MiniProject 3
Classification of Fashion-MNIST Image Data

Flemming Kondrup (260826047) Marwan Jabbour (260913912) Parsa Yadollahi (260869949)

Abstract

Artificial Neural Networks (ANNs) have revolutionized many fields of machine learning, including decision making on image data. In this work, we investigate the use of two ANN models, a Multi-Layer Perceptron (MLP) and a Convolutional Neural Network (CNN) for classification of Fashion-MNIST data. We investigate different architectures, activation functions, regularization, kernel size, padding, and more. We show that our optimal MLP model achieves an accuracy of 89.92% using 1 hidden layer of 256 nodes, a learning rate of 0.1, dropout regularization (keep factor of 0.75), and Leaky Relu as the activation function. In contrast, our CNN approach achieves an accuracy of 91.71% using 2 convolutions and 2 fully connected layers of 128 units. We conclude that deep learning achieves impressive performance in the context of image data and that from our experimentation, the CNN approach slightly outperforms the MLP approach.

1. Introduction

Vision is one of the main senses we use to make decisions on a daily basis. Consequently, it is logical that machine learning (ML), which aims to automate and potentially improve decision-making tasks, would have many applications regarding image data. In fact, ML can be applied to many fields, from medical imaging and pattern recognition to self-driving technology and security/ surveillance (1).

In this work, we implement two ANNs for classification on the Fashion-MNIST image dataset. We develop a Multi-Layer Perceptron (MLP) from scratch and implement a Convolutional Neural Network (CNN) from the PyTorch Library (2). MLPs are feedforward ANNs and consist of a fairly simple architecture with an input layer and an output layer, as well as hidden layers in between. In contrast, CNNs use the idea of convolution by applying filters to the input image, permitting them to capture the spatial dependencies in the data. This makes CNNs renowned for classification tasks involving image data.

We investigate different architectures, activation functions, regularization, and additional parameters. Our MLP model achieves an accuracy of 89.92% using 1 hidden layer of 256 nodes, a learning rate of 0.1, dropout regularization (keep factor of 0.75), and Leaky Relu as the activation function. By using the CNN approach, we can further improve accuracy to 91.71%, using 2 convolutions and 2 fully connected layers of 128 units. Our experiments thus confirm that ANNs offer impressive results in image data classification.

.

2. Datasets

The Fashion-MNIST dataset is composed of 28x28 pixel gray-scaled images, corresponding to 10 classes of fashion items, as shown in Fig. 1.

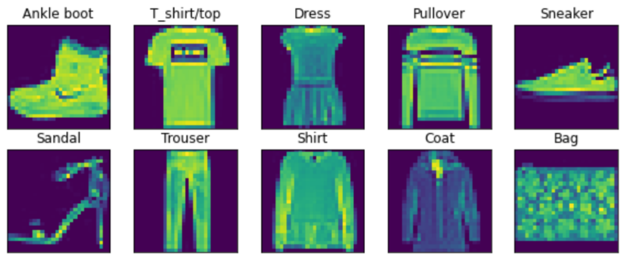

Figure 1. Examples of images from the 10 classes

The data were normalized to the [0,1] range and inputs were reshaped for efficient computation. The class distributions were computed. The training data had an equal distribution among classes, with 6000 images per class (see Fig. 2).

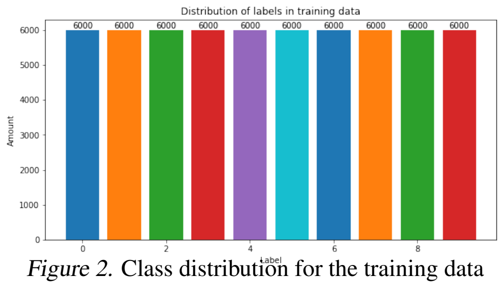

Figure 2. Class distribution for the training data

The testing data had an equal distribution among classes as well, with 1000 images per class (see Fig. 3).

