Name: Meetakshi Setiya
Roll no.: 2019253

ML Assignment 2

All plots are present in the ./plots directory, saved models/weights are in the ./checkpoints directory. I have not included the trained VGG model object in the zip file since it was 500MB in size.

1.
To preprocess the fashion MNIST dataset, I normalised the image pixels to have them take values between 0 and 1 only.

1. Done in Python Notebook Q1.ipynb

2. Done in Python Notebook Q1.ipynb

3. Done in Python Notebook Q1.ipynb

4. Done in Python Notebook Q1.ipynb

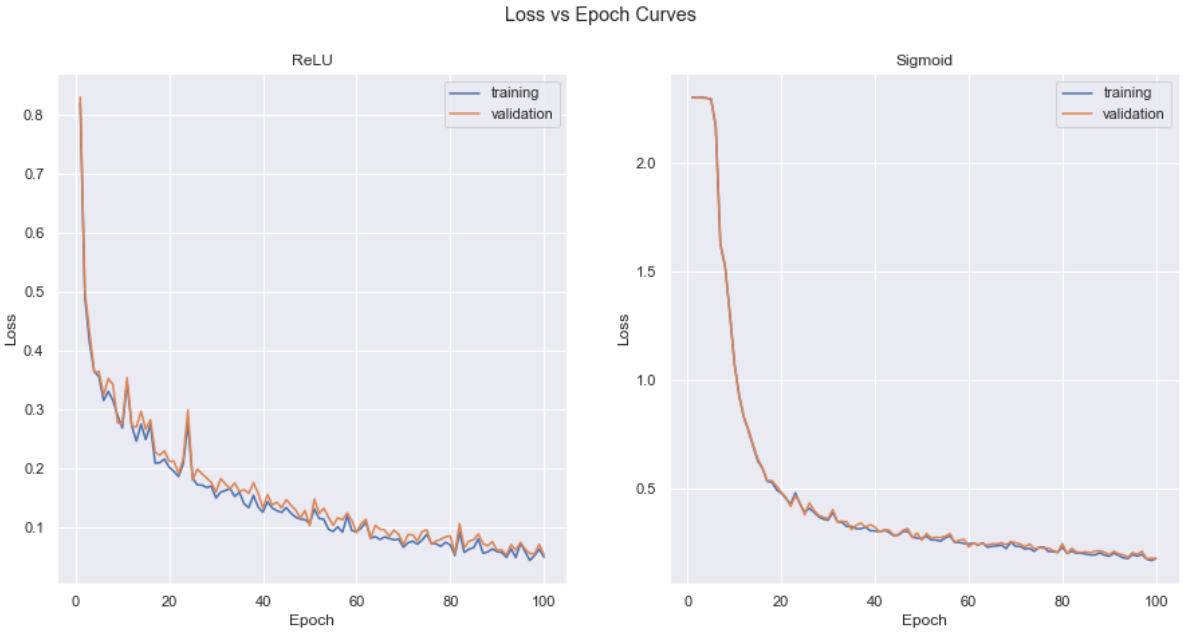
5. Since we were to take batch size=32, I used batch gradient descent instead of stochastic gradient descent.
   Also, since I split the data into 90% training+validation and 10% testing beforehand, to maintain the 80:10:10 ratio mentioned in the question, the validation dataset size was taken to be 0.11 times the training+validation data side.

   Note: While training the neural network for Linear activation function, a high learning rate of 0.1 lead to exploding gradients at around the 27th epoch. The Neural Network was training as expected before that.
   After the loss reached a very high value (NaN), the accuracy on training, testing and validation sets dropped drastically to about 9%-10%. The same happened with sklearn's MLP with 'identity' as the learning rate too. However, when I tried training with a learning rate of 0.0001, I did not face this problem.
   Since we were told to use 0.1 as the learning rate in the question, that is what I have used for now for training my neural network. We were allowed to reduce the learning rate for question 1.6, but for fair comparison I took the same learning rate i.e. 0.1 across for all training instances. *Sklearn's MLP gave NaN first iteration onwards for 0.1 learning rate.*
   Please note that even in Question 1 part 6, the curves are going to be affected by the NaN values training and validation losses took.

   The weights and biases for each model were saved in the directory checkpoints/<name-of-activation-function> with the given name convention.

(b).

Training and Validation Loss vs. epoch curves for ReLU and Sigmoid activations.



(c).

Accuracies reported on the test set for each activation function after 50 and 100 epochs.

