Assignment 3: Classification of Image Data

Group 5: Adam Motaouakkil (260956145), Frédéric Mheir (260636214), Yann Bonzom (260969653)

Abstract

This project explores ways of performing Machine Learning image classification via neural networks using the Fashion-MNIST dataset. We implemented a Multilayer Perceptron (MLP) class, performed extensive model adaptations and explorations, and compared these models to a Convolutional Neural Network (CNN) as is most commonly used with image classification.

Our results were very much in line with the theory on neural networks. Increasing the number of layers, using the non-saturating LeakyReLU activation function, normalizing the data, applying L2 regularization, and performing extensive hyperparameter optimizations brought our 2-layer MLP model to an testing accuracy of 88%. This reveals the importance of these types of adjustments being made to even simple models, since this accuracy is fairly close to that of a much more complex, advanced CNN which had an accuracy of 93%.

Introduction

The Fashion-MNIST dataset consists of 60K training and 10K labeled images, evenly spread over 10 clothing item categories. To make this model work properly with our MLP implementation, we normalized the input data, one-hot-encoded the output data, and vectorized the images to represent each one as a 1D array of pixel values.

Our MLP implementation is heavily based on Prof Li's code. However, major adaptations were made to make use of mini-batching and stochastic gradient descent, making it feasible to train our models on limited RAM and provide faster learning rates. We also added additional functionalities such as a gradient check via small perturbation to ensure our model calculates gradients correctly during the backpropagation step.

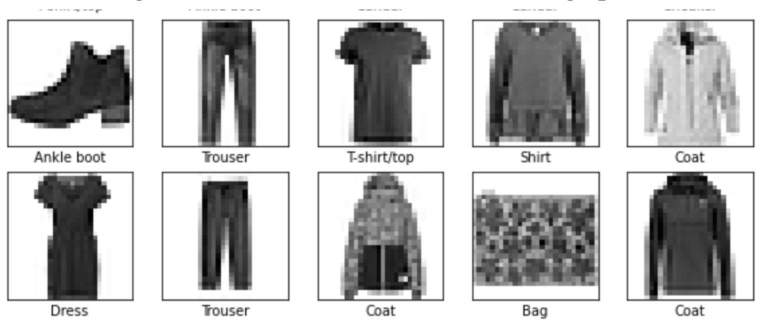
Following extensive experiments, our model yielded accuracy of 82.66% provided by a model with 0 hidden layers, but understood this would still be clearly insufficient in practical use cases. Adding hidden layers improved model performance since the model can extract more complex features from the input images. In combination with the LeakyReLU activation function, this brought our base 2-layer MLP accuracy up by 3.47% to 86.13%. L2 regularization, as expected, further improved model performance since it reduces overfitting to improve the model's ability to generalize to unseen data (with the best accuracy being 88%). Following our grid search exploration to perform hyperparameters, our final model (200 epochs, minibatch size of 200, L2-Regularization, LeakyReLU activation function) achieved a final accuracy of 87%. This is pretty close to the CNN's performance of 93.13%, revealing the immense importance of doing a proper model setup and performing these optimizations to maximize model accuracy, as well as the potential trap of overfitting.

When comparing our model with common Fashion-MNIST classification accuracies found in the literature, we find that our model yields great accuracy and is reliable. Indeed, by looking at *Zalando Research benchmarks for Fashion MNIST*, we see that the highest MLP model obtains 87.7% accuracy, using ReLU. The highest accuracy is 89.7% and uses SVC. However, using other models, it is possible to obtain higher accuracies, as shown in the *Classifying Garments from Fashion-MNIST Dataset Through CNNs* research paper. Their model, consisting of a mix of convolution, max pooling, drop out, and two layers obtained an accuracy of 99.10%. However, it contains 44,426 trainable parameters, which makes it significantly more expensive to train than our model.

