COMP 551 Assignment 3: Neural Network Classification of Image Data

Gulan, Sinan Can; Strizak, Jana; Wen, Zehai (Group 27)
2022/12/05

Abstract: *In order to classify images of clothing articles, we implemented Multi-Layer Perceptrons(MLP) and Convolutional Neural Networks. The number of hidden layers in the MLP was modulated and the best performance was determined by the test accuracy of* 62.35% *to be with 2-hidden layers. Leaky-ReLU was implemented at various negative slopes, with the best being at* 72.53% *when* $s = 2$*, which outperforms the regular ReLU case. The hyperbolic tangent activation produces a low accuracy of* 56.72%*. Adjusting fit batch size produced the highest accuracy* 62.57% *with a size of 128 data points. The unnormalized dataset produced varying results based on the initial conditions of the gradient descent, swinging between* 9.99% *and* 72.23%*. Regularization was implemented and showed worse performances, with the best being* 58.71% *during output layer regularization. The optimal layer is determined to be leaky-ReLU with* $s = 2$*, a batch size of 256, and weight variance of 0.01 on first layer followed by 0.1 on subsequent layers. The best model produced a test accuracy of* 78.78%*, better than the next best PyTorch implementation of LeNet at* 78.48%*.*

Introduction

The Multi-Layer Perceptron (MLP) and the Convolutional Neural Network are two essential modern image classification tools. In this assignment, we implement an MLP from scratch and a Convolutional Neural network using PyTorch to perform a classification task on the Fashion data set provided by the Modified National Institute of Standards and Technology database, which we abbreviate as "Fashion-MNIST". Fashion-MNIST is a benchmark data set for machine learning algorithms. Every data point in the Fashion-MNIST dataset is an 28×28 grayscale (i.e. pixel value chosen from $\{0, \cdots, 255\}$) image with a label taken from the set $\mathcal{C} =$ {T-shirt/top, Trouser, Pullover, Dress, Coat, Sandal, Shirt, Sneaker, Bag, Ankle boot }. It is a more complex extension of the original MNIST handwritten digits data set, which was studied by [5] and has been cited in many papers. One year after its creation, the dataset had been cited in 260 academic papers [4], with Google being the most popular, then University of Cambridge coming in second, and IBM third. A google research paper aimed to identify if a test example belongs to the same distribution as the training data using self-supervised learning for anomaly or outlier detection. This one-category classification showed strong performance on the Fashion-MNIST dataset, with a Rotation Prediction method including kernel density estimation producing the best AUC result [2]. In a completely unpredictable direction, a UCLA study created an all-optical diffractive deep neural network architecture, which uses light and filters to act as neural network layers, which either transmit or reflect the rays coming from the MNIST image. This optical image classification method produced 91.75% classification accuracy [1]. The best classification accuracy reported on this dataset to date was by [3], with a 96.91% classification accuracy. The team fine-tuned a Differential Architecture Search algorithm, a stochastic gradient descent model that includes minimising the validation loss using the delta rule. A initial learning rate of 0.025 was used and updated with each iteration. The SGD included moment and weight decay parameters, and 2 sets of validation were performed, one for architecture search (40% of training) and another for hyperparameter tuning (15% of training). Our goal is to train models with MLPs and convolutional neural networks such that, given an 28×28 grayscale image, our model can tell which category of $\mathcal{C}$ the given image belongs to.

Before we explain how we construct our MLP, let us clarify our notations. Let $f : A_1 \times \cdots \times A_n \to B$ be a function, where the sets A_1, $\cdots$, A_n, and B are Euclidean spaces. Given $(a_1, \cdots, a_n) \in A_1 \times \cdots \times A_n$ and $i \in \{1, \cdots, n\}$, one can always define a function $g : A_i \to B$ by $g(x) = f(a_1, \cdots, a_{i-1}, x, a_{i+1}, \cdots, a_n)$. If f is differentiable at $(a_1, \cdots, a_n)$, then g is differentiable at a_i. In this case,**we will write** $\frac{\partial f}{\partial a_i}(a_1, \cdots, a_n)$ **in place of** $Dg(a_i)$, the total derivative of g at a_i.

