Three Optional Floating-Point Instruction Extensions: RVF, RVD, & RVQ

Category	Name	Fmt	RV32{F|D|Q} (HP/SP,DP,QP Fl Pt)		+RV{64,128}	
Move	Move from Integer	R	FMV.{H|S}.X	rd,rs1	FMV.{D|Q}.X	rd,rs1
	Move to Integer	R	FMV.X.{H|S}	rd,rs1	FMV.X.{D|Q}	rd,rs1
Convert	Convert from Int	R	FCVT.{H|S|D|Q}.W	rd,rs1	FCVT.{H|S|D|Q}.{L|T}	rd,rs1
	Convert from Int Unsigned	R	FCVT.{H|S|D|Q}.WU	rd,rs1	FCVT.{H|S|D|Q}.{L|T}U	rd,rs1
	Convert to Int	R	FCVT.W.{H|S|D|Q}	rd,rs1	FCVT.{L|T}.{H|S|D|Q}	rd,rs1
	Convert to Int Unsigned	R	FCVT.WU.{H|S|D|Q}	rd,rs1	FCVT.{L|T}U.{H|S|D|Q}	rd,rs1
Load	Load	I	FL{W,D,Q}	rd,rs1,imm		
Store	Store	S	FS{W,D,Q}	rs1,rs2,imm		
Arithmetic	ADD	R	FADD.{S|D|Q}	rd,rs1,rs2		
	SUBtract	R	FSUB.{S|D|Q}	rd,rs1,rs2		
	MULtiply	R	FMUL.{S|D|Q}	rd,rs1,rs2		
	DIVide	R	FDIV.{S|D|Q}	rd,rs1,rs2		
	SQuare RooT	R	FSQRT.{S|D|Q}	rd,rs1		
Mul-Add	Multiply-ADD	R	FMADD.{S|D|Q}	rd,rs1,rs2,rs3		
	Multiply-SUBtract	R	FMSUB.{S|D|Q}	rd,rs1,rs2,rs3		
	Negative Multiply-SUBtract	R	FNMSUB.{S|D|Q}	rd,rs1,rs2,rs3		
	Negative Multiply-ADD	R	FNMADD.{S|D|Q}	rd,rs1,rs2,rs3		
Sign Inject	SiGN source	R	FSGNJ.{S|D|Q}	rd,rs1,rs2		
	Negative SiGN source	R	FSGNJN.{S|D|Q}	rd,rs1,rs2		
	Xor SiGN source	R	FSGNJX.{S|D|Q}	rd,rs1,rs2		
Min/Max	MINimum	R	FMIN.{S|D|Q}	rd,rs1,rs2		
	MAXimum	R	FMAX.{S|D|Q}	rd,rs1,rs2		
Compare	Compare Float =	R	FEQ.{S|D|Q}	rd,rs1,rs2		
	Compare Float <	R	FLT.{S|D|Q}	rd,rs1,rs2		
	Compare Float ≤	R	FLE.{S|D|Q}	rd,rs1,rs2		
Categorization	Classify Type	R	FCLASS.{S|D|Q}	rd,rs1		
Configuration	Read Status	R	FRCSR	rd		
	Read Rounding Mode	R	FRRM	rd		
	Read Flags	R	FRFLAGS	rd		
	Swap Status Reg	R	FSCSR	rd,rs1		
	Swap Rounding Mode	R	FSRM	rd,rs1		
	Swap Flags	R	FSFLAGS	rd,rs1		
	Swap Rounding Mode Imm	I	FSRMI	rd,imm		
	Swap Flags Imm	I	FSFLAGSI	rd,imm		

RISC-V Calling Convention

Register	ABI Name	Saver	Description
x0	zero	---	Hard-wired zero
x1	ra	Caller	Return address
x2	sp	Callee	Stack pointer
x3	gp	---	Global pointer
x4	tp	---	Thread pointer
x5-7	t0-2	Caller	Temporaries
x8	s0/fp	Callee	Saved register/frame pointer
x9	s1	Callee	Saved register
x10-11	a0-1	Caller	Function arguments/return values
x12-17	a2-7	Caller	Function arguments
x18-27	s2-11	Callee	Saved registers
x28-31	t3-t6	Caller	Temporaries
f0-7	ft0-7	Caller	FP temporaries
f8-9	fs0-1	Callee	FP saved registers
f10-11	fa0-1	Caller	FP arguments/return values
f12-17	fa2-7	Caller	FP arguments
f18-27	fs2-11	Callee	FP saved registers
f28-31	ft8-11	Caller	FP temporaries

RISC-V calling convention and five optional extensions: 10 multiply-divide instructions (RV32M); 11 optional atomic instructions (RV32A); and 25 floating-point instructions each for single-, double-, and quadruple-precision (RV32F, RV32D, RV32Q). The latter add registers f0-f31, whose width matches the widest precision, and a floating-point control and status register fcsr. Each larger address adds some instructions: 4 for RVM, 11 for RVA, and 6 each for RVF/D/Q. Using regex notation, { } means set, so `L{D|Q}` *is both* `LD` *and* `LQ`*. See risc.org. (8/21/15 revision)*