Datasets

The Fashion-MNIST dataset contains labeled clothing images from the Zalando online store. There are a total of 10 categories of clothing items (for example, 'T-shirt/top', 'Trouser', and 'Pullover') and there are 60K training examples as well as 10K testing examples. The initial images are grayscale 28x28 pixels, and in both training and testing sets there were an equal number of samples for each class. Following a TensorFlow guide on how to work with this dataset, we performed the normalization of the data by dividing all pixel values by 255 (the max value any pixel can take on); this way, all values have a maximum magnitude of 1 (Tensorflow). We also printed a small subset of the images to see what the data looks like, shown below.

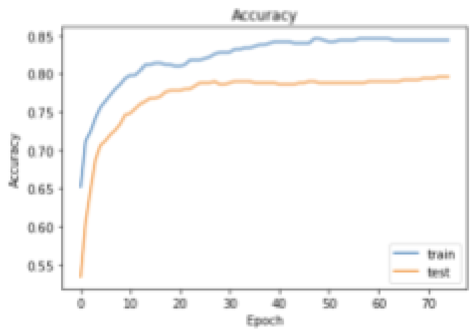
Following this exploration, we processed our data as follows: for the ConvNet model, we kept the data as-is. We saved both normalized and unnormalized input datasets for later exploration. For the final datasets fed into our own models were, we one-hot-encoded the labels and vectorized (i.e., flattened) the 2D image matrices to single size 28x28=784 arrays so it can be fed to the MLP's input layer. Finally, we created small (length 500) random subsets of both the training and testing sets as a way to more quickly evaluate rough accuracies after each MLP training epoch.



Results

Note: In our plots, you may notice that there may be discrepancies between the final test accuracies on the plots and the true test accuracies reported in writing. This is due to the use of subsets of the test and training sets to perform faster evaluations. This significantly helped improve training speeds, while giving accurate plots.

Zero-Layer MLP

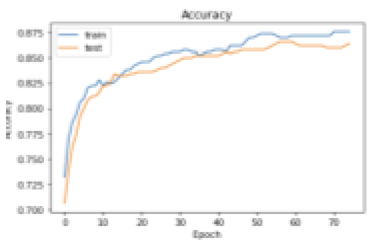


```
Accuracy:  0.8266
```

Testing for Layers

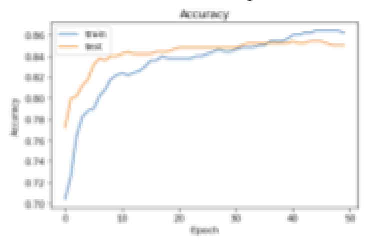
We performed a numerical perturbation check on the weights to ensure that the weights stabilized throughout the fitting of a full batch. We then started our experiments with a mini batch size of 200, 75 epochs, and the ReLU activation function. With no layers, our MLP gave already good results with an accuracy of 82.66%. There are no hidden layers for the MLP to optimize, so it becomes similar to a multi-class regression where the linear regression's weights are updated without accounting for hidden layers' errors. With one hidden layer of 128 units, we obtained a better accuracy of 85.12%. We obtained a similar result (85.08%) with two hidden layers of 128 units per layer. We expect that adding more hidden layers may improve this performance, but due to computational limitations (training time would take unreasonably long due to no GPU optimizations) we did not pursue this path.

