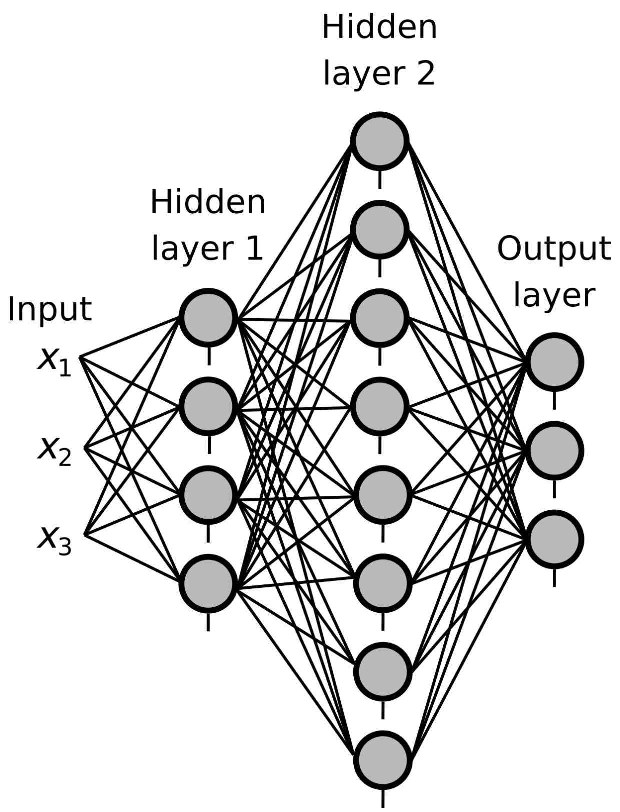
MMAI 5500 -- Assignment 1

Your task is to code and train a neural network in Python using only NumPy. That is, the only library that you are allowed to import is NumPy. It should be imported with `import numpy as np`. Luckily some parts of the code are provided (see the *Code* section below).

The network architecture

The network should have three fully connected weight layers, three inputs and three-class softmax output. The first hidden layer should have four neurons and the second eight neurons. All neurons should have a single bias. See the diagram below for a visual description.



Data

The networks should be trained and validated on the data provided in `assing1_data.csv`.

Load the data using NumPy as follows:

```
fname = 'assign1_data.csv'
data = np.genfromtxt(fname, dtype='float', delimiter=',', skip_header=1)
X, y = data[:, :-1], data[:, -1].astype(int)
X_train, y_train = X[:400], y[:400]
X_test, y_test = X[400:], y[400:]
```

Task

Use the code provided below to complete and train the network. Test it with `probs, loss = forward_pass(X_test, y_test, np.eye(n_class)[y_test])`. You should get an accuracy above 90% in less than 10 epochs of training.

Submission

Submit this assignment as a single `PY` file that loads the data, builds and trains the network and prints the accuracy on the test set. For full marks, the code has to be bug-free, PEP8 formatted (use a PEP8 plugin for your text editor) and result in a test set accuracy above 90%. The submission due date is 8:30 am on May 25, 2021.

Code

Use the code snippets below. Some snippets are incomplete and need you to add a few lines of code. The places where you need to add code are indicated by `... YOUR CODE HERE ...`.

Layer

A dense (aka fully connected) layer

```
class DenseLayer:

    def __init__(self, n_inputs, n_neurons):
        """
        Initialize weights & biases.
        Weights should be initialized with values drawn from a normal
        distribution scaled by 0.01.
        Biases are initialized to 0.0.
        """
        self.weights = 0.01 * np.random.randn(n_inputs, n_neurons)
        self.biases = "... YOUR CODE HERE ..."

    def forward(self, inputs):
        """
        A forward pass through the layer to give z.
        Compute it using np.dot(...) and then add the biases.
        """
        self.inputs = inputs
        self.z = "... YOUR CODE HERE ..."

    def backward(self, dz):
        """
        Backward pass
        """
        # Gradients of weights
        self.dweights = np.dot(self.inputs.T, dz)
        # Gradients of biases
        self.dbiases = np.sum(dz, axis=0, keepdims=True)
        # Gradients of inputs
        self.dinputs = np.dot(dz, self.weights.T)
```

