UNIVERSIDAD SANTIAGO DE CHILE
DEPARTAMENTO INGENIERÍA INFORMÁTICA

Fundamentos de Aprendizaje Profundo
Informe Nº 1: "LABORATORIO 1"

Integrantes:
Antonina Arriagada G

Profesores:

Francisco Muñoz
Gonzalo Acuña

Índice

Índice de figuras

Índice de tablas

1. Introducción

1.1. Modelo Simple: Red Neuronal Feedforward

El modelo `SimpleFFNN` (*Simple Feedforward Neural Network*) es una implementación de una red neuronal totalmente conectada que consiste en tres capas lineales (*fully connected layers*) con funciones de activación no lineales aplicadas entre ellas. Este tipo de modelo es adecuado para tareas de clasificación binaria o regresión cuando la salida está limitada a un rango entre 0 y 1.

1.1.1. Arquitectura del Modelo

La arquitectura del modelo consta de las siguientes capas:

1. Primera capa totalmente conectada (`fc1`): Toma un vector de entrada de tamaño `input_size` y lo transforma en un vector de tamaño `hidden_size1`. Incluye un término de sesgo (*bias*).

2. Segunda capa totalmente conectada (`fc2`): Recibe el vector transformado de la primera capa (dimensión `hidden_size1`) y produce un vector de dimensión `hidden_size2`, también con un término de sesgo.

3. Tercera capa totalmente conectada (`fc3`): Proporciona una salida de tamaño `output_size`, que por defecto es 1. Esta capa aplica la función de activación `sigmoid`, lo que restringe la salida al rango $[0, 1]$.

1.1.2. Funciones de Activación

El modelo utiliza dos funciones de activación:

- `torch.tanh`: Aplicada después de las dos primeras capas (`fc1` y `fc2`), esta función introduce no linealidad al modelo, lo que permite aprender relaciones complejas. La función `tanh` está definida como:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}},$$

  y transforma la entrada en un rango $[-1, 1]$.

- `torch.sigmoid`: Aplicada en la salida del modelo (`fc3`), esta función convierte las activaciones en probabilidades al comprimirlas en el rango $[0, 1]$:

$$\sigma(x) = \frac{1}{1 + e^{-x}}.$$

1.1.3. Forward Pass

El método `forward` define el flujo de datos a través de la red: 1. Se asegura de que la entrada tenga un formato adecuado usando `x.view`. 2. Pasa los datos por cada capa totalmente conectada (`fc1`, `fc2`, `fc3`), aplicando funciones de activación entre capas.

1.1.4. Número de Parámetros

El método `num_parameters` calcula el número total de parámetros entrenables (pesos y sesgos) en el modelo. Este valor es importante para evaluar la complejidad del modelo y su capacidad para aprender patrones.

2. Marco Teórico

2.1. Inicialización de Pesos en Redes Neuronales

La inicialización de los pesos es un mecanismo que define los valores iniciales de los pesos antes de comenzar el entrenamiento de la red, es fundamental ya que define las condiciones iniciales para la propagación hacia adelante (*forward pass*) y la retropropagación (*backpropagation*) durante el entrenamiento. Lo anterior, afecta directamente la estabilidad del modelo, la rapidez de la convergencia y su rendimiento final. Eso significa que una inicialización inadecuada puede provocar problemas como el desvanecimiento o la explosión de gradientes, dificultando la optimización del modelo [1].

2.1.1. Inicialización de Xavier (Glorot):

Propuesta por Glorot y Bengio [1], esta técnica es adecuada para redes neuronales con funciones de activación simétricas como `tanh` o `sigmoid`. La idea central es que los valores iniciales de los pesos se escalen según el tamaño de las capas de entrada y salida, logrando que la varianza de los gradientes se mantenga estable a lo largo de las capas. Esto mejora la propagación del gradiente y mitiga el problema del desvanecimiento:

$$W \sim \mathcal{U}\left(-\sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}, \sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}\right)$$

donde n_{in} y n_{out} son el número de neuronas en las capas de entrada y salida, respectivamente.

2.1.2. Inicialización de He:

Propuesta por He et al. [2], esta técnica está diseñada específicamente para redes neuronales con funciones de activación como `ReLU`. La inicialización de He utiliza una distribución escalada de los pesos para evitar la saturación de las unidades activas y asegurar que el gradiente fluya adecuadamente a través de la red:

$$W \sim \mathcal{N}\left(0, \sqrt{\frac{2}{n_{\text{in}}}}\right)$$

Este método es especialmente útil en redes profundas, donde las funciones de activación `ReLU` pueden causar problemas de desvanecimiento si los pesos no están correctamente inicializados.

2.2. Normalización de Datos

La normalización de los datos de entrada es esencial para garantizar una convergencia rápida y estable del modelo. Este proceso ajusta el rango de los datos, asegurando que los gradientes fluyan de manera consistente y evitando fluctuaciones extremas durante la optimización. La normalización generalmente implica escalar los valores a un rango específico, como $[0, 1]$ o $[-1, 1]$, dependiendo de los requisitos de la arquitectura del modelo [3].

Por otro lado, la estandarización es un enfoque diferente, pero relacionado, que ajusta los datos para que tengan una media de 0 y una desviación estándar de 1. Este proceso implica centrar los datos restando la media y luego escalarlos dividiendo por la desviación estándar. La estandarización es especialmente útil cuando los modelos asumen que los datos de entrada siguen una distribución normal o aproximadamente normal [4].

2.2.1. Normalización

Proceso de escalar los datos para que estén dentro de un rango específico, como $[0, 1]$ o $[-1, 1]$. - Fórmula típica:

$$x' = \frac{x - x_{\text{min}}}{x_{\text{max}} - x_{\text{min}}}$$

donde x_{min} y x_{max} son los valores mínimo y máximo del conjunto de datos. Es adecuada para redes neuronales con funciones de activación que operan en rangos específicos, como la sigmoid o la tanh. Lo anterior, garantiza que todos los valores estén en el mismo rango, lo que mejora la estabilidad numérica, no obstante, puede ser sensible a valores atípicos (outliers) que pueden distorsionar el rango.

2.2.2. Estandarización

Consiste en el proceso de transformar los datos para que tengan una media de 0 y una desviación estándar de 1.
- Fórmula típica:

$$x' = \frac{x - \mu}{\sigma}$$

donde μ es la media de los datos y σ es la desviación estándar.

Si bien puede ser más costosa computacionalmente, ya que requiere calcular la media y la desviación estándar, es adecuada para modelos que asumen distribuciones normales en los datos de entrada, como los basados en SVM o regresión logística y es menos sensible a los valores atípicos que la normalización porque considera la variabilidad en los datos.

2.3. Optimizadores

La optimización es el proceso mediante el cual se ajustan los parámetros de un modelo, como los pesos y sesgos, y así minimizar una función de pérdida. En redes neuronales, los algoritmos de optimización buscan encontrar un conjunto de parámetros que produzca el mejor rendimiento del modelo. A continuación, se describen algunos de los optimizadores más comunes.

2.3.1. Descenso de Gradiente Estocástico (SGD)

El Descenso de Gradiente Estocástico (Stochastic Gradient Descent, SGD) actualiza los parámetros del modelo en cada paso utilizando una muestra aleatoria (o un mini-lote) del conjunto de datos. La actualización de los pesos se calcula como:

$$w_{t+1} = w_t - \eta \nabla L(w_t)$$

donde w_t son los pesos en el paso t, η es la tasa de aprendizaje, y $\nabla L(w_t)$ es el gradiente de la función de pérdida. Aunque es simple y eficiente, el SGD puede converger lentamente o quedar atrapado en mínimos locales.

2.3.2. SGD con Momentum

El SGD con Momentum introduce un término de "momentum"para acumular gradientes pasados y suavizar las actualizaciones. Esto permite superar barreras pequeñas en la superficie de pérdida y acelerar la convergencia:

$$v_t = \gamma v_{t-1} + \eta \nabla L(w_t)$$

$$w_{t+1} = w_t - v_t$$

donde v_t es el término de velocidad acumulada y γ es el coeficiente de momentum ($0 \leq \gamma < 1$).

2.3.3. SGD con Nesterov Momentum

El SGD con Nesterov Momentum es una mejora del SGD con Momentum que calcula el gradiente en un punto anticipado, proporcionando actualizaciones más precisas:

$$v_t = \gamma v_{t-1} + \eta \nabla L(w_t - \gamma v_{t-1})$$

$$w_{t+1} = w_t - v_t$$

Este enfoque ayuda a evitar oscilaciones excesivas y mejora la estabilidad.

2.3.4. RMSProp

El algoritmo RMSProp adapta la tasa de aprendizaje para cada parámetro dividiendo el gradiente por una media móvil de los gradientes al cuadrado:

$$E[g^2]_t = \rho E[g^2]_{t-1} + (1-\rho) g_t^2$$

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

donde ρ es el factor de decaimiento y ϵ es un término para evitar divisiones por cero. RMSProp es útil para problemas no estacionarios y redes profundas.

2.3.5. Adam

El algoritmo Adam combina las ideas de Momentum y RMSProp, utilizando estimaciones de primer y segundo orden de los momentos del gradiente:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$$

donde m_t y v_t son las estimaciones de los momentos, β_1 y β_2 son los coeficientes de decaimiento, y ϵ es un término de estabilidad numérica. Adam es ampliamente utilizado por su eficiencia y capacidad de adaptación.

2.4. Regularización

La regularización es un conjunto de técnicas diseñadas para prevenir el sobreajuste (*overfitting*) en modelos de aprendizaje automático, permitiendo que el modelo generalice mejor a datos no vistos. A continuación, se describen algunos de los métodos más comunes.

2.4.1. Dropout

El *Dropout* es una técnica que introduce aleatoriedad durante el entrenamiento al "desactivar"(es decir, poner en 0) ciertas neuronas de manera aleatoria en cada iteración. Esto fuerza a la red a no depender exclusivamente de conexiones específicas y a aprender representaciones más generalizadas:

$$y = Wx + b \quad \text{con probabilidad } 1 - p,$$

donde p es la probabilidad de desactivar una neurona. Durante la inferencia, no se aplica el *Dropout*, pero las activaciones se escalan por $1 - p$ para mantener la magnitud de las señales consistente [8].

2.4.2. Batch Normalization

La *Batch Normalization* es una técnica que normaliza las activaciones de las capas dentro de un mini-lote, ajustándolas para tener una media cercana a 0 y una varianza cercana a 1. Esto reduce el problema de cambio interno de covariables (*internal covariate shift*) y acelera la convergencia:

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}},$$

$$y_i = \gamma \hat{x}_i + \beta,$$

donde μ_B y σ_B^2 son la media y varianza del mini-lote, y γ y β son parámetros aprendibles [9]. Este método también actúa como una forma de regularización, ya que introduce ruido adicional al cálculo de activaciones.

2.4.3. Regularización L2

La regularización $L2$, también conocida como *weight decay*, penaliza los pesos grandes agregando un término a la función de pérdida proporcional al cuadrado de la magnitud de los pesos:

$$L = L_0 + \frac{\lambda}{2} \sum_i w_i^2,$$

donde L_0 es la pérdida original, w_i son los pesos y λ es el coeficiente de regularización. Este método fuerza al modelo a preferir pesos más pequeños, lo que reduce la complejidad del modelo y mejora la generalización [10].

3. Conjuntos de datos

3.1. Random Dataset

El primer dataset implementado es de tipo `RandomDataSet`, lo cual significa que extiende la clase `Dataset` de `PyTorch`. Este dataset genera datos binarios aleatorios a partir de distribuciones de Bernoulli. Las principales características son:

- **Características (`X`):** Una matriz de tamaño $N \times f$, donde cada entrada se genera a partir de una distribución de Bernoulli con probabilidades aleatorias.
- **Etiquetas (`Y`):** Un vector de tamaño $N \times 1$, generado de manera similar con probabilidades independientes.
- **Número de características:** Definido por el parámetro f, que determina la cantidad de columnas en texttttX.
- **Tamaño del dataset:** Controlado por el parámetro N, que especifica el número de filas.
- **Acceso a los datos:** Mediante el índice i, se puede acceder al par $(\texttt{X}[i], \texttt{Y}[i])$.

3.2. Weather Dataset

El Weather Dataset es un conjunto de datos de tamaño mediano, obtenido desde Kaggle [11], que recopila registros meteorológicos de diferentes regiones de Australia. Está diseñado para proyectos de clasificación binaria en Machine Learning, donde el objetivo es predecir si lloverá al día siguiente (`RainTomorrow`).