ReLU
One Hidden Layer



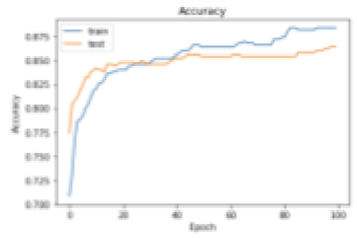
```
ccuracy:  0.8512
```

ReLU
Two Hidden Layers



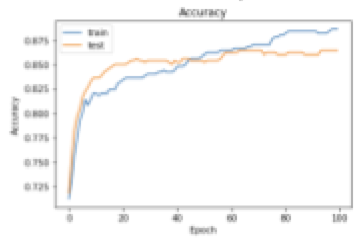
```
Accuracy:  0.8508
```

LeakyReLU
2 hidden layers



```
Accuracy:  0.8613
```

Tanh
2 hidden layers



```
Accuracy:  0.8598
```

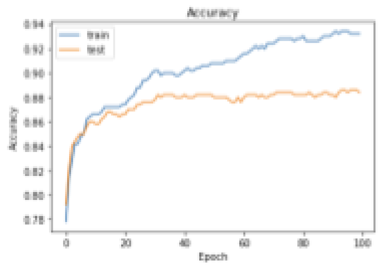
From the charts, we see that the number of epochs play a big role in the accuracy of our model. For instance, less than five epochs would give an accuracy lower than 80%. Although adding more hidden layers provides similar results, it learns faster in fewer epochs. For example, it took about 10 epochs to reach 84% test accuracy with 2 layers, while it took close to 30 with 1 layer. This is likely because the addition of an extra layer allows for the extraction of higher-level features.

Activation Functions and Regularization

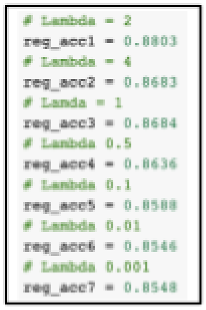
Next, we tested the LeakyReLU activation function on the same 2-layers model, which gave an accuracy of 86.13% after 75 epochs. This is not surprising: with ReLU, if $a < 0$, the entire gradient becomes zero, whereas in LeakyReLU, we still get a non-zero slope. We also tested the Tanh activation function with the 2-layers model. While it gives an accuracy of 85.08%, it behaves like the sigmoid function and, as it saturates, can lead to vanishing gradients which may explain the lower score. Given these results, LeakyReLU provides the best results out of the three activation functions we tested.

We also explored L2 regularization, through which large weights are penalized according to a λ value to reduce overfitting. We tested L2 Regularization with ReLU and different λ. The best accuracy was with $\lambda = 2$ at 88.03%. This increase in accuracy is consistent with other activation functions, obtaining around 88% for regularized Tanh and LeakyReLU.

Regularization
ReLU, Lamda = 2

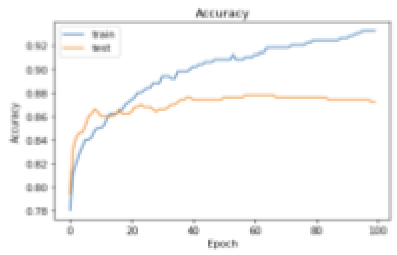


Exploring lambdas



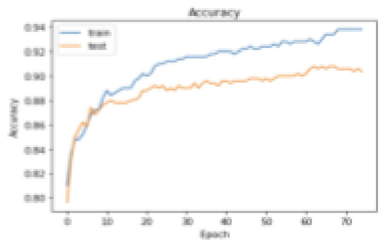
```
# Lambda = 2
reg_acc1 = 0.8803
# Lambda = 4
reg_acc2 = 0.8683
# Landa = 1
reg_acc3 = 0.8684
# Lambda 0.5
reg_acc4 = 0.8636
# Lambda 0.1
reg_acc5 = 0.8588
# Lambda 0.01
reg_acc6 = 0.8546
# Lambda 0.001
reg_acc7 = 0.8548
```

