Fashion-MNIST Task

April 24, 2020

1 Data Understanding

The fashion MNIST dataset contains:

- 60000 28x28 grayscale images in the training set
- 10000 28x28 grayscale images in the test set
- belonging to 10 classes (Tshirt, Trouser, Pullover, Dress, Coat, Sandal, Shirt, Sneaker, Bag, Ankle boot)

Before proceeding with any model/training, it is very important to understand the data, class distributions and visualise it. Figure 1 shows the class distribution of the training dataset, number of samples for each of the 10 classes. The average number of samples for each class is 6003 with standard deviation of 53 samples. This shows that the classes are balanced and hence won't need to deal with class imbalance problem. On visualising the data, classes like shirt, pullover and t-shirt look similar. Later we can pay attention to the confusion matrix to see if the network is able to distinguish well between these classes.

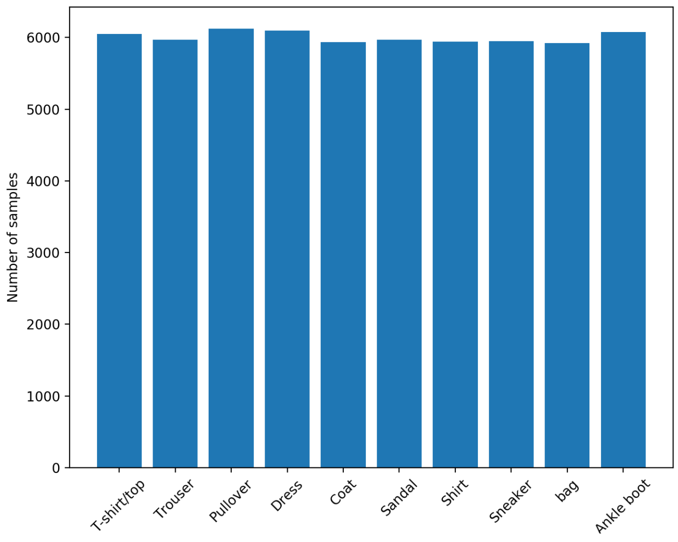

Figure 1: Class distribution of training data

2 Models and pipeline

For the pipeline and experiments, pytorch 1.4.0 was using as the deep learning library. Following steps were performed for the basic step:

- The training data was split into training and validation for experiments and model selection
- Testing data was used in the end after model selection to see how well the models generalise
- Data loaders and transforms such as convert to tensor, data augmentation during fly were set up to be fed into network

2.1 Model architectures

The fashion-MNIST task is a relatively simpler task compared to other publically available datasets like CIFAR-10. Also, the images in fashion-MNIST dataset are small and only in single channel (gray scale). Starting with a deep neural network might not make sense due to the small dimensionality of input (28x28x1). Therefore, simpler CNN based networks were used as a starting point and model complexity was changed accordingly.

To understand the complexity of task, a simple off the shelf LeNet like CNN, shown in table 1, was used as a starting point. The model performed fairly well with an accuracy of 0.913. In order to improve the performance, the model complexity was increased and VGG like CNN was used as shown in table 2. Batch normalisation was used after every convolution layer because it helps in a more stable training in general. Also, experiments with and without dropout were performed to study its impact on model generalisation.

In literature, *transfer learning* has proved to be very useful where models trained on imagenet are used and the last layers are tuned specific to the task. To study how well this could work for our problem, pre-trained *resnet18* was used and the last layer was changed to give 10 outputs, corresponding to the number of classes in the dataset. Each iteration of fine tuning was time consuming, mostly because the images were transformed to a larger size and parsing data into a deeper network is time consuming. In order to be able to use the benefits of resnet blocks and skip connections, a simpler architecture based on resnet blocks with 10 layers was used.

For describing convolutional layers, we use Conv2D(`num_filters, window_size`) as the convention. For fully connected layers we use Dense(`num_units`).

