Student Number: 181334

1. Introduction

In this report, we will be implementing a deep neural network with an input layer, three hidden layers with ReLu non-linear activation and an output or classification layer. We will be training a neural network to classify images from the CIFAR-10 dataset. This report documents the training process investigating the effects of the following hyperparameter settings on accuracy: Batch size, Depth, Width, Convolutional filter size, Dropout, Batchnorm, Max pool, Tanh non-linearity, Optimiser, Weights initialization, Regularisation (weight decay), Learning rate, Learning rate scheduler. Further we explored the number of epochs, momentum and normalization and noted the model accuracy on 10000 test images along with final training loss.

Please refer to Github commits for each stage of the development.

1.1. APPROACH AND METHOD

Our approach to exploring the following hyperparameters is first by beginning with the PyTorch CIFAR-10 tutorial implementation[1] following the official documentation. From here we manipulate each individual hyperparameter sequentially and individually with 5 different values. We then visualize and take the best performing hyperparameter variation of the current hyperparameter being investigated and move on to explore 5 further variations of the next hyperparameter in the list retaining the best performing previous hyperparameter variations.
of the next hyperparameter in the list on top of the best performing previous hyperparameter as found.
We used 10,000 test images and 40,000 training images. (20% test images).
Assumptions: -There are no mislabeled pieces of data in the training data set.
-There are an even division of all images across the 10 classes.
- Reducing loss should yield higher accuracy.

This incremental approach enables as systematic progression towards some solution. However, it is possible that this process drives us to converge into a local minima and suboptimal solution.

Once these existing hyperparameters have been investigated and best performing hyperparameters have been selected by the heuristic of model accuracy, we then investigate further types of hyperparameters with the goal of incremental improvements model accuracy through reducing over/under-fitting.

2. METHODOLOGY

We begin by loading up the CIFAR-10 Tutorial project.

Batch Size:	Accuracy of the network on 10000 test images:	Final Loss after training
1	46%	1.526
4	60%	1.033
6	62%	1.030
10	58%	1.136
8	60%	1.106
7	61%	1.055

From these selection of batch sizes, we conclude that the batch size of 6 produced the lowest loss and highest model accuracy of 62%. A 2% improvement over batch size 4.

From this stage, we see a batch size of 6 produced the highest accuracy of the network of 10000 images of 62% with the final loss after training of 1.030.

We set the batch size to 6 and proceed to experiment with 2, 3, and 4 conv2d layers. Currently our model as per the original implementation has 4 hidden layers.

```
self.conv1 = nn.Conv2d(3, 6, 5)
self.pool = nn.MaxPool2d(2, 2)
self.conv2 = nn.Conv2d(6, 16, 5)
self.fc1 = nn.Linear(16 * 5 * 5, 120)
self.fc2 = nn.Linear(120, 84)
self.fc3 = nn.Linear(84, 10)
```

with forward pass

```
x = self.pool(F.relu(self.conv1(x)))
x = self.pool(F.relu(self.conv2(x)))
x = x.view(-1, 16 * 5 * 5)
x = F.relu(self.fc1(x))
x = F.relu(self.fc2(x))
x = self.fc3(x)
```

This yielded an accuracy of 62% with Final Loss after training of 1.030 on 10000 images.

With 3 hidden layers with the network defined as

```
self.conv1 = nn.Conv2d(3, 6, 5)
self.pool = nn.MaxPool2d(2, 2)
self.conv2 = nn.Conv2d(6, 16, 5)
self.fc1 = nn.Linear(16 * 5 * 5, 120)
self.fc3 = nn.Linear(120, 10)
```

with forward pass

```
x = self.pool(F.relu(self.conv1(x)))
x = self.pool(F.relu(self.conv2(x)))
x = x.view(-1, 16 * 5 * 5)
x = F.relu(self.fc1(x))
x = self.fc3(x)
```

This yielded an accuracy of 61% with Final Loss after training of 1.037 on 10000 images.

```
self.conv1 = nn.Conv2d(3, 6, 5)
self.pool = nn.MaxPool2d(2, 2)
self.conv2 = nn.Conv2d(6, 16, 5)
self.fc1 = nn.Linear(16 * 5 * 5, 10)
```

with forward pass:

```
x = self.pool(F.relu(self.conv1(x)))
x = self.pool(F.relu(self.conv2(x)))
x = x.view(-1, 16 * 5 * 5)
x = F.relu(self.fc1(x))
```

This yielded an accuracy of 60% with Final Loss after training of 1.161 on 10000 images.