Regularization
Tanh, Lamda = 2



```
Accuracy:  0.8808
```

Regularization
Leaky ReLU, Lamda = 2



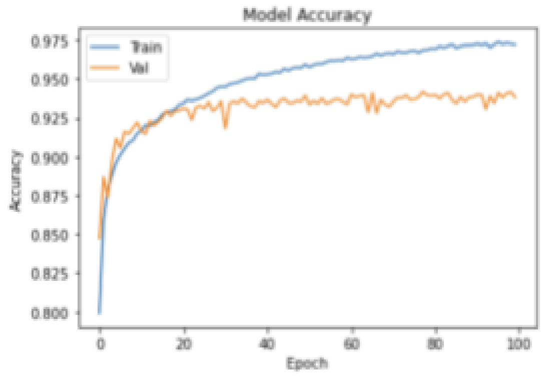
```
Accuracy:  0.8759
```

Normalization, ConvNet, and Model Optimization

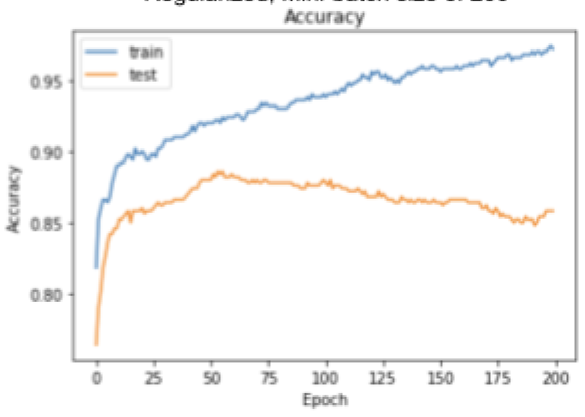
For the non-normalized dataset experiment, we found low accuracies, finishing with 10% and an irregular cross-entropy loss trend. Predictions were missing and adequate accuracies could not be yielded unless the data was normalized. This could be due to the numerical instability of the problem as the values fed into the MLP are very large. A potential solution to this would be to scale the data to make it numerically malleable. Additionally, adjusting the learning rate can help, too. These issues reveal why normalization is an important preprocessing step when training an MLP architecture.

To compare with different model architectures, we implemented a Convolutional Neural Network (CNN), which is the most commonly used architecture for image classification. Following a guide with a purported high testing accuracy, and by using Keras' GPU acceleration, we trained a more complex deep model (Deshpande, 2020). This model utilizes 2D convolution, max poolings, batch normalization and the ADAM optimizer, the ReLU activation function, and varying dropout levels at different layers to reduce overfitting. These additions significantly improve test classification performance with an accuracy of 93.13%. Learning appears to happen very quickly, with test accuracies plateauing after about 30 epochs only. This is likely because this model is able to extract more meaningful features from the images.

ConvNet with Keras



Final Model
200 epoch LeakyReLU
Regularized, Mini-batch size of 200



As mentioned before, using L2 regularization and the LeakyRLU activation function gave the best results. To find the optimal hyperparameters, we performed a grid search to create our most performant model. Using a selection of online guides (see references), we settled for the following: given that our MLP implementation does not make use of GPU acceleration and is thus slow to train (~45 minutes with 2 layers), we used Keras and Scikit-Learn's GridSearchCV using GPU acceleration to find the optimal hyperparameters. We then used these parameters in a final model using our own implementation. Note that we opted not to perform Bayesian optimization via Gaussian process since, via this approach, actually training models over the whole parameter space was very feasible (roughly 1 hour).

Using the optimal hyperparameters we found (batch size 200, 200 epochs, 2 layers, 128 units per layers), we ran the model with L2-regularization and LeakyReLU. The final accuracy is 87.37% in less than 75 epochs. Beyond 75 epochs, testing accuracy decreases while training increases, potentially indicating overfitting. While this is a great accuracy compared to the basic 2-layers ReLU MLP, it is still less than the 93.13% obtained with the ConvNet model.

Discussion and Conclusion

Our best accuracy is 88%, given by L2-regularized LeakyReLU, with two layers, a mini-batch size of 200 and 75 epochs. To improve our performance, there are multiple options available. Since LeakyReLU is the activation function yielding the best result, experimenting with different model parameters would be an option. Despite no notable improvement in our testing from 1 to 2 layers, additional layers should further increase the performance of our model. Another option would be to explore a combination of different activation functions such as Exponential linear unit or GELU.

