CNN Implementation

– Classification on CIFAR-10 Data Set

Abstract

The task of the experiment is to use PyTorch to implement multi-layer perceptron (MLP) and convolutional network (ConvNet) respectively, and complete classification on CIFAR10 dataset images.

In the parameter tuning stage, we look into how kernel size, layer amount, optimizers, pooling methods etc. can affect the training results. The best CNN model chosen based on the performances on the validation set is compared with MLP and ResNet. Due to the time limit, we do not have time to go through parameter tuning on MLP and ResNet, the same as on CNN.

The best performance with 3 convolutional layers we constructed reaches 72.79% accuracy on the test set. The hyperparameters at this occasion are: dropout rate 0.1, learning rate 0.01, momentum 0.9, batch size 50, max epoch 10 with max pooling. The MLP reaches the highest accuracy 52.83% on the test set. As for the ResNet we test, the prediction accuracy is not ideal(seen from the above diagram).

1 Introduction

The CIFAR-10 dataset (Canadian Institute For Advanced Research) is a collection of images that are commonly used to train machine learning and computer vision algorithms. It is one of the most widely used datasets for machine learning research. The CIFAR-10 dataset contains 60,000 32x32 color images in 10 different classes[1]. The classes are labeled as integers ranging from 0-9. They are: 'plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck'.

The task is to implement CNN for classifying these images. Parameter tuning and analysis happen on the train set and the validation set, and the selected optimal model runs predictions on the test set for experiment results. Loss and accuracy scores are used to measure the performance of a model.

This assignment provides no code framework. Therefore, we need to understand the Pytorch module basis and construct custom MLP and CNN.

2 Programming Modules

Pytorch is used for constructing neural networks. It is a machine learning framework based on the Torch library,[4][5][6] used for applications such as computer vision and natural language processing[2]. Use of Numpy in this experiment is limited, since all data is converted to Pytorch tensor for processing. For visualizing data and experiment results, matplotlib is invoked.

3 Data Preprocessing

3.1 Load CIFAR-10

There is a package called ‘torchvision’, which specifically has data loaders for common datasets such as CIFAR10[3]. This provides a huge convenience and avoids writing boilerplate code.

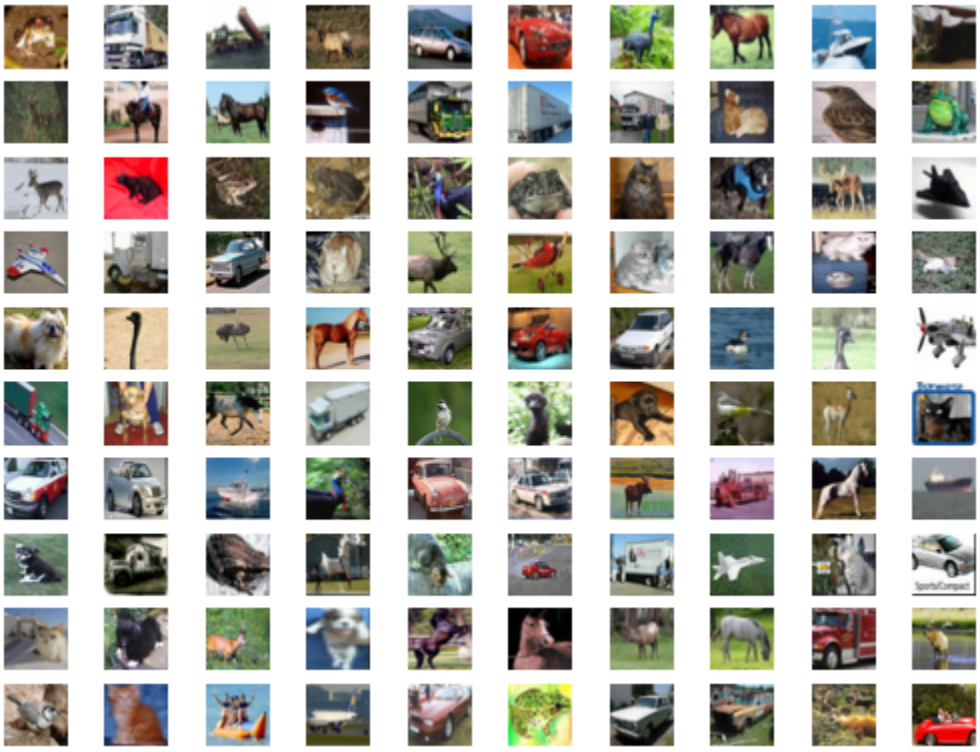
```
# convert data to a normalized torch.FloatTensor
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
    ])
```

One thing needed to mention is that the loaded data is converted to pytorch tensor and normalized(regularized) for later processing and learning.

3.2 Display Images from Dataset

We will display 100 images from the train set, just to see if data is loaded properly.

As can seen from the below image, the result is blurry. However, we are still doing the correct read-in, since the original images are of 32 by 32 resolution, they are meant to be blurred as such.