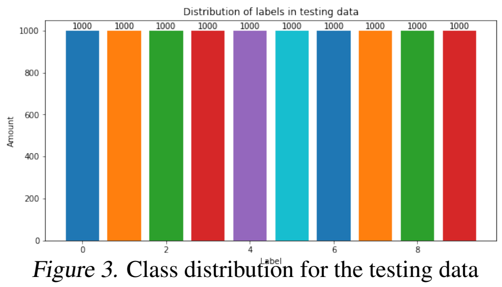

Figure 3. Class distribution for the testing data

3. Results

3.1. MLP - Impact of number of hidden layers

We first compare the accuracies of three models with respectively 0, 1, and 2 hidden layers (see Table 1).

Table 1. Testing accuracy in function of the number of hidden layers (with 128 nodes)

Number of hidden layers	Accuracy
0	84.11%
1	87.67%
2	87.99%

We observe an important jump in accuracy (3.56%) between our models with 0 and 1 hidden layers. When adding a second layer, we observe a small yet existing increase of 0.32% accuracy. There is therefore clear evidence that the depth of the MLP helps improve accuracy, which is expected as it permits better feature extraction and pattern recognition.

3.2. MLP - Impact of Activation Functions

We now compare the change in accuracy based on the activation function, specifically comparing ReLU with Tanh and Leaky-ReLU (see Table 2).

Table 2. Testing accuracy in function of activation functions (default model with 2 layers of 128 nodes)

Activation function	Accuracy
ReLU	87.99%
Tanh	88.11%
Leaky_Relu	88.76%

We observe that Tanh and Leaky_Relu both outperform Relu by 0.12% and 0.77% respectively. The relatively important increase in performance with Leaky_Relu is not surprising, considering it was developed as an improvement to address the challenge of dead neurons (dying Relu).

3.3. MLP - Impact of Dropout Regularization

One solution to potential overfitting in deep learning is the use of dropout regularization. This consists of dropping certain nodes during training randomly based on a specific probability. We here investigate the impact of dropout regularization on performance (see Table 3).

Table 3. Testing accuracy in function of the dropout keep factor, the probability of keeping any given node

Dropout keep factor	Accuracy
0.25	88.14%
0.50	88.66%
0.75	88.46%
1.00	88.41%

We observe that a dropout keep factor of 0.5 leads to optimal testing accuracy. This is in fact known to be a standard factor in the literature, as it is known to be a good balance between decreasing over-fitting and still offering sufficient opportunities for the model to learn accurate weights.

3.4. MLP - Impact of Normalization

As discussed in the *Datasets* section, our data was normalized to the [0,1] scale. We here investigate whether this actually improved our model's performance (see Fig. 4).

Table 4. Testing accuracy in function of the number of hidden layers (with 128 nodes)

Data Normalized	Accuracy
True	87.99%
False	10.00%

We observe that using the non-normalized data leads to an important drop in accuracy, from 87.99% to 10.00%. We trained the model with non-normalized data for 20 epochs up to 300 epochs and consistently found this result. We, therefore, conclude that normalization is essential for efficient learning on the Fashion-MNIST.

3.5. MLP - Bonus: Architecture Search

In section 3.1, we investigated the impact of different amounts of layers on performance. We here further this effort and consider not only changes in the number of layers but also the number of nodes within these layers (see Fig. 5).

Table 5. Testing accuracy in function of network architecture (comparison of models with zero, one and two hidden layers of 64, 128 and 256 nodes)

Architecture	Accuracy
no hidden layer	84.11%
1 hidden layer, 64 nodes	87.71%
2 hidden layers, 64 nodes each	87.87%
1 hidden layer, 128 nodes	87.67%
2 hidden layers, 128 nodes each	87.99%
1 hidden layer, 256 nodes	88.46%
2 hidden layers, 256 nodes each	87.98%

We observe that our optimal accuracy is achieved with 1 hidden layer of 256 nodes. It is interesting to note that for 64 and 128 nodes, adding a second layer permits additional accuracy, but this is not the case with 256 nodes. This suggests that our optimal model (256 nodes) achieves optimal complexity, which would explain why adding complexity on top of it (a second hidden layer) actually hurts performance.

