INFOMCV Assignment 4
Lui Holohan and Yiwen Chen (39)

Description and motivation of your baseline model and four variants

(For your baseline model, add a description and motivation of the architecture and parameters (which layers, which dimensions, how connected, etc.). Also use torchsummary(). For each of the four variants, add a description of which property differs from the baseline model, and why this choice was made. Make sure you name/number your models so you can refer to them. Approx. 0.5-1 page.)

Our LeNet5 baseline model represents a standard baseline implementation of the LeNet5 convolutional neural network architecture. The model was initially introduced by Yann LeCun in 1989 to deal with the weak generalization exhibited by single-layer networks. Therefore, when implementing the baseline model and observing potential variations to it, we wanted to gain an understanding of the original implementation. For instance, originally, max pooling was not used in the original LeNet5 architecture, and the activation function was tanh as opposed to ReLU which is more common today (de facto activation function). For our baseline model we implemented a basic pipeline which would be common today, whilst observing how these choices have changed over time in order to give us potential adaptations for our four variants.

For our baseline model, we have two convolutional layers. The first takes our greyscale image and produces six feature maps using 5x5 filters with padding=2. The second layer takes the six input maps produces 16 feature maps with no padding. We used ReLU as our activation function. We used max pooling for our baseline model. We have three fully connected layers which flatten the output to ultimately 10 units. This allows us to classify the images in the FashionMNIST dataset which has a classification of images ranging from 0-9. Finally, we implemented a softmax layer to produce our probability distribution over the 10 classes. We then used kaiming_uniform to initialize the weights to zero before forward passing through the network.

In our first model variation (LeNet5_var1) we added a dropout layer which we though would potentially improve the models performance by reducing overfitting. We used a dropout rate of 0.5, a common implementation, providing a balance between regularization and sufficient learning during training. We added the dropout layer after the first fully connected layer.

In the second variation (LeNet_var2), we added batch normalization to LeNet_var1 in order to further optimize the training process. In fact, the results show this to be our sharpest drop in a single model performance. Across all models, notwithstanding LeNet5_var2, there is a steady increase in the training and validation scores. The significant drop in performance after adding batch normalization could be for a number of reasons, including appropriating too many regularization techniques, incorrect weight initialization or incorrect batch sizes. Nonetheless, it was interesting to be able to compare this particular model pairwise to LeNet5_var1 and LeNet5_var3, both of which had significantly higher training and validation scores (88.68, 88.56 and 88.86, 88.37 respectively).

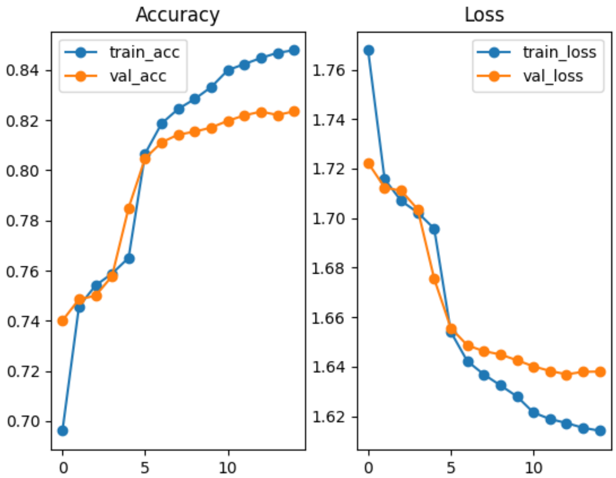
Following this, we changed the activation function from ReLU to LeakyReLU (LeNet_var3). This was to deal with potential dying neurons and make the training more robust. LeakyReLU allows a model to learn from negative inputs, which exhibits its modular potential, making it a perfect element to "add" to out architecture, as it is flexible and versatile. Furthermore, it is capable to learn more complex patterns in the data, and allow us to seek out some small performance in our model (in this case, however, it was the most drastic improvement in both training and validation scores).

