Multilayer Perceptron Classification

In machine learning, the field of artificial neural networks is often just called neural networks or multilayer perceptrons. As we have learned in class, a perceptron is a single neuron model that was a precursor to the larger neural networks that are utilized today.

The building blocks for neural networks are neurons, which are simple computational units that have input signals and produce an output signal using an activation function. Each input of the neuron is weighted with specific values, and while the weights are initially randomized, it is usually the goal of training to find the best set of weights that minimize the output error. The weights can be initialized randomly to small values, but more complex initialization schemes can be used that can have significant impacts on the classification accuracy of the models. A neuron also has a bias input that always has a value of 1.0 and it too must be weighted. These weighted inputs are summed and passed through an activation function, which is a simple mapping that generates an output value from the weighted inputs. Some common activation functions include the sigmoid (logistic) function, the hyperbolic tangent function, or the rectified linear unit function.

These individual neurons are then arranged into multiple layers that connect to each other to create a network called a neural network (or multilayer perceptron). The first layer is always the input layer that represents the input of a sample from the dataset. The input layer has the same number of nodes as the number of features that each sample in the dataset has. The layers after the input layer are called hidden layers because they are not directly exposed to the dataset inputs. The number of neurons in a hidden layer can be chosen based on what is necessary for the problem. The neurons in a specific hidden layer all use the same activation function, but different layers can use different ones. Multilayer perceptrons must have at least one hidden layer in their network.

The final layer is called the output layer and it is responsible for outputting values in a specific format. It is

common for output layers to output a probability indicating the chance that a sample has a specific target class label, and this probability can then be used to make a final clean prediction for a sample. For example, if we are classifying images between dogs and cats, then the output layer will output a probability that indicates whether dog or cat is more likely for a specific image that was inputted to the neural network. The nature of the output layer means that its activation function is strongly constrained. Binary classification problems have one neuron in the output layer that uses a sigmoid activation function to represent the probability of predicting a specific class. Multi-class classification problems have multiple neurons in the output layer, specifically one for each class. In this case, the softmax activation function is used to output probabilities for each possible class, and then you can select the class with the highest probability during prediction.

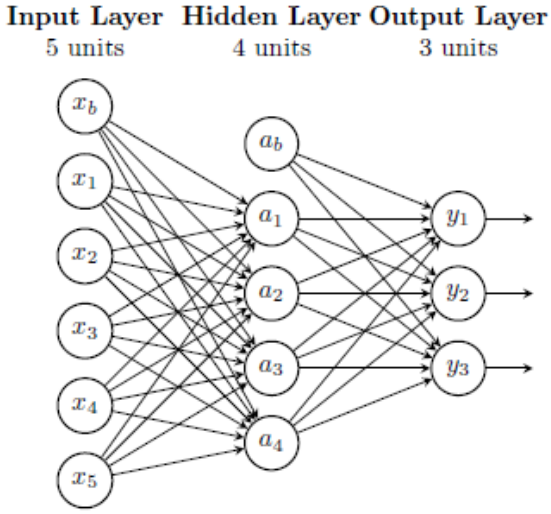
Before training a neural network, the data must be prepared properly. Frequently, the target class values are categorical in nature: for example, if we are classifying pets in an image, then the possible target class values might be either dog, cat, or goldfish. However, neural networks usually require that the data is numerical. Categorical data can be converted to a numerical representation using one-hot encoding. One-hot encoding creates an array where each column represents a possible categorical value from the original data (for the image pet classification, one-hot encoding would create three columns). Each row then has either 0s or 1s in specific positions depending on the class value for that row. Here is an example of one-hot encoding using the dog, cat, or goldfish image classification example, where we are using five test samples and looking at their target class values.

y
dog
cat
cat
goldfish
dog

one-hot encoding →

y_{dog}	y_{cat}	y_{goldfish}
1	0	0
0	1	0
0	1	0
0	0	1
1	0	0

In this assignment, we will specifically be focusing on multilayer perceptron neural networks that are feedforward, fully-connected, and have exactly three layers: an input layer, a hidden layer, and an output layer. A feedforward fully-connected network is one where each node in one layer connects with a certain weight to every node in the following layer. A diagram of such a neural network is shown below, where the input layer has five nodes corresponding to five input features, the hidden layer has four neurons, and the output layer has three neurons corresponding to three possible target class values. The bias terms are also added on as nodes named with subscript of b.



Once the data is prepared properly, training occurs using batch gradient descent. During each iteration, **forward propagation** is performed where training data inputs go through the layers of the network until an output is produced by the output layer. Frequently, the cross-entropy loss is calculated using this output and stored in a history list that allows us to see how quickly the error reduces every few iterations. The output from the output layers is then compared to the expected output (the target class values) and an error is calculated. The output error is then propagated back through the network one layer at a time, and the weights are updated according to the amount that they contributed to the error. This is called **backward propagation**. A parameter called the learning rate is typically used to control how much to change the model in response to the estimated error each time the model weights are updated. Once the maximum number of iterations is reached, the neural network is finished training and it can be used to make new predictions. A prediction is made by using new test data and computing an output using **forward propagation**. When there are multiple output neurons, the output with the highest softmax value is chosen as the predicted target class value.

