"Implementation of an MLP architecture using Keras "

Deep Learning - Lab 03

MEKKI Meriem
MESSIOUD Mohammed Amir

ESI SBA - March 2024

Summary

Table of figures

1. Introduction

The landscape of deep learning is continually shaped by the versatility of neural networks, particularly Multi-Layer Perceptrons (MLPs), in image classification tasks. This lab centers around the comprehensive training of an MLP model using Keras.
A crucial component of this exploration is the optimization of hyperparameters, a critical phase in refining model performance.
This report unfolds the journey from the fundamental training of an MLP on MNIST to its application on CIFAR-10, ultimately delving into the nuances of hyperparameter optimization. The goal is to elucidate the efficacy of MLPs in image classification scenarios and underscore the pivotal role played by hyperparameter tuning in the pursuit of optimal model outcomes.

2. Part I: Implementation of an MLP architecture using Keras on MNIST dataset

In the initial phase of the lab, our focus centered on implementing a Multi-Layer Perceptron (MLP) architecture using Keras on the MNIST dataset.

The process began with the acquisition and splitting of the MNIST dataset into training, validation and testing sets, this tripartite division was crucial for effective model development and assessment. Subsequently, data normalization was applied to ensure consistent model training.

The heart of the implementation involved defining the MLP architecture, experimenting with various configurations to assess their impact on performance. The model was then trained on the training set, and performance metrics such as accuracy and loss were meticulously monitored and plotted to visualize the training progress.

A pivotal aspect of this phase was the exploration of hyperparameter variations, including optimizers, batch sizes, and learning rates, leading to the comparison of models to identify optimal configurations.

Notably, the testing set, distinct from the validation set, was reserved to rigorously evaluate the predictions of the best-chosen model, providing a critical assessment of its real-world performance.

This foundational exploration lays the groundwork for subsequent experiments, offering valuable insights into the intricate relationship between architecture design and hyperparameter choices in deep learning models.

2.1. Data Loading

2.2. Data Splitting : Training set , Validation set, Testing set

2.3. Data Visualization: using matplotlib.pyplot

2.4. Data Normalization:

we divided the pixel values by 255, rescaling the data to a normalized scale ranging from 0 to 1.

2.5. Designing the MLP architecture:

One input layer, two hidden layers (128, 64 neurons) and one fully connected layer. The RELU activation function for hidden layers and the softmax for the fully connected layer. Use The cross-entropy loss function.

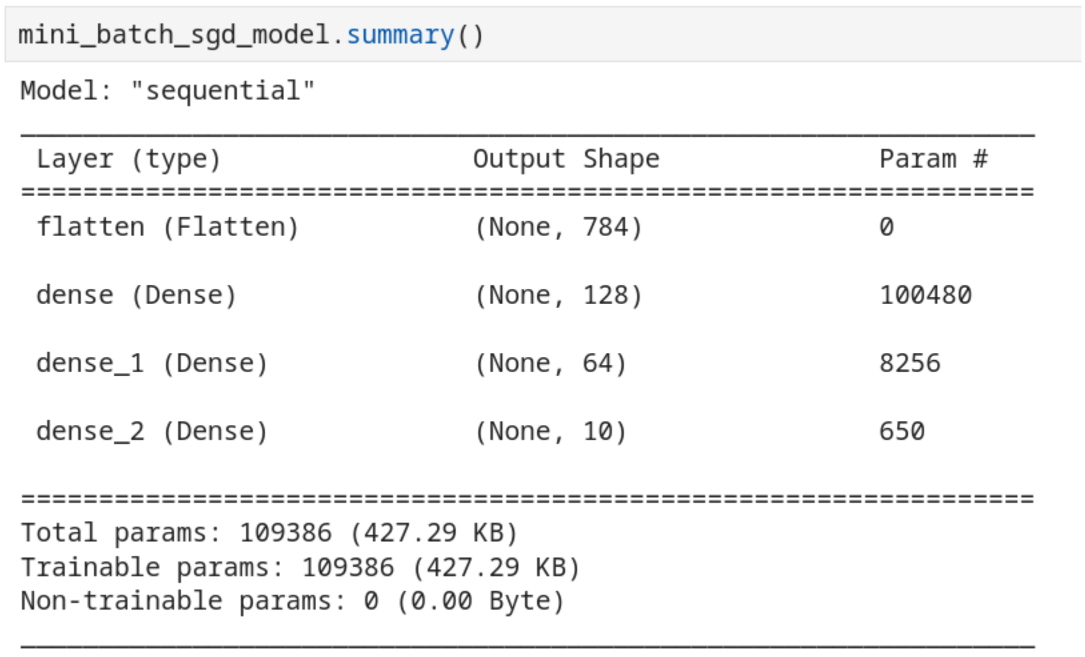
```python
mini_batch_sgd_model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(784,)),
  tf.keras.layers.Dense(128, activation='relu'),
  tf.keras.layers.Dense(64, activation='relu'),
  tf.keras.layers.Dense(10,  activation='softmax')
])
```