Accuracies on test set for my implementation:

	50_epochs	100_epochs
sigmoid	88.4667	88.8
tanh	89.2833	88.45
relu	89.7	89.45
linear	9.95	9.95

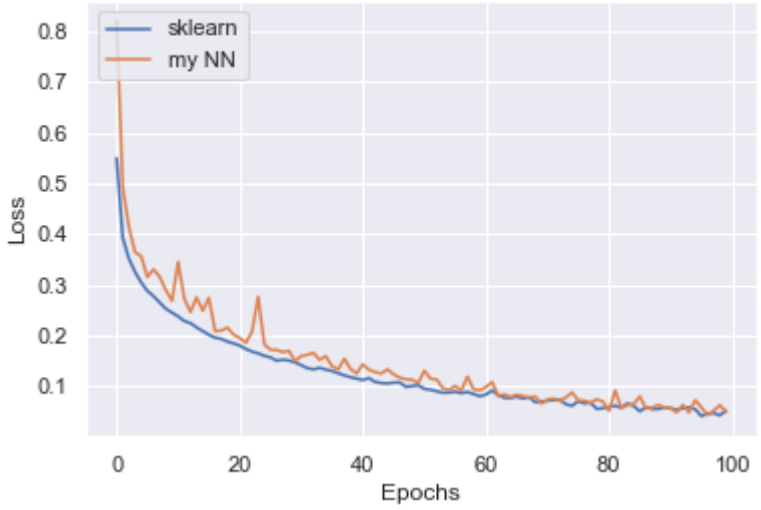
The accuracy for linear activation function is so low because of the messed up weights caused by exploding gradients.

6. I have used 0.1 as the learning rate for each activation function in sklearn's MLP for fair comparison. I obtained NaNs for loss after the 26th iteration in my implementation and first iteration onwards in sklearn's implementation for linear activation.
   I have plotted another curve with sklearn's MLP linear activation learning rate 0.001 compared with my implementation of the same with learning rate 0.1.

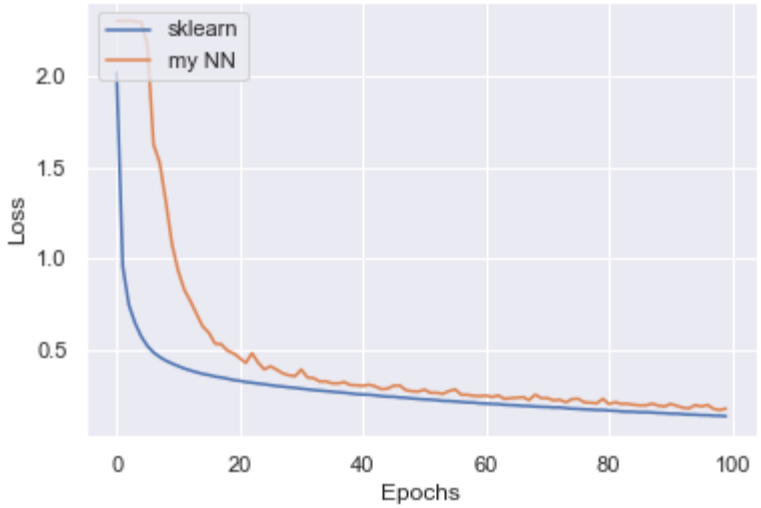
The models were saved in the directory checkpoints/sklearn with the name convention 50/100_<name of activation function>_model.pkl.

Graphs comparing the (training) loss from my neural network and sklearn's implementation for each activation function are added below:

