DEPARTMENT OF ECE, UCLA
ECE 188: THE FOUNDATIONS OF COMPUTER VISION

INSTRUCTOR: Prof. Achuta Kadambi
TA: Rishi Upadhyay

NAME: Krish Patel
UID: 605796227

HOMEWORK 4

PROBLEM	TYPE	TOPIC	MAX. POINTS
1	Analytical	Machine Learning Basics	10
2	Coding	Training a Classifier	15
3	Interview Questions (Bonus)	Miscellaneous	15
4	Analytical	Generative adversarial networks	5

Motivation

The problem set gives you a basic exposure to machine learning approaches and techniques used for computer vision tasks such as image classification. You will train a simple classifier network on CIFAR-10 dataset using google colab. We have provided pytorch code for the classification question, you are free to use any other framework if that's more comfortable.

The problem set consists of two types of problems:

- analytical questions to solidify the concepts covered in the class, and
- coding questions to provide a basic exposure to building a machine learning classifier using pytorch.

This problem set also exposes you to a variety of machine learning questions commonly asked in job/internship interviews

Homework Layout

The homework consists of 3 problems in total, with subparts for each problem. There are 2 types of problems in this homework - analytical and coding. All the problems need to be answered in the Overleaf document. Make a copy of the Overleaf project from here `https://www.overleaf.com/read/pqksjrzjrckj#e991de`, and fill in your answers for the questions in the solution boxes provided.

For the analytical questions you will be directly writing their answers in the space provided below the questions. For the coding problems you need to use the Jupyter notebooks from here: https://colab.research.google.com/drive/1ciRDVvVRyIx5c48yF21BWf-ttnOtx83k?usp=sharing(see the Jupyter notebook for each sub-part which involves coding). You are provided with 1 jupyter notebook for Problem 2. After writing your code in the Jupyter notebook you need to copy paste the same code in the space provided below that question on Overleaf. In some questions you are also required to copy the saved images (from Jupyter) into the solution boxes in Overleaf. For the classification question, upload the provided notebook to google colab, and change the runtime type to GPU for training the classifier on GPU. Refer to question 2 for more instructions/details.

Submission

Submission will be done via Gradescope. You will need to submit three things: (1) this PDF with answers filled out (from Overleaf), (2) Your .ipynb file, (3) a PDF printout of your .ipynb file with all cells executed. You do not need to create a folder, you will be able to upload all three files directly to gradescope.

Software Installation

You will need Jupyter to solve the homework. You may find these links helpful:

- Jupyter (https://jupyter.org/install)
- Anaconda (https://docs.anaconda.com/anaconda/install/)

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:

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

Taking ECE C147 currently (using matrix cookbook as well)

- When you have a function z defined as the inner product of a vector $\mathbf{x}$ with itself ($z = \mathbf{x}^T\mathbf{x}$), the derivative of z with respect to $\mathbf{x}$ $2\mathbf{x}$.
- If z is defined as the trace of the matrix $\mathbf{A}^T\mathbf{A}$, then the derivative of z with respect to $\mathbf{A}$ is twice the matrix $\mathbf{A}$.
- For $z = \mathbf{x}^T\mathbf{A}\mathbf{y}$, finding the partial derivative with respect to $\mathbf{y}$ results in the expression $\mathbf{A}^\mathbf{T}\mathbf{x}$.
- If $\mathbf{z} = \mathbf{A}\mathbf{y}$, the derivative of $\mathbf{z}$ with respect to $\mathbf{y}$ is simply the matrix $\mathbf{A}^T$.

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.

We know that for a Bernoulli distribution, the probability mass function (PMF) is given by:

$$p(y) = k^y(1-k)^{1-y}$$

Given that $\hat{y}$ is the output of the classifier, $p(\hat{y}|x)$ represents the probability of observing $\hat{y}$ given the input x. In binary classification, $\hat{y}$ can take values in the range [0, 1], so $p(y|x)$ is:

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

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

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

Taking the logarithm of both sides gives us the log-likelihood:

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

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

To obtain the loss function $L(y,\hat{y})$ from the log-likelihood $\log p(y|x)$, we simply negate the log-likelihood. Therefore, the loss function is given by:

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

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.