3.3 Prepare Validation Set

We will randomly take 20% of the samples in the train set to form the validation set. Parameter tuning and model choosing will be based on the validation set. The splitting of the train set and the validation set

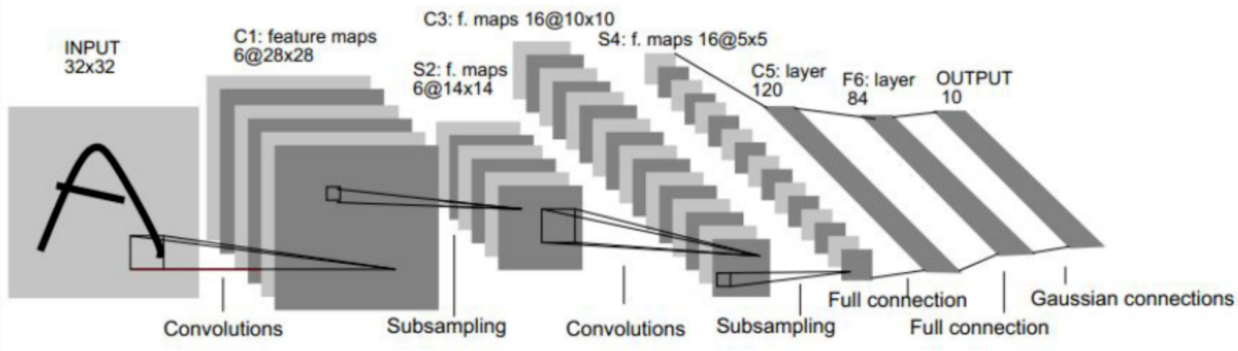
uses ‘SubsetRandomSampler’ from the torch module, inspired by an online article ‘PyTorch [Basics] — Sampling Samplers’[4].

4 Convolutional Neural Network

CNN's are a class of Deep Learning Algorithms that can recognize and classify particular features from images and are widely used for analyzing visual images. A CNN architecture is usually consists of two parts:

1. A convolution tool that separates and identifies the various features of the image for analysis in a process called Feature Extraction.
2. A fully connected layer that utilizes the output from the convolution and predicts based on the features extracted in previous stages.

Below image describes a classic LeNet-5 architecture for CNN, which is the main CNN we use for parameter tuning in this assignment.



One important thing to manually construct a CNN is that we need to calculate the output size of each layer correctly, so that this could be used as in_channels and input sizes for later convolutional layers and fully connected layers.

To compute the output size of a given convolutional layer we can perform the following calculation: We can compute the spatial size of the output volume as a function of the input volume size (W), the kernel/filter size (F), the stride with which they are applied (S), and the amount of zero padding used (P) on the border. The correct formula for calculating how many neurons define the output_W is given by (W−F+2P)/S+1[5].

After implementing the CNN, we go into the parameter tuning stage, to analyze how hyperparameters affect the learning results of CNN.

4.1 Convolutional Layer Amount and Kernel Size

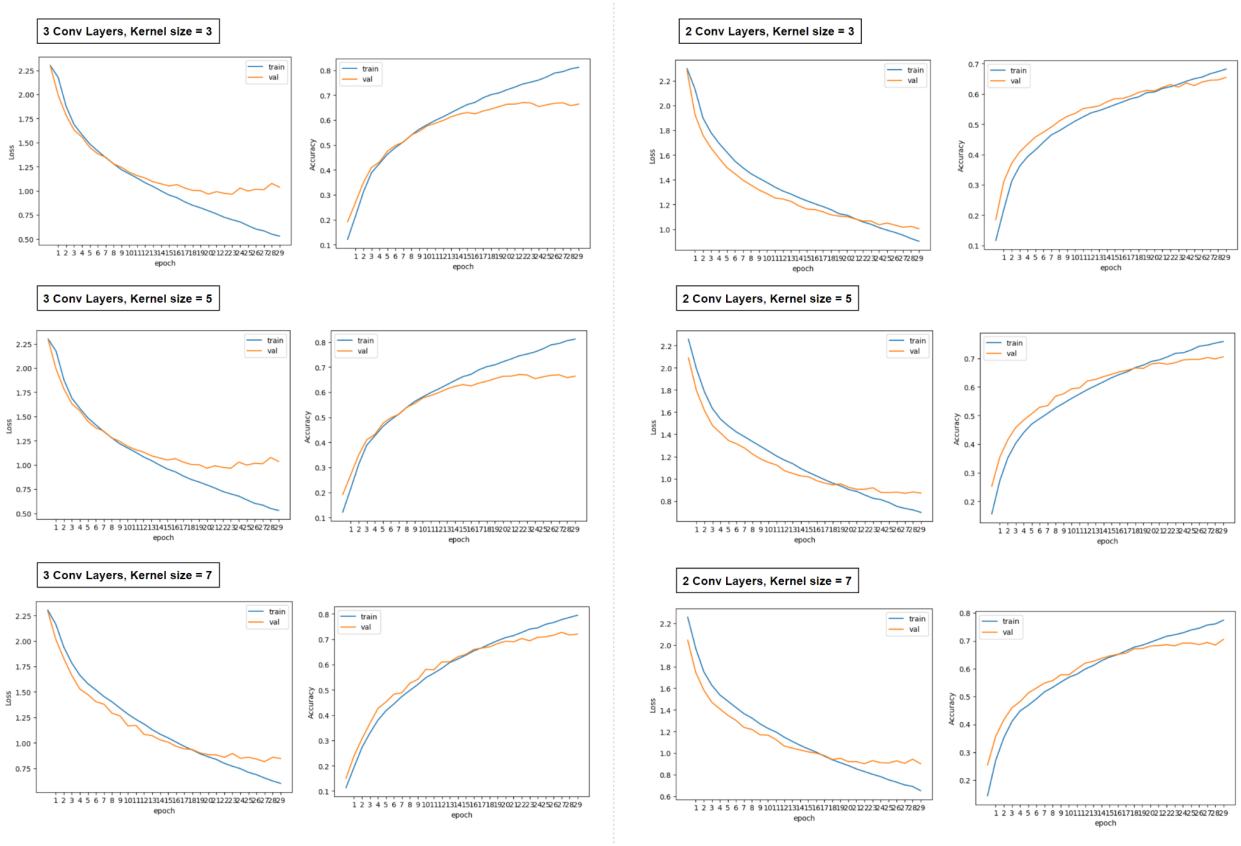
We use batch size 200, epoch 30, lr 0.01 for configuring the optimizer and solver. Nestrove for SGD is off for now, so the momentum will not be used. (3x3), (5x5) and (7x7) kernel sizes are tested for CNN with 2