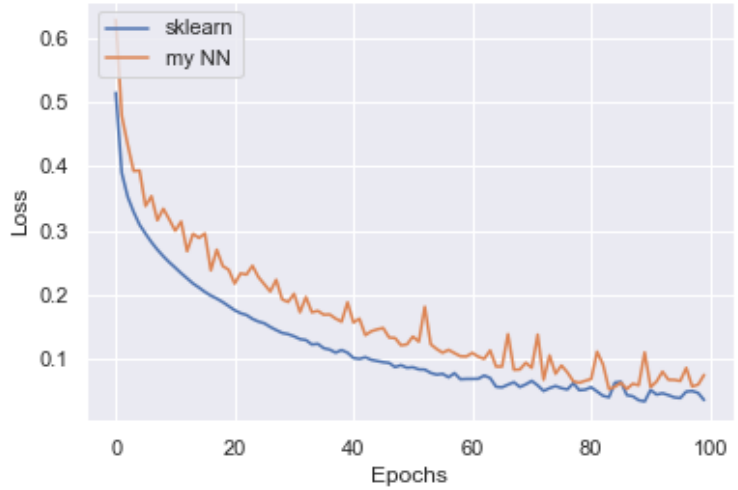
- ReLU



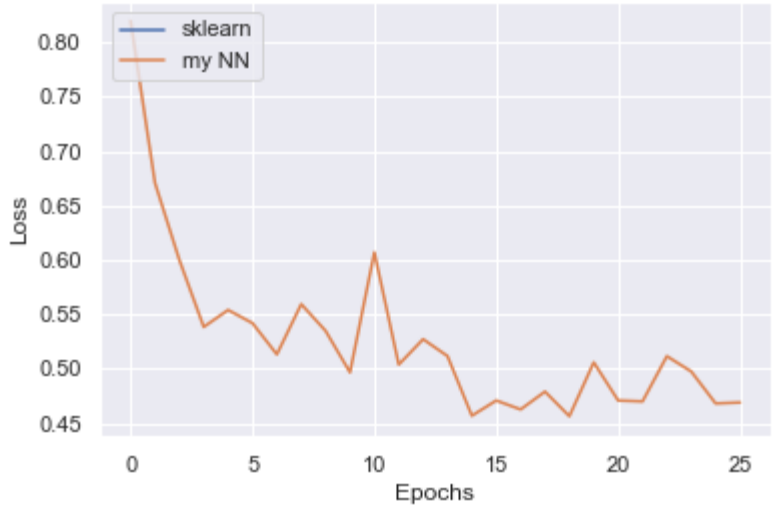
- Sigmoid



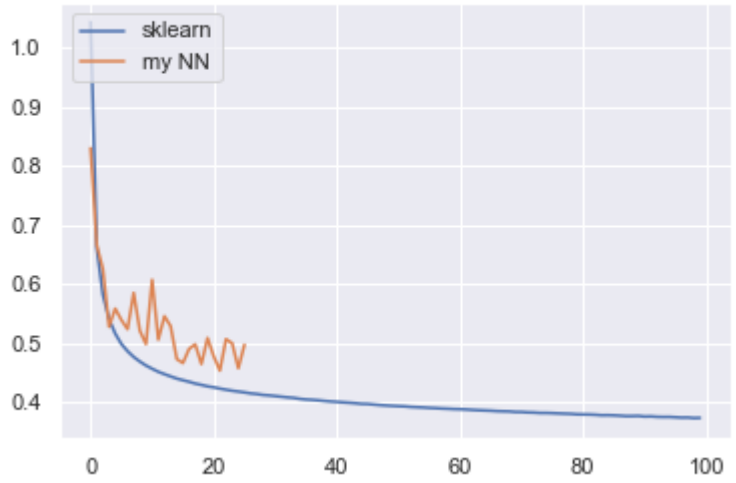
- Tanh



- Linear



Note: sklearn's MLP gave NaN's since the first iteration. Here is another plot of the NN with linear activation function with the learning rate of sklearn's MLP taken as 0.001 compared with my implementation with 0.1 as learning rate.



Accuracy of all the loaded models on the test set:

```
Accuracies on test set for sklearn and my implementation:
+---------+---------------------+-----------------+----------------------+--------------------+
|         |   sklearn_50_epochs |   mine_50_epochs |   sklearn_100_epochs |   mine_100_epochs |
+---------+---------------------+-----------------+----------------------+--------------------+
| sigmoid |             88.3167 |          88.4667 |              87.9833 |              88.8  |
| tanh    |             88.05   |          89.2833 |              87.5667 |              88.45 |
| relu    |             88.7333 |          89.7    |              88.7167 |              89.45 |
| linear  |              9.95   |           9.95   |               9.95   |               9.95 |
+---------+---------------------+-----------------+----------------------+--------------------+
```

Best model overall in terms of testing accuracy is:
My implementation of NN with ReLU as the loss function after 50 epochs. The accuracy obtained is 89.7% with learning rate 0.1 and parameters mentioned in the assignment.

2.

The model parameters used were:

```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace=True)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace=True)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
```

```
    (8): ReLU(inplace=True)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace=True)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace=True)
    (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): ReLU(inplace=True)
    (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU(inplace=True)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace=True)
    (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (25): ReLU(inplace=True)
    (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (27): ReLU(inplace=True)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU(inplace=True)
    (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace=True)
    (2): Dropout(p=0.5, inplace=False)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace=True)
    (5): Dropout(p=0.5, inplace=False)
    (6): Linear(in_features=4096, out_features=10, bias=True)
  )
)
```

The batch size for training, testing and validation was taken to be 100. Train to validation split was 80:20. Test set used was the one provided by Fashion MNIST.
I freezed the weights of all layers except the last dense layers and changed the last layers out_features to 10 from 1000 for fashion MNIST's 10-class classification problem. The model was run for 10 epochs which took about 2 hours on the Tesla T4 GPU on Colab. Due to free GPU usage limits, I had to limit the number of epochs to 10.
Note**:** I am sure that the accuracy can be improved by choosing better pre-processing techniques or a better optimizer or longer training time. However, I had GPU constraints as mentioned before.

1. Preprocessing steps used:
    - Resize image to 256x256 pixels
    - Extract a patch of size 224x224 pixels from the centre of the image.

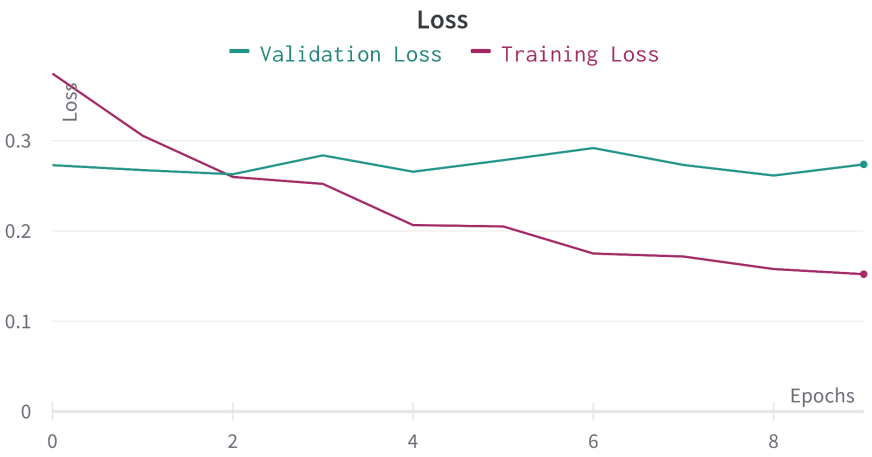
- Convert image to grayscale with 3 channels (since input to VGG has to be an image with 3 channels)
- Convert image to tensor image
- Normalisation using ImageNet's means and standard deviations since we are going to be using the VGG model pretrained on ImageNet.

