Assignment Answer

Assignment GIven By: Anand Mishra

Submission By: Karan Arora (M22AIE235)

Fractal-3 Assignment

April 30, 2023

Question 1: Perceptron [30 points]

1. In how many steps the perception learning algorithm will converge.

2. What will be the final decision boundary? Show step-wise-step update of weight vector using computation as well as hand-drawn plot.

Answer 1:

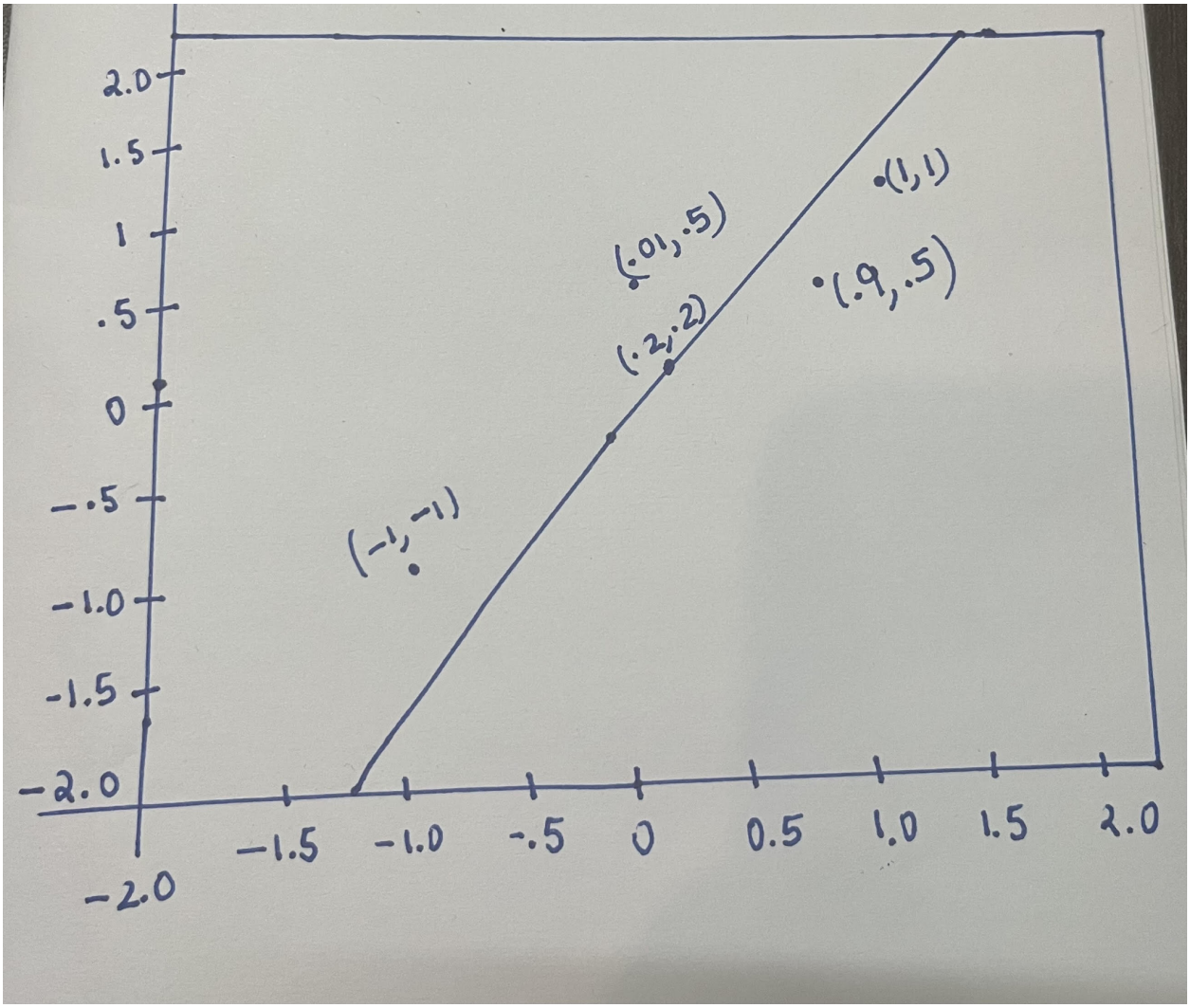
a) Total Steps needed by the perception learning algorithm to converge is 7

