2021 Fall STATSM231A/CSM276A: Pattern Recognition and Machine Learning

Yaxuan Zhu
Department of Statistics
University of California, Los Angeles
9401 Bolter Hall., Los Angeles, California 90095
`yaxuanzhu@g.ucla.edu`

Abstract

In this course, you will learn multiple algorithms in machine learning. In general, for each homework, you are required to write the corresponding codes. We will provide a structure of code and a detailed tutorial online. Your job is (1) to deploy and understand the code. (2) Run the code and output reasonable results. (3) Make ablation study and parameter comparison. (4) Write the report with algorithms, experiment results and conclusions. For detailed instructions, please check the specification of each homework.

Homework submission forms

You have to submit both code and reports. For the code, zip everything into a single zip file. For the report, use this latex template. When submitting, no need to submit the tex file. Submit one single pdf file only.

1 Written Part

1.1 Problem1

Before answering the question, let us briefly discuss the structure of a fully connected neural network. For classification, the general structure is as follows:

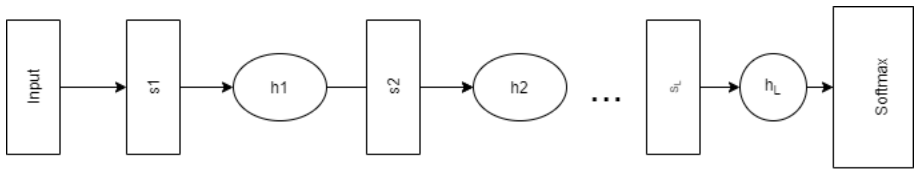

Figure 1: An example feed forward neural network

Here, the output is passed through a set of hidden layers that consist of a multiplication with weights and a non-linearity. The output of the last layer is passed through a softmax function, which generates the probabilities for each class. The softmax operation is defined as follows:

$$\sigma(h_{L_i}) = \frac{e^{h_{L_i}}}{\sum_{j=1}^{K} e^{h_{L_j}}}$$

where K is the amount of classes, h_{L_i} is the output of the ith neuron at h_L and h_{L_j} is the output of the jth neuron at h_L

When deriving the back-propagation, we need a loss function. In classification tasks, the most commonly used loss function is cross entropy, which is defined as:

$$H_{y'}(y) := -\sum_{i=1}^{K} y_i \log(\hat{y_i})$$

where $\hat{y_i}$ is $\sigma(h_{L_i})$ and y_i is the desired probability for that class. Since the class probabilities are one-hot encoded, y_i is 1 for the actual class of the sample and 0 for every other class. The $\hat{y_i}$ probabilities will be generated by the soft-max function.

Now that we have a better understanding of neural networks. We can derive the back-propagation formula. To do so, we need to use the chain rule starting from the loss function. For the set of weights at the lth layer or w_l, we need to include the partial derivative of every term that is dependent on w_l in the back-propagation formula. If we do so, we get the following:

$$\frac{\partial Loss}{\partial w_l} = \frac{\partial Loss}{\partial h_L} \prod_{i=l+1}^{L} (\frac{\partial h_i}{\partial s_i} \frac{\partial s_i}{\partial h_{i-1}}) \frac{\partial h_l}{\partial s_l} \frac{\partial s_l}{\partial w_l}$$

Note that when l = L or when we are calculating the gradient for the final layer, the product term will not be in the formula since there are no layers before getting to the final layer during back-propagation.

Now, all we have to do is calculate each partial derivative.

The last layer of the neural network must be made up as many neurons as there are output classes, since each neuron will be responsible for a specific class. Therefore, to calculate the first partial derivative, we need to calculate partial derivative of the loss function with respect to each neuron in the final layer and combine the results in a vector. In other words:

$$\frac{\partial Loss}{\partial h_L} = \begin{bmatrix} \frac{\partial Loss}{\partial h_{L_1}} \\ \frac{\partial Loss}{\partial h_{L_2}} \\ \dots \\ \frac{\partial Loss}{\partial h_{L_k}} \end{bmatrix}$$

where each h_{L_i} is the output of the ith neuron at the output layer L. Now, all we have to do is find each element of the vector.