I chose to use the preprocessing steps- resize and crop since these are data augmentation techniques that help to generalise the model better. Grayscale with 3 channels had to be used to get a 3 channel image that could be input to VGG, normalisation was required since it is recommended that pre-trained models on ImageNet be used with ImageNet's mean and standard deviation transforms.

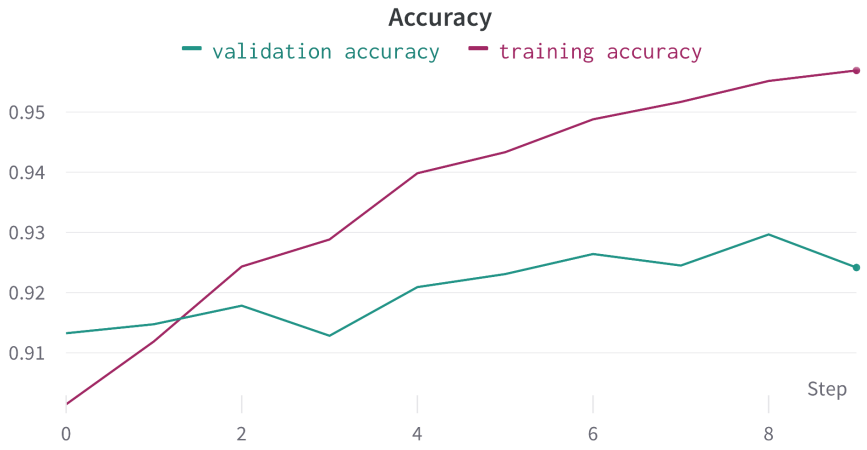
I used Adam optimizer since it has faster computation time and also lesser hyperparameters required for tuning.
The hyperparameters used were the default ones for Adam optimiser i.e. lr=0.001, eps=1e-08, weight_decay=0. I tried changing the learning rate but the model started overfitting (for 10 epochs). I could not try much parameter tuning because of GPU usage limits as each epoch takes about 10 minutes to run.

As recorded by wandb, here are the required plots:

