1 Machine Learning Basics (10 points)

1.1 Calculating gradients (2.0 points)

A major aspect of neural network training is identifying optimal values for all the network parameters (weights and biases). Computing gradients of the loss function w.r.t these parameters is an essential operation in this regard (gradient descent). For some parameter w (a scalar weight at some layer of the network), and for a loss function L, the weight update is given by $w := w - \alpha \frac{\partial L}{\partial w}$, where α is the learning rate/step size.

Consider (a) w, a scalar, (b) $\mathbf{x}$, a vector of size $(m \times 1)$, (c) $\mathbf{y}$, a vector of size $(n \times 1)$ and (d) $\mathbf{A}$, a matrix of size $(m \times n)$. Find the following gradients, and express them in the simplest possible form (boldface lowercase letters represent vectors, boldface uppercase letters represent matrices, plain lowercase letters represent scalars):

- $z = \mathbf{x}^T\mathbf{x}$, find $\frac{dz}{d\mathbf{x}}$
- $z = Trace(\mathbf{A}^T\mathbf{A})$, find $\frac{dz}{d\mathbf{A}}$
- $z = \mathbf{x}^T\mathbf{A}\mathbf{y}$, find $\frac{\partial z}{\partial \mathbf{y}}$
- $\mathbf{z} = \mathbf{A}\mathbf{y}$, find $\frac{d\mathbf{z}}{d\mathbf{y}}$

You may use the following formulae for reference:

$$\frac{\partial z}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial z}{\partial x_1} \\ \frac{\partial z}{\partial x_2} \\ \vdots \\ \frac{\partial z}{\partial x_m} \end{bmatrix}, \frac{\partial z}{\partial \mathbf{A}} = \begin{bmatrix} \frac{\partial z}{\partial A_{11}} & \frac{\partial z}{\partial A_{12}} & \cdots & \frac{\partial z}{\partial A_{1n}} \\ \frac{\partial z}{\partial A_{21}} & \frac{\partial z}{\partial A_{22}} & \cdots & \frac{\partial z}{\partial A_{2n}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial z}{\partial A_{m1}} & \frac{\partial z}{\partial A_{m2}} & \cdots & \frac{\partial z}{\partial A_{mn}} \end{bmatrix}, \frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_2}{\partial x_1} & \cdots & \frac{\partial y_n}{\partial x_1} \\ \frac{\partial y_1}{\partial x_2} & \frac{\partial y_2}{\partial x_2} & \cdots & \frac{\partial y_n}{\partial x_2} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_1}{\partial x_m} & \frac{\partial y_2}{\partial x_m} & \cdots & \frac{\partial y_n}{\partial x_m} \end{bmatrix}$$

$z = \mathbf{x}^T\mathbf{x}, \frac{dz}{d\mathbf{x}} = 2x$

$z = Trace(\mathbf{A}^T\mathbf{A}), \frac{dz}{d\mathbf{A}} = Trace(\frac{d\mathbf{A}^T\mathbf{A}}{d\mathbf{A}}) = \sum_{i=1}^{m} 2a_{ii}$

$$z = \mathbf{x}^T\mathbf{A}\mathbf{y}, z = \begin{bmatrix} x_1 & x_2 & \dots & x_m \end{bmatrix} \begin{bmatrix} A_{11} & A_{12} & \dots & A_{1n} \\ A_{21} & A_{22} & \dots & A_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ A_{m1} & A_{m2} & \dots & A_{mn} \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} x_1 \sum_{i=1}^{n} A_{1i}y_i \\ x_2 \sum_{i=1}^{n} A_{2i}y_i \\ \vdots \\ x_m \sum_{i=1}^{n} A_{mi}y_i \end{bmatrix}$$

$$\frac{\partial z}{\partial \mathbf{y}} = \begin{bmatrix} x_1A_{11} \\ x_2A_{22} \\ \vdots \\ x_mA_m \end{bmatrix} = A^Tx$$

$\mathbf{z} = \mathbf{A}\mathbf{y}, \frac{d\mathbf{z}}{d\mathbf{y}} = \frac{d}{d\mathbf{y}}Ay = A$

1.2 Deriving Cross entropy Loss (6.0 points)

In this problem, we derive the cross entropy loss for binary classification tasks. Let $\hat{y}$ be the output of a classifier for a given input x. y denotes the true label (0 or 1) for the input x. Since y has only 2 possible values, we can assume it follow a Bernoulli distribution w.r.t the input x. We hence wish to come up with a loss function $L(y,\hat{y})$, which we would like to minimize so that the difference between $\hat{y}$ and y reduces. A Bernoulli random variable (refresh your pre-test material) takes a value of 1 with a probability k, and 0 with a probability of $1-k$.