- **Columnas destacadas:**
  - `Date`: Fecha del registro.
  - `Location`: Ubicación geográfica.
  - `MinTemp` y `MaxTemp`: Temperatura mínima y máxima.
  - `Rainfall`: Cantidad de lluvia registrada (mm).
  - `Humidity9am` y `Humidity3pm`: Humedad relativa a las 9 AM y 3 PM.
  - `Pressure9am` y `Pressure3pm`: Presión atmosférica a las 9 AM y 3 PM.
  - `RainToday`: Indica si llovió el día actual (`Yes` o `No`).
  - `RainTomorrow`: Variable objetivo que indica si lloverá al día siguiente (`Yes` o `No`).
- **Ejemplo de datos:**

```
Date, Location, MinTemp, MaxTemp, Rainfall, RainToday, RainTomorrow
2008-12-01, Albury, 13.4, 22.9, 0.6, No, No
2008-12-02, Albury, 7.4, 25.1, 0.0, No, No
```

3.2.1. Preprocesamiento

Se realizó un preprocesamiento del `Weather Dataset` a fin de buscar consistencia en los datos. Para ello se realizaron las siguientes operaciones:

- **Carga y descripción inicial:**
  - Se carga el conjunto de datos desde un archivo CSV y se obtiene un resumen estadístico para identificar valores nulos y características importantes.
  - Se eliminan columnas irrelevantes como `Unnamed: 0` y `Date`.
- **Gestión de valores nulos:**
  - Para columnas numéricas, los valores nulos se reemplazan con la mediana de cada columna.
  - Para columnas categóricas, los valores nulos se reemplazan con el modo (valor más frecuente).
  - En la variable objetivo `RainTomorrow`, los valores nulos se reemplazan con `No` y la columna se binariza (`Yes` = 1, `No` = 0).
- **Codificación de variables categóricas:**
  - Se utiliza codificación `one-hot encoding` para convertir las variables categóricas en variables binarias, eliminando la primera categoría para evitar redundancia.
- **Normalización:**
  - Las columnas numéricas se escalan utilizando normalización estándar (*z-score*) para que tengan media 0 y desviación estándar 1, mejorando el rendimiento de los modelos.
- **División del conjunto de datos:**
  - El conjunto de datos se divide en tres subconjuntos: entrenamiento (70 %), validación (15 %) y prueba (15 %), garantizando que las proporciones de la variable objetivo se mantengan iguales en cada subconjunto (*stratified splitting*).

3.3. MNIST Dataset

El MNIST Dataset es un conjunto de datos ampliamente utilizado en tareas de clasificación y reconocimiento de dígitos escritos a mano. Contiene imágenes en escala de grises de tamaño 28x28 píxeles, cada una asociada con una etiqueta que representa un dígito del 0 al 9. Este conjunto de datos es ideal para entrenar y evaluar modelos de aprendizaje automático, especialmente redes neuronales.

- **Procesamiento del conjunto de datos:**
  - Se utiliza la biblioteca `torchvision` para descargar y procesar el conjunto de datos.
  - Las transformaciones aplicadas incluyen:
    - `ToTensor()`: Convierte las imágenes en tensores.
    - `Normalize((0.5,), (0.5,))`: Normaliza los valores de los píxeles al rango [-1, 1].
    - `Lambda(lambda x: x.view(-1))`: Aplana las imágenes de 28x28 a un vector de 784 elementos.
  - El conjunto de datos original se divide en subconjuntos de entrenamiento, validación y prueba:
    - **Entrenamiento:** 80 % de los datos originales de entrenamiento.
    - **Validación:** 20 % de los datos originales de entrenamiento.
    - **Prueba:** Conjunto de datos separado para evaluación final.
- **Columnas destacadas:** Aunque MNIST no contiene columnas como un conjunto tabular, las etiquetas de las imágenes representan los dígitos del 0 al 9.
- **Ejemplo de datos:**

```
Imagen: [tensor de 784 valores]
Etiqueta: 5
Imagen: [tensor de 784 valores]
Etiqueta: 3
```

4. Procedimiento

Se realizaron 38 experimentos etiquetados como `EXP + Número de la experiencia`. El detalle de todas las curvas de puede encontrar en el apéndice 7.1.

4.1. Neuronas por capa

Se iniciaron las pruebas con distintos valores de neuronas en las capas a fin de ver cómo se comportaban las curvas de pérdida (*loss*) y las curvas de convergencia (*accuracy*).

Tabla 1: Pruebas realizadas con diferentes configuraciones de neuronas por capa

Capa 1 (Neuro.)	**Capa 2 (Neuro.)**
400	300
600	400
800	600

4.2. Prueba de inicializaciones

Luego se realizaron pruebas iniciando el modelo con dos configuraciones de peso. Tanto la inicialización de Xavier y la inicialización de He. Se esperaba que la de Xavier rindiera mejor debido a que las capas de la red neuronal simple están compuestas por funciones de activación `tanh` y `sigmoid`.

Para lo anterior se definieron dos funciones que permitían reutilizar el código. Toman el modelo y por cada capa instanciada realizan la inicialización según la técnica definida en los pesos.

4.3. Prueba de optimizadores e inicializaciones

Con lo anterior, se realizó una serie de pruebas con distintos optimizadores, los cuales también se definieron en funciones aparte para su reutilización. Para esta prueba se tomó una red de 300 y 200 neuronas por capa respectivamente. Se iteró entre cada inicialización y a su vez por cada optimizador. Cada optimizador fue definido con una tasa de aprendizaje de 0.01.

Además, para comprobar la efectividad y el detalle de las curvas de pérdida y precisión, los optimizadores que tuvieron mejores resultados fueron nuevamente comparados con sus respectivas inicializaciones.

4.4. Prueba de regularización

En los siguientes modelos, se utilizó un optimizador SGD con momentum, definido con una tasa de aprendizaje (`lr`) de 0.005. Los pesos iniciales del modelo fueron configurados con la inicialización de He y el entrenamiento se llevó a cabo durante 20 épocas utilizando `train_loader` y `val_loader` para los conjuntos de entrenamiento y validación, respectivamente, y ejecutándose en un dispositivo `cuda`.

Se evaluó además, la implementación de Dropout, para ello se definió un modelo parecido al `SimpleFFNN` pero se agregó entre las capas aquella que realiza dropout en base a la probabilidad establecida.

Por otro lado, también se definió otro modelo para Batch Normalization, donde se agregaron entre las capas para producir la normalización al pasar a la siguiente de ellas. Se espera que tengan un efecto en disminuir la pérdida al inicio del entrenamiento.

En el siguiente experimento se buscó unir los dos anteriores. Por ende, se definió un modelo que combine las operaciones que se le realizan a las capas. Debido a la naturaleza de las técnicas, se dejó batch norm antes que dropout, para que esta última no interfiera en el cálculo de la normalización que realiza la primera.

Para la regularización L2, se utilizaron dos formas de implementar y probar su aplicación. En primera instancia, se probó agregando la regulación directamente en el cálculo de la pérdida. Mientras que existe otra forma que es utilizando `weight_decay` en los optimizadores, que realiza la actualización de los parámetros en el descenso del gradiente. Lo que implica eficiencia en el sentido de que calcula el valor de la regularización para cada peso específico [12].

Luego se unen todas las técnicas de regularización para evaluar su trabajo en conjunto. Para la L2 se utilizó `weight_decay` en un optimizador de descenso de gradiente con Nesterov.

En el EXP17 se agrega Early Stopping. Se busca detener el entrenamiento antes de alcanzar el número máximo de épocas, con el objetivo de evitar el sobreajuste. Por ende, se agregó en la iteración `train_model_loop` en conjunto a un valor `patience` que define el número máximo de épocas consecutivas en las que la pérdida de validación puede no mejorar antes de detener el entrenamiento.

4.5. Modelo Ensemble

En el experimento EXP18, se entrenó un ensamble compuesto por 8 modelos. Se busca generar modelos individuales a partir de muestras de bootstrap del conjunto de entrenamiento, donde cada muestra es procesada mediante un DataLoader independiente. Para cada modelo, se definió un modelo de una red neuronal simple, y se optimizó con SGD con Nesterov y regularización l2. Cada modelo del ensamble fue entrenado durante 20 épocas, utilizando el conjunto de validación para evaluar su desempeño. Las pérdidas de entrenamiento y validación, así como las precisiones, fueron registradas individualmente para cada modelo del ensamble, a fin de poder realizar un análisis individual.

Además, para evaluar el modelo, se construyó una función que permite promediar las predicciones de cada modelo para así generar una salida final. La precisión y la pérdida se calculan comparando estas predicciones con las etiquetas reales, permitiendo medir el desempeño general del ensamble.

4.6. Pruebas extra con cuda y cpu

A continuación, se buscó evaluar qué sucedía cuando se modificaba el *device* utilizado para entrenar los modelos. Por ende, se agregaron más muestras (10000) y se utiliza una red neuronal simple como las utilizadas anteriormente. La idea es ver el impacto con diferentes configuraciones de batch size sobre el hardware. Al principio se evalúa un tamaño batch de 128 y luego de 64. Se generan estas pruebas debido a que el conjunto de entrenamiento es dividido en *mini-batches* los cuales irán al hardware y se debe buscar cómo aprovechar este recurso adecuadamente.

5. Resultados y discusión

5.0.1. Iniciación de modelo y loop

Con las pruebas mencionadas anteriormente, se buscó las mejores configuraciones para los siguientes *datasets*. Para ello se revisarán los resultados obtenidos estableciendo una interpretación de los mismos.

Respecto a EXP1, EXP2, EXP3: En los tres experimentos, la pérdida de entrenamiento disminuye consistentemente, indicando que el modelo está aprendiendo a ajustarse a los datos de entrenamiento. EXP1 tiene una pérdida inicial más baja debido a una arquitectura más simple (menos neuronas), mientras que en EXP2 y EXP3, la pérdida inicial es más alta, probablemente debido al mayor número de parámetros, lo que dificulta la optimización inicial.

La precisión aumenta constantemente en los tres experimentos, con EXP3 alcanzando los valores más altos debido a la mayor capacidad de aprendizaje del modelo. EXP1 tiene un crecimiento más estable y menos pronunciado, reflejando la menor capacidad del modelo para aprender patrones más complejos.

No se considera la curva de la validación debido a que al ser un Random Dataset no tendrá buen rendimiento en la aleatoriedad en sus resultados.

5.0.2. Prueba de inicializaciones

Si se analizan los experimentos EXP4 y EXP5 3, entonces se puede decir que en ambos experimentos la pérdida va disminuyendo. En EXP4 (Xavier) la pérdida inical es más alta y su disminución tiende a ser más gradual lo cual puede explicarse porque este tipo de inicialización es preferente para funciones de activación `tanh` definidas en el modelo. Por otro lado, He tiene una pérdida inical más baja y converge más rápida, debido a que es una inicialización que favorece gradientes más grandes lo cual favorece el aprendizaje inicial. Respecto a la precisión del entrenamiento, ambos aumentan y en Xavier se establece en un valor ligeramente menor que HE, el cual alcanza valores más altos, lo cual se puede dar porque facilita un flujo más eficiente de los gradientes durante el aprendizaje.

5.0.3. Prueba de optimizaciones e inicializaciones

En lo que se refiere a la figura 4, se puede ver que en este experimento la inicialización He tuvo menores pérdidas en comparación con Xavier. Además, que los mejores optimizadores se encuentran SGD + Momentum y SGD Nesterov, mostrando que la incorporación de Momentum facilita una convergencia más eficiente y rápida. Lo anterior también se replica en la comparación de la precisión. Hasta el momento se quedan como la mejor combinación.

Esto también se puede ver en detalle en las figuras 5 y 6 donde el optimizador SGD + Momentum con He alcanza incluso valores de 0 al final del entrenamiento, siendo además más rápido en converger que Xavier. En la curva de precisión, se puede evidenciar cómo He llega antes al tope mientras que Xavier llega en la última época.

Mientras que Xavier mantiene los gradientes estables al inicio del entrenamiento, He ajusta los valores iniciales de los pesos para que los gradientes sean más grandes, permitiendo que el modelo aprenda más rápido y alcance una mejor convergencia. Al combinar He con optimizadores como Nesterov, que anticipa la dirección del gradiente antes de actualizar los parámetros, se evita movimientos innecesarios, lo anterior permite una mayor precisión y estabilidad en las actualizaciones. Esto hace que la combinación de He y Nesterov sea más rápida y eficiente, maximizando la capacidad del modelo para minimizar la pérdida y alcanzar mejores resultados.

