Fundamentals of Machine Learning (Spring 2025)

Homework #4 (100 Pts, Due date: May 28)

Student ID 2021312088

Name Lee Hakmyung

Instruction: We provide all codes and datasets in Python. Once you have solved the problems, submit two files as follows.

- **'ML_HW4_YourName_STUDENTID.zip':** All codes in the directory and your document.
- **'ML_HW4_YourName_STUDENTID.pdf':** Your document converted into pdf.

NOTE: Please write your code in the 'EDIT HERE' signs. Editing other parts is not allowed.

(1) Backpropagation and MLP

(a) [15 pts] Write your code in 'model/functions.py' to implement the backpropagation method for the 'Sigmoid,' 'ReLU,' 'GELU,' 'Linear layer,' and 'Sigmoid Layer.' Use the approximated GELU for implementation. Note that GELU is a smooth **activation function** that weights each input x by the probability that a standard normal variable is below x.

$$\mathrm{GELU}(x) = xP(X \le x) = x\Phi(x), \qquad \Phi(x) = \frac{1}{2}\left(1 + \tanh\left[\sqrt{\frac{2}{\pi}}\left(x + 0.044715x^3\right)\right]\right)$$

$$\approx \; x\sigma(1.702x)$$

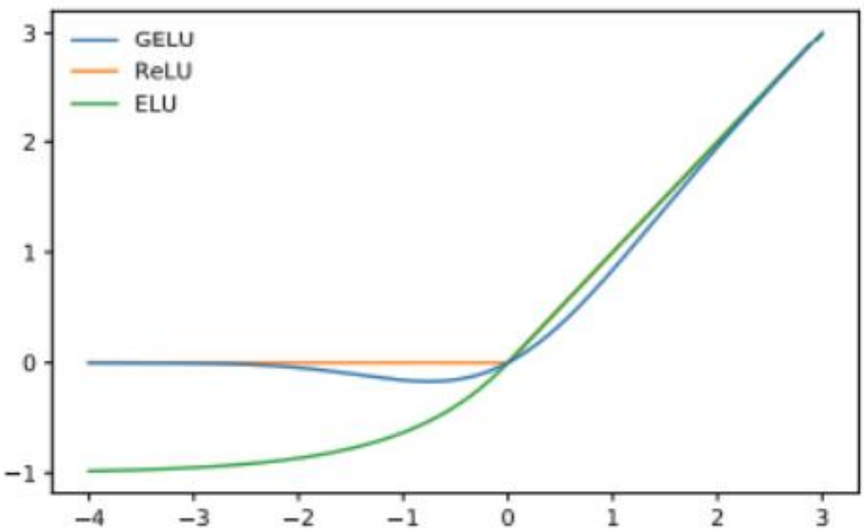

그림 1 GELU, ReLU, ELU 비교

Capture your code and explain the code based on the principle of the backpropagation method.

```python
class ReLU:
    def backward(self, d_prev):
        """
        ReLU Backward.

        z --> (ReLU) --> out
        dz <-- (dReLU) <-- d_prev(dL/dout)

        [Inputs]
            d_prev : Gradients until now.
            d_prev = dL/dk, where k = ReLU(z).

        [Outputs]
            dz : Gradients w.r.t. ReLU input z.
        """
        dz = None
        # =============== EDIT HERE ===============

        dz = d_prev.copy()
        dz[self.zero_mask] = 0

        # ========================================

        return dz
```

self.zero_mask is a boolean array that marks where the original input to ReLU was less than or equal to zero.

For those inputs, the derivative is 0, so we set the gradient to 0 accordingly.

For the rest (where input > 0), the gradient is unchanged.

```python
class GELU:
    def backward(self, d_prev):
        """
        GELU Backward using sigmoid approximation.

        z --> (GELU) --> out
        dz <-- (dGELU) <-- d_prev(dL/dout)

        [Inputs]
            d_prev : Gradients until now.
            d_prev = dL/dk, where k = GELU(z).

        [Outputs]
            dz : Gradients w.r.t. GELU input z.
        """
        dgelu = None


        # =============== EDIT HERE ===============

        x = self.x
        sigma = self.sigmoid_term
        dsigma_dx = 1.702 * sigma * (1 - sigma)
        dgelu = d_prev * (sigma + x * dsigma_dx)

        # ========================================

        return  dgelu
```

$dL/dx = dL/dy \cdot [\sigma + x \cdot 1.702 \cdot \sigma(1-\sigma)]$

self.x stores the input to GELU from the forward pass. sigma is $\sigma(1.702x)$ from the forward step.

dsigma_dx calculates the derivative of sigmoid using $\sigma(1-\sigma)$, multiplied by 1.702 due to the chain rule.

