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:

$$\begin{aligned}
\frac{d\sigma(z)}{dz} &= \frac{d}{dz}(1 + \exp(-z))^{-1} \\
&= -(1 + \exp(-z))^{-2} \cdot \frac{d}{dz}(1 + \exp(-z)) \\
&= -(1 + \exp(-z))^{-2} \cdot (-\exp(-z)) \\
&= (1 + \exp(-z))^{-1} \cdot \Big(\exp(-z) \cdot (1 + \exp(-z))^{-1}\Big) \\
&= \sigma(z) \cdot (1 - \sigma(z))
\end{aligned}$$

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 algorithm 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: $\delta_n = 1 - 0.9933 = 0.0067$.

Logistic regression with SGD: $w_j \leftarrow w_j + 0.0067\alpha \cdot x_{n,j}$.

Perceptron algorithm: no update since $\hat{y} = \text{sgn}(\mathbf{w}^\top \mathbf{x}_i + b) = y$.

Rubric:

i. (+1) Correct residue
ii. (+2) Correct logistic regression update (+1 if the sign is incorrect)
iii. (+2) Correct Perceptron update
iv. Not necessary to write update for bias

(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: $\delta_n = 1 - 0.2689 = 0.7311$.

Logistic regression with SGD: $w_j \leftarrow w_j + 0.7311\alpha \cdot x_{n,j}$.

Perceptron algorithm: $w_j \leftarrow w_j + \alpha \cdot x_{n,j}$ since $\hat{y} = \text{sgn}(\mathbf{w}^\top \mathbf{x}_i + b) = -1 \neq y$.

Rubric:

i. (+1) Correct residue
ii. (+2) Correct logistic regression update (+1 if the sign is incorrect)
iii. (+2) Correct Perceptron update (+1 if there is no learning rate α/if the sign is incorrect)
iv. Not necessary to write update for bias

(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 difference is the factor term (δ_n) determining the influence of input data ($\boldsymbol{x}_i$) in the update step. For Perceptron algorithm, the magnitude of the factor is always constant if the data point is incorrectly classified and 0 if correctly classified. For SGD update, the contribution of the data point depends on the magnitude of the residue.

Rubric:

i. (+1) for stating it's different.

ii. (+2) for pointing out the influence of x_i for Perceptron algorithm is constant while the influence for SGD depends on the residue. (+1) for only pointing out that Perceptron does not update when making a correct prediction without discussing the update of incorrect predictions.

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) The magnitude of the weight w_j will be encouraged to grow continuously during each step of gradient descent and can go to infinity or -infinity.

(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: $w_j \leftarrow w_j - \alpha\frac{\partial J(\mathbf{w},b)}{\partial w_j}$, where $\frac{\partial J(\mathbf{w},b)}{\partial w_j} = \frac{1}{N}\sum_{i=1}^{N}\Big(\sigma\big(y_i(\boldsymbol{w}^T\boldsymbol{x}_i + b)\big) - 1\Big)y_i x_{i,j} + 0.2w_j$

Rubric:

i. (+5) if correct.

ii. (+5) Also correct:

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha\frac{\partial J(\mathbf{w},b)}{\partial \mathbf{w}}, \text{ where } \frac{\partial J(\mathbf{w},b)}{\partial \mathbf{w}} = \frac{1}{N}\sum_{i=1}^{N}\Big(\sigma\big(y_i(\boldsymbol{w}^T\boldsymbol{x}_i + b)\big) - 1\Big)y_i\mathbf{x}_i + 0.2\mathbf{w}$$

.

iii. (+2) If coefficient is correct for w_j (0.2) but there is sum over j before w_j.

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

Solution: It prevents weights from going to infinity by reducing the magnitude of the weight at every iteration.

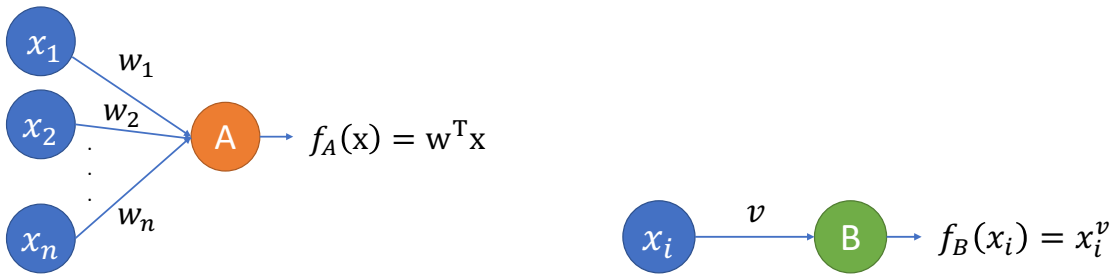
Rubric:

i. (+5) Anything related to preventing the weight from growing to infinity or negative infinity: adding penalty to the weight; reducing weight magnitude at each iteration; ect.

