DEEP LEARNING

TP2 REPORT

- Sehili Chaima
- Hachoud Mohammed

PART 01

TRAINING AN MLP (KERAS)

Model Architecture :

```
tf.random.set_seed(1234)
def mlp_model():
    mlp_model = keras.Sequential([
    tf.keras.Input(shape=(784,)),
    Dense(128, activation='relu', name='hidden_layer_1'),
    Dense(64, activation='relu', name='hidden_layer_2'),
    Dense(10, activation='softmax', name='output_layer'),
])
    return mlp_model
```

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 hidden_layer_1 (Dense)      (None, 128)               100480

 hidden_layer_2 (Dense)      (None, 64)                8256

 output_layer (Dense)        (None, 10)                650

=================================================================
Total params: 109,386
Trainable params: 109,386
Non-trainable params: 0
_________________________________________________________________
```

Figure 01: Model Architecture

The MLP architecture consists of an **input layer, two hidden layers with 128 and 64 neurons respectively,** employing the **RELU activation function.** The final layer is a **fully connected layer using softmax activation** to produce output probabilities for different classes. The model is trained using **the cross-entropy loss function**

Comparing between SGD Vs Batch SGD Vs Mini-Batch SGD

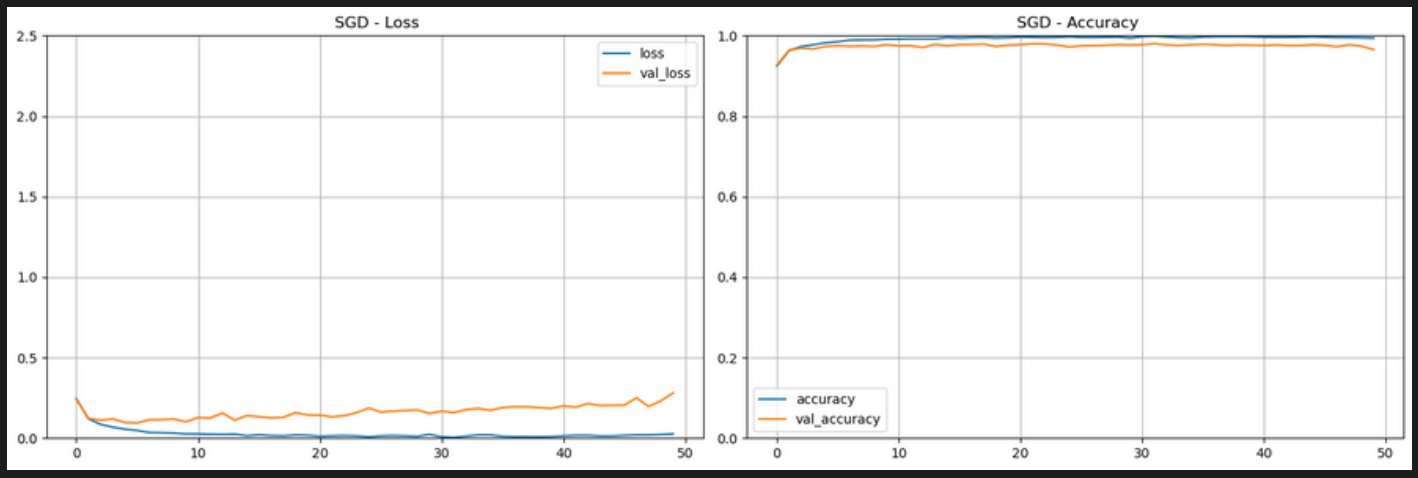

Figure 02: SGD learning curves

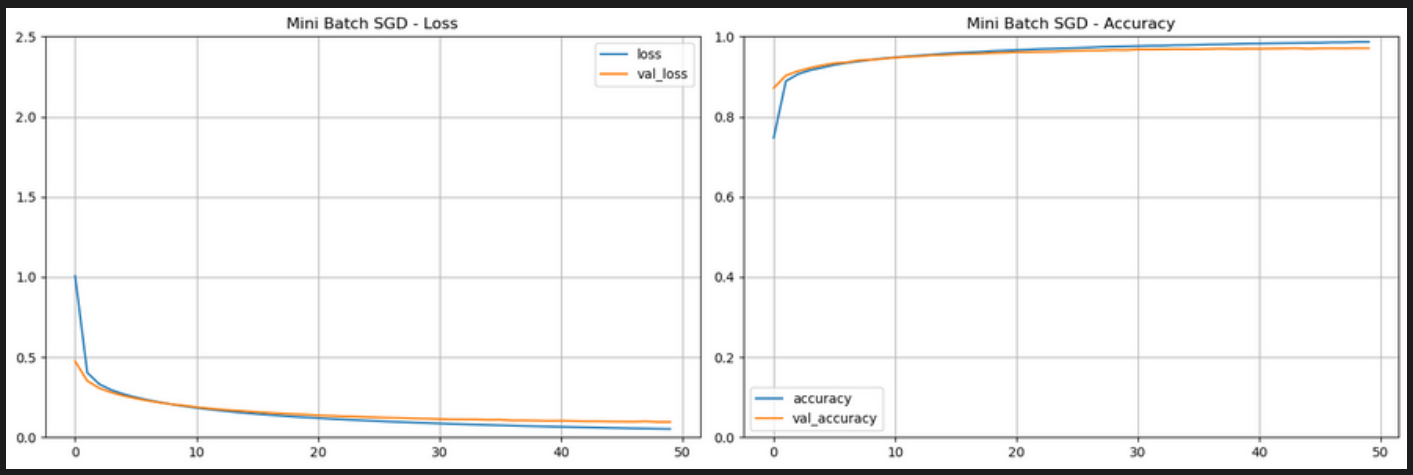

Figure 03: Mini Batch SGD

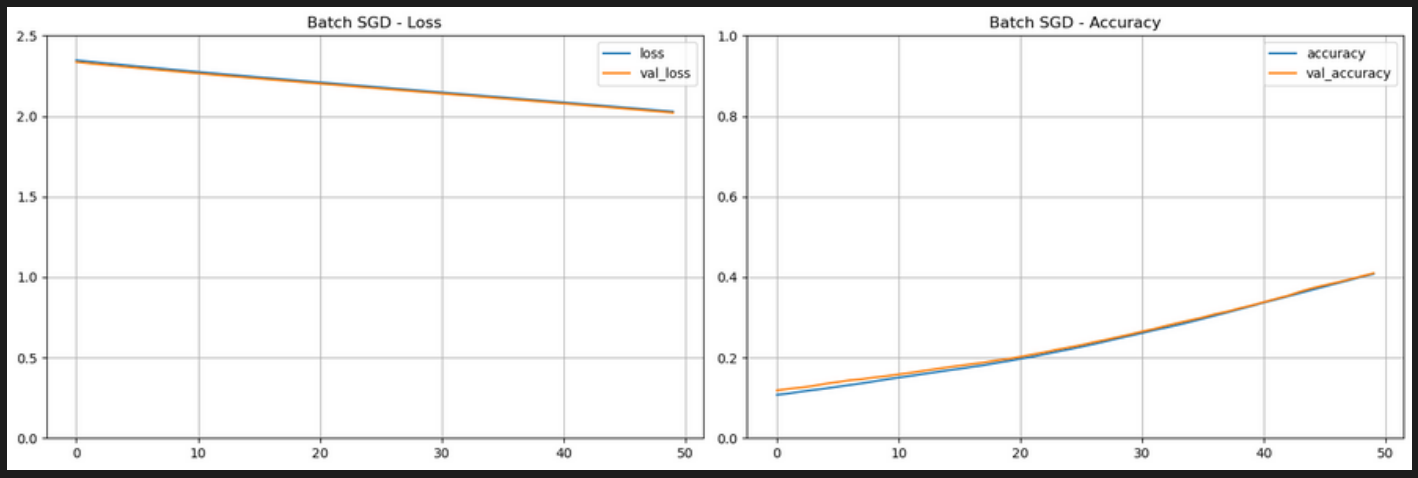

Figure 04: Batch SGD

Learning time comparison between SGD, Batch SGD and Mini-Batch SGD:

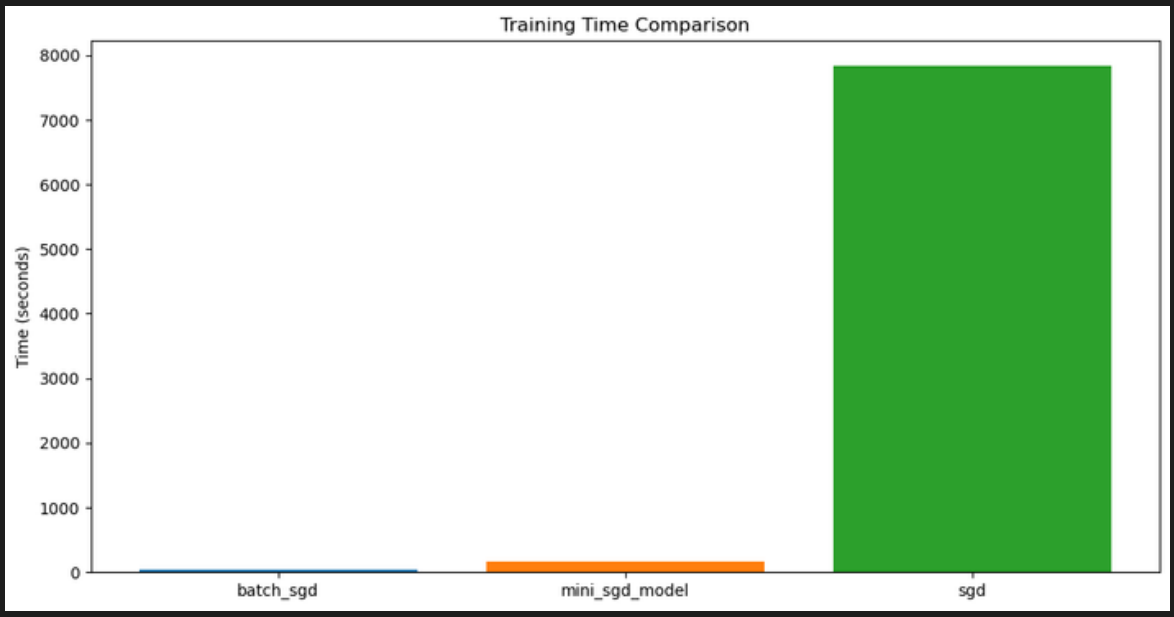

