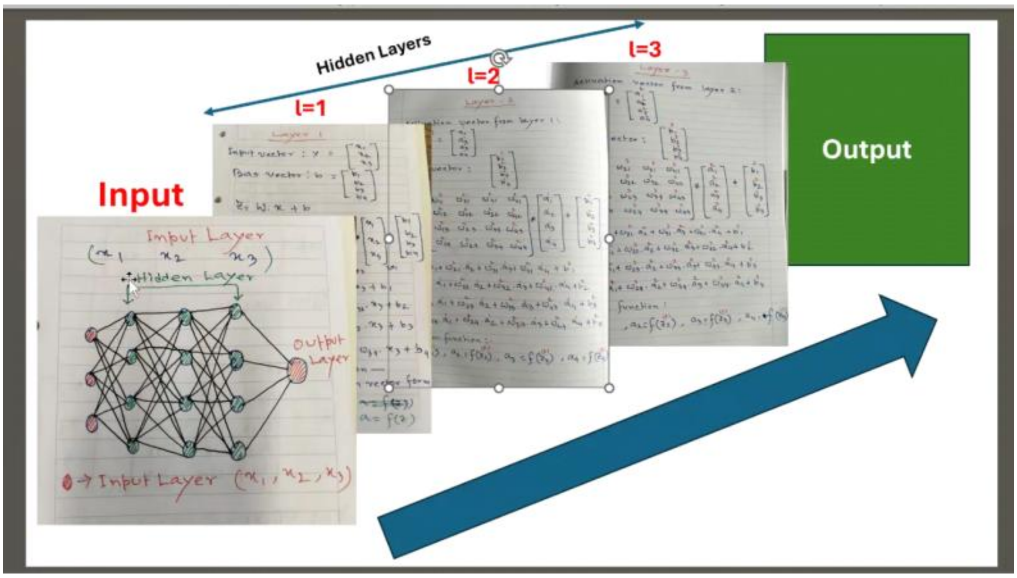
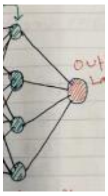
MLP Hands-on & Code Explanation Contd.

25 October 2025 09:52





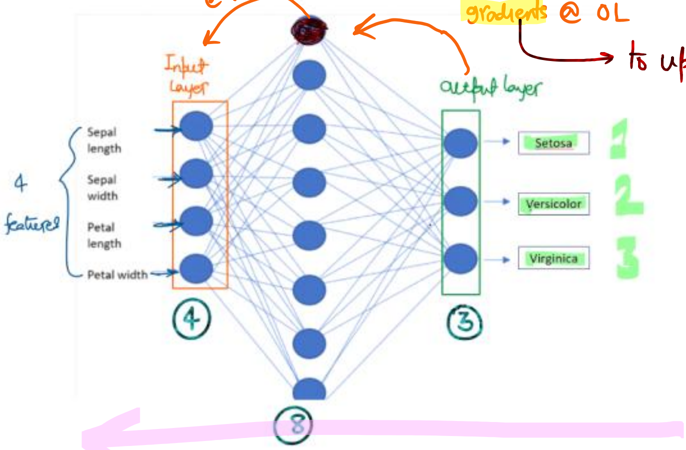
$$Z_0 = W_{11}^{(4)} a_1^{(4)} + W_{21}^{(4)} a_2^{(4)} + W_{31}^{(4)} a_3^{(4)} + W_{41}^{(4)} a_4^{(4)} + b_1^{(4)}$$

to update weights and biases between HL and IL

gradient @ HL

Hidden layer

gradients @ OL → to update weights and biases b/w OL and HL



Backward Propagation

MLP NN Model (from scratch) code flow

1. loading the IRIS dataset
   - using scikit utils
2. created class called NeuralNetwork to model the architecture