The loss function can be thought of as follows:

$$\begin{aligned} Loss &= -\textstyle\sum_{i=1}^{K} y_i log(\frac{e^{h_{L_i}}}{\sum_{j=1}^{K} e^{h_{L_j}}}) \\ &= -\textstyle\sum_{i=1}^{K} y_i log(e^{h_{L_i}}) + \sum_{i=1}^{K} (y_i) log(\sum_{j=1}^{K} e^{h_{L_j}}) \end{aligned}$$

Since y is one hot encoded, $\sum_{i=1}^{K}(y_i)$ will be 1. So, we get:

$$Loss = -\textstyle\sum_{i=1}^{K} y_i h_{L_i} + log(\sum_{j=1}^{K} e^{h_{L_j}})$$

Using this formula, we can get the partial derivative with respect to h_{L_i} as follows:

$$\begin{aligned} \frac{\partial Loss}{\partial h_{L_i}} &= -y_i + \frac{\partial \sum_{j=1}^{K} e^{h_{L_j}}}{\partial h_{L_i}} \frac{1}{\sum_{j=1}^{K} e^{h_{L_j}}} \\ &= -y_i + \frac{e^{h_{L_i}}}{\sum_{j=1}^{K} e^{h_{L_j}}} \\ &= -y_i + \hat{y_i} \end{aligned}$$

To get $\frac{\partial Loss}{\partial h_L}$, the above result must be obtained for each neuron and all of the results must be combined in a vector. In vector form, the result can be thought as:

$$\frac{\partial Loss}{\partial h_L} = \hat{y} - y$$

Where $\hat{y}$ is the output vector of the model after the softmax operation and y is the one-hot encoded vector of expected class probabilities.

Now, let us move on to the other partial derivatives in our derived formula.

We can easily see that:

$$\frac{\partial h_i}{\partial s_i} = \frac{\partial f_i(s_i)}{\partial s_i} = f'(s_i)$$

$$\frac{\partial s_i}{\partial h_{i-1}} = w_i$$

$$\frac{\partial h_l}{\partial s_l} = f'(s_l)$$

$$\frac{\partial s_l}{\partial w_l} = h_{l-1}$$

Plugging these to our original formula, we get the following:

$$\frac{\partial Loss}{\partial w_l} = (\hat{y} - y) \prod_{i=l+1}^{L} (f'(s_i)w_i) f'(s_l) h_{l-1}$$

When $l = L$ or when we are calculating the back-propagation for the final layer, we get:

$$\frac{\partial Loss}{\partial w_L} = (\hat{y} - y) f'(s_L) h_{L-1}$$

When $1 < l < L$ or when we are calculating the back-propagation for any layer between the final and first layer, we get:

$$\frac{\partial Loss}{\partial w_l} = (\hat{y} - y) \prod_{i=l+1}^{L} (f'(s_i)w_i) f'(s_l) h_{l-1}$$

When $l = 1$ or when we are calculating the back-propagation for the first layer, we get:

$$\frac{\partial Loss}{\partial w_1} = (\hat{y} - y) \prod_{i=2}^{L} (f'(s_i)w_i) f'(s_1) x$$

Now, let us do the same operation for the bias terms b_l

$$\frac{\partial Loss}{\partial b_l} = \frac{\partial Loss}{\partial h_L} \prod_{i=l+1}^{L} \left(\frac{\partial h_i}{\partial s_i} \frac{\partial s_i}{\partial h_{i-1}}\right) \frac{\partial h_l}{\partial s_l} \frac{\partial s_l}{\partial b_l}$$

We have already calculated most of these terms. The only one we need is $\frac{\partial s_l}{\partial b_l}$. This can easily seen to be 1. So, the general formula for the bias becomes:

$$\frac{\partial Loss}{\partial b_l} = (\hat{y} - y) \prod_{i=l+1}^{L} (f'(s_i)w_i) f'(s_l)$$

When $l = L$, we get:

$$\frac{\partial Loss}{\partial b_L} = (\hat{y} - y) f'(s_L)$$

When $1 < l < L$, we get:

$$\frac{\partial Loss}{\partial b_l} = (\hat{y} - y) \prod_{i=l+1}^{L} (f'(s_i)w_i)f'(s_l)$$

When $l = 1$, we get:

$$\frac{dLoss}{db_1} = (\hat{y} - y) \prod_{i=2}^{L}(f'(s_i)w_i)f'(s_1)$$

Thus, we managed the derive the back-propagation derivatives for all of the neural network parameters in the question.