convolutional layers and 3 convolutional layers. Pooling in this section is using average polling. And the activation function is ReLU.

As can seen from the image, prediction accuracy scores on the validation set have mostly fallen into the 0.6-0.7 range, which is not ideal learning at all. 3 convolutional layers are generally better than 2 convolutional layers.

The best performances for 3 convolutional layers are with (7x7) kernel size, and with 2 convolutional layers, kernel size (5x5) and (7x7) have similar results, but (5x5) is slightly better.

Due to the accuracy being not ideal, we will need further parameter tuning. The activation functions, learning rate, momentum, batch size and epoch will be tested first.



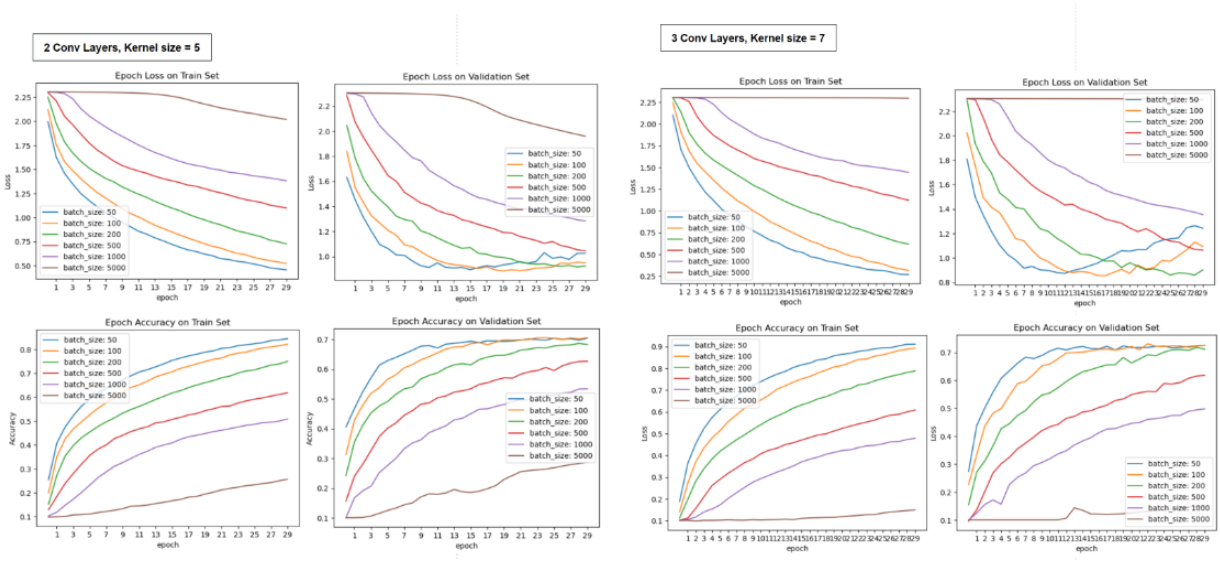
Here adds some research and analysis on how to choose proper kernel size. Convolutional neural networks work on 2 assumptions -

1. Low level features are local.
2. What's useful in one place will also be useful in other places.

Kernel size should be determined by how strongly we believe in those assumptions for the problem at hand. In short, there is no direct connection between kernel size and prediction accuracy, and there is no specific way to choose kernels' dimensions or sizes.

Larger kernel sizes seem to have better learning results in our cases(higher accuracy, lower loss, smaller differences between train set and validation set). One way to interpret this would be that our model needs higher complexity to address the classification rules, since increasing kernel size usually effectively increases computational complexity and the total number of parameters in the network[6].

4.2 Batch Size



For both CNN networks, the larger the batch size, the slower the model converges, and the lower the prediction accuracy is. Both models have best performances when the batch size is 50. The CNN with 3 convolutional layers is better than with 2 convolutional layers, with higher accuracy on the validation set and lower minimum loss.

When the batch size is set to 50, the model starts to get overfitting at round epoch 12. Loss on the validation starts to rise and keeps falling on the train set. This is because the model tries to memorize the noises and features that are irrelevant or make confusion on the classification rules. This overfitting happens with smaller batch sizes: 50 and 100. Small batch size with large epochs can cause frequent local oscillations.