Class Name: Neural Network → encapsulate all the functions and data

hyper parameters

saving

```
class NeuralNetwork:
    def __init__(self, input_size, hidden_size, output_size, learning_rate = 0.01, epochs=100):
        self.input_size = input_size #no. of neurons or features in the input layer
        self.hidden_size = hidden_size #no. of neurons in the hidden layer (1st hidden layer)
        self.output_size = output_size #no. of neurons in the output layer
```

```python
class NeuralNetwork:
    def __init__(self, input_size, hidden_size, output_size, learning_rate = 0.01, epochs=100):
        self.input_size = input_size #no. of neurons or features in the input layer
        self.hidden_size = hidden_size #no. of neurons in the hidden layer (1st hidden layer)
        self.output_size = output_size #no. of neurons in the output layer
        self.learning_rate = learning_rate #to set the user defined learning rate for the gradient descent; default is set to 0.01
        self.epochs = epochs #no. of training epochs; default is set to 100
```

hyper parameters

saving hyperparameters as class attributes

```python
np.random.randn(4,8)
```

```
array([[ 0.02132455, -0.92791469, -1.03656354, -0.22140865,  0.09622226,
        -0.64681248,  1.25850286, -0.09090267],
       [ 2.27278845, -0.41466057, -0.46482698, -0.6166402 ,  0.97249302,
        -0.62470459,  0.48731251,  1.63385723],
       [-0.05230344,  0.66887845,  2.21904133,  0.28694164, -1.00359424,
         1.39173151, -0.54528544,  1.4929583 ],
       [ 0.32451736, -1.23305328,  1.25362685,  1.0296573 ,  0.58698445,
         0.10907541, -1.38118501,  1.14109582]])
```

4 X 8

```python
        ### INITIALIZE WEIGHTS and BIASES
```

↳ Initializing the network parameters : weights and biases.

```python
        ###################################
        # INPUT LAYER TO HIDDEN LAYER
        ###################################
```

shape = (Input size, Hidden size) : (4, 8)

IL HL

```python
        self.W1 = np.random.randn(self.input_size, self.hidden_size)*0.01 #random weights initialized from std. normal distribution
        self.b1 = np.zeros((1, self.hidden_size)) #adding a zero bias values for the neurons in the hidden layer
```

↳ initialized to zero values

↳ (initializing with small random values) from a std. normal distribution

to keep values small and stable

```python
        ###################################
        # HIDDEN LAYER TO OUTPUT LAYER
        ###################################
        self.W2 = np.random.randn(self.hidden_size, self.output_size)*0.01 #random weights initialized from std. normal distribution
        self.b2 = np.zeros((1, self.output_size)) #adding a zero bias values for the neurons in the output layer
```

$Z_1 = X \cdot W_1 + b_1$

$Z_2 = a \cdot W_2 + b_2$

(Hidden_layer_output)

"output of Z_1"

3. Creating placeholders for keeping a track of performance metrics such as
   - loss vs epoch
   - accuracy vs epoch.

```python
        ########################################
        # LOSS & ACCURACY HISTORY for PLOTTING
        ########################################
        self.loss_history = [] #empty list initialized to store the losses during triaining epochs
        self.accuracy_history = [] #empty list initialized to store the accuracy values during training epochs
```

4. Creating different activation functions to input z and get activated values as per the respective layer

```python
    #### ADD SOME ACTIVATION FUNCTIONS

    ### ReLU Activation Function
    def relu(self, z):
        return np.maximum(0,z)
```

→ Hidden layer(s)

```python
    ### Derivative of ReLU for backpropagation
    def relu_derivative(self,z):
        return np.where(z>0, 1, 0)
```

→ Hidden layer(s)

```python
    ### Given IRIS is a multi-class problem, we need to use Softmax AF

    ### Softmax Activation Funtion
    def softmax(self, z):
        exp_values = np.exp(z - np.max(z, axis=1, keepdims=True)) #subtract max for numerical stability
        return exp_values/np.sum(exp_values, axis=1, keepdims= True)
```

→ output layer.