Let D, C, and T be positive integers. By a **MLP**, we mean the tuple $(f, \{f^t\}_{t=0}^T)$ of a function $f : \mathbb{R}^D \to \mathbb{R}^C$ and a finite sequence of differentiable functions $\{f^t\}_{t=0}^T$ satisfying the following two requirements:

1. The domain of each f^t is the cartesian product of two Euclidean spaces: one space for the input x^t and the other for the (optional) parameter θ^t. f^t maps (x^t, θ^t) to $f(x^t, \theta^t)$, which again belongs to some Euclidean spaces.

2. Given all the θ^t's, the dimensions are controlled such that the following equation is well-defined and

can be treated as the definition of $f(x)$, for every $x \in \mathbb{R}^D$:

$$f(x) = f^T(f^{T-1}(\cdots(f^0(x,\theta^0))\cdots,\theta^{T-1}),\theta^T)$$

In other words, let $x \in \mathbb{R}^D$ be given. The following are well defined: $x^0 = x$, $x^1 = f^0(x^0,\theta^0)$, $\cdots$, $x^{T+1} = f^T(x^T,\theta^T)$. And $f(x) = x^{T+1}$.

Each f^t is known as a **layer** of the MLP. We call the collection $\{f^t\}_{t=0}^T$ a **neural network**.

The definition we made is more general than what was made in most texts. To get our MLP, we narrow it down by imposing furthermore the following constraints:

1. T is odd and $f^T : \mathbb{R}^C \to \mathbb{R}^C$ does not have any parameter and is the **softmax** function $f^T(x_0,\cdots,x_{C-1}) = (\sum_{c=0}^{C-1} e^{x_c})^{-1}(e^{x_0},\cdots,e^{x_{C-1}})$
2. Let $t \in \{0,\cdots,T-1\}$. If t is odd, then f^t does not have any parameter and is known as an **activation function**. If t is even, then f^t is affine linear with slopes/intersections being the parameters. We call f^t a **linear layer**. More precisely, there exists a real matrix W^t and a vector b^t such that $\theta^t = (W^t, b^t)$ and $f^t(x^t) = W^t x^t + b^t$. W^t is known as **weight** and b^t is known as **bias**.

We call f^0 the **input** layer and f^{T-1} the **softmax output layer** to convert result to probabilities. The **number of hidden layers** or **depth** is defined to be $\frac{T}{2}-1$ and is equal to the number of activation functions.

By a **loss function**, we mean any function $J : \mathbb{R}^C \to [0,\infty)$. By **training** a MLP, we mean we update all parameters θ^t such that the expectation (i.e. average) of $L(x) = J(f(x))$ is minimized with x ranging over some subset of $\mathbb{R}^D$. In this assignment, we always use **cross entropy** to quantify the error. Given training example $(x,y) \in \mathbb{R}^D \times \mathbb{R}^C$, we write $\hat{y} = f(x)$ and put:

$$J(y) = -\sum_{c=0}^{C-1} y_c \ln \hat{y}_c$$

There will be an experiment which we need to perform regularisation on layer t. This means, instead of working with L, we work with $L' = L + r\left|\theta^t\right|^2$ for some fixed (small) constant $r > 0$.

After loading (and standardizing if required) the data, we train the MLP with this loss function. We implement mini batch stochastic gradient descent as follows: first we initialize all the weights and bias randomly, with mean 0 and some positive **parameter variance**. For each iteration, we pick a small part (a **batch**) of the training set. Let a training example $(x,y) \in \mathbb{R}^D \times \mathbb{R}^C$ be given. Define $x^0,\cdots,x^{T+1}$ as before. We see that, by chain rule:

$$\frac{\partial L}{\partial \theta^t} = \frac{\partial L}{\partial x^{T+1}}\frac{\partial x^{T+1}}{\partial x^T}\cdots\frac{\partial x^{t+1}}{\partial \theta^t}$$

All quantities are evaluated at the current parameters and at x^0, $\cdots$, x^{T+1}. Therefore, $\frac{\partial L}{\partial \theta^t}$ is known for every t. At the i'th iteration, we pick a mini-batch, compute the **average** of $\frac{\partial L}{\partial \theta^t}$, and update:

$$\theta^t = \theta^t - \alpha_i(\text{average of}\frac{\partial L}{\partial \theta^t}\text{ on the batch}); \quad \alpha_i = \frac{\lambda}{i+1}$$

Here $\lambda > 0$. This definition is thanks to the **Robbins-Monro schedule**.