Table 1: Network Configuration for Simple LeNet like CNN

	Hyperparameters
conv_relu_maxpool_1	Conv2D(16, 5x5) + relu activation + MaxPool(2x2, 2)
conv_relu_maxpool_2	Conv2D(32, 5x5) + relu activation + MaxPool(2x2, 2)
conv_relu_maxpool_3	Conv2D(64, 5x5) + relu activation + MaxPool(2x2, 2)
dropout_fc_relu_1	0.5 dropout + Dense(128) + relu activation
dropout_fc_relu_2	0.5 dropout + Dense(256) + relu activation
fc_1	Dense(10)

Table 2: Network Configuration for VGG like CNN

	Hyperparameters
conv_batchnorm_relu_1	Conv2D(64, 3x3) + batch normalisation + relu activation
conv_batchnorm_relu_2	Conv2D(64, 3x3) + batch normalisation + relu activation
maxpool_dropout_1	MaxPool(2x2, 2)+ 0.4 dropout
conv_batchnorm_relu_3	Conv2D(64*2, 3x3) + batch normalisation + relu activation
conv_batchnorm_relu_4	Conv2D(64*2, 3x3) + batch normalisation + relu activation
maxpool_dropout_2	MaxPool(2x2, 2)+ 0.4 dropout
conv_batchnorm_relu_5	Conv2D(64*4, 3x3) + batch normalisation + relu activation
conv_batchnorm_relu_6	Conv2D(64*4, 3x3) + batch normalisation + relu activation
average pool_1	MaxPool(7x7, 7)
fc_1	Dense(10)

2.2 Data transforms and training specifications

For training data, transformations such as normalisation and data augmentations was performed. The idea is to apply transformations to the input images that must keep the label invariant: for example, slightly rotating, shift and horizontal flip. For the test/validation images, normalisation and conversion to tensor was performed as shown in Listing 1.

Listing 1: Data transformations applied

```python

from torchvision import transforms

train_augment_transforms = transforms.Compose([
    transforms.RandomHorizontalFlip(0.5), # data aug, horizontal flip
    transforms.RandomAffine(degrees=10, translate=(0.1, 0.1)), # data aug ↩
        slight rotation and shift
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,),)
])

val_transforms = transforms.Compose([transforms.ToTensor(), ↩
                    transforms.Normalize((0.5,), (0.5,),)])
```

For transfer learning, to make our dataset compatible with the input of the resent18 architecture, the transformations shown in Listing 2 were performed.

Listing 2: Data transformations for transfer learning

```python
from torchvision import transforms
```

```

data_transforms = {
    'train_transfer_learning': transforms.Compose([
        transforms.RandomResizedCrop(224),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Lambda(lambda x: x.repeat(3, 1, 1)),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
    'val_transfer_learning': transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Lambda(lambda x: x.repeat(3, 1, 1)),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ])}
```

The metric chosen for the model was *cross entropy loss* which performed well for the experiments. All classes had almost same number of samples hence didn't have to deal with class imbalance problem.

3 Experiments

All the experiments were run on NVIDIA TITAN V GPU machine. The optimiser used for the experiments was ADAM with learning rate of 0.002, except for resnet18 pre-trained network where the learning rate used was 0.001. Table 3 contains comparison of all the models used, comparison based on number of model parameters, time used for training each epoch/iteration, validation loss and accuracy. From the experiments, we can see that the VGG like CNN and Resnet18 pretrained models have comparable performance on the validation dataset. However, the number of parameters in Resnet18 model has 10 times more parameters and is 7 times slower than the VGG like CNN. Therefore the clear winner is VGG like CNN model.

The code for the project can be found here: https://github.com/randomgen1/fashion-MNIST-classification.git

Table 3: Comparing models

Model	**number of parameters**	**time each epoch(s)**	**Val loss**	**Val accuracy**
LeNet like CNN	15	173k	0.252	0.913
VGG like CNN	22	1,148k	0.165	0.948
Resnet18 pretrained	150	11,181k	0.18	0.947
Resnet10	25	1,229k	0.22	0.924

A comparison of the models based on validation loss and accuracy over epochs can be found below:

1. Comparison between LeNet like CNN and VGG like CNN: As can be seen in figure 2, adding complexity to the model and going from LeNet like CNN to VGG like CNN makes sense as it leads to improvement in performace of model based on validation loss and accuracy.