1. The high training accuracy and low test accuracy are because the model is overfitting(is much more complex than what would be considered ideal). By adding more training data, we would be able to decrease the ”effective complexity” of the model with respect to the training data, and would force the model to learn a more regularized and generalized decision boundary.
2. This would be a good regularization technique, as this penalizes high weights and thus encourages the weights to be smaller by including the l2 norm of the weights in the loss function.
3. This would increase the complexity of the model, increasing the model's ability to learn the noise of the training data, and instead will cause more overfitting.
4. This would be a valid way to solve overfitting, as validation sets are used for model selections and compare models. This way, it chooses a model based on ”validation” accuracy instead of training set 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.

The transforms.Normalize() function in PyTorch is used to normalize the pixel values of an image tensor. It subtracts the mean and divides by the standard deviation along each channel. To modify this function for grayscale images instead of RGB images, we need to calculate the mean and standard deviation across all channels (which is just one channel for grayscale images) and then pass these values accordingly.

```
 //printing num_test_samples and num_train_samples
print('Number of training samples:{}'.format(len( train_data )))
print('Number of test samples:{}'.format(len( test_data )))
```

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.

(i) **Convolutional Layer 1:**

Type: Convolutional (2D)
Input Channels: 3 (RGB channels)
Output Channels: 6
Kernel Size: 5x5
Stride: 1
Padding: 0
Activation Function: ReLU

Max Pooling Layer 1:

Type: Max Pooling (2D)
Kernel Size: 2x2
Stride: 2

Convolutional Layer 2:

Type: Convolutional (2D)
Input Channels: 6
Output Channels: 16
Kernel Size: 5x5
Stride: 1
Padding: 0
Activation Function: ReLU

Max Pooling Layer 2:

Type: Max Pooling (2D)
Kernel Size: 2x2
Stride: 2

FC Layer 1:

Type: Linear (Fully Connected)
Input Size: 16 * 5 * 5 (Output channels * Height * Width after pooling)
Output Size: 120
Activation Function: ReLU

FC Layer 2:

Type: Linear (Fully Connected)
Input Size: 120
Output Size: 84
Activation Function: ReLU

Output Layer:

Type: Linear (Fully Connected)
Input Size: 84
Output Size: 10 (for 10 different classes)
No Activation Function (or softmax probability)

(ii) It controls the amount of padding (zero padding) along the sides of the image. This preserves the dimensions of the image, and can be manipulated to change the dimensions of an image by certain pixel amounts.

(iii) The max pooling layer allows for the downsizing of the image by doing the following. It looks at all the values of the image within the kernel dimensions, and keeps only the one with the highest dimension. If max pooling had a kernel size of 2, it would reduce the dimensions of the output by 1/2 (on both width and height)

(iv) If we change the kernel size to 3 for the convolutional layers without adjusting other parameters, the receptive field of the filters would decrease, resulting in smaller feature maps after convolution and pooling operations. Thus, if we pass a test input through the network, it would not return an output of the same size as the input and would cause a dimensionality issue.

To maintain the output size, we would need to adjust the stride or padding parameters. Increasing the padding or decreasing the stride would ensure that the spatial dimensions of the feature maps remain the same, allowing the network to return an output of the same size as the input.

