CM146, Fall 2022
Problem Set 2: Logistic Regression and Neural Networks
Due Nov. 15, 2022 at 11:59 pm

1 Logistic Regression [20 pts]

Consider the logistic regression model for binary classification that takes input features $\boldsymbol{x}_i \in R^m$ and predicts $y_i \in \{-1, 1\}$. As we learned in class, the logistic regression model fits the probability $P(y_i = 1|\boldsymbol{x}_i)$ using the *sigmoid* function:

$$P(y_i = 1) = \sigma(\boldsymbol{w}^T \boldsymbol{x}_i + b) = \frac{1}{1 + \exp(-\boldsymbol{w}^T \boldsymbol{x}_i - b)}, \tag{1}$$

where the sigmoid function is as follows, for any $z \in \mathbb{R}$,

$$\sigma(z) = (1 + \exp(-z))^{-1}.$$

Given N training data points, we learn the logistic regression model by minimizing the following loss function:

$$J(\boldsymbol{w}, b) = \frac{1}{N} \sum_{i=1}^{N} \log\Big(1 + \exp\big(-y_i(\boldsymbol{w}^T \boldsymbol{x}_i + b)\big)\Big). \tag{2}$$

(a) **(7 pts)** Prove the following. For any $z \in \mathbb{R}$, the derivative of the sigmoid function is as follows:

$$\frac{d\sigma(z)}{dz} = \sigma(z)\big(1 - \sigma(z)\big).$$

(Hint: use the chain rule. And $\frac{d}{dx}\exp(x) = \exp(x)$.)

Solution:

$$\begin{aligned}
\frac{d\sigma(z)}{dz} &= \frac{d}{dz}(1 + \exp(-z))^{-1} \\
&= (1 + \exp(-z))^{-2} \times \frac{d}{dz}(1 + \exp(-z)) \\
&= (1 + \exp(-z))^{-2}(-\exp(-z)) \\
&= (1 + \exp(-z))^{-1}(1 + \exp(-z))^{-1}(-\exp(-z)) \\
&= \sigma(z)\left(\frac{-\exp(-z)}{1 + \exp(-z)}\right) \\
&= \boxed{\sigma(z)(1 - \sigma(z))}
\end{aligned}$$

Parts of this assignment are adapted from course material by Andrea Danyluk (Williams), Tom Mitchell, Matt Gormley and Maria-Florina Balcan (CMU), Stuart Russell (UC Berkeley), Carlos Guestrin (UW), Dan Roth (UPenn) and Jessica Wu (Harvey Mudd).

Suppose we are updating the weight $\mathbf{w}$ and bias term b with the gradient descent algorithm and the learning rate is α. We can derive the update rule as

$$w_j \leftarrow w_j - \alpha \frac{\partial J(\mathbf{w}, b)}{\partial w_j}, \text{ where } \frac{\partial J(\mathbf{w}, b)}{\partial w_j} = \frac{1}{N} \sum_{i=1}^{N} \Big(\sigma\big(y_i(\boldsymbol{w}^T \boldsymbol{x}_i + b)\big) - 1 \Big) y_i x_{i,j}$$

where w_j denotes the j-th element of the weight vector $\mathbf{w}$ and $x_{i,j}$ denotes the j-th element of the vector $\mathbf{x}_i$. And

$$b \leftarrow b - \alpha \frac{\partial J(\mathbf{w}, b)}{\partial b}, \text{ where } \frac{\partial J(\mathbf{w}, b)}{\partial b} = \frac{1}{N} \sum_{n=1}^{N} \Big(\sigma\big(y_i(\boldsymbol{w}^T \boldsymbol{x}_i + b)\big) - 1 \Big) y_i$$

We now compare the stochastic gradient descent algorithm for logistic regression with the Perceptron algorithm. Let us define residue for example i as $\delta_i = 1 - \sigma\big(y_i(\boldsymbol{w}^T \boldsymbol{x}_i + b)\big)$. Assume the learning rate for both algorithms is α.

(b) **(5 pts)** If the data point $(\mathbf{x}_i, y_i)$ is correctly classified with high confidence, say $y_i = 1$ and $\mathbf{w}^\top \mathbf{x}_i + b = 5$. What will be the residue? For the logistic regression, what will be the update with respect to this data point? What about the update of the Perceptron algorithm? (**Hint**: you may assume $\sigma(5) \approx 0.9933$. The answer should take the form $w_j \leftarrow w_j - [?]$ or $w_j \leftarrow w_j + [?]$ where you fill in [?]. You can directly use $x_{i,j}$ and α as variables.)

