CS640 Homework 2: Neural Network

In this assignment, you will

1. derive both forward and backward propagation,
2. implement a neural network from scratch, and
3. run experiments with your model.

Collaboration

You are allowed to work in a team of at most **three** on the coding part(**Q2**), but you must run the experiments and answer written questions independently.

Instructions

General Instructions

In an ipython notebook, to run code in a cell or to render Markdown+LaTeX press `Ctrl+Enter` or `[>|]` (like "play") button above. To edit any code or text cell (double) click on its content. To change cell type, choose "Markdown" or "Code" in the drop-down menu above.

Most of the written questions are followed up a cell for you enter your answers. Please enter your answers in a new line below the **Answer** mark. If you do not see such cell, please insert one by yourself. Your answers and the questions should **not** be in the same cell.

Instructions on Math

Some questions require you to enter math expressions. To enter your solutions, put down your derivations into the corresponding cells below using LaTeX. Show all steps when proving statements. If you are not familiar with LaTeX, you should look at some tutorials and at the examples listed below between $..$. The OEIS website can also be helpful.

Alternatively, you can scan your work from paper and insert the image(s) in a text cell.

Submission

Once you are ready, save the note book as PDF file (File -> Print -> Save as PDF) and submit via Gradescope.

Q0: Name(s)

Please write your name in the next cell. If you are collaborating with someone, please list their names as well.

Answer

Q1: Written Problems

Consider a simple neural network with three layers: an input layer, a hidden layer, and an output layer.

Let $w^{(1)}$ and $w^{(2)}$ be the layers' weight matrices and let $b^{(1)}$ and $b^{(2)}$ be their biases. For convention, suppose that w_{ij} is the weight between the ith node in the previous layer and the jth node in the current one.

Additionally, the activation function for both layers is the sigmoid function $\sigma(x) = \frac{1}{1+e^{-x}}$. Let $z^{(1)}$ and $z^{(2)}$ be the outputs of the two layers before activation, and let $a^{(1)} = \sigma(z^{(1)})$ and $a^{(2)} = \sigma(z^{(2)})$.

Lastly, we choose the L2 loss $L(y_{\text{true}}, y_{\text{predict}}) = \frac{1}{2}(y_{\text{true}} - y_{\text{predict}})^2$ as the loss function.

Q1.1: Forward Pass

Suppose that

$w^{(1)} = \begin{bmatrix} 0.4 & 0.6 & 0.2 \\ 0.3 & 0.9 & 0.5 \end{bmatrix}, b^{(1)} = [1, 1, 1]$; and

$w^{(2)} = \begin{bmatrix} 0.2 \\ 0.2 \\ 0.8 \end{bmatrix}, b^{(2)} = [0.5]$.

If the input is $a^{(0)} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$, what is the network output? Show your calculation steps and round your answer to 4 decimals.

[Answer]

$$z^{(1)} = {w^{(1)}}^T a^{(0)} + {b^{(1)}}^T$$
$$a^{(1)} = \sigma(z^{(1)})$$
$$z^{(2)} = {w^{(2)}}^T a^{(1)} + {b^{(2)}}^T$$
$$a^{(2)} = \sigma(z^{(2)})$$

After plugging in numbers and rounding, the final answer is 0.8221.

Show code

Q1.2: Backward Propagation

Use the chain rule to derive the expressions of the following gradients:

1. $\frac{\partial L}{\partial w^{(2)}}$ and $\frac{\partial L}{\partial b^{(2)}}$
2. $\frac{\partial L}{\partial w^{(1)}}$ and $\frac{\partial L}{\partial b^{(1)}}$

Your final answers should only include the variables appeared in the question.

Hint #1: Begin by writing down the chain of partial derivatives, and then plug in predefined variables.

Hint #2: While plugging in predefined variables, be careful about the dimensions and orientation. You can first write down the expressions in the element level and then figure out the matrix form.

Hint #3: The derivative of $\sigma(x)$ is $\sigma(x)(1-\sigma(x))$.

Hint #4: The LaTex code for dot product and element-wise product: $\cdot$ and $\odot$.

