Image Classification on CIFAR-10 with Optimized CNN

Shaolong Li A16159625
Zikang Chen A15941751

COGS 181 Final Project Report
24 March 2023

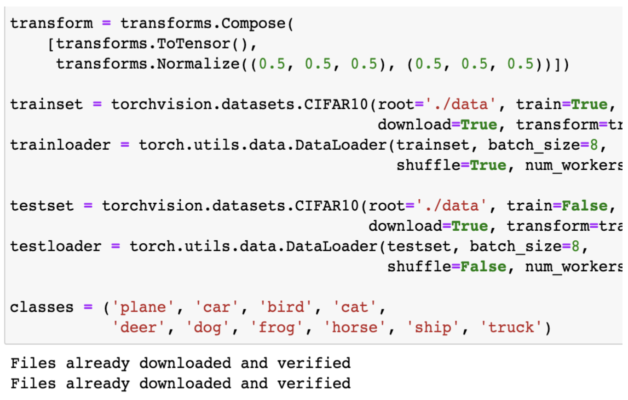
Abstract

Convolutional neural networks (CNN) have attracted a tremendous amount of research attention, as they have been successfully used in a wide range of image recognition and classification tasks. In this paper, we try to classify images in the CIFAR-10 dataset using a CNN and investigate the effectiveness of various optimization techniques. By improving the accuracy of CNN, we were able to find the network design and optimization methods that are most suitable and effective for image classification on the CIFAR-10 dataset. We also trained a residual neural network (RNN) and the AlexNet on the dataset, and we found RNN outperformed other models. The findings suggest that CNNs can be a powerful tool for image classification tasks.

1 Introduction

We aimed to classify the images in the CIFAR-10 dataset using a convolutional neural network (CNN). The dataset contains 60000 32-by-32 images of 10 classes, with 50000 training images and 10000 test images. It is further divided into 5 training sets and 1 test set, where each test set has 1000 random images from each class, and each training set contains the remaining images in random order. CNNs have a wide range of applications in image recognition, image classification, and video analysis. A CNN is a deep neural network that takes in an image as input, assigns weights to different aspects of the image, and classifies

cat truck dog frog



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

trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True, transform=tr
trainloader = torch.utils.data.DataLoader(trainset, batch_size=8,
                                          shuffle=True, num_workers

testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                       download=True, transform=tra
testloader = torch.utils.data.DataLoader(testset, batch_size=8,
                                         shuffle=False, num_workers

classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
```

```
Files already downloaded and verified
Files already downloaded and verified
```

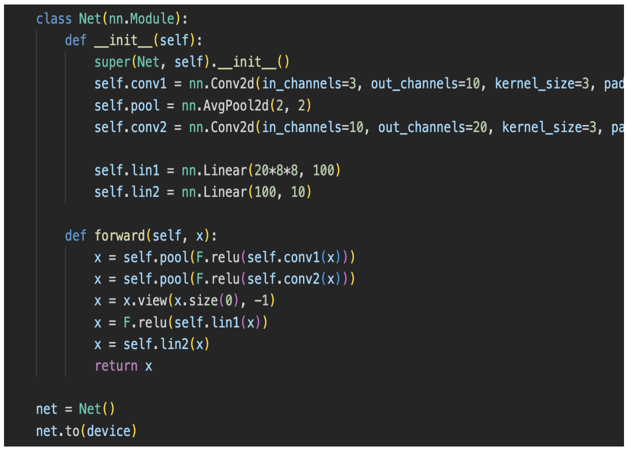
the image into a specific class. In order to optimize the performance, we implemented CNN with different activation and pooling functions, as well as optimization, dropout, and normalization methods. For each modification, we tuned the hyperparameters of each function to improve the performance. We also tried adding more layers to the CNN. Besides, we trained ResNet and AlexNet and compared their performance.

2 Methods

We first implemented a baseline CNN to serve as a comparison model. We then attempted to improve the performance of CNN on the dataset by using different activation functions, pooling functions, and optimizers. We also tried to add more layers to CNN. In addition, we experimented with batch normalization and dropout. Last but not least, we extensively tuned the hyperparameters, including the learning rate, number of input and output channels, batch size, number of epochs, and parameters of the activation functions. In addition to the typical CNN, we experimented with ResNet and AlexNet to see their performance compared with our model.

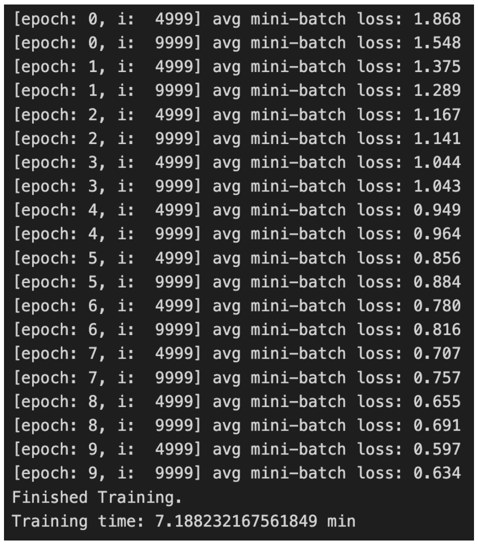
3 Experiment

In our baseline model, we implemented the CNN with three blocks, including two convolutional blocks and one linear block, the same as the model provided in hw4. Each convolutional block contains a 3-by-3 convolutional layer with a padding size of 1, a ReLU activation layer, and a 2-by-2 average pooling layer. The last block contains a flattened function, a linear layer, and a ReLU activation layer, followed by another linear layer. We chose cross-entropy as the loss function, and we chose stochastic gradient descent (SGD) with a learning rate of 0.001 and a momentum of 0.9 as the optimizer. By training the CNN with 10 epochs and a batch size of 4, this model produces an accuracy of 65% on the test set, which will be used as a baseline for later comparison.

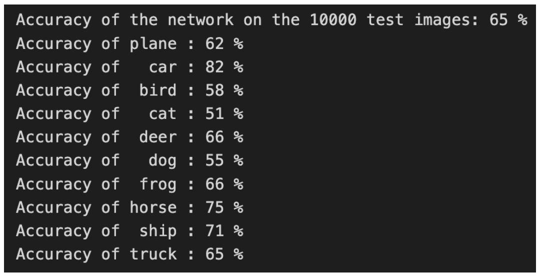


```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=10, kernel_size=3, pad
        self.pool = nn.AvgPool2d(2, 2)
        self.conv2 = nn.Conv2d(in_channels=10, out_channels=20, kernel_size=3, pa

        self.lin1 = nn.Linear(20*8*8, 100)
        self.lin2 = nn.Linear(100, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(x.size(0), -1)
        x = F.relu(self.lin1(x))
        x = self.lin2(x)
        return x

net = Net()
net.to(device)
```

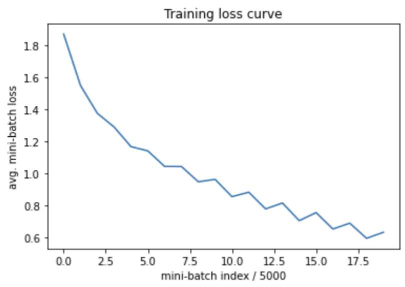
Source: above code was from HW4



