Project 2 Report

A. Explain the concept and code detail of your MLP model, including MLPClassifier, activation functions, optimizers (30%)

- **MLPClassifer**
  - `__init__`
    - `self.layers` defines the number of nodes in each layer (input, hidden, and output).
    - `self.activate_function` applies nonlinear transformations, enabling the model to learn complex patterns. It contains both activation and derivative methods.
    - `self.optimizer` is used to update weights during training.
    - `self.learning_rate` scales updates to avoid overly large or small changes in the weights.
    - `self.n_epoch` defines the number of full passes through the dataset.
    - `self.use_activation` is a boolean value to decide the usage of the activation function.
    - `self.weights` and `self.bias` are initialized with a Gaussian distribution.
    - `self.deltas` is defined as a list with the length the same as `self.weights`

```
def __init__(self, layers, activate_function, optimizer, learning_rate, n_epoch =
1000, use_activation=True):
    self.layers = layers
    self.activate_function = activate_function
    self.optimizer = optimizer
    self.learning_rate = learning_rate
    self.n_epoch = n_epoch
    self.use_activation = use_activation
    self.weights = []
    self.bias = []
    self.layer_inputs = []

    for i in range(len(layers) - 1):
        self.weights.append(np.random.randn(layers[i], layers[i + 1]))
        self.bias.append(np.random.randn(layers[i + 1]))

    self.deltas = [None] * len(self.weights)
```

- `fit`

  Performs training over multiple epochs.

  - For each epoch:
    - First, calls `forwardPasss` to get predictions for the current batch.
    - Then, it computes the gradients via `backwardPass`.
    - Finally, it uses `update` to update weights and biases.

```
def fit(self, X_train, y_train):
    """ Fit method for MLP, call it to train your MLP model """
    for _ in range(self.n_epoch):
        for X_batch, y_batch in zip(X_train, y_train):
            self.forwardPass(X_batch)
            self.backwardPass(y_batch)
            self.update()
```

- `forwardPass`

  The forward pass computes the output by propagating inputs through each layer.

  a. We start with the initial input `X` and store it in `self.layer_inputs`.

  b. For each layer:
     - `net` computes the weighted sum of inputs plus bias for the layer.
     - If we are using the activation function:
       - `output` applies the activation function to `net`
     - Otherwise, `net` is assigned to `output` directly.

  c. The final layer's output is returned as the prediction.

```
def forwardPass(self, X):
    self.layer_inputs = [X]

    for i in range(len(self.weights)):
        net = np.dot(self.layer_inputs[-1], self.weights[i]) + self.bias[i]
        if self.use_activation:
            output = self.activate_function.activate(net)
        else:
            output = net
        self.layer_inputs.append(output)

     return self.layer_inputs[-1]
```

- `backwardPass`

The backward pass computes the gradients to adjust weights and biases based on the error between predictions and true values.

- For the output layer, we calculate the error between the predicated and actual values to get the delta for the last layer if we are using the activation function.
- For each hidden layer, we propagate the error backward using the chain rule if we are using the activation function. Otherwise, we propagate the error directly.

```
def backwardPass(self, y):
    error = self.layer_inputs[-1] - y
    if self.use_activation:
        self.deltas[-1] = error *
self.activate_function.derivative(self.layer_inputs[-1])
    else:
        self.deltas[-1] = error

    for i in range(len(self.deltas) - 2, -1, -1):
        error = np.dot(self.deltas[i + 1], self.weights[i + 1].T)
        if self.use_activation:
            self.deltas[i] = error *
self.activate_function.derivative(self.layer_inputs[i + 1])
        else:
            self.deltas[i] = error
```

- `update`

In this function, we use the calculated deltas to adjust the weights and biases.

- For each layer:
  - Calculate the gradient using the dot product between the inputs to the layer and the deltas. Note that we are using the `i+1` th input with the `i` th delta to match the shape of the matrices.
  - Update the weights and biases using the optimizer respectively

```
def update(self):
    for i in range(len(self.weights)):
        gradient_w = np.dot(self.layer_inputs[i+1].T, self.deltas[i])
        self.weights[i] = self.optimizer.update(self.weights[i], gradient_w,
self.learning_rate)
        self.bias[i] = self.optimizer.update(self.bias[i], np.mean(self.deltas[i],
axis=0), self.learning_rate)
```

- Predictions: `predict` and `predict_proba`
  - `predict_proba` returns the probability of belonging to a certain class.
  - `predict` thresholds these probabilities to return binary class predictions (0 or 1).

```
    def predict(self, X_test):
        y_hat = self.predict_proba(X_test)
        return np.array([1 if i > 0.5 else 0 for i in y_hat])

    def predict_proba(self, X_test):
        return self.forwardPass(X_test)
```

-

Activation Functions

1. `__init__`

The constructor initializes the activation function type.

```
    def __init__(self, type='relu'):
        self.type = type
```

2. `activate`: This method applies the selected activation function to an input tensor `x`, which could be a layer's output.

```
    def activate(self, x):
        if self.type == 'relu':
            return np.maximum(0, x)
        elif self.type == 'sigmoid':
            x = np.clip(x, -500, 500)
            return 1 / (1 + np.exp(-x))
        elif self.type == 'tanh':
            return np.tanh(x)
        else:
            raise ValueError('Unsupported activation type.')
```

- ReLU: Returns `x` if positive, otherwise `0`.

$$R(x) = \begin{cases} x, & \text{if } x > 0 \\ 0, & \text{if } x \leq 0 \end{cases}$$

- Sigmoid: Maps `x` to a range (0, 1). Note that clipping `x` avoids overflow in large values.