The full gradient dgelu is computed using the formula above.

```python
class Sigmoid:
    def backward(self, d_prev):
        """
        Sigmoid Backward.

        z --> (Sigmoid) --> self.out
        dz <-- (dSigmoid) <-- d_prev(dL/d self.out)

        [Inputs]
            d_prev : Gradients until now.

        [Outputs]
            dz : Gradients w.r.t. Sigmoid input z which is self.out.
        """

        dz = None
        # =============== EDIT HERE ===============

        dz = d_prev * self.out * (1 - self.out)

        # =========================================
        return dz
```

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

self.out stores the sigmoid output $\sigma(x)$. The product self.out * (1 - self.out) gives the derivative.

Then we apply the chain rule by multiplying it with d_prev, the gradient coming from the next layer

```python
class Linear:
    def backward(self, d_prev):
        """
        Linear layer backward
        x and (W & b) --> z -- (activation) --> hidden
        dx and (dW & db) <-- dz <-- (activation) <-- hidden

        - Backward of activation
        - Gradients of W, b

        [Inputs]
            d_prev : Gradients until now.

        [Outputs]
            dx : Gradients of input x
        """
        dx = None
        self.dW = None
        self.db = None
        # =============== EDIT HERE ===============

        self.dW = np.dot(self.x.T, d_prev)
        self.db = np.sum(d_prev, axis=0)
        dx = np.dot(d_prev, self.W.T)

        # =========================================

        return dx
```

For a linear layer: $z = xW+b$

During backpropagation: We receive dL/dz, the gradient of the loss with respect to the output.

self.x.T is the transposed input matrix; this allows batch matrix multiplication with d_prev

self.db sums the gradient across all samples in the batch (axis=0)

dx is the gradient to be passed to the previous layer, representing how the input should be updated

```python
class SigmoidLayer:
    def backward(self, d_prev=1):
        """
        Calculate gradients of input (x), W, b of this layer.
        Save self.dW, self.db to update later.

        x and (W & b) --> z -- (activation) --> y_hat --> Loss
        dx and (dW & db) <-- dz <-- (activation) <-- dy_hat <-- Loss

        [Inputs]
            d_prev : Gradients until here. (Always 1 since its output layer)

        [Outputs]
            dx : Gradients of output layer input x (Not MLP input x!)
        """
        batch_size = self.y.shape[0]
        d_prev= 1
        d_bce = ((self.y_hat - self.y.reshape(-1,1)) / (self.y_hat * (1 - self.y_hat))) / batch_size

        sig_d_prev= d_prev * d_bce
        lin_d_prev = None

        dx = None
        # =============== EDIT HERE ===============
        """
        you should calculate gradient of sigmoid and linear layer
        ! tip: use backward functions that you have already written!!!!
        """
        # call backward
        lin_d_prev = self.sigmoid.backward(sig_d_prev)
        dx = self.linear.backward(lin_d_prev)

        # ========================================

        self.dW = self.linear.dW
        self.db = self.linear.db
        return dx
```

Computes dL/dy^: self.y_hat is the predicted output y^. self.y is the true label.

Since d_prev = 1 (this is the final loss layer), we just multiply with d_bce. This is the total gradient that flows into the sigmoid activation.

Calls the previously defined Sigmoid.backward() and Linear.backward() methods

These methods compute the gradient:

Through the sigmoid: dL/dz=dL/dy^·y^(1−y^).

Through the linear layer: updates self.dW, self.db, and returns dL/dx.

(b) [15 pts] The given code '0_MLP.py' trains a binary classification MLP model with a **movie review sentiment analysis dataset**. It loads pre-encoded 768-dimensional users' movie review embeddings ('x_train.npy' and 'x_test.npy'). The goal is to predict the user's sentiment between positive (1) and negative (0). Dataset: https://huggingface.co/datasets/stanfordnlp/sst2

Run the '0_MLP.py' script five times with different hidden dimension sizes and activation function choices. Record the **test accuracy** for each run.
For the above three given examples, please use the default hyperparameters specified in the code.
For the last two results, you can change any hyperparameter if you want. The accuracy of the two test accuracy should be higher than that of the third given example ([64, 8], [GELU, GELU]).