```
[epoch: 0, i:  4999] avg mini-batch loss: 1.868
[epoch: 0, i:  9999] avg mini-batch loss: 1.548
[epoch: 1, i:  4999] avg mini-batch loss: 1.375
[epoch: 1, i:  9999] avg mini-batch loss: 1.289
[epoch: 2, i:  4999] avg mini-batch loss: 1.167
[epoch: 2, i:  9999] avg mini-batch loss: 1.141
[epoch: 3, i:  4999] avg mini-batch loss: 1.044
[epoch: 3, i:  9999] avg mini-batch loss: 1.043
[epoch: 4, i:  4999] avg mini-batch loss: 0.949
[epoch: 4, i:  9999] avg mini-batch loss: 0.964
[epoch: 5, i:  4999] avg mini-batch loss: 0.856
[epoch: 5, i:  9999] avg mini-batch loss: 0.884
[epoch: 6, i:  4999] avg mini-batch loss: 0.780
[epoch: 6, i:  9999] avg mini-batch loss: 0.816
[epoch: 7, i:  4999] avg mini-batch loss: 0.707
[epoch: 7, i:  9999] avg mini-batch loss: 0.757
[epoch: 8, i:  4999] avg mini-batch loss: 0.655
[epoch: 8, i:  9999] avg mini-batch loss: 0.691
[epoch: 9, i:  4999] avg mini-batch loss: 0.597
[epoch: 9, i:  9999] avg mini-batch loss: 0.634
Finished Training.
Training time: 7.188232167561849 min
```

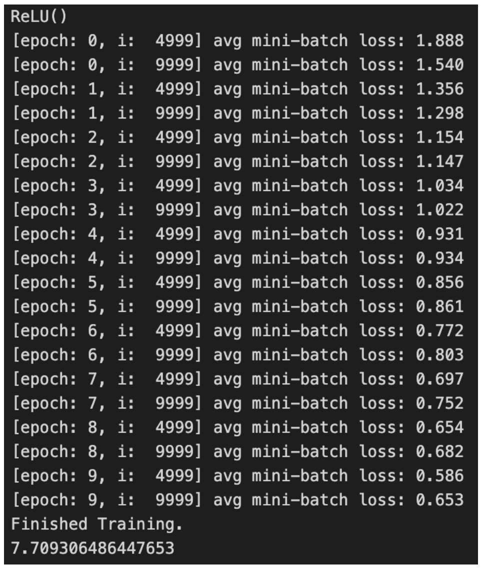


```
Accuracy of the network on the 10000 test images: 65 %
Accuracy of plane : 62 %
Accuracy of   car : 82 %
Accuracy of  bird : 58 %
Accuracy of   cat : 51 %
Accuracy of  deer : 66 %
Accuracy of   dog : 55 %
Accuracy of  frog : 66 %
Accuracy of horse : 75 %
Accuracy of  ship : 71 %
Accuracy of truck : 65 %
```

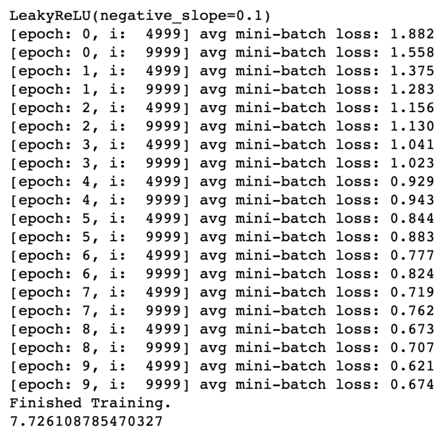


3.1 Activation Functions

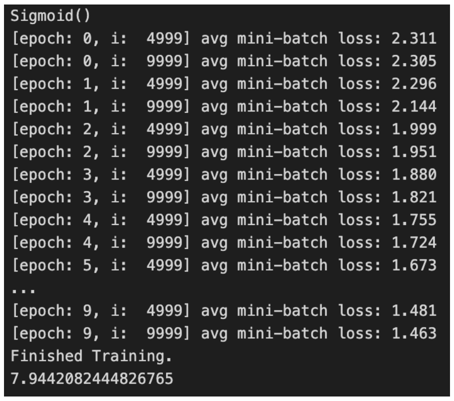
We tried using different activation functions including ReLU, LeakyReLU, Sigmoid, Tanh, and ELU. We then ran our model on these five activation functions each with the default parameter and the test accuracy for each one is as follows: ReLU: 64%, LeakyReLU: 64%, Sigmoid: 48%, Tanh: 59%, and ELU: 62%.



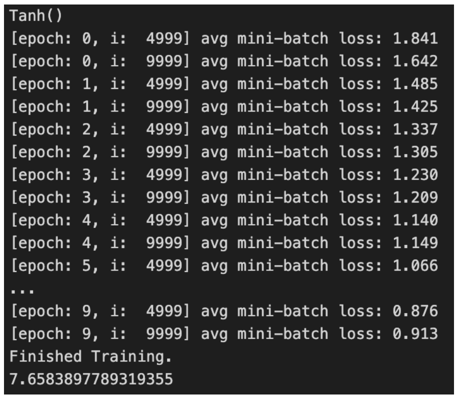
```
ReLU()
[epoch: 0, i:  4999] avg mini-batch loss: 1.888
[epoch: 0, i:  9999] avg mini-batch loss: 1.540
[epoch: 1, i:  4999] avg mini-batch loss: 1.356
[epoch: 1, i:  9999] avg mini-batch loss: 1.298
[epoch: 2, i:  4999] avg mini-batch loss: 1.154
[epoch: 2, i:  9999] avg mini-batch loss: 1.147
[epoch: 3, i:  4999] avg mini-batch loss: 1.034
[epoch: 3, i:  9999] avg mini-batch loss: 1.022
[epoch: 4, i:  4999] avg mini-batch loss: 0.931
[epoch: 4, i:  9999] avg mini-batch loss: 0.934
[epoch: 5, i:  4999] avg mini-batch loss: 0.856
[epoch: 5, i:  9999] avg mini-batch loss: 0.861
[epoch: 6, i:  4999] avg mini-batch loss: 0.772
[epoch: 6, i:  9999] avg mini-batch loss: 0.803
[epoch: 7, i:  4999] avg mini-batch loss: 0.697
[epoch: 7, i:  9999] avg mini-batch loss: 0.752
[epoch: 8, i:  4999] avg mini-batch loss: 0.654
[epoch: 8, i:  9999] avg mini-batch loss: 0.682
[epoch: 9, i:  4999] avg mini-batch loss: 0.586
[epoch: 9, i:  9999] avg mini-batch loss: 0.653
Finished Training.
7.709306486447653
```



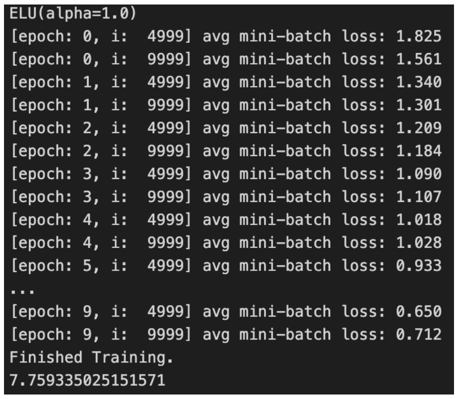
```
LeakyReLU(negative_slope=0.1)
[epoch: 0, i:  4999] avg mini-batch loss: 1.882
[epoch: 0, i:  9999] avg mini-batch loss: 1.558
[epoch: 1, i:  4999] avg mini-batch loss: 1.375
[epoch: 1, i:  9999] avg mini-batch loss: 1.283
[epoch: 2, i:  4999] avg mini-batch loss: 1.156
[epoch: 2, i:  9999] avg mini-batch loss: 1.130
[epoch: 3, i:  4999] avg mini-batch loss: 1.041
[epoch: 3, i:  9999] avg mini-batch loss: 1.023
[epoch: 4, i:  4999] avg mini-batch loss: 0.929
[epoch: 4, i:  9999] avg mini-batch loss: 0.943
[epoch: 5, i:  4999] avg mini-batch loss: 0.844
[epoch: 5, i:  9999] avg mini-batch loss: 0.883
[epoch: 6, i:  4999] avg mini-batch loss: 0.777
[epoch: 6, i:  9999] avg mini-batch loss: 0.824
[epoch: 7, i:  4999] avg mini-batch loss: 0.719
[epoch: 7, i:  9999] avg mini-batch loss: 0.762
[epoch: 8, i:  4999] avg mini-batch loss: 0.673
[epoch: 8, i:  9999] avg mini-batch loss: 0.707
[epoch: 9, i:  4999] avg mini-batch loss: 0.621
[epoch: 9, i:  9999] avg mini-batch loss: 0.674
Finished Training.
7.726108785470327
```