2 Coding part: Neural Network (Fully connected Layer and CNN)

2.1 Problems

In this assignment, the main goal is to build a neual network model that is capable of determining the class an image belongs to. To achieve this task, we define a Convolutional Neural Network (CNN) and train this network on 32x32 RGB images acquired from the CIFAR-10 dataset

2.2 Introduction

The CNN model employed in the given tutorial is composed of four main components. Let us briefly describe each component of the model.

2.2.1 Convolutional Neural Network

The first component in the model is the convolutional layer, which is what gives the name of CNNs. Convolutional neural networks are used to extract patterns from image data that can aid in the task the model is trying to solve. This is done by traversing an image in a sliding window fashion vertically and horizontally by a set of weights. This allows the model to look at a field within the image when determining important patterns instead of treating each pixel as individual values. The convolutional operation can best be understood by looking at the following image:

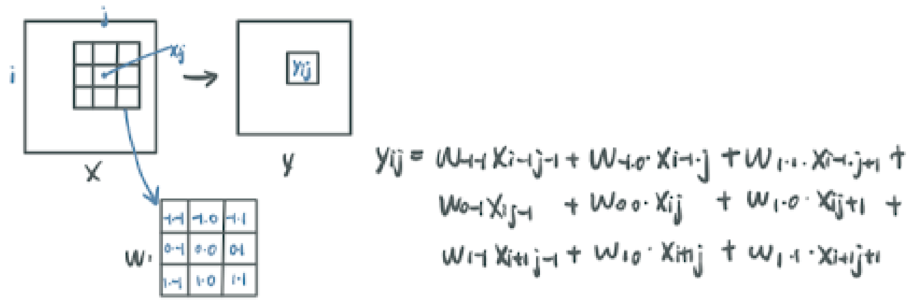
j
i
xij
yij
X
y
W
-1-1 | -1.0 | -1.1
0-1 | 0.0 | 0.1
1.-1 | 1.0 | 1.1
yij = W-1-1 Xi-1j-1 + W-1.0 · Xi-1.j + W1.1 · Xi-1.j+1 +
W0-1 Xij-1 + W0 0 · Xij + W1.0 · Xij+1 +
W1-1 Xi+1j-1 + W10 · Xi+1j + W1.1 · Xi+1j+1

Figure 2: The calculations done by a CNN on the ith row and jth column of an image

Here, a 3x3 convolutional filter with a set of weights W is traversed around an image. For each pixel, a dot product between the layer weights of that pixel and the value of the pixel is performed, which gives the output of the convolutional layer. This operation is performed for all the channels in the input. Overall, in a convolutional layer there are number of input channels times number of output channels amount of sets of weights, where each weight set is a matrix. The size of the matrix is determined by the kernel size of the layer. The formula for the output dimension for the CNN is as follows:

$$dimension_{out} = \frac{dimension_{in} + 2p - k}{s} + 1$$

where $dimension_{out}$ is the dimension of the output of the CNN, $dimension_{in}$ is the dimension of the input, p is the amount of padding, k is the kernel size and s is the stride.

The weight coefficients of the CNN are updated using a variation of backpropagation. The simplest form of backpropagation is obtained when stochastic gradient descent is utilised to update model parameters. Here, the weights are updated as follows:

$$W_{new} = W_{old} - \alpha \frac{\partial Loss}{\partial W}$$

where α is the learning rate and W is the weight matrix in the CNN.

2.2.2 Non-Linearity

The non-linearity function is used after each convolutional layer in order to allow the model to learn non-linear relationships between input and the desired output. In the tutorial, the ReLU nonlinearity is utilised. However, there are also other common non-linearities used to train CNNs such as: leaky ReLU and tanh. The formulas for these non-linearities are:

$$ReLU(x) = max(0, x)$$
$$LeakyReLU(x) = \begin{cases} \alpha x & : x \leq 0 \\ x & : x > 0 \end{cases}$$
$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

where a is the slope of the leaky ReLU.

