Fashion MNIST Predictions

Yu Miyauchi
University of California, Irvine
Irvine, CA 92617
ymiyauch@uci.edu

Jeffrey Ng
University of California, Irvine
Irvine, CA 92617
jeffrn4@uci.edu

Fu Hui
University of California, Irvine
Irvine, CA 92617
fhui2@uci.edu

Abstract

In this project, we applied Support Vector Machine (SVM) and Convolutional Neural Network (CNN) models toward the Fashion-MNIST (Mixed National Institute of Standards and Technology) dataset. We were able to realize a validation accuracy of 88.56 percent on SVM and 92.8 percent on CNN. We gained a deeper understanding of different classification models, and how they perform on dataset of images.

1 Data exploration

1.1 Example images and distribution of targets

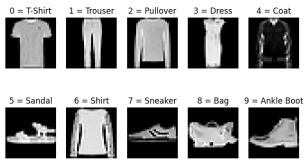

Figure 1: examples

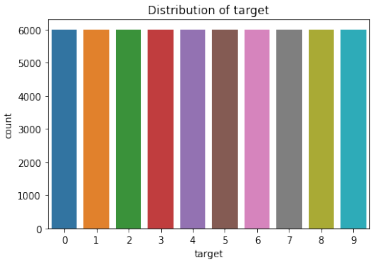

Figure 2: Distribution of targets

Figure 1 shows examples from each class. Figure 2 shows the distribution of data. Since the target is uniformly distributed, we will use K-Fold cross validation to evaluate our models. If the targets were skewed, we had thought about stratified or group k-fold to split targets evenly into training and validation data.

1.2 Data Visualization using UMAP

Visualizing how images could be classified may give us some intuitions to the distribution of dataset. Since visualization in 784 dimensions is improbable, we reduced them to two dimensions using UMAP. We used UMAP because it can cluster points more closer and performs faster than t-SNE or PCA.

Based on figure 3, the images are not linearly separable, as such, we investigate non linear models. Some classes can be easily separated: trouser, sandal, sneaker, bag, and ankle boot. Others are more difficult: T-shirt/top, pullover, dress, coat, and shirt.

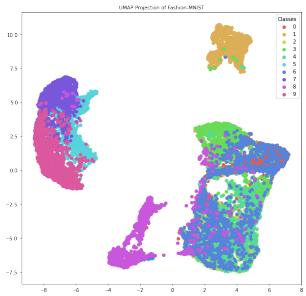

Figure 3: Data visualization using UMAP

2 Data Preprocessing

2.1 Normalization

To make our models converge faster and to make the distance between pixels smaller, we normalized our data.

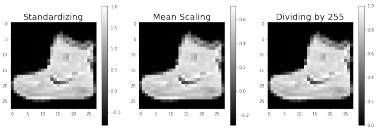

Figure 4: Data Normalization

Here are some statistic of original image before data processing. Mean: 72.94035 Max: 255 Min: 0 Standard Deviation: 90.02118. Standardization – Mean: 0, Min: -0.810, Max: 2.022, Standard Deviation: 1.0. Mean Scaling – Mean: 0, Min: -0.286, Max: 0.714, Standard Deviation: 0.353. Dividing by 255 – Mean: 0.286, Min: 0.0, Max: 1.0, Standard Deviation: 0.353.

2.2 Feature Extraction

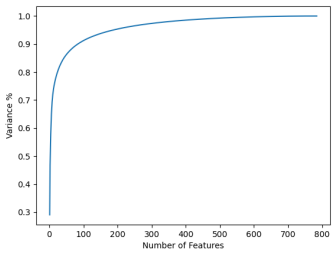

Figure 5: Variance

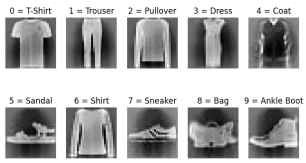

Figure 6: Reconstructed examples

We implemented PCA to capture the greatest amount of information while also reducing the dimension size. Figure 5 depicts the cumulative variance based on the S vector when computing SVD. We decided to pick 400 principal component to reconstruct our dataset with while retaining 98.53% variance.