Figure 05: Time comparison
(SGD, Batch SGD, Mini Batch SGD)

Batch SGD emerges as a compelling choice in optimization frameworks due to its ability to efficiently traverse parameter space with reduced noise. Unlike **Stochastic Gradient Descent (SGD)**, which exhibits pronounced oscillations, **Batch SGD** delivers smoother convergence trajectories. Moreover, **Batch SGD** often outperforms **SGD in the same number of iterations,** showcasing superior convergence.

Mini-Batch SGD approach a balance between the two, offering a compromise between **convergence smoothness** and **computational efficiency**.

Through evaluating **1000 iterations, Batch SGD** has demonstrated remarkable **speed** and **convergence** precision, resulting in a **negligible variance** between **training** and **validation** metrics.

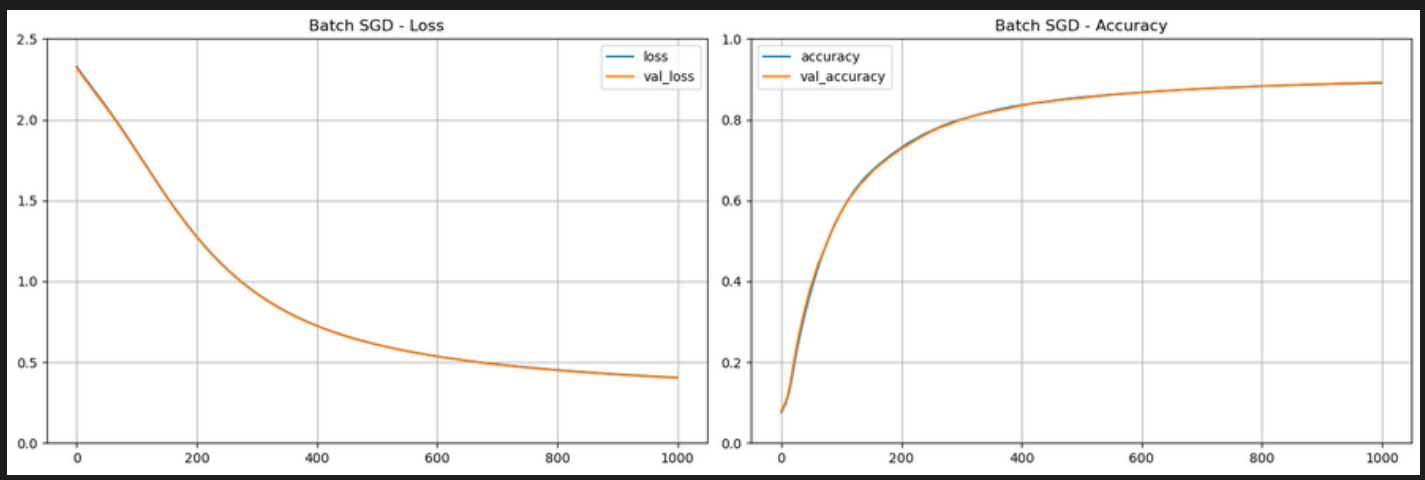

Figure 06: Training Batch SGD on 1000 epochs

Training time for Batch SGD on 1000 epochs : **594.5109355449677 seconds**
Whereas Stochastic Gradient descent on 50 epochs : **7830.665495872498 seconds**

Comparing between Mini-Batch SGD , with decay & with decay and momentum

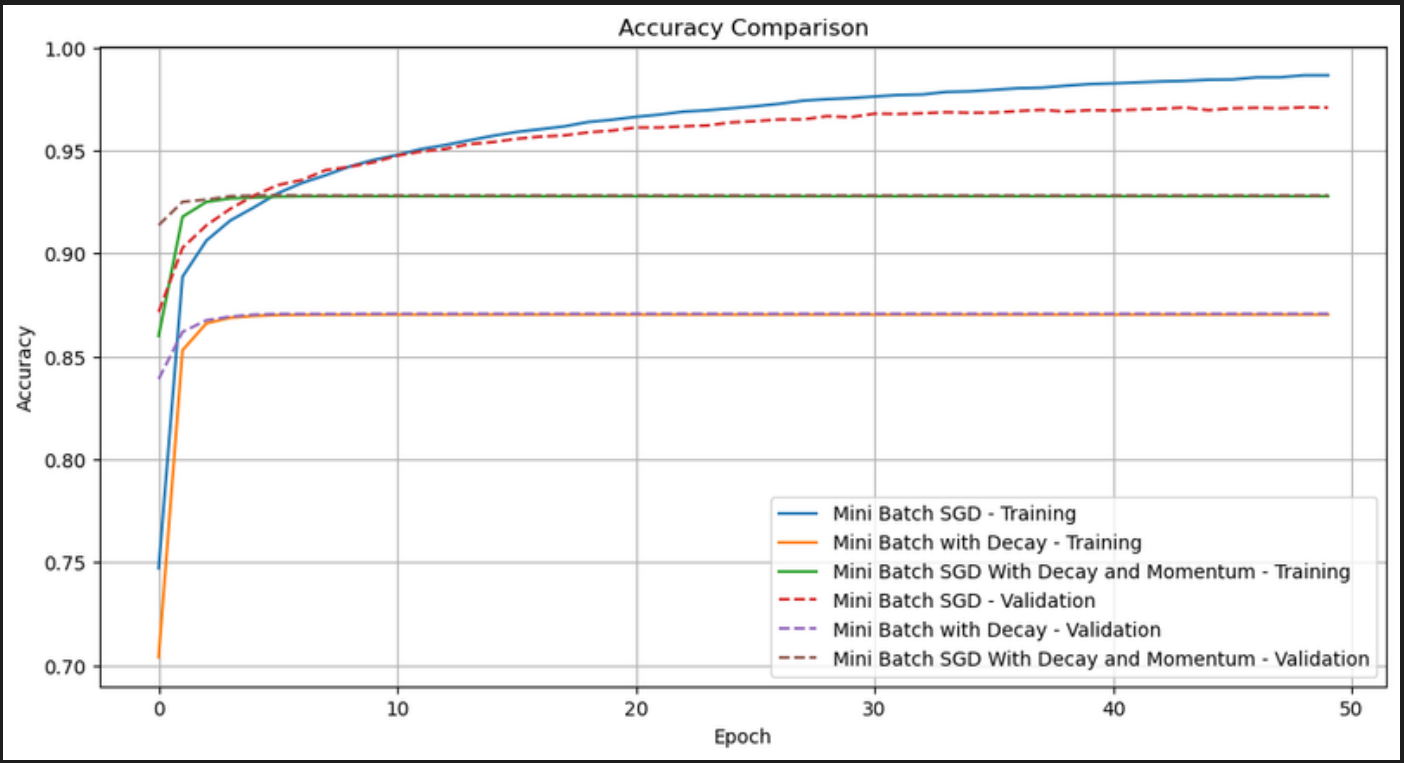

Figure 07: Accuracy Comparison

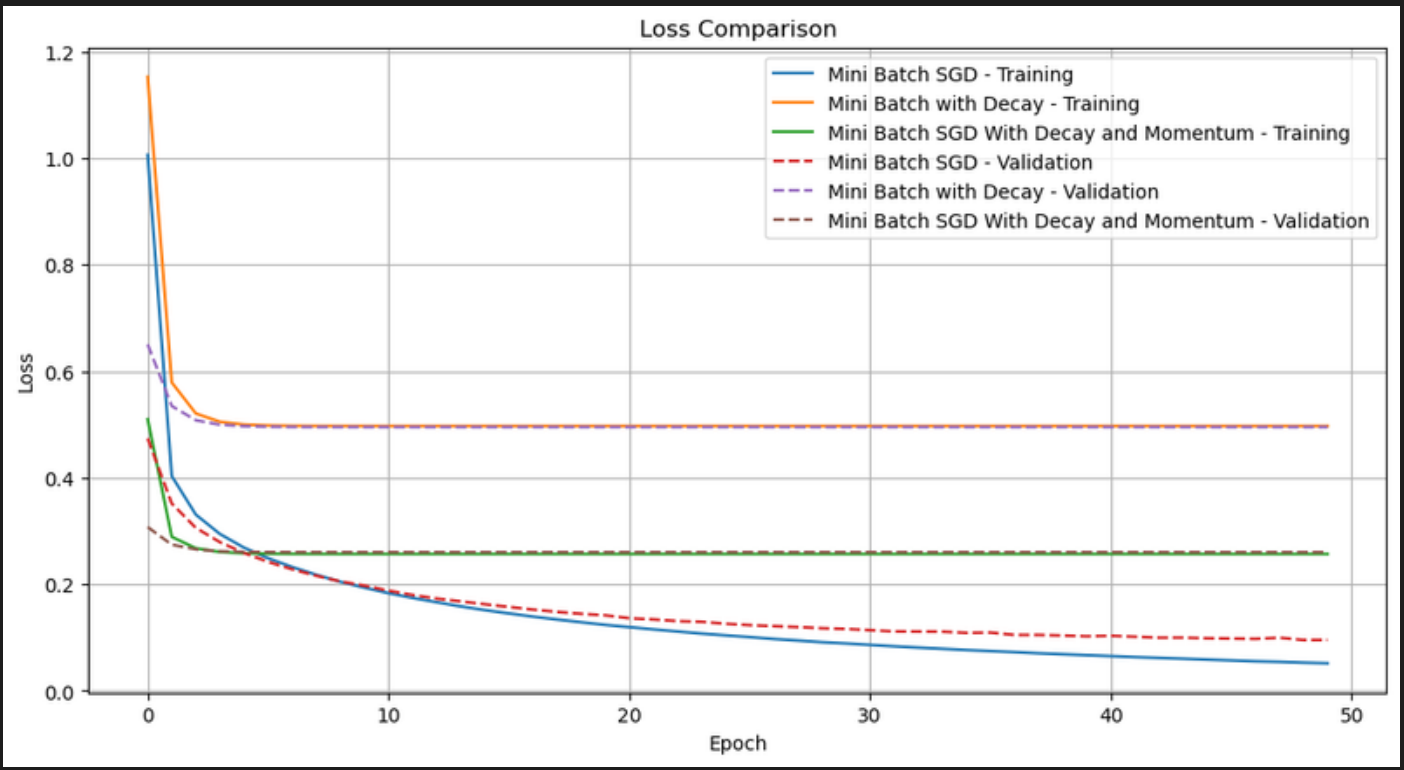

Figure 08: Loss Comparison

Training time Comparison between Mini-Batch SGD , with decay & with decay and momentum

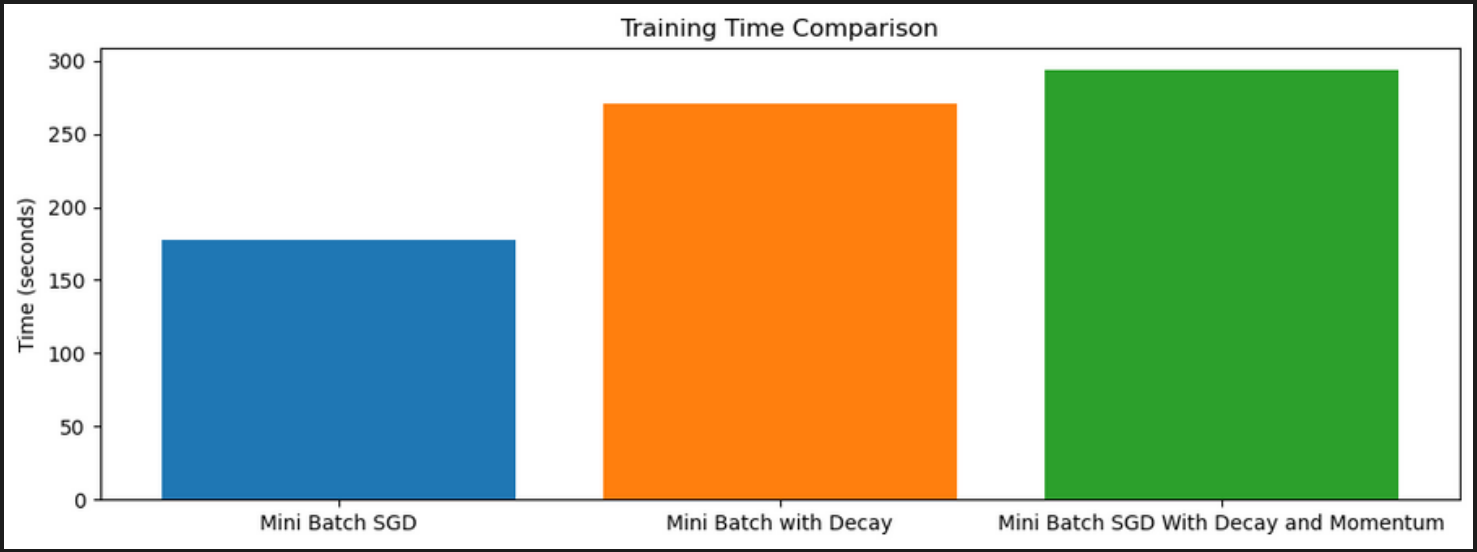

Figure 09: Time Comparison

The three algorithms compared are: **Mini-batch SGD , with decay of 1e-6, with decay of 1e-6 and momentum**

- The first algorithm, `**Mini-batch SGD**`, has a accuracy of %97, but has the shortest time of 177 seconds.