From here, we see that reducing hidden layers reduced accuracy.

```
self.conv1 = nn.Conv2d(3, 6, 5)
self.pool = nn.MaxPool2d(2, 2)
self.conv2 = nn.Conv2d(6, 16, 5)
self.fc1 = nn.Linear(16 * 5 * 5, 240)
self.fc2 = nn.Linear(240, 84)
self.fc3 = nn.Linear(84, 10)
```

with forward pass:

```
x = self.pool(F.relu(self.conv1(x)))
x = self.pool(F.relu(self.conv2(x)))
x = x.view(-1, 16 * 5 * 5)
x = F.relu(self.fc1(x))
x = F.relu(self.fc2(x))
x = self.fc3(x)
```

This yielded an accuracy of 64% with Final Loss after training of 1.005 on 10000 images. We then proceeded to replace the width of 240 with 480. With 480, we achieved a final loss of 1.001 but this only yielded an accuracy of 61%. As we observe the loss before the end of training went as low as 0.974 before increasing to 1.001, we suspect this to be overfitting. We tried again by changing the width from 480 to 320 and we observed a final loss of 0.974 yielding 60% accuracy. Further, using value 180 resulted in accuracy of 61% with loss of 0.997.
We set the value back to 240 which yielded an accuracy of 64% and proceed to investigate convolutional filter size.

Next we experiment varying the convolutional filter size. We resized the first convolution filter size to a (3, 3) kernel_size. This led to a final loss of 1.005 and accuracy of 10000 images of 61%. We found no gain in performance for using a smaller kernel size.

Next we introduced a dropout layers between the fully connected layers.

```
self.conv1 = nn.Conv2d(3, 6, 5)
self.pool = nn.MaxPool2d(2, 2)
self.conv2 = nn.Conv2d(6, 16, 5)
self.fc1 = nn.Linear(16 * 5 * 5, 240)
self.fc2 = nn.Linear(240, 84)
self.fc3 = nn.Linear(84, 10)
self.dropout = nn.Dropout(0.25)
```

with the forward pass

```
x = self.pool(F.relu(self.conv1(x)))
x = self.pool(F.relu(self.conv2(x)))
x = x.view(-1, 16 * 5 * 5)
x = F.relu(self.fc1(x))
x = self.dropout(x)
x = F.relu(self.fc2(x))
x = self.dropout(x)
x = self.fc3(x)
```

We first experiment with a dropout rate of 0.25.
This neural network resulted in an accuracy of 9%
We adjust the dropout rate to 0.1 this gives accuracy of 60%. The dropout rate of 0.01 gives accuracy of 62%.
We adjust the forward pass to have only 1 dropout layer and dropout rate 0.25.

```
x = self.pool(F.relu(self.conv1(x)))
x = self.pool(F.relu(self.conv2(x)))
x = x.view(-1, 16 * 5 * 5)
x = F.relu(self.fc1(x))
x = self.dropout(x)
x = F.relu(self.fc2(x))
x = self.fc3(x)
```

This dropout has an accuracy of 60%.
We adjust the forward pass as

```
x = self.pool(F.relu(self.conv1(x)))
x = self.pool(F.relu(self.conv2(x)))
x = x.view(-1, 16 * 5 * 5)
x = F.relu(self.fc1(x))
x = self.dropout(x)
x = F.relu(self.fc2(x))
x = self.fc3(x)
```

The dropout rate of 0.01 in this configuration yielded an accuracy of 63%. A decrease from our 64% previous best.