Solution:

$$\begin{aligned} \delta_i &= 1 - \sigma(y_i(\mathbf{w}^T \mathbf{x_i} + b)) \\ &= 1 - \sigma(1(5)) \\ &= 1 - \sigma(5) \\ &= 1 - 0.9933 \\ &= \boxed{0.0067} \end{aligned}$$

Logistic Regression Update: $w_j \leftarrow w_j - \alpha \left(\frac{1}{1} \sum_{i=1}^{1} (\sigma(1(5)) - 1) \right) (1) x_{i,j}$

$$\boxed{w_j \leftarrow w_j + 0.0067 \alpha x_{i,j}}$$

Perceptron Update: $\boxed{w_j \leftarrow w_j}$

(c) **(5 pts)** If the data point $(\mathbf{x}_i, y_i)$ is misclassified, say $y_i = 1$ and $\mathbf{w}^\top \mathbf{x}_i + b = -1$. What will be the residue and how is the weight updated for the two algorithms? (**Hint**: you may assume $\sigma(-1) \approx 0.2689$.)

Solution:

$$\begin{aligned}\delta_i &= 1 - \sigma(y_i(\mathbf{w}^T \mathbf{x_i} + b)) \\ &= 1 - \sigma(1(-1)) \\ &= 1 - \sigma(-1) \\ &= 1 - 0.2689 \\ &= \boxed{0.7311}\end{aligned}$$

$$\text{Logistic Regression Update: } w_j \leftarrow w_j - \alpha \left(\frac{1}{1} \sum_{i=1}^{1} (\sigma(1(-1)) - 1) \right) (1) x_{i,j}$$

$$\boxed{w_j \leftarrow w_j + 0.7311 \alpha x_{i,j}}$$

$$\text{Perceptron Update: } w_j \leftarrow w_j + \alpha y_i x_{i,j}$$

$$\boxed{w_j \leftarrow w_j + \alpha x_{i,j}}$$

(d) **(3 pts)** For the update step in SGD and perceptron algorithm, is the influence of the input data $\mathbf{x}_i$ of the same magnitude? If not, what is the difference?

Solution:

The influence of the input data $\mathbf{x_i}$ is not the same across both algorithms. In logistic regression, the input data's impact is multiplied by a factor equal to the residue, which is a value between 0 and 1, times the learning rate. On the other hand, in Perceptron, the input data's impact is 0 when the classification is correct and multiplied by the learning rate if the classification is wrong.

2 Regularization [10 pts]

Data is separable in one dimension if there exists a threshold t such that all data that have values less than t in this dimension have the same class label. Suppose we train logistic regression for infinite iterations according to our current gradient descent update on the training data that is separable in at least one dimension.

(a) **(2 pts)** (multiple choice) If the data is linearly separable in dimension j, i.e. $x_{i,j} < t$ for all $(\mathbf{x}_i, y_i)$ such that $y_i = -1$. Which of the following is true about the corresponding weight w_j when we train the model for infinite iterations?

(A) The weight w_j will be 0.

(B) The weight w_j will be encouraged to grow continuously during each iteration of SGD and can go to infinity or -infinity.

Solution:

B

(b) **(5 pts)** We now add a new term (called $l2$-regularization) to the loss function

$$J(\boldsymbol{w}, b) = \frac{1}{N}\sum_{i=1}^{N} \log\Big(1 + \exp\big(-y_i(\boldsymbol{w}^T\boldsymbol{x}_i + b)\big)\Big) + 0.1\sum_{j=0}^{M} w_j^2. \tag{3}$$

What will be the update rule of w_j for the new loss function?

Solution:

$$w_j \leftarrow w_j - \frac{\partial J(\mathbf{w}, b)}{\partial w_j}$$

$$\leftarrow w_j - \left(\frac{1}{N}\sum_{i=1}^{N}(\sigma(y_i(\mathbf{w}^T\mathbf{x_i} + b)) - 1)y_i x_{i,j} + 0.2\sum_{j=0}^{M} w_j\right)$$

(c) **(3 pts)** How does $l2$ regularization help correct the problem in (a)?

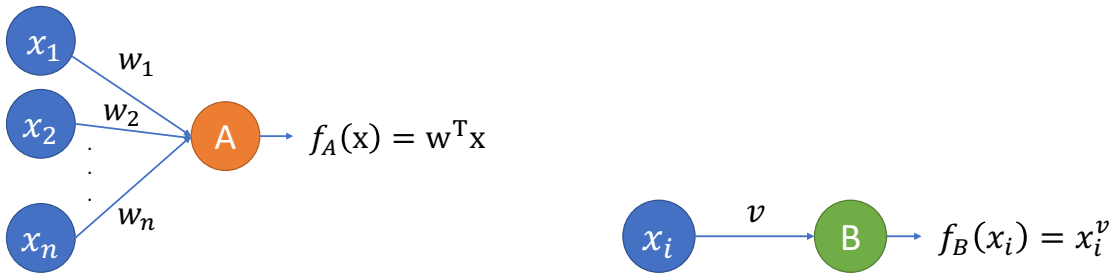
Solution:

By adding the extra term, $l2$ regularization counteracts the main term of the update, pushing the new weight closer to 0 than a normal update. This allows us to counteract the effect of overfitting that caused the problem in `(a)`.

3 Neural Networks and Deep Learning [20 pts]

In the following set of questions, we will discuss feed-forward and backpropagation for neural networks.