5.0.4. Prueba de regularizaciones

Respecto a las regularizaciones implementadas las cuales se pueden observar en 10 y 11, en Dropout la pérdida se mantiene sostenida pero no es mejor que cuando se utiliza Batch Normalization por sí sola llegando a una pérdida de 0.2403 y precisión del 98 %. Esto también se ve reflejado en las curvas de precisión, donde Dropout incorpora desorden a la curva y puede no ser beneficio en particular para este conjunto de datos, a diferencia de Batch Norm que ayuda con la normalización de las activaciones estabilizando la curva y llegando a buenos resultados. En ese sentido es necesario analizar el tipo de datos sobre los cuales se trabaja para ver qué técnicas se acomodan mejor. En este caso, Dropout introduce un pequeño sacrificio en el ajuste del modelo para mejorar la generalización.

Con respecto a la regularización de tipo L2 que se muestra en las figuras 12 y 13, cuando la regularización L2 se suma explícitamente a la función de pérdida se puede ver una curva con una disminución constante. Pero parte desde valores de pérdida sobre 1.5 y disminuye hasta un poco antes que 0.6. A diferencia de `weight_decay` qie comienza bajo el 0.8 y sigue disminuyendo hasta casi 0. Eso también se demuestra en las curvas de precisión donde `weight_decay` llega antes al tope. Lo anterior puede explicarse porque la implementación de l2 en la librería implica regularizar directamente cada peso en cada paso del optimizador sin interferir con el gradiente de la función de pérdida.

Luego, se probó combinar Batch Normalization, Dropout y L2 con `weight_decay` (EXP16). Dropout sigue afectando la tendencia de la curva, aunque sí disminuye de manera constante obteniendo mejores resultados que cuando se aplicó Dropout sin L2. Si bien Dropout agrega ruido, L2 penaliza los valores grandes de los pesos, por ende se podría decir que actúa como una fuerza regularizadora adicional que suaviza estas fluctuaciones y evita que los pesos crezcan de forma descontrolada.

Finalmente se agrega Early Stopping y si bien funciona, no tiene mucha incidencia en un conjunto de datos donde la validación no corresponderá debido a que no tiene un comportamiento predecible entre el conjunto de entrenamiento y validación.

5.0.5. Prueba de bootstrap

A continuación, se busca analizar los resultados obtenidos desde la figura 18, los cuales son curvas de pérdida sobre el entrenamiento. Mostrando un buen descenso en la pérdida. Incluso si se valida el modelo *ensemble*, obtiene un 46,53 %, lo cual responde a la aleatoriedad del conjunto y se deberá validar con otro modelo para confirmar un correcto desempeño de esta implementación.

5.0.6. Pruebas sobre CUDA y CPU

En relación a la 17 se puede observar que ambas curvas decaen correcamente, siendo GPU levemente más fluída, principalmente por su facilidad para realizar operaciones en paralelo. En relación a los tiempos, la CPU comienza desde un punto menor que la GPU (probablemente por demora en la carga de los datos en el hardware), sin embargo, estos casi 0.2 segundos se mantienen hasta la época 10, lo cual podría ser por la naturaleza de los datos en la CPU y cómo se tiene que ir optimizando para estabilizarse. La GPU se estabiliza cercano a los 0.10 segundos mucho antes.

En relación a las curvas de aprendizaje, entre 128 y 64 de batch tiene un tema respecto a la cantidad de pérdida y precisión alcanzada. Mientras

6. Referencias bibliográficas

[1] Glorot, X., & Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics* (pp. 249-256). PMLR. Disponible en: http://proceedings.mlr.press/v9/glorot10a.html

[2] He, K., Zhang, X., Ren, S., & Sun, J. (2015). Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification. In *Proceedings of the IEEE International Conference on Computer Vision* (pp. 1026-1034). Disponible en: https://arxiv.org/abs/1502.01852

[3] LeCun, Y., Bottou, L., Orr, G. B., & Müller, K. R. (1998). Efficient backprop. In *Neural networks: Tricks of the trade* (pp. 9-50). Springer. Disponible en: https://link.springer.com/chapter/10.1007/3-540-49430-8_2

[4] Ioffe, S., & Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning* (pp. 448-456). PMLR. Disponible en: https://arxiv.org/abs/1502.03167

[5] Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep Learning. MIT Press. Disponible en: https://www.deeplearningbook.org/

[6] Ruder, S. (2016). An overview of gradient descent optimization algorithms. Disponible en: https://arxiv.org/abs/1609.04747

[7] Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. Disponible en: https://arxiv.org/abs/1412.6980

[8] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1), 1929-1958. Disponible en: http://jmlr.org/papers/v15/srivastava14a.html

[9] Ioffe, S., & Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. En *Proceedings of the 32nd International Conference on Machine Learning* (pp. 448-456). PMLR. Disponible en: https://arxiv.org/abs/1502.03167

[10] Ng, A. (2004). Feature selection, L_1 vs. L_2 regularization, and rotational invariance. En *Proceedings of the 21st International Conference on Machine Learning* (pp. 78-85). Disponible en: https://ai.stanford.edu/~ang/papers/icml04-l1l2.pdf

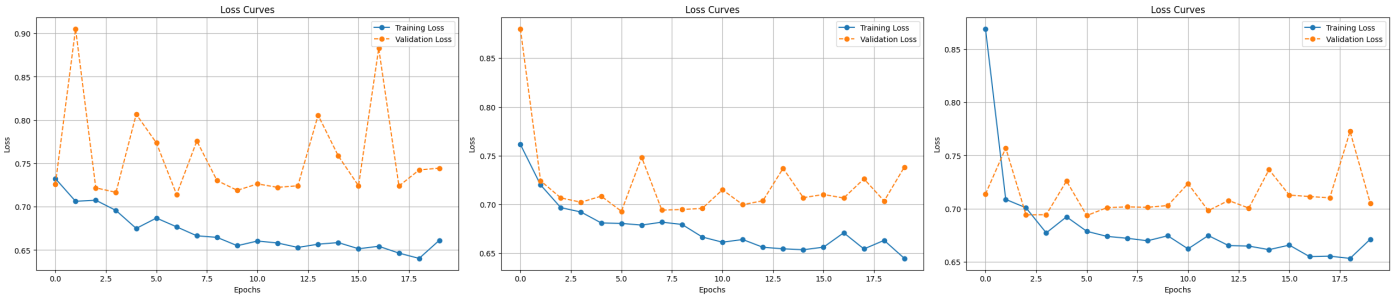
[11] Rever3nd. (2023). Weather Data. Disponible en: https://www.kaggle.com/datasets/rever3nd/weather-data/data

[12] PyTorch. (n.d.). Stochastic Gradient Descent (SGD). Disponible en: https://github.com/pytorch/pytorch/blob/main/torch/optim/sgd.py

7. Apéndices

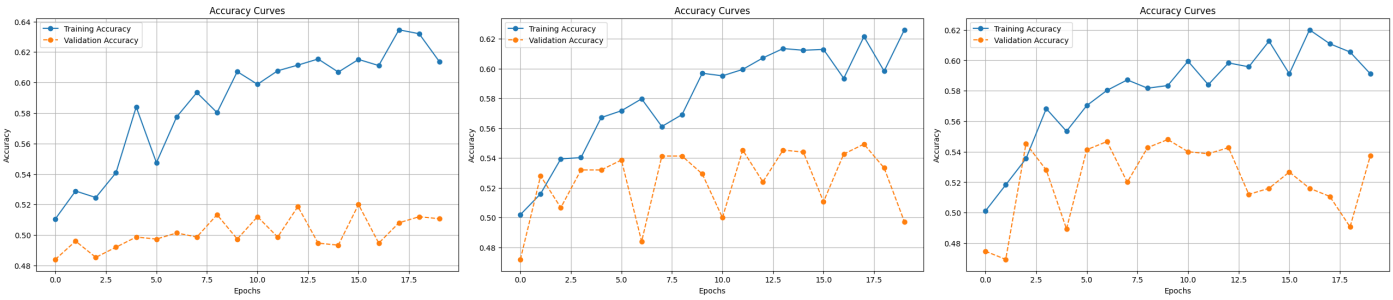
7.1. Resultados

7.1.1. Iniciación de modelo y loop



EXP1 Loss - 400/200 EXP2 Loss - 600/400 EXP3 Loss - 800/600

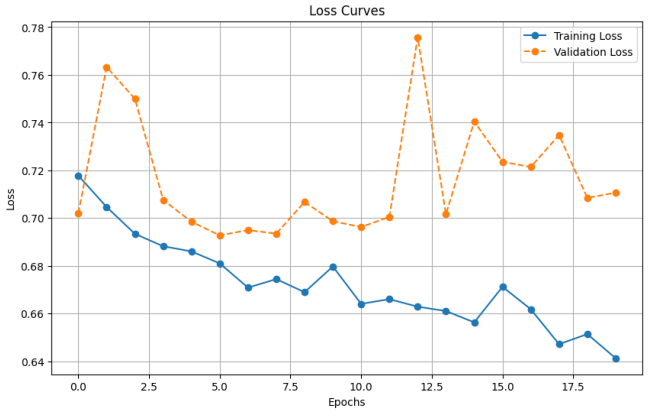
Figura 1: Curva de pérdida para EXP1, EXP2, EXP3.



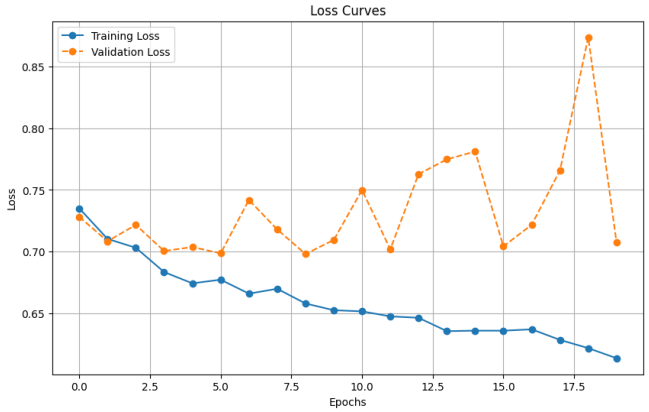
EXP1 Acc - 400/200 EXP2 Acc - 600/400 EXP3 Acc - 800/600

Figura 2: Curva de precisión para EXP1, EXP2, EXP3.

7.1.2. Prueba de inicializaciones

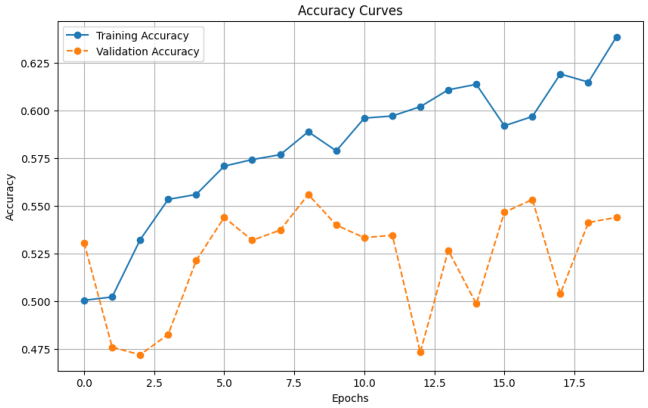


EXP4 Loss - Xavier

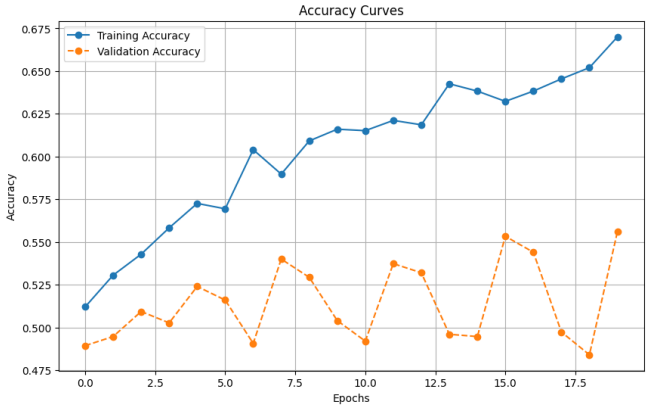


EXP5 Loss - He

Figura 3: Curva de pérdida para EXP4, EXP5.