2.2.3 Pooling Layer

The pooling layer is used to reduce the dimension of the CNN outputs. The idea is to combine the learned features in the CNN output in a way that only makes the salient features remain. This aids in model learning and is also used to prevent over-fitting. In the tutorial, the max pooling layer is utilised as the desired pooling method. The methodology of max pooling can be illustrated as follows:

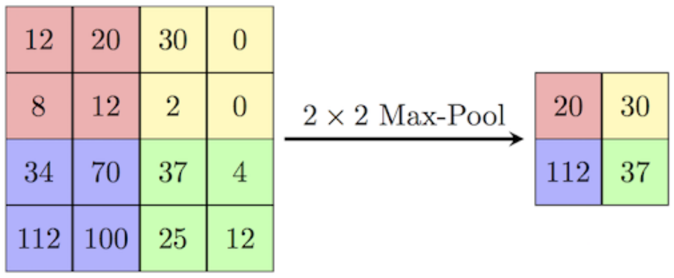

Figure 3: The illustration of the max pooling operation

As can be seen, max pooling only lets the maximum value in the section it is looking at pass. This also implies halving the dimension of the input.

2.2.4 Fully Connected Neural Network

A Fully Connected Neural Network (FCNN) is a model where sets of neurons are connected to each other one after the other. Each set is called a hidden layer. The general structure of the model can be seen on Fig. 4

Each neuron is connected to every neuron at the next layer with a set of weights. Each hidden layer also has a bias term which is just a coefficient. The goal of this structure is to find a set of weights between connections such that a loss function is minimised. This makes the model suitable for both classification and regression tasks by simply playing with the loss function.

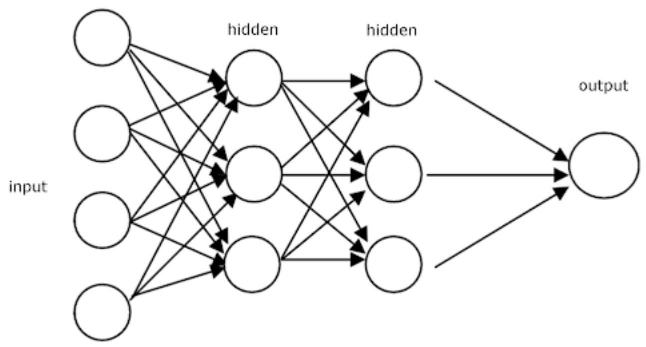

Figure 4: The illustration of a fully connected neural network with 2 hidden layers

The output of each hidden layer is calculated as follows:

$$Out = f(W \cdot x + b)$$

where f() is the activation function or non linearity of the hidden layer, W is the set of weights, x is the input to the layer and b is the bias. For the final layer in a classification task, the softmax operation is used as the activation function.

The parameters of the neural network are updated using a variation of stochastic gradient descent. In the vanilla form, the updates are as follows:

$$W_{new} = W_{old} - \alpha \frac{\partial Loss}{\partial W}$$
$$b_{new} = b_{old} - \alpha \frac{\partial Loss}{\partial b}$$

Now that we have a better understanding of the components of the model, let us move on to the experimental part.

2.3 Experiments

2.3.1 Part 1: Drawing training progress

How to run The code for this section is contained in the "Part 1_Draw loss and acc" subfolder. Here, modelClass.py defines the model structure and trainTestNeuralNet.py is used to train and test the model on the dataset. The code is written using the Spyder IDE. You can run the code section by section or you can run it all together. To use the sections, you need to be using the Spyder IDE. Each section in trainTestNeuralNet.py is briefly explained using comments. To load the libraries, run section 1. To get the data, run section 2. To train the model, run section 3. To plot the graphs, run sections 6 and 7. To get how the model performs on 20 random test images, run section 8.