We will consider two kinds of hidden nodes/units for our neural network. The first cell (Cell A) operates as a weighted sum with no activation function. Given a set of input nodes and their weights, it outputs the weighted sum. The second cell (Cell B) operates as a power function. It takes only one input along with the weight and outputs the power of input to the weight as the output. We provide illustrations for both the nodes in Figure 1. Note that $\mathbf{x} = [x_1, x_2, \ldots, x_n]^T$ and $\mathbf{w} = [w_1, w_2, \ldots, w_n]^T$.



(a) (b)

Figure 1: An illustration of Cell A and Cell B used in the neural network. (a) On the left, we have Cell A - Weighted sum. It simply computes the weighted sum over the input nodes. (b) On the right, we have Cell B - It computes power of the input node over the weight

Using these two kinds of hidden nodes, we build a two-layer neural network as shown in Figure 2. The inputs to the neural network is $\mathbf{x} = [x_1, x_2, x_3]^T$ and the output is $\hat{y}$. The parameters of the model are $\mathbf{v} = [v_1, v_2, v_3]^T$ and $\mathbf{w} = [w_1, w_2, w_3]^T$.

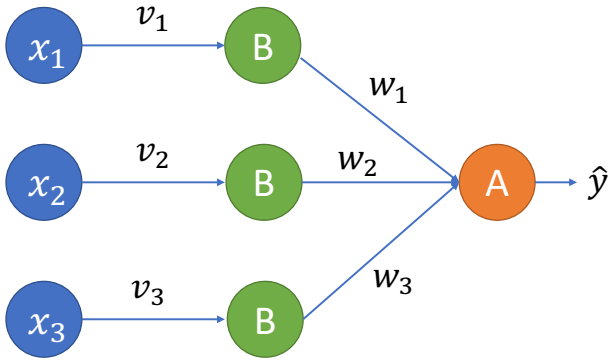


Figure 2: Architecture of the neural network.

(a) **(4 pts)** We would like to compute the feed-forward for the network. Mathematically, compute the expression for $\hat{y}$ given the neural network in terms of $\mathbf{x}, \mathbf{v}, \mathbf{w}$.

Solution:

$$\begin{aligned}\hat{y} &= f_A(\mathbf{B}) \\ &= w_1(f_B(x_1)) + w_2(f_B(x_2)) + w_3(f_B(x_3)) \\ &= \boxed{w_1(x_1^{v_1}) + w_2(x_2^{v_2}) + w_3(x_3^{v_3})}\end{aligned}$$

(b) **(2 pts)** Utilizing the expression for $\hat{y}$ computed in (a), compute the predicted value $\hat{y}$ when $\mathbf{x} = [1, 3, 2]^T, \mathbf{v} = [1, 0, 2]^T, \mathbf{w} = [-4, 3, 1]^T$

Solution:

$$\begin{aligned}\hat{y} &= w_1(x_1^{v_1}) + w_2(x_2^{v_2}) + w_3(x_3^{v_3}) \\ &= -4(1^1) + 3(3^0) + 1(2^2) \\ &= \boxed{3}\end{aligned}$$

We want to now utilize the backpropagation algorithm to update the weights of this neural network. First we will compute the gradients with respect to the weights. For the following parts, we are only interested in updating v_1 and w_1 in the network.

(c) **(5 pts)** Let's derive the expression for gradients for the weights. More specifically, compute the expressions for $\frac{\partial \hat{y}}{\partial w_1}$ and $\frac{\partial \hat{y}}{\partial v_1}$. Using the values described in part (b), compute the final values of these expressions. (**Hint:** Use the expression computed in part (a))

Solution:

$$\begin{aligned}\hat{y} &= w_1(x_1^{v_1}) + w_2(x_2^{v_2}) + w_3(x_3^{v_3}) \\ \frac{\partial \hat{y}}{\partial w_1} &= \frac{\partial}{\partial w_1}(w_1(x_1^{v_1}) + w_2(x_2^{v_2}) + w_3(x_3^{v_3})) \\ &= x_1^{v_1} \\ &= 1^1 \\ &= \boxed{1} \\ \frac{\partial \hat{y}}{\partial v_1} &= \frac{\partial}{\partial v_1}(w_1(x_1^{v_1}) + w_2(x_2^{v_2}) + w_3(x_3^{v_3})) \\ &= w_1 \times \frac{\partial}{\partial v_1}(x_1^{v_1}) \\ &= w_1 \times x_1^{v_1} \times \ln(x_1) \\ &= w_1 \ln(x_1) x_1^{v_1} \\ &= -4(0)(1^1) \\ &= \boxed{0}\end{aligned}$$

(d) **(3 pts)** Assuming the loss function is $\mathcal{L} = (y - \hat{y})^2$ where y is the true value of the output and $\hat{y}$ is the predicted output from the neural network. Compute the expression for $\frac{\partial \mathcal{L}}{\partial \hat{y}}$. Let the actual value of $y = 5$. What will the value for this expression be?

