Statistical Pattern Recognition

ENEE633/CMSC828C Project - 2 Report

Implementing SVM Classifier and Deep Learning Architectures for Fashion-MNIST Dataset

---

Varsha Eranki (116204569)

Contents

List of Figures

1 Introduction

- We were asked to implement the SVM Classifier and Deep Learning Architectures for classifying the Fashion-MNIST dataset. It consists of 60,000 28 x 28 pixel grayscale images in its training set which are categorized into 10 classes, and the testing set consists of 10,000 images.
- Each training and test example is assigned to one of the following labels: 0=T-shirt/top, 1=Trouser, 2=Pullover, 3=Dress, 4=Coat, 5=Sandal, 6=Shirt, 7=Sneaker, 8=Bag, 9=Ankle boot
- I have implemented the flattened data for all the architectures except for the ResNet50 architecture.
- I have implemented Linear, Kernel(Polynomials defined in the SVM section) and RBF SVM by applying PCA and LDA for Dimensionality Reduction for all types.
- I have also implemented SVM classifier without Dimensionality Reduction for computing its accuracy and computative cost(time).
- For Deep Learning architectures I have implemented the traditional CNN using TensorFlow and Numpy libraries and ResNet50 Architecture using Keras in my second model.
- The results obtained, their computation time, trade-offs and architectures have been described below.

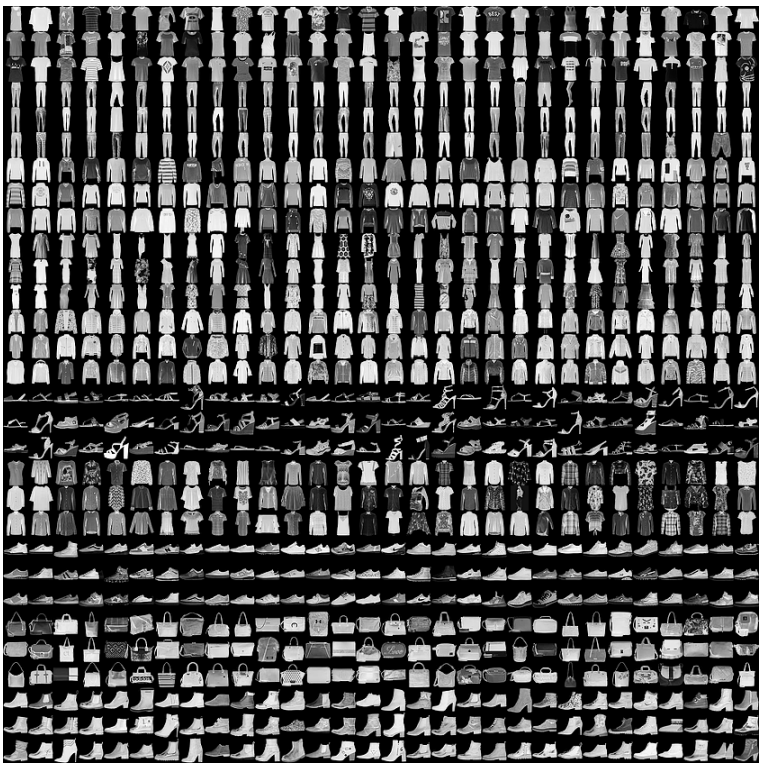

Figure 1: Visualization of the Fashion-MNIST Dataset

2 Overview of the Project

The project code is divided into six Python files out of which three are the program code files available as Jupyter Notebooks(.ipynb) and Python source file(.py), and three are utility files(.py format) for MNIST-reader, TensorFlow and Resnet to run it in batches, to read and load data.
The three program code files are:

1. SVM.ipynb (also as SVM.py) which consists of the SVM Classifier code for Linear, Kernel and RBF type.
2. DL_TF.ipynb (also as DL_TF.py) which consists of Convolutional Neural Network architecture using TensorFlow and Numpy
3. DL_ResNet.ipynb (also as DL_ResNet.py) consists of Resnet50 architecture using TensorFlow and Keras