```
Sigmoid()
[epoch: 0, i:  4999] avg mini-batch loss: 2.311
[epoch: 0, i:  9999] avg mini-batch loss: 2.305
[epoch: 1, i:  4999] avg mini-batch loss: 2.296
[epoch: 1, i:  9999] avg mini-batch loss: 2.144
[epoch: 2, i:  4999] avg mini-batch loss: 1.999
[epoch: 2, i:  9999] avg mini-batch loss: 1.951
[epoch: 3, i:  4999] avg mini-batch loss: 1.880
[epoch: 3, i:  9999] avg mini-batch loss: 1.821
[epoch: 4, i:  4999] avg mini-batch loss: 1.755
[epoch: 4, i:  9999] avg mini-batch loss: 1.724
[epoch: 5, i:  4999] avg mini-batch loss: 1.673
...
[epoch: 9, i:  4999] avg mini-batch loss: 1.481
[epoch: 9, i:  9999] avg mini-batch loss: 1.463
Finished Training.
7.9442082444826765
```

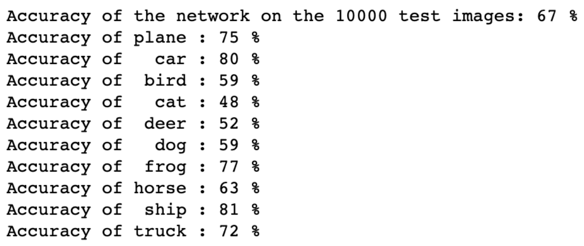


```
Tanh()
[epoch: 0, i:  4999] avg mini-batch loss: 1.841
[epoch: 0, i:  9999] avg mini-batch loss: 1.642
[epoch: 1, i:  4999] avg mini-batch loss: 1.485
[epoch: 1, i:  9999] avg mini-batch loss: 1.425
[epoch: 2, i:  4999] avg mini-batch loss: 1.337
[epoch: 2, i:  9999] avg mini-batch loss: 1.305
[epoch: 3, i:  4999] avg mini-batch loss: 1.230
[epoch: 3, i:  9999] avg mini-batch loss: 1.209
[epoch: 4, i:  4999] avg mini-batch loss: 1.140
[epoch: 4, i:  9999] avg mini-batch loss: 1.149
[epoch: 5, i:  4999] avg mini-batch loss: 1.066
...
[epoch: 9, i:  4999] avg mini-batch loss: 0.876
[epoch: 9, i:  9999] avg mini-batch loss: 0.913
Finished Training.
7.6583897789319355
```



```
ELU(alpha=1.0)
[epoch: 0, i:  4999] avg mini-batch loss: 1.825
[epoch: 0, i:  9999] avg mini-batch loss: 1.561
[epoch: 1, i:  4999] avg mini-batch loss: 1.340
[epoch: 1, i:  9999] avg mini-batch loss: 1.301
[epoch: 2, i:  4999] avg mini-batch loss: 1.209
[epoch: 2, i:  9999] avg mini-batch loss: 1.184
[epoch: 3, i:  4999] avg mini-batch loss: 1.090
[epoch: 3, i:  9999] avg mini-batch loss: 1.107
[epoch: 4, i:  4999] avg mini-batch loss: 1.018
[epoch: 4, i:  9999] avg mini-batch loss: 1.028
[epoch: 5, i:  4999] avg mini-batch loss: 0.933
...
[epoch: 9, i:  4999] avg mini-batch loss: 0.650
[epoch: 9, i:  9999] avg mini-batch loss: 0.712
Finished Training.
7.759335025151571
```

Since ReLU doesn't have any learnable parameters, it is hard to further improve the performance of our model using ReLU, therefore, LeakyReLU and ELU seem to be two possible approaches. We then tried LeakyReLU with a negative slope of 0.2, which gives us a test accuracy of 66%. This is slightly better than our first attempt with a negative slope of 0.1. We then did a parameter tuning with a negative slope of 0.3, 0.2, 0.18, 0.17, 0.15, and 0.12, and found out that the model has the highest accuracy when the negative slope is 0.15, which gives a test accuracy of 67%.

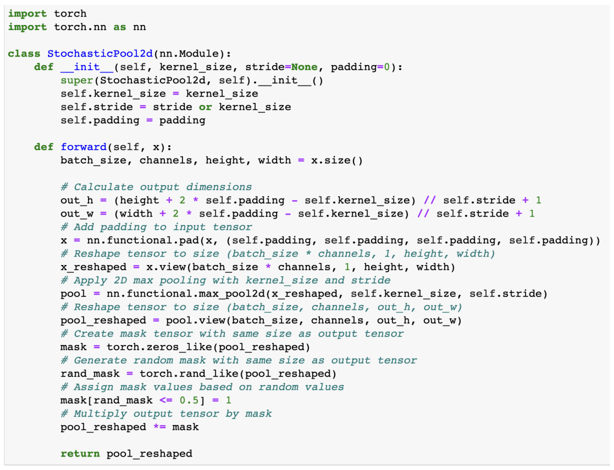