During experimentation, unless explicitly stated, the reader should assume we're training and testing with the standardized dataset (centered and normalized) dataset, using our default configurations for the hyperparameters. These default configurations are: $\lambda = 0.05$ in learning rate, 128 hidden units for any activation function, 2000 training iterations, batch size = 30, parameter variance = 0.1, depth = 2, activation functions = $relu$, and $r = 0.01$ in regularization.

We have carried out 6 experiments investigating the following phenomena. In order, we investigative the effect of model depth, activation functions, normalized data, regularization, and convolutional neural networks on accuracies. Our results are summarised as follows:
1. We tested five perceptrons with depths $\{0,1,2,3,4\}$ respectively. (Note that the perceptron with 0 layers reduces to linear multiclass prediction using softmax.) We saw an increase in accuracy up until depth = 2: the perceptron with 2 hidden layers achieved the best testing accuracy of 62.37%. After that point, we saw a decrease in performance with each layer added. We suspect that when there are too many layers, too much information is discarded by ReLU and the model overfits, which causes the accuracy to decrease.
2. Since tanh is saturating, it performs worse than ReLU, whereas Leaky ReLU showed great performance with 72.53% accuracy. We also tested different scalars for Leaky ReLU and surprisingly found that the model performs best with the unusual choice of negative slope = 2.
3. Testing with different batch sizes, we saw that accuracy is positively correlated with batch size. This

is most likely because larger batch sizes allow for our model to consider more data.
4. The default parameter variance (0.1) is unsuitable for training on unnormalized data. However, when we change the parameter interval to 0.01, we get 72.23% accuracy, which suggests our model performs better with even lower variance parameters.
5. The regularized models has a slighly lower accuracy than the same model without regularization. This is expected as our weights are not that large anyways and therefore regularization is unnecessary.
6. As expected, the use of convolutional layers bolted our testing accuracy around the 80% range.

Data Sets

Fashion-MNIST presents its training/testing data set in the format of two arrays. The X array has shape $N \times D$, where N is the number of data points and $D = 784 = 28 \times 28$. The Y array has shape N and has values being one of $\{0, \cdots, 9\}$ corresponding to the set $\mathcal{C}$. For training, $N = 60000$. For testing $N = 10000$. We emphasise that we always use **numpy float64** for anything related to X and **numpy uint8** for anything related to Y.

The dataset is uniformly distributed:

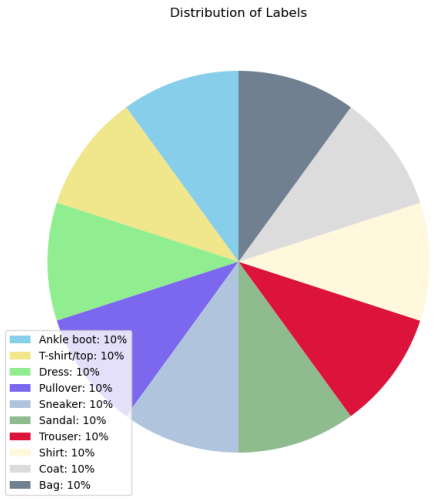

Figure 1: Class Distribution of the Training Set

By **standardization** (or **pre processing**), we are referring to the **centering** (subtraction of mean) and **normalization** (division by standard deviation) of the data, to ensure a mean of 0 and a standard deviation of 1. This will decrease the range of the input data, consequently improving numerical stability and solution convergence. Although MLP is not scale-sensitive, we chose to normalize the data anyway, which proved to be significant as we saw a negative correlation between input variance and model performance, which is further discussed in the following sections.