Experiment Description Here, we are tasked with displaying how loss and accuracy changes through the epochs when the model given in the tutorial is trained. To show this, first we changed the batch size to 32. Then, we recreated the model using PyTorch. The structure of the model in the tutorial was as follows:

```
Input (dim = 3*32*32) -->
convLayer (OutputChannel = 6; Kernel = 5*5; Stride = 1) -->
Relu -->
MaxPooling (Kernel = 2*2) -->
convLayer (OutputChannel = 16; Kernel = 5*5; Stride = 1) -->
Relu -->
MaxPooling (Kernel = 2*2) -->
```

```
Flatten -->
Fully Connected(Input neuron size=16*5*5, output neuron size = 120) -->
Relu -->
Fully Connected(Input neuron size=120, output neuron size = 84) -->
Relu -->
Fully Connected(Input neuron size=84, output neuron size = 10) -->
Softmax -->
Class Probabilities
```

The model structure was not changed in this experiment. However, we modified the training function so that at the end of each epoch, the model is tested on both the test data and the training data. We noted the accuracy and loss for each test. This process was repeated for 10 epochs and the accumulated results were displayed using matplotlib. The results are as follows:

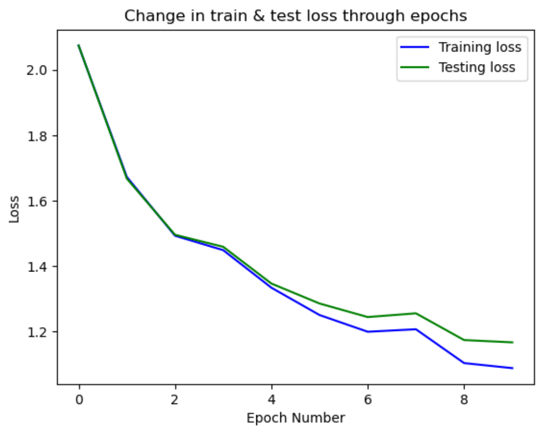

Figure 5: The change in loss on the training and test data through epochs

As can be seen, both the training and testing loss values decrease through the epochs. This is expected since as the model iterates over the data, the weights get updated and the model learns the input data better, which allows it to perform better on unseen data. In the end, it can be seen that the testing loss is slightly higher than the training loss. This is expected since the model performs updates based on the training data so it performs better on that set.

Now let us look at the change in accuracy.

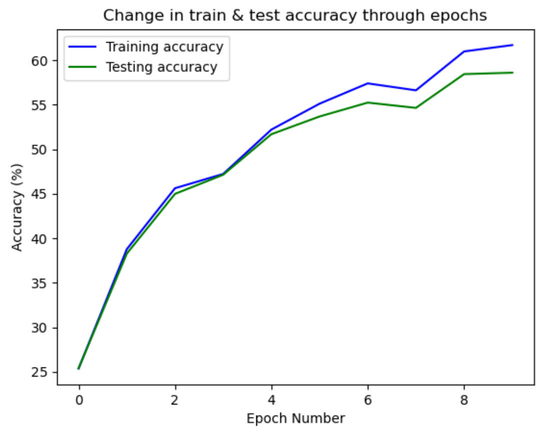

Figure 6: The change in accuracy on the training and test data through epochs

As can be seen, the accuracy is inversely related to the loss. As the epochs progress the model learns the data and performs better and is therefore capable of achieving a higher level of accuracy. As before, the training accuracy is slightly higher than the test accuracy because the model makes updates based on the training set.

Now, let us see how the model performs on a subset of the test set. Here, to ensure all the classes are covered, 2 images from each class in the test set were randomly picked. The result is the following:

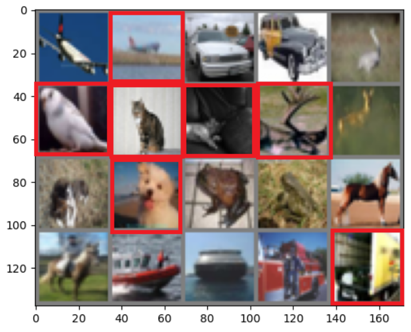

Figure 7: A sample of 20 images from the test set. The images highlighted in red are where the model mislabelled the data

To better understand the model labelling, let us construct a matrix for the predicted and actual label of each image.