Index	Num_epochs	Learning rate	Hidden dimension [dim1, dim2]	Activation	Test_Accuracy
1	100	0.01	[64, 8]	[ReLU, ReLU]	0.663
2	100	0.01	[32, 8]	[Sigmoid, Sigmoid]	0.571
3	100	0.01	[64, 8]	[GELU, GELU]	0.697
4	250	0.01	[64, 16]	[GELU, ReLU]	0.703
5	150	0.02	[128, 32]	[GELU, GELU]	0.731

Capture the screen of your 4th and 5th experiment results.

```
Train Accuracy = 0.692
Test Accuracy = 0.691
Train Accuracy = 0.693
Test Accuracy = 0.691
Train Accuracy = 0.693
Test Accuracy = 0.697
Train Accuracy = 0.694
Test Accuracy = 0.703
Train Accuracy = 0.697
Test Accuracy = 0.703
```

4th)

```
Train Accuracy = 0.770
Test Accuracy = 0.714
Train Accuracy = 0.776
Test Accuracy = 0.720
Train Accuracy = 0.776
Test Accuracy = 0.726
Train Accuracy = 0.780
Test Accuracy = 0.726
Train Accuracy = 0.780
Test Accuracy = 0.731
```

5th)

Extending the second layer from 8 to 16 units provided more capacity. GELU in the first layer likely helped with expressive features, while ReLU in the second may have supported more stable gradient flow. A longer training schedule (250 epochs) also contributed to better convergence. Although the improvement was modest, it outperformed the baseline.

Increasing both layer widths significantly boosted the model's capacity. The GELU-GELU combination worked well, especially given the BERT-style embeddings used as input. Despite a relatively high learning rate, the model trained effectively - possibly due to GELU's smoothness and the wider architecture. This setup achieved the best result among all experiments.

(2) DecisionTree

(a) [10 pts] Implement the function ‘compute_entropy’ and ‘selection_criteria’ in ‘model/DecisionTree.py.’ The entropy, conditional entropy and information gain are defined as follows:

$$H(X) = -\sum_{x \in X} p(x) \log_2 p(x),$$

$$H(Y|X) = \sum_{x \in X} p(x) H(Y|X = x), IG(Y|A) = H(Y) - H(Y|A)$$

Capture your code in the block.

```python
def compute_entropy(labels):
    entropy = None
    # ============================== EDIT HERE ==============================
    """
    Compute the entropy of the labels.
    The entropy is defined as:
    H(X) = -sum(p(x) * log2(p(x))) for all x in X
    where p(x) is the probability of x in the labels.
    """

    from collections import Counter
    label_counts = Counter(labels)
    total_count = len(labels)
    entropy = 0

    for count in label_counts.values():
        p = count / total_count
        entropy -= p * np.log2(p)

    # =======================================================================

    return entropy
```

Counter(labels) counts how many times each class appears. total_count is the total number of labels.

For each label class: It calculates the probability p of that class. Then it adds -p * log2(p) to the entropy value.

```python
class DecisionTree():
    def selection_criteria(self, df):
            # ============================== EDIT HERE ==============================

            """
            Compute the information gain of the feature.
            The information gain is defined as:
            IG(X|Y) = H(X) - H(X|Y)
            where H(X) is the entropy of the labels and H(X|Y) is the conditional entropy of the labels given the feature Y.
            The conditional entropy is defined as:
            H(X|Y) = sum(p(y) * H(X|Y=y)) for all y in Y
            """

            entropy = compute_entropy(labels)

            # Compute conditional entropy
            conditional_entropy = 0
            for val in distinct_values:
                indices = feature_data == val
                subset_labels = labels[indices]
                p = np.sum(indices) / len(labels)
                conditional_entropy += p * compute_entropy(subset_labels)

            # Compute information gain
            information_gain = entropy - conditional_entropy

            # =======================================================================

            if information_gain > max_gain:
                max_gain = information_gain
                best_feature = feature
        if max_gain <= 0:
            return None
        return best_feature
```

This block computes the conditional entropy, which shows how much uncertainty remains after splitting the data based on the current feature. distinct_values are all the unique values in the feature.

For each value: It selects the labels that belong to that value. It calculates the proportion p of data that has that value. It computes the entropy of this subset of labels. Then it adds p * entropy to the conditional entropy.

The result is a weighted sum that shows the expected entropy after the split.

(b) [10 pts] Fill in the blank using the code provided in '1_DecisionTree.py'.

Max depth	Accuracy
3	0.7240
4	0.7708
5	0.8438
6	0.8281

(c) [10 pts] Identify 1) the max depth that yields the **best performance** and 2) the max depth that **does not cause further differences in test accuracy**. Based on the train_test split and the characteristic of the "Tic-Tac-Toe" dataset, explain why this phenomenon occurs.

