Northeastern University, Khoury College of Computer Science

CS 6120 — Assignment 2

Due: January 27, 2025 (100 points)

YOUR NAME + LDAP

In this homework, we will be building a neural network from scratch to predict the sentiment of a tweet using a two-layer neural network. In Python, *you must only use the* `numpy` *libraries.* The components to submit to Gradescope are as follows.

1. Python code (`assignment2.py`) with implementations of inference - Q1, gradients - Q3, optimization - Q4, and training - Q5. You may use our template code.
2. A PDF with all the theoretical solutions in Q2. Show your work in the derivations.
3. Results of your optimization - Q5

Question 1: Inference: Implementation

In this question, your answer will be in Python and submitted via Gradescope. You will implement a function that computes the output of a two layer neural network predicting a single value, defined by the function f:

$$\begin{aligned} \hat{y} &= f(W_1, \mathbf{b}_1, W_2, b_2) \\ &= \sigma(W_2\sigma(W_1\mathbf{x} + \mathbf{b}_1) + b_2) \end{aligned} \tag{1.1}$$

where $\mathbf{h} \in \mathbb{R}^\ell$ is the output of the first layer and the parameters in the above function assume the following:

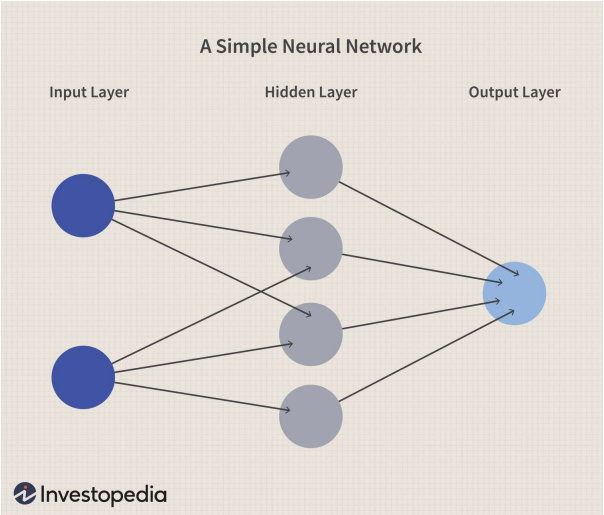
- d is the number of features / dimensions in the input vector $\mathbf{x}$
- ℓ is the number of dimensions in the hidden layer
- $\mathbf{x} \in \mathbb{R}^d$ is an input vector
- $y \in \mathbb{R}$ is the output value, which is either 0 or 1
- $W_1 \in \mathbb{R}^{\ell \times d}$ is the weight matrix in the first layer

- $\mathbf{b}_1 \in \mathbb{R}^\ell$ is the bias matrix in the first layer
- $W_2 \in \mathbb{R}^{1\times\ell}$ is the weight matrix in the second layer
- $b_2 \in \mathbb{R}$ is the bias matrix in the second layer

The sigmoid function used above is the following:

$$\sigma(\mathbf{z}) = \frac{1}{1+e^{-z}} \tag{1.2}$$

The neural network diagram of the above equations has following form:



We will be applying this neural network to the problem of sentiment analysis of tweets.

Q 1.1: Implementation of the Sigmoid Function

Implement the sigmoid function as defined in equation (1.2). Here, $\mathbf{x} \in \mathbb{R}^d$ or $\mathbb{R}^{N\times d}$, depending on if it's a vector or N vectors. Subsequently, the output will be of size ℓ or $N \times \ell$, depending on what the input is. The function signature where you must enter your solution in replacing `<YOUR CODE HERE>` is as follows:

```
def sigmoid( x ):
  """
  This function implements the logistic regression algorithm.

  Args:
    x: A numpy array of shape (d,) or (N, d) representing
      the input data.

  Returns:
    y: A numpy array of shape (l,) or (N, l) representing
      the output data.
  """
```

```
  y = <YOUR CODE HERE>
  return y
```