```
Accuracy of the network on the 10000 test images: 67 %
Accuracy of plane : 75 %
Accuracy of   car : 80 %
Accuracy of  bird : 59 %
Accuracy of   cat : 48 %
Accuracy of  deer : 52 %
Accuracy of   dog : 59 %
Accuracy of  frog : 77 %
Accuracy of horse : 63 %
Accuracy of  ship : 81 %
Accuracy of truck : 72 %
```

We also tried some small negative slope values such as 0.01 and 0.001 but obtained similar results as when the slope is around 0.1 and 0.2. We then tried ELU with alpha values of 0.1, 0.2, 0.5, 0.8, and 1. These five alpha values each produced test accuracy of 64%, 65%, 64%, 63%, and 64% respectively, so the alpha value seems not to have much effect on the performance of our model. With some research, we found other activation functions such as PReLU, SiLU, Mish, and GELU. We tried each of these and obtained test accuracy of 64%, 65%, 65%, and 64%. In summary, LeakyReLU with a negative slope of 0.15 provided us with a slightly higher test accuracy on this dataset besides ReLU. Given that the difference is not very significant and may be due to the randomness in testing, we decided to stick with ReLU and went on trying different pooling methods.

3.2 Pooling Functions

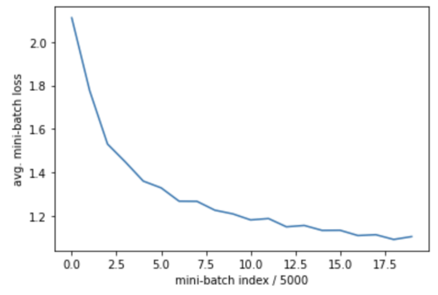
After doing some research, we found totally four different pooling methods, which are AveragePooling2D, MaxPooling2D, LPPooling2D, and StochasticPooling2D. Since our default model uses Average pooling, we then tried Max Pooling, LPPooling, and Stochastic Pooling, which gave us test accuracy of 67%, 67%, and 59% respectively.

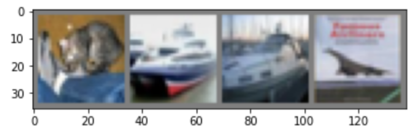


```python
import torch
import torch.nn as nn

class StochasticPool2d(nn.Module):
    def __init__(self, kernel_size, stride=None, padding=0):
        super(StochasticPool2d, self).__init__()
        self.kernel_size = kernel_size
        self.stride = stride or kernel_size
        self.padding = padding

    def forward(self, x):
        batch_size, channels, height, width = x.size()

        # Calculate output dimensions
        out_h = (height + 2 * self.padding - self.kernel_size) // self.stride + 1
        out_w = (width + 2 * self.padding - self.kernel_size) // self.stride + 1
        # Add padding to input tensor
        x = nn.functional.pad(x, (self.padding, self.padding, self.padding, self.padding))
        # Reshape tensor to size (batch_size * channels, 1, height, width)
        x_reshaped = x.view(batch_size * channels, 1, height, width)
        # Apply 2D max pooling with kernel_size and stride
        pool = nn.functional.max_pool2d(x_reshaped, self.kernel_size, self.stride)
        # Reshape tensor to size (batch_size, channels, out_h, out_w)
        pool_reshaped = pool.view(batch_size, channels, out_h, out_w)
        # Create mask tensor with same size as output tensor
        mask = torch.zeros_like(pool_reshaped)
        # Generate random mask with same size as output tensor
        rand_mask = torch.rand_like(pool_reshaped)
        # Assign mask values based on random values
        mask[rand_mask <= 0.5] = 1
        # Multiply output tensor by mask
        pool_reshaped *= mask

        return pool_reshaped
```

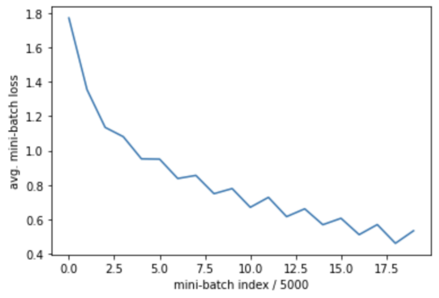
Source: above code was generated from Chatgpt

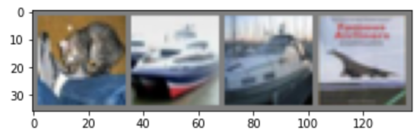
The StochasticPooling will "randomly pick activation from each region based on probability distribution; therefore, each value in the pooling area has an equal probability of being selected"(Definition provided by Chatgpt). After doing some research, we learned that Stochastic Pooling works best when the model is more complex because a complex model is prone to overfitting. Our model is pretty simple, which contains only two convolutional layers and two fully connected linear layers. Therefore, we concluded that instead of preventing overfitting, the randomness introduced by Stochastic Pooling actually lowered the accuracy.



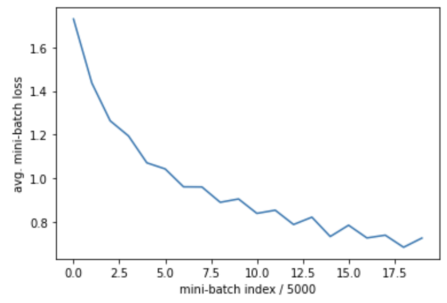


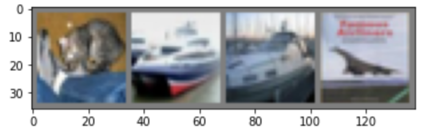
```
GroundTruth:    cat  ship  ship plane
Predicted:   ship   car  ship plane
Accuracy of the network on the 10000 test images: 59 %
```





```
GroundTruth:    cat  ship  ship plane
Predicted:    dog   car plane plane
Accuracy of the network on the 10000 test images: 67 %
```

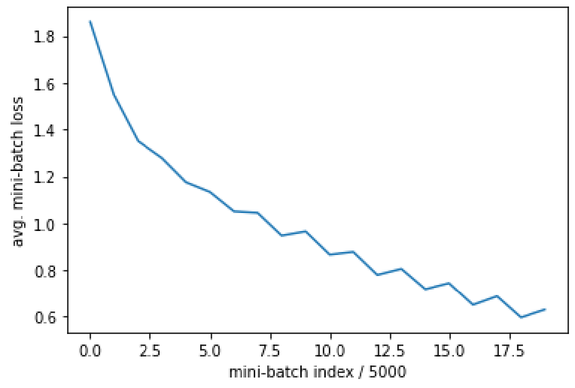