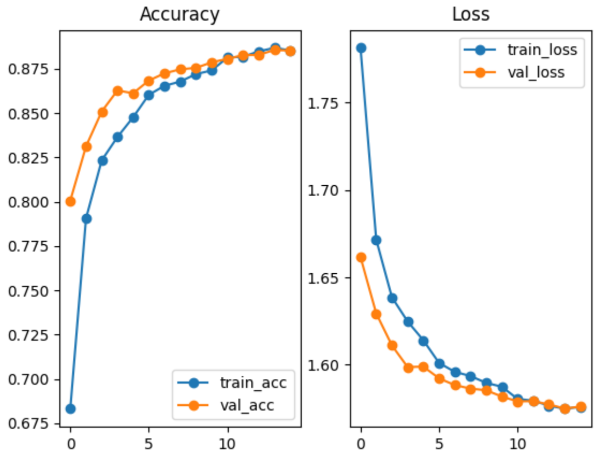
For the final model variation (LeNet_var4), we replaced max pooling with average pooling. We assumed this would help with the model's ability to generalize its training to the test dataset. It also seemed logical to include average pooling when considering our previous alterations; namely changing the activation function and implementing batch normalization. This seemed to improve the model's accuracy, which made sense as this pooling method is designed to become less sensitive to

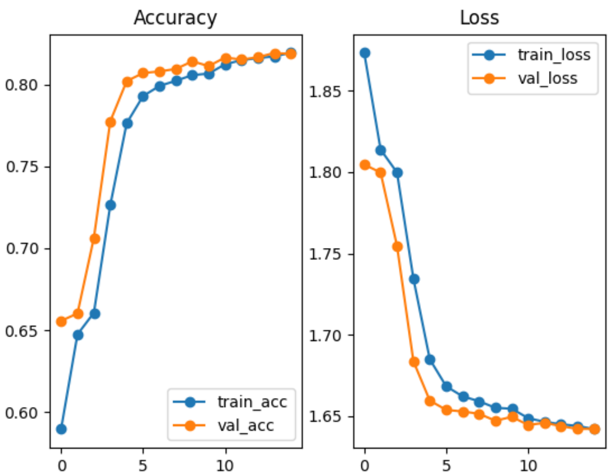
noise or variations in the input data. Also, knowing the history of the LeNet5 implementation, we were curious as to what the effects of this method would have on our model.

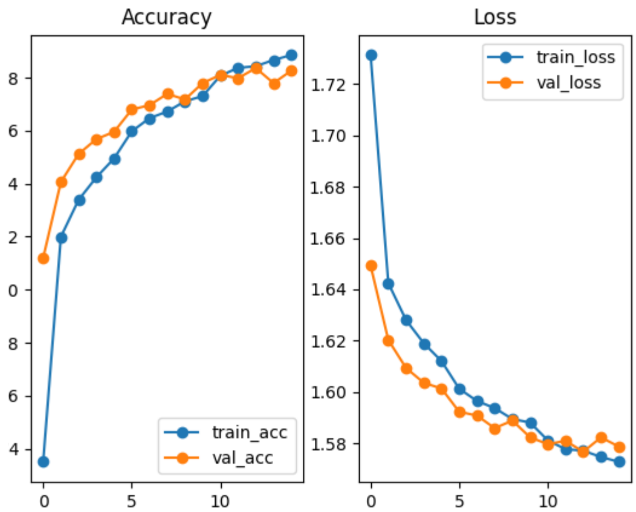
Training and validation loss for all five models

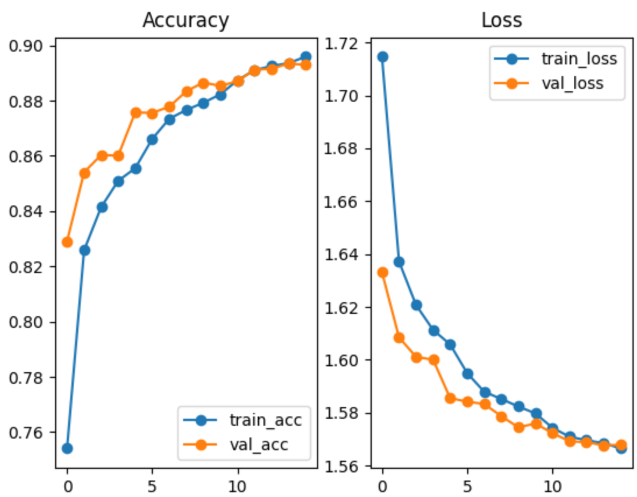
From top-left to bottom: LeNet5_baseline; LeNet5_var1; LeNet5_var2; LeNet5_var3; LeNet5_var4;