- the second algorithm, `**Mini-batch SGD with decay of 1e-6**`, has slightly lower accuracy of 87%, and it takes slightly longer to train at 297.15 which is significantly longer than the other two algorithms.

- The third algorithm, `**Mini Batch SGD with decay of 1e-6 and momentum**`, has a significantly accuracy of %92, with a training time of 293.73 seconds.

Mini-batch SGD with `**decay**` would `converge` Lower than **Mini-batch SGD**, and the addition of `**momentum**` would improve convergence speed and stability as well.

In conclusion, we can say that **Mini-batch SGD** would be the `**preferred**` algorithm among the three, as it achieves a `higher accuracy` than **Mini-batch SGD whith Decay** and has a relatively `short training time` compared to **SGD with decay and momentum.**

Comparing between Mini-Batch SGD , with decay & SGD with decay and momentum

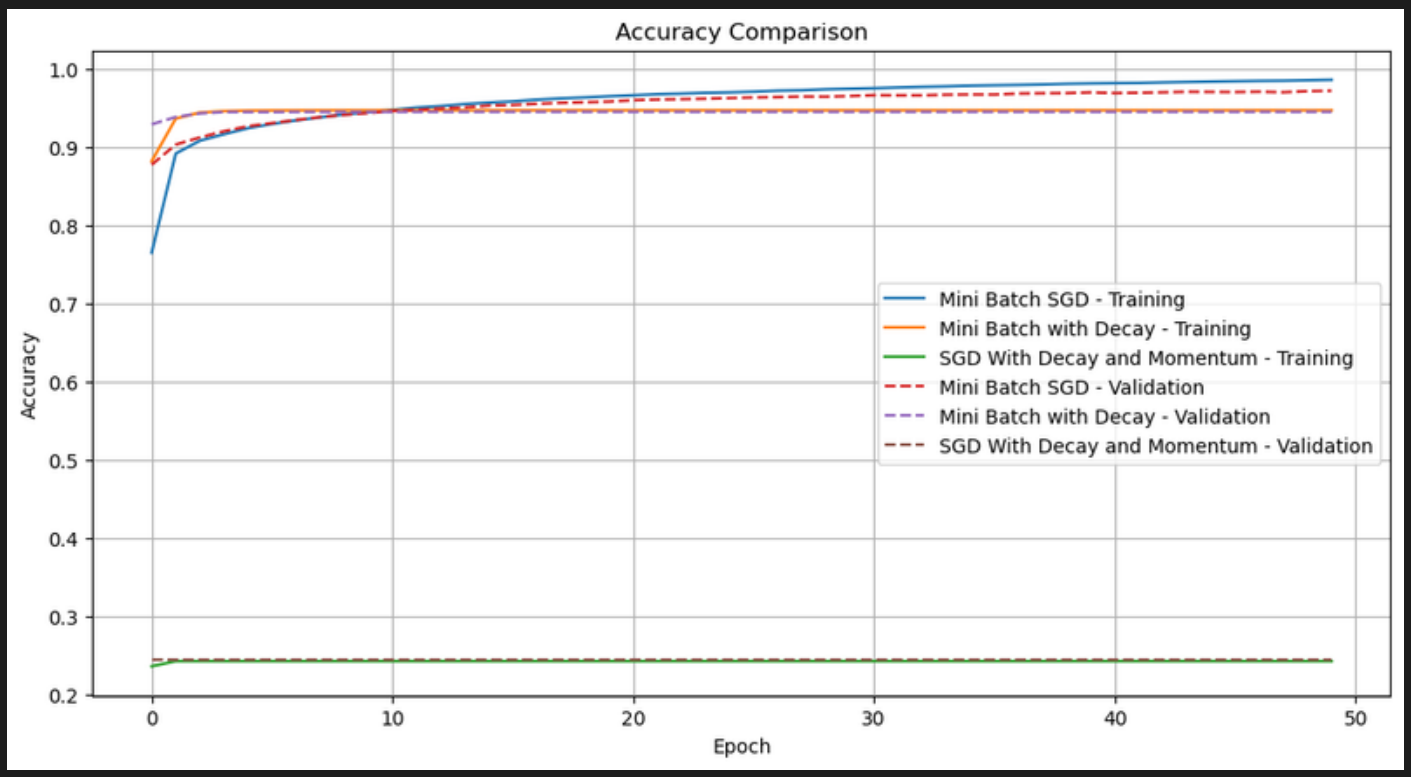

Figure 10: Accuracy Comparison

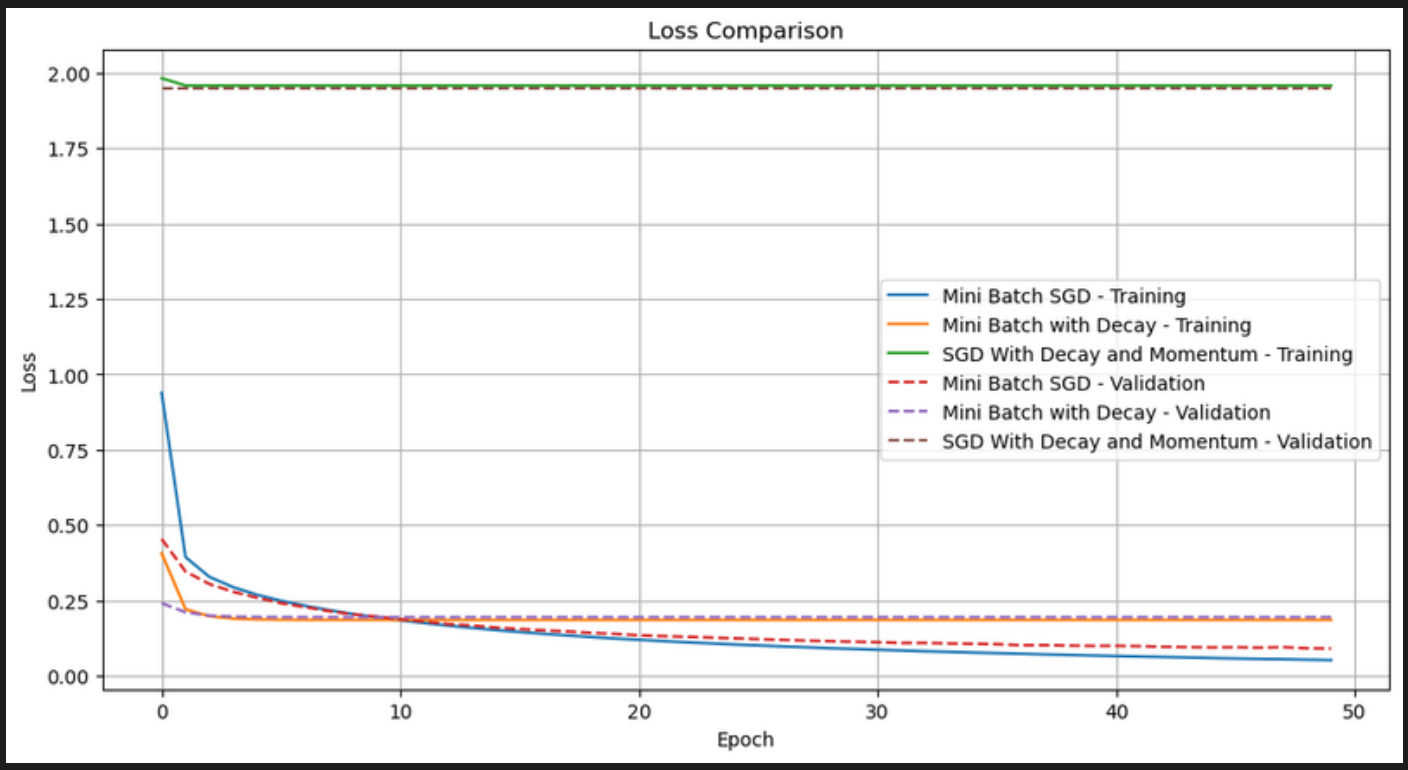

Figure 11: Loss Comparison

Training time Comparison between Mini-Batch SGD , with decay & SGD with decay and momentum

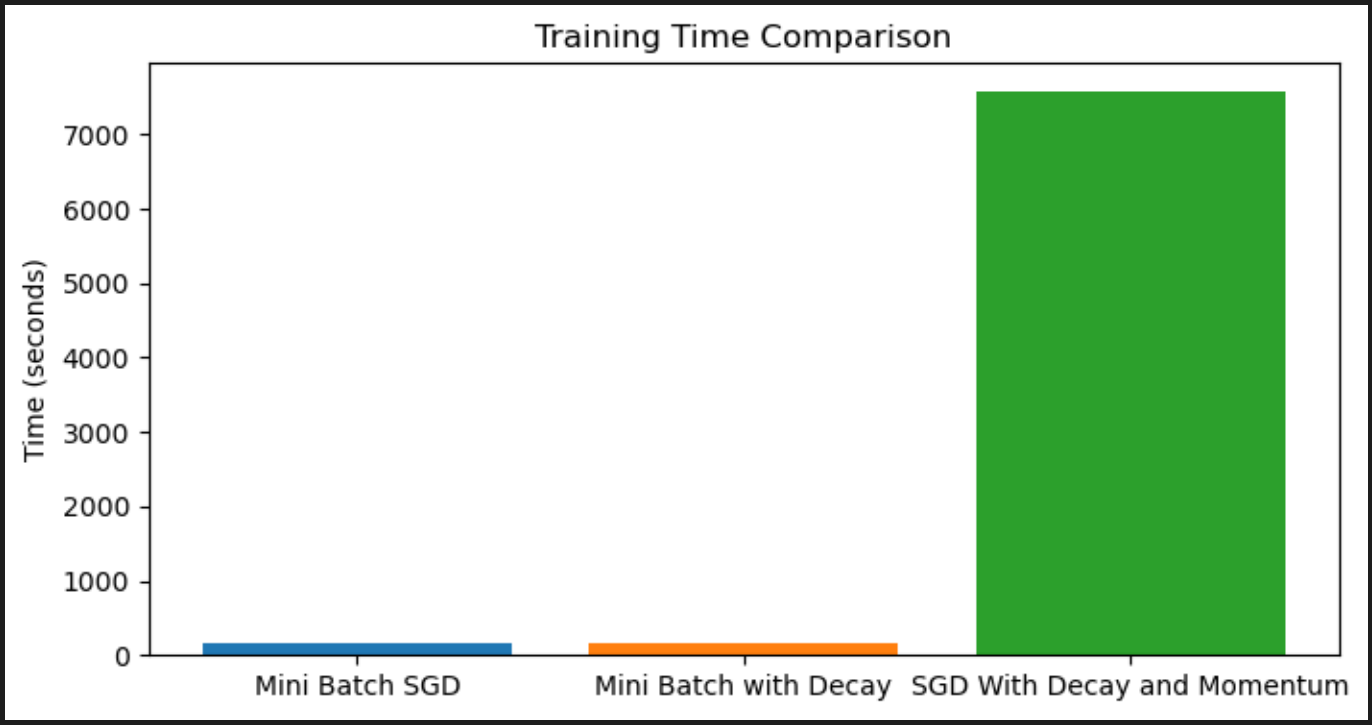

Figure 12: Time Comparison

SGD with decay of 1e-6 and momentum, has a significantly lower accuracy of **20%** which could indicate that it **got stuck in a local minimum during training**. Moreover, it has **the longest training time** (7577.678876161575) .