Results

As previously mentioned, throughout all experiments, we fix the number of epochs to be 2000, the number of hidden units to be 128, and $\lambda = 0.05$ for the learning rate. The number of iterations is chosen to be 2000 because we will be using batch size 30 and we want to use the whole training set once, which has 60000 examples, however we do experiment with larger batch sizes. The choice of learning rate was made on the basis that it allowed numerical stability as well as convergence within a reasonable amount of time. ***We do not have enough space for the accuracy graph of every experiment. Please refer to our code.***

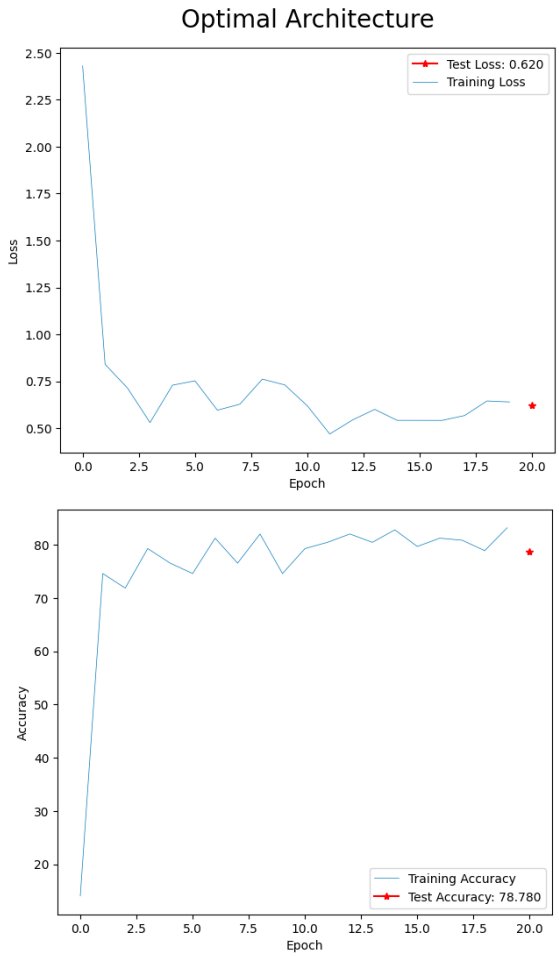

Figure 2: Train/Test Performance of the Best Model

Effect of Depth on Accuracy

The accuracies at various MPL layer depths are displayed in figure 1. Accuracies are relatively close in all tested cases, from no hidden layers to 5 ReLU hidden

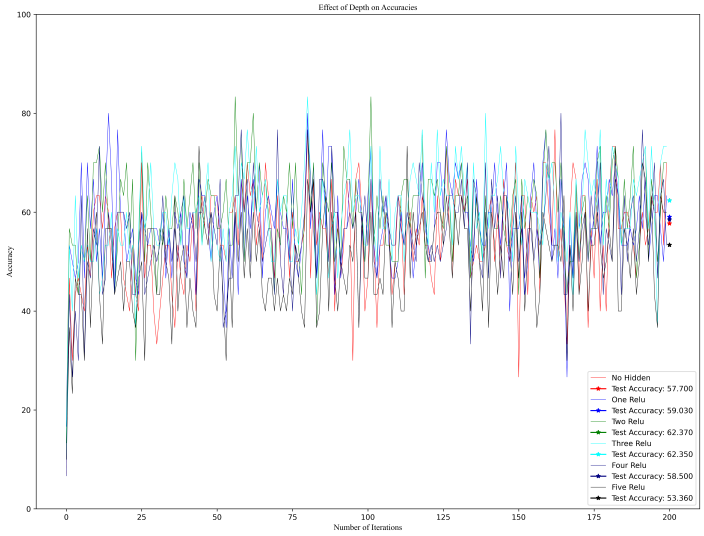

Figure 3: Effect of Depth on Accuracies

layers. The optimal with a test accuracy of 62.37% is the 2-layer case, with the 3-layer ReLU being close by at 62.35%. The worst was the 4-hidden layer MLP with an accuracy of 58.5%. The noise in the accuracy is large as it fluctuates greatly during the fitting iterations with a slow general upward trend.

Effect of Activation Functions on Accuracy
The test accuracies for various activation functions are shown in figure 2.The best test accuracy was 72.53% for the leaky ReLU case with negative slope = 2. The ReLU models with smaller scalar values performed slightly worse, with accuracies decreasing with the ReLU scalar parameter to 65.03% and 62.87% with $s = 0.2, 0.02$, respectively. The worst method was with a hyperbolic tangent activation which produced a prediction accuracy of 56.72%.