Solution:

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial \hat{y}} &= \frac{\partial}{\partial \hat{y}}(y - \hat{y})^2 \\ &= 2(y - \hat{y})(-1) \\ &= \boxed{-2(y - \hat{y})} \\ \frac{\partial \mathcal{L}}{\partial \hat{y}} &= -2(y - \hat{y}) \\ &= -2(5 - 3) \\ &= \boxed{-4}\end{aligned}$$

(e) **(3 pts)** Using parts (c) and (d), compute the values for the expressions for $\frac{\partial \mathcal{L}}{\partial v_1}$ and $\frac{\partial \mathcal{L}}{\partial w_1}$. (**Hint:** Use the chain rule)

Solution:

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial v_1} &= \frac{\partial \mathcal{L}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial v_1} \\ &= -4(0) \\ &= \boxed{0} \\ \frac{\partial \mathcal{L}}{\partial w_1} &= \frac{\partial \mathcal{L}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w_1} \\ &= -4(1) \\ &= \boxed{-4}\end{aligned}$$

We will finally use the gradient descent algorithm to update the weights for v_1 and w_1. Note that the gradient descent update formula for a parameter w is given as $w \leftarrow w - \eta \frac{\partial \mathcal{L}}{\partial w}$. Assume the learning rate $\eta = 1$ for the following question.

(f) **(3 pts)** Using the gradient descent update rule explained above, compute the updated values for v_1 and w_1.

Solution:

$$\begin{aligned}v_1 &\leftarrow v_1 - \eta \frac{\partial \mathcal{L}}{\partial v_1} \\ &\leftarrow 1 - 1(0) \\ &\leftarrow \boxed{1} \\ w_1 &\leftarrow w_1 - \eta \frac{\partial \mathcal{L}}{\partial w_1} \\ &\leftarrow -4 - 1(-4) \\ &\leftarrow \boxed{0}\end{aligned}$$

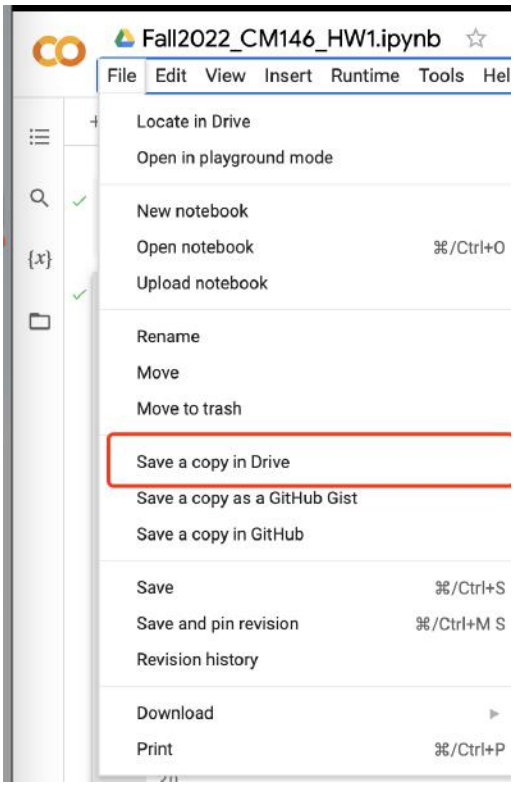
4 Programming exercise : Implementing Logistic Regression and Neural Networks [50 pts]

Introduction

In this problem, we will continue to work on the mushroom classification task which was used in HW1. The dataset is adapted from the UCI Machine Learning Repository and it contains descriptions of hypothetical samples corresponding to 23 species of gilled mushrooms. Each mushroom is described in terms of physical characteristics, and the goal is to classify mushrooms as *edible* or *poisonous*. We will apply decision trees and k-nearest neighbors. Since this dataset is relatively simple for classification, we only use 6 features out of 22 features in the original dataset. Features we use include: cap-shape, cap-color, gill-color, stalk-root, veil-type, ring-number.

For all the coding, please refer to the following Colab notebook Fall2022-CM146-HW2.ipynb.

Before executing or writing down any code, please make a copy of the notebook and save it to your own google drive by clicking the "File" → "Save a copy in Drive".



You will probably be prompted to log into your Google account. Please make sure all the work you implement is done on your own saved copy. You won't to able to make changes on the the original notebook shared with the entire class.

The notebook has marked blocks where you need to code:

```
### ========== TODO : START ========== ###
### ========== TODO : END ========== ###
```

Submission instructions for programming problems

- Please save the execution output in your notebook. When submitting, please export the notebook to a `.ipynb` file by clicking "File" → "Download .ipynb" and upload the notebook to BruinLearn.
- Your code should be commented appropriately. Importantly:
  - Your name should be at the top of the file.
  - Each class and method should have an appropriate doctsring.
  - Include some comments for anything complicated.

  There are many possible solutions to this assignment, which makes coding style and comments important for graders to conveniently understand the code.
- Please submit all the plots and the rest of the solutions (other than codes) to Gradescope.