(i) Write an expression for $p(y|x)$, which is the probability that the classifier produces an observation $\hat{y}$ for a given input. Your answer would be in terms of $y,\hat{y}$. Justify your answer briefly.

$$p(y|x) = \hat{y}^y(1-\hat{y})^{1-y}$$

(ii) Using (i), write an expression for $\log p(y|x)$. $\log p(y|x)$ denotes the log-likelihood, which should be maximized.

$$\log p(y|x) = y\log(\hat{y}) + (1-y)\log(1-\hat{y})$$

(iii) How do we obtain $L(y,\hat{y})$ from $\log p(y|x)$? Note that $L(y,\hat{y})$ is to be minimized

$$L(y,\hat{y}) = -\log(p(y|x))$$
$$L(y,\hat{y}) = -y\log(\hat{y}) - (1-y)\log(1-\hat{y})$$

1.3 Perfect Classifier (?) (2.0 points)

You train a classifier on a training set, achieving an impressive accuracy of 100 %. However to your disappointment, you obtain a test set accuracy of 20 %. For each suggestion below, explain why (or why not) if these suggestions may help improve the testing accuracy.

1. Use more training data
2. Add L2 regularization to your model
3. Increase your model size, i.e. increase the number of parameters in your model
4. Create a validation set by partitioning your training data. Use the model with highest accuracy on the validation set, not the training set.

This problem is known as overfitting: the model is too trained on the original data set such that it cannot predict from unseen data well. Using more training data could help with this issue. Clean data that is not too noisy could make the model more accurately predict variables. There is more variation and diversity with more data, therefore the model is not as fixated on

the original training data. L2 regularization makes the weights of data small and close to zero, therefore features from the original data set will have less of an influence on predicting unseen data - this will help improve the test accuracy. Increasing the model size will lead to more overfitting. The model will have increased complexity, which will likely overfit the data even more. Creating a validation set by partitioning training data is known as k-fold validation is a method of preventing overfitting. The hyperparameters, or specific higher-level parameters specific to different iterations of the model would be trained, leading to a model with better testing accuracy.

2 Implementing an image classifier using PyTorch (15.0 points)

In this problem you will implement a CNN based image classifier in pytorch. We will work with the CIFAR-10 dataset. Follow the instructions in jupyter-notebook to complete the missing parts. For this part, you will use the notebook named PSET4_Classification. For training the model on colab gpus, upload the notebook on google colab, and change the runtime type to GPU.

2.1 Loading Data (2.0 points)

(i) Explain the function of transforms.Normalize() function (See the Jupyter notebook Q1 cell). How will you modify the arguments of this function for gray scale images instead of RGB images.
(ii) Write the code snippets to print the number of training and test samples loaded.

Make sure that your answer is within the bounding box.

(i) The transforms.Normalize() function uses the mean and standard deviation of the data set in order to standardize each channel. The formula is image $= \frac{\text{image} \times \text{std}}{\text{mean}}$
(ii) Grayscale:

```
transforms.Normalize((0.5, ), (0.5, ))
```

2.2 Classifier Architecture (6.0 points)

(See the Jupyter Notebook) Please go through the supplied code that defines the architecture (cell Q2 in the Jupyter Notebook), and answer the following questions.

1. Describe the entire architecture. The description for each layer should include details about kernel sizes, number of channels, activation functions and the type of the layer.
2. What does the padding parameter control?
3. Briefly explain the max pool layer.
4. What would happen if you change the kernel size to 3 for the CNN layers without changing anything else? Are you able to pass a test input through the network and get back an output of the same size? Why/why not? If not, what would you have to change to make it work?
5. While backpropagating through this network, for which layer you don't need to compute any additional gradients? Explain Briefly Why.