Listing 1: Architecture

```python
mini_batch_sgd_model.compile(
    optimizer=tf.keras.optimizers.SGD(learning_rate=0.01),
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)
```

Listing 2: compilation process

model.summary() can be used the check the architecture



```
mini_batch_sgd_model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten (Flatten)           (None, 784)               0

 dense (Dense)               (None, 128)               100480

 dense_1 (Dense)             (None, 64)                8256

 dense_2 (Dense)             (None, 10)                650

=================================================================
Total params: 109386 (427.29 KB)
Trainable params: 109386 (427.29 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Figure 1: Model architecture

2.6. Model Training

We first trained the designed architecture based on SGD, then Mini batch SGD (64 the size batch), then batch SGD in 50 epochs and set the learning rate to 0.01. After that we Plot the learning curves for each strategy (loss and accuracy).

Here are the results we obtained for each strategy:

2.6.1. Mini Batch SGD (batch size = 64)

Epoch 50/50 loss: 0.0511 - accuracy: 0.9866 - val_loss: 0.0935 - val_accuracy: 0.9716

Running time : 184.04779696464539 seconds

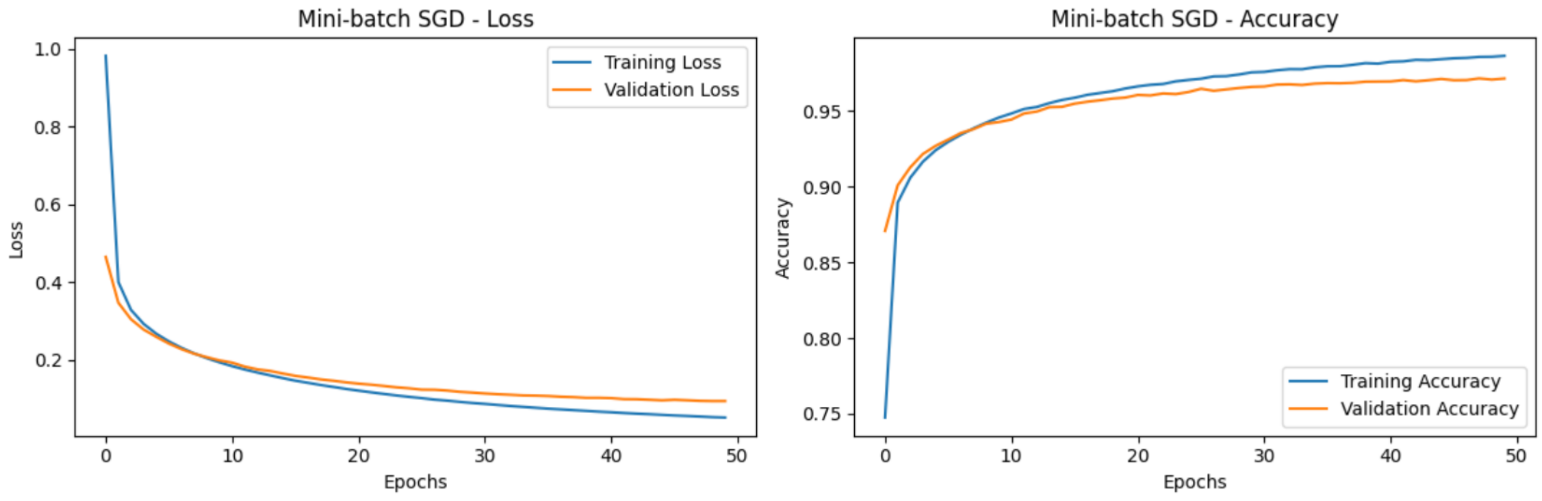


Figure 2: Loss and Accuracy for Mini Batch SGD strategy

2.6.2. Batch SGD (entire training set)

Epoch 50/50 loss: 0.0170 - accuracy: 0.9970 - val_loss: 0.0897 - val_accuracy: 0.9731

Running time : 338.01215839385986 seconds

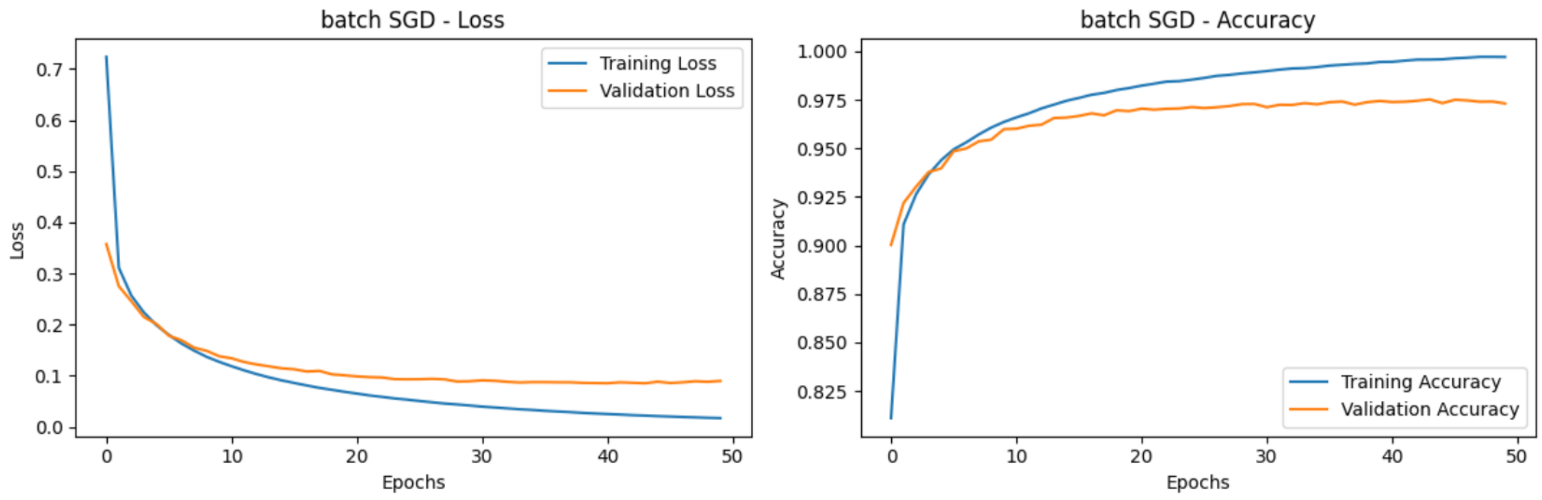


Figure 3: Loss and Accuracy for Batch SGD strategy

2.6.3. SGD (on one random sample)

Epoch 50/50 - loss: 1.3706e-05 - accuracy: 1.0000 - val_loss: 0.1369 - val_accuracy: 0.9830

Running time: 10324.142413377762 seconds

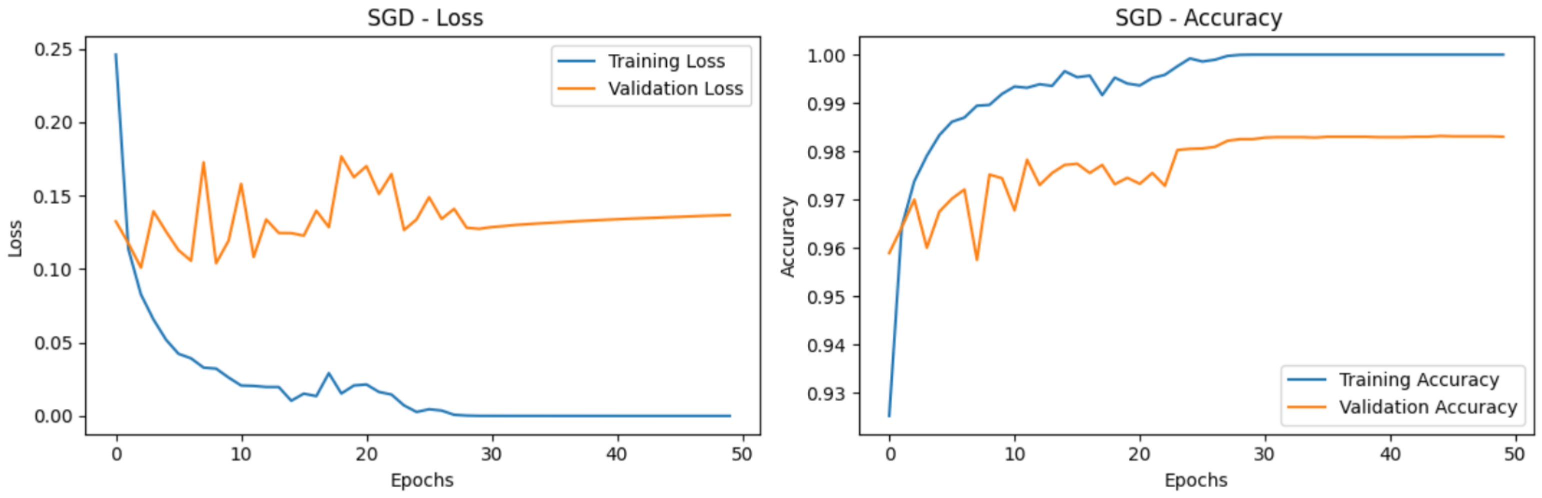


Figure 4: Loss and Accuracy for SGD strategy