The three utility code files are:

1. resnets_utils (imported in DL_ResNet)
2. tf_utils (imported in DL_ResNet and DL_TF)
3. mnist_reader (imported in all the three program code files)

The best accuracy was test accuracy was noticed in Deep Learning using TensorFlow in building Conventional CNN (96.95) and also SVM when PCA of 30 components is applied. The Kernel SVM of third degree polynomial gave the best accuracies (96-98). The longest computative time was noticed in SVM classifier when computed without PCA or LDA, which is obvious as we have to fit the classifier for a huge number of data points (60,000 X 784). But, if we increase the number of training epochs for ResNet50 Architecture, the time consumption increases at the rate of about an hour/epoch.

For easy computation and hoping for lesser time consumption, the grayscale images were normalized, i.e, the pixel values were computed on a scale from 0 to 1 instead of 0 to 255. This makes it easy during Forward Propagation and Backward propagation calculations for Neural Networks as they do not have to train under large numbers consuming way more time. The same goes for fitting the line and curves for SVM Classifier.

Built-in functions have been implemented for adopting SVM classifier by importing sklearn library and I have implemented the Deep learning architectures using Numpy, TensorFlow and Keras as I have learnt them from an online course in Summer'19.

3 Support Vector Machine Classifier

The Support vector machine classifier is a widely used Clustering algorithm for Unlabeled data after training on the training data.

In our problem case, we implement SVM classifer to classify the data linearly, using the kernel trick(polynomial) and also using RBF, i.e, linearly seperable and non-linearly seperable data points. It is best known for computing low errors as it computes the largest distance between the training data points of different classes and fits a hyperplane accordingly. SVM can be inferred as extension of Perceptron models. It is observed that the effficiency of SVM model depends on the selection of the kernel, this can be observed in the experimental results computed below.

3.1 SVM Linear Classifer

The Linear SVm Classifier creates the maximum margin hyperplane in order to classify the data points linearly. i have implemented Linear SVM using PCA, LDA and also without both. Linear SVM was also verified and compared to Kernel SVM of polynomial degree one, where the results were very similar and insignificant in their difference. There are total seven Linear SVMs that were plotted and it took the least computative time among the three.

3.2 SVM Kernel Classifier

The SVM Kernel was computed for polynomial degrees 1, 2, 3, 5 and 8, out of which the SVM that fit the third degree polynomial scored the maximum accuracy. this was computed for PCA of 15, 30, 50, 75 and 100 components, LDA and without the Both (which therefore consumed about 11 hours of computation time)

3.3 SVM Radial Bias Function(RBF) Classfier

The Radial Bias function SVM was computed on the same parameters as the Linear SVM and it had an overall score of scoring great accuracies aw well. It implements the kernel trick to compute a similarity measure.

4 Experimental Results of SVM

- By applying LDA to Test and Train Data, Test Accuracies obtained are: (section-5 of SVM.ipynb)

  1. Linear SVM = 89.33
  2. Kernel SVM:

  a. Polynomial of degree 1 = 89.29

  b. Polynomial of degree 2 = 88.24

  c. Polynomial of degree 3 = 91.74

  d. Polynomial of degree 5 = 90.61

  e. Polynomial of degree 8 = 86.78

  3. RBF SVM = 92.55

- By applying PCA to Test and Train Data, Test Accuracies obtained are: (section-6 of SVM.ipynb)
  For PCA having number of components = 15 (Section 6.1 of SVM.ipynb)

  1. Linear SVM = 88.45

2. Kernel SVM:

a. Polynomial of degree 1 = 88.38

b. Polynomial of degree 2 = 94.39

c. Polynomial of degree 3 = 96.68

d. Polynomial of degree 5 = 95.99

e. Polynomial of degree 8 = 92.06

3. RBF SVM = 97.07

For PCA having number of components = 30 (Section 6.2 of SVM.ipynb)