- Training and validation loss vs epochs



- Training and validation accuracy vs epochs

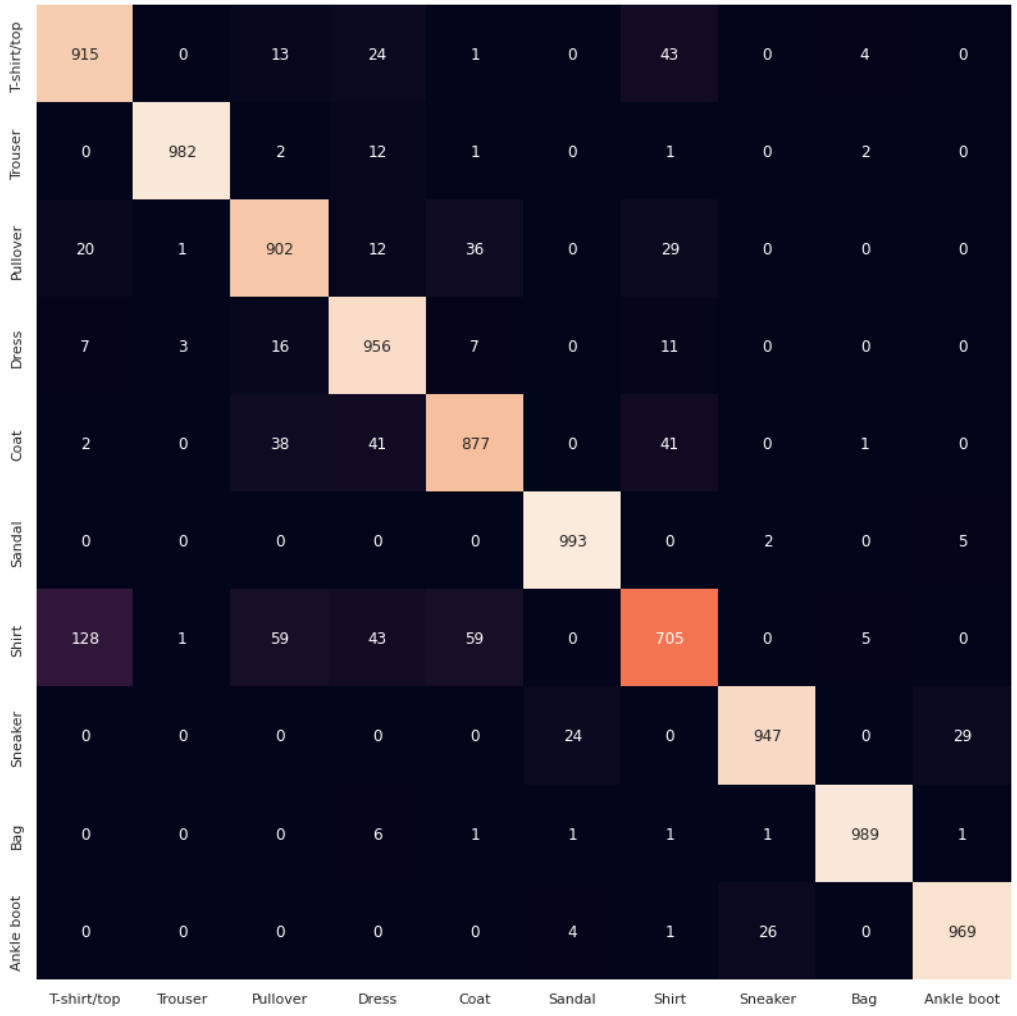


Observations:

- The training accuracy is increasing after each epoch. Compared to that, the testing validation set accuracy shows a general increasing trend but it is quite unstable and keeps fluctuating. It does end up being better than where it started though.
- The training loss reduces after each epoch, and the validation loss remains pretty much in the same range after 10 epochs.
- This observation might be due to overfitting of the model. Better hyperparameter tuning could help here. If I had enough GPU, I would have done that.
- With decrease in loss, accuracy increases for the training set. For the validation set, loss kind of reduces which brings in some increase in accuracy.
- The situation of not-reducing validation loss can be overcome by:
    - Making the model less complex - adding dropouts, reducing the number of layers etc.
    - Trying a different pre-processing technique
    - Choosing better hyperparameters.

2. Average accuracy obtained on the test set: **91.64%**

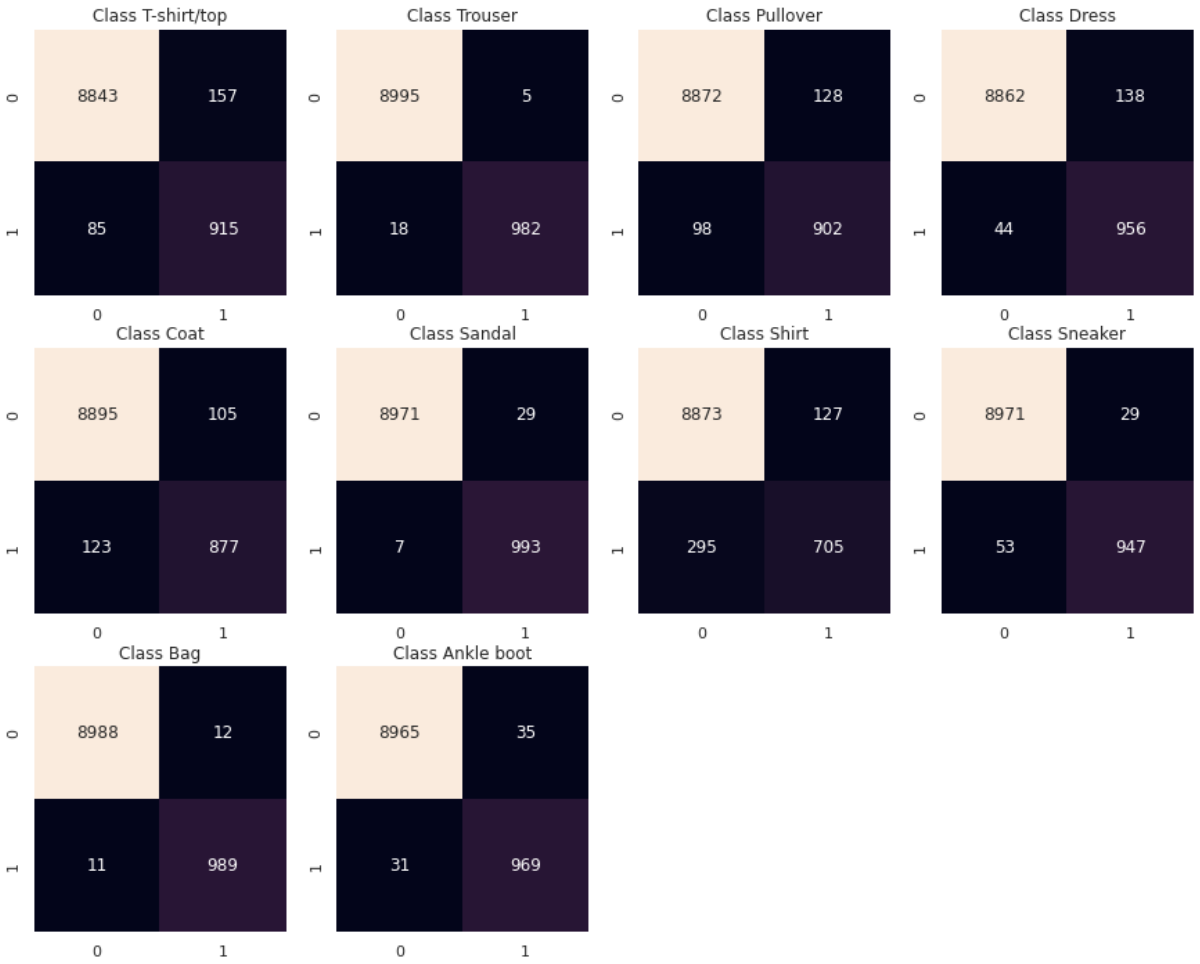
   Confusion matrix for the test set:



3. Class wise accuracies obtained:

```
Class-wise Accuracies:
Accuracy for class T-shirt/top : 91.5 %
Accuracy for class Trouser : 98.2 %
Accuracy for class Pullover : 90.2 %
Accuracy for class Dress : 95.6 %
Accuracy for class Coat : 87.7 %
Accuracy for class Sandal : 99.3 %
Accuracy for class Shirt : 70.5 %
Accuracy for class Sneaker : 94.7 %
Accuracy for class Bag : 98.9 %
Accuracy for class Ankle boot : 96.9 %
```

Class-wise confusion matrices:



Findings:

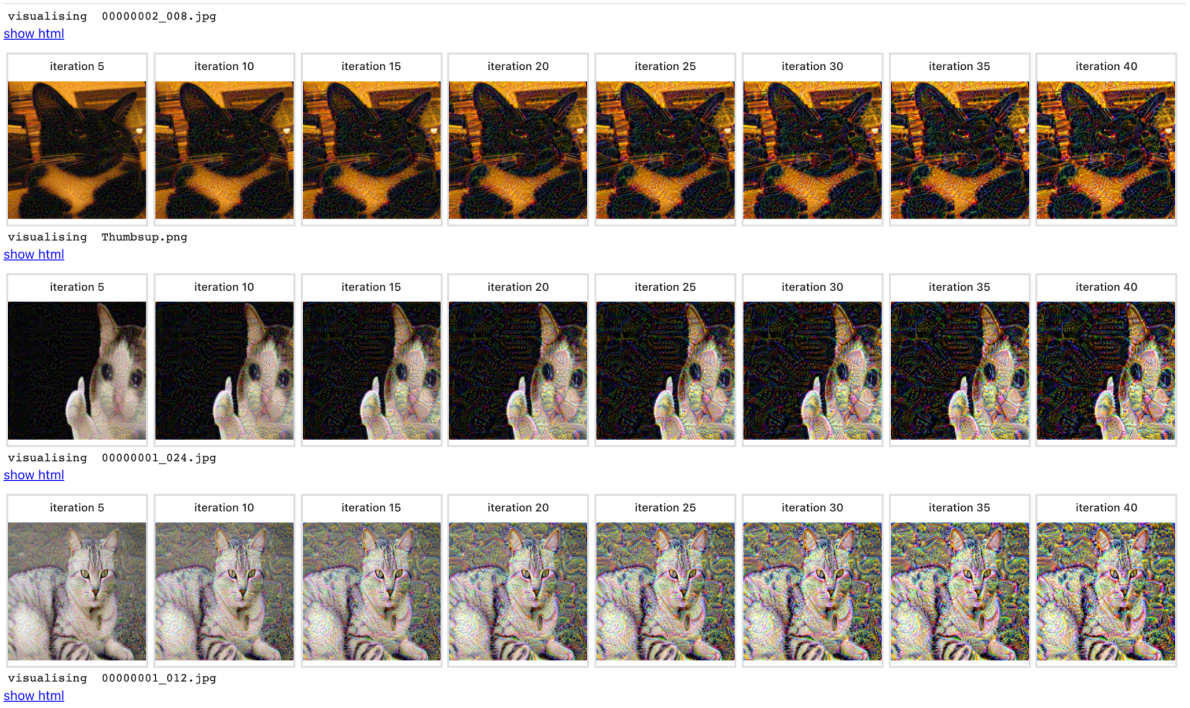
- The class Sandal was the most correctly identified with an accuracy of 99.3%
- The class Shirt was the least correctly identified with an accuracy of 70.5%
- From the confusion matrix, it can be seen that Shirt was most confused with T-shirt/top.
- Coat also had a low accuracy and was most confused with Dress and Shirt.
- Shirt was also majorly confused with Pullover and Coat.
- Pullover was confused with Coat (majorly).
- Sneakers were confused with Ankle Boots and Sandals.
- Shirt had the most number of False Negatives.
- T-shirt/Top had the most number of False Positives.