Q 1.2: One Layer Inference: Logistic Regression

Implement the forward propagation for logistic regression, as it has been written in equation (1.3) as follows:

$$\mathbf{h} = \sigma(W\mathbf{x} + \mathbf{b}) \tag{1.3}$$

Here, $\mathbf{x} \in \mathbb{R}^d$ or $\mathbb{R}^{d \times N}$, depending on if it's a vector or N vectors. Subsequently, the output will be of size ℓ or $\ell \times N$, depending on what the input is. Additionally, your function will take in a weighting matrix $W \in \mathbb{R}^{\ell \times d}$ and a bias vector $\mathbf{b} \in \mathbb{R}^{\ell}$. The function signature where you must enter your solution in replacing `<YOUR CODE HERE>` is as follows:

```
def inference_layer(X, W, b):
  """
  Implements the forward propagation for the logistic regression model.

  Args:
    X: The input data, of shape (number of features, number of examples).
    w: Weights, a numpy array of shape 1 x d.
    b: Bias, a scalar.

  Returns:
    y: The output of shape 1 or 1 x N
  """
  <YOUR CODE HERE>
  return y
```

Q 1.3: Two Layer Inference: Neural Network

An additional layer makes this a neural network. You can add this additional layer by taking the output of the function that you wrote in Q(1.2) and plugging it in as the input to another logistic regression function. The two layer neural network with an intermediate hidden layer $\mathbf{h}$ can be written as the following.

$$\hat{y} = \sigma(\boldsymbol{w}_2\sigma(W_1\boldsymbol{x} + \boldsymbol{b}_1) + \boldsymbol{b}_2) \tag{1.4}$$

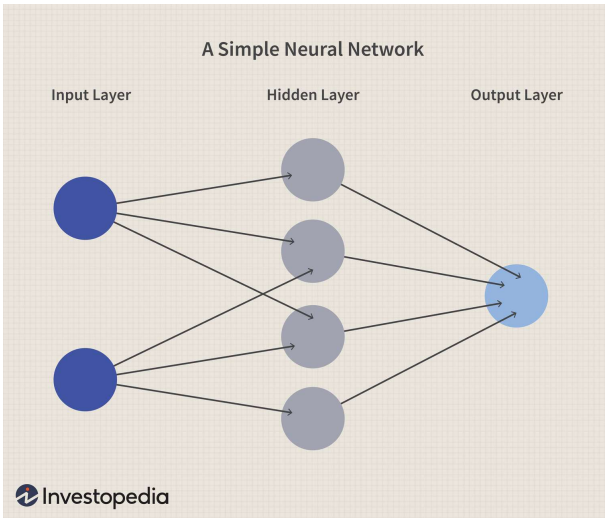
Breaking down the above equation (1.4), if we substitute the variable $\mathbf{h}$ for the intermediate (first layer) output as:

$$\boldsymbol{h} = \sigma(W_1\boldsymbol{x} + \boldsymbol{b}_1) \tag{1.5}$$

we can also write:

$$\hat{y} = \sigma(\boldsymbol{w}_2\boldsymbol{h} + \boldsymbol{b}_2) \tag{1.6}$$

where $\hat{y}$ is the predicted output. The neural network diagram of the above equations looks like:



Write a function that implements the forward propagation described of a two layer neural network as described in in (1.4). Use the function that you wrote in (1.3).The function signature is this:

```
def inference_2layers(X, W1, W2, b1, b2)
  """
  Implements the forward propagation of a two layer neural network
  model.

  - Let d be the number of features (dimensionality) of the input.
  - Let N be the number of data samples in your batch
  - Let H be the number of hidden units

  Args:
  X: The input data, with shape either d or d x N
  W1: Weights for the first weight matrix: shape = H x d
  w2: Weights for the second matrix: shape = 1 x H
  b1: Bias value for first layer
  b2: Bias value for second layer

  Returns:
  y: Singular output data (1, 0), with shape either 1 or N.
  """
  y = <YOUR CODE HERE>
  return y
```

Note: the general purpose inference layer that you wrote in Q(1.2) is shape ℓ or $\ell \times N$, where we built flexibility into whether or not the output has multiple outputs. In this question, we're building this neural network to predict sentiment, where we need to predict only a single label, i.e., zero or one. So the output in this question has the dimensionality specified in the above signature.