Link to your model weights

(Link should be accessible by r.w.poppe@uu.nl and m.ning@uu.nl.)

https://github.com/B00tiam/CV_assignment/tree/main/Assignment_4/models

Table with training and validation top-1 accuracy for all five models

(Fill the table below.)

Model name	Training top-1 accuracy (%)	Validation top-1 accuracy (%)
LeNet5_baseline	84.79	82.35
LeNet5_var1	88.68	88.56
LeNet5_var2	81.92	81.92
LeNet5_var3	88.86	88.37
LeNet5_var4	89.59	89.34

Discuss your results in terms of your model

The complexity does not increase significantly across the five models. In the first two models (LeNet5_ baseline and LeNet5_var1) the total parameters are 61,706 which increase to 61,750 in the subsequent model variants.

The types of layers in the baseline model include two convolutional layers and three fully connected layers, as well as the ReLU activation functions, linear, pooling and final softmax layers. In LeNet5_var1, a dropout layer is added. Following the inclusion of dropout, in LeNet5_var2, batch normalization is introduced after every convolutional layer with 6 and then 16 feature maps. Including batch normalization greatly reduced both the training and validation scores, whilst increasing the training and validation loss. However, after further adapting the architecture in LeNet5_var3 by changing the activation function from ReLU to LeakyReLU, the scores seemed to significantly improve, and continue on their upward trajectory. This eventually culminated in our final architecture, LeNet5_var4, which was technically our most complicated network, with 61,750 total parameters, all of which were trainable, and 15 total layers compared to our baseline model, which had only 12 layers.

The estimated total size of the baseline model was 0.35 MB, which increased to 0.40 MB in the last model (LeNet5_var4)

Discuss the differences between the two models evaluated on the test set

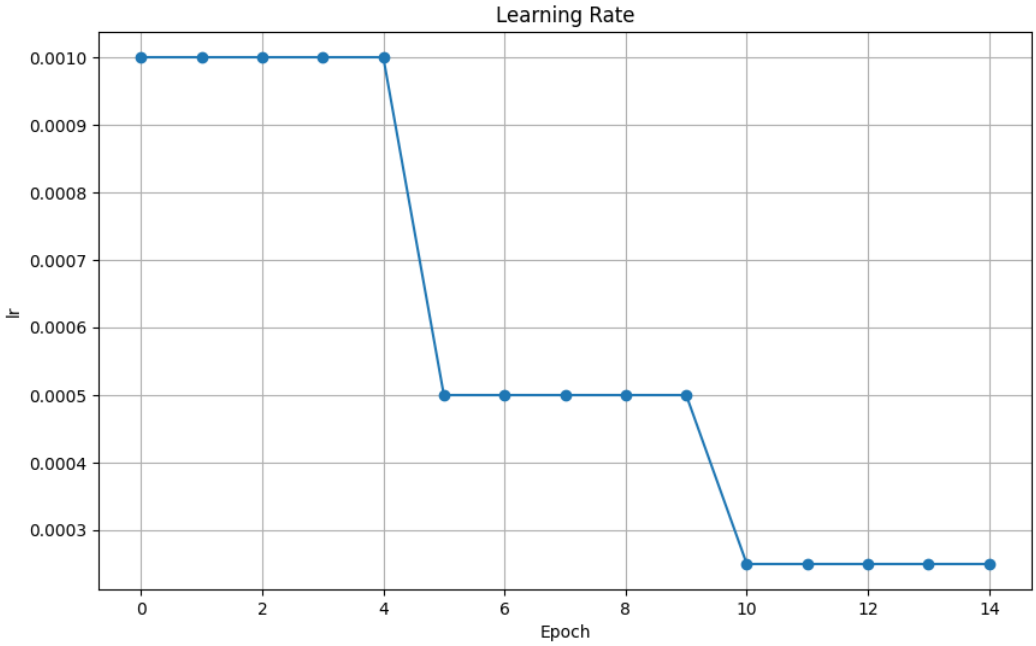
The two models we tested were the two models with the highest training and validation accuracy, namely, LeNet_var3 and LeNet_var4. There is only one major difference between the two architectures, namely, the difference in pooling from max pooling to average pooling. Again, we took inspiration from the original architecture for this implementation. It improved the models performance in both training and validation scores over LeNet_var3. Clearly, as indicated by the improvement generally from LeNet5_var3 and LeNet5_var4, the data in the FashionMNIST dataset (which, it is worth noting, is distinct from the dataset upon which LeNet5 was originally trained and tested on). It is clear that more nuance is required when discerning items of clothing than hand-drawn numbers. Average pooling increased the training accuracy and validation scores which suggests that the model improved by picking on on subtler details and differences from across the data