The accuracy of **Mini Batch SGD** is clearly **higher** than **Mini Batch SGD with Decay**,but we can notice in the plot **the variance** between the training and validation . Whereas for **Mini Batch with decay,** the accuracy is not really bad but **less** than **Mini Batch SGD** but the **variance** between Training and validation is almost **zero**.

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

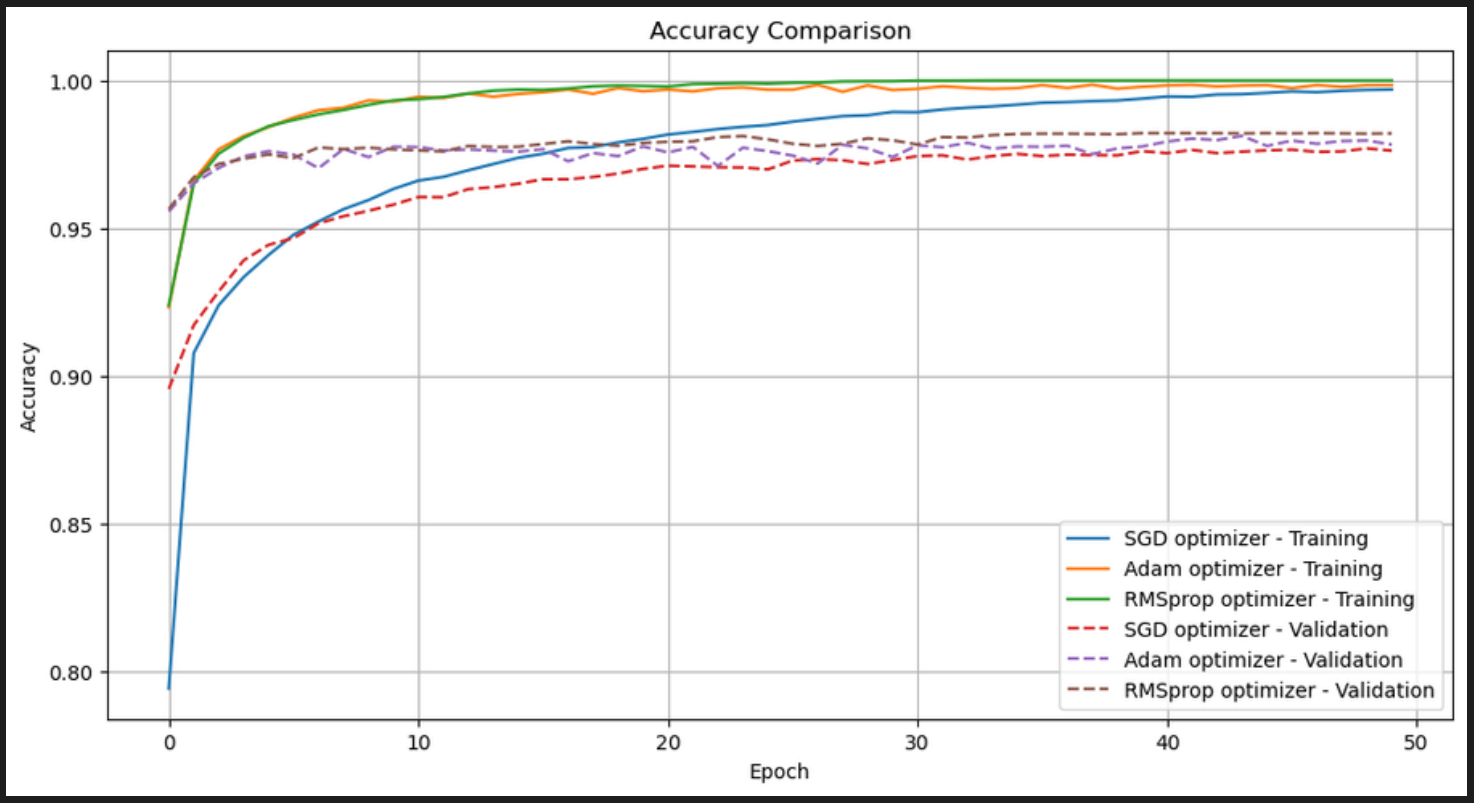

Figure 13: Accuracy

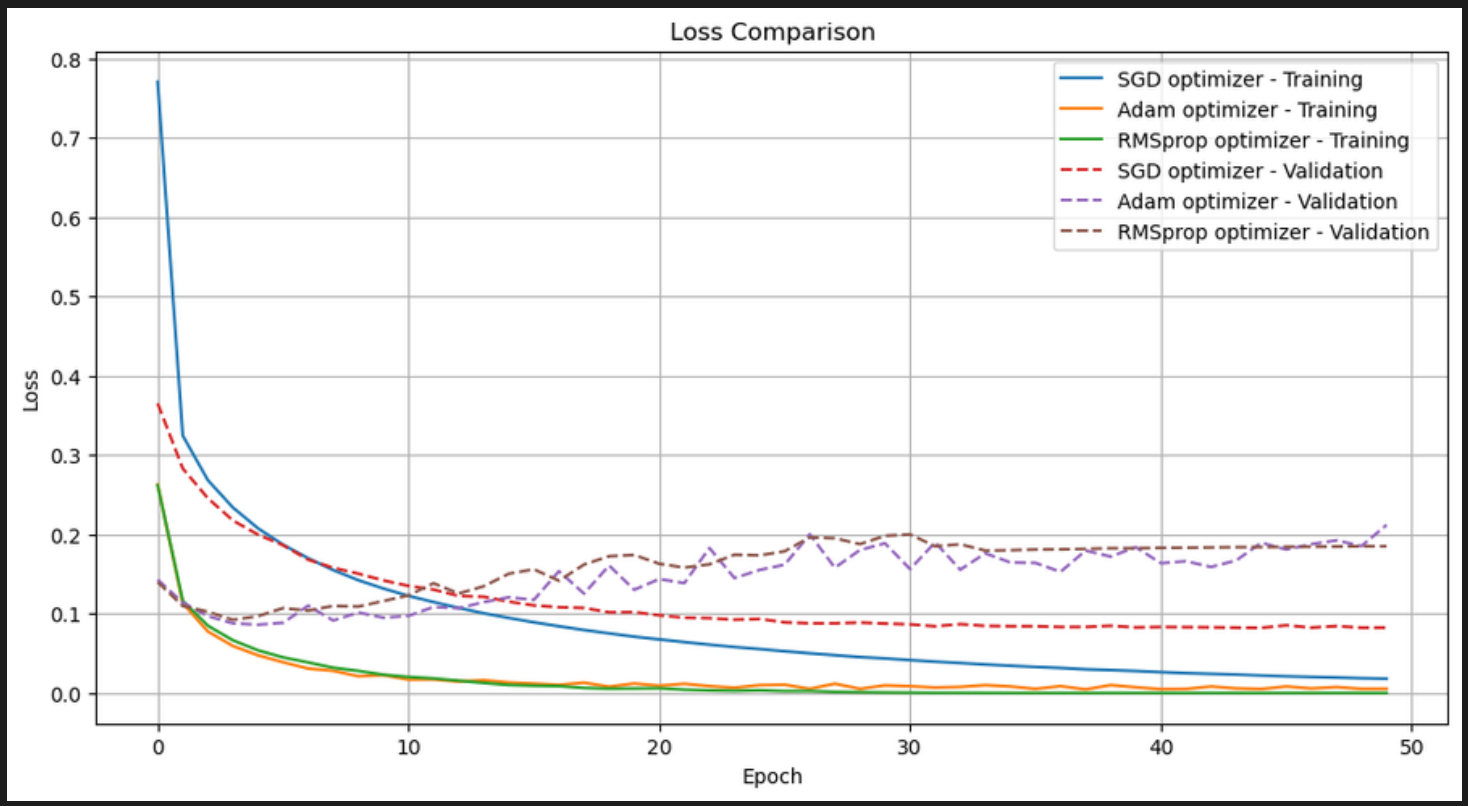

Figure 14: Loss

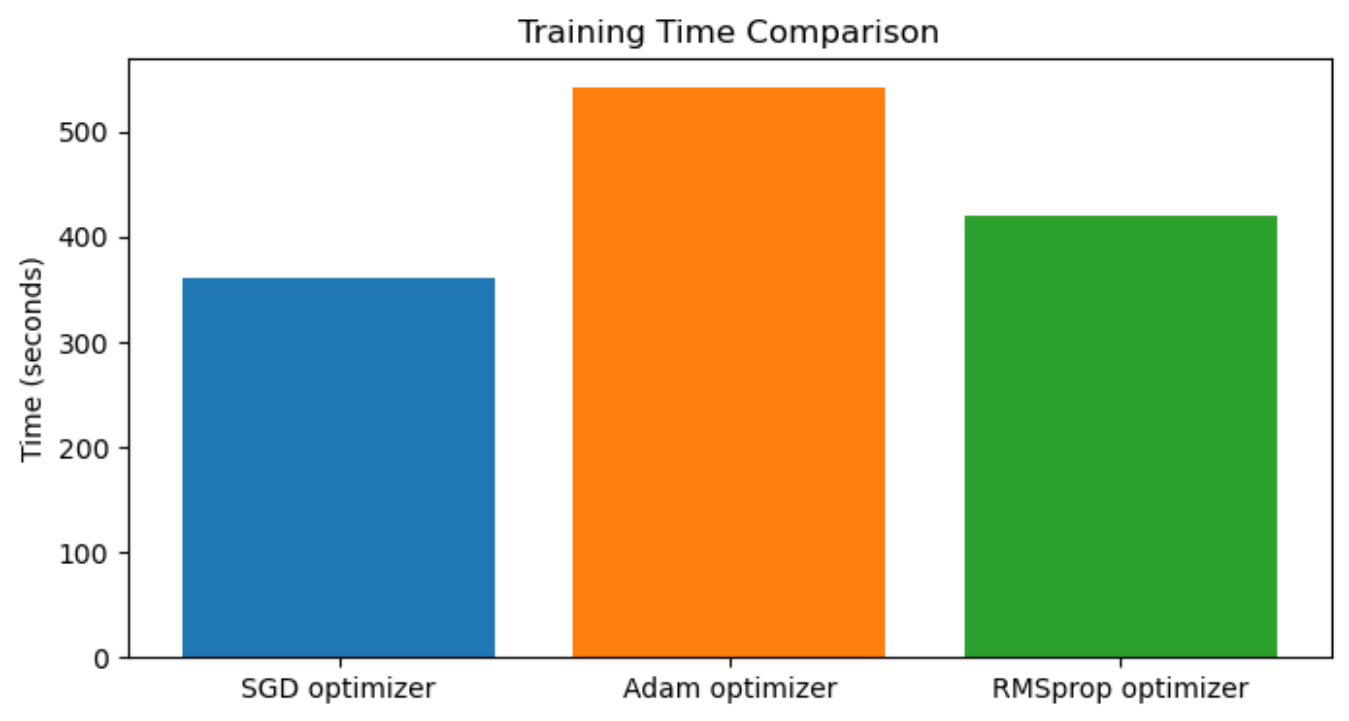

Figure 15: Training Time

SGD, **Adam**, and **RmsProp** are all **`optimization`** algorithms used to train neural networks. The main difference between these algorithms is how they update the parameters of the network during training.

- **`SGD`** updates the parameters based on the gradient of the loss function with respect to the parameters.

- **`Adam`** and **`RmsProp`** both adjust the learning rate dynamically during training.

Both **'ADAM '** and **'RmsProp'** performed better than **'SGD'** when it comes to both the accuracy and the loss, with a little **ossillations** by **'ADAM'** optimizer.

In terms of **loss 'SGD'** results in **training loss** of **0.0503** and **val_loss** of **0.0954** , whereas **'RmsPROP' training loss i**s approximately 0 and the **validation loss** is **0.15** (variance more than SGD)

we picked **'RmsProp'** as the best model as it gave an **accuracy** of **100%** on the **training** set and **98%** on the **validation** set better than **'SGD'** which **training accuracy** is : **0.98** and **validation accuracy** is **0.97** (same variance)

Making predictions on some samples from the test set with RmsProp (lr=0.001) Model.

Exporting the model:

```
model_rmsprop.save('models/finalized_model')
```

Importing the model:

```
model = keras.models.load_model('models/finalized_model')
```

Making Predictions on some test data:

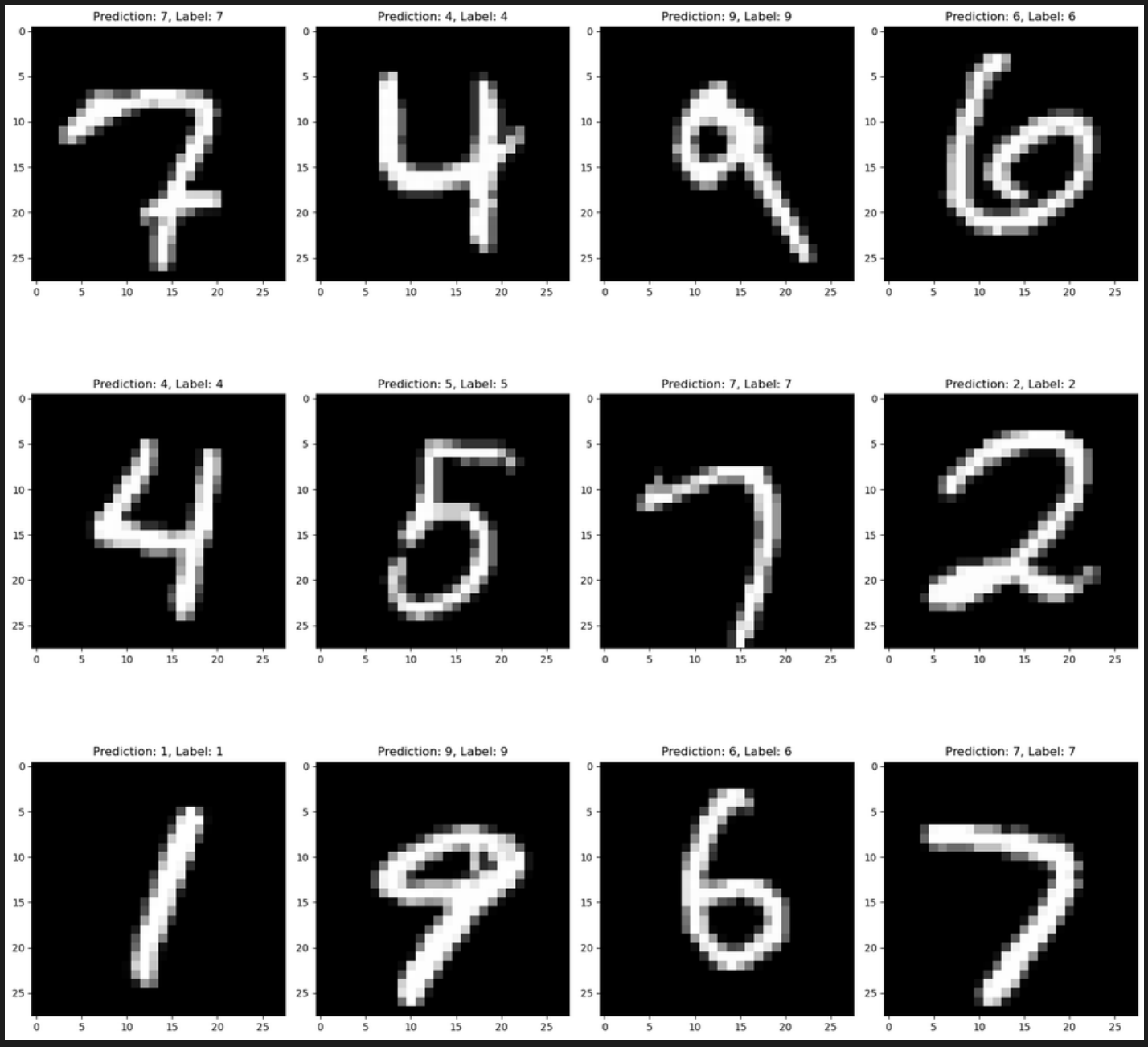

Figure 16: Test

PART 02

OPTIMIZING HYPERPARAMETERS (KERAS)

Model Architecture:

```
tf.random.set_seed(1234)
def mlp_model():
    mlp_model = keras.Sequential([
    tf.keras.Input(shape=(3072,)),
    Dense(128, activation='relu', name='hidden_layer_1'),
    Dense(64, activation='relu', name='hidden_layer_2'),
    Dense(10, activation='softmax', name='output_layer'),
])
    return mlp_model
```

Training with Mini-batch SGD : batch size 128, a learning rate of 0.01, for 50 epochs:

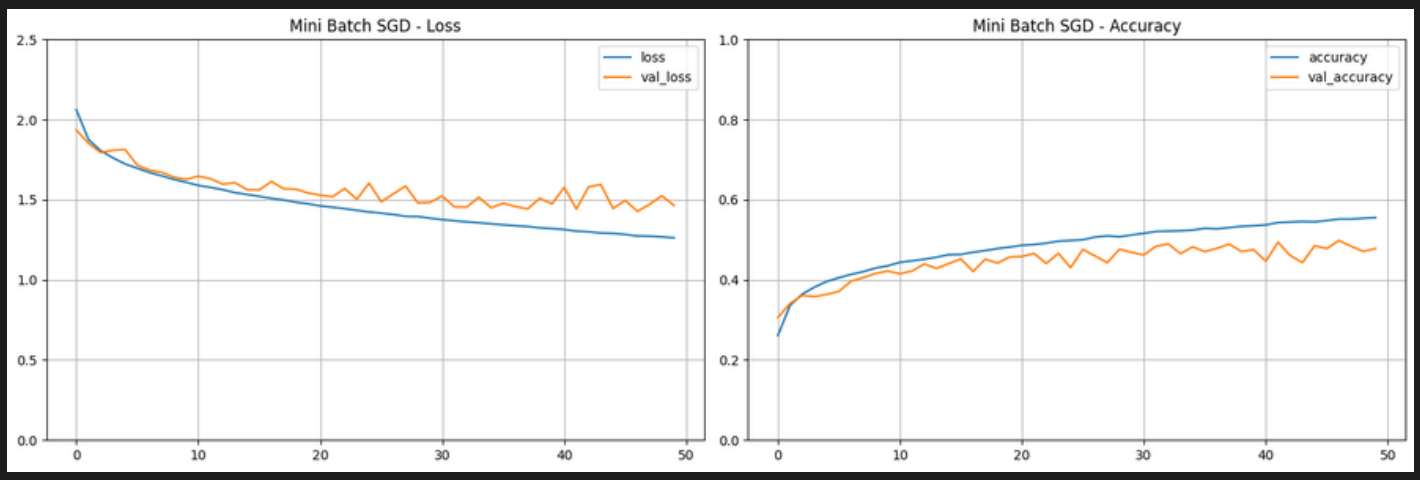

Figure 17 Mini Batch SGD

Adding L2 Regularization:

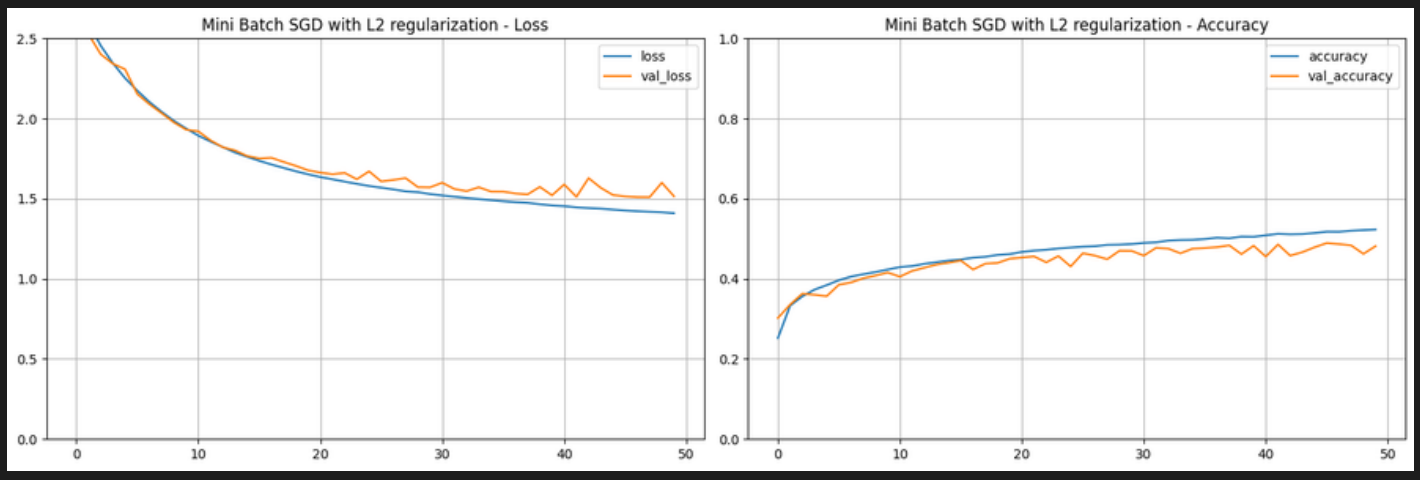

Figure 18 Mini Batch SGD with L2 normalization

We can clearly see that it helped with the generalization as the variance between the training and validation sets decreased in both loss and accuracy.

Comparison between Mini Batch SGD with and without L2 regularization:

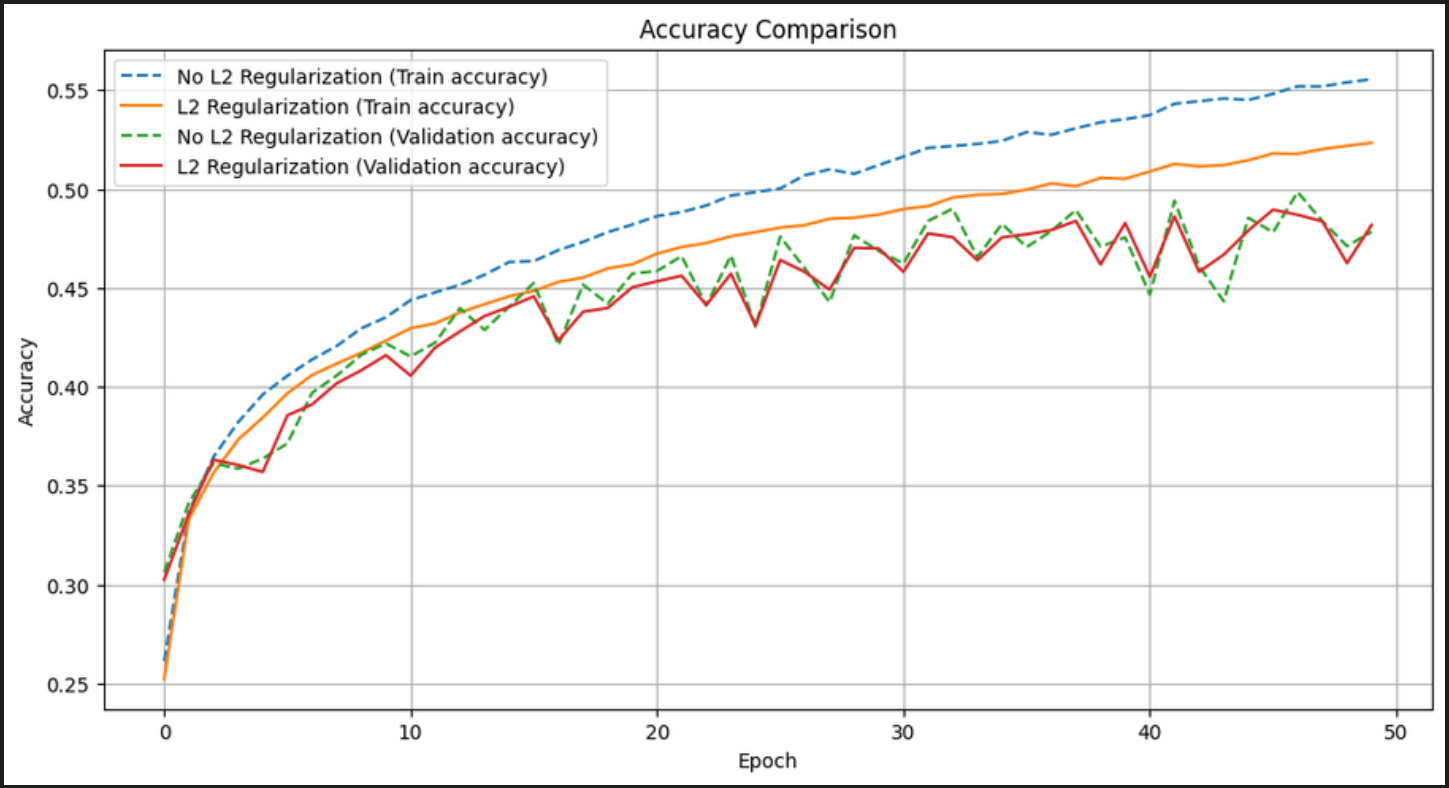