1. Layer 1 - conv1
3*6=18 convolutions, kernel size: 5, stride: 1, 3 normalized channels, no padding, 6 feature maps, relu activation function
Layer 2 - max pool
max pooling, kernel size: 2, output 6 14×14 channels
Layer 3 - conv2
6*16=96 convolutions, 16 feature maps, kernel size: 5, stride: 1, 16 10×10 output channels,

no padding, relu activation function
Layer 4 - max pool
max pooling, kernel size: 2, 16 5×5 output channels
Layer 5 - fc1
fully connected layer, 16×5×5=400 input, 120 output, $400 \times 120 = 48000$ weights, relu activation function
Layer 6 - fc2
fully connected layer, 120 input, 84 output, $120 \times 84 = 10080$ weights, relu activation function
Layer 7 - fc3
fully connected layer, 84 input, 10 output, $84 \times 10 = 840$ weights

2. The padding parameter controls how many layers of zeroes to add to the image borders.
3. The max pool layer takes the maximum value of a group of pixels in order to down sample the image data. This lowers the complexity of the image data by reducing the number of parameters and thus prevents overfitting.
4. Chaning the kernel size to 3 would prevent the 2nd max pooling layer from producing equal conv2d layers since the grid would not be able to slide across the 2nd conv2 output, which would be 13×13. This will result in a $16 \times 16 \times 6$ 2nd max pooling layer and thus we would need to change the size of the first fully connected layer.

```
self.fc1 = nn.Linear(16 * 6 * 6, 120),
#-----more code-----
x = x.view(-1, 16 * 6 * 6)
```

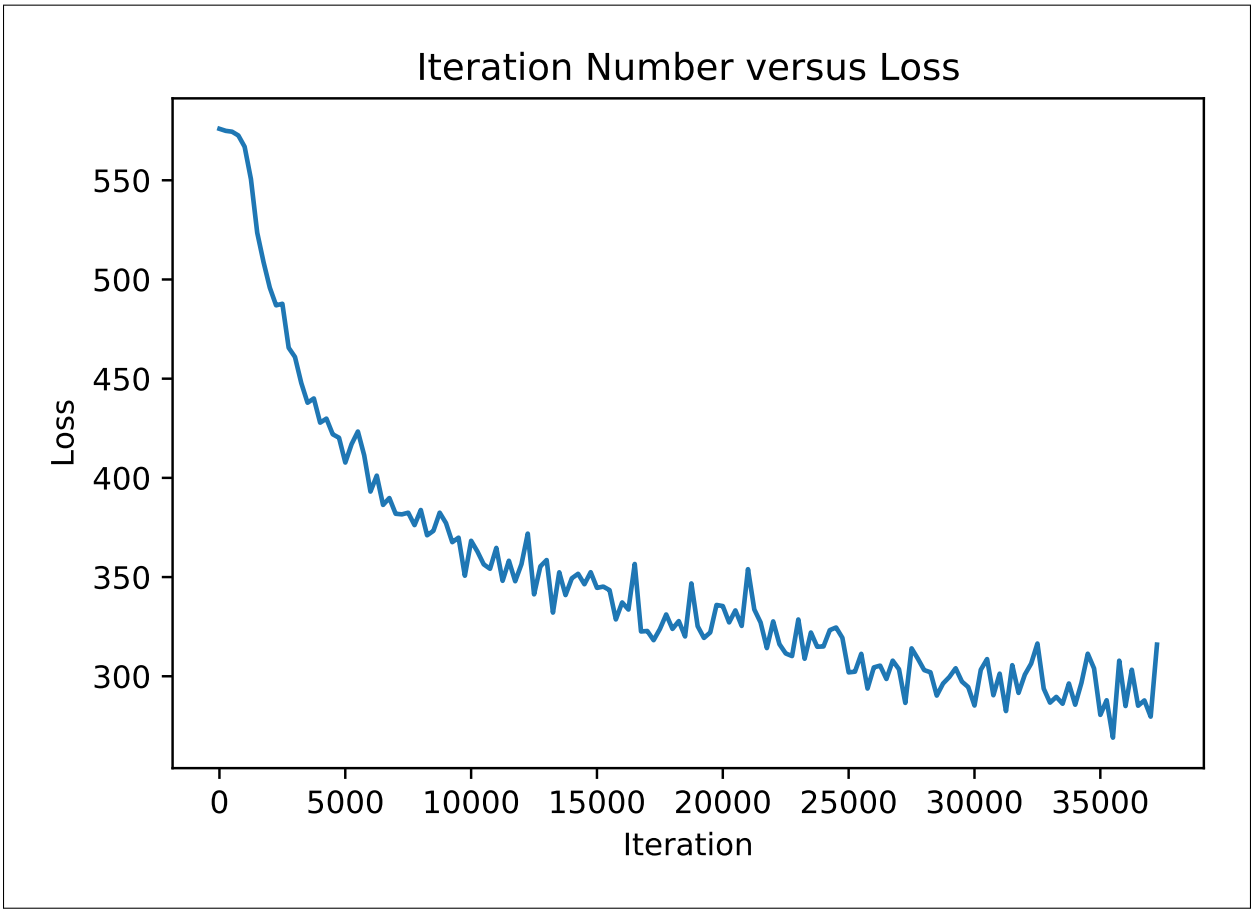
5. We do not need to take the additional gradients for the initial output since it would just be the gradient with respect to itself, which is 1.