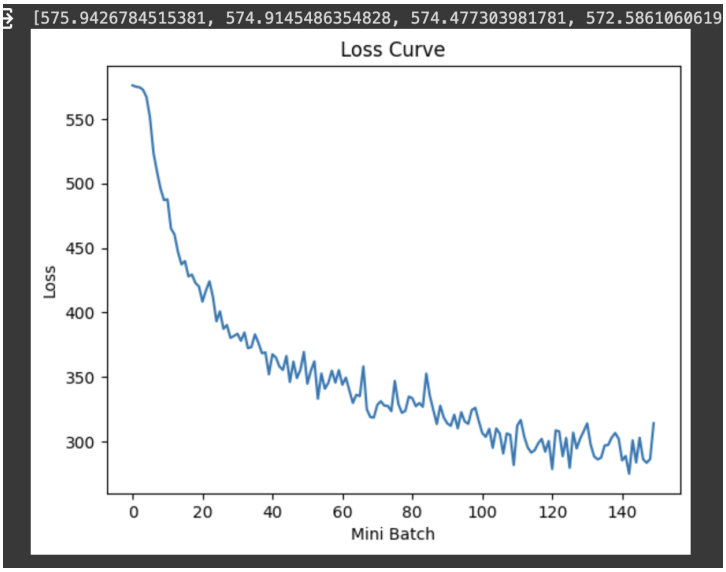
(v) You would not need to compute any additional gradients for max pooling, as it is non parametric and has no learnable parameters. Gradients are only passed to the positions of the largest values, and thus no need to compute gradients for this layer/operation.

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.

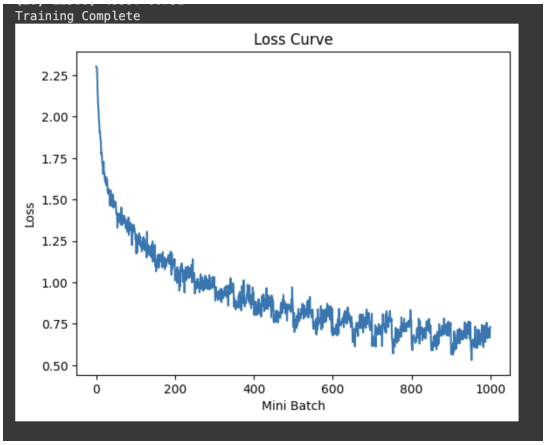
```
### Complete the code in the training box

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

net = Net().cuda()
criterion = nn.CrossEntropyLoss()
```

```
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)



num_epochs = 3
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.cuda())
        loss = criterion(outputs, labels.cuda())
        loss.backward()
        optimizer.step()
        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)

## complete the code to plot the running loss per 250 mini batches curve

def plot_loss_curve(running_loss_list):
    ## complete code
    plt.plot(running_loss_list)
    plt.xlabel('Mini Batch')
    plt.ylabel('Loss')
    plt.title('Loss Curve')
    plt.show()
plot_loss_curve
```



(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

```
for epoch in range(num_epochs):
    running_loss = 0.0
    for i, data in enumerate(train_loader, 0):
        inputs, labels = data
        inputs, labels = inputs.cuda(), labels.cuda()
        optimizer.zero_grad()
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
        if i % 250 == 249:
          print('[{},{}]loss:{:.3f}'.format(epoch+1,i+1,running_loss/250))
          running_loss_list.append(running_loss / 250)
          running_loss = 0.0
print('Training Complete')
```



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

During backpropagation, gradients are computed for each parameter of the neural network with respect to the loss function. If these gradients are not reset to zero before the next pass (iteration) of the network, they would accumulate across iterations.

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.