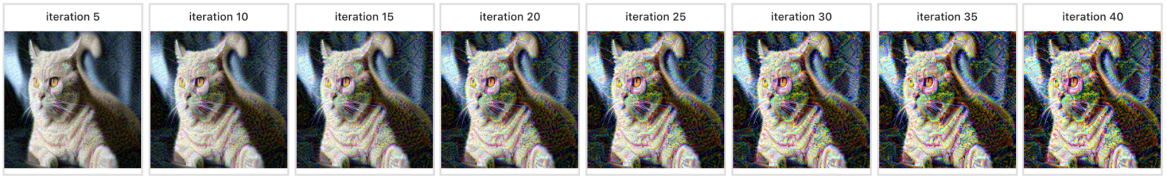
---

3. Extra Credit

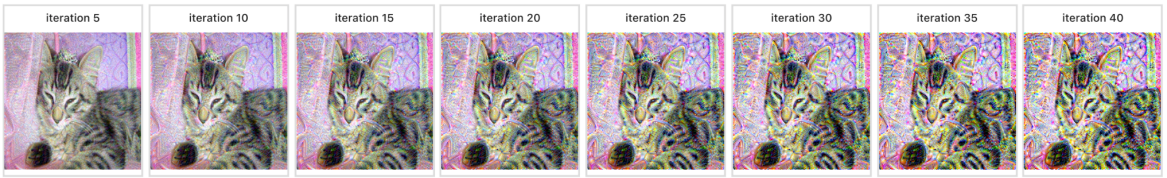
Please note that we were asked to use cat images as inputs to the VGG model trained on Fashion MNIST. Therefore, the results might not be the most optimal and inferences that would have been made, cannot.

1. I passed the given cat images dataset to the custom trained VGG model from Question 2. The layer visualised was 24, filter selected was 1. Iterations used were 40 and the images were displayed after every 5 iterations. The feature maps obtained are given below.