2.6.4. Mini Batch SGD with decay (1e-6)

Epoch 50/50 - loss: 0.0537 - accuracy: 0.9860 - val_loss: 0.0992 - val_accuracy: 0.9702

Running time for mini batch sgd with decay: 160.92882251739502 seconds

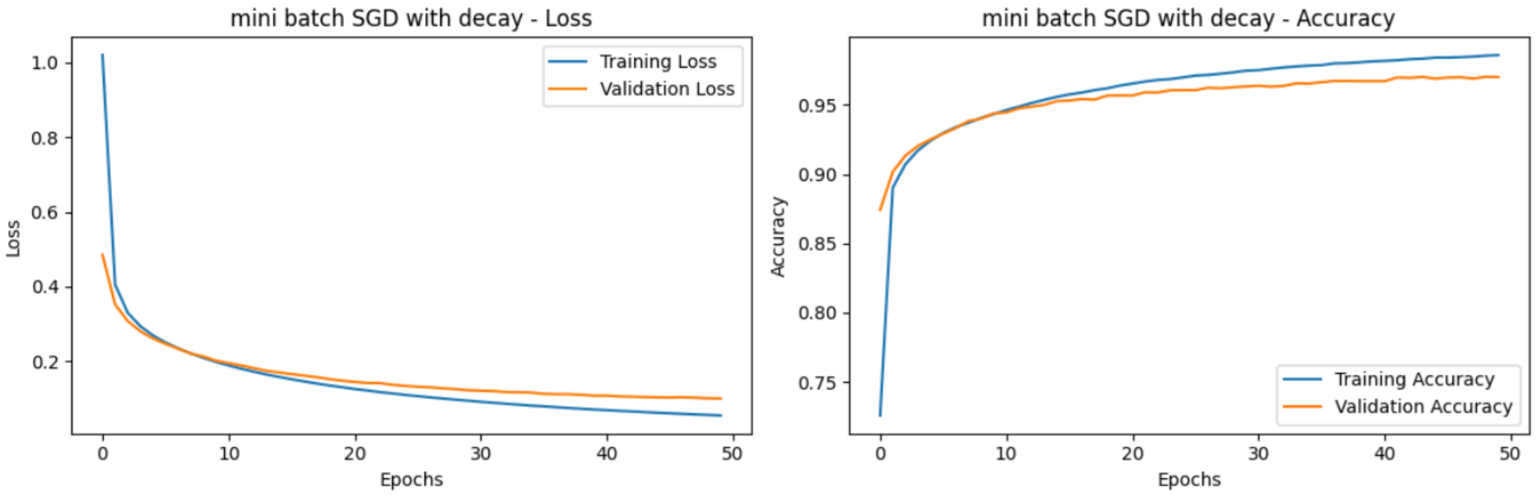


Figure 5: Loss and Accuracy for Mini Batch SGD with decay (1e-6) strategy

2.6.5. SGD with decay and momentum

Epoch 50/50 - loss: 0.0056 - accuracy: 0.9981 - val_loss: 0.4096 - val_accuracy: 0.9699

Running time: 11553.495469331741 seconds

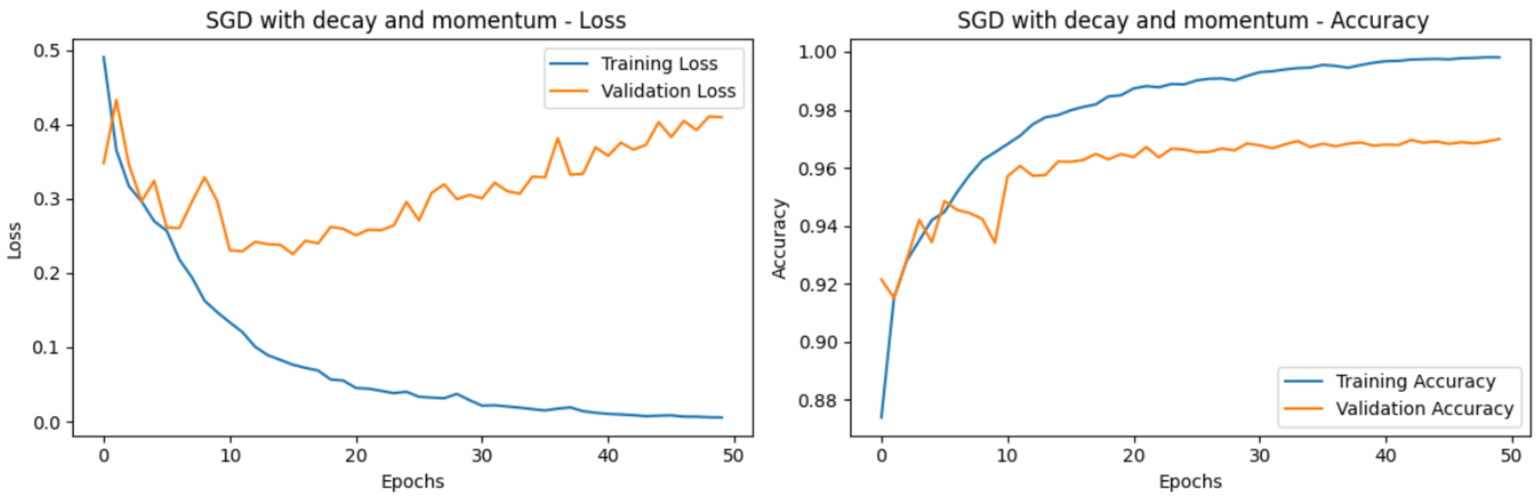


Figure 6: Loss and Accuracy for SGD with decay and momentum strategy

2.6.6. Adam (lr=0.001)

Epoch 50/50 - loss: 0.0045 - accuracy: 0.9988 - val_loss: 0.1980 - val_accuracy: 0.9774

Running time: 357.7459170818329 seconds

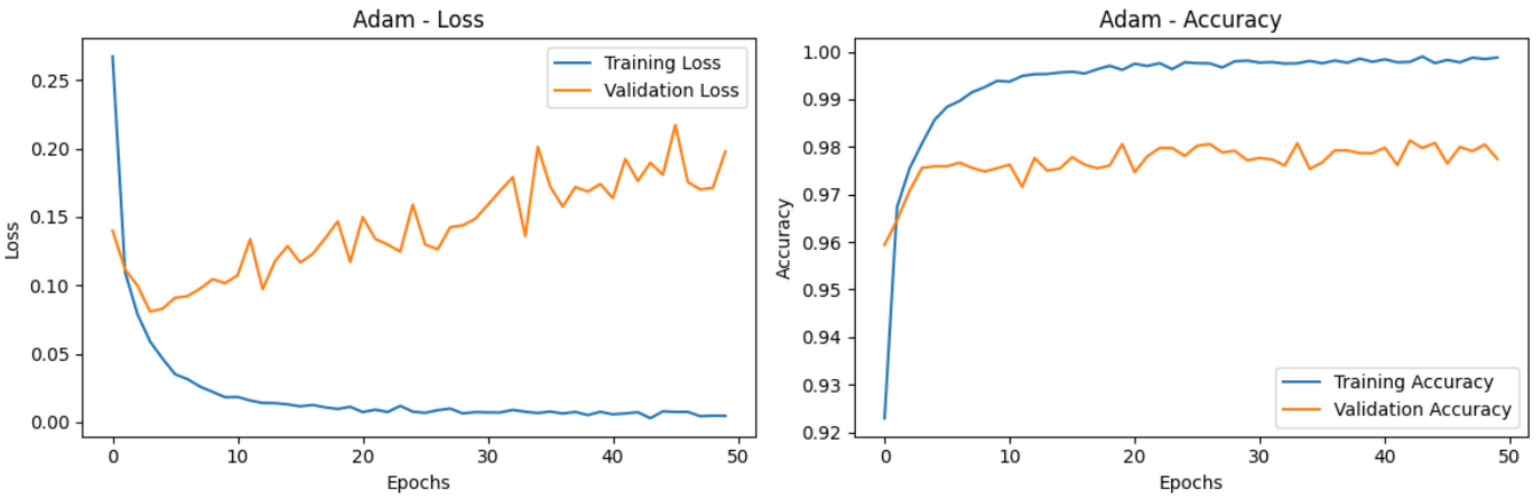


Figure 7: Loss and Accuracy for Adam (lr=0.001) strategy

2.6.7. RmsProp (lr=0.001)

Epoch 50/50 - loss: 7.7320e-07 - accuracy: 1.0000 - val_loss: 0.1817 - val_accuracy: 0.9812

Running time: 346.4854516983032 seconds

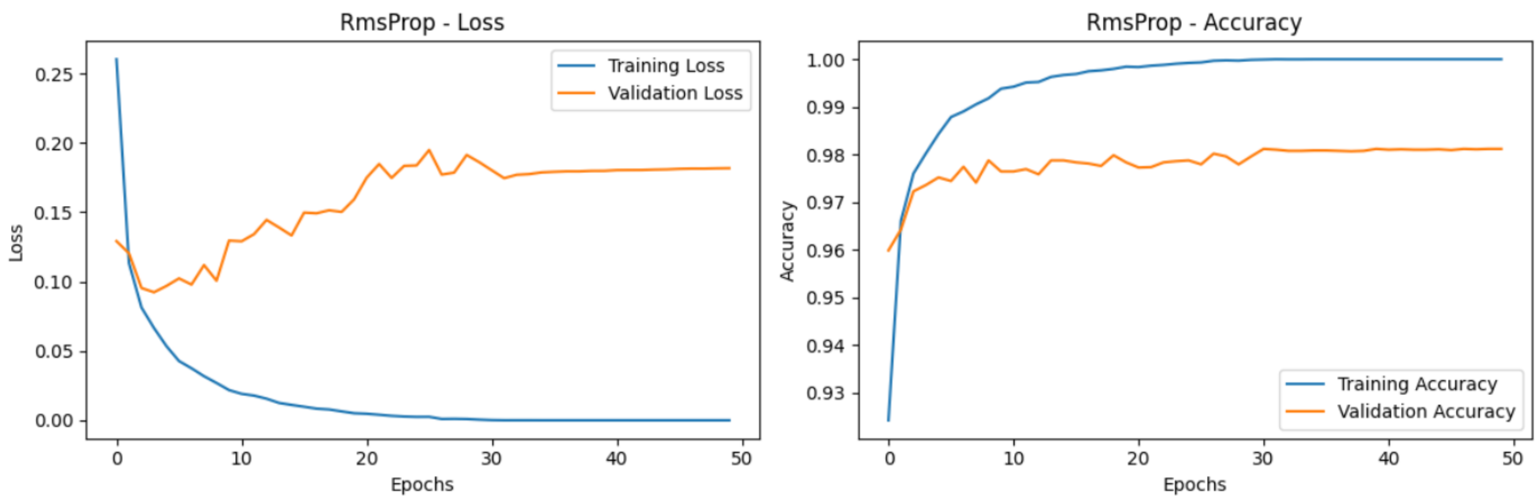


Figure 8: Loss and Accuracy for RmsProp (lr=0.001) strategy

2.7. Comparaiosn between all models

2.7.1. Training times of all the models

- Training time for mini batch sgd : 184.04779696464539 seconds
- Training time for batch sgd : 338.01215839385986 seconds
- Training time for sgd : 10324.142413377762 seconds
- Training time for mini batch sgd with decay: 160.92882251739502 seconds
- Training time for sgd with decay and momentum: 11553.495469331741 seconds
- Training time for adam: 357.7459170818329 seconds
- Training time for RmsProp : 346.4854516983032 seconds

2.7.2. Plot Accuracies for our models

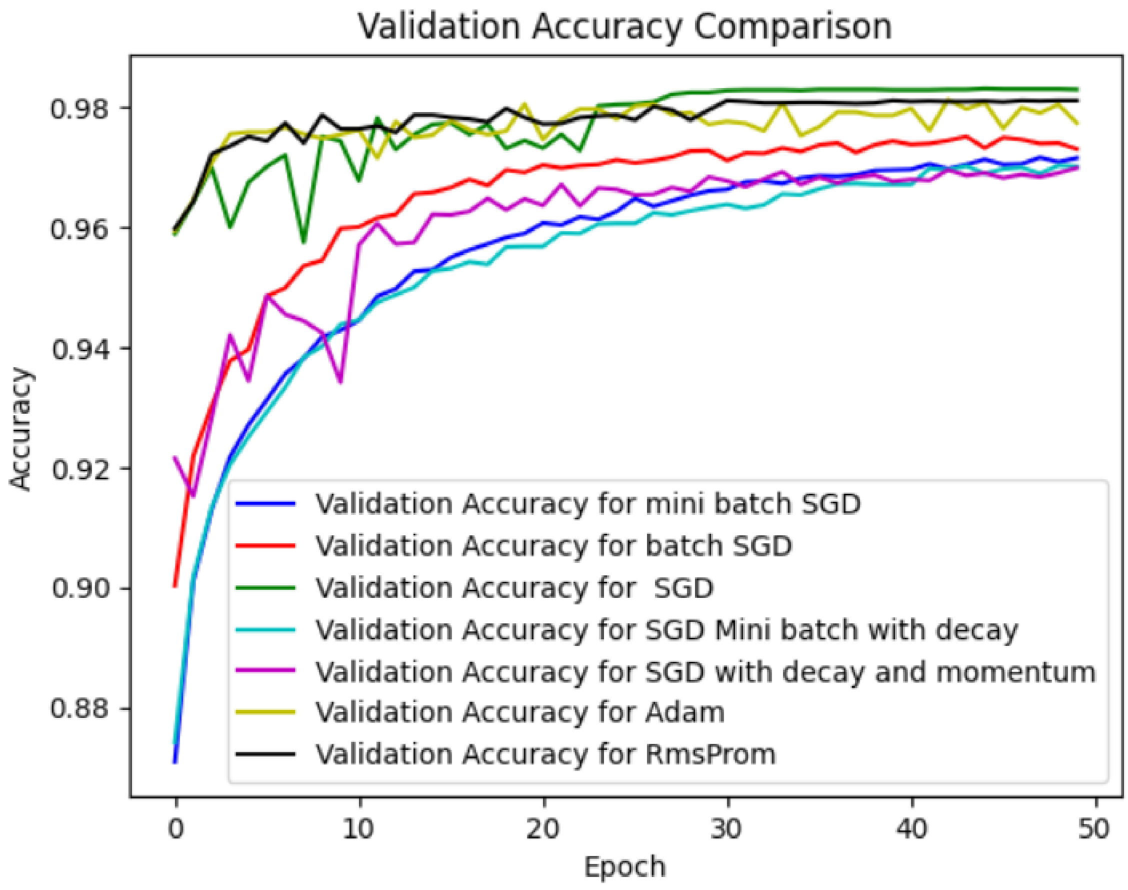


Figure 9: Losses and Accuracies for all models

2.7.3. Comparaison between mini batch SGD, mini batch SGD with decay (1e-6) and SGD with decay (1e-6) and momentum.