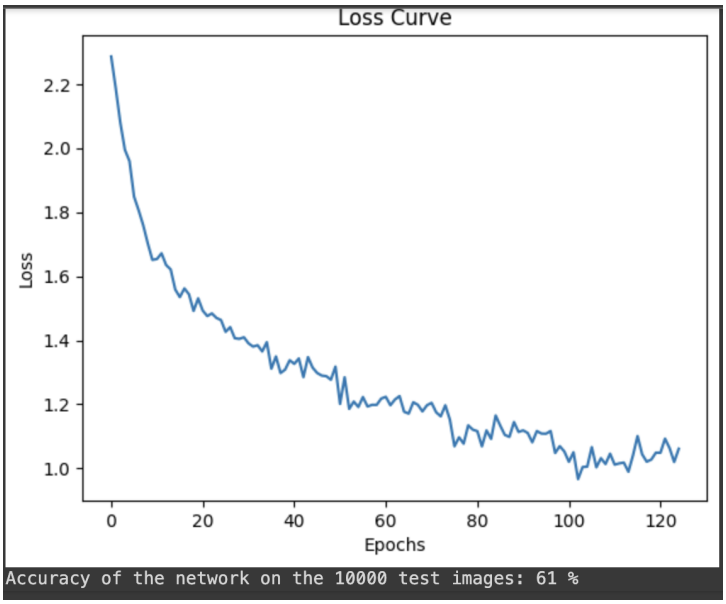
```
### Accuracy on whole data set
correct = 0
total = 0
with torch.no_grad():
    for data in test_loader:
        images, labels = data
        outputs = net(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
acc  =  100 * correct / total ## stores the accuracy computed in the above loop
print('Accuracy of the network on the 10000 test images: %d %%' % (acc))
```

```
Accuracy of the network on the 10000 test images: 61 %
```

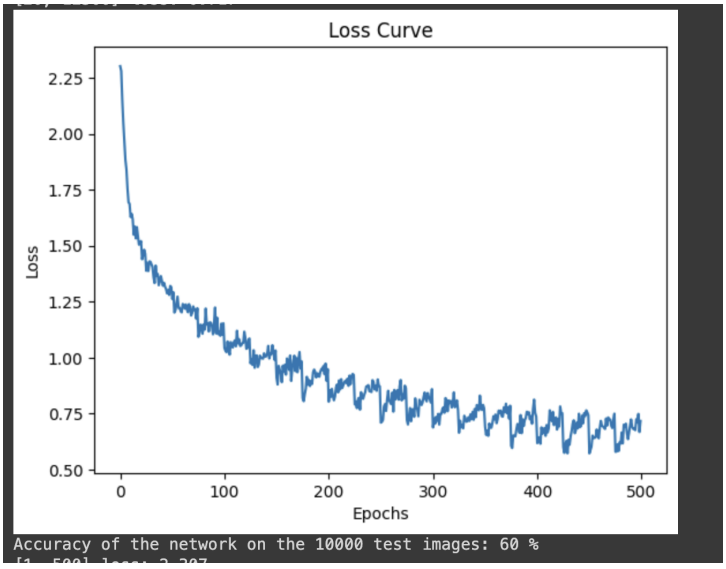
(ii) Train the network on the GPU with the following configurations, and report the testing accuracies and running loss curves -