CODE to the question =

```
import numpy as np
given_value =
  np.array([[1,1],[-1,-1],[0,0.5],[0.1,0.5],[0.2,0.2],[0.9,0.5]]) #this
  array is given in the question paper
y=[1,-1,-1,-1,1,1]
op_value = 0
w = [1,1] # As mentioned in the question we are initializing the
  weights as [1,1]
w = np.append(w,1) #appending our weight of 0 to 1
print(w)
print("conversion start...")
steps = 0
while (op_value != len(given_value)):
  steps = steps +1
  for value_taken_for_ref in range(len(given_value)):
    x = np.append(given_value[value_taken_for_ref,0:2],1)
    if y[value_taken_for_ref]==1:
      if np.dot(np.transpose(w),x)>=0:
        op_value=op_value+1
      else:
        w=w+x
    else:
      if np.dot(np.transpose(w),x)<0:
        op_value=op_value+1
      else:
        w=w-x
  if(op_value != len(given_value)):
    op_value=0
print("convergence steps: "+ str(steps-1))
print(w)
```

```
OUTPUT:
The algorithm took 3 steps to converge
convergence steps: 3
[ 1.4 -0.4  0. ]
Weights finally allocated were 1.4 and -0.4 to the algo.
```

B) Final Decision Boundary is as follows.



Problem 2: Learning to implement Neural Network [30 points]

1. Gurmukhi Handwritten Digit Classification: Gurmukhi is one of the popular Indian scripts widely used in the Indian state of Punjab. In this part of the assignment, our goal is to develop a neural network solution (a simple NN, not a CNN) for classifying Gurmukhi digits. We provide you Handwritten Gurmukhi digit dataset here:

Dataset link

Modify the code provided in here and a video tutorial here, and develop a robust neural network to classify the Gurmukhi digits. Higher performance on the test set will have bonus points. Briefly write your observation and submit your code so that we can evaluate your implementation at our end. (10 points)

ANSWER 2 is IMPLEMENTED COMPLETELY IN THE NOTEBOOK ATTACHED

Problem 3: Chart Image Classification using CNN [40 points]

Problem statement: You have to develop a CNN-based classification architecture for clas sifying a given chart image to one of five chart classes, namely “Line”,“Dot Line”,“Horizontal Bar”,“Vertical Bar”, and “Pie” chart.
Task 1: Download the dataset from drive link given below.
Dataset link

Use the train and val images for training and validation in an appropriate ratio (e.g., 80% for training and 20 % for validating). The CSV file contains corresponding labels for the images.