Effect of Batch Sizes on Accuracy
The test accuracies as a function of batch sizes is shown in the green of figure 2. As expected, the accuracy jumps from 60.14% to 62.01% when batch size increases from 2 to 15, respectively. With larger batch sizes, the change gain in accuracy decreases with each experiment, with a maximum at the largest tested batch size of 128 at 62.57%.

Effect of Normalization on Accuracy
When the model is trained on un-normalized data, the training and testing accuracy falls significantly to 9.99%. To balance out this phenomenon, we initialised our weights with lower variance (0.01) to see a significant increase in accuracy 72.23%, suggesting that our model enjoys low-variance.

Effect of Regularization on Accuracy
The Regularization accuracies are shown in orange figure 2. The accuracies do not significantly change with layer regularization. the highest accuracy was when the regularization was applied to the output layer leading to an accuracy of 58.71%, with the lowest being at the first layer with 56.44%. These accuracies are all smaller than in the unregularized case 2 hidden layer MLP case 62.37%.

Best Model
We've found the best performing architecture to be: two hidden leaky ReLU (negative slope $s = 2$) layers with 128 hidden units, batch size 256, parameter variance 0.01 at first layer and 0.1 at subsequent layers. This model converged fairly quickly and achieved 78.78% in accuracy, which is comparable with convolutional networks mentioned in the following section.

Convolutional Neural Networks
Using PyTorch have built a CNN that predicted with 75.66% accuracy, making it better than most of our models tested. This CNN has two convolution layers (output size = 3, 6 ; kernel = 2, 4 respectively)with batch normalization and max pooling (kernel = 2, stride = 2). It has an accuracy close to our optimal model. We have also made a 'modernized' implementation of the pioneering LeNet which has great accuracy with 78.48%. Differently than the classic LeNet, our implementation uses Leaky ReLU with negative slope 2 instead of the saturating tanh, as well as batch normalization at each convolution layer.

Discussion and Conclusion

Effect of Depth on Accuracy
We can see that the depth increases performance as we go from 0 to 2 layers. The highest accuracy is achieved when we have exactly two layers. Afterwards, the accuracy decreases as we increase the layers. The third relu layer does not change much accuracy but the later ones decrease a lot. This result is expected. When there are not many layers, the model is too simple and cannot give good results. When there are too many layers the model overfits the training data, causing a decrease in testing accuracy. Our key takeaway here is that adding complexity is not always beneficial: sometimes less is more. We suggest curious readers to look into models that can dynamically set their depth.

Effect of Activation Functions on Accuracy
Since the hyperbolic tangent is a saturating activation function, it performs considerably worse than ReLU and Leaky ReLU. While normally $s \in (0, 1]$ is chosen for the negative slope of Leaky ReLU, our model performed best with this constant set to 2. Further investigation as to why this is the case and also experimen-