- 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

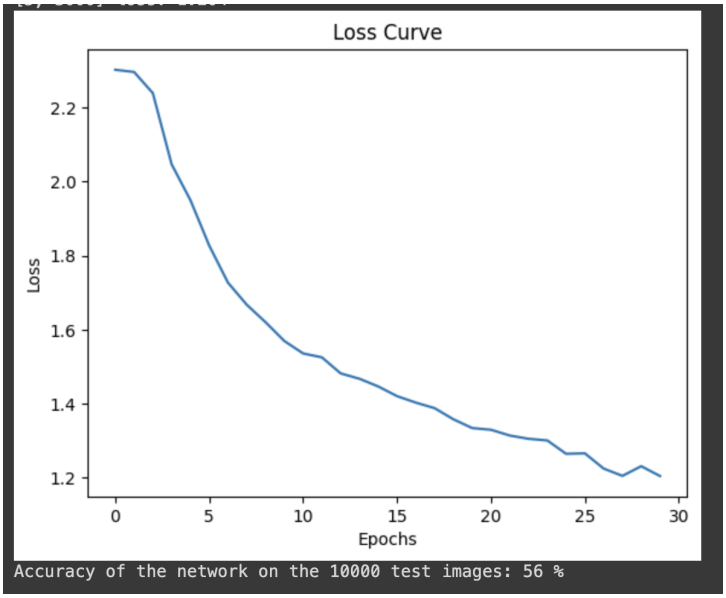
Batch size : 4, Epochs : 5



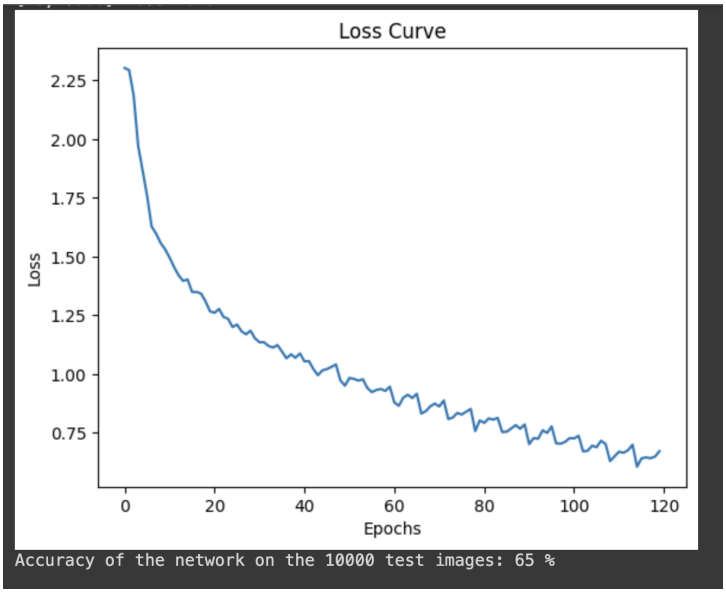
Batch size : 4, Epochs : 20



Batch size : 16, Epochs : 5



Batch size : 16, Epochs : 20



(iii) Explain your observations in (ii)

As we can see in the graphs above, having too many epochs might cause overfitting.This is visible when comparing the accuracies of training with 4 mini batches, using 5 and 20 epochs.

The one with 20 epochs gives a lower accuracy than the one with 5 epochs, even though it might do better in terms of training accuracy. However, a large batch size may also help improve the performance of the model(as the model with a batch size of 16 and a number of epochs of 20 performs the best out of all of these models, and these parameters can be chosen on the validation set before testing a model and implementing it to do real-world data predictions.

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 acts as a regularize as it helps stabilize and control the learning process of a neural network. It standardizes the inputs to each layer of the network, by normalizing the values of each feature in the input data to have a mean of zero and a standard deviation of one. This helps the model as it prevents the network from being too sensitive to small changes in the input data.

(ii) In training, we carry out batch normalization using statistics derived from the mini batches, however, when doing testing, we don't have batches. Thus, we just use the statistics of the training data to normalize batches for testing(inference) time.

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) The height (H) would be:

$$H = \frac{64 - 3 + 2 \times 0}{1} + 1 = 62$$

The width (W) would be:

$$W = \frac{64 - 3 + 2 \times 0}{1} + 1 = 62$$

And $C = C_{\text{out}} = 64$.

(ii) For the desired output size we solve for the following $32 = (64 - 3 + 2 \times P)/S + 1$, thus we have S = 3 and P = 16.

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 parameters/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 uses the l2 norm of the weights and adds it to the loss function. Thus, to minimize the loss function, we must also minimize the weights of the neural network.

The gradient for the neural network with the squared L2 norm would be:

$$w = w - \eta \nabla_w (L(y,\hat{y}) + \lambda w^T w)$$

Expanding the gradient descent update for $L(y,\hat{y}) + \lambda w^T w$, we get:

$$w = w - \eta \left(\nabla_w L(y,\hat{y}) + 2 * \lambda w\right)$$

Comparing this with the update for $L(y,\hat{y})$, we see that the term $2\lambda w$ causes the weights to decay. This term subtracts a fraction of the weight vector from itself at each update step, leading to weight decay and ultimately regularization.