[Answer]

Second layer:

$$\begin{aligned}\frac{\partial L}{\partial w_i^{(2)}} &= \frac{\partial L}{\partial a^{(2)}} \cdot \frac{\partial a^{(2)}}{\partial z^{(2)}} \cdot \frac{\partial z^{(2)}}{\partial w_i^{(2)}} \\ &= L'(y_{\text{true}}, a^{(2)}) \cdot f_2'(z^{(2)}) \cdot a_i^{(1)} \\ &= (a^{(2)} - y_{\text{true}})\sigma(z^{(2)})(1-\sigma(z^{(2)}))a_i^{(1)}\end{aligned}$$

$$\begin{aligned}\frac{\partial L}{\partial b^{(2)}} &= \frac{\partial L}{\partial a^{(2)}} \cdot \frac{\partial a^{(2)}}{\partial z^{(2)}} \cdot \frac{\partial z^{(2)}}{\partial b^{(2)}} \\ &= L'(y_{\text{true}}, a^{(2)}) \cdot f_2'(z^{(2)}) \cdot 1 \\ &= (a^{(2)} - y_{\text{true}})\sigma(z^{(2)})(1-\sigma(z^{(2)}))\end{aligned}$$

Then, in matrix form

$$\frac{\partial L}{\partial w^{(2)}} = a^{(1)} \cdot ((a^{(2)} - y_{\text{true}}) \odot (\sigma(z^{(2)})(1-\sigma(z^{(2)}))))^T.$$
$$\frac{\partial L}{\partial b^{(2)}} = ((a^{(2)} - y_{\text{true}}) \odot (\sigma(z^{(2)})(1-\sigma(z^{(2)}))))^T.$$

Let

$$\eta = (a^{(2)} - y_{\text{true}}) \odot (\sigma(z^{(2)})(1-\sigma(z^{(2)}))).$$

First layer:

$$\begin{aligned}\frac{\partial L}{\partial w_{ij}^{(1)}} &= \frac{\partial L}{\partial a^{(2)}} \cdot \frac{\partial a^{(2)}}{\partial z^{(2)}} \cdot \frac{\partial z^{(2)}}{\partial a_j^{(1)}} \cdot \frac{\partial a_j^{(1)}}{\partial z_j^{(1)}} \cdot \frac{\partial z_j^{(1)}}{\partial w_{ij}^{(1)}} \\ &= L'(y_{\text{true}}, a^{(2)}) \cdot f_2'(z^{(2)}) \cdot w_j^{(2)} \cdot f_1'(z^{(1)}) \cdot a_i^{(0)}\end{aligned}$$

$$\frac{\partial L}{\partial w_{ij}^{(1)}} = \frac{\partial L}{\partial a^{(2)}} \cdot \frac{\partial a^{(2)}}{\partial z^{(2)}} \cdot \frac{\partial z^{(2)}}{\partial a_j^{(1)}} \cdot \frac{\partial a_j^{(1)}}{\partial z_j^{(1)}} \cdot \frac{\partial z_j^{(1)}}{\partial w_{ij}^{(1)}}$$
$$= L'(y_{\text{true}}, a^{(2)}) \cdot f_2'(z^{(2)}) \cdot w_j^{(2)} \cdot f_1'(z^{(1)})$$

Then, in matrix form

$$\frac{\partial L}{\partial w^{(1)}} = a^{(0)} \cdot (w^{(2)} \cdot \eta \odot f_1'(z^{(1)}))^T$$
$$= a^{(0)} \cdot (w^{(2)} \cdot \eta \odot (\sigma(z^{(1)}) \odot (1 - \sigma(z^{(1)}))))^T$$
$$\frac{\partial L}{\partial b^{(1)}} = (w^{(2)} \cdot \eta \odot f_1'(z^{(1)}))^T$$
$$= (w^{(2)} \cdot \eta \odot (\sigma(z^{(1)}) \odot (1 - \sigma(z^{(1)}))))^T$$

Q2: Implementation

In this part, you need to construct a neural network model (almost) from scratch, run experiments, and write reports. We provide a script of skeleton code as well as three datasets.

Your tasks are the following.

1. Build your network model following the instruction.
2. Run experiments and produce results.
3. Interpret and discuss your results.