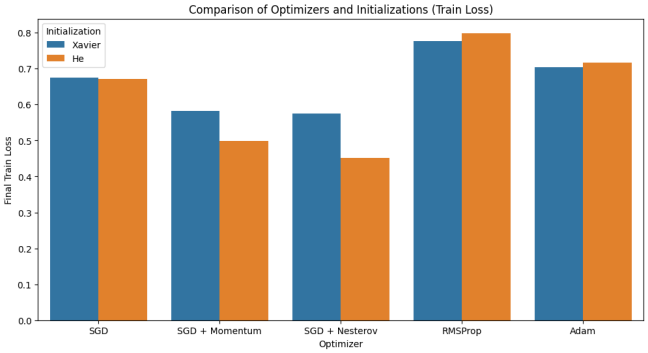
EXP4 Acc - Xavier



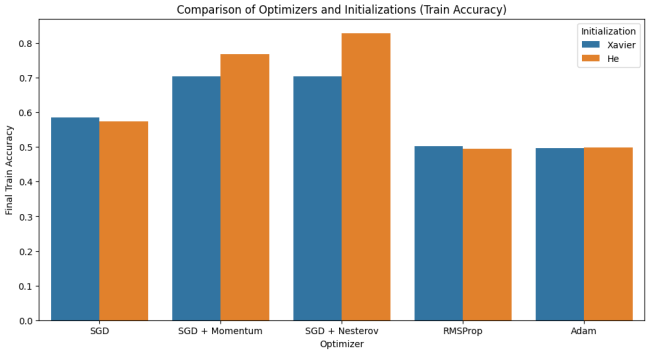
EXP5 Acc - He

Figura 4: Curva de precisión para EXP4, EXP5.

7.1.3. Prueba de optimizadores



EXP4 Loss

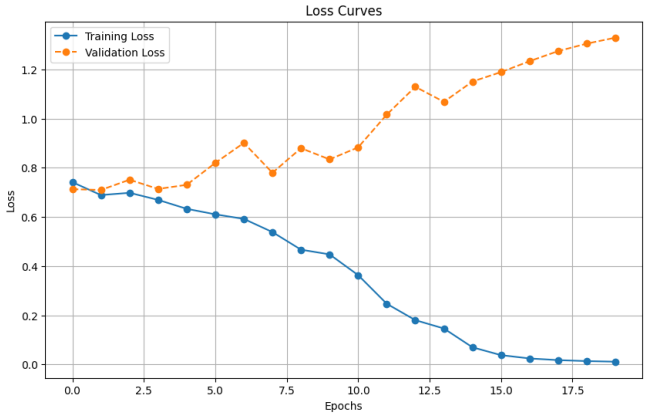


EXP5 Acc

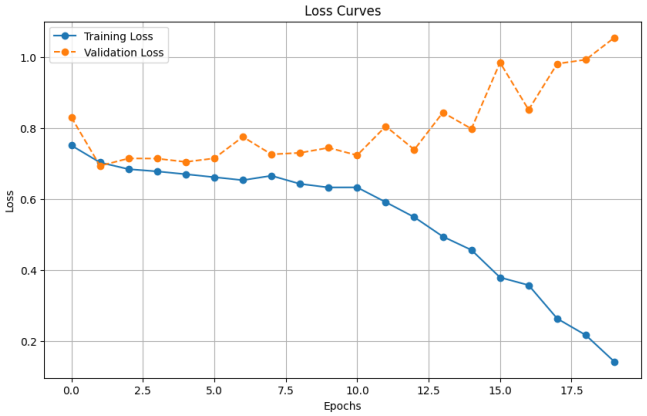
Figura 5: Curva de precisión para EXP6.

7.1.4. Prueba de optimizadores e inicializaciones

Para descenso del gradiente con Momentum:

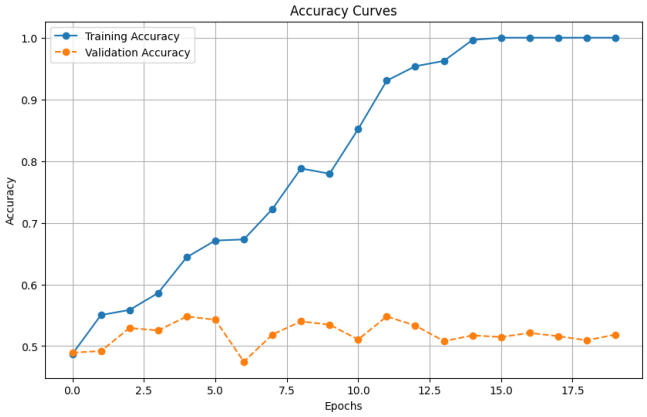


EXP7 Loss - He

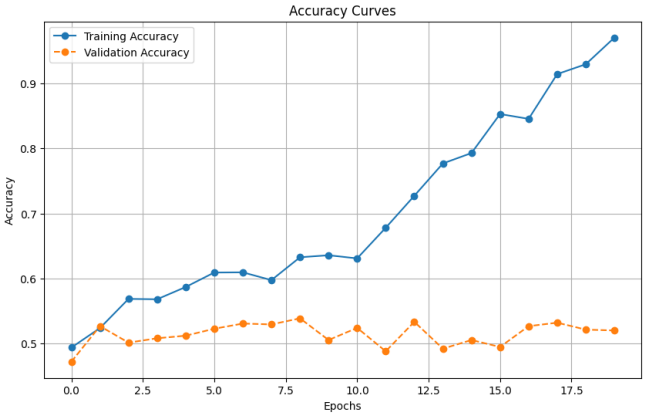


EXP8 Loss - Xavier

Figura 6: Curva de pérdida para EXP7 y EXP8.



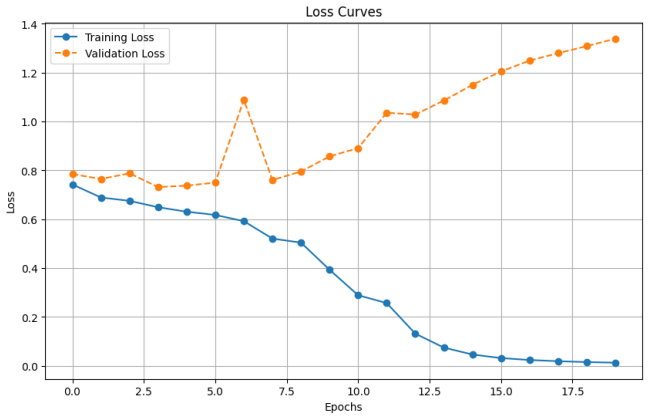
EXP7 Acc - He



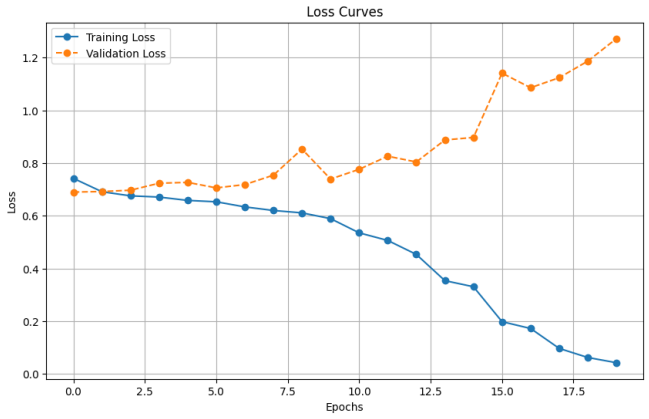
EXP8 Acc - Xavier

Figura 7: Curva de precisión para EXP7 y EXP8.

Para descenso del gradiente con Nesterov:

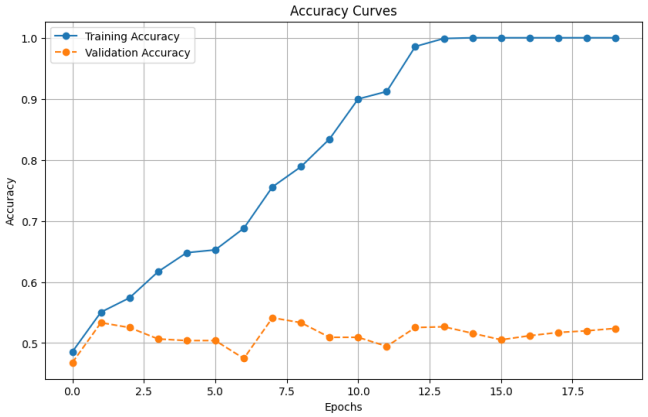


EXP9 Loss - He

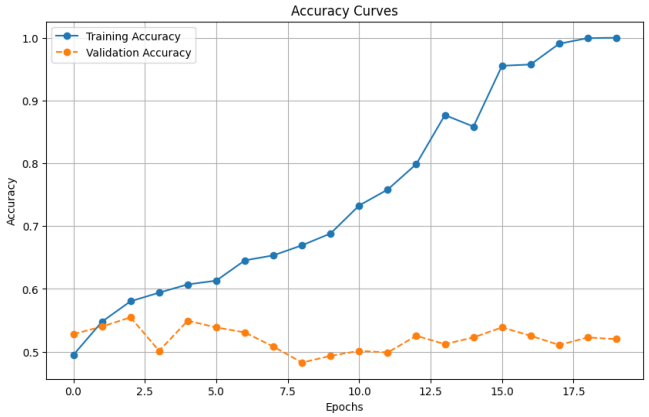


EXP10 Loss - Xavier

Figura 8: Curva de pérdida para EXP9 y EXP10.



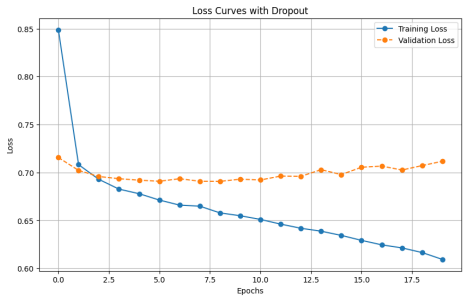
EXP9 Acc - He



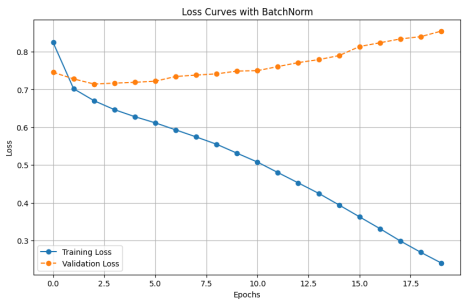
EXP10 Acc - Xavier

Figura 9: Curva de precisión para EXP9 y EXP10.

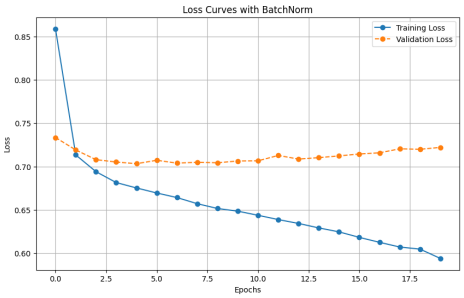
7.1.5. Prueba de regularización



EXP11 Loss - Dropout

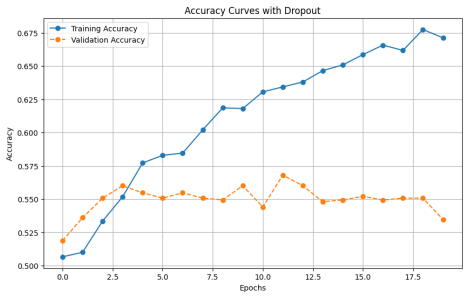


EXP12 Loss - Batch Normalization

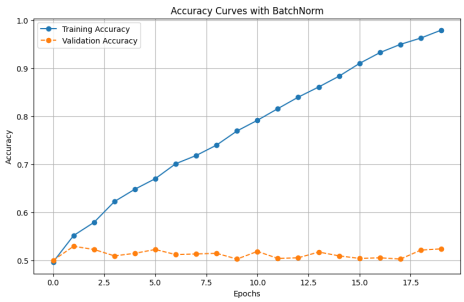


EXP13 Loss - Batch Normalization con Dropout

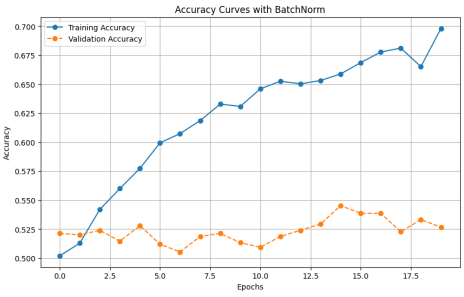
Figura 10: Curva de pérdida para EXP11, EXP12, EXP13.



