Mini Project 3 - COMP 551 W2022

Sung Jun Lee, Joseph Boehm, Namdar Kabolinejad - April 3, 2022

1 Abstract

In this project, we assess the performance of different MLP and CNN models on the Fashion-MINIST dataset, where the training and testing dataset contained 60,000 and 10,000 gray-scaled images, respectively. We attempt to build our own MLP model from scratch with support for mini-batch stochastic gradient descent and use the prebuilt TensorFlow CNN model for comparison. We seek to test and compare the performance of models with varying architectures and hyperparameters. We experiment by changing the number of hidden layers, nodes per hidden layer, activation functions, dropout rates, and learning rates. Our main goal is to observe the effects of changing architectures and hyperparameters on the model accuracy. Additionally, in our experiments we also test the effects of less commonly used additional activation functions as a creative endeavor. We create a linear, cube root, and swish function on top of the ones already tested in the experiment. Through our thorough experiments, we discover that a model using the sigmoid activation function performs the most consistently with a 1 hidden layer with 64 nodes performing the best when trained on a normalized dataset without any dropout, although a 10% dropout rate works just as well. When compared to CNNs from professional machine learning libraries which are specially optimized for images, the final test accuracy achieved on the CNN models surpasses the MLP by about 20%.

2 Introduction

One of the major applications of Machine Learning is in computer vision and image recognition. A major model used for image classification in ML are CNNs which are specialized variants of MLPs. They are both Neural Network models that attempt to gain an in-depth understanding of complex real-world scenes by performing various types of recognition tasks on visual data. Given that CNNs can understand spatial relations between pixels of images, they tend to outperform MLPs in image classification. Moreover, when using a specific model, the specific architectures and hyperparameters used in the model affect the performance greatly [3]. In this project, we are focused on examining the accuracy of a variety of MLP and CNN architectures on a 10-class image classification task based on the popular Fashion-MNIST dataset which contains 28x28 grayscale images of clothing [1]. We start by importing and preparing the data. Moreover, we build multiple MLP architectures from scratch and use the TensorFlow CNN model to compare their training accuracies. We finally train and test the models on a version of the Fashion-MINIST dataset and tune the hyper-parameters to optimize their performance. Out of the MLP architectures used, the model with 1 hidden layer and 64 nodes had the best accuracy of 69.49% when trained on a normalized dataset without any dropout and using the Sigmoid activation function, however, the model with 3 hidden layers and 128 units and Cubic root activation function had a similar test accuracy of 69.48%. State-of-the-art models and researchers have achieved nearly 87% accuracy using more complicated MLP models than ours [1]. On the other hand, the accuracy of around 90% achieved on the CNN model is similar to the result presented in other research [2]. We seek to achieve similar or perhaps slightly lower accuracies by using primitive MLPs and basic hyperparameters.

3 Datasets

We use Keras to directly load the Fashion-MNIST dataset. The dataset contains 60,000 28x28 grayscale images of 10 fashion categories, along with a test set of 10,000 images. This dataset is often used as an alternative for the popular MNIST dataset. The set contains the following 10 class labels: T-shirt/top, Trouser, Pullover, Dress, Coat, Sandal, Shirt, Sneaker, Bag, Ankle boot.

To better understand the dataset prior to any training, we first sought to understand the distribution of labels. After plotting the distribution, we discovered that the distribution of labels was uniform. This is an important detail as it means that a randomly guessing model should at least have an accuracy of 10%. Then we preprocess the data. The datasets are loaded from TensorFlow as numpy ndarrays. We initially normalize the data to get a pixel range between -1 and 1 using the `normalize_data()` function. Normalization is important as it creates small contained values for our input vectors which are less prone to overflow issues. We then proceed to flatten the datasets, which is beneficial as working with 2D arrays can be quite challenging. Given the rather large number of training instances available and the fact that the dataset on its own is quite clean, we did not find it necessary to apply image augmentation.

4 Results

For **task 1**, we began by creating three different models, one with no hidden layers, one with one hidden layer with 128 units and ReLU activations and one with 2 hidden layers, each with 128 units and ReLU activation functions. We found that 0.01 or 0.0001 learning rate, 100 epochs, and 10000 batch size worked best through many experiments.