In conclusion, the MLP exploration showed that model depth increased the rate at which a model learns, requiring less epochs to reach optimal solutions. Activation functions that reduce numerical errors yield greater accuracies, and regularization helps the model generalize better by penalizing large weights. The numerical stability of the MLP is greatly affected by the data's formatting, explaining why normalized data behaves adequately as opposed to non-normalized data. Existing optimized MLP models benefit from GPU optimization as parallel computing speeds up training; this enables fast training of much more complex, deeper models. Grid search helps find optimal parameters, but further testing is required to make sure our models do not behave unusually. Finding the right number of epochs is difficult, as too few epochs give lower accuracy, but too many leads to overfitting which also decreases accuracy. Lastly, the ConvNet demonstrated how efficient a model can be with different numerical manipulations such as convolution, batching, and max pooling.

Statement of Contributions

Adam Motaouakkil and Frédéric Mheir worked together on the MLP implementation and adaptations. Yann Bonzom worked on data exploration and processing. The exploration, tests, and write-up, were evenly split among all team members.

References

Introduction

- *Zalando Research benchmarks for Fashion MNIST:* http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/
- *Classifying Garments from Fashion-MNIST Dataset Through CNNs*, Alisson Steffens Henrique, Anita Maria da Rocha Fernandes, Rodrigo Lyra, Valderi Reis Quietinho Leithardt, Sérgio D. Correia, Paul Crocker, Rudimar Luis Scaranto Dazzi, ISSN: 2415-6698, February 16, 2021: https://www.researchgate.net/publication/349553873_Classifying_Garments_from_Fashion-MNIST_Dataset_Through_CNNs

Dataset and Preprocessing

- *Basic Classification - Classify Image of Clothing*, TensorFlow: https://www.tensorflow.org/tutorials/keras/classification
- *Fashion MNIST Dataset*, Zalando Research: https://github.com/zalandoresearch/fashion-mnist

MLP Implementation

- *Gradient Descent tutorial*, COMP 551: https://colab.research.google.com/github/yueliyl/comp551-notebooks/blob/master/GradientDescent.ipynb
- *Regularization Tutorial*, COMP 551: https://colab.research.google.com/github/yueliyl/comp551-notebooks/blob/master/Regularization.ipynb
- *Perceptron Tutorial*, COMP 551: https://colab.research.google.com/github/yueliyl/comp551-notebooks/blob/master/Perceptron.ipynb
- *Tensorflow on MNIST tutorial*, COMP 551: https://colab.research.google.com/github/probml/pyprobml/blob/master/notebooks/book1/13/mlp_mnist_tf.ipynb
- *1-Layer MLP tutorial*, COMP 551: https://colab.research.google.com/github/yueliyl/comp551-notebooks/blob/master/MLP.ipynb
- *Numpy-L-MLP tutorial,* COMP 551: https://colab.research.google.com/github/yueliyl/comp551-notebooks/blob/master/NumpyDeepMLP.ipynb
- *Lectures*, COMP 551, Professor Yue Li

ConvNet implementation - Colab code heavily inspired by this source

- *Fashion MNIST/CNN Beginner (98%)*, Rutvik Deshpande, Kaggle: https://www.kaggle.com/code/rutvikdeshpande/fashion-mnist-cnn-beginner-98/notebook

Hyperparameters Tuning with Grid Search - Colab code heavily inspired from those sources

- *How to Grid Search Hyperparameters from Deep Learning Models in Python with Keras*, Jn Brownlee, July 2 2022: https://machinelearningmastery.com/grid-search-hyperparameters-deep-learning-models-python-keras/
- *Tuning the hyper-parameters of an estimator*, scikit-learn: https://scikit-learn.org/stable/modules/grid_search.html
- *AutoDiff-MLP tutorial*, COMP 551: https://colab.research.google.com/github/yueliyl/comp551-notebooks/blob/master/AutoDiffMLP.ipynb#scrollTo=g22608tCOA3M