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 use the spatial structure in images by using convolutional layers, which have shared weights across the input. This parameter sharing significantly reduces the number of parameters compared to fully connected networks, making CNNs more efficient in terms of memory usage, number of neurons, and computational time.

2. CNNs are capable of learning features that are invariant to translations in the input image, achieved through the use of conv. and pooling layers, which downsamples the input spatially and build off previous convolution layer's outputs. As a result, CNNs are better off for tasks where the spatial location of features isn't that important, such as object recognition in images.

4 GAN (Bonus) (5.0 points)

4.1 Understanding GANs- Loss function (2.0 points)

Mathematically express the overall GAN loss function being used. For a (theoretically) optimally trained GAN: (a) what is the ideal behavior of the discriminator, and (b) what is the value of the overall loss function?

The loss function in GANs is essentially like a performance metric, indicating how well the model is doing. It comprises two main components: one for the discriminator and one for the generator.

The discriminator's role is to distinguish between real and fake data. Its loss function assesses how accurately it identifies real data as real and fake data as fake. On the other hand, the generator aims to produce fake data that resembles real data. Its loss function measures how effectively it fools the discriminator into classifying its fake data as real.

In an ideal scenario: 1. The discriminator should excel at distinguishing real from fake, correctly identifying real data as real and fake data as fake. 2. The overall loss function decreases to its minimum value. This indicates that the generator successfully generates fake data that closely resembles real data, making it difficult for the discriminator to differentiate between the two.

In essence, the objective of a GAN is to generate synthetic data that is indistinguishable from real data, which should be more challenging for a discriminating adversary.

The overall loss function in a GAN is the sum of the discriminator loss (L_{D}) and the generator loss (L_{G}). Thus, the formula for the overall loss function (L_{GAN}) is:

$$L_{\text{GAN}} = L_{\text{D}} + L_{\text{G}}$$

4.2 Understanding GANs- Gradients (2.0 points)

Assume that you are working with a GAN having the following architecture:

Generator: Input: $\mathbf{x}$ shape $(2,1)$ $\rightarrow$ Layer: $\mathbf{W_g}$ shape $(5,2)$ $\rightarrow$ ReLU $\rightarrow$ Output: $\mathbf{y}$ shape $(5,1)$
Discriminator: Input: $\mathbf{z}$ shape $(5,1)$ $\rightarrow$ Layer: $\mathbf{W_d}$ shape $(1,5)$ $\rightarrow$ Sigmoid $\rightarrow$ Output: b shape $(1,1)$

Therefore, the generator output is given by, $\mathbf{y} = ReLU(\mathbf{W_g x})$, and the discriminator output is given by $b = Sigmoid(\mathbf{W_d z})$. Express the gradient of the GAN loss function, with respect to the weight matrices for the generator and discriminator.

You may use the following information:

$$Sigmoid(x) = \frac{1}{1+e^{-x}}$$

$$ReLU(x) = \begin{cases} x, & \text{if } x \geq 0 \\ 0, & \text{if } x < 0 \end{cases}$$

Hint: Remember that the input here is a vector, not an image.

4.3 Understanding GANs- Input distributions (1.0 points)

While training the GAN, the input is drawn from a normal distribution. Suppose in a hypothetical setting, each time the input is chosen from a different, randomly chosen probability distribution. How would this affect the training of the GAN? Justify your answer mathematically.

During GAN training, drawing input from a normal distribution provides a stable/consistent training signal, allowing for effective learning under underlying data distribution. However, if the input is randomly chosen from various probability distributions, it increased in inconsistency and instability, preventing convergence and disrupting the training. This ”variability” in input distributions could hinder the models' ability to capture the underlying data distribution accurately, reducing the effectiveness of GAN training.