	Train Accuracy	Test Accuracy
0 Layers	69.78	**68.27**
1 Layer	63.24	62.34
2 Layers	62.53	61.64

Table 1: Train and test accuracies for models with varying depths. (lr=0.01 for 0/1 layer and 0.0001 for 2 layers, epochs=100)

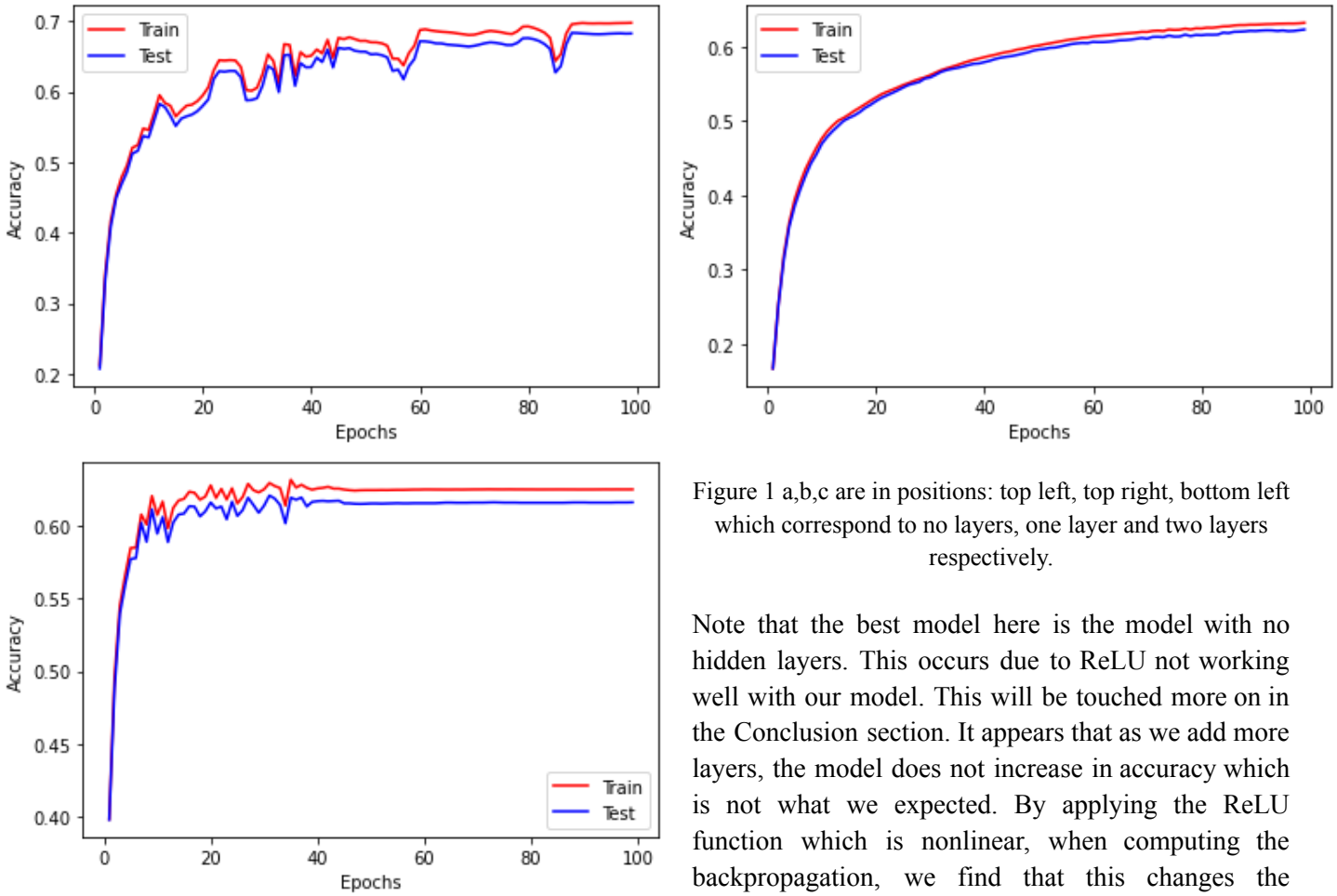

Figure 1 a,b,c are in positions: top left, top right, bottom left which correspond to no layers, one layer and two layers respectively.