2.3 Training the network (3.0 points)

(i) (See the Jupyter notebook.) Complete the code in the jupyter notebook for training the network on a CPU, and paste the code in the notebook. Train your network for 3 epochs. Plot the running loss (in the notebook) w.r.t epochs.

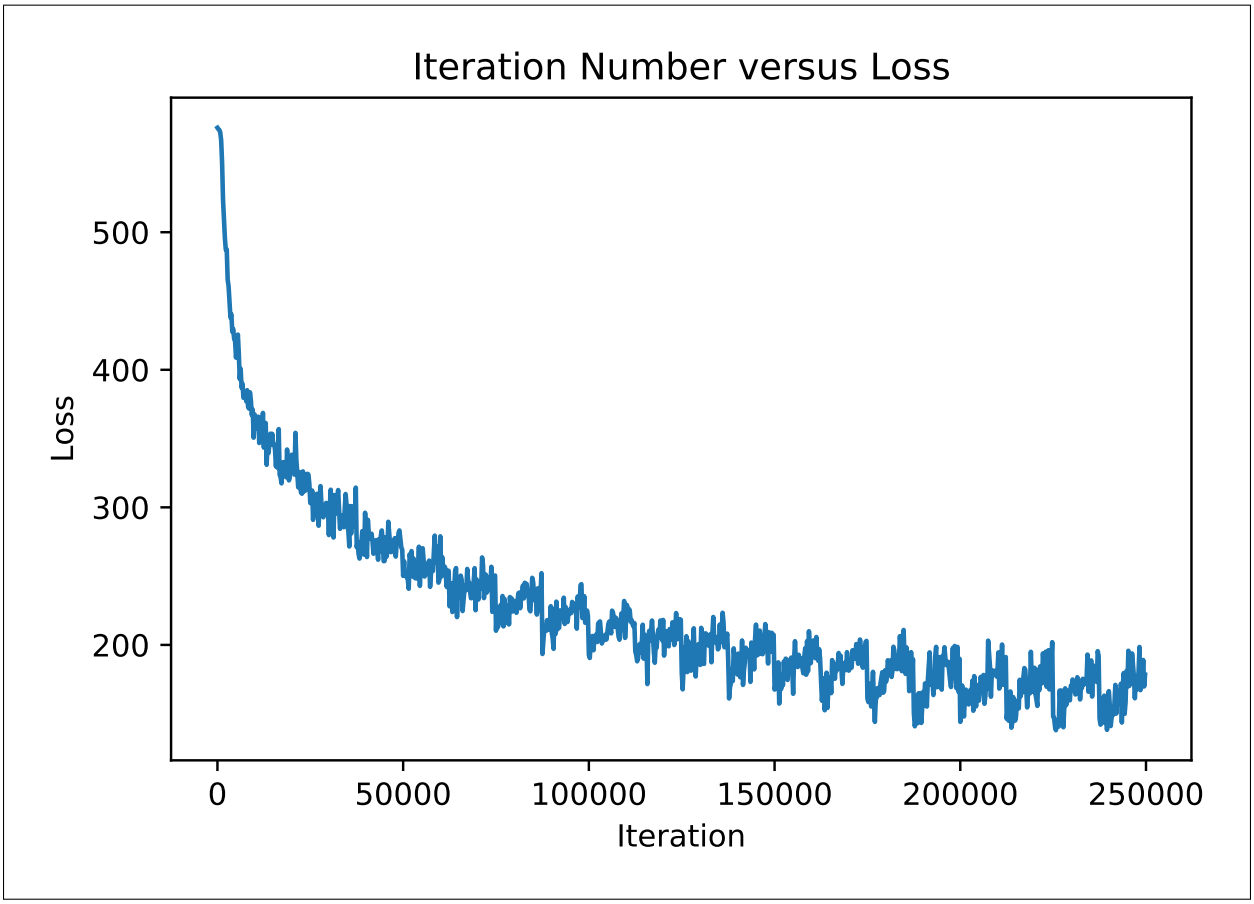
```
## for reproducibility
torch.manual_seed(7)
np.random.seed(7)

## Instantiating classifier
net = Net()

## Defining optimizer and loss function
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
```

```
## Defining Training Parameters

num_epochs = 3 # 2 for CPU training, 10 for GPU training
running_loss_list = [] # list to store running loss in the code below
for epoch in range(num_epochs):  # loop over the dataset multiple times
    running_loss = 0.0
    for i, data in enumerate(train_loader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data
        #================================================#
        # Fill in the training loop here.
        optimizer.zero_grad()
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        #================================================#
        # print statistics
        running_loss += loss.cpu().item()
        if i % 250 == 249:    # print every 250 mini-batches
            print('[{}, {}] loss: {:.3f}'.format(epoch + 1, i + 1,
             ↪  running_loss / 250))
            running_loss_list.append(running_loss)
            running_loss = 0.0

print('Training Complete')
PATH = './net.pth'
torch.save(net.state_dict(), PATH)


def plot_loss_curve(running_loss_list):
    iters = len(running_loss_list)*250

    fig, ax = plt.subplots()
    ax.plot(np.arange(1, iters, 250), np.array(running_loss_list))
    ax.set_xlabel("Iteration")
    ax.set_ylabel("Loss")
    ax.set_title("Iteration Number versus Loss")
    return fig

plot_loss_curve(running_loss_list).savefig('data/solutions/question_3_1.pdf',
 ↪  format='pdf', bbox_inches='tight')
```