4.1 Creating Datasets and DataLoaders [5 pts]

Datasets and Dataloaders are pytorch specific wrappers useful for maintaining and batching data. They help in dynamically loading and keeping track of batches while training/inference of machine learning algorithms. We will wrap our current data into datasets and dataloaders for this part.

(a) **(5 pts)** Prepare `train_loader`, `valid_loader`, and `test_loader` by using `TensorDataset` and `DataLoader`. We expect to get a batch of pairs $(\mathbf{x}_n, y_n)$ from the dataloader. Please set the train batch size to 16 and test batch size to 32.

You can refer https://pytorch.org/docs/stable/data.html for more information about `TensorDataset` and `DataLoader`.

Solution:

```
def get_dataloaders(X_train, y_train, X_val, y_val, X_test, y_test, train_batch_size=16, test_batch_size=32):
    ### ========== TODO : START ========== ###
    # part a: Create dataloaders for train, validation and test sets
    train_dataset = TensorDataset(X_train, y_train)
    train_loader = DataLoader(train_dataset, batch_size=train_batch_size)
    val_dataset = TensorDataset(X_val, y_val)
    val_loader = DataLoader(val_dataset, batch_size=test_batch_size)
    test_dataset = TensorDataset(X_test, y_test)
    test_loader = DataLoader(test_dataset, batch_size=test_batch_size)
    ### ========== TODO : END ========== ###

    return train_loader, val_loader, test_loader
```

4.2 One Layer Neural Network [11 pts]

For one-layer network, we consider a network from input dimension to the output dimension. In other words, we learn a 6×1 weight matrix $\mathbf{W}$. Given a $\mathbf{x}_n$, we can compute the probability of output as $\mathbf{p}_n = \sigma(\mathbf{W}^\top \mathbf{x}_n)$, where $\sigma(.)$ is the sigmoid function.

We will train our network using the *cross entropy loss*. For recap,

$$-\sum_{n=0}^{N}\sum_{c=0}^{C} \mathbb{1}(c = y_n) \log(\mathbf{p}_{n,c})$$

where N is the number of examples, C is the number of classes, and $\mathbb{1}$ is the indicator function.

(b) **(5 pts)** Prove the equivalence of the current setup of the one-layer network with a logistic regression model. Mainly, we want to show that the predicted output and the loss terms are equivalent for both the models for a given $(\mathbf{x}_n, y_n)$.

Solution:

Predicted output:

$$P_{\text{NN}}(y = 1) = \sigma(\mathbf{w}^T \mathbf{x_n})$$
$$P_{\text{Logistic}}(y = 1) = \sigma(\mathbf{w}^T \mathbf{x_n} + b)$$

$\sigma(\mathbf{w}^T \mathbf{x_n}) = \sigma(\mathbf{w}^T \mathbf{x_n} + b)$ if the bias term, b, is folded into $\mathbf{w}$ for the NN

Loss function:

$$\mathcal{L}_{\text{NN}} = -\sum_{n=0}^{N}\sum_{c=0}^{C} \mathbb{1}(c = y_n) \log(\mathbf{p}_{n,c})$$

$$\mathcal{L}_{\text{Logistic}} = \arg\max_{\mathbf{w},b} -\sum_{n=0}^{N} \log(1 + \exp(-y_i(\mathbf{w}^T \mathbf{x_i} + b)))$$

$$\begin{aligned}
\mathcal{L}_{\text{NN}} &= \arg\max_{\mathbf{w},b} -\sum_{n=0}^{N}\sum_{c=0}^{C} \mathbb{1}(c = y_n) \log(\mathbf{p}_{n,c}) \\
&= \arg\max_{\mathbf{w},b} -\sum_{n=0}^{N} \mathbb{1}(0 = y_n) \log(\mathbf{p}_{n,0}) + \mathbb{1}(1 = y_n) \log(\mathbf{p}_{n,1}) \\
&= \arg\max_{\mathbf{w},b} -\sum_{n=0}^{N} 0 \times \log(\mathbf{p}_{n,0}) + 1 \times \log(\mathbf{p}_{n,1}) \\
&= \arg\max_{\mathbf{w},b} -\sum_{n=0}^{N} \log(\mathbf{p}_{n,1}) \\
&= \arg\max_{\mathbf{w},b} -\sum_{n=0}^{N} \log(\sigma(\mathbf{w}^T \mathbf{x_n} + b)) \\
&= \arg\max_{\mathbf{w},b} -\sum_{n=0}^{N} \log(1 + \exp(-(\mathbf{w}^T \mathbf{x_i} + b))
\end{aligned}$$

$$\arg\max_{\mathbf{w},b} -\sum_{n=0}^{N} \log(1+\exp(-(\mathbf{w}^T\mathbf{x_i}+b)) = \arg\max_{\mathbf{w},b} -\sum_{n=0}^{N} \log(1+\exp(-y_i(\mathbf{w}^T\mathbf{x_i}+b))) \text{ if } y \in \{0,1\}$$

(c) **(3 pts)** Implement the constructor of `OneLayerNetwork` with `torch.nn.Linear` and implement the `forward` function to compute the outputs of the single fully connected layer i.e. $\sigma(\mathbf{W}_1^\top \mathbf{x}_n)$. Notice that we use the sigmoid function here as the activation.