Note that the best model here is the model with no hidden layers. This occurs due to ReLU not working well with our model. This will be touched more on in the Conclusion section. It appears that as we add more layers, the model does not increase in accuracy which is not what we expected. By applying the ReLU function which is nonlinear, when computing the backpropagation, we find that this changes the gradients based on the inputs. If the function was linear then the gradient would not depend on the input and lead to a very limited model which would be incapable of modeling the complex relationships that a neural network can support.

For **task 2**, we copy the model from above with 2 hidden layers and each layer has 128 units, we change the activation function from ReLU to Tanh and Leaky-ReLU which leads to the following accuracies shown in the table below. Note that Leaky-ReLU performs the best here, this is due to the gradient of Leaky-ReLU being non-zero when the input value is less than 0; it solves the "dying ReLU" problem found with standard ReLU. For ReLU, we find that when too many of the input values are less than 0, then the gradient becomes 0 for a lot of the inputs which gives very little information. Tanh performs the worst out of all these models, this performs poorly due to the gradient approaching 0 as the inputs get too large or too small. For this test, the results are as expected. We found that 0.1, 0.01 or 0.0001 learning rate (depends on activation function), 100 epochs, and 10000 batch size worked best through many experiments.

Activation Function	Train Accuracy	Test Accuracy
ReLU	62.53	61.64

Leaky-ReLU (alpha = 0.01)	65.72	**64.75**
Tanh	55.75	54.98

Table 2: Train and test accuracies of different activation functions. (lr=0.0001, 0.01, 0.1 respectively, epochs=100)

For **task 3**, we experimented with the ReLU model from task 2 by adding dropout regularization, to do this we created a hyper-parameter that chooses a percent of the model to perform dropout on, the hyperparameter is used to create a list of 1's and 0's which are then multiplied to the weights of the model. As shown in the table below, we can see that the accuracy of the model decreases as we increase the dropout percentage too much which makes sense as we may remove important information from the model. In real life usage, we should probably use < 25% dropout. We found that 0.01 learning rate, and 10000 batch size worked best through many experiments. We reduced the number of epochs to 50 because 100 would sometimes result in overflow.

	Train Accuracy	Test Accuracy
10% Dropout	65.23	**64.51**
20% Dropout	63.69	62.46
30% Dropout	60.45	59.34

Table 3: Accuracies of models with different levels of dropout. (lr=0.01, epochs=50)

For **task 4**, we train and test the model on unnormalized images. We use a model with 2 hidden layers with each layer having 128 units and ReLU activation functions. As shown in the table below, we got lower accuracies compared to the same model trained on normalized images. This makes sense as the range for inputs is a lot larger which can cause the weights and outputs to be more volatile. We found that 0.1 learning rate, 100 epochs, and 10000 batch size worked best through many experiments.

	Train Accuracy	Test Accuracy
Normalized	62.53	**61.64**
Unnormalized	56.06	54.95

Table 4: Accuracies of models trained on normalized and unnormalized data (lr=0.1, epochs=100)

For **task 5**, we use the Keras library to create a CNN with 2 conv2D layers with filter size 32, 64 and 2 dense layers with ReLU activation functions. We found that the model outperformed our model. The reason being that the CNN uses multiple convolution layers which utilizes the fact that we are classifying images and not just a random vector. The CNN achieved a training accuracy of 99.17 and a test accuracy of 90.74. Thus this model performs very well on the Fashion-Mnist dataset compared to our typical MLP models.