```
GroundTruth:    cat  ship  ship plane
Predicted:    cat   car plane plane
Accuracy of the network on the 10000 test images: 67 %
```

3.3 Optimizers

Moreover, we implemented the CNN with different optimizers to optimize the loss function in different ways. We first used the stochastic gradient descent optimizer. SGD calculates the error and updates weights for each training sample. We specified the learning rate to be 0.001 and momentum to be 0.9 and left the other parameters as default values. SGD produced an accuracy of 65% on the overall dataset.



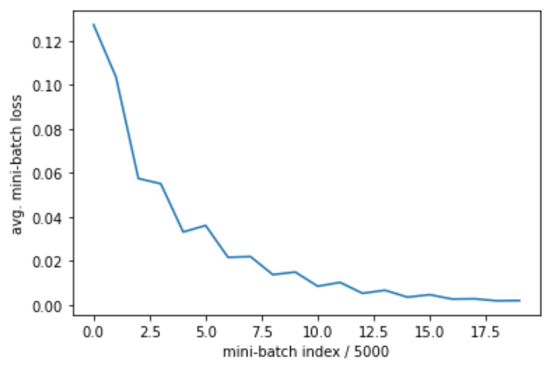
Next, we used the Adam algorithm as the optimizer, which uses momentum and an adaptive learning rate. We used the default parameters. The Adam algorithm yielded an accuracy of 64%.

We then chose the adaptive gradient (Adagrad) algorithm as the optimizer. It is a gradient-based optimization where each parameter has a different learning rate. Adagrad optimizer produced an accuracy of 66%.

We also used the Adadelta optimization algorithm. Similar to Adagrad, it also uses an adaptive learning rate. Besides, it uses a moving window to update gradients. Adadelta achieved an accuracy of 64%.

Then, we tried the AdamW optimizer. AdamW decouples weight decay from the gradient update, which makes it different from the Adam optimizer. We were able to achieve an accuracy of 63% using AdamW.

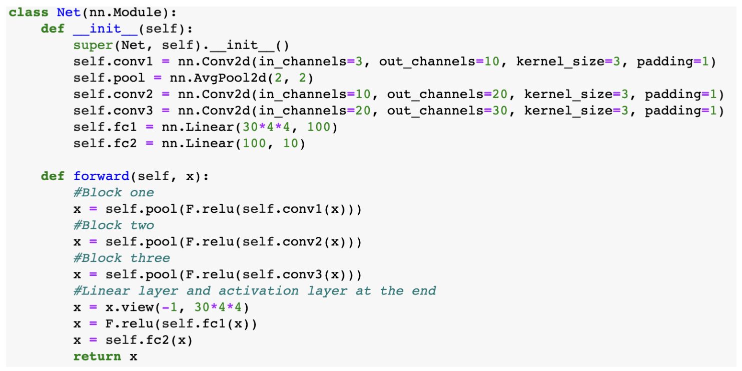
Moreover, we used the Adamax optimizer, which uses the infinity norm. Using Adamax, we obtained an accuracy of 66%.



Lastly, we used the Root Mean Squared Propagation (RMSProp) optimizer, which uses a decaying average of partial gradients. RMSProp produced an accuracy of 35% on the dataset, which is the lowest accuracy among all optimizers. We could improve the accuracy by using a lower learning rate. To summarize, SGD, Adagrad, and Adamax produced the highest accuracy on the dataset, so they are probably the best-suited optimizer for image classification on the CIFAR-10 dataset using CNN.

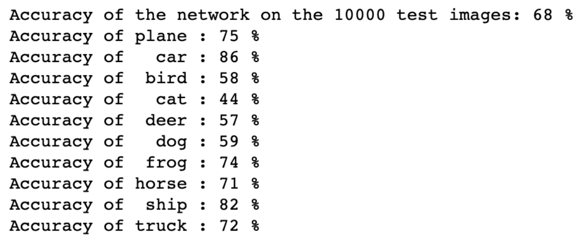
3.4 Number of Layers

After experimenting with different optimizers, we tried adding more layers to our CNN to improve its performance. To add more layers to our CNN, we first tried to add a block that is the same as the first two blocks, which we added at the end of the second block. Right now, our CNN contains three convolutional layers, three activation layers, and three pooling layers. At the end of these three blocks, we have two fully connected linear layers and the activation layer.



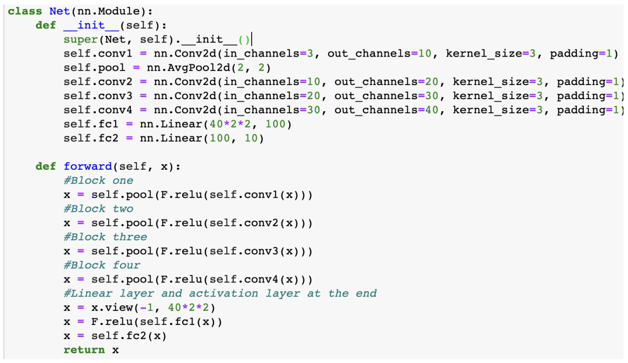
```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=10, kernel_size=3, padding=1)
        self.pool = nn.AvgPool2d(2, 2)
        self.conv2 = nn.Conv2d(in_channels=10, out_channels=20, kernel_size=3, padding=1)
        self.conv3 = nn.Conv2d(in_channels=20, out_channels=30, kernel_size=3, padding=1)
        self.fc1 = nn.Linear(30*4*4, 100)
        self.fc2 = nn.Linear(100, 10)

    def forward(self, x):
        #Block one
        x = self.pool(F.relu(self.conv1(x)))
        #Block two
        x = self.pool(F.relu(self.conv2(x)))
        #Block three
        x = self.pool(F.relu(self.conv3(x)))
        #Linear layer and activation layer at the end
        x = x.view(-1, 30*4*4)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

We then ran our CNN and produced a test accuracy of 68% which is 3% higher than our original CNN.



```
Accuracy of the network on the 10000 test images: 68 %
Accuracy of plane : 75 %
Accuracy of   car : 86 %
Accuracy of  bird : 58 %
Accuracy of   cat : 44 %
Accuracy of  deer : 57 %
Accuracy of   dog : 59 %
Accuracy of  frog : 74 %
Accuracy of horse : 71 %
Accuracy of  ship : 82 %
Accuracy of truck : 72 %
```

Since the current strategy is effective, we then add one more block to our CNN to make it contain four convolutional layers, four activation layers, and four pooling layers. We ran our CNN but got a test accuracy of 62%.

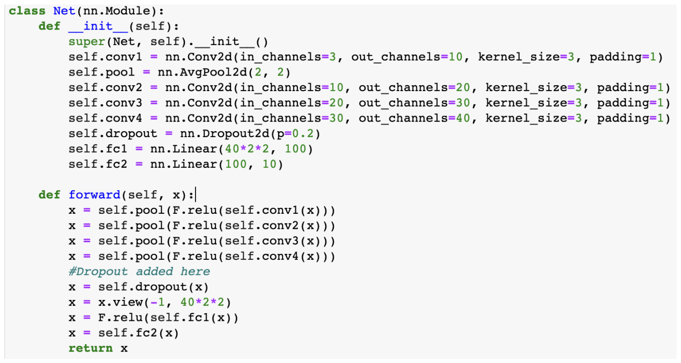


```python
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=10, kernel_size=3, padding=1)
        self.pool = nn.AvgPool2d(2, 2)
        self.conv2 = nn.Conv2d(in_channels=10, out_channels=20, kernel_size=3, padding=1)
        self.conv3 = nn.Conv2d(in_channels=20, out_channels=30, kernel_size=3, padding=1)
        self.conv4 = nn.Conv2d(in_channels=30, out_channels=40, kernel_size=3, padding=1)
        self.fc1 = nn.Linear(40*2*2, 100)
        self.fc2 = nn.Linear(100, 10)

    def forward(self, x):
        #Block one
        x = self.pool(F.relu(self.conv1(x)))
        #Block two
        x = self.pool(F.relu(self.conv2(x)))
        #Block three
        x = self.pool(F.relu(self.conv3(x)))
        #Block four
        x = self.pool(F.relu(self.conv4(x)))
        #Linear layer and activation layer at the end
        x = x.view(-1, 40*2*2)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