Q2.1: Import Packages

The packages that have been imported in the following block should be sufficient for this assignment, but you are free to add more if necessary. However, keep in mind that you **should not** import and use any neural network package. If you have concern about an addition package, please contact us via Piazza.

```
import numpy as np
from sklearn.model_selection import StratifiedKFold
from sklearn.metrics import confusion_matrix, accuracy_score, precision_score, recall_score, f1_score
import matplotlib.pyplot as plt
from matplotlib.ticker import MaxNLocator
import pandas
```

Q2.2: Define Activation and Loss Functions

Complete the following functions. The ones starting with a "d" are the derivatives of the corresponding functions.

Definitions:

1. sigmoid: $\sigma(x) = \frac{1}{1+e^{-x}}$
2. softmax: softmax(x) $= \frac{e^{x_i}}{\sum_i e^{x_i}}$
3. L2 loss: $L(y_{\text{true}}, y_{\text{predict}}) = \frac{1}{2}(y_{\text{true}} - y_{\text{predict}})^2$
4. cross entropy loss: $L(y_{\text{true}}, y_{\text{predict}}) = -\sum_i y_{\text{true}}[i] \cdot \log y_{\text{predict}}[i]$

```
def sigmoid(x):
    pass

def d_sigmoid(x):
    pass

def softmax(x):
    pass

def l2_loss(YTrue, YPredict):
    pass

def d_l2_loss(YTrue, YPredict):
    pass

def cross_entropy_loss(YTrue, YPredict):
    pass
```

```
18
19 def d_cross_entropy_softmax(YTrue, YPredict):
20     pass
```

Q2.3: Define the Layer Class

Complete the `initialize_weights` function, which initializes the weights and biases with small random values. The `__init__` function should be left as it is.

Hint: It is recommended that you define weights and bias separately for clarity.

```
 1 class Layer:
 2     def __init__(self, n_input, n_output, bias = True):
 3         self.n_input = n_input
 4         self.n_output = n_output
 5         self.bias = bias
 6         self.initialize_weights()
 7
 8     def initialize_weights(self):
 9         """
10         Initializes the weights and biases with small random values.
11         """
12         rng = np.random.default_rng(2) # for re-producibility, do not change this
13         ########################## start of your code ##########################
14
15         ########################## end of your code ############################
16
```

Q2.4: Define the Network Class

Complete the `fit` and `predict` functions as instructed in the comments. Do not change their input arguments, but you are free to add functions as necessary. The `__init__` function should be left as it is.

Hint #1: This is the heaviest part of this assignment. We recommend you to first go over the math carefully before starting this part.

Hint #2: You are strongly encouraged to use numpy for matrix operations. When doing multiplication, please be careful about the dimensions, as well as the difference between the "*" operator, numpy's `multiply` function, and numpy's `dot` function.

```
 1 class Network:
 2     def __init__(self, layers, activation_list, d_activation_list, loss_function, d_loss_function):
 3         self.layers = layers
 4         self.activation_list = activation_list
 5         self.d_activation_list = d_activation_list
 6         self.loss_function = loss_function
 7         self.d_loss_function = d_loss_function
 8
 9     def fit(self, X, Y, learning_rate, reg_lambda):
10         """
11         This is the training function. It should return the average loss over samples.
12         """
13         loss, n_sample = 0, len(X)
14
15         ########################## start of your code ##########################
16         # first, initialize zero gradients
17
18
19         # next, for each sample,
20         # 1. compute outputs from each layer (via some forward function);
21         # 2. compute and accumulate the loss (via the self.loss_function); and
22         # 3. compute and accumulate the gradients (via some backprog function)
23
24
25         # then, update weights and biases using the corresponding gradients
26         # don't forget to take the mean before updating
27
28         ########################## end of your code ############################
29
30         # lastly, return the average loss
31         return loss / n_sample
32
33     def predict(self, X, threshold = None):
34         """
35         This function predicts the labels for samples in X. The parameter threshold
```

```
        is used when the labels are binary and there is only one node in the final
        layer of the network.
        """
        YPredict = []

        ########################## start of your code ##########################
        # for each sample, run a forward pass and append the predicted label to YPredict

        ########################## end of your code ############################

        # return as a numpy array
        return np.array(YPredict)

```

Q2.5: Test Model

Use the following example code to test your model with some simple data.