From here we introduced 2 batch norm1d layers

```
self.conv1 = nn.Conv2d(3, 6, 5)
    self.pool = nn.MaxPool2d(2, 2)
    self.conv2 = nn.Conv2d(6, 16, 5)
    self.fc1 = nn.Linear(16 * 5 * 5, 240)
    self.batchnorm = nn.BatchNorm1d(240)
    self.fc2 = nn.Linear(240, 84)
```

```
        self.batchnorm = nn.BatchNorm1d(84)
        self.fc3 = nn.Linear(84, 10)
```
with forward pass:
```
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
```
This led to a final loss of 1.013 with an accuracy of 62%
From this CNN, we explore the previously explored batch sizes with the following results:
Note: We recreated the initial neural network from the batch size exploration phase to produce this table to ensure we get a fair comparison.

```
self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.batchnorm = nn.BatchNorm1d(120)
        self.fc2 = nn.Linear(120, 84)
        self.batchnorm = nn.BatchNorm1d(84)
        self.fc3 = nn.Linear(84, 10)
```

Batch Size:	Accuracy on 10000 test images with no batch norm	Final Loss after training No batch norm	Accuracy on 10000 test images with batch norm1d	Final Loss after training With batch norm
1	46%	1.526	46%	1.453
4	60%	1.033	61%	1.069
6	62%	1.030	61%	1.072
10	58%	1.136	60%	1.118
8	60%	1.106	62%	1.077
7	61%	1.055	62%	1.052

Here we see how this batch norm layers affect the accuracy with a selection of batch sizes. As we know we have a more accurate model using the previously most high scoring model
```
self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 240)
        self.batchnorm = nn.BatchNorm1d(240)
        self.fc2 = nn.Linear(240, 84)
        self.batchnorm = nn.BatchNorm1d(84)
        self.fc3 = nn.Linear(84, 10)
```
with forward pass

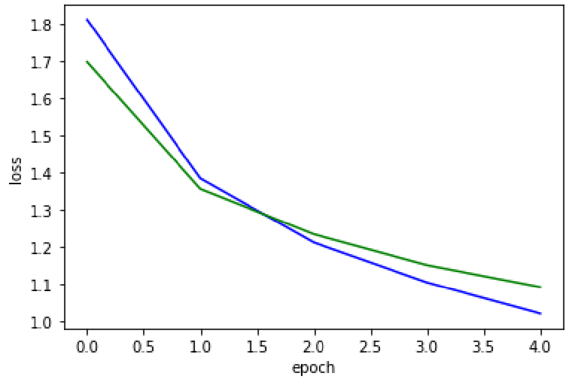
Batch Size:	Accuracy on 10000 test images with batch norm1d Width 120	Final Loss after training With batch norm	Accuracy on 10000 test images with batch norm1d Width 240	Final Loss after training With batch norm with Width 240
1	46%	1.453	49%	1.518
4	61%	1.069	62%	0.970
6	61%	1.072	60%	1.069
10	60%	1.118	61%	1.095
8	62%	1.077	61%	1.039
7	62%	1.052	62%	0.997

Here we see we were able to achieve a lower loss but no noticeable improvement in model accuracy.

We use the following network yielding a 63% accuracy with loss 0.971.
```
        self.conv1 = nn.Conv2d(3, 6, 5)
        #self.pool = nn.AvgPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 240)
        self.fc2 = nn.Linear(240, 84)
        self.fc3 = nn.Linear(84, 10)
```
with forward pass:
```
        x = F.relu(self.conv1(x))
        x = F.relu(self.conv2(x))
        x = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
```
The effects of this yield an accuracy of 57% with final training loss of 1.127.

We then change the AvgPool2d to LPPool2d which yielded an accuracy of 61% with final training loss of 0.999.
We set the AvgPool2d(2, 2) back to MaxPool2d(2, 2) and proceed to explore use of Tanh non-linearity.

We adjust the forward pass as follows:
```
x = self.pool(F.tanh(self.conv1(x)))
        x = self.pool(F.tanh(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = F.tanh(self.fc1(x))
        x = F.tanh(self.fc2(x))
        x = self.fc3(x)
```
to give an accuracy of 61% with final loss of 1.066



We now use the Relu activation function as this yielded the highest accuracy. Next we experiment with optimizers SGD, Adam and RMSProp. The accuracies across 10000 test images are:
Accuracy from SGD = 62%.
Accuracy from Adam = 60%.
Accuracy from RMSProp = 10%.