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

- Tanh: Maps `x` to a range (-1, 1).

$$\tanh(x) = \frac{2}{1+e^{-2x}} - 1$$

3. `derivative`: This method calculates the derivative of the activation function.

```
def derivative(self, x):
    if self.type == 'relu':
        return np.where(x > 0, 1, 0)
    elif self.type == 'sigmoid':
        sig = self.activate(x)
        return sig * (1 - sig)
    elif self.type == 'tanh':
        return 1 - np.tanh(x)**2else:
    raise ValueError('Unsupported activation type.')
```

- ReLu: Returns `1` for positive values of `x`, and `0` otherwise.

$$R'(x) = \begin{cases} 1, & \text{if } x > 0 \\ 0, & \text{if } x \leq 0 \end{cases}$$

- Sigmoid: Uses the output of the sigmoid function to compute `sig * (1 - sig)`, which represents the slope at any point. This approach optimizes calculations in the backpropagation step.

$$\sigma'(x) = \sigma(x)(1-\sigma(x))$$

- Tanh: `1 - tanh(x)^2` is derived from the fact that `tanh'(x) = 1 - tanh(x)^2`.

$$\tanh'(x) = 1 - \tanh^2(x)$$

-

Optimizers

- `__init__`

  The constructor initializes the optimizer's parameters:

  - `self.type` specifies the type of the optimizer.
  - `self.momentum` is the momentum factor.

- `self.epsilon` is a small constant.
- `self.cache` and `self.velocity` are dictionaries to store momentum and accumulated gradients.

```
    def __init__(self, type='sgd'):
        self.type = type
        self.momentum = 0.9
        self.epsilon = 1e-8
        self.cache = {}
        self.velocity = {}
```

- `update`

This method adjusts weights based on the specified optimizer type.

```
    def update(self, weights, gradients, learning_rate):
        if self.type == 'sgd':
            return weights - learning_rate * gradients
        if self.type == 'momentum':
            if 'weights' not in self.velocity:
                self.velocity['weights'] = np.zeros_like(weights)
            self.velocity['weights'] = self.momentum * self.velocity['weights'] + gradients
            return weights - learning_rate * self.velocity['weights']
        if self.type == 'rmsprop':
            if 'cache' not in self.cache:
                self.cache['weights'] = np.zeros_like(weights)
            self.cache['weights'] = 0.9 * self.cache['weights'] + 0.1 * (gradients ** 2)
            return weights - learning_rate * gradients / (np.sqrt(self.cache['weights']) +
    self.epsilon)
        raise ValueError('Unsupported optimizer type.')
```

Explanation of Optimizers

- SGD
  - We update the weights using the formula:

    - `weights` are the current weights of the model.
    - `learning_rate` controls the step size of each update.
    - `gradients` are the calculated gradients.

```
        if self.type == 'sgd':
            return weights - learning_rate * gradients
```

- Momentum

- This is the accelerated SGD, it is done by taking the direction of the previous update into account. Also, it calculates an exponentially weighted moving average of past gradients using this formula:

  - `self.velocity['weights']` stores the moving average of gradients.
  - `self.momentum * self.velocity['weights'] + gradients` updates this moving average.

```
    if self.type == 'momentum':
        if 'weights' not in self.velocity:
            self.velocity['weights'] = np.zeros_like(weights)
        self.velocity['weights'] = self.momentum * self.velocity['weights'] +
gradients
        return weights - learning_rate * self.velocity['weights']
```

- RMSprop
  - This optimizer adapts the learning rate by dividing the gradient by the square root of the accumulated squared gradients based on this formula:

    - `self.cache['weights']` accumulates the squared gradients using a decay rate of `0.9` for past gradients.
    - `np.sqrt(self.cache['weights']) + self.epsilon` stabilizes updates by scaling down large gradients and boosting small ones.

```
    if self.type == 'rmsprop':
        if 'cache' not in self.cache:
            self.cache['weights'] = np.zeros_like(weights)
        self.cache['weights'] = 0.9 * self.cache['weights'] + 0.1 * (gradients **
2)
        return weights - learning_rate * gradients /
(np.sqrt(self.cache['weights']) + self.epsilon)
```

B. Please answer the following questions (20%)

1. **What is gradient explosion, if you encounter this problem, how do you solve it? Please be precise and concise. (10%)**

   Gradient explosion is a problem that occurs when the gradients of the loss function accumulate and result in extremely large updates to neural network model weights during training. There are multiple solutions to this problem:

   a. Limit the maximum value of the gradients by setting a threshold.

b. Change activation functions that are less prone to saturation.

c. Use lower learning rate.