3.6. MLP - Bonus: Impact of Batch Size

An important parameter is the batch size, which defines the number of samples that will be propagated in the network. We here investigate different batch sizes and their effect on performance (see Fig. 9).

Table 6. Testing accuracy in function of batch size

Batch Size	Accuracy
100	88.23%
200	88.39%
400	88.17%
600	87.21%

We observe that increasing batch size from 100 to 200 leads to a 0.16% improvement in accuracy, but any further increase leads to a decrease in performance. We thus conclude that the optimal batch size is 200 samples.

3.7. MLP - Bonus: Impact of Learning Rate

Learning rate is the hyperparameter that controls how much to change the model in response to the estimated error at each timestep. We here investigate different learning rates and their effect on performance (see Fig. 7).

Table 7. Testing accuracy in function of learning rate

Learning rate	Accuracy
0.001	77.67%
0.005	83.66%
0.01	85.51%
0.05	87.82%
0.1	87.51%
0.2	86.35%

We observe that increasing the learning rate up to 0.05 leads to an important improvement in accuracy (+ 10.15% compared to the learning rate of 0.001). Further increase in the learning rate leads to a decrease in accuracy, suggesting an optimal learning rate of 0.05.

3.8. MLP - Bonus: Sigmoid Activation Function

In section 3.2, we investigated the impact of different activation functions on performance. We here further this effort and consider an additional function, the sigmoid function (see Fig. 8).

We observe that the Leaky Relu function well outperforms the Sigmoid function (+ 3.98%). This is not surprising as the Sigmoid function is known to suffer from vanishing gradient challenge, leading to reduced performance.

Table 8. Testing accuracy in function of activation functions (we here compare the sigmoid function with our previously optimal function, Leaky Relu)

Activation function	Accuracy
Sigmoid	84.78%
Leaky_Relu	88.76%

3.9. MLP - Bonus: Leaky Relu Gamma Parameter Search

The Leaky Relu activation function follows the following transform formula: h(x) = max (0,x) + gamma * min(0,x). We set a range of gamma values from 0.001 to 1.0 and observed the model's accuracy change accordingly. Our model has one hidden layer with 128 nodes with a default run of 20 epochs. As can be seen in the table below, there is a slight decline in accuracy as the value of gamma increases. The highest accuracy, 0.8851, is obtained for gamma = 0.001.

Table 9. Gamma Values for Leaky Relu

Gamma	Accuracy
0.001	88.51%
0.01	86.7%
0.1	87.99%
0.25	86.43%
0.5	85.63%
1.0	83.61%

3.10. MLP - Bonus: Overall Large Parameter Search

Throughout our results section, we have considered various parameters individually. Nonetheless, various variables can affect each other, and it is thus essential to bring together all these aspects through a large grid parameter search. Here is a list of all the parameters that were considered:

Table 10. Values considered for each variable

hidden layers	[],[64],[64,64],[128], [128,128],[256],[256,256]
max epochs	100,200,300
activation functions	ReLU, Leaky-ReLU, Tanh, Sigmoid
batch size	100, 200, 400, 600
learning rate	0.1, 0.01, 0.05, 0.001
keep dropout ratio	0.25, 0.5, 0.75, 1

Our search revealed an optimal model achieving an accuracy of 89.92%. This model contains one hidden layer of 256 nodes, a batch size of 200, a learning rate of 0.1, and a keep ratio of 0.75.