```
Accuracy of the network on the 10000 test images: 62 %
Accuracy of plane : 68 %
Accuracy of   car : 61 %
Accuracy of  bird : 42 %
Accuracy of   cat : 62 %
Accuracy of  deer : 57 %
Accuracy of   dog : 44 %
Accuracy of  frog : 57 %
Accuracy of horse : 75 %
Accuracy of  ship : 75 %
Accuracy of truck : 79 %
```

This indicates a CNN with three convolutional blocks might be the most appropriate architecture, so we will explore this hypothesis further in the batch normalization section. Since right now our CNN contains a total of 15 layers (including 2 linear layers and an activation layer in between at the end), we suspect that our model is overfitting, which causes the test accuracy to drop from 68% to 62%.

After doing some research on how to prevent overfitting, we decided to use Dropout2d() to randomly set some input channels to zero. We added the dropout layer at the end of the four convolutional blocks.

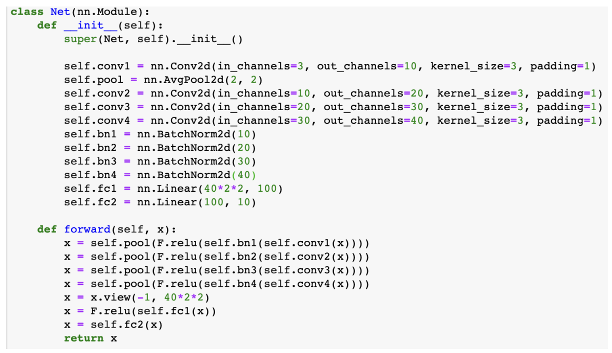


```python
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=10, kernel_size=3, padding=1)
        self.pool = nn.AvgPool2d(2, 2)
        self.conv2 = nn.Conv2d(in_channels=10, out_channels=20, kernel_size=3, padding=1)
        self.conv3 = nn.Conv2d(in_channels=20, out_channels=30, kernel_size=3, padding=1)
        self.conv4 = nn.Conv2d(in_channels=30, out_channels=40, kernel_size=3, padding=1)
        self.dropout = nn.Dropout2d(p=0.2)
        self.fc1 = nn.Linear(40*2*2, 100)
        self.fc2 = nn.Linear(100, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = self.pool(F.relu(self.conv3(x)))
        x = self.pool(F.relu(self.conv4(x)))
        #Dropout added here
        x = self.dropout(x)
        x = x.view(-1, 40*2*2)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

We then ran our CNN and this time it produced a test accuracy of 65%, which is an improvement but still not what we expected. Since Dropout2d() has a hyper-parameter p, which determines the fraction of channels to be set to 0, we decided to manipulate p with values p=0.1 and p=0.3 (originally we have p=0.2). We then ran our CNN but the test accuracy remained at 65%. Given that dropout is not very helpful, we decided to use batch normalization to reduce the overfitting.

```
Accuracy of the network on the 10000 test images: 65 %
Accuracy of plane : 68 %
Accuracy of   car : 79 %
Accuracy of  bird : 53 %
Accuracy of   cat : 49 %
Accuracy of  deer : 47 %
Accuracy of   dog : 52 %
Accuracy of  frog : 81 %
Accuracy of horse : 70 %
Accuracy of  ship : 79 %
Accuracy of truck : 72 %
```

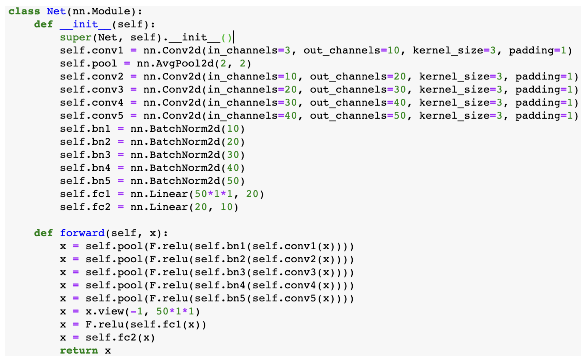
“Batch normalization normalizes the activation vectors from the convolutional layer by recentering and rescaling” (Definition provided by Chatgpt) and can be used to prevent overfitting. We added batch normalization after each convolution layer and before each activation layer. By batch normalizing the four convolutional layers, test accuracy jumped from 62% to 71%.



```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()

        self.conv1 = nn.Conv2d(in_channels=3, out_channels=10, kernel_size=3, padding=1)
        self.pool = nn.AvgPool2d(2, 2)
        self.conv2 = nn.Conv2d(in_channels=10, out_channels=20, kernel_size=3, padding=1)
        self.conv3 = nn.Conv2d(in_channels=20, out_channels=30, kernel_size=3, padding=1)
        self.conv4 = nn.Conv2d(in_channels=30, out_channels=40, kernel_size=3, padding=1)
        self.bn1 = nn.BatchNorm2d(10)
        self.bn2 = nn.BatchNorm2d(20)
        self.bn3 = nn.BatchNorm2d(30)
        self.bn4 = nn.BatchNorm2d(40)
        self.fc1 = nn.Linear(40*2*2, 100)
        self.fc2 = nn.Linear(100, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.bn1(self.conv1(x))))
        x = self.pool(F.relu(self.bn2(self.conv2(x))))
        x = self.pool(F.relu(self.bn3(self.conv3(x))))
        x = self.pool(F.relu(self.bn4(self.conv4(x))))
        x = x.view(-1, 40*2*2)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

```
Accuracy of the network on the 10000 test images: 71 %
Accuracy of plane : 83 %
Accuracy of   car : 85 %
Accuracy of  bird : 62 %
Accuracy of   cat : 62 %
Accuracy of  deer : 71 %
Accuracy of   dog : 51 %
Accuracy of  frog : 74 %
Accuracy of horse : 74 %
Accuracy of  ship : 74 %
Accuracy of truck : 75 %
```

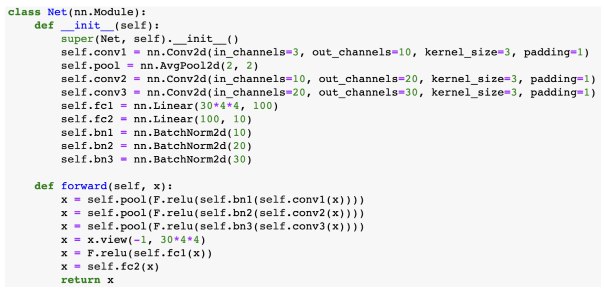
We then added one more convolutional block to see if adding more layers can further improve the accuracy with the help of normalization. So now our CNN contains five convolutional layers, five activation layers, and five pooling layers. We again batch-normalized each convolutional layer and ran our CNN, but this time the accuracy dropped to 68%.



```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=10, kernel_size=3, padding=1)
        self.pool = nn.AvgPool2d(2, 2)
        self.conv2 = nn.Conv2d(in_channels=10, out_channels=20, kernel_size=3, padding=1)
        self.conv3 = nn.Conv2d(in_channels=20, out_channels=30, kernel_size=3, padding=1)
        self.conv4 = nn.Conv2d(in_channels=30, out_channels=40, kernel_size=3, padding=1)
        self.conv5 = nn.Conv2d(in_channels=40, out_channels=50, kernel_size=3, padding=1)
        self.bn1 = nn.BatchNorm2d(10)
        self.bn2 = nn.BatchNorm2d(20)
        self.bn3 = nn.BatchNorm2d(30)
        self.bn4 = nn.BatchNorm2d(40)
        self.bn5 = nn.BatchNorm2d(50)
        self.fc1 = nn.Linear(50*1*1, 20)
        self.fc2 = nn.Linear(20, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.bn1(self.conv1(x))))
        x = self.pool(F.relu(self.bn2(self.conv2(x))))
        x = self.pool(F.relu(self.bn3(self.conv3(x))))
        x = self.pool(F.relu(self.bn4(self.conv4(x))))
        x = self.pool(F.relu(self.bn5(self.conv5(x))))
        x = x.view(-1, 50*1*1)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

```
Accuracy of the network on the 10000 test images: 68 %
Accuracy of plane : 79 %
Accuracy of   car : 80 %
Accuracy of  bird : 47 %
Accuracy of   cat : 41 %
Accuracy of  deer : 62 %
Accuracy of   dog : 70 %
Accuracy of  frog : 78 %
Accuracy of horse : 73 %
Accuracy of  ship : 77 %
Accuracy of truck : 77 %
```