Q 1.4: Implementation of BCE Loss

Implement the forward propagation for the binary cross-entropy loss function, as it has been written as follows:

$$\mathcal{L} = -\frac{1}{N}\sum_{i}^{N} y^{(i)} \log \hat{y}^{(i)} + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)}) \tag{1.7}$$

Here, $\mathbf{x} \in \mathbb{R}^d$ or $\mathbb{R}^{N \times d}$, depending on if it is a vector or N vectors. Subsequently, the output will be of size ℓ or $N \times \ell$, depending on what the input is. Additionally, your function will take in a weighting matrix $W \in \mathbb{R}^{\ell \times d}$ and a bias vector $\mathbf{b} \in \mathbb{R}^{\ell}$.

The function signature, where you must enter your solution in replacing `<YOUR CODE HERE>`, is as follows:

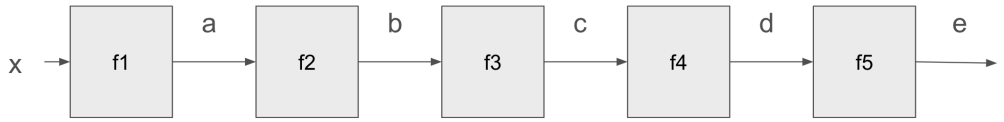
```
def bce_forward(yhat, y):
  """
  This function calculates the gradient of the logistic regression cost
  function with respect to the weights and biases.

  Args:
    yhat: A numpy array of shape (N,) representing the predicted outputs
    y: A numpy array of shape (N,) representing the target labels.

  Returns:
    loss_value
  """
  loss_value = <YOUR CODE HERE>
  return loss_value
```

Question 2: Gradients

In this question, we will derive all the relevant gradients for the logistic regression, including the most typical loss function, the binary cross-entropy loss function (BCE loss). We will then *implement* the functions for the gradients so that we can eventually do gradient descent. It helps to understand the Chain Rule, which Tensorflow and PyTorch both make extensive use of. Recall the chain rule:



$$\text{deriv } f_5(f_4(f_3(f_2(f_1(x))))) = f_5'(d) \times f_4'(c) \times f_3'(b) \times f_2'(a) \times f_1'(x)$$

Figure 2.1: Pictorial Depiction of the Chain Rule

In our case, we have two layers, $f_1(\mathbf{x})$ and $f_2(\mathbf{h})$, and the final function we wish to take the derivative of is

$$\hat{y} = f_2(f_1(\mathbf{x})) \tag{2.1}$$

which is depicted in

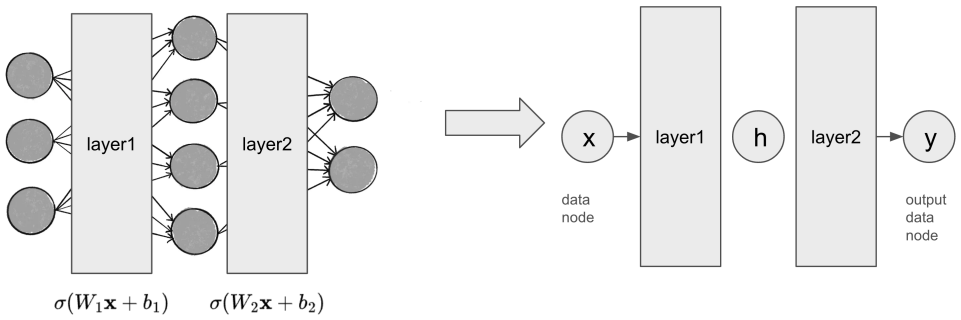


Figure 2.2: The Edges Depict Weight Matrices

Adding the loss function, the block diagram looks likes this.