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 $\mathbf{w} = [w_1, w_2, w_3]^T$.

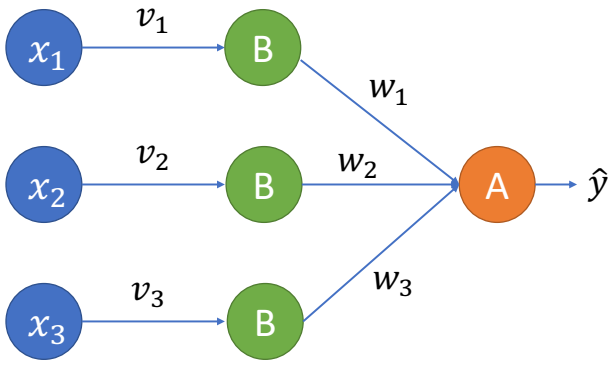


Figure 2: Architecture of the neural network.

(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:** $w_1 x_1^{v_1} + w_2 x_2^{v_2} + w_3 x_3^{v_3}$
Rubric:

i. 4 if answer is correct
ii. 2 if minor errors in the form
iii. 0 otherwise

(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:** 3
Rubric:

i. 2 if answer is correct
ii. 0 otherwise

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:** $\frac{\partial \hat{y}}{\partial w_1} = x_1^{v_1} = 1$ and $\frac{\partial \hat{y}}{\partial v_1} = w_1 x_1^{v_1} \log x_1 = 0$
Rubric:

i. 5 if both answers are correct
ii. 4 if expression form is correct but final numerical answer is wrong
iii. 2.5 if only one answer is correct or minor errors in forms for both answers
iv. 0 otherwise

(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:** $\frac{\partial \mathcal{L}}{\partial \hat{y}} = -2(y - \hat{y}) = -6$
Rubric:

i. 3 if answer is correct
ii. 1.5 if minor errors in the form/answer
iii. 0 otherwise

(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:** $\frac{\partial \mathcal{L}}{\partial w_1} = -6$ and $\frac{\partial \mathcal{L}}{\partial v_1} = 0$
Rubric:

i. 3 if both answers are correct
ii. 1.5 if only one answer is correct

iii. 0 otherwise

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:** $w_1 = 2$ and $v_1 = 1$
Rubric:

i. 3 if both answers are correct

ii. 1.5 if only one answer is correct

iii. 0 otherwise

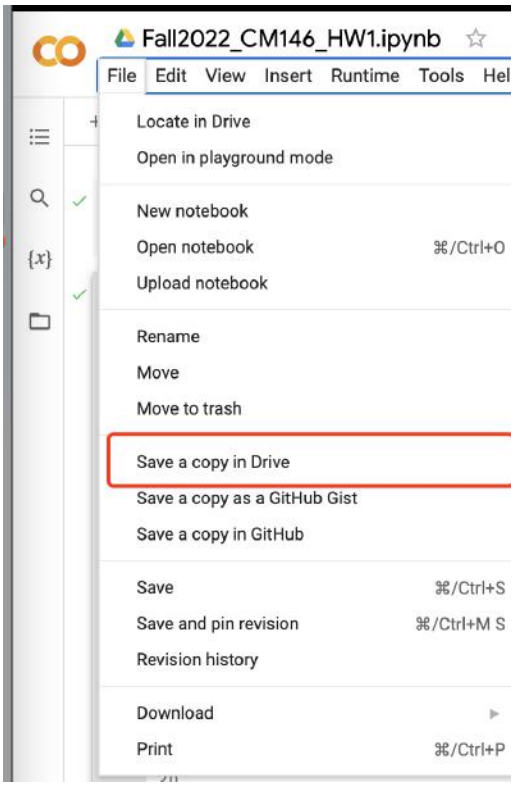
4 Programming exercise : Implementing Logistic Regression and Neural Networks [50 pts]

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 validation/test batch size to 32. Submit the code of this function as part of your report.