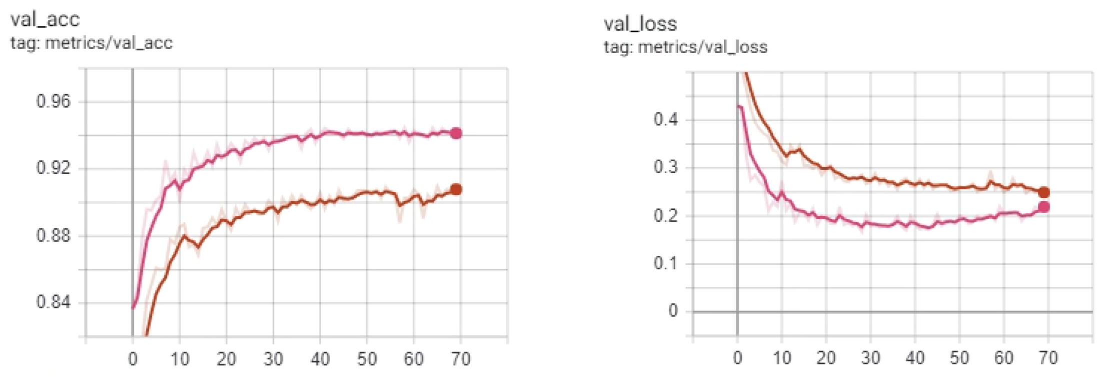

Figure 2: Validation accuracy and loss curves, VGG like CNN: pink, LeNet like CNN: red

2. With and without dropout in VGG like CNN: As can be seen in figure 3, the performance of the two configurations is comparable. However the blue curve with dropout leads to better generalisation in validation loss after epoch 40. Therefore it makes sense to actually use models with dropout to avoid overfitting.

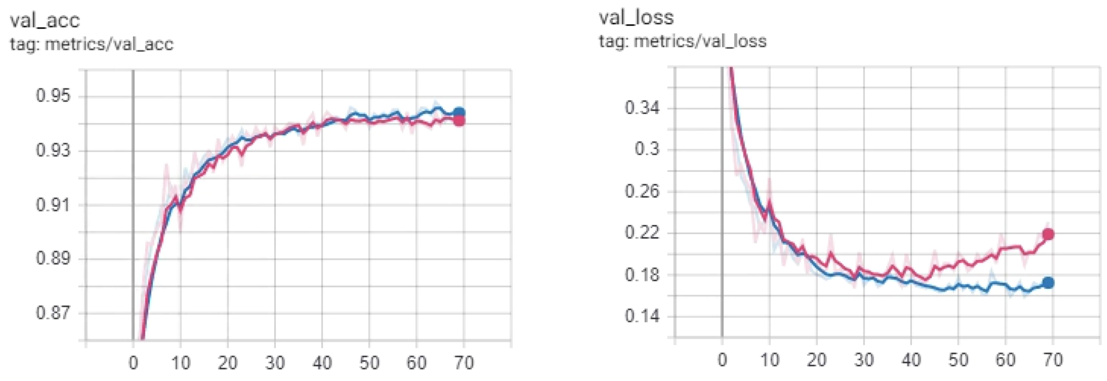

Figure 3: Validation accuracy and loss curves for VGG like CNN, with dropout: blue, without dropout: pink

3. Comparing the two best performing models: As seen in figure 4, the validation accuracy and loss of the VGG like model and resnet18 pre-trained is comparable. We don't observe any added advantage of using a pre-trained model. Infact, fine tuning a pre-trained model takes more time due to more model parameters.

4. Comparison of VGG like CNN, LeNet like CNN and resnet10 model: As can be seen in the figure 5, resnet10 model (inspired from resnet blocks) performs better than the LeNet like CNN but worse than the VGG like CNN. It would be interesting to run more experiments like adding dropout, hyper-parameter tuning for better understanding of the observation.

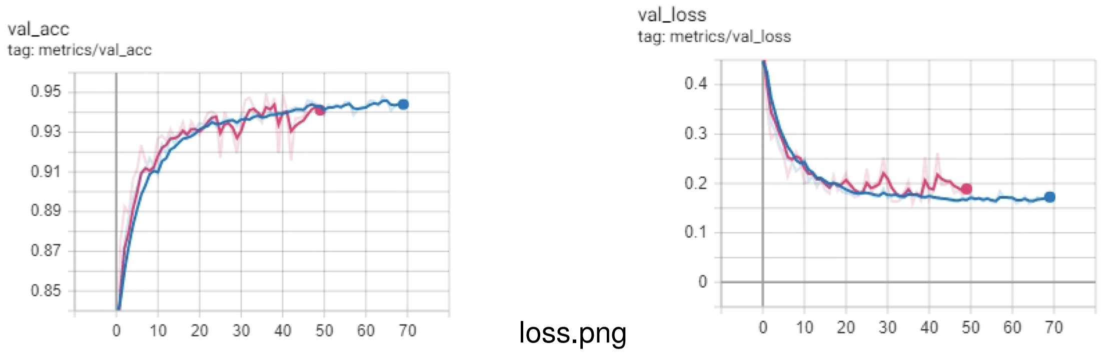

Figure 4: Validation accuracy and loss curves, VGG like CNN: blue, resnet18 pre-trained CNN: pink