(ii) (See the Jupyter notebook.) Modify your training code, to train the network on the GPU. Paste here the lines that need to be modified to train the network on google colab GPUs. Train the network for 20 epochs

```
net = Net().cuda()
#-----more code-----
# Fill in the training loop here.
optimizer.zero_grad()
outputs = net(inputs.cuda())
loss = criterion(outputs.cuda(), labels.cuda())
#-----more code-----
running_loss += loss.cuda().item()
```



(iii) Explain why you need to reset the parameter gradients for each pass of the network

For each mini batch of data taken from the population, there is a different gradient between batches. Therefore, we must reset the parameter gradients to account for each batch.

2.4 Testing the network (4.0 points)

(i) (See the jupyter-notebook) Complete the code in the jupyter-notebook to test the accuracy of the network on the entire test set.

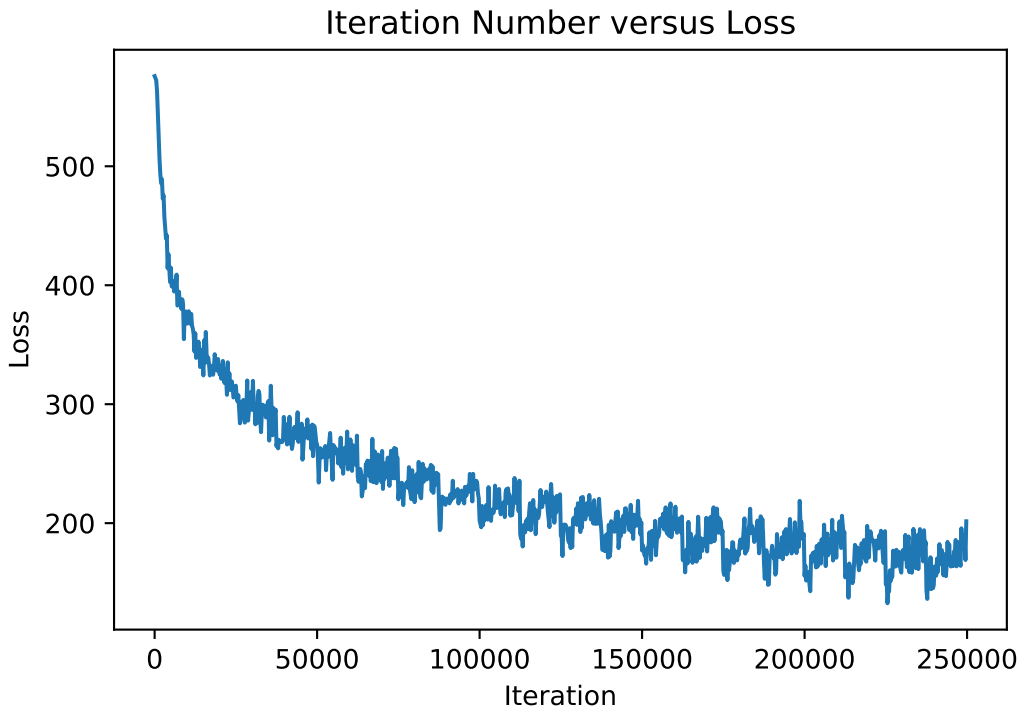
```
### Accuracy on whole data set
correct = 0
total = 0
with torch.no_grad():
    for data in test_loader:
        images, labels = data
        outputs = net(images)
```

```
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
acc  = correct / total * 100
print('Accuracy of the network on the 10000 test images: %d %%' % (acc))
```

(ii) Train the network on the GPU with the following configurations, and report the testing accuracies and running loss curves -

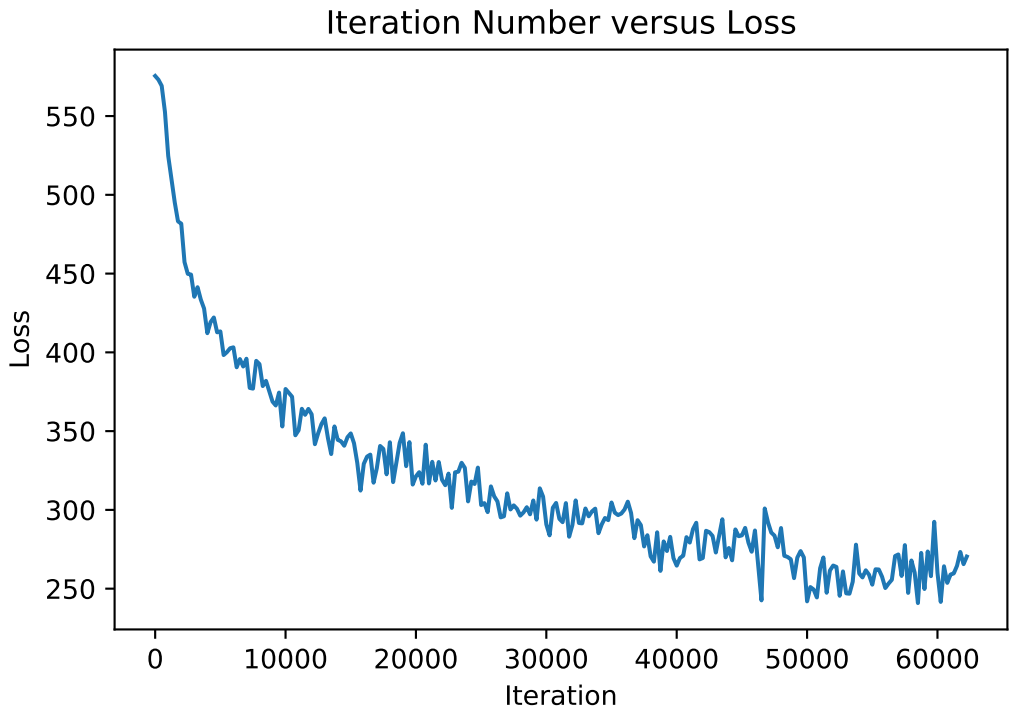
- Training Batch Size 4, 20 training epochs
- Training Batch Size 4, 5 epochs
- Training Batch Size 16, 5 epochs
- Training Batch Size 16, 20 epochs