Figure 19: Accuracy Comparison

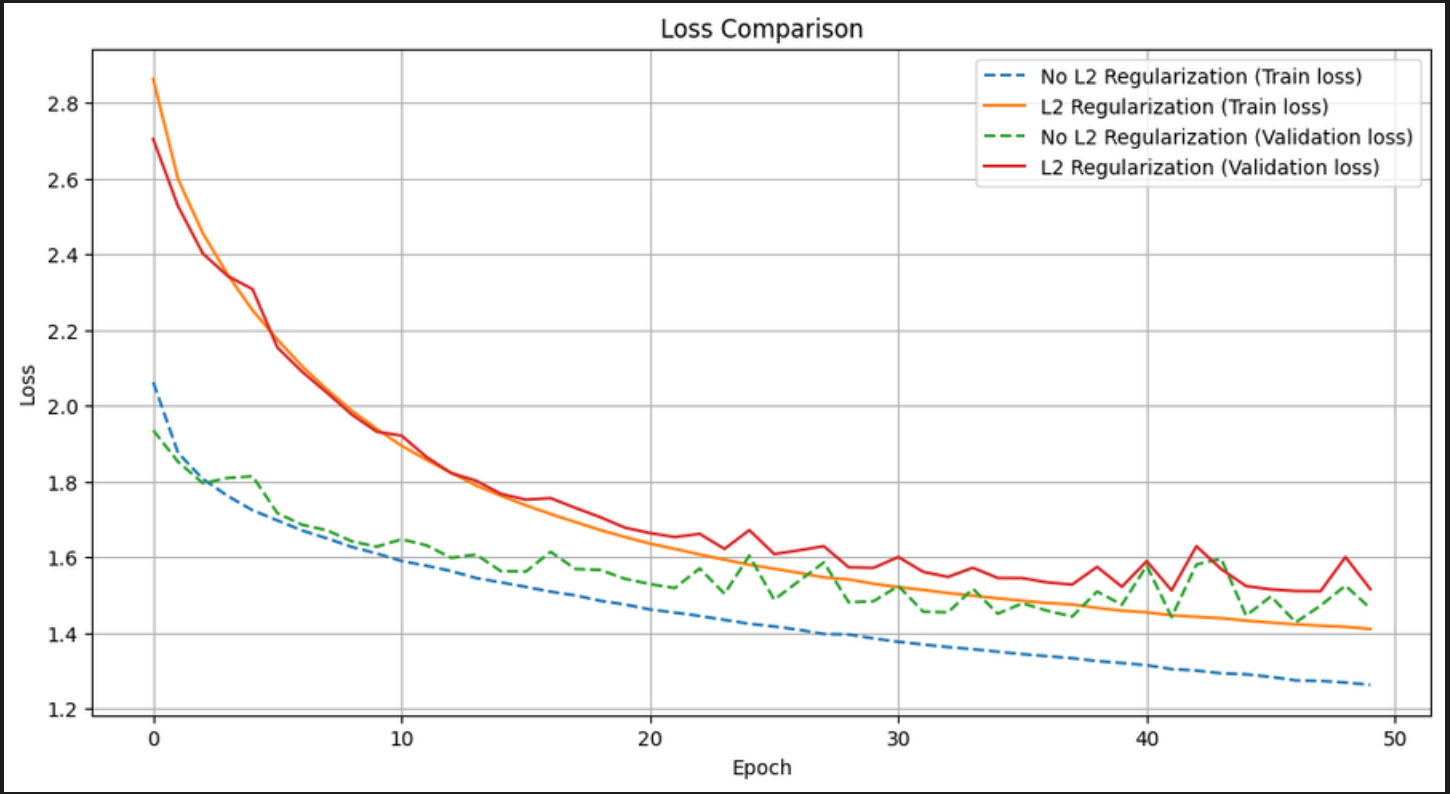

Figure 20: Loss Comparison

Comparison between Dropout 0.2, 0.3, 0.5

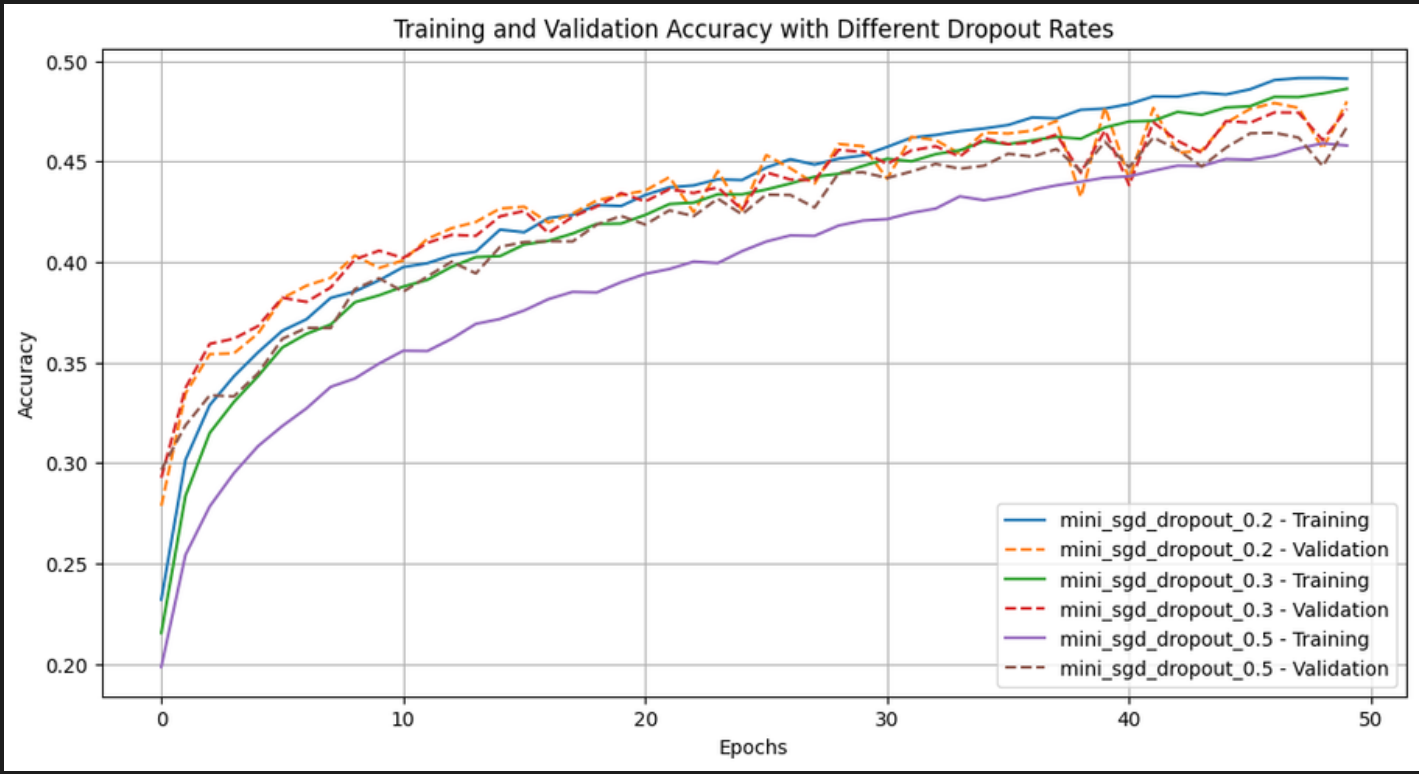

Figure 21:Accuracy with Different Dropout Rates

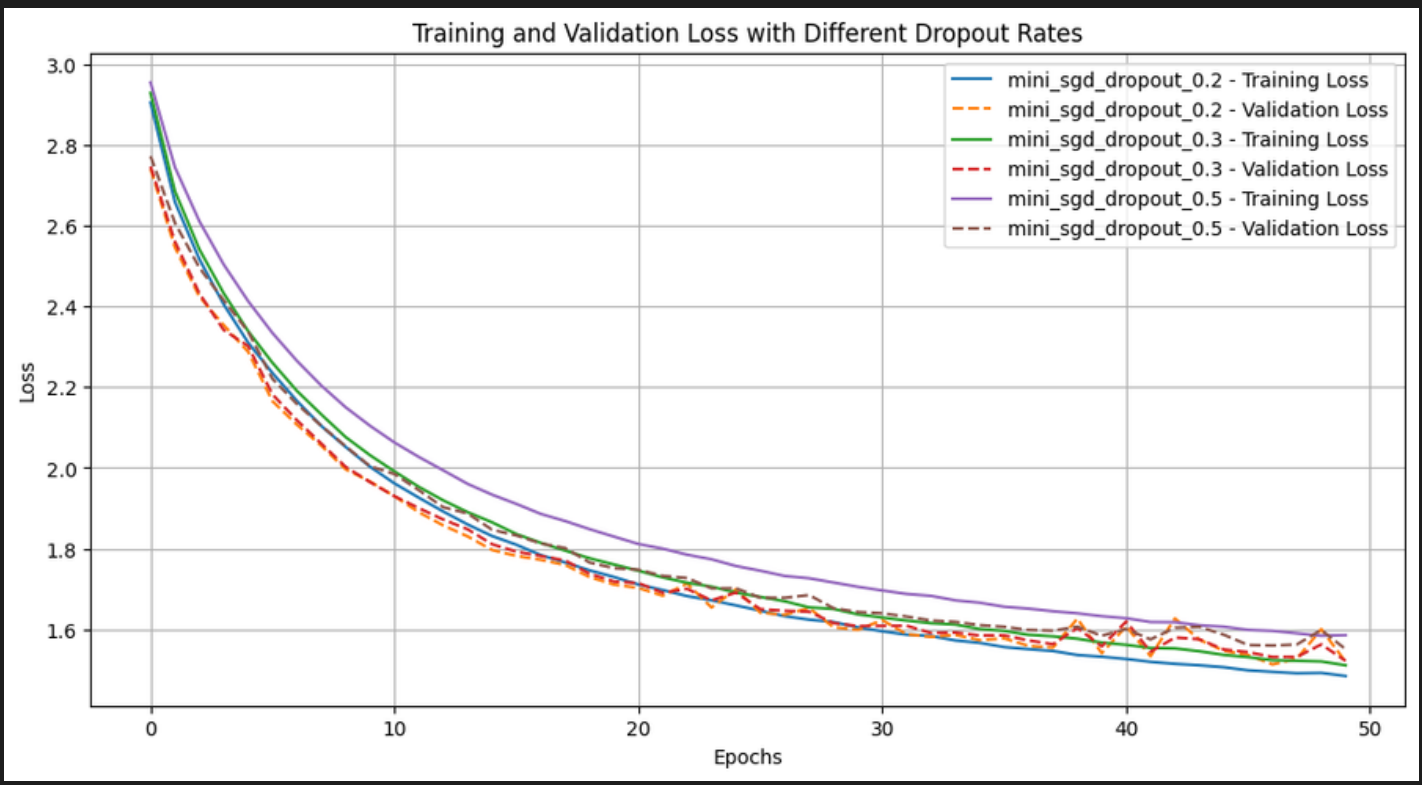

Figure 22 : Loss with Different Dropout Rates

We can clearly see that **`Dropout 0.3 `** performed better when it comes to the observation of the variance between the training set and the validation set (**`accuracy : 0.4890`, `val_accuracy: 0.4829`**)

Where as **`Dropout with rate 0.2`** performed better in the training set (**`training set accuracy =0.4978 val_accuracy: 0.4780`**)
both rates gave approximate results, At the end, we will go with **`Rate =0.3`**

Using early stopping

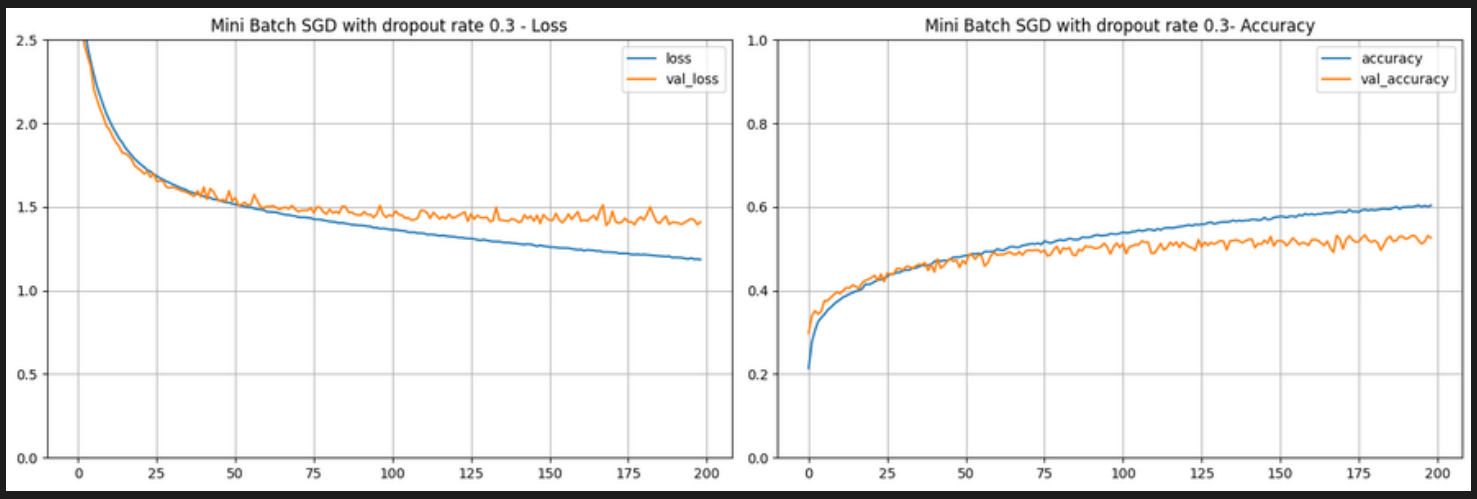

Figure 23 :Loss and Accuracy with Dropout Rate = 0.3 and early stopping