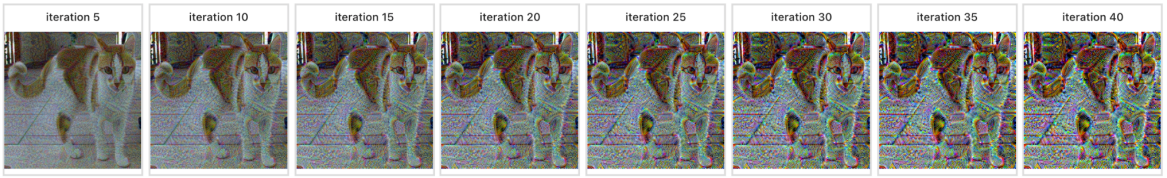




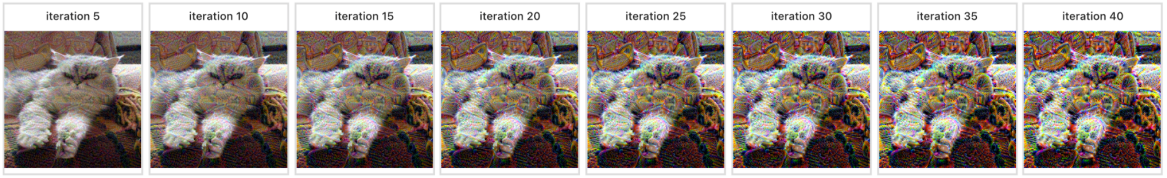
visualising 00000001_011.jpg
show html



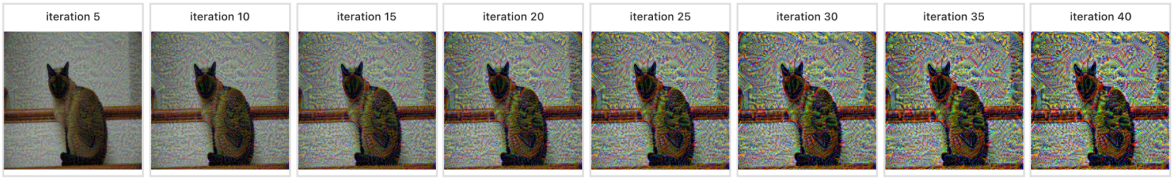
visualising 00000001_029.jpg
show html



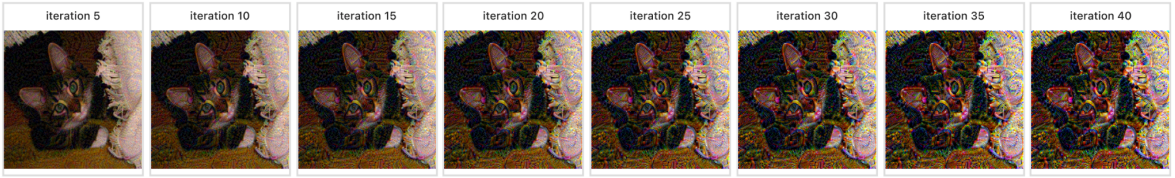
visualising 00000001_000.jpg
show html



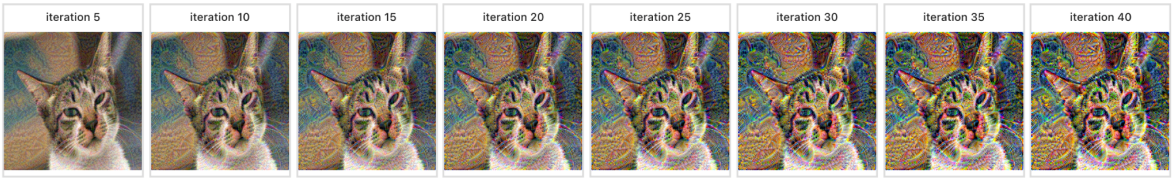
visualising 00000001_016.jpg
show html



visualising 00000001_017.jpg
show html

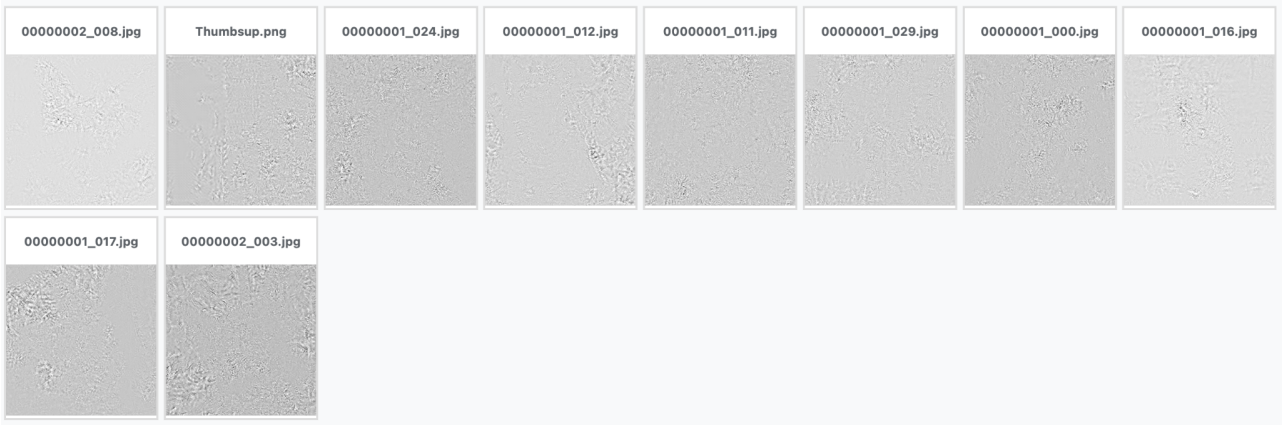


visualising 00000002_003.jpg
show html



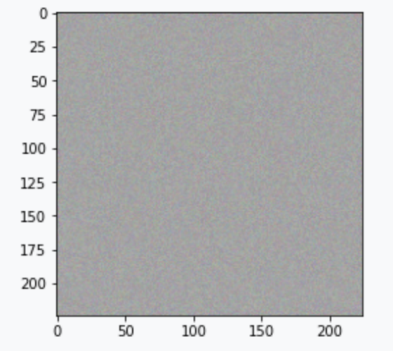
The images show how after each instance of gradient updation, the features in the image begin to get “highlighted” for filter 5. Each filter shades/focuses on a particular feature. The feature maps start capturing fine details of the image which would help in generalisation.

2. Using vanilla_backprop.py to visualise the given cat images:



These are basically gradients generated with vanilla backpropagation for each cat image for target class=2. Therefore, class 2 i.e. Pullover is taken as the target class here and gradients are calculated for each image with respect to that. Gradients array as an image is returned and displayed.

3. The randomly generated image used was:



Using this image to visualise **filter 5** of each layer in the CNN:

Here, the image is propagated through different layers of the CNN and the feature maps obtained for filter 5 that shades some particular feature of the training set is obtained. Since the image

doesn't carry much information (visually), the feature maps also do not look visually very discernible.

ML A2 Theory Questions

May 6, 2022

1 Question 1

Entropy basically is a measure of uncertainty in a system. Entropy for the different states y_i of a distribution with probabilities $p(y_i)$ is defined as:

$$S(y) = -\sum_i p(y_i) \log p(y_i)$$

Now, consider Y_p to be the measured distribution and Y_l to be the actual or theoretical distribution. The cross entropy for Y_p and Y_l is given by:

$$H(Y_p, Y_l) = -\sum_i p_{Y_p}(y_i) \log p_{Y_l}(y_i)$$

The KL divergence between them is defined as the composition of the entropy of distribution Y_p and the expectation of distribution Y_l in terms of the distribution Y_p. The expression is as follows:

$$D_{KL}(Y_p||Y_l) = \sum_i p_{Y_p}(y_i) \log p_{Y_p}(y_i) - \sum_i p_{Y_p}(y_i) \log p_{Y_l}(y_i)$$

KL divergence between Y_p and Y_l basically describes how different Y_p is from the perspective of Y_l. Now, from the expressions of H and D_{KL}, it can be easily seen that:

$$H(Y_p, Y_l) = D_{KL}(Y_p||Y_l) + S_{Y_p}$$

Thus, when S_{Y_p} is constant, then minimizing $H(Y_p, Y_l)$ is equivalent to minimizing $D_{KL}(Y_p||Y_l)$. For our case, Y_p and Y_l basically take the form of predicted labels and ground truth labels respectively. In all practical machine learning tasks, y i.e. the ground truth is given to us and we are required to find the predicted labels $\hat{y}$. Thus, the $P(Y_p)$ and $P(Y_l)$ become $P(y|x)$ and $P(\hat{y}|x)$ respectively. Thus, for the loss from KL divergence to be equal to the loss from cross entropy, $S_{y|x}$ has to be 0. Or, $P(y|x)$ has to be constant. Which is actually true since in supervising learning will always have prior knowledge of x and y and thus, $P(y|x)$.