EXP11 Acc - Dropout



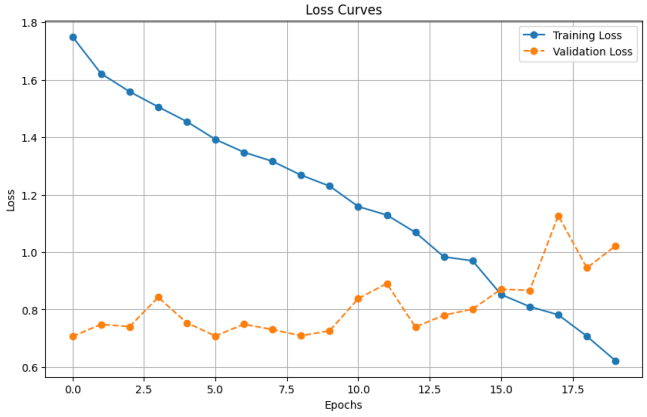
EXP12 Acc - Batch Normalization



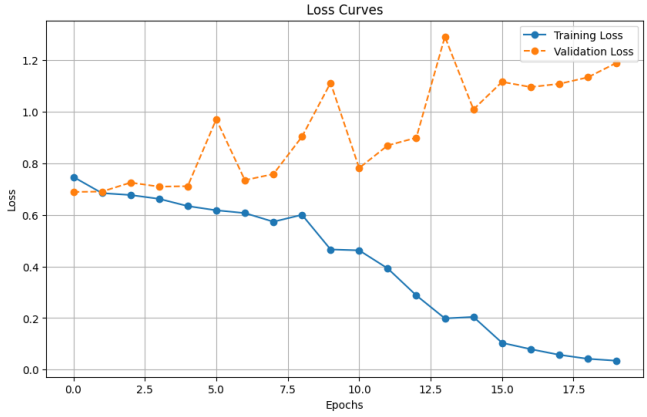
EXP13 Acc - Batch Normalization con Dropout

Figura 11: Curva de presición para EXP11, EXP12, EXP13.

Para regularización de tipo L2:

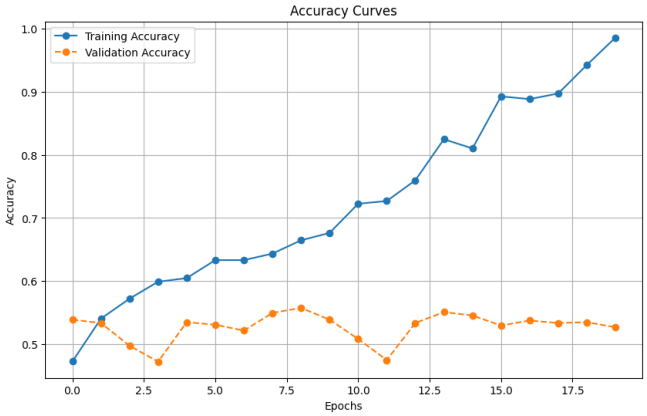


EXP14 Loss

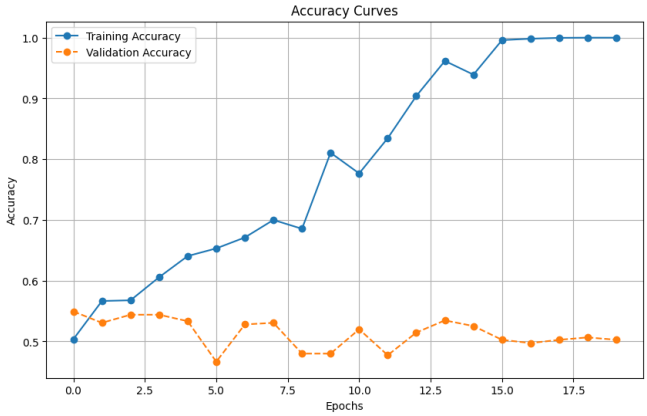


EXP15 Loss

Figura 12: Curva de pérdida para EXP14 y EXP15.



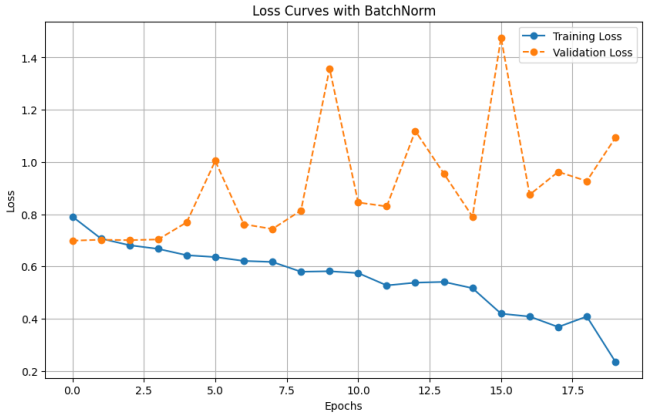
EXP14 Acc



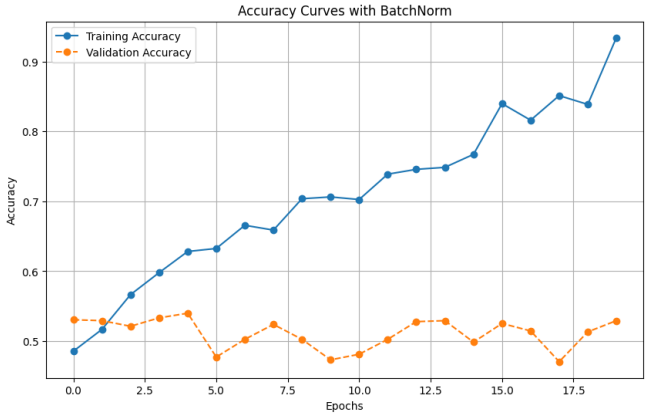
EXP15 Acc

Figura 13: Curva de precisión para EXP14 y EXP15.

Todas las regularizaciones juntas:



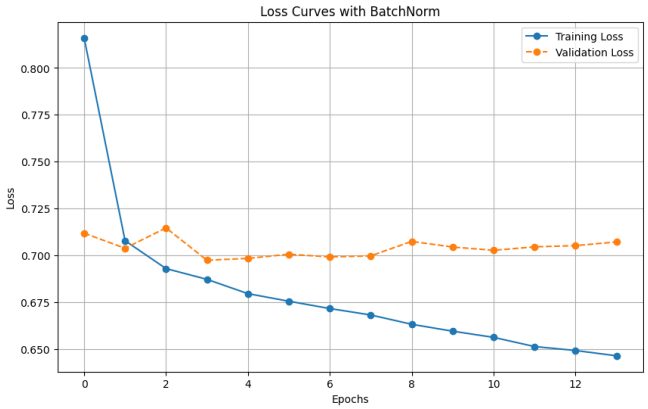
EXP16 Loss



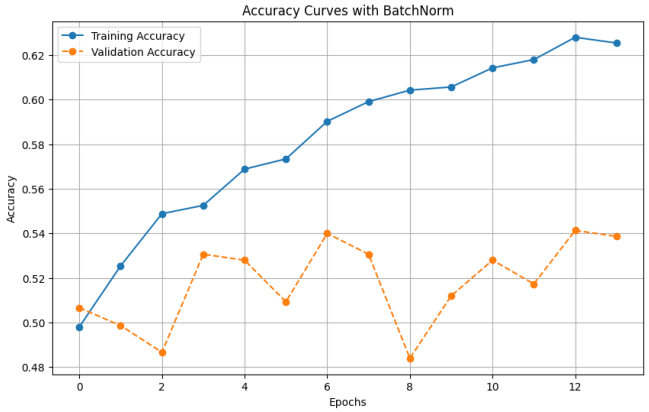
EXP16 Acc

Figura 14: Curva de precisión para EXP16.

Agregando Early Stopping:



EXP17 Loss



EXP17 Acc

Figura 15: Curva de precisión para EXP17.

7.1.6. Prueba de modelo ensamble

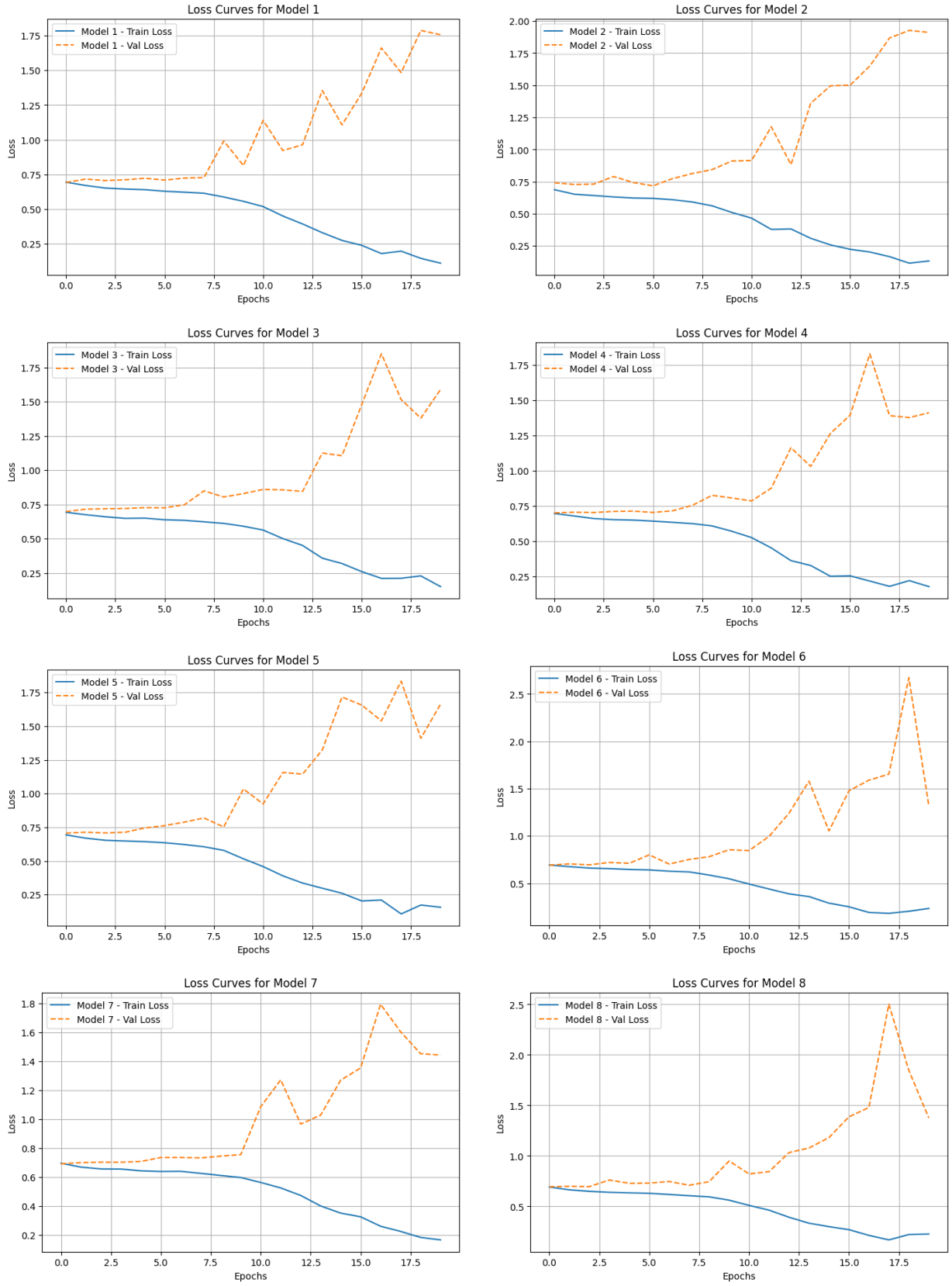
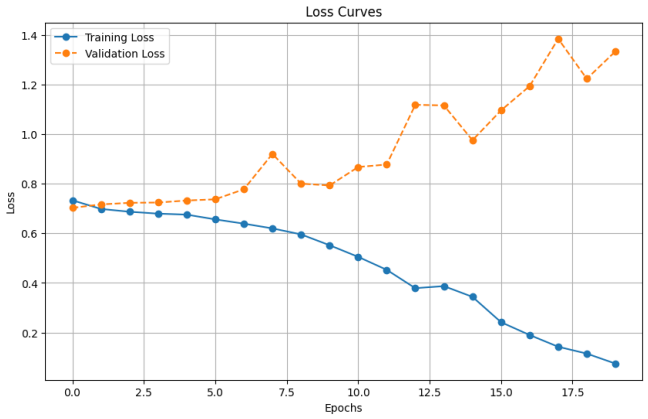
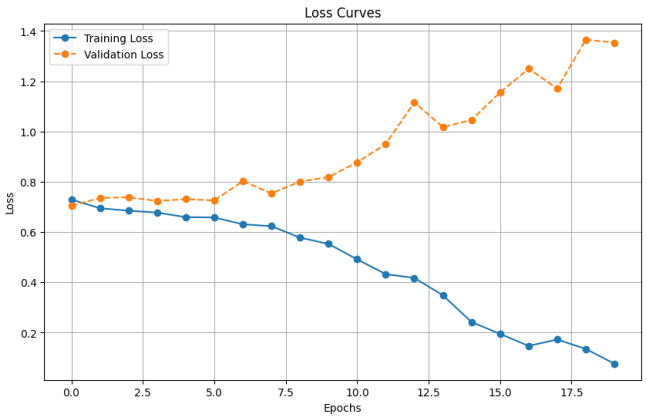


Figura 16: Gráficos de cada modelo del ensamble.