Table 1: The predicted and actual labels of the 20 images sampled from the test set

Label: Plane Prediction: Plane	Label: Plane Prediction: Ship	Label: Car Prediction: Car	Label: Car Prediction: Car	Label: Bird Prediction: Bird
Label: Bird Prediction: Cat	Label: Cat Prediction: Bird	Label: Cat Prediction: Bird	Label: Deer Prediction: Bird	Label: Deer Prediction: Deer
Label: Dog Prediction: Dog	Label: Dog Prediction: Deer	Label: Frog Prediction: Frog	Label: Frog Prediction: Frog	Label: Horse Prediction: Horse
Label: Horse Prediction: Horse	Label: Ship Prediction: Ship	Label: Ship Prediction: Ship	Label: Truck Prediction: Truck	Label: Truck Prediction: Cat

As there are a total of 7 errors, the accuracy of the model on this set is $13/20 = 65\%$. From the previous section, we found out that the accuracy of the model on the entire test set is 58.6% after training the model for 10 epochs, so there is a 6.4% discrepancy between the results. However, since the accuracies are not too far from each other, the 20 images we have can be said to be representative of the test set. However, since the test set contains more images, using that set to make comments about model performance would be wiser.

2.3.2 Part 2: Comparison of the Baseline with other models

How to run The code for this section is contained in the "Part 2_Comparison Methods subfolder. Here, the code to create fully connected neural neetworks without the convolutional layers is in the Neural Network subfolder. In this subfolder, mainCode.py is for training and testing the models and neuralNetworkDefiner.py is for defining each model architecture. Each section of mainCode.py is briefly explained using comments. Run section 1 to load the libraries, section 2 to get train and test data, section 3 to import model descriptions, section 4-7 to train the models and section 8 to test a specific model. To run the code section by section, you need to be using the Spyder IDE.

The code to create different convolutional neural networks is in the CNNs subfolder. Here, mainCode.py is for training and testing the CNN models and cnnDefiner.py is for defining each CNN model architecture. Each section of mainCode.py is briefly explained using comments. Run section 1 to load the libraries, section 2 to get train and test data, section 3 to import model descriptions, section 4-10 to train and test different models. To run the code section by section, you need to be using the Spyder IDE.

Experiment Description In this section, we are asked to modify the model in the tutorial in order to see how the change in model description affects the final results. In total, 11 different models were created for this comparison section. Let us briefly describe each of them.

2 layer neural network: For this model, the convolutional and pooling layers were removed from the original model. Instead, a neural network with 1 hidden layer and 1 output layer was utilised. The first hidden layer had 32*32*3 input neurons and 120 output neurons. The output layer had 120 input neurons and 10 output neurons. Other parameters of the model (learning rate, activation function, etc.) were not changed.

3 layer neural network: For this model, the convolutional and pooling layers were removed from the original model. Instead, a neural network with 2 hidden and 1 output layer was utilised. The first layer had 32*32*3 input neurons and 120 output neurons. The second hidden layer had 120 input neurons and 50 output neurons. The output layer had 50 input neurons and 10 output neurons. Other parameters of the model (learning rate, activation function, etc.) were not changed.

4 layer neural network: For this model, the convolutional and pooling layers were removed from the original model. Instead, a neural network with 3 hidden and 1 output layer was utilised. The first layer had 32*32*3 input neurons and 120 output neurons. The second hidden layer had 120 input neurons and 50 output neurons. The third hidden layer had 50 input neurons and 30 output neurons.The output layer had 30 input neurons and 10 output neurons. Other parameters of the model (learning rate, activation function, etc.) were not changed.

5 layer neural network: For this model, the convolutional and pooling layers were removed from the original model. Instead, a neural network with 4 hidden and 1 output layer was utilised. The first hidden layer had 32*32*3 input neurons and 120 output neurons. The second hidden layer had 120 input neurons and 50 output neurons. The third hidden layer had 50 input neurons and 30 output neurons. The fourth hidden layer had 30 input neurons and 20 output neurons. The output layer had 20 input neurons and 10 output neurons. Other parameters of the model (learning rate, activation function, etc.) were not changed.

Double channel CNN: For this model, the architecture of the original model was not changed. However, the output channel of the first convolutional layer was doubled to be 12. Resultingly,the input channel of the second convolutional layer had to be set to 12 as well. Furthermore, the output channel of the second convolutional layer was doubled to be 32. To accommodate this change, the number of input neurons of the first fully connected layer was set to be 32*5*5. Other parameters of the model were not changed.

1 convolutional layer CNN: For this model, the second convolutional layer was removed from the original model. However, this changed the dimension of the input to the fully connected layer to 14*14 for each output channel of the CNN. So, the number of input neurons of the first fully connected layer was set to be 6*14*14. Other parameters of the model were not changed.