1. Linear SVM = 92.49

2. Kernel SVM:

a. Polynomial of degree 1 = 92.41

b. Polynomial of degree 2 = 97.78

c. Polynomial of degree 3 = 98.09

d. Polynomial of degree 5 = 97.71

e. Polynomial of degree 8 = 93.02

3. RBF SVM = 98.21

For PCA having number of components = 50 (Section 6.3 of SVM.ipynb)

1. Linear SVM = 93.76

2. Kernel SVM:

a. Polynomial of degree 1 = 93.70

b. Polynomial of degree 2 = 98.06

c. Polynomial of degree 3 = 98.14

d. Polynomial of degree 5 = 96.77

e. Polynomial of degree 8 = 85.05

3. RBF SVM = 98.26

For PCA having number of components = 75 (Section 6.4 of SVM.ipynb)

1. Linear SVM = 94.28

2. Kernel SVM:

a. Polynomial of degree 1 = 94.06

b. Polynomial of degree 2 = 97.91

c. Polynomial of degree 3 = 97.67

d. Polynomial of degree 5 = 92.72

e. Polynomial of degree 8 = 50.58

3. RBF SVM = 98.00

For PCA having number of components = 100 (Section 6.5 of SVM.ipynb)

1. Linear SVM = 94.38

2. Kernel SVM:

a. Polynomial of degree 1 = 94.25

b. Polynomial of degree 2 = 97.74

c. Polynomial of degree 3 = 97.13

d. Polynomial of degree 5 = 82.57

e. Polynomial of degree 8 = 19.08

3. RBF SVM = 97.74

- SVM without PCA or LDA, Test Accuracies obtained are: (section-7 of SVM.ipynb)

1. Linear SVM = 94.04

2. Kernel SVM:

a. Polynomial of degree 1 = 93.19

b. Polynomial of degree 2 = 91.4

c. Polynomial of degree 3 = 58.67

d. Polynomial of degree 5 = 11.35

e. Polynomial of degree 8 = 11.35

3. RBF SVM = 94.46

5 Deep Learning Architectures

5.1 Convolutional Network using TensorFlow and Numpy

The CNN which was computed using TensorFlow and Numpy has the following architecture: LINEAR - RELU - LINEAR - RELU - LINEAR - SOFTMAX Its pretty obvious to have Relu non linear activations followed by Softmax in the end as it is a Multi-Class Classification problem. This architecture gave the most accuracy in the project and also took less than 10 minutes to train and test the network. For both the neural networks Cross Entropy loss function was implemented.

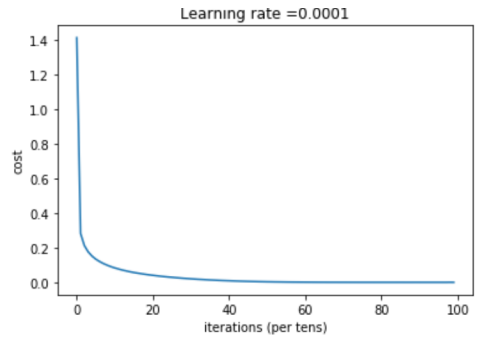

Figure 2: Graph plot for Number of Iterations vs Cost (loss function)

5.2 ResNet Architecture

ResNets stands for Residual Neural networks. It came into picture as an update to very Deep Neural Networks in order to overcome the problem of Vanishing and exploding gradients. It consists of skip connections which allows activation from one layer to be suddenly fed it to another layer. Its activations can be written as $a^{[l+2]} = g^{[l+2]}(z^{[l+2]} + a^{[l]})$ Where l is the number of the layer in the Neural Network.