1)

From the table above:

The **highest accuracy** is achieved at **depth 5**.

At **depth 6**, performance slightly **decreases**, showing no further improvement.

2)

The Tic-Tac-Toe dataset has a fixed structure. Each game state follows clear logical rules, and the total number of possible game states is limited. Because of this, the decision tree can learn the important patterns without needing to grow too deep.

When the depth is increased to 5, the model captures the most important features and patterns. This depth allows it to correctly classify many different game states without memorizing too many details.

If we make the tree deeper, the model might start to **memorize** specific examples from the training data. This can lead to **overfitting**, which means the model performs well on training data but worse on new, unseen data.

Therefore: **Depth 5** gives the **best generalization**.

Going beyond depth 5 does not help, and may even hurt performance slightly.

(3) [40 pts] (Kaggle challenge) This task is the imbalanced multi-class classification problem on the Wine Quality dataset. Please use the given data to predict the class of test data. Use data sampling and learned techniques from the class to improve accuracy. Please refer to **"https://www.kaggle.com/competitions/skku-ml-2025-1-hw-4"** and follow the instructions.

If you can't connect to the site : "https://www.kaggle.com/t/c4717cafbd934a98aa08bc47ed7a8cab"

[Notes]

Please refer to the Kaggle description

- Please submit your code for the model that scores the highest in the Kaggle competition in 'Kaggle/.'

[Competition Rules]

- Do not cheat.
- Use Python.
- No limitation on Python libraries. (Pytorch, Tensorflow, etc.)
- You must use "{Student ID}_{Name}" for your team name in the Kaggle competition.
- No late submission in the Kaggle competition.
- Any use of external data is prohibited. However, you can freely utilize the given data
- Your submission to Kaggle is limited to five times a day.

Answer:

1. Introduction

This project aims to solve an **imbalanced multi-class classification** problem using the *Wine Quality Dataset* provided on Kaggle. The task is to predict wine quality scores (ranging from 4 to 8) based on several chemical properties. To achieve better performance, we applied both **deep learning** and **ensemble learning** methods introduced in class, with a focus on handling class imbalance and combining model strengths.

2. Data Preprocessing

- **Dataset**: X_train.npy, y_train.npy, and X_test.npy were provided.
- The original labels (4–8) were shifted to 0–4 for internal modeling.
- We performed **stratified train-validation split** to preserve label distribution.
- **StandardScaler** was used to normalize features for MLP training, which helps stabilize the learning process.

3. Models and Training

We trained three classifiers:

MLP Classifier (Neural Network)

The MLP (Multi-Layer Perceptron) model was implemented in the provided MLP.py file using PyTorch. It consists of three fully connected layers with batch normalization and non-linear activation.

Model Architecture:

```
self.fc1 = nn.Linear(input_size, 128)
self.bn1 = nn.BatchNorm1d(128)

self.fc2 = nn.Linear(128, 64)
self.bn2 = nn.BatchNorm1d(64)

self.fc3 = nn.Linear(64, num_classes)
self.gelu = nn.GELU()
```

- Activation Function: GELU (Gaussian Error Linear Unit), which is smoother than ReLU and often provides better performance.
- Optimizer: AdamW, Loss: CrossEntropyLoss.
- Trained for 150 epochs with early stopping and saved as best_model.pth.

Random Forest

- RandomForestClassifier with 500 trees.
- No maximum depth; used all features with "sqrt" strategy.
- Trained using default Gini impurity criterion.

XGBoost

- Gradient boosting with 800 trees, learning rate 0.05, max depth 6.
- Objective: multi:softprob, Evaluation metric: mlogloss.
- Used subsampling and column sampling to prevent overfitting.

4. Ensemble Strategy

To improve robustness and performance:

- **Soft voting ensemble** was applied by averaging the class probabilities from all three models.
- Ensemble prediction = argmax(mean([MLP, RF, XGB])).

5. Evaluation Results (on Validation Set)

```
📊 Validation 성능
  • MLP           acc: 0.6231 | macro-F1: 0.5703
  • RandomForest  acc: 0.6808 | macro-F1: 0.6103
  • XGBoost       acc: 0.6795 | macro-F1: 0.6125
  • Ensemble★     acc: 0.6859 | macro-F1: 0.6279
```

6. Conclusion

Through this project, I gained hands-on experience in:

- Handling **imbalanced classification problems**,
- Applying **MLP** with PyTorch,
- Using **ensemble techniques** such as soft voting,
- Understanding the **importance of stratified sampling** in validation,
- Practicing **real-world model evaluation** with F1-score and accuracy.