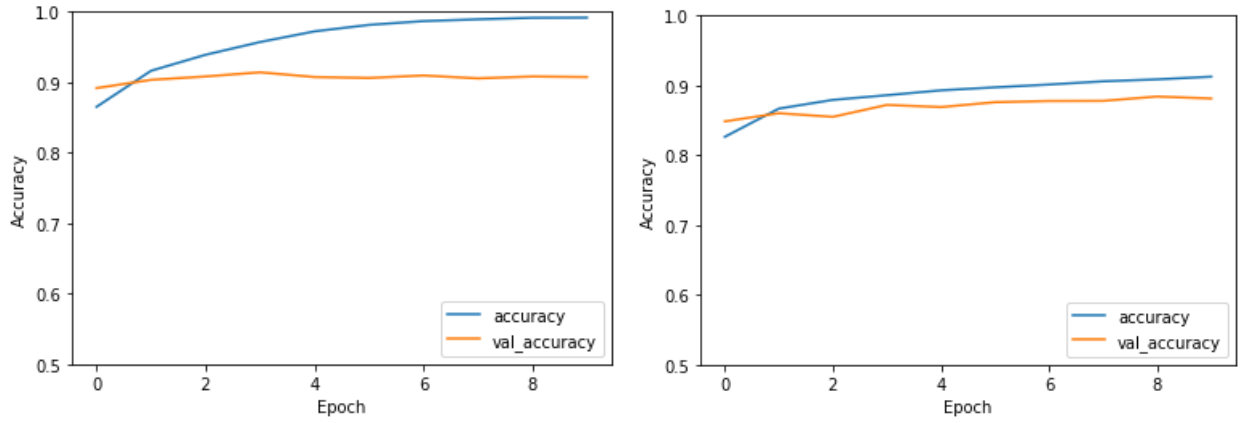

Figure 2: CNN train and test accuracy history (left) and Keras Neural Network training history (right)

Additionally, we also created a Neural Network using Keras with 2 hidden layers where each layer has 128 units and ReLU activation function. We see that the training accuracy of this model is 91.25 and the testing accuracy is 88.13. This outperforms our MLP model made from scratch which is most likely due to the numerical stability provided by the models of the Keras library.

For the last experiment, **task 6**, we attempted to get the best model accuracy using different activation functions, widths and depths for the model. This was done through grid search cross validation. We used ReLU but also created Sigmoid, Swish, Linear, and Cubic Root functions from scratch as additional activation functions. For reference, the Swish function is x*Sigmoid(x). For our creative endeavor, we chose to explore the effects that different activation functions would have on model accuracy. We chose to implement sigmoid and swish because they are widely used in real life settings. We chose to try linear (y=x) because we were curious what the identity function would do to our model. And we created the cube root function from scratch because we figured out that it looked very similar to sigmoid and did not suffer from the vanishing gradient problem. We tested hidden layer numbers of 1, 2 and 3 and widths of 32, 64 and 128 units. As seen in the table below, the Swish activation performs poorly and so does the ReLU function, but ReLU performs poorly due to the learning rate (0.5) being kept constant and we can't fine tune every single model trained as that would take too long. The results from Linear are not included because every test we performed broke down due to numerical instability and did not yield any useful results. The Cubic Root function performs surprisingly well, except the Sigmoid function does perform better than the cubic root and is more stable. The best model is the Sigmoid function with 1 hidden layer and a width of 64 units, it had a test accuracy of 69.49. We found that 0.5 learning rate, and 10000 batch size worked best through many experiments. We had to reduce epochs to 50 because the experiments would take too long.

	Width		ReLU	Sigmoid	Cubic Root	Swish
1 Hidden Layer	32 Units	train_acc	31.30	70.02	69.63	37.29
		test_acc	31.58	69.31	68.18	37.04
	64 Units	train_acc	37.69	70.21	35.90	43.86
		test_acc	37.98	**69.49**	64.98	**43.47**
	128 Units	train_acc	53.38	69.94	40.10	43.94
		test_acc	**52.27**	68.99	40.24	43.39
2 Hidden Layers	32 Units	train_acc	9.99	65.33	57.40	8.39
		test_acc	10.00	64.15	56.62	8.75
	64 Units	train_acc	9.99	69.35	56.80	7.48
		test_acc	10.01	68.50	56.34	7.39
	128 Units	train_acc	9.99	68.82	27.20	7.48
		test_acc	10.01	67.81	27.08	7.39
3 Hidden Layers	32 Units	train_acc	10.02	67.85	37.20	10.00
		test_acc	10.02	66.36	36.66	10.00
	64 Units	train_acc	10.00	68.91	51.03	10.00
		test_acc	10.00	67.57	50.46	10.00
	128 Units	train_acc	10.01	67.78	70.20	10.00
		test_acc	10.03	67.09	**69.48**	10.00

Table 5: Accuracies of a wide variety of models with different widths, depths and activation functions. (lr=0.5, epochs=50)

5 Discussion & Conclusion

From all of our experiments, the main takeaway we learned was that it is surprisingly difficult and unstable to properly train a multilayer neural network from scratch. Because neural networks are inherently very complicated models, there are very many places where things can go wrong, and we learned that hyperparameters are very important to the model's success.