3 convolutional layer CNN: For this model, another convolutional and max pooling layer was added between the second convolutional layer and the first fully connected layer of the original model. The added convolutional layer had 16 input channels and 26 output channels while the max pooling layer had a kernel size of 2. However, in this setup, the dimension of the input to the first fully connected layer was too low. So, the kernel size of each convolutional layer was decreased to be 3. This meant that the input to the fully connected layer was now 2*2 for each output channel of the previous convolutional layer. So, the input size of the first fully connected layer was modified to be 26*2*2. Other parameters of the model were not changed.

Leaky ReLU CNN: For this model, the original model architecture was kept the same. However, the activation function for the convolutional and fully connected layers was set to be the leaky ReLU function with a negative slope of 0.01 instead of the ReLU function used in the original model. Other parameters of the model were not changed.

Tanh CNN: For this model, the original model architecture was kept the same. However, the activation function for the convolutional and fully connected layers was set to be the tanh function instead of the ReLU function used in the original model. Other parameters of the model were not changed.

Less Learning Rate CNN: For this model, the original model architecture was kept the same. However, the learning rate used during training was set to be 0.0001 instead of the 0.001 value used in training the original model. Other parameters of the model were not changed.

More Learning Rate CNN: For this model, the original model architecture was kept the same. However, the learning rate used during training was set to be 0.01 instead of the 0.001 value used in training the original model. Other parameters of the model were not changed.

In addition to these models, the original model depicted in the tutorial was tested on training and testing data to get our baseline (since the original model already uses the ReLU activation, there is no need to create another model for the ReLU activation as prompted by the homework. Instead of creating another model with the same parameters for ReLU, the original model was used to get the performance of such a network).

Now, let us see how each model performs when compared to the original model in terms accuracy and loss on the train and test sets. As can be seen from Table 2, the best result on the training set and test set in terms of accuracy is obtained when the architecture of the original model is kept but the learning rate is increased. This is expected since the model is trained for only 10 epochs. However, this does not imply that a higher learning rate will always give better results, since having a high learning rate could easily make the model miss the optimum set of weights.

Table 2: The performance of different models on the training and test sets

Model Type	**Train Loss**	**Train Acc**	**Test Loss**	**Test Acc**
Original CNN	1.08	61.69%	1.16	58.58%
2 layer neural network	1.18	59.06%	1.36	51.95%
3 layer neural network	1.15	59.71%	1.35	52.6%
4 layer neural network	1.18	58.39%	1.37	51.69%
5 layer neural network	1.22	56.32%	1.39	50.67%
Double channel CNN	0.93	67.43%	1.03	64.09%
1 conv layer CNN	1.02	63.78%	1.21	56.9%
3 conv layer CNN	1.27	54.7%	1.29	53.7%
Leaky ReLU CNN	1.15	58.9%	1.21	56.7%
Tanh CNN	1.11	60.2%	1.16	58.96%
Less learning rate CNN	1.88	31.2%	1.88	31.7%
More learning rate CNN	0.77	72.5%	1.19	61.3%

2.4 Conclusion

In conclusion, we managed to train and test the model in the tutorial using PyTorch and the CIFAR-10 dataset and create and train alternate models to compare the results with the original model. Since our code uses GPU to train and test the model, there were not significant delays caused by model training and evaluation. However, loading the dataset took some time.

Reference

Sharma, Sagar. “Activation Functions in Neural Networks.” Medium, Towards Data Science, 4 July 2021, https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6.

Gour, Rinu. “Artificial Neural Network for Machine Learning-Structure amp; Layers.” Medium, Medium, 15 Feb. 2019, https://medium.com/@rinu.gour123/artificial-neural-network-for-machine-learning-structure-layers-2a275f73f473.

Mishra, Mayank. “Convolutional Neural Networks, Explained.” Medium, Towards Data Science, 2 Sept. 2020, https://towardsdatascience.com/convolutional-neural-networks-explained-9cc5188c4939.

"CNN: Introduction to Pooling Layer." GeeksforGeeks, 29 July 2021, https://www.geeksforgeeks.org/cnn-introduction-to-pooling-layer/.