Given the fact that three convolutional blocks produced the highest test accuracy when we run it without normalization, we decided to normalize this three-block model to see if it still has the highest accuracy.



```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=10, kernel_size=3, padding=1)
        self.pool = nn.AvgPool2d(2, 2)
        self.conv2 = nn.Conv2d(in_channels=10, out_channels=20, kernel_size=3, padding=1)
        self.conv3 = nn.Conv2d(in_channels=20, out_channels=30, kernel_size=3, padding=1)
        self.fc1 = nn.Linear(30*4*4, 100)
        self.fc2 = nn.Linear(100, 10)
        self.bn1 = nn.BatchNorm2d(10)
        self.bn2 = nn.BatchNorm2d(20)
        self.bn3 = nn.BatchNorm2d(30)

    def forward(self, x):
        x = self.pool(F.relu(self.bn1(self.conv1(x))))
        x = self.pool(F.relu(self.bn2(self.conv2(x))))
        x = self.pool(F.relu(self.bn3(self.conv3(x))))
        x = x.view(-1, 30*4*4)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

```
Accuracy of the network on the 10000 test images: 72 %
Accuracy of plane : 79 %
Accuracy of   car : 89 %
Accuracy of  bird : 59 %
Accuracy of   cat : 56 %
Accuracy of  deer : 73 %
Accuracy of   dog : 54 %
Accuracy of  frog : 73 %
Accuracy of horse : 80 %
Accuracy of  ship : 80 %
Accuracy of truck : 80 %
```

We then ran this model and just as we expected, the accuracy increased back to 72%, which is one percent higher than the four-block model. Therefore, we can conclude that three convolutional blocks are probably the best architecture for our CNN to classify images in the CIFAR-10 dataset.

3.5 Hyperparameters Tuning

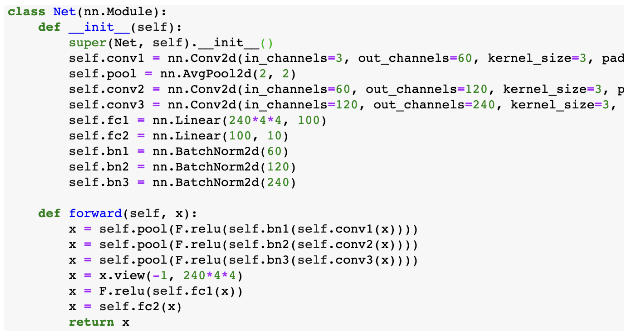
After experimenting with adding more layers, we decided to continue with the model with three convolutional layers. We first went on to test different learning rates. Because our default learning rate is 0.001, we then tried 0.1, 0.01, and 0.0001. After running, we found that when the learning rate is 0.1, the model simply predicts everything as category "dog". Also, the model finished training very quickly which means the "model converged too quickly to a suboptimal"(Definition provided by Chatgpt). With a learning rate equal to 0.01, we got a test accuracy of 68%, and with a learning rate equal to 0.0001, the model trained very slowly and no batch loss got printed out so the model probably got stuck because the learning rate is too low. Therefore, we decided that the original learning rate of 0.001 is actually pretty good. After testing the learning rate, we went on to test different batch sizes. Our original batch size is 4 so we decided to test batch sizes equal to 8, 16, and 32. After running our model, the test accuracy for 8, 16, and 32 are 73%, 72%, and 71%. Given that 8 seems to be a proper batch size, we then went on to test different epochs with batch size 8. Our default epoch is 10 so we decided to test epochs equal to 20 and 40. With 20 epochs, we got a test accuracy of 74% but when we increased the epochs to 40, the accuracy dropped to 72% and it started to take a very long time to train. To solve this problem, we increase the batch size to 16 to speed up the training. With 40 epochs and a batch size of 16, the test accuracy went back to 74%. Given the result doesn't differ very much, we decided to stick with epochs of 20 and a batch size of 8.

To further increase the accuracy, we tried to adjust the input and output channels size of the convolution layers to let the model take in more features. In our original model, the channel size is as follows:

First convolutional layer: in_channels=3, out_channels=10

Second convolutional layer: in_channels=10, outchannels=20

Third convolutional layer: in_channels=20, out_channels=30



```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=60, kernel_size=3, pad
        self.pool = nn.AvgPool2d(2, 2)
        self.conv2 = nn.Conv2d(in_channels=60, out_channels=120, kernel_size=3, p
        self.conv3 = nn.Conv2d(in_channels=120, out_channels=240, kernel_size=3,
        self.fc1 = nn.Linear(240*4*4, 100)
        self.fc2 = nn.Linear(100, 10)
        self.bn1 = nn.BatchNorm2d(60)
        self.bn2 = nn.BatchNorm2d(120)
        self.bn3 = nn.BatchNorm2d(240)

    def forward(self, x):
        x = self.pool(F.relu(self.bn1(self.conv1(x))))
        x = self.pool(F.relu(self.bn2(self.conv2(x))))
        x = self.pool(F.relu(self.bn3(self.conv3(x))))
        x = x.view(-1, 240*4*4)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

We then increased the channel size to let it start at 60 and double it in the next two layers. After running this model, our test accuracy jumped to 80%, which is a huge improvement.

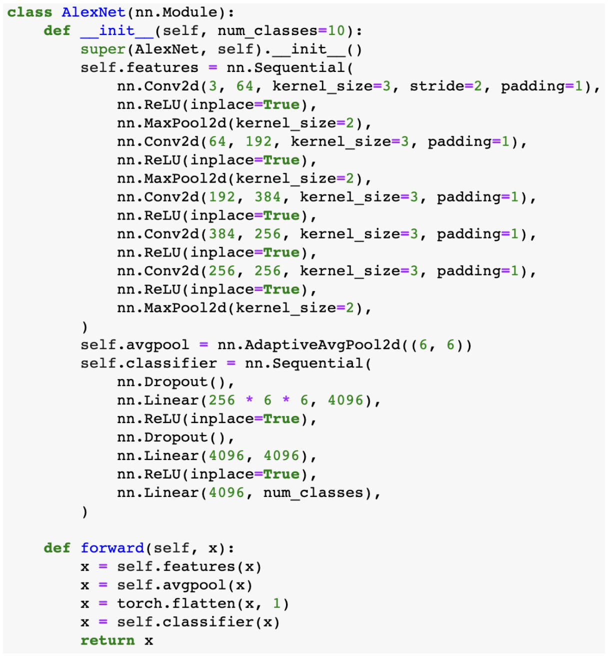
```
Accuracy of the network on the 10000 test images: 80 %
Accuracy of plane : 80 %
Accuracy of   car : 88 %
Accuracy of  bird : 68 %
Accuracy of   cat : 69 %
Accuracy of  deer : 74 %
Accuracy of   dog : 68 %
Accuracy of  frog : 86 %
Accuracy of horse : 90 %
Accuracy of  ship : 90 %
Accuracy of truck : 89 %
```