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

Solution:
train_dataset = TensorDataset(X_train, y_train)
train_loader = DataLoader(train_dataset, batch_size=train_batch_size)
(Similarly for val and test)

Rubric:

i. 5 if some code using TensorDataset and DataLoader classes as shown above

ii. 3 if minor mistakes in the code

iii. 0 otherwise

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:

The outputs for logistic regression is given as $P(y_n = 1) = \sigma(\mathbf{W}^\top \mathbf{x}_n) = p_n$. Thus the outputs for one layer neural network and logistic regression are the same.

For loss, we can write the cross-entropy as

$$\begin{aligned}
\mathcal{L} &= -\sum_{n=0}^{N}\sum_{c=0}^{C} \mathbb{1}(c = y_n) \log(\mathbf{p}_{n,c}) \\
&= -\sum_{n=0}^{N} \mathbb{1}(y_n = 0) \log(\mathbf{p}_{n,0}) + \mathbb{1}(y_n = 1) \log(\mathbf{p}_{n,1}) \\
&= -\sum_{n=0}^{N} \mathbb{1}(y_n = 0) \log\left(1 - \mathbf{p}_{n,1}\right) + \mathbb{1}(y_n = 1) \log\left(\mathbf{p}_{n,1}\right) \\
&= -\sum_{n=0}^{N} \mathbb{1}(y_n = 0) \log\left(1 - \frac{1}{1 + e^{\mathbf{W}^\top \mathbf{x}_n}}\right) + \mathbb{1}(y_n = 1) \log\left(\frac{1}{1 + e^{\mathbf{W}^\top \mathbf{x}_n}}\right) \\
&= -\sum_{n=0}^{N} \mathbb{1}(y_n = 0) \log\left(\frac{1}{1 + e^{-\mathbf{W}^\top \mathbf{x}_n}}\right) + \mathbb{1}(y_n = 1) \log\left(\frac{1}{1 + e^{\mathbf{W}^\top \mathbf{x}_n}}\right) \\
&= -\sum_{n=0}^{N} \mathbb{1}(y_n = -1) \log\left(\frac{1}{1 + e^{-\mathbf{W}^\top \mathbf{x}_n}}\right) + \mathbb{1}(y_n = 1) \log\left(\frac{1}{1 + e^{\mathbf{W}^\top \mathbf{x}_n}}\right) \\
&\qquad \text{...Converting } c = 0 \text{ label to } c = -1 \\
&= -\sum_{n=0}^{N} \log\left(\frac{1}{1 + e^{y_n \mathbf{W}^\top \mathbf{x}_n}}\right)
\end{aligned}$$

which is equivalent to the MLE based loss for logistic regression

Rubric:

i. (+1) Outputs of logistic regression and One layer neural network are the same

ii. (+2) (Proving loss equivalance) Successful conversion of indicator function (or addition of indicator function to logistic loss term)

iii. (+2) (Proving loss equivalance) Successful conversion of the labels from 0 to -1 or vice-versa. Completing the proof

(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. Submit the code of this function as part of your report.

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:
init:
self.linear = torch.nn.Linear(input_features, 1)
self.output_activation = torch.nn.Sigmoid()
forward:
outputs = self.output_activation(self.linear(x))

Rubric:

i. 3 if some code using LINEAR and SIGMOID classes as shown above
ii. 2 if minor mistakes
iii. 1 if attempted to write some code / incorrect architecture
iv. 0 otherwise

(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`. Submit the code of this function as part of your report.

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:
model = OneLayerNetwork(in_features)
criterion = torch.nn.BCELoss(reduction='sum')
optimizer = torch.optim.SGD(params=model.parameters(), lr=lr)

Rubric:

i. 3 if some code using ONELAYERNETWORK, BCELOSS and SGD classes as shown above
ii. 1 if attempted to write some code / incorrect architecture
iii. 0 otherwise

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. Submit the code of this function as part of your report.