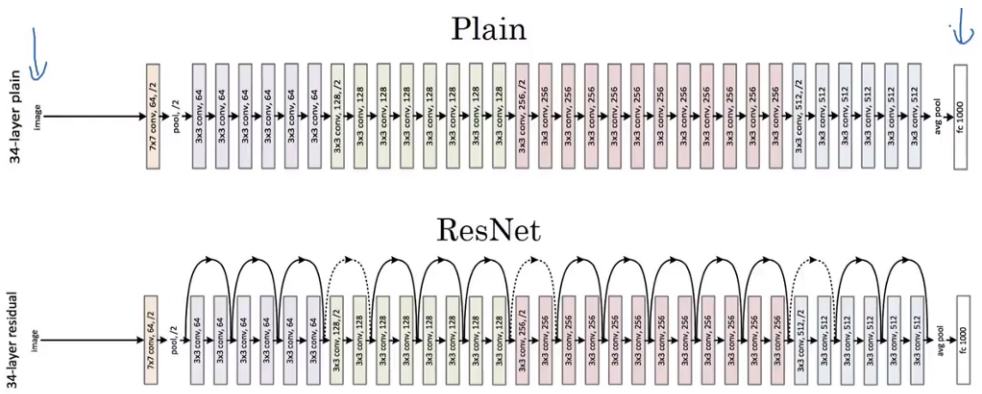

Figure 3: Block Diagram of ResNet and Skip Connections

Since it may hurt to do on a training set for a Deep Neural Network sometimes, a ResNet is the key. And what you see in ResNets is a lot of same dimensional Convolutional operations. The ResNet that was implemented has the following architecture in order:
CONV2D - BATCHNORM - RELU - MAXPOOL - CONVBLOCK - IDBLOCK*2 - CONVBLOCK - IDBLOCK*3- CONVBLOCK - IDBLOCK*5 - CONVBLOCK - IDBLOCK*2 - AVGPOOL - TOPLAYER

An Identity block is used for skip connections between alternative layer activation to have the same dimension. Zero Padding and max Pooling have been implemented for the output to reduce the size of representation and to speed up computation as well as the features we detect to make it more robust. Max Pooling is an operation where the features detected anywhere in the Input Matrix are preserved in Max Pooling due to their higher values of pixel values under the assumption that high numbers is a result of feature detection task. To speed up the training I have implemented Batch Normalization of size as 500 for 6 epochs, this approximately took 1.25 hours/epoch.

6 Experimental Results of Deep Learning Architectures

- **Convolutional Network using TensorFlow and Numpy (DL_TF.ipynb)** was trained on 500 epochs for a batch size of 100. Cost for the epochs:

  Cost after epoch 0: 1.414395

  Cost after epoch 50: 0.083237

  Cost after epoch 100: 0.040651

  Cost after epoch 150: 0.020357

Cost after epoch 200: 0.009163

Cost after epoch 250: 0.003324

Cost after epoch 300: 0.000993

Cost after epoch 350: 0.000269

Cost after epoch 400: 0.000076

Cost after epoch 450: 0.000025

And the Train Accuracy: 100.0 giving a Test Accuracy: 96.95

- **ResNet (DL_ResNet)**

Epoch 1/6 60000/60000 [==============================] - 4419s 74ms/step - loss: 0.8579 - acc: 0.7092

Epoch 2/6 60000/60000 [==============================] - 4340s 72ms/step - loss: 0.4776 - acc: 0.8283

Epoch 3/6 60000/60000 [==============================] - 4500s 75ms/step - loss: 0.3754 - acc: 0.8637

Epoch 4/6 60000/60000 [==============================] - 4571s 76ms/step - loss: 0.3249 - acc: 0.8818

Epoch 5/6 60000/60000 [==============================] - 4188s 70ms/step - loss: 0.2858 - acc: 0.8942

Epoch 6/6 60000/60000 [==============================] - 4094s 68ms/step - loss: 0.2567 - acc: 0.9045

10000/10000 [==============================] - 38s 4ms/step

Loss = 0.5509574840545655 Test Accuracy = 0.8307