For large batch sizes(1000 and 5000), especially when the batch size is set to 5000, the loss curve becomes flat, this is a typical underfitting.

We will use: 3 convolution layers, 7x7 kernel size, batch size 50 and epoch 20 for later testing. In the next section, we will compare the differences momentum values going to make on the learning results.

4.3 Momentum

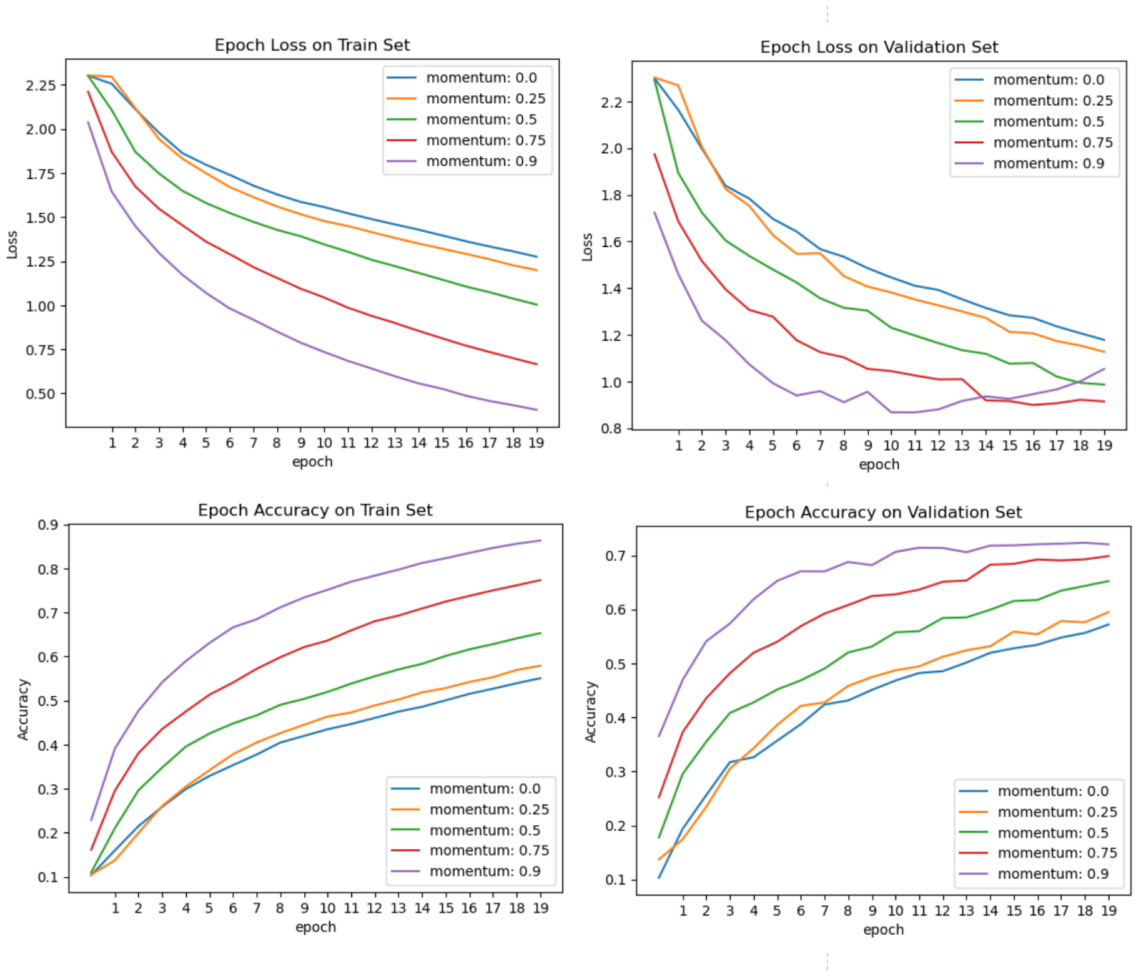
Given the hyperparameter combination above, we will evaluate how momentum values affect the learning results.

When the momentum is 0.0, this means we will not introduce any momentum to our optimization calculation, the network carries out the worst behavior.

The performance of the network gets better as the momentum value grows. Momentum in neural nets helps them get out of local minima points so that a more important global minimum is found.

High momentum values also result in faster convergence, since the nature of the momentum is to help accelerate gradient vectors in the right directions. Momentum accumulates the gradient of the past steps to determine the direction to go[7].