2. **What is overfitting? If you encounter this problem, how do you solve it? Please be precise and concise. (10%)**

   Overfitting occurs when a model learns the training data too well. The model would catch noises and outliers in the dataset, which may lead to poor generalization to new data. Here are some solutions to the problem:

   a. Feature selection, ignore the irrelevant data to avoid noise.

   b. Increase the size and diversity of the training dataset through rotation, scaling, etc.

C. Discussion (10%)

1. **Compare the result with and without activation functions. Do you think activate function is necessary in neural networks?**

   a. **With Activation Function:** The model does a good job with high accuracy. The F1 score and other measurements also show that the model is performing well. Activation functions help the model learn more complex patterns in the data by adding non-linearity.

   b. **Without Activation Function:** The model does poorly without activation functions, with an accuracy of only about 51.1%. The F1 score and MCC scores are very low. Without activation functions, the network can only do simple calculations.

   c. **Do We Need Activation Functions?**

      Yes, activation functions are very important in neural networks. They allow the model to learn and understand complicated patterns in the data. Without them, the model can only solve simple problems, which limits its usefulness.

```
Activation Function: sigmoid
Acc: 0.81944
F1 score: 0.77966
MCC: 0.65416
Scoring: 0.74767
```

```
Without Activation Function
Acc: 0.65278
F1 score: 0.00000
MCC: 0.00000
Scoring: 0.19583
```

2. **Compare the results of three different activate functions.**

Sigmoid and Tanh outperformed ReLU.

Two results are identical, and here are my speculations of why this happened:

a. The weight initialization may make three models converge similarly.

b. The learning weight is too low for the model to show the difference between the functions.

c. The data is relatively simple.

```
Activation Function: sigmoid
Acc: 0.81944
F1 score: 0.77966
MCC: 0.65416
Scoring: 0.74767

Activation Function: tanh
Acc: 0.81944
F1 score: 0.77966
MCC: 0.65416
Scoring: 0.74767

Activation Function: relu
Acc: 0.65278
F1 score: 0.00000
MCC: 0.00000
Scoring: 0.19583
```

3. **Compare the results of three different optimizers.**
   - The momentum optimizer significantly outperformed both SGD and RMSProp.
   - While better than SGD, RMSProp shows improvements in accuracy (65.28%) but still has a low F1 score and MCC of 0.00000, indicating it struggles with distinguishing classes effectively.
   - SGD shows the worst performance, with an accuracy of only 58.33%. The negative MCC suggests that the model is performing worse than random guessing, which is indicated by the F1 score of 0.00.

```
Optimizer: sgd
Acc: 0.58333
F1 score: 0.00000
MCC: -0.19924
Scoring: 0.10527

Optimizer: rmsprop
Acc: 0.65278
F1 score: 0.00000
MCC: 0.00000
Scoring: 0.19583

Optimizer: momentum
Acc: 0.81944
F1 score: 0.77966
MCC: 0.65416
Scoring: 0.74767
```

D. Bonus (10%)

Implement general MLP model (n layers). That is, make your MLP model scalable so that it can scale to any depth and width. Then, Compare the result of different depths (e.g. depth= 2, 5, 10). Do you think a deeper MLP model is better in this project?

- The model with a depth of 3 (which is used initially in this project) has the best performance. It strikes a good balance between complexity and performance.
- The model with a depth of 2 performed poorly, with a negative overall scoring.
- The model with a depth of 10 had a slightly better performance. It took much time too run, which might be a severe issue to large datasets.

```
Activation Function: sigmoid
Acc: 0.81944
F1 score: 0.77966
MCC: 0.65416
Scoring: 0.74767
```

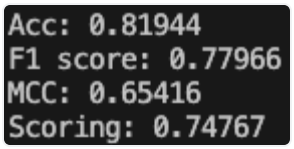
```
Layers: [8, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 1]
Acc: 0.65278
F1 score: 0.00000
MCC: 0.00000
Scoring: 0.19583
```

```
Layers: [8, 100, 100, 1]
Acc: 0.18056
F1 score: 0.06349
MCC: -0.65416
Scoring: -0.15257
```

In my opinion, a deeper MLP is not better in this project due to the following reasons:

1. As the number of layers increases, the performance gains from adding more layers may become minimal. Beyond a certain point, the benefits of increased depth can decrease.
2. Deeper networks have a higher likelihood of overfitting, especially if the training dataset is small or not diverse enough.
3. More layers mean longer training times. In my case, it took more than 1 hour to train the model.
4. Deeper models can be more complex and may require careful tuning of hyper-parameters to perform well. This added complexity can complicate the training process.

E. Performance



I've logged a lot of different results by using different activation functions, optimizer, etc, and this is the best performance from all the experiments.

Reference

1. https://d2l.ai/chapter_optimization/sgd.html
2. https://hackmd.io/@kk6333/rJ553YG1s
3. https://hackmd.io/@allen108108/H1l4zqtp4