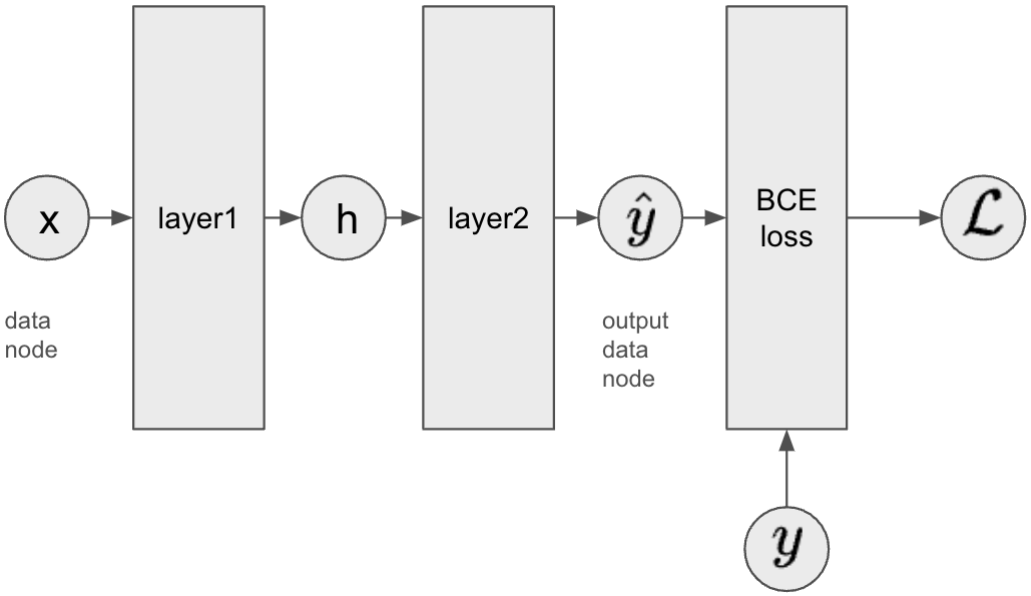


Figure 2.3: Block Diagram of DNN Layers

You will derive the gradient $\nabla_{W_1,W_2,\mathbf{b}_1,\mathbf{b}_2} f(W_1, W_2, \mathbf{b}_2, \mathbf{b}_1, y, \mathbf{x})$ for each parameter. That is, we will be taking partial derivatives with respect to each parameter. For our derivations, we will start from the back (at the cost function) and work our way to each parameter. We'll do so step by step in the following sub-questions.

Q 2.1: Gradient for the Loss Function

Derive the gradient for the BCE loss $\mathcal{L}(\hat{y}, y)$, the last block in Fig. 2.3, with respect to its input, $\hat{y}$. (Note, this is *not* the input of the entire function, but rather the input to the loss function.). Recall the BCE equation in (1.7):

$$\mathcal{L}(\hat{y}, y) = -\frac{1}{N}\sum_{i}^{N} y^{(i)} \log \hat{y}^{(i)} + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)})$$

Write your solution to:

$$\frac{\partial \mathcal{L}}{\partial \hat{y}} = ??$$

Q 2.2: Gradient Derivation for Sigmoid

Derive the gradient for the sigmoid function as defined in equation (1.2). The solution to this question should be turned into Gradescope via PDF. This derivation can have $\sigma(\cdot)$ in it if need be. You can assume that both the input and outputs have the dimensions: $\mathbf{x} \in \mathbb{R}^d$. The following relationships / properties should help you.