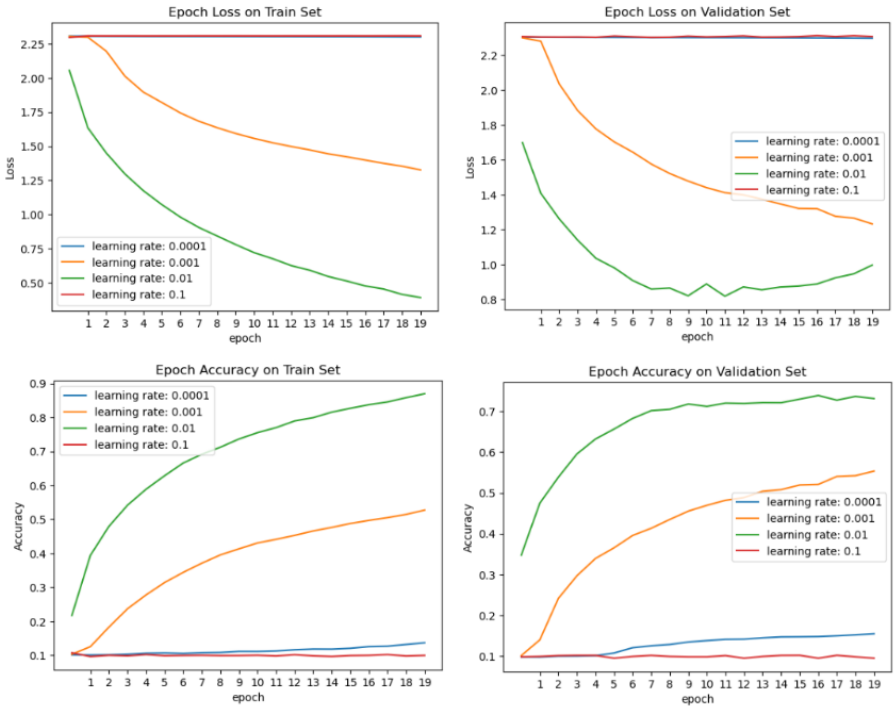
The best result is with momentum 0.9, and we can see the model starts overfitting around epoch 13. In the later section, we will still maintain the max epoch to 20, since learning rate and drop out factors can also have an impact over overfitting/underfitting, and how the model converges.



4.4 Learning Rate

A learning rate that is too large can cause the model to converge too quickly to a suboptimal solution, whereas a learning rate that is too small can cause the process to get stuck[8]. This is the reason why CNN networks behave badly with learning rate 0.1 and 0.0001.

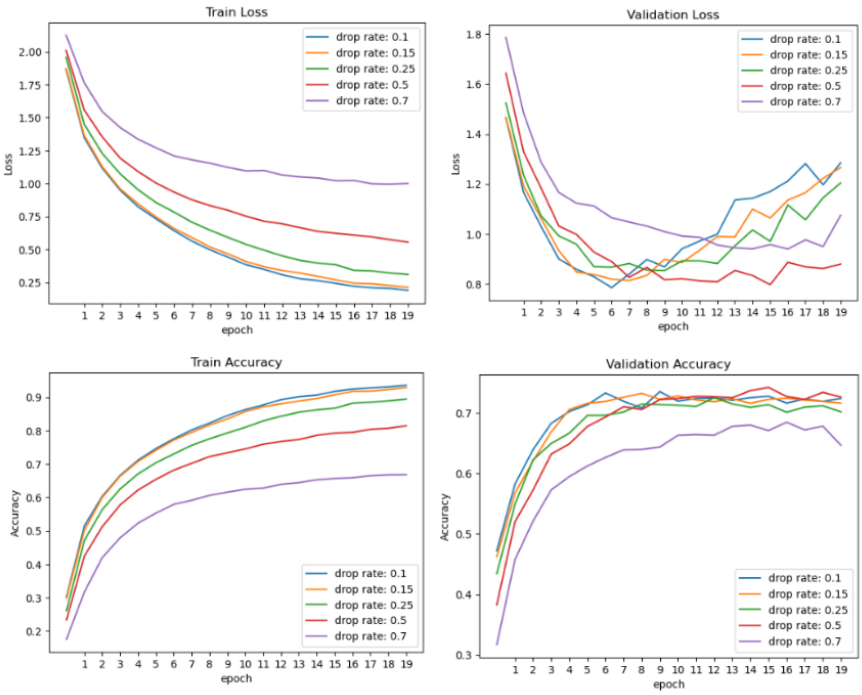
Among 0.01 and 0.001, the network has lower loss, higher accuracy and faster convergence when the learning rate is greater. A more cautious way of selecting a proper learning rate would be to have smaller steps in between this range, but due to the time limit, we will settle with 0.01 here.



4.5 Drop Rate

A learning rate that is too large can cause the model to converge too quickly to a suboptimal solution, whereas a learning rate that is too small can cause the process to get stuck[8]. This is the reason why CNN networks behave badly with learning rate 0.1 and 0.0001.

Among 0.01 and 0.001, the network has lower loss, higher accuracy and faster convergence when the learning rate is greater. A more cautious way of selecting a proper learning rate would be to have smaller steps in between this range, but due to the time limit, we will settle with 0.01 here.