7.1.7. Prueba entre CUDA y CPU

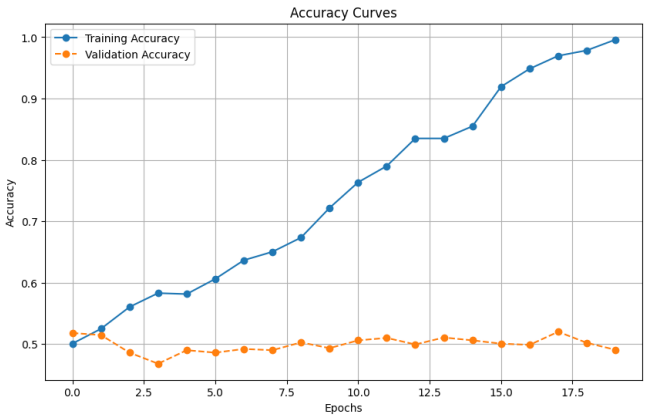


EXP19 Loss - CUDA/128

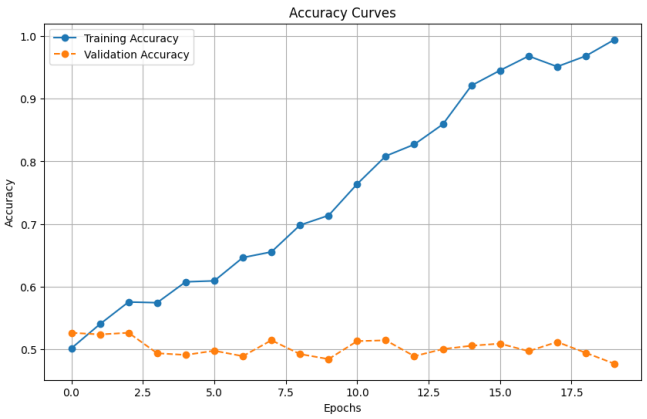


EXP20 Loss - CPU/128

Figura 17: Curva de pérdida para EXP19 y EXP20.

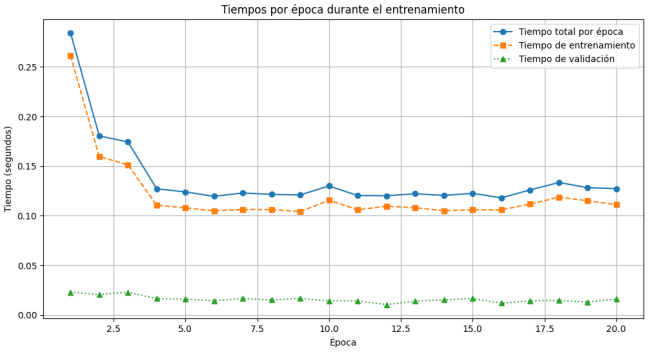


EXP19 Acc - CUDA/128

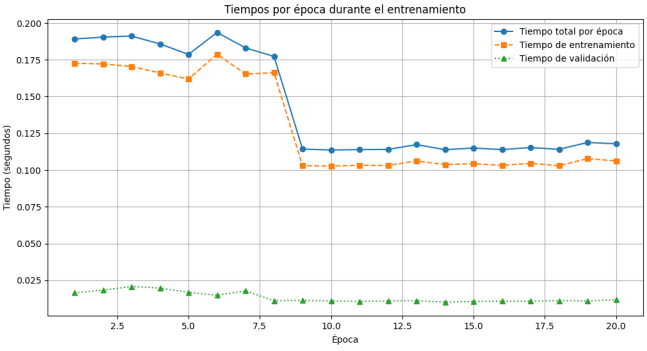


EXP20 Acc - CPU/128

Figura 18: Curva de precisión para EXP19 y EXP20.

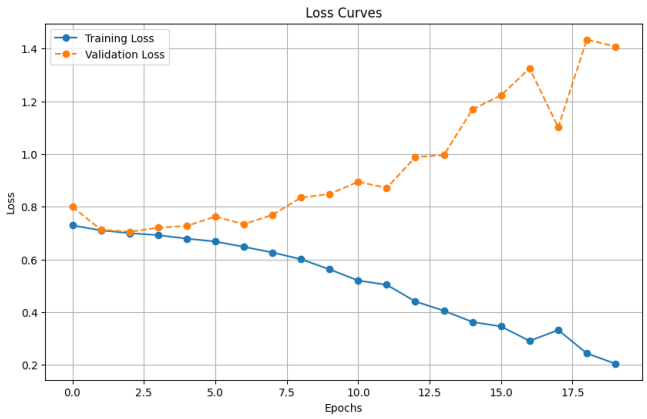


EXP19 Time - CUDA/128

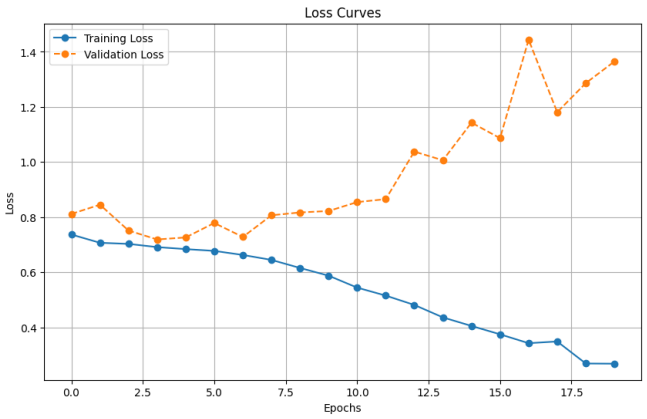


EXP20 Time - CPU/128

Figura 19: Curva de tiempo para EXP19 y EXP20.

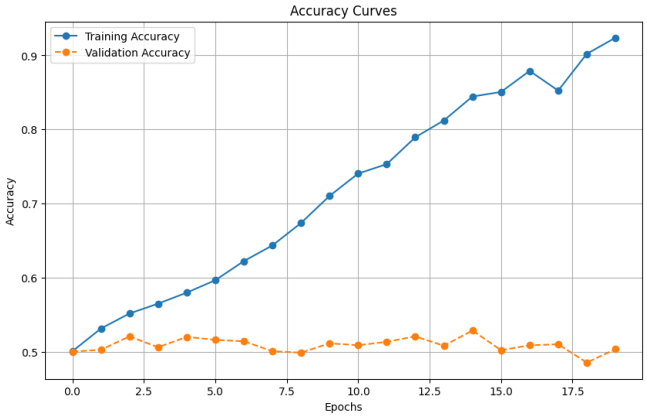


EXP21 Loss - CPU/64

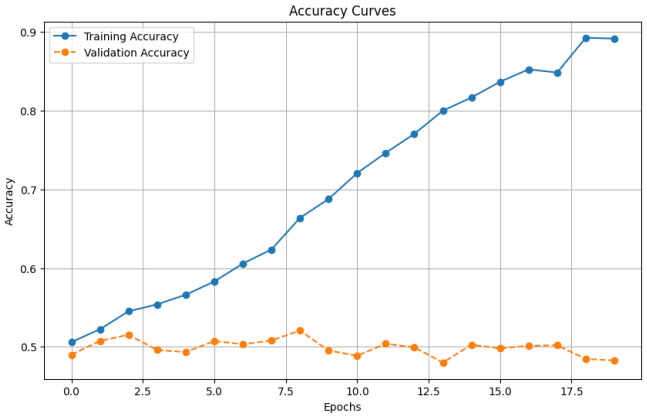


EXP22 Loss - CUDA/64

Figura 20: Curva de pérdida para EXP19 y EXP20.

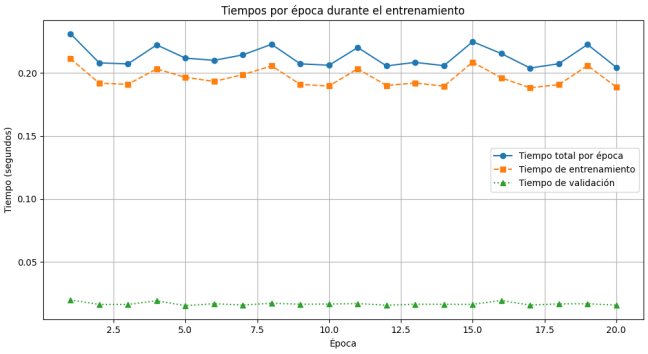


EXP21 Acc - CPU/64

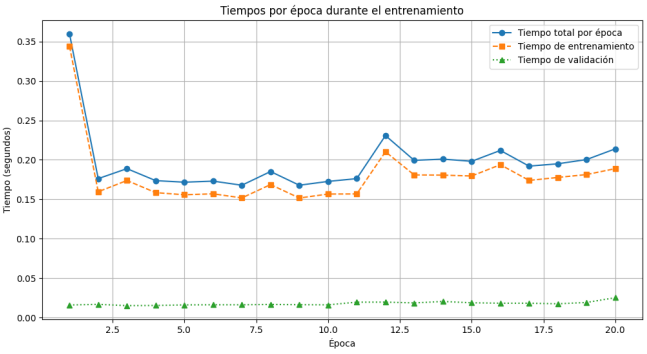


EXP22 Acc - CUDA/64

Figura 21: Curva de precisión para EXP21 y EXP22.



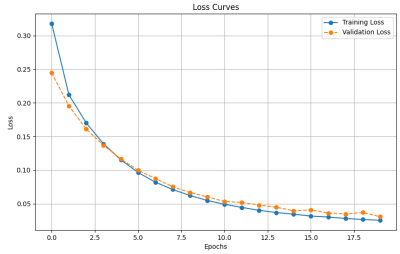
EXP21 Time - CPU/64



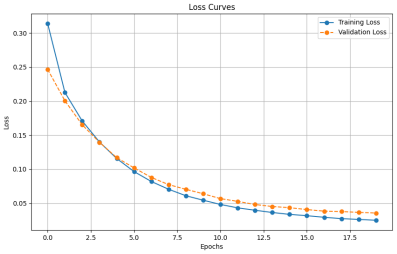
EXP22 Time - CUDA/64

Figura 22: Curva de tiempo para EXP21 y EXP22.

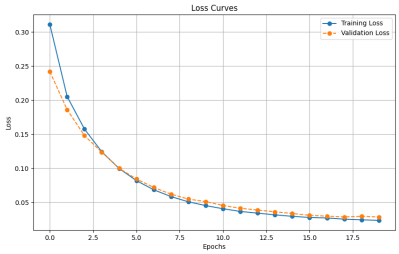
7.1.8. Weather Dataset



EXP23 Loss - SimpleFFNN

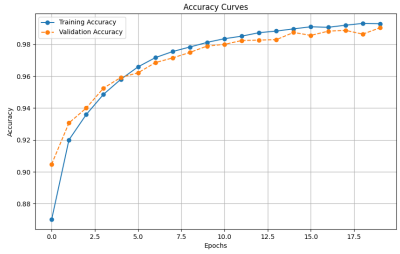


EXP24 Loss - Early Stopping

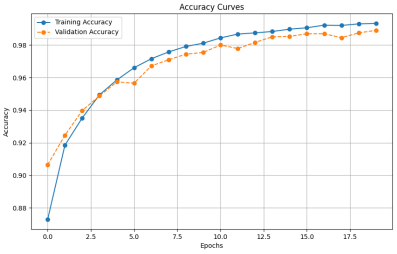


EXP25 Loss - Init Xavier

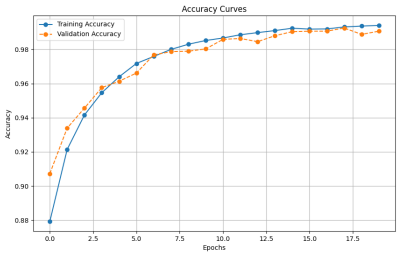
Figura 23: Curva de pérdida para EXP23, EXP24, EXP25.