Note: Activation functions introduce "non-linearity" into the neural network model — without them, multiple layers would just collapse into a single layer transformation.

```python
    ####################################
    #1. FORWARD PROPAGATION
    ####################################
    def forward(self, X):
```

it performs forward propagation

- it takes input as X
- feeds through the hidden layer (with ReLU as AF)
- feeds activated values through the o/p layer using Softmax as AF.

```python
        ####################################
        # INPUT LAYER TO HIDDEN LAYER
        ####################################
        self.z1 = np.dot(X, self.W1) + self.b1 #computing z1 = W1X + b1
        self.a1 = self.relu(self.z1) # plugging z1 into `ReLU` activation function to get output: a1
```

$Z_1 = X \cdot W_1 + b_1$

$a_1 = f \ldots$

```python
####################################
self.z1 = np.dot(X, self.W1) + self.b1 #computing z1 = W1X + b1
self.a1 = self.relu(self.z1) # plugging z1 into `ReLU` activation function to get output: a1
####################################
# HIDDEN LAYER TO OUTPUT LAYER
####################################
self.z2 = np.dot(self.a1, self.W2) + self.b2 #computing z2 = W2a1 + b2
self.probs = self.softmax(self.z2) #plug z2 into softmax activation function to get output as probabilities
return self.probs
```

$z_1 = X \cdot W_1 + b_1$

$a_1 = f_{relu}(z_1)$

as AF.

applying softmax activation function to convert z2 logit scores (raw scores) into probabilities

averages of -ve log-likelihood across the entire batch (120 rows)

IRIS → is a multi-class

[to compute multi-class cross entropy loss] - pls read about MCCE

```python
####################################
#2.COMPUTE LOSS & ACCURACY
####################################
### Cross-entropy loss for mult-class classification
def compute_loss(self, y_true, probs):
    loss = -np.mean(np.sum(y_true * np.log(probs), axis=1))
    return loss
```

sum across classes

one-hot encoded true/actual labels

(row-wise)

$$-\frac{1}{N}\sum_{i=1}^{N} y_i \cdot log(p(y_i)) + (1-y_i) \cdot log(1-p(y_i))$$

: Binary cross Entropy

Actuals log(odds)

compares predicted labels with true labels and computes the avg. accuracy.

```python
### Compute accuracy
def compute_accuracy(self, y_true, probs):
    predictions = np.argmax(probs, axis=1)
    true_labels = np.argmax(y_true, axis=1)
    return np.mean(predictions == true_labels)
```

for each sample, it picks the class with the highest probability as the predicted class label.

converts OHE y_true labels into class indices

returns a boolean-array indicating which predictions are correct.

converts boolean arrays to floats (True = 1 and False = 0) and then find avg. to give final accuracy.

for two random rows/samples

```python
y_true = [[0, 1, 0], [1, 0, 0]] # two samples or two rows
```

→ actual labels

versicolor setosa

```python
probs = [[0.7, 0.1, 0.2], [0.9, 0.05, 0.05]] # probabilities
```

→ predictions

setosa setosa

out of two samples → 1st one is mis-classified, → 2nd one is correctly classified $= \frac{1}{2} \times 100 = 50\%$.

BACKPROPAGATION → TASK: Understand & interpret the code block for backpropagation : due on oct 26.

```
def backward(self, X, y):

    ### Using Batch Gradient Descent (BGD)

    ### Number of rows/training examples
    m = X.shape[0] #all rows in the data

    ### Gradients of the loss w.r.t. weights and biases of the output layer
    delta3 = self.probs - y #error at the output layer
    dW2 = np.dot(self.a1.T, delta3)/m #gradient of the loss w.r.t. weights of the output layer --> dW2
    db2 = np.sum(delta3, axis=0, keepdims=True)/m #gradient of the loss w.r.t bias of the output layer --> db2

    ### Gradients of the loss w.r.t. weights and biases of the hidden layer
    delta2 = np.dot(delta3, self.W2.T)*self.relu_derivative(self.z1) #using derivative of ReLU
    dW1 = np.dot(X.T, delta2)/m #gradient of the loss w.r.t. weights of the hidden layer --> dW1
    db1 = np.sum(delta2, axis=0, keepdims=True)/m #gradient of the loss w.r.t. bias of the hidden layer --> db1


    ### Update weights & biases parameters across the layers (Hidden & Output Layer)
    self.W2 -= self.learning_rate * dW2
    self.b2 -= self.learning_rate * db2
    self.W1 -= self.learning_rate * dW1
    self.b1 -= self.learning_rate * db1
```

TASK # Add testing/validation block to the above code:

Due on: Nov 1.