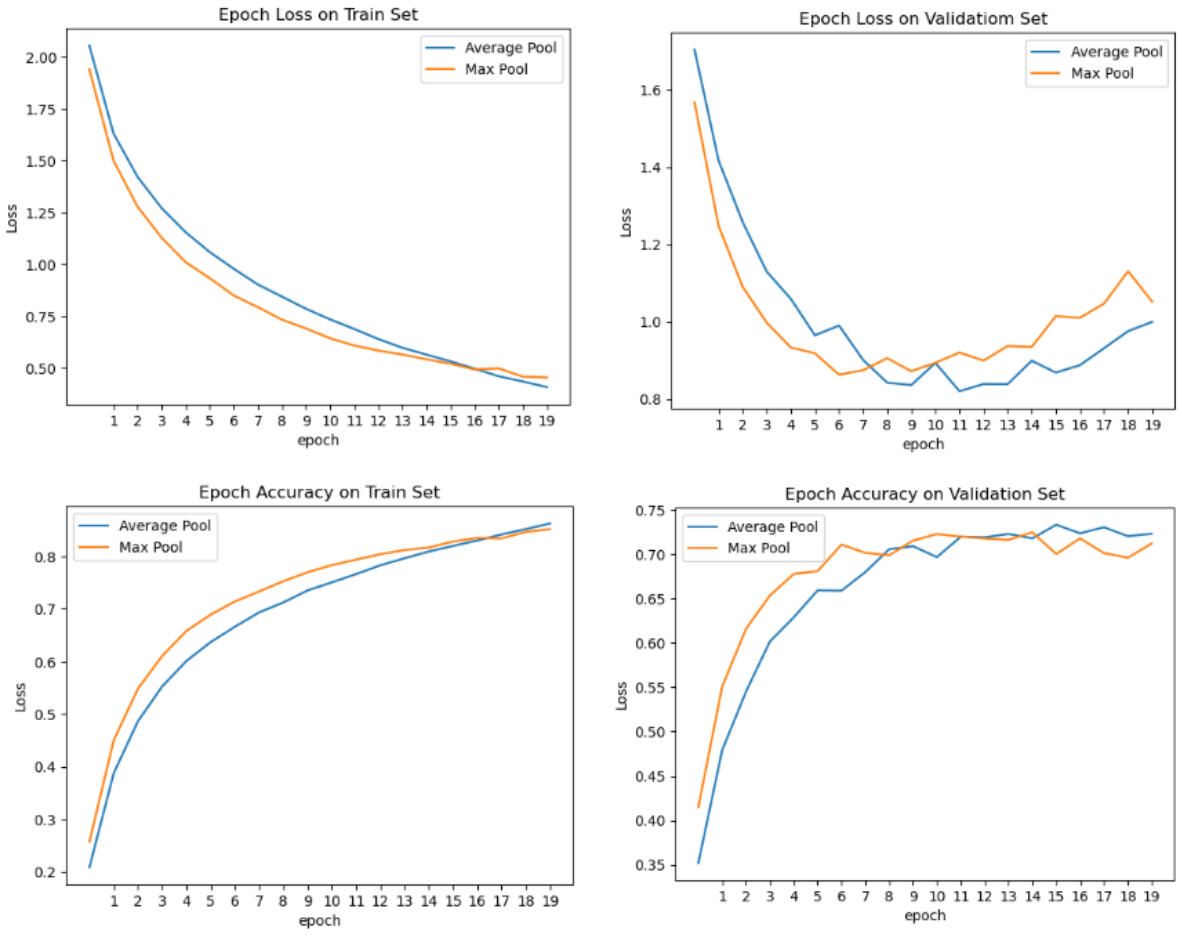


4.6 Max Pooling vs Average Pooling

Pooling layers are used to reduce the dimensions of the feature maps. Thus, it reduces the number of parameters to learn and the amount of computation performed in the network. The pooling layer summarizes the features present in a region of the feature map generated by a convolution layer.

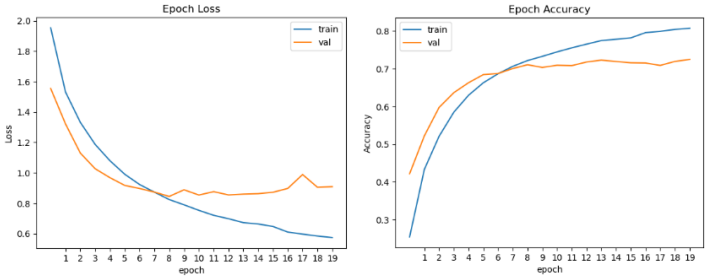
This has the effect of preserving the most salient features in each pooling region, while discarding less relevant information. Max pooling is often used in CNNs for object recognition tasks, as it helps to identify the most distinctive features of an object, such as its edges and corners. Whilst average pooling is on the contrary, it preserves more information than max pooling, but may also dilute the most salient features[9].

In our case, since max pooling works generally better, we could make an assumption that the input data has lots of fluctuations and with the help of max pooling, the model becomes more resilient to these fluctuations.



4.7 Kernel size changed based on layers

In the previous examples, we stabilized the kernel size to 7x7 after Section 4.1 and 4.2. In this section, we will use different kernel sizes (from large to small) on the three convolutional layers. The network produces similar results, but better than the network with all the same kernel sizes. The overfitting and fluctuations on the late epochs seem to be reduced also.