3.11. CNN - Performance of the PyTorch model

Our CNN model was implemented using Python's PyTorch Library (2). It consists of 2 convolutional layers and 2 fully connected layers, with ReLU as activation function and Stochastic Gradient Descent to minimize the loss function. Additionally, we added a MaxPool activation function piped to the convolution result, reducing the size of our matrix by half. We chose a Cross-Entropy strategy since we are categorizing non-binary categorical variables.

After running the matrix through the first convolution and the first MaxPooling function, our output matrices have a 14x14 pixel ratio. We then repeat this process once more since it is a two layers CNN and are left with an initial image 16x smaller with dimensions of 7x7 pixels.

Compared to MLP, our CNN implementation took 18 epochs to reach its maximum accuracy of 91.71%. Figure 4 demonstrates the cost and accuracy trade-off of the model. We notice that the cost and accuracy have an inverse relationship such that when the cost decreases the accuracy increases.

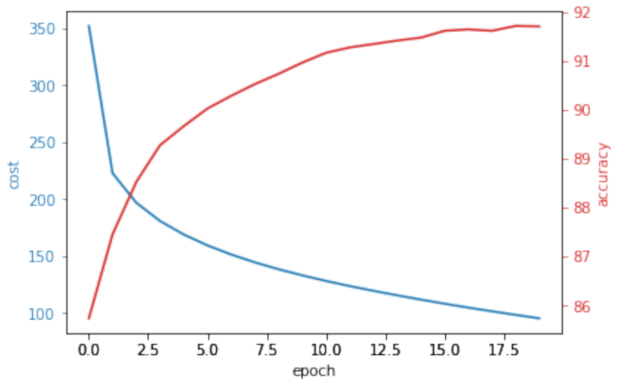

Figure 4. Epoch vs. Cost and Accuracy

3.12. CNN - Bonus: Impact of hyperparameters

Two main hyperparameters are the kernel and padding, which have a close relation. The kernel is a *n x n* matrix which acts as a filter to extract the features from the images. A common choice of the kernel is between 3x3 and 5x5 (3). The padding refers to the number of pixels added to an image when it is being processed by the kernel. After running our matrix through a convolution, our output matrix dimension is: $O = I + 2 * P - K + 1$,

where O and I are the dimension of the output and input matrix, P is the padding and K the kernel. To ensure that the input and output matrices share the same dimension, the padding and kernel must be bound by the following formula: $2 * P = K - 1$. Let us note that increasing the kernel leads to an increase in computation time since the number of parameters increases quadratically with the kernel (3).

When increasing the two hyperparameters for both convolutions, we observe a result that is just slightly higher than for the original 3x3 kernel size. This increases accuracy but also the computation time, by more than 25% for each kernel size over our initial 3x3 kernel (Figure 5). Because accuracy only slightly increases at a high computational cost, we recommend keeping the kernel size at 3x3. We also notice that the accuracy peaks when the kernel has dimension 7, then slowly declines. This may be due to the fact that at very high dimensional kernels, we start losing some information regarding smaller features.

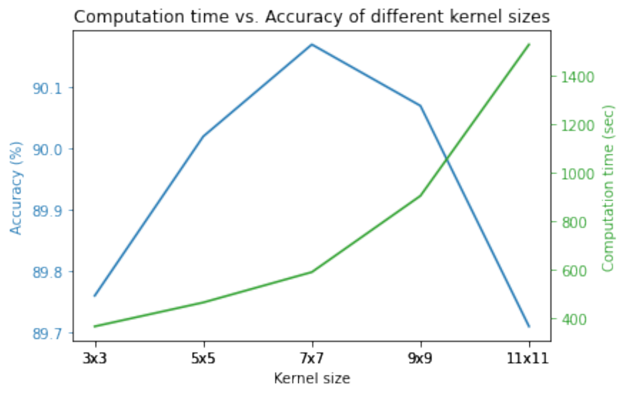

Figure 5. Computation time and accuracy vs. kernel dimension

3.13. CNN - Bonus: Performance of the Tensorflow model

In section 3.10, we implemented a CNN using PyTorch with 2 convolutions and 2 fully connected layers as previously mentioned. We pushed this experiment further by modifying the architecture of the CNN to be LetNet-5 with the help of the Tensorflow library.