2.3 Autoencoder

Another feature extraction method we tested was Convolutional Autoencoder. First we trained the model on the training set and using that we applied the first part of the model, being the encoder, to the test set. The result was a compressed version of the data with only 196 features.

3 Model Exploration

3.1 Support vector machine (SVM)

Our validation strategy is K-Fold validation because our targets are uniformly distributed as noted in data exploration part. Also, we used three splits because the accuracy of five splits and three splits didn't change and we needed to speed up the process given the large dataset.

For hyper parameter tuning, We used 30000 data points to search. Number of data could change combinations of best parameters; however, SVM is less likely to fall in this case given its limited parameter range unlike decision tree. We fixed gamma to be 'scale' because 'auto' consistently produced accuracy around 0.7.

From figure 7, Radial basis function (rbf) kernel and C=10, regularization parameter, performed best. This is possibly because rbf can make more complex boundary lines and C=10 keeps boundary not too complex.

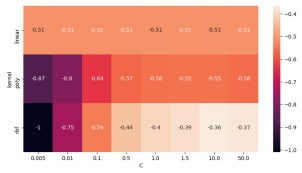

Figure 7: SVC Parameters performance

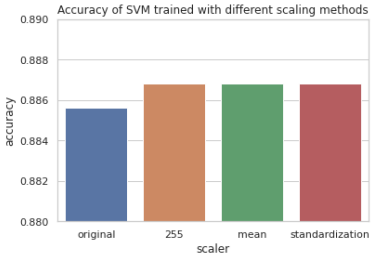

Figure 8: Accuracy w/ diff scaling

We further narrowed down the value of C to eight which produces best loss. SVM trained with three splits and with these parameters produced accuracy of 0.8864, 0.8873, and 0.8833 for each fold and mean accuracy of 0.8856.

About overfitting and underfitting, if we make C larger than 10, SVM will start to overfit because it is the penalty of the model misclassifying data points. Conversely, if we make C much smaller than 10, SVM will underfit to the data as it can not create more complex boundary lines.

About kernel functions, linear kernel underfitted to the data producing worse accuracy because it could not create complex decision boundaries like rbfs'. We suspected this might be the case as noted in the data exploration section.

We also trained SVM with 400 PCA components. This produced the mean validation accuracy of 0.9034 and a test accuracy of 0.9037. It slightly improved accuracy possibly from the regular SVM because PCA removed some noise.

For scaling methods, overall accuracy of SVM model slightly improved for three methods possibly because SVM could perform better with scaled distance between pixels. But, those scaling methods produced same accuracy possibly because they were not different enough for the boundary line to cross points.

We also trained SVM with feature extracted by convolutional autoencoder. This produced mean validation accuracy of 0.8654. This was probably because we did not a complex enough autoencoder or the fact that we constrained the dataset to hard.

3.2 Convolutional neural network (CNN)

We trained the CNN using PyTorch. Given the large data, we did not try all combinations of parameters but instead tuned the hyper-parameters in order of importance. We might fall into a local optimum, but this is better than not doing hyper-parameter tuning. Also, considering the randomness of the neural network, we ran these hyper parameter tuning several times to make sure the result is consistent.

We tuned in the following order: number of convolutional layers, node size in the convolutional layer, node size in a dense layer, and drop out rate. We fixed parameters that have not been tuned to be general values found by manually testing CNN. Those are a hundred butches per one run, two convolutional layers (32 nodes and 64 nodes consecutively), two dense layers (32*3*3-64 nodes and 64-10 nodes consecutively), and 0.2 drop rate.

From figure 6, three convolutional layers performed better possibly because CNN can build more complex functions. Since Fashion MNIST is more complex than MNIST data, this could be the case. For the node size of convolutional layers, 32-64-128 performed consistently well. For the number of dense layers, we did not visualize this, but two layers and one layer did not change scores, so we chose one layer. For the node size of the dense layer, 32 performed well. For the drop out rate, 0.2 and 0.3 worked best, with us ending up using 0.25. Lower drop out rate worked because the dataset does not have outliers and the model did not need to be generalized so much.