Here is the Answer Code:

```
# importing the libraries

import pandas as pd

import numpy as np
```

```
# for reading and displaying images
from skimage.io import imread
import matplotlib.pyplot as plt
from sklearn import preprocessing

# for creating validation set
from sklearn.model_selection import train_test_split

# for evaluating the model
from sklearn.metrics import accuracy_score
from tqdm import tqdm

# PyTorch libraries and modules
import torch
from torch.autograd import Variable
from torch.nn import Linear, ReLU, CrossEntropyLoss, Sequential, Conv2d,
MaxPool2d, Module, BatchNorm2d
from torch.optim import Adam

# loading dataset
```

```
train = pd.read_csv('Chart/train_val.csv')

test = pd.read_csv('Chart/test.csv')


sample_submission = pd.read_csv('Chart/prediction.csv')


train.head()


# loading training images

train_img = []

for img_name in tqdm(train['image_index']):

    # defining the image path

    image_path = 'Chart/train_val/' + str(img_name) + '.png'

    # reading the image

    img = imread(image_path, as_gray=True)

    # normalizing the pixel values

    img /= 255.0

    # converting the type of pixel to float 32

    img = img.astype('float32')

    # appending the image into the list

    train_img.append(img)
```

```
# converting the list to numpy array

train_x = np.array(train_img)

# defining the target

train_y = train['type'].values

train_x.shape



# create validation set

train_x, val_x, train_y, val_y = train_test_split(train_x, train_y,
test_size = 0.2)

(train_x.shape, train_y.shape), (val_x.shape, val_y.shape)



encoder = preprocessing.LabelEncoder()



# converting training images into torch format

train_x = train_x.reshape(800, 1, 128, 128)

train_x  = torch.from_numpy(train_x)



# converting the target into torch format

train_y = encoder.fit_transform(train_y)



train_y = train_y.astype(int);
```

```
train_y = torch.from_numpy(train_y)


# shape of training data

train_x.shape, train_y.shape


# converting validation images into torch format

val_x = val_x.reshape(200, 1, 128, 128)

val_x  = torch.from_numpy(val_x)


# converting the target into torch format

val_y = encoder.fit_transform(val_y)


val_y = val_y.astype(int);

val_y = torch.from_numpy(val_y)


# shape of validation data

val_x.shape, val_y.shape


class Net(Module):

    def __init__(self):

        super(Net, self).__init__()
```

```
        self.cnn_layers = Sequential(
            # Defining a 2D convolution layer
            Conv2d(1, 4, kernel_size=3, stride=1, padding=1),
            BatchNorm2d(4),
            ReLU(inplace=True),
            MaxPool2d(kernel_size=2, stride=2),
            # Defining another 2D convolution layer
            Conv2d(4, 4, kernel_size=3, stride=1, padding=1),
            BatchNorm2d(4),
            ReLU(inplace=True),
            MaxPool2d(kernel_size=2, stride=2),
        )

        self.linear_layers = Sequential(
            Linear(4 * 32 * 32, 10)
        )

    # Defining the forward pass
    def forward(self, x):
        x = self.cnn_layers(x)
```

```
        x = x.view(x.size(0), -1)
        x = self.linear_layers(x)
        return x


# defining the model
model = Net()
# defining the optimizer
optimizer = Adam(model.parameters(), lr=0.07)
# defining the loss function
criterion = CrossEntropyLoss()
# checking if GPU is available
if torch.cuda.is_available():
    model = model.cuda()
    criterion = criterion.cuda()


def train(epoch):
    model.train()
    tr_loss = 0
    # getting the training set
    x_train, y_train = Variable(train_x), Variable(train_y)
```

```
y_train = y_train.long()


# getting the validation set
x_val, y_val = Variable(val_x), Variable(val_y)
y_val = y_val.long()
# converting the data into GPU format
if torch.cuda.is_available():
    x_train = x_train.cuda()
    y_train = y_train.cuda()
    x_val = x_val.cuda()
    y_val = y_val.cuda()


# clearing the Gradients of the model parameters
optimizer.zero_grad()


# prediction for training and validation set
output_train = model(x_train)
output_val = model(x_val)


# computing the training and validation loss
loss_train = criterion(output_train, y_train)
```

```
    loss_val = criterion(output_val, y_val)
    train_losses.append(loss_train)
    val_losses.append(loss_val)

    # computing the updated weights of all the model parameters
    loss_train.backward()
    optimizer.step()

# defining the number of epochs
n_epochs = 50
# empty list to store training losses
train_losses = []
# empty list to store validation losses
val_losses = []
# training the model
for epoch in range(n_epochs):
    train(epoch)

# plotting the training and validation loss
with torch.no_grad():
    plt.plot(train_losses, label='Training loss')
```

```
    plt.plot(val_losses, label='Validation loss')
    plt.legend()
    plt.show()


# prediction for training set
with torch.no_grad():
    output = model(train_x)


softmax = torch.exp(output).cpu()
prob = list(softmax.numpy())
predictions = np.argmax(prob, axis=1)


# accuracy on training set
accuracy_score(train_y, predictions)


# prediction for validation set
with torch.no_grad():
    output = model(val_x)


softmax = torch.exp(output).cpu()
prob = list(softmax.numpy())
```

```
predictions = np.argmax(prob, axis=1)


# accuracy on validation set

accuracy_score(val_y, predictions)


# loading test images

test_img = []

for img_name in tqdm(test['image_index']):

    # defining the image path

    image_path = 'Chart/test/' + str(img_name) + '.png'

    # reading the image

    img = imread(image_path, as_gray=True)

    # normalizing the pixel values

    img /= 255.0

    # converting the type of pixel to float 32

    img = img.astype('float32')

    # appending the image into the list

    test_img.append(img)


# converting the list to numpy array

test_x = np.array(test_img)
```

```
test_x.shape


# converting training images into torch format

test_x = test_x.reshape(50, 1, 128, 128)

test_x  = torch.from_numpy(test_x)

test_x.shape


# generating predictions for test set

with torch.no_grad():

    output = model(test_x)


softmax = torch.exp(output).cpu()

prob = list(softmax.numpy())

predictions = np.argmax(prob, axis=1)


predictions = encoder.inverse_transform(predictions)


# replacing the label with prediction

sample_submission['type'] = predictions

sample_submission.head()
```

```
# saving the file

sample_submission.to_csv('Chart/prediction.csv', index=False)
```