The more inclusive feature extraction that the model obtained from implementing average pooling had a slight improvement on generalizing from the training to the test data, as exhibited by the difference in the two models performance on the test data (LeNet5_var3 = 87.76; LeNet5_var = 88.78)

Choice tasks

(Indicate which ones you did, and how you did them; Approx. half a page.)

[Choice 1] -



In order to implement choice task 1, we slightly modified our original train_validate.py script and more specifically the train_validate() function to include the StepLR function from the torch library (torch.optim.lr_scheduler) which allowed us to specify a step size (how many epochs) and a gamma value (which was by how much we wanted to reduce the learning rate, in this case by 0.5)

[Choice 2] –

In order to implement choice task 2, we used KFold() from the library scikit-learn, which can help us split the whole dataset into K folds. We used loop to split data into different indexes and ran our baseline model in each fold.

By running the best model in 5-fold cross validation, we got its performance: the best accuracy value of each folds are: 0.8904, 0.8889, 0.8904, 0.8879, 0.8877, and they belong to these epochs in each of the folds: 13, 14, 14, 13, 14.

[Choice 4] –

We used multiple methods to implement data augmentation, here is our code:

```
transform = transforms.Compose([
    transforms.RandomRotation(10),  # Rotate randomly up to 10 degrees
    transforms.RandomHorizontalFlip(),  # Randomly flip horizontally
    transforms.ColorJitter(brightness=0.1, contrast=0.1, saturation=0.1, hue=0.1),
    # Adjust brightness, contrast, saturation, and hue
    transforms.ToTensor(),
    transforms.Normalize( mean: (0.5,), std: (0.5,))
])
```

In our code, RandomRotation(10) can rotate images randomly, with a maximum rotation angle of 10 degrees; RandomHorizontalFlip() can randomly flip images horizontally to increase data diversity; transforms.ColorJitter() can randomly adjust the brightness, contrast, saturation and hue of an image. By implementing these techniques, we got a better accuracy on test dataset.

[Choice 5] –

In order to implement the t-SNE(t -Distributed Stochastic Neighbor Embedding), we used the TSNE method from the library scikit-learn. This method can represent dimensionally reduced data distribution in a low-dimensional space to help us analyse performance of models. Take the var as example: we can analyse the label classification of model on prediction better by using both confusion matrix and t-SNE graph.

```
Test Accuracy: 0.8517
Confusion matrix of test:
[[844   0  29  37   8   3  57   0  22   0]
 [  0 965   3  24   4   1   1   0   2   0]
 [ 12   0 842   8 107   0  26   0   5   0]
 [ 25   6  27 886  33   2  17   0   2   2]
 [  1   1 124  38 803   0  24   0   9   0]
 [  0   0   0   2   0 924   0  44   2  28]
 [199   3 169  38 135   1 425   0  30   0]
 [  0   0   0   0   0  17   0 902   1  80]
 [  2   2   9   2   3   5   2   4 970   1]
 [  0   0   0   0   0   5   0  38   1 956]]
```