According the the plot the graphs of mini batch SGD and mini batch SGD with decay are almost congruent, however the validation accuracy of mini batch SGD is slightly greater than the validation accuracy of mini batch SGD with decay, in the other hand the validation accuracy of SGD with decay and momentum at the first epochs was greater than these two but it dicreased at the last epochs and its accuracy became less than the mini batch SGD, mini batch SGD with decay (1e-6) models. Regarding the training time: the longest training time recorded was for sgd with decay and momentum, then mini batch sgd then mini batch sgd with decay

the similarity in the training curves between mini-batch SGD and mini-batch SGD with decay suggests that the decay factor (1e-6) might not have a substantial impact on the optimization process for this particular configuration

Explanaton for the reason SGD with decay and momentum took longer in training:

- Training on a single random sample introduces a high variance in each weight update since it considers only one data point at a time. This variance can result in a less stable convergence trajectory and might require more iterations to converge.
- Training on one sample at a time reduces the degree of parallelism that can be achieved during optimization. Many modern hardware architectures, such as GPUs, are designed to handle parallel processing efficiently. Mini-batch or batch SGD allows for more parallelism by processing multiple samples simultaneously.
- Using only one sample at a time may result in a less accurate estimation of the true gradient, especially when dealing with noisy or complex datasets. Mini-batch SGD, by contrast, benefits from aggregating information across multiple samples, providing a more robust estimate of the gradient and potentially converging more efficiently.
- Training on a single sample can lead to noisy updates, which may result in oscillations or slower convergence. In contrast, mini-batch SGD allows for a degree of noise reduction through averaging gradients over a subset of the dataset.