EXP23 Acc - SimpleFFNN



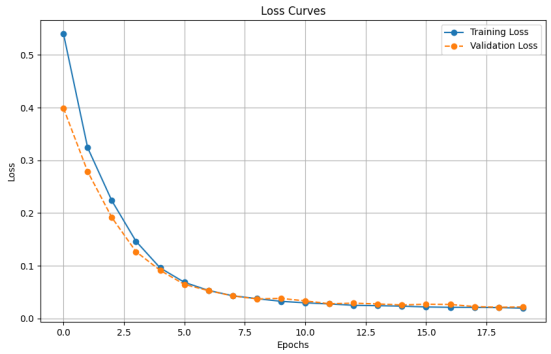
EXP24 Acc - Early Stopping



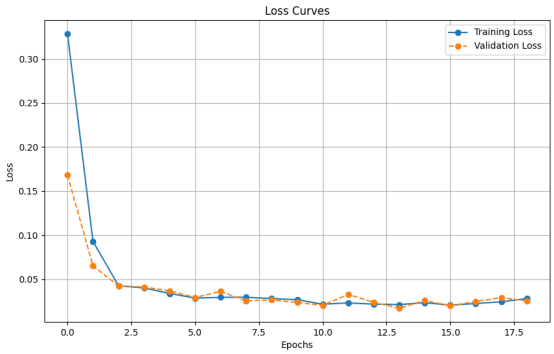
EXP25 Acc - Init Xavier

Figura 24: Curva de precisión para EXP23, EXP24, EXP25.

7.1.9. Weather Dataset - Dropout y Batch Normalization

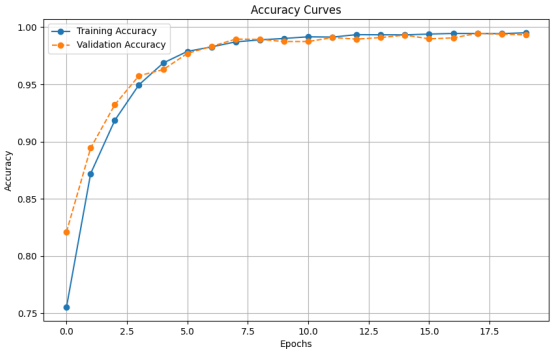


EXP26 Loss

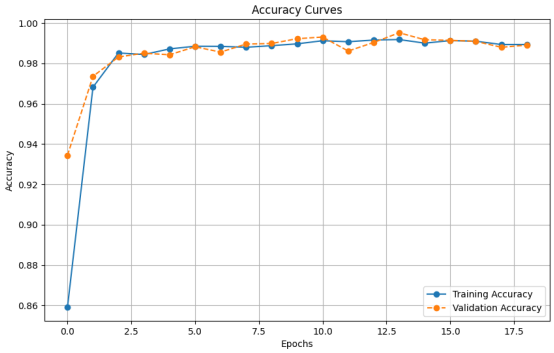


EXP27 Loss

Figura 25: Curva de pérdida para EXP26 y EXP27.



EXP26 Acc



EXP27 Acc

Figura 26: Curva de precisión para EXP26 y EXP27.

7.1.10. Weather Dataset - Modelo Ensemble

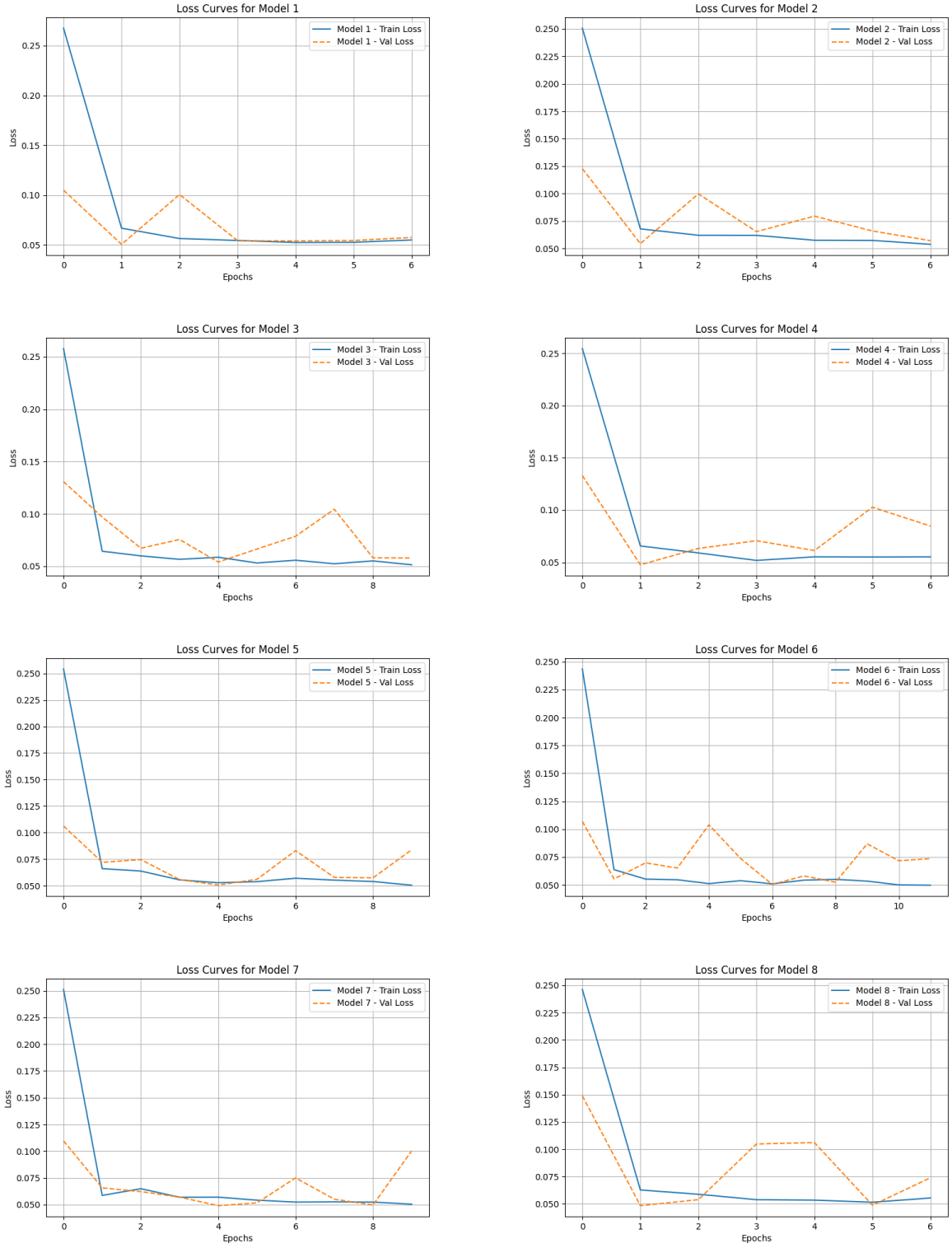
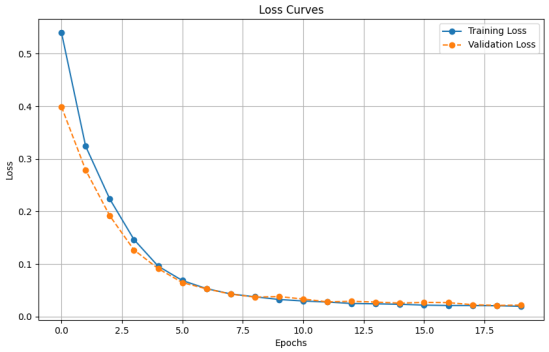
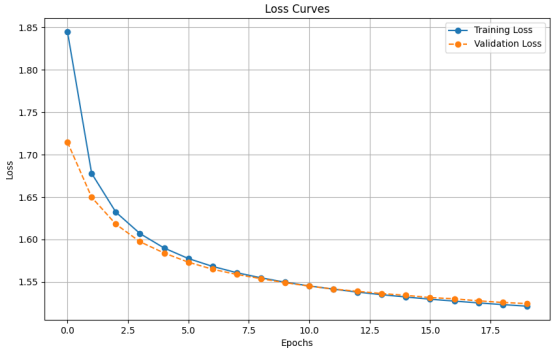


Figura 27: Gráficos de cada modelo del ensamble de EXP26-ENS.

7.1.11. MNIST Dataset - Modelo simple

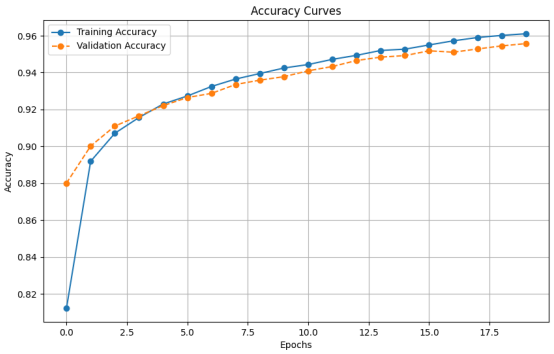


EXP26 Loss - Batch 64

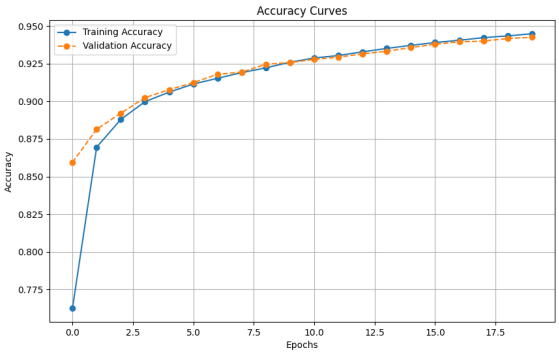


EXP29 Loss - Batch 128

Figura 28: Curva de pérdida para EXP28 y EXP29.



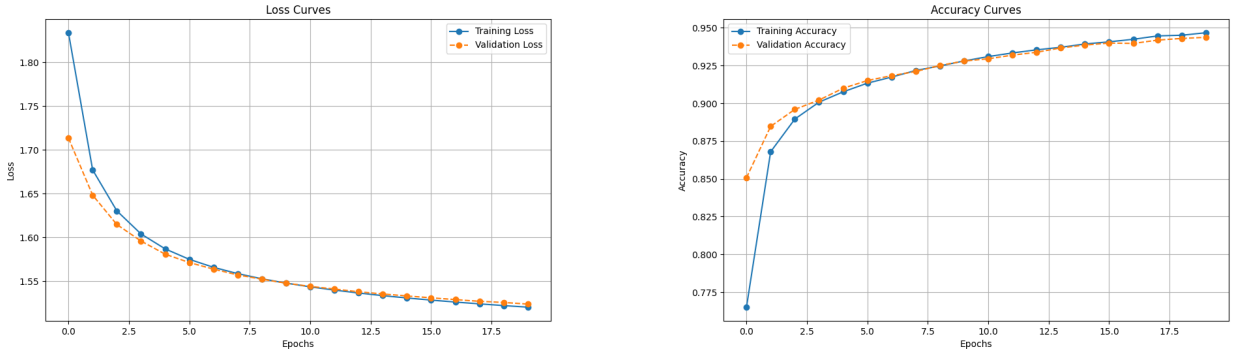
EXP28 Acc - Batch 64



EXP29 Acc - Batch 128

Figura 29: Curva de precisión para EXP28 y EXP29.