For scaling our images, we tried methods introduced in data processing. From figure 9, dividing images by 255 reduced log loss from 0.27 to 0.25 possibly because CNN wasn't disturbed by a large distance in the pixels' values. Standardizing did not work because the dataset did not have many outliers.

About overfitting and underfitting, we can see that validation loss didn't go up as training loss goes down. Also, the validation loss looks saturated; therefore, we could create CNN without overfitting nor underfitting.

Figure 12 and 13 shows final model learning curves and its structure. We achieved the accuracy of 0.929 for validation data and 0.923 for test data.

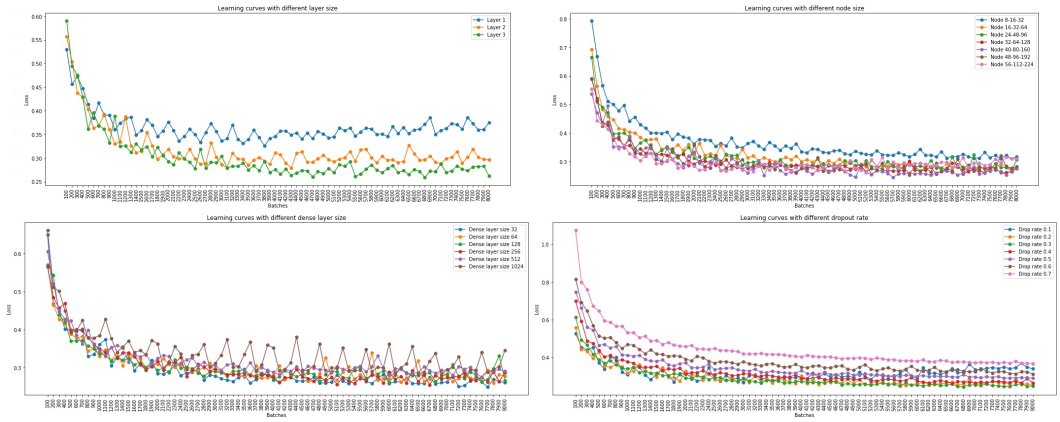

Figure 9: CNN Hyper parameter tuning

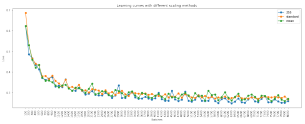

Figure 10: CNN trained with different scaling methods

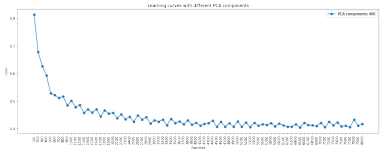

Figure 11: CNN with PCA

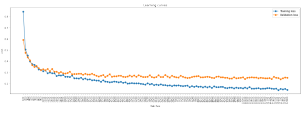

Figure 12: Learning curves of final model

Layer (type)	Output Shape	Param #
Conv2d-1	[-1, 32, 28, 28]	320
BatchNorm2d-2	[-1, 32, 28, 28]	64
MaxPool2d-3	[-1, 32, 14, 14]	0
Dropout2d-4	[-1, 32, 14, 14]	0
Conv2d-5	[-1, 64, 14, 14]	18,496
BatchNorm2d-6	[-1, 64, 14, 14]	128
MaxPool2d-7	[-1, 64, 7, 7]	0
Dropout2d-8	[-1, 64, 7, 7]	0
Conv2d-9	[-1, 128, 7, 7]	73,856
BatchNorm2d-10	[-1, 128, 7, 7]	256
MaxPool2d-11	[-1, 128, 3, 3]	0
Dropout2d-12	[-1, 128, 3, 3]	0
Linear-13	[-1, 32]	36,896
Linear-14	[-1, 10]	330

Figure 13: Final model structure