- The chain rule: the gradient of a function f of another function g is the product of the gradients. Let $g(x) = z$, then we can write:

$$\nabla_x \quad f(g(x)) \quad = \quad z' \cdot f'(z)$$

- The derivative of an exponent is the exponent (times the derivative of the argument).

$$\nabla_x \quad e^x = e^x$$

Write your solution to:

$$\frac{\partial \sigma(z)}{\partial z} = ??$$

Q 2.3: Gradient for a Single Layer: Logistic Gradient

Derive the gradient for the sigmoid function as defined in equation (1.2). The solution to this question should be turned into Gradescope via PDF. This derivation can have $\sigma(\cdot)$ in it if need be. You can assume that both the input and outputs have the dimensions: $\mathbf{x} \in \mathbb{R}^d$. The following relationships / properties should help you.

- The chain rule: the gradient of a function f of another function g is the product of the gradients. Let $g(x) = z$, then we can write:

$$\nabla_x \quad f(g(x)) \quad = \quad z' \cdot f'(z)$$

- The derivative of an exponent is the exponent (times the derivative of the argument).

$$\nabla_x \quad e^x = e^x$$

For the function

$$\hat{y} = \sigma(W\mathbf{h} + \mathbf{b})$$

derive the three derivatives of $\hat{y}$ for $W, \mathbf{b}$, and the input $\mathbf{h}$:

$$\begin{aligned} \frac{\partial \hat{y}}{\partial W} &= ?? \\ \frac{\partial \hat{y}}{\partial \mathbf{h}} &= ?? \\ \frac{\partial \hat{y}}{\partial \mathbf{b}} &= ?? \end{aligned}$$

Q 2.4: Gradient Derivation for all parameters

Now we will put it all together and derive all the final gradients with the chain rule.
For the function in (1.7), write out the gradients for $\mathcal{L}$ with respect to all the desired variables:

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial W_2} &= ?? \\ \frac{\partial \mathcal{L}}{\partial \mathbf{b}_2} &= ?? \\ \frac{\partial \mathcal{L}}{\partial W_1} &= ?? \\ \frac{\partial \mathcal{L}}{\partial \mathbf{b}_1} &= ?? \end{aligned}$$

Question 3: Implement Gradients

After deriving all the gradients, we can implement the gradients for all of your parameters for any given input and output.

```
def gradients(X, y, W1, W2, b1, b2):
  '''
  Calculate the gradients of the cost functions with respect to W1, W2, b1, and

  Args:
    W1: Weight np.array of shape (h, d), first layer
    b1: Bias np.array of shape (h,), first layer
    W1: Weight np.array of shape (1, h), second layer
    b1: Bias value, second layer
```

```
    X: Input data of shape (d, N) representing the input data.
    y: Target / output labels of shape (N,)

  Returns:
    dL/dW1: Derivative of cost fxn w/r/t W1
    dL/db1: Derivative of cost fxn w/r/t b1
    dL/dW2: Derivative of cost fxn w/r/t W2
    dL/db2: Derivative of cost fxn w/r/t b2
    L: Loss value
  '''
  return dW1, dW2, db1, db2, L
```

Question 4: Optimization

In this section, you will implement gradient descent for the logistic regression with the binary cross-entropy loss. Please use the functions that you've implemented in the prior questions in order to optimize for the parameters W and $\mathbf{b}$. Feel free to use the data that we've provided here in a multi-class classification problem to do gradient descent, which can be optimized with:

$$\begin{aligned} W^+ &\leftarrow W - \lambda \nabla_W \mathcal{L}_{BCE}(W, \mathrm{x}, \mathbf{y}, \mathbf{b}) \\ \mathbf{b}^+ &\leftarrow \mathbf{b} - \lambda \nabla_{\mathbf{b}} \mathcal{L}_{BCE}(W, \mathbf{x}, \mathbf{y}, \mathbf{b}) \end{aligned} \tag{4.1}$$