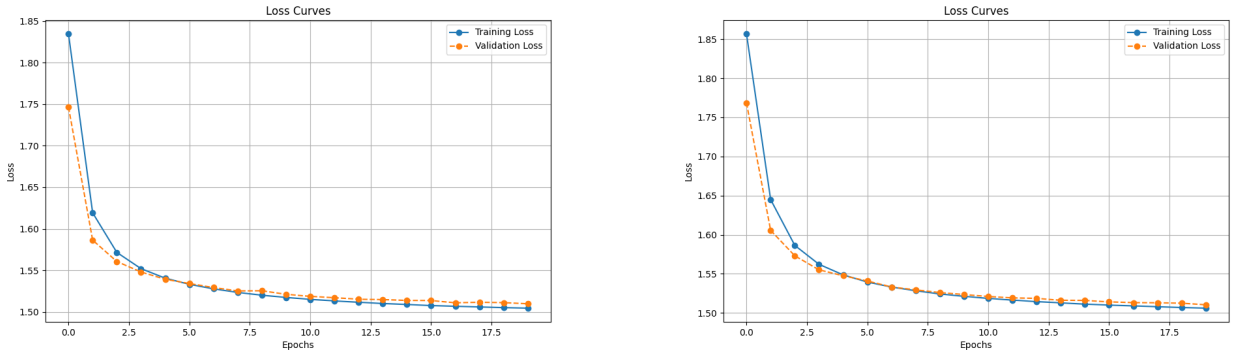
7.1.12. MNIST Dataset - Init He



EXP30 Loss EXP30 Acc

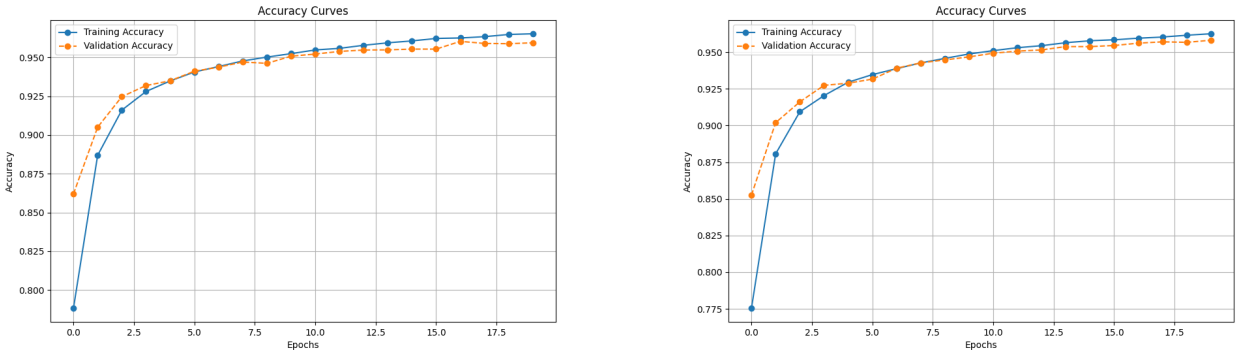
Figura 30: Curva de precisión para EXP17.

7.1.13. MNIST Dataset - Batch Normalization



EXP31 Loss - 200/100 EXP32 Loss - 50/100

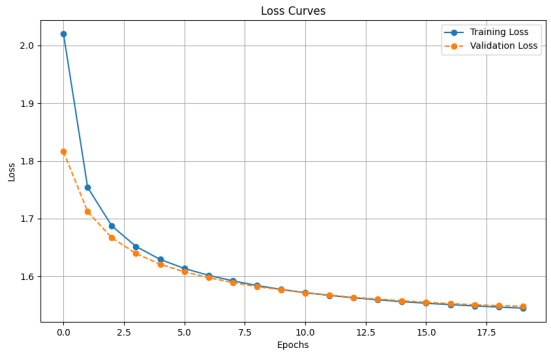
Figura 31: Curva de pérdida para EXP31 y EXP32.



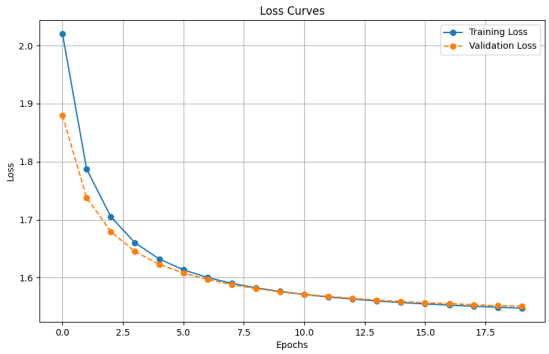
EXP31 Acc - 200/100 EXP32 Acc - 50/100

Figura 32: Curva de precisión para EXP31 y EXP32.

7.1.14. MNIST Dataset - Batch Normalization y Dropout

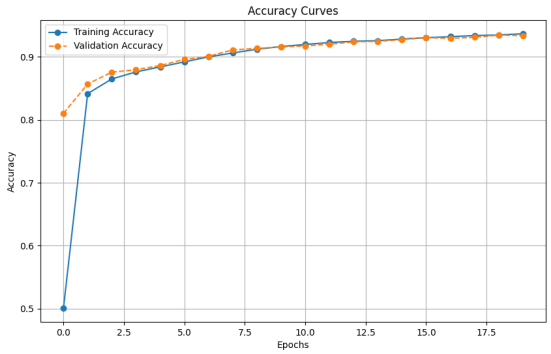


EXP33 Loss - Dropout

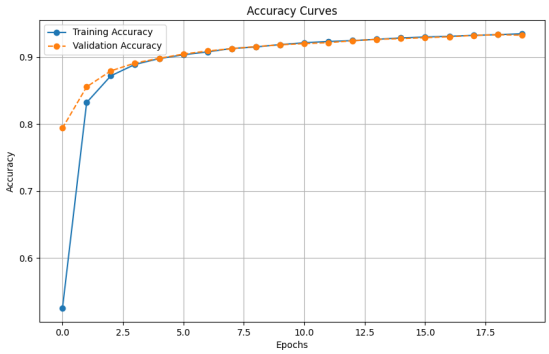


EXP34 Loss - Batch Norm Dropout

Figura 33: Curva de pérdida para EXP33 y EXP34.



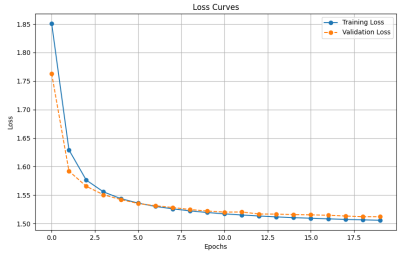
EXP33 Loss - Dropout



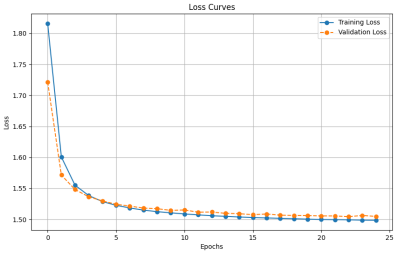
EXP34 Loss - Batch Norm Dropout

Figura 34: Curva de precisión para EXP33 y EXP34.

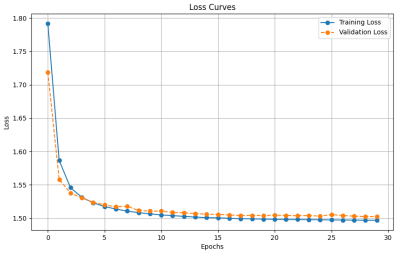
7.1.15. MNIST Dataset - Número de Neuronas



EXP36 Loss - 120/60

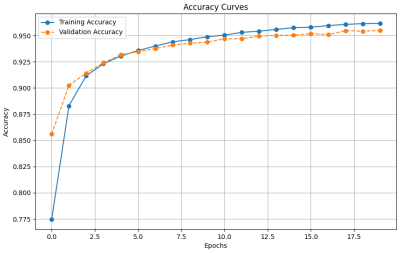


EXP37 Loss - 200/100

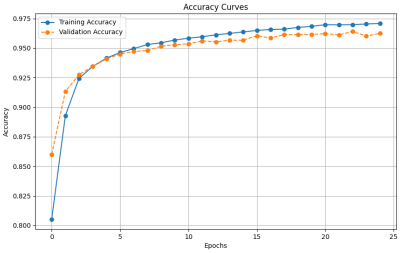


EXP37 Loss - 600/300

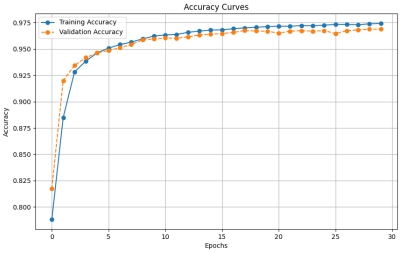
Figura 35: Curva de pérdida para EXP36, EXP37, EXP38.



EXP36 Acc - 120/60



EXP37 Acc - 200/100



EXP38 Acc - 600/300

Figura 36: Curva de precisión para EXP36, EXP37, EXP38.

7.1.16. MNIST Dataset - Modelo Ensemble

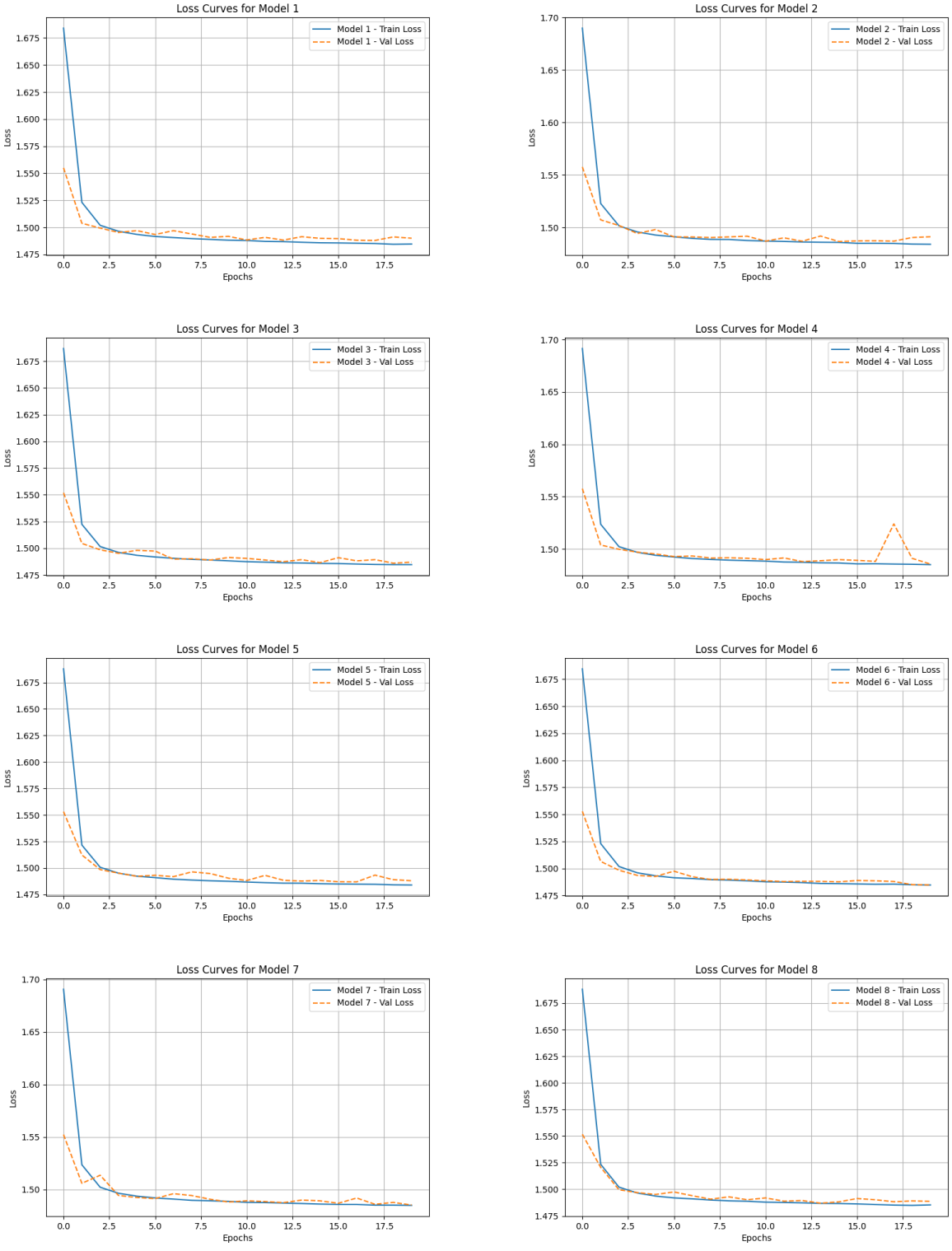


Figura 37: Gráficos de cada modelo del ensamble de EXP38-ENS.