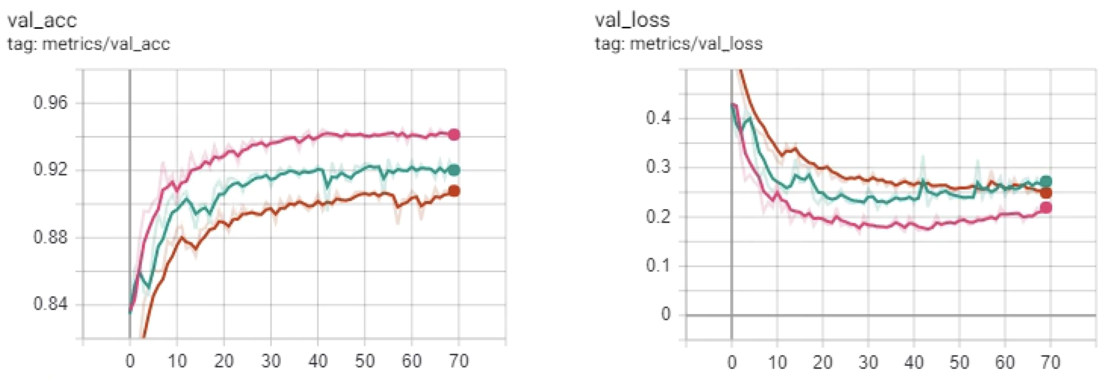

Figure 5: Validation accuracy and loss curves, VGG like CNN: pink, resnet10: green, LeNet like CNN: red

5. Model prediction and testing: On the best model from the VGG like CNN, testing was performed on the unseen 10k images. The test accuracy was found to be 0.9402, loss: 0.192, precision: 0.9402, recall: 0.9402, F1: 0.939. Figure 6 shows the confusion matrix on the test dataset. We can see that some samples belonging to *shirt* get confused with *t-shirt/top* and *coat* class. Similarly some samples from *coat* class get confused with *pullover* class. This is an interesting observation as we observe the same confusion when we visualise the images.

4 Conclusions and future work

From the experiments, it can be concluded that the Fashion-MNIST is a relatively simpler task as a simpler version of VGG inspired network gives a good accuracy of ~95%. This shows that we don't need deeper models like ResNet50 or inception-net for the task. However, as a future work, it would make sense to try architectures containing a few inception blocks or a shallower version of densenet like model.

From the confusion matrix in figure 6, it can be concluded that some classes get confused like shirt, T-shirt, pullover. This can be overcome by using siamese network like setup with triplet loss

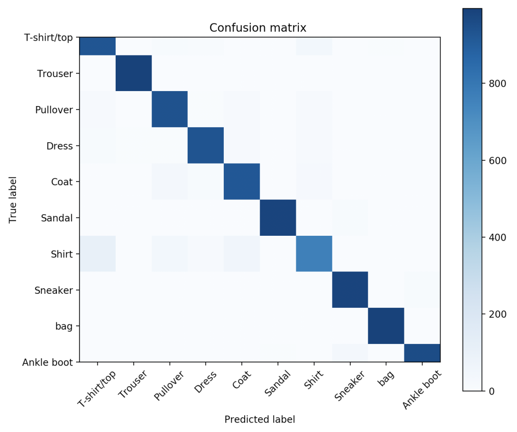

Figure 6: Confusion matrix on unseen test data

specific to the classes getting confused so as to increase the distance between these classes in the embedding space (eg. second last layer feature space in a CNN model). Triplet loss along with sampling techniques like distance weighted sampling (distance in the embedding space) might improve performance specific to the classes getting confused.

5 Bonus task

For the object detection task, YOLO (you only look once) can be very useful where the video frame is split into grids and each grid has some bounding boxes on which we can detect objects from our 10 fashion-MNIST classes. Generally, YOLO comes with pre-trained networks like Darknet, pre-trained for the classes in COCO dataset. We can replace this network with our model trained on fashion-MNIST dataset and use YOLO for object detection in frames.