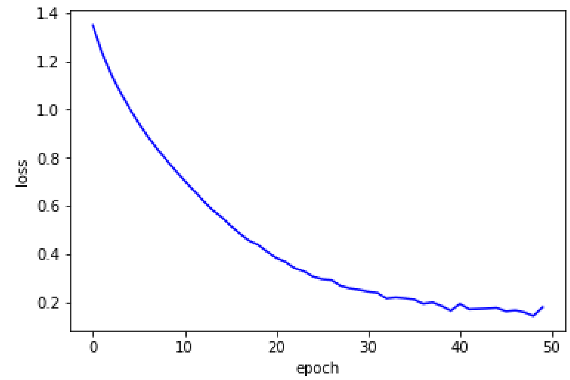
Next revert to SGD then we use He initialization of weights:[2]

```
def initialize_weights(self):
    for m in self.modules():
        if isinstance(m, nn.Conv2d):
            nn.init.kaiming_uniform_(m.weight)
            if m.bias is not None:
                nn.init.constant_(m.bias, 0)

        elif isinstance(m, nn.Linear):
            nn.init.kaiming_uniform(m.weight)
            nn.init.constant_(m.bias, 0)
```

This resulted in accuracy of 59% and final training loss of 1.007. To further explore, we only apply the He weights to the Conv2d layer to give an accuracy of 61% with loss at 0.973.
We then remove the initialize_weights[2] function and we introduce a weight decay. With a weight decay of 1, the loss remains constant and yields a 10% accuracy as the loss remains constant. We found varying we found 0.1 and 5 weight decay caused the same result. We conclude the weight decay causes this specific model to not learn anything as 10% accuracy is effectively random.
We therefore omitted this weight decay and proceeded to explore the learning rate.
Changing the learning rate to 0.0005 resulted in accuracy of 60%. We then increased the Epochs from 5 to 10 to achieve a 62% accuracy.
We then adjusted the learning rate to 0.001, but increased the Epochs to 25 to achieve an accuracy of 63% with final loss of 0.345.

We then increased the Epochs to 50 with learning rate 0.001. This resulted in an accuracy of 59% despite having 0.289 loss.
With 50 Epochs, we half the learning rate to investigate this effect. We found this gave a loss of 0.179 and accuracy of 59%.



We adjust the learning rate to 0.0005 with 10 epochs. We change the normalization to 0.45 to achieve an accuracy of 64%. We then increased the epochs to 15 to achieve 65% accuracy. We increased the epochs to 20 and achieved an accuracy of 62%. We suspect this to be a case of Overfitting..
We changed epochs back to 15 and investigated the momentum hyperparameter. We changed momentum to 0.75. This had a final loss of 0.866 at the end of training and accuracy of 64%, a 1% decline. We then try changing the momentum to 0.95 resulting in 60% accuracy and final loss of 0.525. We set the momentum back to 0.9.
We found that a significant improvement in performance was adjusting the normalization of the images from 0.5 to 0.45 so we then further experimented with normalization of 0.475. This resulted in an accuracy of 65% with a final training loss of 0.601.

3. DISCUSSION

After following this approach, we found no meaningful improvement in model accuracy beyond 65%. To better understand the model training process, monitoring training using TensorBoard[3] to see the accuracy as it trains would likely identify the point at which the accuracy begins to decline enabling us greater insight into correcting overfitting and underfitting.
This challenges our initial assumption that minimizing loss should inherently cause an increase in accuracy. We showed that this is not the case. Further work should investigate to what extent each of these hyperparameters affect the training accuracy/loss and document inferences of overfitting or underfitting.
Further work should plot accuracy along with loss to better build intuitions on if we are over or underfitting our model to our data.

References

[1] Pytorch.org. 2021. Training a Classifier — PyTorch Tutorials 1.8.1+cu102 documentation. [online] Available at: <https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html > [Accessed 20 May 2021].https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html

[2] MachineCurve. 2021. He/Xavier initialization & activation functions: choose wisely – MachineCurve. [online] Available at: <https://www.machinecurve.com/index.php/2019/09/16/he-xavier-initialization-activation-functions-choose-wisely/> [Accessed 20 May 2021].

[3] MachineCurve. 2021. He/Xavier initialization & activation functions: choose wisely – MachineCurve. [online] Available at: <https://www.machinecurve.com/index.php/2019/09/16/he-xavier-initialization-activation-functions-choose-wisely/> [Accessed 20 May 2021].