Mini-batch SGD with decay demonstrated a shorter training time compared to standard mini-batch SGD. The incorporation of learning rate decay in the former allowed for an adaptive and decreasing learning rate over time, preventing overshooting and promoting more efficient convergence.

2.7.4. Comparaison between SGD (lr=0.01), Adam (lr=0.001), and RmsProp (lr=0.001) optimizers

Theoretical explanations for the observed higher validation accuracy with RMSprop over Adam and batch SGD can be attributed to the adaptive learning rate mechanisms employed by RMSprop. RMSprop adapts the learning rates individually for each parameter based on the historical gradient information. This adaptability allows RMSprop to perform well in different parts of the parameter space and can be particularly advantageous in scenarios with varying and noisy gradients. Adam, while also an adaptive optimizer, may be more sensitive to certain hyperparameter choices, and its momentum term can introduce additional complexity. Batch SGD, on the other hand, uses a fixed learning rate for all parameters, which might result in suboptimal updates, especially in the presence of varying gradients. The performance hierarchy among optimizers can vary depending on the specific characteristics of the dataset and problem at hand.

2.8. Export the best model and perform predictions

the chosen model: SGD (because it lead to highest validation accuracy in the last epoch) - All predictions were correclty predicted

3. Part II: Optimizing hyperparameters (Keras)

In the the second part of this lab, focused on optimizing hyperparameters, we replicated the foundational operations performed in the initial part of the experiment on CIFAR-10 dataset. This encompassed essential steps such as data loading, meticulous data splitting into training, validation, and testing sets, and data normalization to ensure a consistent and effective training environment.