4.8 Section Conclusion and Model Chosen

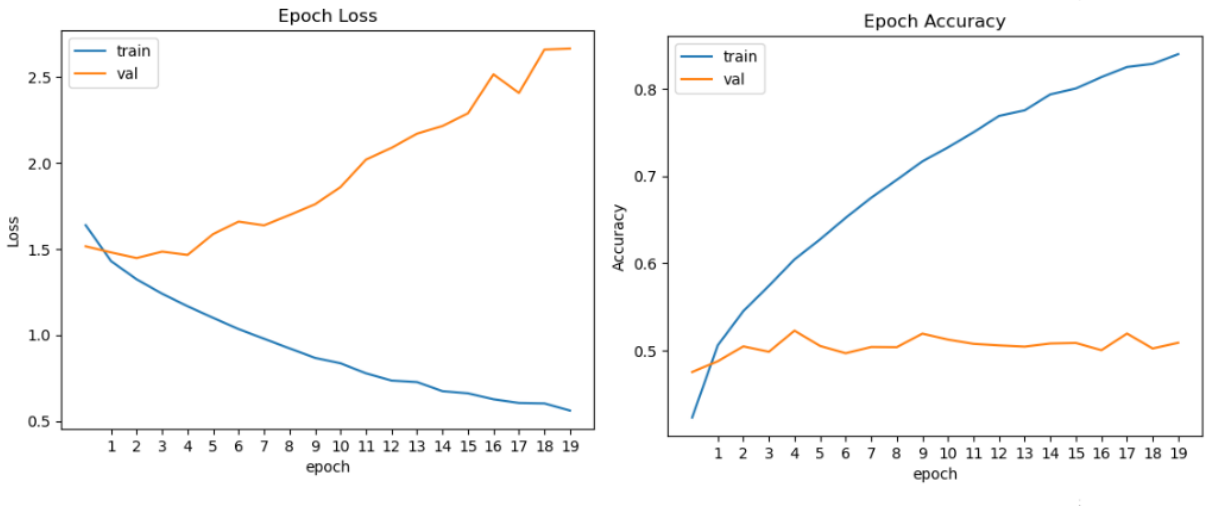
Except for the parameters tested in the above section, we also run tests on different activation functions and optimizers. The diagrams are lost, but ReLU activation and SGD optimizer with learning rate 0.01 and momentum 0.9 give out the best performances under the parameter combination we have selected.

```
class CNN_Net(nn.Module):
    def __init__(self, activation='relu', drop_out=.5):
        super(CNN_Net, self).__init__()

        self.activation = activation

        # initialize/register convolutional layers and pooling layers

        # I: 3*32*32 O: 16*32*32
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=16, kernel_size=7, stride=1, padding=3)
        # I: 16*32*32 -> Pool -> 16*16*16 O: 32*16*16
        self.conv2 = nn.Conv2d(in_channels=16, out_channels=32, kernel_size=5, stride=1, padding=2)
        # I: 32*16*16 -> Pool -> 32*8*8 O: 64*8*8 -> final pool: 64*4*4
        self.conv3 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, stride=1, padding=1)

        self.pool1 = nn.MaxPool2d(2, 2)  # there is also max pooling

        # full connected layers
        self.fc1 = nn.Linear(64*4*4, 512)
        self.fc2 = nn.Linear(512, 64)
        self.fc3 = nn.Linear(64, 10)

        # dropout to avoid overfitting
        self.dropout = nn.Dropout(p=drop_out)

    def forward(self, x):
        if self.activation == 'relu':
            x = self.pool1(F.relu(self.conv1(x)))
            x = self.pool1(F.relu(self.conv2(x)))
            x = self.pool1(F.relu(self.conv3(x)))
            # after convolution, the train data will have batch size rows and 16*5*5 features
            x = x.view(-1, 64*4*4)
            x = self.dropout(F.relu(self.fc1(x)))
            x = self.dropout(F.relu(self.fc2(x)))
            x = self.fc3(x)
        if self.activation == 'sigmoid':
            x = self.pool1(F.sigmoid(self.conv1(x)))
            x = self.pool1(F.sigmoid(self.conv2(x)))
            x = self.pool1(F.sigmoid(self.conv3(x)))
            # after convolution, the train data will have batch size rows and 16*5*5 features
            x = x.view(-1, 64*4*4)
            x = self.dropout(F.sigmoid(self.fc1(x)))
            x = self.dropout(F.sigmoid(self.fc2(x)))
            x = self.fc3(x)
        return x
```

Below is how our CNN finally looks like. We will use batch size 50 and max epoch 10 for conclusion making.

5 Multilayer Perceptron

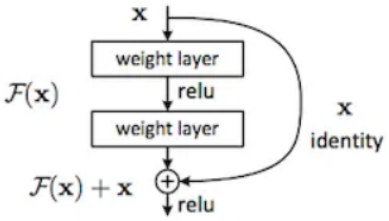
```
# for full connect, we need to flatten the image data to 1d array

def flatten(x):
    N = x.shape[0] # read in N, C, H, W
    return x.view(N, -1)

class Flatten(nn.Module):
    def forward(self, x):
        N = x.shape[0] # read in N, C, H, W
        return x.view(N, -1)

fc_model = nn.Sequential(
    Flatten(),
    nn.Linear(3*32*32, 1536),
    nn.ReLU(),
    nn.Linear(1536, 128),
    nn.ReLU(),
    nn.Linear(128, 64),
    nn.ReLU(),
    nn.Linear(64, 10),
)
```



We use a simple 3 layer MLP for comparison with the CNN network. As can seen from the diagram, the loss on the validation set starts to grow at quite an early epoch(4). The highest accuracy score on the validation set is only around 0.5. MLP does not have an ideal performance on CIFAR10 image classification.