```
# Define EarlyStopping
callback = keras.callbacks.EarlyStopping(monitor='val_loss',verbose=1,patience=30)
```

Training stopped at epoch: **199**

Adding a batch Normaliztion Layer

```
Dense(128, activation='relu', name='hidden_layer_1'),
# Add Batch Normalization
keras.layers.BatchNormalization(),
```

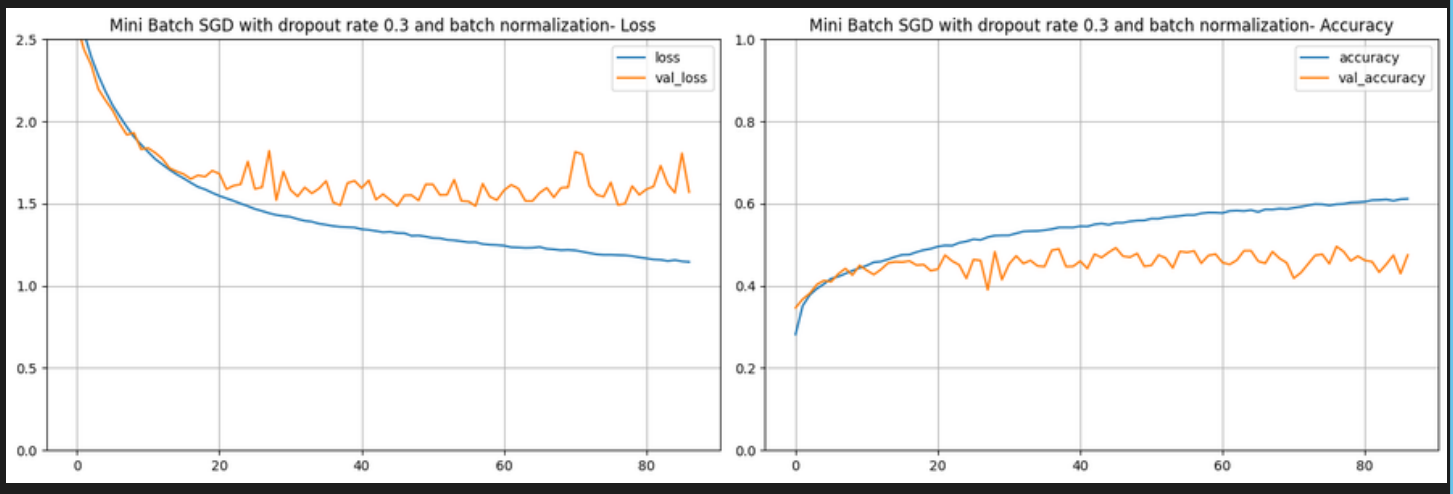

Figure 24 : Loss and Accuracy with Dropout Rate = 0.3 and Batch Normalization

Usualy **Batch Normalization** improves **Neural network training** by reducing internal covariate shift and accelerating convergence, leading to better performance.
Here, in our dataset **``we didn't see the best result``** (pretty much the same results with and without batch normaliztion)

Random Search

```
distributions = {
    'learning_rate': [0.001, 0.01, 0.1, 0.2],
    'dropout_rate': [0.1, 0.2, 0.3, 0.4],
    'batch_size': [32, 64, 128, 256]
}

clf = RandomizedSearchCV(KerasClassifier(build_model,learning_rate=0.001,dropout_rate=0.3),
                         distributions, n_iter=10,cv=3,verbose=3,random_state=1)
search_model = clf.fit(x_train, y_train, epochs=50,  validation_data=(x_cv, y_cv))
```

```
print('Best Parameters: ', search_model.best_params_)
```

```
Best Parameters:  {'learning_rate': 0.01, 'dropout_rate': 0.2, 'batch_size': 64}
```

Training the given best model by the random search:

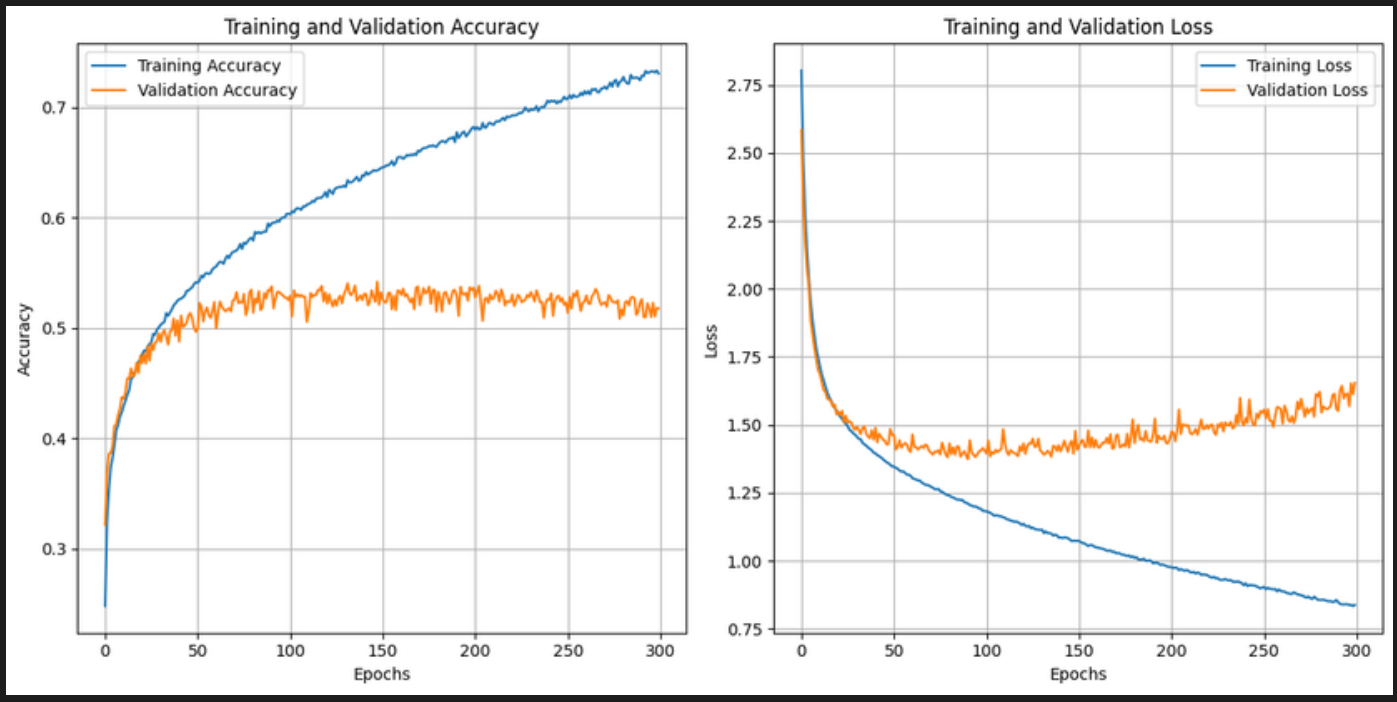

Figure 24 : Loss and Accuracy of the chosen parameters.

Comparing training time:

This comparison is not really accurate because when we used early stopping we updated the number of epochs to 300, in order to observe the early stopping! , overall there's no big difference when it comes to training time.

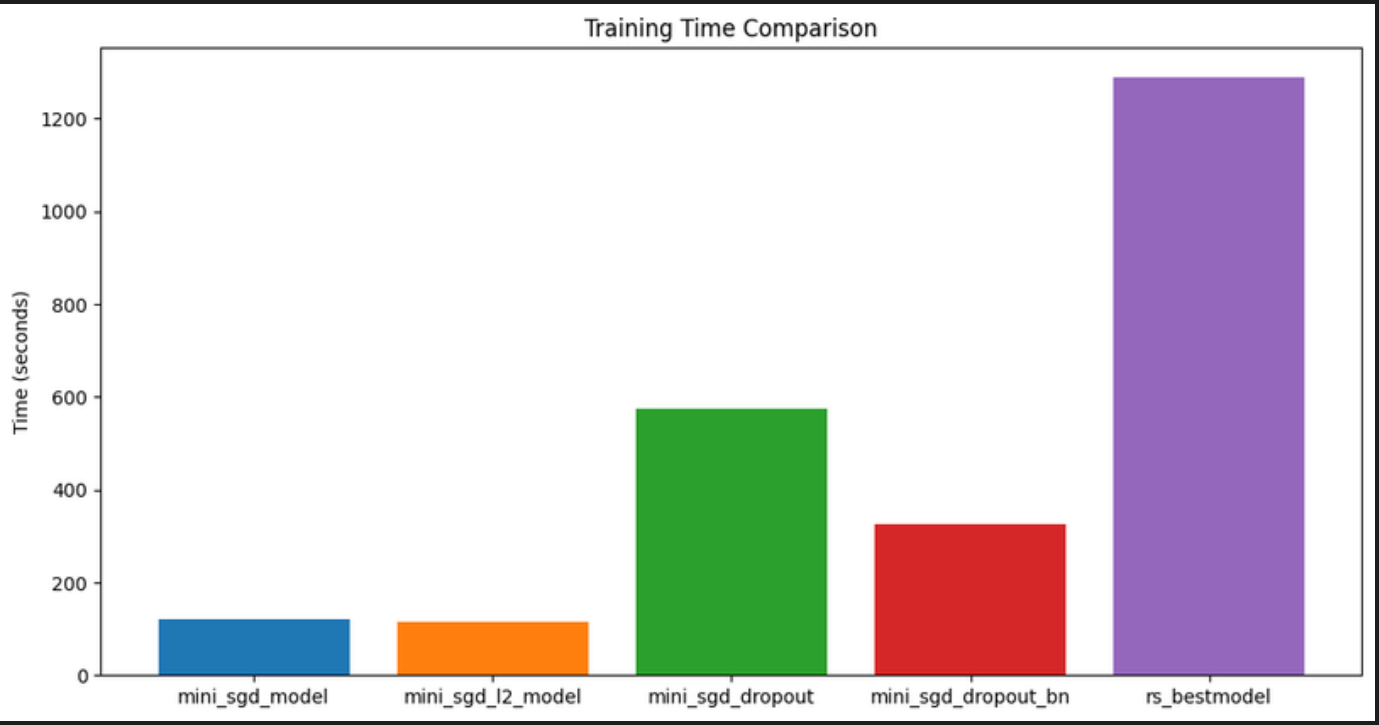

Figure 25 : Training times

Conclusion

In the world of machine learning, especially in deep learning, there are tons of settings to adjust called hyperparameters. It's like fine-tuning a musical instrument – you have endless options to make it sound just right. But not all adjustments are necessary for every situation. It's important to grasp the problem you're dealing with and apply only the tweaks that are really needed. This way, you can get the best results without overwhelming yourself with unnecessary complexity.