Building upon this, we extended our exploration by incorporating optimization techniques to enhance model performance. The architecture of the models was refined by introducing L2 norm regularization, strategically placing dropout layers to mitigate overfitting, implementing early stopping techniques to prevent unnecessary training iterations, and incorporating batch normalization layers for improved convergence.

Additionally, to systematically explore the hyperparameter space, we employed a random search approach, allowing for a more comprehensive examination of potential configurations.

The incorporation of regularization, dropout, early stopping, batch normalization, and random search collectively contributed to a more nuanced understanding of hyperparameter tuning and its impact on deep learning model outcomes.

3.1. Data Loading (CIFAR-10 using Keras API)

3.2. Data Splitting : Training set , Validation set, Testing set

3.3. Data Visualization: using matplotlib.pyplot

3.4. Data Normalization:

we divided the pixel values by 255, rescaling the data to a normalized scale ranging from 0 to 1.

3.5. Training

we trained the same designed architecture of the first part of this lab using Mini-batch Stochastic Gradient Descent (SGD) with a batch size of 128, a learning rate of 0.01, for 50 epochs. the we plotted the learning curves on the train and validation data. Here are the results we obtained:

3.6. Performing Optimization techniques

3.6.1. L2 norm regularization

Here we added L2 norm regularization to the second fully connected layer, and plotted the learning curves on the train and validation data. then compared the results obtained between the architecture with and without L2 norm.