Task 2: Implement a two-layer Convolutional Neural Network, and calculate accuracy, and loss and plot the obtained loss. Briefly write your observation and submit your code so that we can4 evaluate your implementation at our end.
Task 3: Finetune a pretrained network (e.g., AlexNet) for this task and report the results.

Problem 4: Gradient Descent and Backprop [10 points]

1. What is the difference between Stochastic Gradient Descent and Mini Batch Gradient Descent? *(2 points)*

2. Consider a 3-layer network shown in Figure 1:
   Given that $f = w^{[3]}_1 a^{[2]}_1 + w^{[3]}_2 a^{[2]}_2$.
   Compute the following derivatives: $\frac{\delta f}{\delta Z^{[2]}}, \frac{\delta f}{\delta Z^{[1]}}, \frac{\delta f}{\delta z^{[2]}_1}, \frac{\delta f}{\delta w_{11}}$. *(8 points)*

ANSWER 1: Stochastic Gradient Descent: What is stochastic gradient descent (or SGD, for short)? SGD is a variant of the optimization algorithm that saves us both time and computing space while still looking for the best optimal solution. In SGD, the dataset is properly shuffled to avoid pre-existing orders then partitioned into m examples. This way the stochastic gradient descent python algorithm can then randomly pick each example of the dataset per iteration (as opposed to going through the entire dataset at once). A stochastic gradient descent example will only use one example of the training set for each iteration. And by doing so, this random approximation of the data set removes the computational burden associated with gradient descent while achieving iteration faster and at a lower convergence rate. The process simply takes one random stochastic gradient descent example, iterates, then improves before moving to the next random example. However, because it takes and iterates one example

at a time, it tends to result in more noise than we would normally like.

3. *Mini-Batch Gradient Descent: A mini-batch gradient descent is what we call the bridge between the batch gradient descent and the stochastic gradient descent. The whole point is like keeping gradient descent to stochastic gradient descent side by side, taking the best parts of both worlds, and turning it into an awesome algorithm. So, while in batch gradient descent we have to run through the entire training set in each iteration and then take one example at a time in stochastic, mini-batch gradient descent simply splits the dataset into tiny batches. Hence, it is not running through the entire sample at once, neither is it taking one example at a time. This creates some sort of balance in the algorithm where we can find both the robustness of stochastic and the computational efficiency of batch gradient descent.*

ANSWER B: Consider following 3-layer network

x1 and *x2* are input nodes

x1 and *x2* represent the nodes in Hidden Layer 1

a1(1) and *a2(1)* represent the activation of the nodes in Hidden Layer 1

z1(1) , *z2(1)* , *z1(2)* and *z2(2)* represent the weighted sums in Hidden Layers 1 and 2

wij(k) represents the weight connecting node i in layer k-1 to node j in layer k

f is the output node

$Z1 = z11 \;\; z21 = w111 \;\; w121 \;\; w211 \;\; w221 \;\; x1\; x2$, $A1 = a11 \;\; a21 = z11 \;\; z21$

$Z[2] = z12 \;\; z22 = w112 \;\; w122 \;\; w212 \;\; w222 \;\; a1[1]\; a2[1]$, $A2 = a12 \;\; a22 = z12 \;\; z22$

Given that $f = w1[3]\; a1[2] + w2[3]\; a2[2]$. Compute the following derivatives: $\delta fz1[2]$, $\delta fz[2]$, $\delta fz[1]$, $\delta fw11$

a

Given the 3-layer network as described, to compute the derivative $\delta fz1[2]$, we will use the chain rule.