3.13.1. The impact of convolutions

We ran experiments using different numbers of convolutions with the LetNet-5 CNN. We tested the accuracy using one, two, and three convolutions. We used the same hyperparameters as the ones used while experimenting with the CNN implemented using PyTorch. Experiments were run until loss functions and accuracies converged.

Table 11. Testing accuracies and time required in function of number of convolutions

	Accuracy	Time (ms)
1 convolution	86.92%	442.93
2 convolutions	87.48%	1582.53
3 convolutions	88.40%	2902.90

We notice that when using more convolutions, the accuracy increases slightly but computation time increases significantly. For example, adding a second convolution increased computation time by more than 3 times but only increase the testing accuracy by less than 1%. This could likely be because we've already extracted most features in the first convolution, so adding more would simply be redundant only resulting in overfitting the dataset.

3.13.2. THE IMPACT OF DROPOUT

Dropout Regularization is a form of adding regularization to a deep neural network in the effort of minimizing overfitting and improving generalization error. It works by probabilistically removing inputs to a layer. The effect of this is making a layer have the same behavior as a layer with a different number of nodes and connectivity to the prior layer.

Table 12. Testing accuracies in function of dropout factor, where the dropout value defines the probability of dropping a random node

Dropout value	Accuracy
0.0	87.97%
0.2	87.62%
0.4	86.33%
0.6	85.68%
0.8	83.16%

We notice that increasing the dropout rate decreases the accuracy of our CNN model. This could be caused by several reasons. For example, that our network was too simple so every path was essential. Further experiments could aim to test different architectures and observe the effect of dropout in these scenarios.

4. Discussion and Conclusion

In this work, we investigated the use of two approaches, an MLP and CNN, for classification of image data from the Fashion-MNIST dataset. We find that both models achieve very high accuracy, with peak accuracies being 89.92% and 91.71% respectively for the MLP and CNN. There is a multitude of reasons why CNNs return high accuracies and are generally preferred over MLPs. CNNs work well with data that have spacial relationships and are effective in reducing the number of parameters without losing on the quality of the models. This is done using a pooling layer, which downscales the image inputs. They also leverage the fact that nearby pixels are more strongly related than distant ones and analyze them using the kernel. Compared to CNN, MLP takes vectors as inputs, cannot downscale the input size, and cannot specify the sparsity of the connectivity between layers, meaning increased computation complexity and time and the weights are harder to train (4).

Various future directions could permit us to build on top of this study. Firstly, it would be interesting to work with different types of pre-existing CNN architectures such as LeNet or AlexNet from Python libraries like Keras. It would be interesting to observe the difference in performance between our current implementation of CNN and that implemented using other libraries. Secondly, future works could aim at implementing additional normalization methods, for example, batch normalization which adds a layer before the activation function and applies normalization to the inputs. This has the effect of stabilizing the learning process and dramatically reducing the number of training epochs required to train deep networks (5). Additionally, future works could aim to test our models on more complex datasets to validate their effectiveness.

5. Statement of Contributions

All members working contributed equally to the completion of the project and report.

References

[1] Nanonets - ml-based image processing. retrieved from https://nanonets.com/blog/machine-learning-image-processing/.

[2] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[3] About convolutional layer and convolution kernel. retrieved from https://www.sicara.ai/blog/2019-10-31-convolutional-layer-convolution-kernel.

[4] Cnn vs mlp for image classification. retrieved from https://medium.com/analytics-vidhya/cnn-convolutional-neural-network-8d0a292b449.

[5] A gentle introduction to batch normalization for deep neural networks. retrieved from https://machinelearningmastery.com/batch-normalization-for-training-of-deep-neural-networks/.