We will implement gradient descent in equation (4.1). Use the following function signature, updating all your parameters.

```
def update_params(batchx, batchy, W1, B1, W2, b2, lr = 0.01):
    '''
    Updates your parameters
        batchx: Mini-batch of features
        batchy: Corresponding mini-batch of labels
        W1: Starting value of W1
        b1: Starting value of b1
        W2: Starting value of W2
        b2: Starting value of b2
        lr: Learning parameter, default to 0.01

    Returns
        W1: New value of W1
        b1: New value of b1
        W2: New value of W2
        b2: New value of b2
    '''
    <YOUR CODE HERE>
    return W1, b1, W2, b2
```

Question 5: Putting it All Together

Download the data `twitter_data.pkl` and utility function `utils.py` from the course website. It's at:

`https://course.ccs.neu.edu/cs6120s25/data/twitter/`

You can load the data and extract features with the following code:

```
# Import utility functions with NLTK
import pickle
import numpy as np
import nltk
from utils import extract_features
nltk.download('twitter_samples')
nltk.download('stopwords')


def train_nn(filename, hidden_layer_size, iters=1e6, lr = 0.01):
    '''
    Reads in data (twitter_data.pkl), initiates parameters, extracts features
    and returns the parameters

    Returns:
        W1, b1, W2, b2
    '''

    # Load the data
    with open(filename, 'rb') as f:
        data = pickle.load(f)

    train_x = data['train_x']
    train_y = data['train_y']
    test_x = data['test_x']
    test_y = data['test_y']
    freqs = data['freqs']

    # Review the data
    print("Training Data: ", train_x[0])

    # This is how you extract features
    print("Features ", extract_features(train_x[0], freqs))

    return None, None, None, None
```

Train your neural network, evaluating with your test data (no training on test data, please). Print out your training loss curves and your evaluation loss curves. Print out a few examples from your test set. You will need to extract the features as detailed before with `extract_features`.

Submission Instructions

Upload examples, your derivations, data, your loss curves, and all your code to Gradescope into file called `assignment2.*` and `assignment2.py`. You can consolidate these into three files total. These include:

Code All your code saved in a Python file called `assignment2.py`. We will be expecting multiple different functions in `assignment2.py`, including:

- Q1.1 - `sigmoid`
- Q1.2 - `inference_layer`
- Q1.3 - `inference_2layers`
- Q1.4 - `bce_forward`
- Q3 - `gradients`
- Q4 - `update_params`
- Q5 - Well commented training code

Data your model parameters saved in a pickle file called `assignment2.pkl`. We will be expecting multiple parameters for both layers of your neural network, which you can save with the following code.

```
params = {
    'W1': W1,
    'W2': W2,
    'b1': b1,
    'b2': b2,
}

with open('assignment2.pkl', 'wb') as f:
    pickle.dump(data, f)
```

Plots and Derivations your plots and derivations in a PDF in a file called `assignment2.pdf`.

- Q2.1 - Loss function gradient $\frac{\partial \mathcal{L}}{\partial \hat{y}}$
- Q2.2 - Sigmoid gradient $\frac{\partial \sigma(z)}{\partial z}$
- Q2.3 - Derivations for gradients w/r/t outputs for W, $\mathbf{h}$, and $\mathbf{b}$
- Q2.4 - Gradient derivations for all paramaeters $W_1, W_2, \mathbf{b}_1, \mathbf{b}_2$
- Q5 - Optimization loss curve plots, examples from your test set. For example, you can do something like this:

```
my_tweet = 'This is a ridiculously bright movie. The plot was terrible
and I was sad until the ending!'
y_hat = inference_2_layers(
    extract_features(my_tweet, freqs), W1, W2, b1, b2)
print(y_hat)
if y_hat > 0.5:
    print('Positive sentiment')
else:
    print('Negative sentiment')
```