One potential reason is that, although both MLP and CNN can be used for Image classification, MLP takes vector as input and CNN takes tensor as input so CNN can understand spatial relation(relation between nearby pixels of image)between pixels of images better thus for complicated images CNN will perform better than MLP[10].

5 ResNet



As the network gets deeper, instead of steadily decreasing on loss, after attaining a minimum value, the error rate starts increasing again. This happens due to the exploding and vanishing gradient descent problem which also causes overfitting of the model, hence increasing the error.Residual Networks have proved to be quite efficient in solving this problem, as they use a a skip connection or a "shortcut" between every two layers along with using direct connections among all the layers. This allows us to take activation from one layer and feed it to another layer, even much deeper in the neural network, hence sustaining the learning parameters of the network in deeper layers[11].

The whole ResNet code is inspired and learnt from: 'Pytorch ResNet implementation from Scratch'[11]. Since the time budget for this assignment is quite limited, we do not have time for parameter tuning and run tests regarding the ResNet algorithm, and the prediction result is not ideal.

6 Conclusion

The task of the experiment is to use PyTorch to implement multi-layer perceptron (MLP) and convolutional network (ConvNet) respectively, and complete classification on CIFAR10 dataset images.

In the parameter tuning stage, we look into how kernel size, layer amount, optimizers, pooling methods etc. can affect the training results. The best CNN model chosen based on the performances on the validation set is compared with MLP and ResNet. Due to the time limit, we do not have time to go through parameter tuning on MLP and ResNet, the same as on CNN.

The best performance with 3 convolutional layers we constructed reaches 72.79% accuracy on the test set. The hyperparameters at this occasion are: dropout rate 0.1, learning rate 0.01, momentum 0.9, batch size 50, max epoch 10 with max pooling. The MLP reaches the highest accuracy 52.83% on the test set. As for the ResNet we test, the prediction accuracy is not ideal(seen from the above diagram). Among all these

	train loss	train acc	val loss	val acc	test acc
MLP (3 hidden layers) Batch Size: 50 Max Epoch: 20	0.8532	0.6974	1.6538	0.5234	0.5283
CNN(3 Convolutional Layers) Batch size: 50 Dropout Rate 0.1 Max Epoch 10 Momentum 0.9 Learning Rate 0.01 Max Pooling	1.0665	0.6582	0.9573	0.6681	0.7279
ResNet50 Batch size 50, Learning Rate 0.001, Momentum 0.9 Max Epoch 10	1.0496	0.6423	1.4683	0.5834	0.6035
ResNet50 Batch size 100, Learning Rate 0.001, Momentum 0.9 Max Epoch 30	0.4125	0.8553	1.70052	0.6008	0.6343

tracked models, the 3-convolutional-layers CNN has the best learning results. However, ResNet by theory should be better than classic LeNet-5, since ResNet is known as one of the most efficient Neural Network Architectures, as they help in maintaining a low error rate much deeper in the network[12]. Our experiment has such conflicting results due to the fact that we have not found the proper parameter settings within the limited tests we run for ResNet.

To improve the learning, we could test more on the ResNet model and try to find out the hyperparameter that leads to a global minimum convergence. Besides working on the network parameters only, we should also consider data preprocessing on the images, such as image flipping, contrast enhancing, brightness adjustment and so on.

At the time this assignment is handed in, experiments and tests are still running. Due to the limitation of time, although not being able to complete all planned experiments and achieve ideal prediction accuracy, we will keep working on this afterwards on our own for better understanding over the convolutional neural network.

.

7 References

[1] The CIFAR-10 dataset https://www.cs.toronto.edu/~kriz/cifar.html

[2] PyTorch https://en.wikipedia.org/wiki/PyTorch

[3] TRAINING A CLASSIFIER https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html

[4] PyTorch [Basics] — Sampling Samplers
https://towardsdatascience.com/pytorch-basics-sampling-samplers-2a0f29f0bf2a

[5] Convolutional Neural Nets
https://medium.com/@priyankapatel2205/convolutional-neural-nets-1813eee0510

[6] How to choose the optimal kernel size?
https://www.devron.ai/kbase/how-to-choose-the-optimal-kernel-size#:~:text=Increasing%20kernel%20size%20effectively%20increases,choosing%20the%20optimal%20kernel%20size

[7] What is momentum in neural network?
https://datascience.stackexchange.com/questions/84167/what-is-momentum-in-neural-network

[8] How to use Learning Curves to Diagnose Machine Learning Model Performance
https://machinelearningmastery.com/learning-curves-for-diagnosing-machine-learning-model-performance/

[9] CNN | Introduction to Pooling Layer https://www.geeksforgeeks.org/cnn-introduction-to-pooling-layer/

[10] MLP vs CNN vs RNN Deep Learning, Machine Learning Model
https://www.linkedin.com/pulse/mlp-vs-cnn-rnn-deep-learning-machine-model-momen-negm/

[11] Pytorch ResNet implementation from Scratch https://www.youtube.com/watch?v=DkNIBBBvcPs

[12] ResNets: Why Do They Perform Better than Classic ConvNets? (Conceptual Analysis) https://towardsdatascience.com/resnets-why-do-they-perform-better-than-classic-convnets-conceptual-analysis-6a9c82e06e53