Solution:
init:
self.linear1 = torch.nn.Linear(input_features, hidden_features)
self.activation = torch.nn.Sigmoid()
self.linear2 = torch.nn.Linear(hidden_features, 1)
forward:
outputs = self.activation(self.linear2(self.activation(self.linear1(x))))

Rubric:

i. 3 if some code using LINEAR and SIGMOID classes as shown above (verify if architecture is correct)
ii. 2 if minor mistakes
iii. 1 if attempted to write some code / incorrect architecture
iv. 0 otherwise

(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`. Submit the code of this function as part of your report.

Solution:
model = TwoLayerNetwork(in_features, hidden_size, first_activation=activation)
criterion = torch.nn.BCELoss(reduction='sum')
optimizer = torch.optim.SGD(params=model.parameters(), lr=lr)

Rubric:

i. 3 if some code using ONELAYERNETWORK, BCELOSS and SGD classes as shown above
ii. 1 if attempted to write some code / incorrect architecture
iii. 0 otherwise

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). Submit the code of this function as part of your report.

```
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:
optimizer.zero_grad()
preds = model(batch_X)
loss = criterion(preds, batch_y)
loss.backward()
optimizer.step()

Rubric:

(a) 8 if the code is exactly or almost similar to the solution code

(b) 5 if minor mistakes made

(c) 2 if attempted to write code but totally wrong or major mistake

(d) 0 otherwise

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:

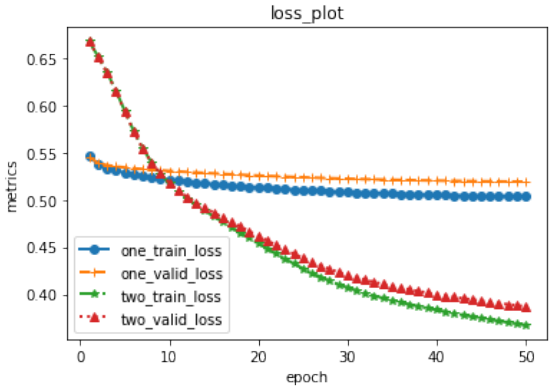


Figure 3: Loss Plot

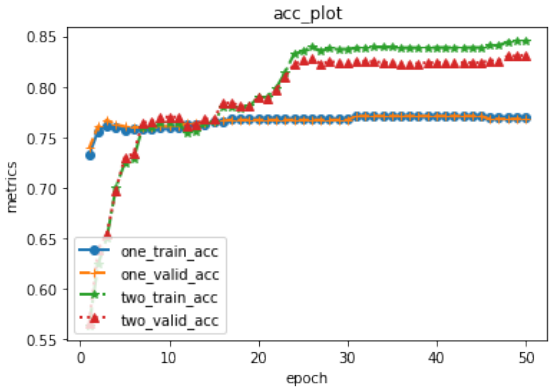


Figure 4: Accuracy Plot

Rubric:

i. (+2) if one layer loss and accuracy saturates

ii. (+2) if two layer loss reduces and accuracy increases/saturates (ensure accuracy is better than one layer)

iii. (+1) if train loss is lesser than validation for both the models

iv. (+1) if two-layer loss starts above one-layer and two-layer accuracy starts below one-layer

v. (-1) if the plot is majorly unlabeled or difficult to interpret

(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: 0.77 Two Layer NN: 0.83

Rubric:

i. (+1.5) if the reported results are more (or nearly equal) than the numbers above (for One Layer NN)
ii. (+1.5) if the reported results are more (or nearly equal) than the numbers above (for Two Layer NN)
iii. (+1) if explanation is on the lines of two layer NN is more powerful/has more parameters/is more expressive.

(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

$$\begin{bmatrix} 358 & 164 \\ 85 & 466 \end{bmatrix}$$

Confusion Matrix for Two Layer NN

$$\begin{bmatrix} 387 & 135 \\ 46 & 505 \end{bmatrix}$$

Rubric: Overall, numbers individually might vary. Need to verify trend.

i. (+2) One of the non-diagonals is high (i.e. high false positives or high false negatives)
ii. (+2) Two Layer NN improves the diagonal values and reduces the non-diagonal values
iii. (+1) Future improvements discuss reducing the false positive/false negatives

(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:
Many ways to achieve the model improvement. Major ways include increasing the number of epochs, increasing the learning rate, increasing hidden size, changing activation to 'tanh'.

Rubric:

i. (+2) Loss curves show that the train and validation loss for new model has reduced

ii. (+2) Accuracy curves show that the train and validation accuracies have increased for the new model

iii. (+1) Test accuracy is higher for the new model