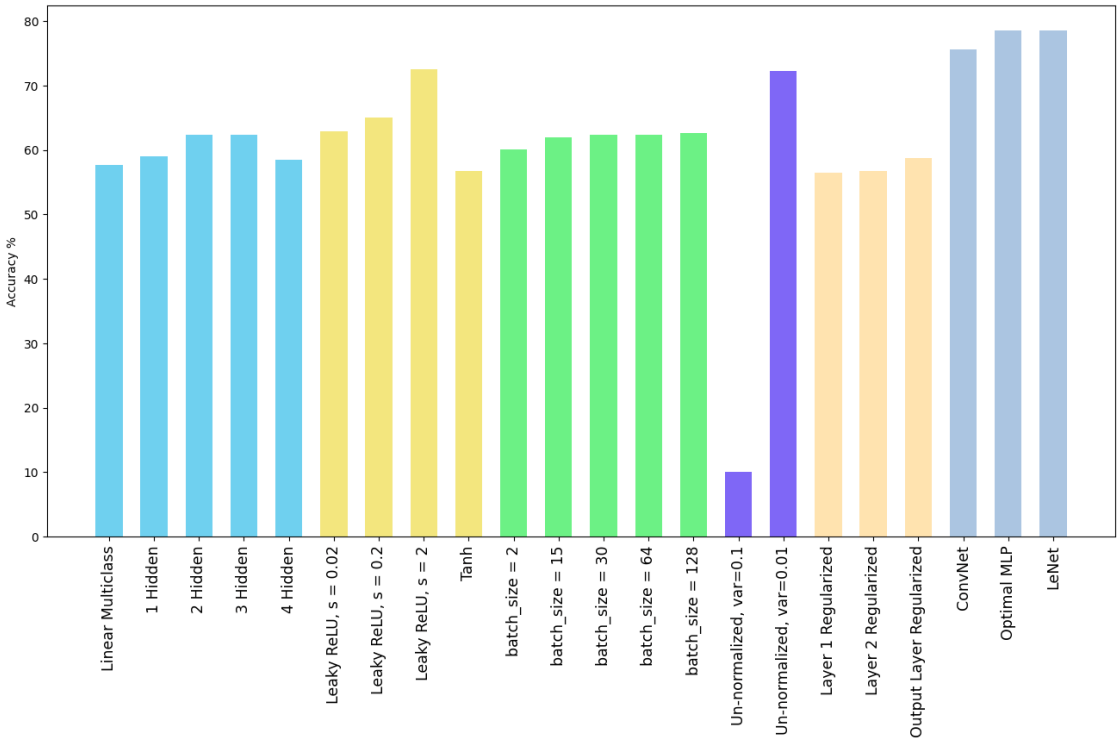

Figure 4: Test Accuracies for All Experiments

tation with this choice of scalar on different datasets should be done.

Effect of Batch Sizes on Accuracy
It is very exciting to see that the accuracy increases monotonically as the batch size increases. We are seeing the law of large numbers in action!

Effect of Normalization on Accuracy
Our observation shows that the normalized data can tolerate a higher parameter variance, which is easier to train. Unnormalized data contains more information from the original distribution. It is harder to control. But, it gives better results if it can be trained.

Effect of Regularization on Accuracy
When a model is trained with regularization, the update is not fully according to the gradient. This will definitely decrease the testing accuracy, which is reflected in our results. Therefore, whenever ordinary training is possible, it should be preferred.

Convolutional Neural Network
Convolution is certainly one of the most important revolutions in the field of image recognition. Simply by adding two layers of convolution onto our MLP, we see an accuracy increase of almost 10%. Although our optimal architecture surprisingly beat the CNN, this is natural as each of its parameters were carefully tuned as a result of a multitude of experiments. In fact, as mentioned before, there are many CNN models, which rely on more complex methodologies, that attain even higher than 90% accuracy on this dataset.

Further, as MLP is by nature not translation invariant, we would see a decrease in performance if the dataset contained reversed, clipped or tilted images, whereas even the simplest CNN would outperform an MLP in this case. Therefore it is important to take our results with a grain of salt.

In conclusion, this project implemented MLPs from scratch and CNN from Python's PyTorch. Many parameters, such as the number of hidden layers, type of activation function, batch size, data normalization, regularization and convolution, along with their effects, were investigated. The best model we came up with is a two hidden 128 unit Leaky-ReLU with a negative slope of 2, batch size of 256, and a parameter variance of 0.01 on first and 0.1 on subsequent layers, producing an accuracy of 78.78%. This is comparable in performance, to our PyTorch implementation of 'modernised LeNet' with an accuracy of 78.48%.

Statement of Contributions

The multi-layer perceptron class along with latter experiments is a combined effort of Gulan, Sinan Can and Wen, Zehai. Furthermore, Gulan, Sinan Can coded out the convolutional neural network using pytorch. Finally, the report is a combined effort of all members.

References

[1] Xing Lin et al. All-optical machine learning using diffractive deep neural networks. *Optical Computing*, Science(361):1004–1008, 2018.

[2] Tanushree Shenwai. A new google ai research study discovers anomalous data using self supervised learning. *MarkTechPost*, 2021.

[3] Muhammad Umar Karim; Kyung Chong-Min Tanveer, Muhammad Suhaib; Khan. Fine-tuning darts for image classification. 2020.

[4] Han Xiao. Fashion-mnist: Year in review. *Tech Blog - Neural Search AI Engineering*, 2022.

[5] Kashif; Vollgraf Roland Xiao, Han; Rasul. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *arXiv:1708.07747*, 2017.