Firstly, the model architecture was quite influential. While we expected that simply increasing the depth and width would perform better, our testing results showed that model width did not really impact the model's accuracy, and that

a large value for depth was not always ideal. Perhaps the task itself is not complicated enough to require a very expressive model. For our CNN architecture for task 5, we opted to choose commonly used filter and kernel sizes for the MNIST dataset upon research.

Next, we discovered that the activation function, and the learning rate were correlated and very important to the model's success. Depending on the activation function, we got drastically different accuracies even though other hyperparameters were kept the same. In our tests, we found that activation functions which keep the output between a certain range or grow very slowly at extremes such as sigmoid, cube root, tanh, performed the most consistently over a large number of trials. Any function which did not taper or have limits at extremes such as ReLU, Leaky-ReLU, Swish, were very hard to train due to their instability. A lot of our initial testing with ReLU resulted in a model which barely learned anything because it either suffered from the "dying ReLU" problem where the gradients go to 0 or the values blew up in magnitude over layers of calculations and resulted in NaN values at the end. Even though a learning rate of 0.5 worked fine for all the other activation functions, for the ReLU style functions, we had to significantly reduce the learning rate in order to prevent these issues. This issue is best seen in the results of task 1 compared to task 6. With significantly smaller learning rates, the ReLU models performed well in task 1 and 2, but in task 6 where the learning rate became much bigger, the model failed to learn anything beyond a 1 hidden layer network. Another aspect that had quite an impact was the initialization of weights. While we initially selected random values from the standard normal distribution, we found that this sometimes resulted in large values, which would often cause overflow for the ReLU based models. We thus found success when we switched to sampling our initial weights from the uniform distribution bound between -1 and 1 then multiplied by 0.1 to further scale down.

The number of epochs was also important. For most models, the majority of the learning was done in the first 20 epochs, but as we let it run to 100, we saw steady increases in accuracy without any overfitting. Perhaps in the future, a higher number of epochs around 500 may be worth investigating to see if this steady trend continues. However, due to resource and time constraints, we opted for 100 epochs. As for dropout, in our experiments, because our models did not overfit, we did not see any benefit to using dropout. In fact, as we increased the dropout percentage, we saw a decline in accuracy. In a future experiment, it would be interesting to test the effects of dropout on an overfitted model.

Lastly, we would like to acknowledge the superiority of utilizing CNNs over standard MLPs for image classification tasks. Using CNNs consistently scored about 20 percentage points higher in accuracy than using MLPs. While our models could not have been fine tuned, we do not believe that we could have achieved around 90% in testing accuracy just using MLPs. In fact, a CNN's ability to leverage spatial/2D information seems to have been quite a bit more influential in the accuracy than we had initially estimated.

6 Statement of Contributions

Sung Jun Lee - Wrote MLP & SGD code, Created/ran multiple experiments, Wrote Conclusions
Joseph Boehm - Contributed to creating the MLP model, Created/ran multiple experiments, Wrote Results
Namdar Kabolinejad - Wrote Preprocessing code, Wrote Abstract, Introduction, Datasets

References

[1] Xiao, H., Rasul, K., & Vollgraf, R. (2017). Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms. doi:10.48550/ARXIV.1708.07747

[2] Chen, F., Chen, N., Mao, H., & Hu, H. (2018). Assessing four Neural Networks on Handwritten Digit Recognition Dataset (MNIST). doi:10.48550/ARXIV.1811.08278

[3] Hsieh, P.-C., & Chen, C.-P. (2018). Multi-task Learning on MNIST Image Datasets.