What to Do. Your goal in this part is to implement a feedforward fully-connected multilayer perceptron classifier with one hidden layer (as shown in the description above) from scratch. As before, your GitHub repository contains the skeleton code for two files that will be used to implement the algorithm: `utils.py` and `multilayer_perceptron.py`.

This time, the functions you need to concern yourself with in the `utils.py` file are the unimplemented functions defined after the distance functions. Specifically, these functions are: `identity`, `sigmoid`, `tanh`, `relu`, `cross_entropy`, and `one_hot_encoding`.

The `multilayer_perceptron.py` file defines the `MultilayerPerceptron` class that we will use to implement the algorithm from scratch. Just like the previous part, the `__init__` function has already been properly implemented for you. The attributes for the class itself are described in detail in the skeleton code. When creating the `MultilayerPerceptron` object, the following parameters must be specified (or their default values will be used):

- `n_hidden`: the number of neurons in the one hidden layer of the neural network.

- `hidden_activation`: represents the activation function of the hidden layer (can be either `'identity'`, `'sigmoid'`, `'tanh'`, or `'relu'`).
- `n_iterations`: represents the number of gradient descent iterations performed by the `fit(X, y)` method.
- `learning_rate`: represents the learning rate used when updating neural network weights during gradient descent.

Between these two files, there are **nine functions** that you are required to implement for this part. The nine functions currently raise a `NotImplementedError` in order to clearly indicate which functions from the skeleton code you must implement yourself. Like before, comment out or remove the `raise NotImplementedError(...)` lines and implement each function so that they work as described in the documentation. You may assume that the input data features are all numerical features and that the target class values are categorical features. As a reminder from the guidelines, feel free to create other functions as you deem necessary, but the functions defined by the skeleton code itself must work as intended for your final solution, and therefore you cannot change the parameters for these functions. This is required because we may call any of these functions directly when testing your code. The nine functions you must implement and their descriptions are as follows:

- `identity(x, derivative = False)` in `utils.py`: computes and returns the identity activation function of the given input data `x`. If `derivative = True`, the derivative of the activation function is returned instead.
- `sigmoid(x, derivative = False)` in `utils.py`: computes and returns the sigmoid (logistic) activation function of the given input data `x`. If `derivative = True`, the derivative of the activation function is returned instead.
- `tanh(x, derivative = False)` in `utils.py`: computes and returns the hyperbolic tangent activation function of the given input data `x`. If `derivative = True`, the derivative of the activation function is returned instead.
- `relu(x, derivative = False)` in `utils.py`: computes and returns the rectified linear unit activation function of the given input data `x`. If `derivative = True`, the derivative of the activation function is returned instead.
- `cross_entropy(y, p)` in `utils.py`: computes and returns the cross-entropy loss, defined as the negative log-likelihood of a logistic model that returns `p` probabilities for its true class labels `y`.
- `one_hot_encoding(y)` in `utils.py`: converts a vector `y` of categorical target class values into a one-hot numeric array using one-hot encoding: one-hot encoding creates new binary-valued columns, each of which indicate the presence of each possible value from the original data.
- `_initialize(X, y)` in `multilayer_perceptron.py`: function called at the beginning of `fit(X, y)` that performs one-hot encoding for the target class values and initializes the neural network weights (`_h_weights`, `_h_bias`, `_o_weights`, and `_o_bias`).
- `fit(X, y)` in `multilayer_perceptron.py`: fits the model to the provided data matrix `X` and targets `y`.
- `predict(X)` in `multilayer_perceptron.py`: predicts class target values for the given test data matrix `X` using the fitted classifier model.

Testing your Implementation. As with the previous part, running the driver program `main.py` allows you to test your implementation of the `MultilayerPerceptron` class and its associated functions. Assuming you have already installed the three packages (`numpy`, `pandas`, and `scikit-learn` as discussed before), you can test your multilayer perceptron implementation by entering the command shown below on your terminal.

```
python3 main.py mlp
```

You can also test both your k-nearest neighbors implementation and your multilayer perceptron implementation one after the other by entering the following command.

```
python3 main.py all
```

You should see the following terminal output if the driver program has tested your multilayer perceptron implementation without runtime errors.

```
$ python3 main.py mlp
Loading the Iris and Digits Datasets...

Splitting the Datasets into Train and Test Sets...

Standardizing the Train and Test Datasets...

Testing Multilayer Perceptron Classification...
- Iris Dataset Progress:    [==================================================] 100%
- Digits Dataset Progress:  [==================================================] 100%

Exporting MLP Results to HTML Files...

Done Testing Multilayer Perceptron Classification!

Program Finished!  Exiting the Program...
```

Just like before, you should now see two new HTML files in your project directory, this time named `mlp_iris_results.html` and `mlp_digits_results.html`. The format of the generated tables are the same as before, but with different columns representing the parameters of the `MultilayerPerceptron` class. If you have correctly implemented the multilayer perceptron, the accuracy scores computed for your implementation and the `scikit-learn` implementation should be somewhat similar in most cases. However, unlike with k-nearest neighbors, **you may see some more substantial variation in the accuracy scores between the implementations** because the way that the model weights are initialized can make a big difference in the final prediction accuracy score of the model. Your implementation is likely just fine as long as you are seeing a decent number of cases where your implementation and the `scikit-learn` implementation accuracy scores are close to each other. If there are any concerns regarding this, feel free to make a Q&A Community post about it and the course staff will help you out.