Activations

ReLu

```
class ReLu:
    """
    ReLu activation
    """

    def forward(self, z):
        """
        Forward pass
        """
        self.z = z
        self.activity = "... YOUR CODE HERE ..."
```

```
    def backward(self, dactivity):
        """
        Backward pass
        """
        self.dz = dactivity.copy()
        self.dz[self.z <= 0] = 0.0
```

Softmax

```
class Softmax:

    def forward(self, z):
        """
        """
        e_z = np.exp(z - np.max(z, axis=1, keepdims=True))
        self.probs = e_z / e_z.sum(axis=1, keepdims=True)

        return self.probs

    def backward(self, dprobs):
        """
        """
        # Empty array
        self.dz = np.empty_like(dprobs)

        for i, (prob, dprob) in enumerate(zip(self.probs, dprobs)):
            # flatten to a column vector
            prob = prob.reshape(-1, 1)
            # Jacobian matrix
            jacobian = np.diagflat(prob) - np.dot(prob, prob.T)
            self.dz[i] = np.dot(jacobian, dprob)
```

Loss function

Crossentropy loss

```
class CrossEntropyLoss:

    def forward(self, probs, oh_y_true):
        """
        Use one-hot encoded y_true.
        """
        # clip to prevent division by 0
        # clip both sides to not bias up.
        probs_clipped = np.clip(probs, 1e-7, 1 - 1e-7)
        # negative log likelihoods
        loss = -np.sum(oh_y_true * np.log(probs_clipped), axis=1)

        return loss.mean(axis=0)

    def backward(self, probs, oh_y_true):
        """
        Use one-hot encoded y_true.
        """
        # Number of examples in batch and number of classes
        batch_sz, n_class = probs.shape
        # get the gradient
        self.dprobs = -oh_y_true / probs
        # normalize the gradient
        self.dprobs = self.dprobs / batch_sz
```

Optimizer

Stochastic Gradient Descent

Note that this optimizer only updates a single layer, and have thus, to be called for each layer.

```
class SGD:
    """
    """
    def __init__(self, learning_rate=1.0):
        # Initialize the optimizer with a learning rate
        self.learning_rate = learning_rate

    def update_params(self, layer):
        layer.weights = "... YOUR CODE HERE ..."
        layer.biases = "... YOUR CODE HERE ..."
```

Helper functions

Convert probabilities to predictions

```
def predictions(probs):
    """
    """
    y_preds = np.argmax(probs, axis=1)
    return y_preds
```

Accuracy

```
def accuracy(y_preds, y_true):
    """
    """
    return np.mean(y_preds == y_true)
```

One-hot encoding

```
oh_y_true = np.eye(n_class)[y_true]
```

Training

A single forward pass through the entire network.

```
def forward_pass(X, y_true, oh_y_true):
    """
    """
    dense1.forward(X)
    activation1.forward(dense1.z)
    "... YOUR CODE HERE ..."
    probs = "... YOUR CODE HERE ..."
    loss = "... YOUR CODE HERE ..."

    return probs, loss
```

A single backward pass through the entire network.

```
def backward_pass(probs, y_true, oh_y_true):
    """
```

```python
    """
    "... YOUR CODE HERE ..."
```

Initialize the network and set hyperparameters

For example, number of epochs to train, batch size, number of neurons, etc.

```python
    "... YOUR CODE HERE ..."
    dense1 = "... YOUR CODE HERE ..."
    activation1 = ReLu()
    "... YOUR CODE HERE ..."
    output_activation = "... YOUR CODE HERE ..."
    crossentropy = CrossEntropyLoss()
    optimizer = SGD()
```

Training loop

```python
for epoch in range(epochs):

    print('epoch:', epoch)

    for batch_i in range(n_batch):
        # Get a mini-batch of data from X_train and y_train. It should have batch_sz examples.
        "... YOUR CODE HERE ..."
        # One-hot encode y_true
        "... YOUR CODE HERE ..."

        # Forward pass
        "... YOUR CODE HERE ..."

        # Print accuracy and loss
        "... YOUR CODE HERE ..."

        # Backward pass
        "... YOUR CODE HERE ..."

        # Update the weights
        "... YOUR CODE HERE ..."
```