We have $f = w1[3]\; a1[2] + w2[3]\; a2[2]$.

Let's assume the activation function for the hidden layers is $\sigma(x)$. Then,

$a1([2]) = \sigma(z1(2))$

Now, we need to compute the derivative δf/δz1(2)

*δf/δz1(2) = (δf/δa1([2])) * (δa1([2]))/δz1(2))*

Since f is a function of a1([2]), we have:

δf/δa1([2]) = w13

Next, we compute the derivative of the activation function σ(x) with respect to z1(2)

δa1([2]) /δz1(2) = σ'(z1(2))

Now we can put everything together:

*δf/δz1(2) = w13 * σ'(z1(2))*

This derivative represents the rate of change of the output f with respect to the weighted sum z1(2) in the second hidden layer.

b

Given the 3-layer network as described, to compute the derivative δfZ[2], we will use the chain rule. Since z([2]) is a vector with components z1(2) and z2(2) , the derivative δf/δz([2]) will also be a vector with corresponding components.

We have f = w

...

Problem 5: Neural network in practice [10 points] 1. What's the

risk of tuning hyperparameters using a test dataset? *(2 points)* 2. Give two strategies for addressing the overfitting problem in neural networks.*(2 points)*

3. How do you decide input layer and output layer size for solving a particular problem using a neural network?*(2 points)*
4. Explain the Sigmoid activation function.*(2 points)*

ANSWER 5.

1. The model will not generalize for invisible data because it fits the test set too well.

Tuning the model hyperparameters for a test means that the hyperparameters may be too good for testing. If the same test is used to estimate performance, it will lead to overestimation. The measuring system is used only for testing, not for correction.

Using a separate validation set for tuning and a test set for performance evaluation provides an unbiased, accurate measure of performance. finally determines the values of the model parameters of the learning algorithm. to work. We cannot calculate their value from the data.Hyperparameter tuning is the process of selecting a combination of hyperparameters to optimize model performance. It works by running multiple tests in a single training session. All tests are performed on your training application with the hyperparameter values you choose, specified within the parameters you specify. After this process is complete, you will be given a set of hyperparameter values that match your model to ensure the best results.

2. -Reduce the network's capacity by removing layers or reducing the number of elements in the hidden layers.

-Apply regularization, which comes down to adding a cost to the loss function for large weights.

-Use Dropout layers, which will randomly remove certain features by setting them to zero.

3. There are many methods for determining the correct number of neurons to use in the hidden layers, such as the following:

A. The number of hidden neurons should be between the size of the input layer and the size of the output layer.

B. The number of hidden neurons should be 2/3 the size of the input layer, plus the size of the output layer.

C. The number of hidden neurons should be less than twice the size of the input layer.

Moreover, the number of neurons and number of layers required for the hidden layer also depends upon training cases, amount of outliers, the complexity of, the data that is to be learned, and the type of activation functions used.

Most of the problems can be solved by using a single hidden layer with the number of neurons equal to the mean of the input and output layers. If fewer neurons are chosen it will lead to underfitting and high statistical bias. Whereas if we choose too many neurons it may lead to overfitting, high variance, and increases the time it takes to train the network.

4. The sigmoid function is mathematically defined as 1/(1+e^(-x)) where x is the input value and e is the mathematical constant 2,718. This function determines the input value between 0 and 1, making it suitable for binary distribution and logistic regression problems. The range of the function is (0,1) and its definition is (-infinity, + infinity).

One of the important features of the sigmoid function is the "S" shape.

As the input value increases, the output value starts to increase gradually, then quickly reaches 1 and finally stabilizes. This tool provides a non-trivial function for modeling decision boundaries in binary classification problems.

Another feature of sigmoid is its output, which is often used to train neural networks. The derivative of a function is defined as f(x)(1-f(x)) where f(x) is the output of the function. Derivatives are useful in training neural networks as they allow the network to further adjust neuron weights and biases.

It is also worth noting that the sigmoid function has some limitations. For example the output of sigmoid is always between 0 and 1, which causes problems when the output of the network has to be greater than or less than 0. Other functions such as ReLU and tanh can be used for this.

5. Learning rate is a hyper-parameter that controls the weights of our neural network with respect to the loss gradient. It defines how quickly the neural network updates the concepts it has learned.