Since increasing channel size works very well, we double the channel size again to let it start with 120 and end with 480, which boost our test accuracy by just 1%. To see if it can be further improved, we doubled it again to let it start with 240 and end with 960, but this time the test accuracy dropped back to 80%. It seems like the channel size has reached its limit.

```
Accuracy of the network on the 10000 test images: 81 %
Accuracy of plane : 78 %
Accuracy of   car : 88 %
Accuracy of  bird : 67 %
Accuracy of   cat : 56 %
Accuracy of  deer : 79 %
Accuracy of   dog : 78 %
Accuracy of  frog : 87 %
Accuracy of horse : 91 %
Accuracy of  ship : 93 %
Accuracy of truck : 90 %
```

3.6 AlexNet and ResNet

After experimenting with CNN on the dataset, we also attempted to classify the images using other existing CNN architectures – AlexNet and ResNet. AlexNet contains a total of 5 convolutional layers and 3 fully connected linear layers. "AlexNet first used MaxPooling after the ReLU activation function to extract the most prominent local feature and used just one average pooling to get the global feature that is useful for classification"(Definition provided by Chatgpt). AlexNet has three fully connected linear layers at the end. Also, before the three fully connected layers, AlexNet uses dropout to prevent overfitting. Since AlexNet was initially used on the ImageNet dataset, it has a num_classes of 1000. To run it on the CIFAR-10 dataset, we change the num_classes to 10. We ran the AlexNet with batch_size=8 and epochs=20 and got an accuracy of 81%. If we compare the result of AlexNet with the CNN we defined in the last paragraph, we can see that AlexNet has a more balanced accuracy for each class (highest accuracy for ship and car with 88% and lowest for the bird with 68%). The CNN we defined is less balanced in accuracy for each class, with the highest accuracy of 93% for the ship and the lowest accuracy of 56% for the cat. Also, AlexNet has an average higher batch loss than our CNN but in the end, has the same test accuracy as our CNN, which means AlexNet does a better job of preventing overfitting than our model.



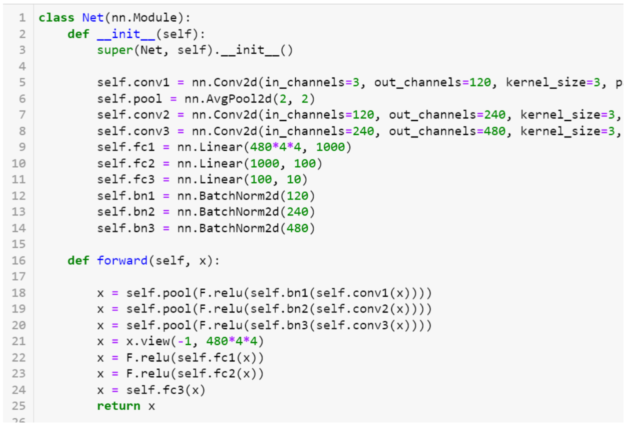
```python
class AlexNet(nn.Module):
    def __init__(self, num_classes=10):
        super(AlexNet, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=3, stride=2, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2),
            nn.Conv2d(64, 192, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2),
            nn.Conv2d(192, 384, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(384, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2),
        )
        self.avgpool = nn.AdaptiveAvgPool2d((6, 6))
        self.classifier = nn.Sequential(
            nn.Dropout(),
            nn.Linear(256 * 6 * 6, 4096),
            nn.ReLU(inplace=True),
            nn.Dropout(),
            nn.Linear(4096, 4096),
            nn.ReLU(inplace=True),
            nn.Linear(4096, num_classes),
        )

    def forward(self, x):
        x = self.features(x)
        x = self.avgpool(x)
        x = torch.flatten(x, 1)
        x = self.classifier(x)
        return x
```

Source: above code was generated from Chatgpt

Last but not least, we trained a ResNet on the dataset. This ResNet (Liu, 2017) contains a basic block and a residual block. The basic block consists of 2 convolutional blocks that contain a convolutional layer with batch normalization followed by a ReLU activation layer, as well as a shortcut block that contains a convolutional layer with an expansion factor and batch normalization and that can be activated when the convolutional layer is set to be expanded. The residual block contains two hidden layers, where each layer consists of 2 basic blocks with different numbers of input and output channels and an optional shortcut block. Training with the Adamax optimizer on 20 epochs and a batch size of 8, this ResNet produced an accuracy of 86% on the dataset, which is much higher than the typical CNN, and slightly higher than AlexNet. Similar to AlexNet, ResNet produced more balanced accuracies across all classes than our CNN with the best performance. It achieved an accuracy of 91% in the car and truck class, which is the highest, and 72% in the bird class, which is the lowest.

4 Conclusion

In conclusion, we successfully classified images in the CIFAR-10 dataset using a CNN and found the most suitable CNN architecture and optimization methods for image classification on the CIFAR-10 dataset. Specifically, a CNN with 3 convolutional blocks and 3 linear blocks appears to have the best performance. Each convolutional block comprises a batch-normalized convolutional layer followed by a ReLU activation layer and an average pooling layer. Each linear block comprises a linear layer and a ReLU activation layer. We found ReLU and Leaky ReLU to be the best activation functions, SGD, Adagrad, and Adamax to be the best optimizers, and average pooling, max pooling, and LP pooling perform relatively the same. A batch size of 8 appears to produce the best training and test result. However, due to the limited computing power of the GPU we have access to, our hyperparameter tuning might not be comprehensive enough.



```python
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()

        self.conv1 = nn.Conv2d(in_channels=3, out_channels=120, kernel_size=3, p
        self.pool = nn.AvgPool2d(2, 2)
        self.conv2 = nn.Conv2d(in_channels=120, out_channels=240, kernel_size=3,
        self.conv3 = nn.Conv2d(in_channels=240, out_channels=480, kernel_size=3,
        self.fc1 = nn.Linear(480*4*4, 1000)
        self.fc2 = nn.Linear(1000, 100)
        self.fc3 = nn.Linear(100, 10)
        self.bn1 = nn.BatchNorm2d(120)
        self.bn2 = nn.BatchNorm2d(240)
        self.bn3 = nn.BatchNorm2d(480)

    def forward(self, x):

        x = self.pool(F.relu(self.bn1(self.conv1(x))))
        x = self.pool(F.relu(self.bn2(self.conv2(x))))
        x = self.pool(F.relu(self.bn3(self.conv3(x))))
        x = x.view(-1, 480*4*4)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

We found that AlexNet produced a similar result to the best-performing CNN, but it is probably less prone to overfitting. Our results show that ResNet outperforms CNN and AlexNet, so it is probably the neural network that is best suited for image classification on the CIFAR-10 dataset.

Our findings demonstrate the effectiveness of CNNs in image classification tasks and highlight the importance of optimizing hyperparameters and choosing the appropriate architecture, activation function, and optimizer for the dataset. This paper contributes to the growing field of deep learning and calls attention to the potential of CNNs for real-world applications. Further study can explore more advanced CNN architectures, such as ResNext, and optimization techniques to improve the performance of image classification models.

References

Liu, K. (2017) “Kuangliu/Pytorch-CIFAR.” GitHub, https://github.com/kuangliu/pytorch-cifar.