You can refer to https://pytorch.org/docs/stable/generated/torch.nn.Linear.html for more information about `torch.nn.Linear` and https://pytorch.org/docs/stable/generated/torch.nn.Sigmoid.html for more information about `torch.nn.Sigmoid`.

Solution:

```
class OneLayerNetwork(torch.nn.Module):
    def __init__(self, input_features):
        # input_features: int
        super(OneLayerNetwork, self).__init__()

        ### ========== TODO : START ========== ###
        ### part c: implement OneLayerNetwork with torch.nn.Linear. Use sigmoid as the activation
        self.linear = torch.nn.Linear(input_features, 1)
        self.sigmoid = torch.nn.Sigmoid()
        ### ========== TODO : END ========== ###

    def forward(self, x):
        # x.shape = (n_batch, n_features)

        ### ========== TODO : START ========== ###
        ### part c: implement the foward function
        x = self.linear(x)
        outputs = self.sigmoid(x)
        ### ========== TODO : END ========== ###
        return outputs
```

(d) **(3 pts)** In this part, we will create an instance of the model, set up the loss criterion and initialize the optimizer as well. Create an instance of `OneLayerNetwork`, set up a criterion with `torch.nn.BCELoss` with the aggregation of loss as the sum (see `reduction` parameter), and set up a stochastic gradient descent (SGD) optimizer with learning rate as a parameter using `torch.optim.SGD` [2 pts]

You can refer to https://pytorch.org/docs/stable/optim.html for more information about `torch.optim.SGD`. Since we have only two classes, we use the Binary Cross Entropy Loss - `torch.nn.BCELoss`. We can refer to https://pytorch.org/docs/stable/generated/torch.nn.BCELoss.html for more information.

Solution:

```
def init_oneLayerNN(in_features, lr):
    # input_features: int -> Number of input features
    # lr: float -> Learning Rate

    ### ========== TODO : START ========== ###
    ### part d: prepare the OneLayerNetwork model, criterion, and optimizer
    model = OneLayerNetwork(in_features)
    criterion = torch.nn.BCELoss(reduction='sum')
    optimizer = torch.optim.SGD(model.parameters(), lr=lr)
    ### ========== TODO : END ========== ###
    return model, criterion, optimizer
```

4.3 Two Layer Neural Network [6 pts]

For one-layer network, we consider a network from input dimension to a hidden dimension and then back to the output dimension.In other words, the first layer will consist of a fully connected layer with $6 \times h$ weight matrix $\mathbf{W}_1$ and a second layer consisting of $h \times 1$ weight matrix $\mathbf{W}_2$ where h is the size of the hidden layer. Given a $\mathbf{x}_n$, we can compute the probability vector $\mathbf{p}_n = \sigma(\mathbf{W}_2^\top \sigma(\mathbf{W}_1^\top \mathbf{x}_n))$, where $\sigma(.)$ is the sigmoid function.

(e) **(4 pts)** Implement the constructor of `TwoLayerNetwork` with `torch.nn.Linear` and implement the `forward` function to compute the outputs i.e. $\sigma(\mathbf{W}_2^\top \sigma(\mathbf{W}_1^\top \mathbf{x}_n))$. Notice that we use the sigmoid function here as the activation for both layers.

Solution:

```
class TwoLayerNetwork(torch.nn.Module):
    def __init__(self, input_features, hidden_features, first_activation='sigmoid'):
        # input_features: int -> Number of input features
        # hidden_features: int -> Size of the hidden layer
        # first_activation: str -> Activation to use for the first hidden layer

        super(TwoLayerNetwork, self).__init__()
        ### ========== TODO : START ========== ###
        ### part e: implement TwoLayerNetwork with torch.nn.Linear. Use sigmoid as the activation for both layers
        self.linear1 = torch.nn.Linear(input_features, hidden_features)
        self.sigmoid1 = torch.nn.Sigmoid()
        self.linear2 = torch.nn.Linear(hidden_features, 1)
        self.sigmoid2 = torch.nn.Sigmoid()
        ### ========== TODO : END ========== ###

    def forward(self, x):
        # x.shape = (n_batch, n_features)

        ### ========== TODO : START ========== ###
        ### part e: implement the foward function
        x = self.linear1(x)
        x = self.sigmoid1(x)
        x = self.linear2(x)
        outputs = self.sigmoid2(x)
        ### ========== TODO : END ========== ###
        return outputs
```

(f) **(2 pts)** Create an instance of `TwoLayerNetwork`, set up a criterion with `torch.nn.BCELoss` with the aggregation of loss as the sum (see `reduction` parameter), and set up a stochastic gradient descent (SGD) optimizer with learning rate as a parameter using `torch.optim.SGD` [2 pts]