The higher validation accuracy observed after adding L2 norm regularization aligns with theoretical expectations. L2 norm regularization adds a penalty term to the loss function, discouraging large weights in the model. This discouragement helps prevent overfitting by promoting a simpler and more generalized model. In practice, L2 norm regularization can lead to improved generalization performance, reducing the risk of the model fitting noise in the training data. The observed higher validation accuracy suggests that L2 norm regularization effectively controlled overfitting, leading to a more robust and accurate model on unseen data, consistent with the theoretical understanding of regularization techniques.

3.6.2. Dropout layers

we added dropout layers with rates of 0.2, 0.3, and 0.5 to the second fully connected layer, and compared the results obtained by plotting the curves

The increase in validation accuracy after adding dropout layers aligns with the theoretical expectations of dropout regularization. Dropout introduces randomness during training, preventing overfitting by discouraging reliance on specific connections. While higher dropout rates can initially reduce accuracy, they effectively act as a form of regularization, promoting a more generalized model. The observed improvement in validation accuracy suggests that the dropout layers have successfully enhanced the model's ability to generalize and perform well on unseen data.

3.6.3. Early stopping

here we used the early stopping technique, and specified the epoch at which the training stopped.

The use of EarlyStopping with a patience of 5 and monitoring 'val_loss' allows the training to automatically stop if the validation loss does not improve for five consecutive epochs. The theoretical explanation is that EarlyStopping helps prevent overfitting by monitoring the model's performance on the validation set. If the model's performance plateaus or worsens over consecutive epochs, it indicates that further training may not lead to improved generalization. In our case, the training stopped at epoch 11 because there was no significant improvement in validation loss for five consecutive epochs, supporting the theoretical idea of early stopping to prevent overfitting and save computation resources.

3.6.4. Batch Normalization layers

Here we added a batch normalization layer after the first hidden layer, and trained using mini- batch SGD.

The significant increase in training accuracy and a slight increase in validation accuracy after adding a Batch Normalization layer align with the theoretical expectations. Batch Normalization has regularization effects that can mitigate overfitting by normalizing activations and reducing internal covariate shift during training. However, in some cases, it might not entirely eliminate overfitting, especially if the model capacity is high or other regularization techniques are not applied. Adjusting the model's complexity or incorporating additional regularization methods could further address overfitting concerns.

3.6.5. Random Search

we used random search to find the best hyperparameters for learning_rate, dropout_rate, and batch_size, and specified the parameters that ensure the best results.

The obtained result from the random search indicates that the best hyperparameters for the model include a learning rate of 0.01 and a dropout rate of 0.3. A learning rate of 0.01 suggests a moderate step size for weight updates, while a dropout rate of 0.3 indicates a level of regularization during training. These hyperparameters likely strike a balance between effective learning and preventing overfitting on the dataset. The absence of batch_size in the provided result suggests that it might have been set separately or that the default value was deemed suitable during the random search. Overall, these hyperparameters were determined to yield the best results during the search process.