Make sure to produce a decreasing loss curve here before moving on.

```
from sklearn import datasets

X, Y = datasets.load_iris(return_X_y = True)
X, Y = X[:100, :2], Y[:100]
rng = np.random.default_rng(2)
indices = [i for i in range(100)]
rng.shuffle(indices)
X, Y = X[indices], Y[indices]


# assemble your model
layers = [Layer(2, 4), Layer(4, 1)]
model = Network(layers, [sigmoid, sigmoid], [d_sigmoid, d_sigmoid], l2_loss, d_l2_loss)

# specify training parameters
epochs = 100
learning_rate = 1e-2
reg_lambda = 0

# capture the loss values during training
loss = np.zeros(epochs)

# start training
for epoch in range(epochs):
    loss[epoch] = model.fit(X, Y, learning_rate, reg_lambda)

# plot the losses, the curve should be decreasing
plt.plot([i for i in range(epochs)], loss)
plt.title("Training Loss")
plt.xlabel("Epoch")
plt.show()

```

Q3: Real Data Experiments with Dataset 1

In this section, you will implement experiments with dataset1. There are two subsets in this dataset: linearly and nonlinearly.

For each subset, your tasks are the following:

1. Split it using StratifiedKFold with K = 5. Make sure the splitting is **random** (preferrably seeded).
2. For each split, perform training and test with an instance of your model.
3. Compute the **confusion matrix**. The values should be ***accumulated*** across all folds.
4. Compute the **performance results**: accuracy, precision, recall, and F1. The values should the ***average*** across all folds.

Please show the results clearly (one item at a time).

Q3.1: LinearXY

```
# write your code in this block
```

Q3.2: NonLinearXY

```
# write your code in this block
```

Q4: Real Data Experiments with Dataset 2

Dataset2 has been split into training and test subsets, so you only need to load them accordingly.

In this part, you need to try out different model parameter values and observe how they affect the results.

For each of the questions below, show **performance results** as four lists. An eample output is the following:

```
accuracy scores: [1., 1., 1., 1.]
precision scores: [1., 1., 1., 1.]
recall scores: [1., 1., 1., 1.]
f1 scores: [1., 1., 1., 1.]
```

Use the following function to obtain one-hot encoded labels. Note that the returned labels are by default **row vectors**.

```
from sklearn.preprocessing import OneHotEncoder

# Simply pass the labels as two 1D arrays.
def one_hot_encode(YTrain, YTest):
    encoder = OneHotEncoder(sparse_output = False)
    return encoder.fit_transform(YTrain.reshape(-1, 1)), encoder.transform(YTest.reshape(-1, 1))
```

Q4.1: Epochs

Experiment with at least **5** different choices of total epochs.

```
# write your code in this block
```

Q4.2: Learning Rate

Experiment with at least **5** different choices of learning rates.

```
# write your code in this block
```

Q4.3: Regularization Parameter

Experiment with at least **3** different choices of regularization parameter.

```
# write your code in this block
```

Q4.4: Network Structure

Experiment with at least **5** different choices of network structure. This includes number of layers and number of nodes in each layer.

Hint: Try experimenting with increasing complexity.

```
# write your code in this block
```

Q5: Follow-up Questions

For each question below, provide a short answer. You can cite your code if needed.

Q5.1: Briefly describe the workflow of how your model classifies the data.

[Answer]

Q5.2: In your own words, explain how the forward propagation in your model works.

[Answer]

Q5.3: In your own words, explain how the backward propagation in your model works.

[Answer]

Q5.4: In theory, how do the total number of epochs, the learning rate, and the regularization parameter impact the performance of model? Does any of the theoretical impact actually happen in your result? If so, point them out.

[Answer]

Epochs When the number is too small, the model doesn't enough and hence it cannot predict the test data well. When the number is too large, the model tends to overfit the training data, also resulting in poor performance on the test data.

Learning Rate Learning rate is the step size of the gradient descent. Therefore, if the value is too small, it may take the model longer to fit the data. On the other hand, if the value is too large, the model may "overstep" and hence cannot reach the optimal point.

Regularization Parameter The role of regularization is to prevent overfitting. If the parameter is too small, then it cannot contribute enough to the weight update and hence may not be able to prevent overfitting. On the other hand, if the value is too large, it may contribute too much to the update, which prevents the model from learning.