Training Batch Size 4, 20 training epochs:
Accuracy: 61%

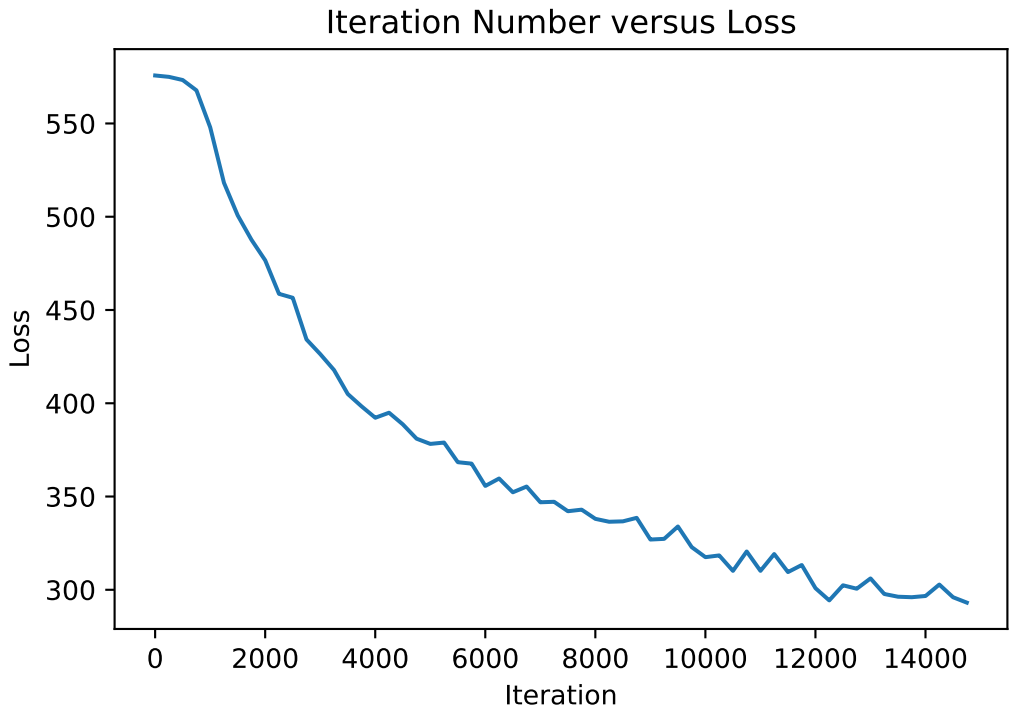


Training Batch Size 4, 5 training epochs:

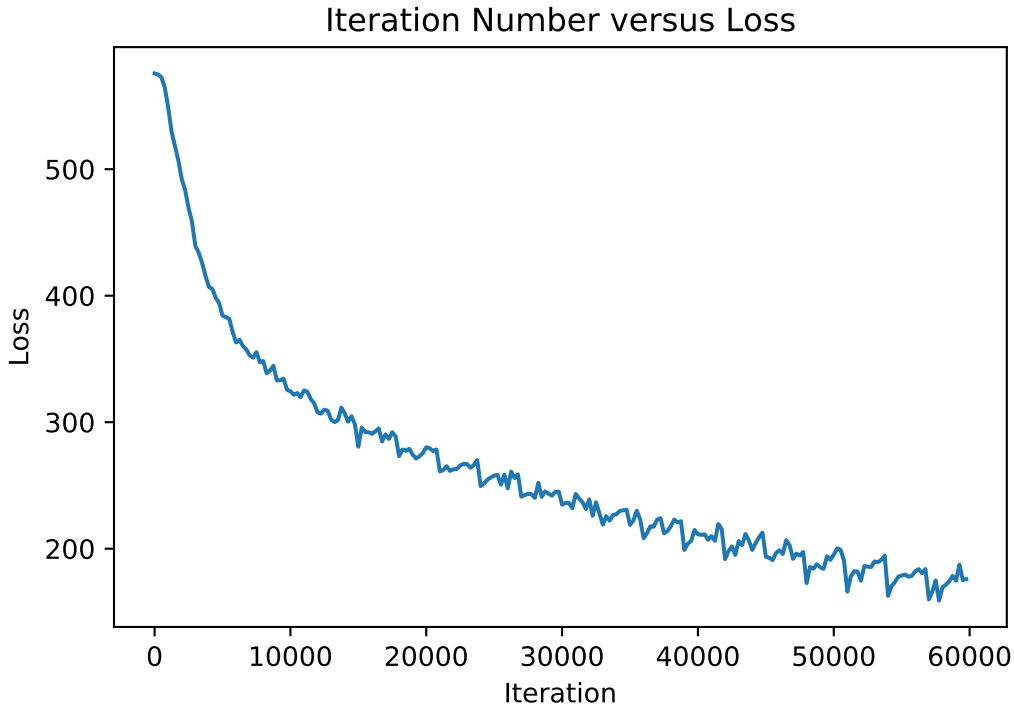
Accuracy: 61%



Training Batch Size 16, 5 training epochs:
Accuracy: 61%



Training Batch Size 16, 20 training epochs:
Accuracy: 77%



(iii) Explain your observations in (ii)

With the smaller batch size, there is more variation in the loss and thus a much more rough curve. A larger batch size contributed however toward greater accuracy due to to the more accurate gradients computed.

3 Interview Questions (15 points)

3.1 Batch Normalization (4 points)

Explain
(i) Why batch normalization acts as a regularizer.
(ii) Difference in using batch normalization at training vs inference (testing) time.

(i.) Batch normalization standardizes the activations and inputs, scaling batches according to corresponding means and standard deviations. This adds noise to the layers, making the weights be adjusted and thus causing a regularization effect.
(ii.) Using batch normalization at inference time uses the mean and standard deviation of the entire population while during training only uses the mean and standard deviation of and for each batch.

3.2 CNN filter sizes (4 points)

Assume a convolution layer in a CNN with parameters $C_{in} = 32$, $C_{out} = 64, k = 3$. If the input to this layer has the parameters $C = 32, H = 64, W = 64$.

(i) What will be the size of the output of this layer, if there is no padding, and stride = 1
(ii) What should be the padding and stride for the output size to be $C = 64, H = 32, W = 32$

(i) 62×62, 64 channels
(ii) H=32

$$H = 32 = \lfloor \frac{H_{in} + 2 \times \text{padding} - \text{dilation} \times (\text{kernel size} - 1) - 1}{\text{stride}} + 1 \rfloor$$

$$32 = \lfloor \frac{64 + 2 - \text{padding} - 1(3-1) - 1}{\text{stride}} + 1 \rfloor$$

$$32 = \lfloor \frac{63 - \text{padding}}{\text{stride}} + 1 \rfloor$$

$$31 \times \text{stride} = 63 - \text{padding}$$

$$31(2) = 63 - (1)$$

Padding = 1, Stride = 2

3.3 L2 regularization and Weight Decay (4 points)

Assume a loss function of the form $L(y, \hat{y})$ where y is the ground truth and $\hat{y} = f(x, w)$. x denotes the input to a neural network (or any differentiable function) $f()$ with paramters/weights denoted by w. Adding $L2$ regularization to $L(y, \hat{y})$ we get a new loss function $L'(y, \hat{y}) = L(y, \hat{y}) + \lambda w^T w$, where λ is a hyperparameter. Briefly explain why $L2$ regularization causes weight decay. Hint: Compare the gradient descent updates to w for $L(y, \hat{y})$ and $L'(y, \hat{y})$. Your answer should fit in the given solution box.

L2 regularization causes weight decay since the weights are smaller in the new loss function $L'(y, \hat{y})$. The weights in the network are squared in $\lambda w^T w$ and added to the original loss function, thus with the smaller, more penalized weights there is weight decay.

3.4 Why CNNs? (3 points)

Give 2 reasons why using CNNs is better than using fully connected networks for image data.

1. CNNs make the image data more compact and thus reduce the number of parameters. Thus, the fully connected network used after convolution will have less parameters and perform better.
2. CNNs use convolution to preserve important structural information otherwise initially deleted in fully connected networks. These features will be used after convolution in the fully connected network which will improve the feature detection.