Solution:

```
def init_twoLayerNN(in_features, hidden_size, first_activation, lr):
    # input_features: int -> Number of input features
    # hidden_features: int -> Size of the hidden layer
    # first_activation: str -> Activation to use for the first hidden layer
    # lr: float -> Learning Rate

    ### ========== TODO : START ========== ###
    ### part f: prepare the TwoLayerNetwork model, criterion, and optimizer
    model = TwoLayerNetwork(in_features, hidden_size, first_activation)
    criterion = torch.nn.BCELoss(reduction='sum')
    optimizer = torch.optim.SGD(model.parameters(), lr=lr)
    ### ========== TODO : END ========== ###

    return model, criterion, optimizer
```

4.4 Training the Model [8 pts]

Once we have initialized both the models, we will train the models. We will be using mini-batch SGD for training the models i.e. instead of computing gradients and updating weights using all examples, we will compute them for a smaller subset. Mini-batch SGD is fast in computation and helps fast convergence while not being too random in terms of gradient directions. You can read up more about the differences of gradient descent and mini-batch SGD here - https://towardsdatascience.com/batch-mini-batch-stochastic-gradient-descent-7a62ecba642a.

(g) **(8 pts)** We provide you with a base code for batching using the dataloaders created before. You have to implement the training process. This includes initializing gradients to zeros, forward pass over the model, computing the loss, computing the gradients using backpropagation, and updating model parameters. You can refer to following tutorial to understand what each of these steps do - https://pytorch.org/tutorials/recipes/recipes/zeroing_out_gradients.html.

If you implement everything correctly, after running the `train` function (later in the colab), you should get results similar to the following (numbers can be different, but they should be non-zero). [8 pts]

```
Start training OneLayerNetwork...
| epoch  1 | train loss 0.546384 | train acc 0.732357 | valid loss ...
| epoch  2 | train loss 0.537560 | train acc 0.755881 | valid loss ...
| epoch  3 | train loss 0.533699 | train acc 0.761309 | valid loss ...
...
```

Solution:

```python
def train(model, criterion, optimizer, train_loader, valid_loader, num_epochs, logging_epochs=1):
    print("Start training model...")

    train_loss_list = []
    valid_loss_list = []
    train_acc_list = []
    valid_acc_list = []
    for epoch in range(1, num_epochs+1):
        model.train()
        for batch_X, batch_y in train_loader:
            ### ========== TODO : START ========== ###
            ### part g: Build the training paradigm - Zero out gradients, forward pass, compute loss, loss backward, update model
            optimizer.zero_grad()
            outputs = model(batch_X)
            loss = criterion(outputs, batch_y)
            loss.backward()
            optimizer.step()
            ### ========== TODO : END ========== ###

        train_loss = evaluate_loss(model, criterion, train_loader)
        valid_loss = evaluate_loss(model, criterion, valid_loader)
        train_acc = evaluate_acc(model, train_loader)
        valid_acc = evaluate_acc(model, valid_loader)
        train_loss_list.append(train_loss)
        valid_loss_list.append(valid_loss)
        train_acc_list.append(train_acc)
        valid_acc_list.append(valid_acc)

        if logging_epochs > 0 and epoch % logging_epochs == 0:
            print(f"| epoch {epoch:2d} | train loss {train_loss:.6f} | train acc {train_acc:.6f} | valid loss {valid_loss:.6f} | valid acc {valid_acc:.6f}

    return train_loss_list, valid_loss_list, train_acc_list, valid_acc_list
```

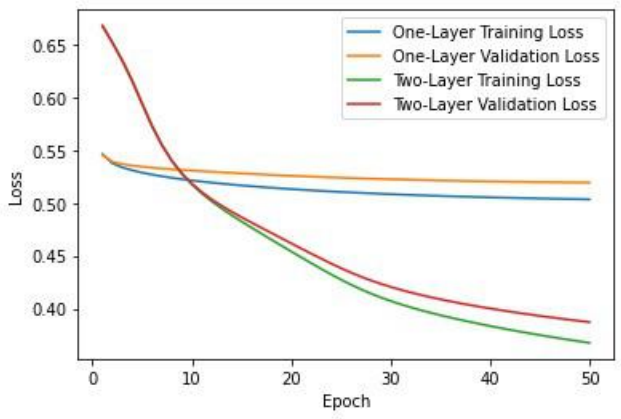
4.5 Putting it all together - Performance Evaluation [20 pts]

Now we will run the main code and call these various functions (we have given the code, you have to run it). You should train both the models using the default hyperparameters i.e.

```
lr = 0.001
num_epochs = 50
hidden_size = 6
activation = 'sigmoid'
```

(h) **(6 pts)** Now we will compare the performance of both the models by plotting the loss and the accuracies. Generate a plot depicting how `one_train_loss`, `one_valid_loss`, `two_train_loss`, `two_valid_loss` varies with epochs. Include the plot in the report and describe your findings.

Solution:

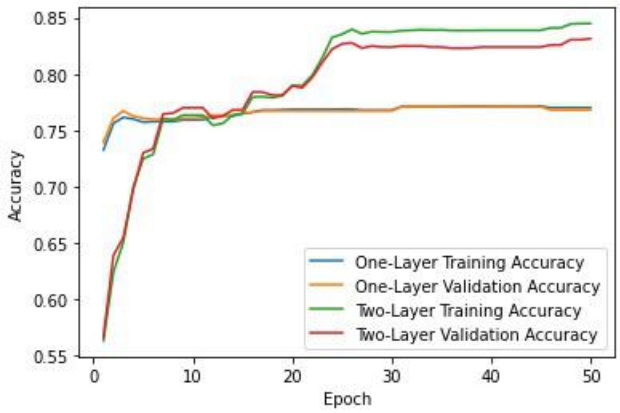


In this plot, we see that the one-layer NN loss started out significantly smaller than the two-layer NN loss. However, as the NN was trained, the two-layer loss fell below the one-layer loss around epoch 10. In addition, the one-layer loss seems to plateau, while the two-layer loss seems to exhibit a significant trend downwards, even at epoch 50. For both NNs, the training loss seems to end up lower than the validation loss.

Generate a plot depicting how `one_train_acc`, `one_valid_acc`, `two_train_acc`, `two_valid_acc` varies with epochs. Include the plot in the report and describe your findings.

Solution:

In this plot, we see that the one-layer NN accuracy started out significantly higher than the two-layer NN accuracy. However, as the NN was trained, the two-layer accuracy rose above the one-layer accuracy around epoch 15. In addition, the one-layer accuracy seems to plateau early on, while the two-layer accuracy increases consistently until around epoch 25. For both NNs, the training accuracy seems to end up higher than the validation accuracy, although the difference is much smaller for the one-layer NN. Unlike the loss plot, the trend we observe here isn't monotonic, and forms jagged edges.



(i) **(4 pts)** Calculate and report the test accuracy of both the one-layer network and the two-layer network. Explain why we get such results.

Solution:

```
One-Layer NN Test Accuracy:
        -- 0.778
Two-Layer NN Test Accuracy:
        -- 0.835
```

We get these results because the two-layer neural network has done a better job at fitting the dataset (as seen by the plots from `(h)`), so the test accuracy of the two-layer NN is higher than the test accuracy of the one-layer NN.

(j) **(5 pts)** Model analysis is a really important component for improving the model performance. In this part, you will implement the function for confusion matrix and create the confusion matrix for the validation data. Report the confusion matrix and your observations for future improvements. You may utilize the `confusion_matrix` library from sklearn - https://scikit-learn.org/stable/modules/generated/sklearn.metrics.confusion_matrix.html.

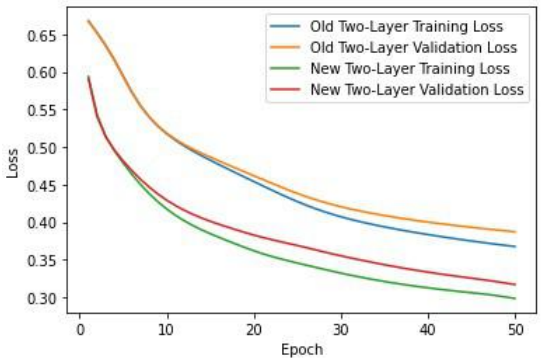
Solution:

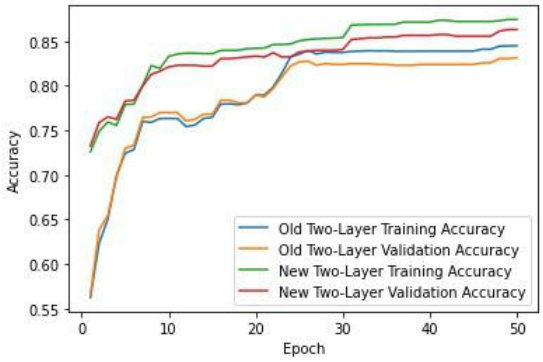
```
Confusion Matrix for One Layer NN:
[[358 164]
 [ 85 466]]


Confusion Matrix for Two Layer NN:
[[387 135]
 [ 46 505]]
```

In both confusion matrices, we see that there is a significant number of false positives (135 out of 640 negative samples for the two-layer NN). This tells us that our model currently predicts too many samples as positive, and this should be a factor in how we improve it in the future.

(k) **(5 pts)** Neural networks are sensitive to hyperparameters. In this part, you will tune some hyperparameters to improve the model performance further. We will consider only the `TwoLayerNetwork` for the tuning. By changing only the default hyperparameters specified above, show an improvement in the model performance (an improvement of 1 point in test accuracy will be awarded full credit). Report your old and new model training/validation loss curves, accuracy curves and test accuracies. Also report the new tuned hyperparameter values.

Solution:






```
Old Two-Layer NN Test Accuracy:
        -- 0.835
New Two-Layer NN Test Accuracy:
        -- 0.866
Improvement:
        -- 0.031
```

```
# Define hyper-parameters
lr = 0.003 # changed from 0.001
hidden_size = 6
activation = 'sigmoid'
num_epochs = 50
```