Neural Network from Scratch

Problem Statement

This assignment required building an image classification algorithm using neural nets which recognises handwritten digits from 0 to 9. This algorithm has been implemented using only NumPy, Pandas and train_test_split from sklearn.

Dataset

We worked on the MNIST dataset of handwritten digits, which contained a total of 42000 images. These images were given in terms of pixel values of each pixel of 784 pixels of each image (28 * 28).
Since the dataset consisted only of numerical columns and no null values, no preprocessing was needed. Hence, we take a section of 1000 images in order to pass through the model and divide these images into training and validation sets.

Model

There was only one type of model i.e a neural network with the following layers:
- Input layer with 784 neurons
- Hidden layer with 1000 neurons
- Output layer with 10 neurons

The train set of 670 images was then trained over 200 epochs with a learning rate of 0.01. The following sections describe the different parts of the code.

Class NeuralNetworkClassifier

The object of this class is initialised using the list of layers. The constructor of this class initialises the following:
- The list of weight matrices (self.wts): This list is of the size one greater than that of the list of layers. The first value of self.wts is an empty list in order to make the indexing the forward propagation simpler to interpret. Each matrix is of the dimension no_of_neurons_in_next * (no_of_neurons_in_prev + 1). This is because we do not have a bias term and we append a 1 at the end of each layer's activation vector. Each matrix is a randomly generated matrix of values in the range [0, 1)
- The list of activations of each layer (self.a)
- The list of preactivation of each layer (self.z): self.a[l] = activation_function(self.z[l])
- Dictionary of the index to (slope, intercept) pair for linear activation layers: For each layer with linear activation, we have the dictionary to map the layer index to the pair of slope and intercept. Both the values are random numbers in [0, 1)

Note: The constructor throws an exception if layers[l][1] != layers[l + 1][0] (In accordance with the corrected format.)

fit_once(self, X, Y, alpha)

For each row (x, y) in (X, Y), we do the following:

- Run a forward_propogation(x)
- Run a backward_propogation(y)

Note that before doing the above steps we validate the dataset given i.e:

- no. of columns in X should be equal to the no. of neurons in the input layer, else an exception is thrown
- no. of classes in y should be equal to the no. of neurons in the output layer, else an exception is thrown.
- Set the learning rate self.alpha = alpha

predict(self, X)

For each row x in X, the forward propagation step is run.
Note that before doing the above step we validate the dataset given i.e:

- no. of columns in X should be equal to the no. of neurons in the input layer, else an exception is thrown

categorical_cross_entropy_loss(self, y, yhat)

(Reference: https://towardsdatascience.com/cross-entropy-loss-function-f38c4ec8643e)

forward_propogation(self, x)

We iterate through the number of layers and find the activation of each layer using the weight vector found by the network. Initially, the weight vector is randomly set. This method returns the activation of the last (output) layer.

backward_propogation(self, y)

We iterate through each layer and update the weight matrix with the gradient for each value in the matrix. Note that in most of the resources on the internet, mean squared error was used instead of categorical cross-entropy. Hence, I calculated the value of this gradient using categorical cross-entropy as the loss function using methods similar to the ones in the online resources.

$a^{(l)} \rightarrow$ Activation in the lth layer.

$w^{(l)} \longrightarrow$ weight matrix of the lth layer.

~~$b^{(l)} \longrightarrow$ Bias of the lth layer~~

At each layer l,

$$z^{(l)} = W^{(l)} a^{(l-1)} \cancel{+ b^{(l)}}$$

$$W^{(l)} \equiv K_L \times (K_{L-1} + 1)$$

$$a^{(l-1)} \equiv (K_{L-1} + 1) \times 1$$

$$a^{(l)} = (K_L + 1) \times 1$$

$$a^{(l)} = \begin{bmatrix} a_1^{(l)} \\ a_2^{(l)} \\ a_3^{(l)} \\ \vdots \\ a_{K_L}^{(l)} \\ 1 \end{bmatrix}$$

$$W^{(l)} = \begin{bmatrix} w_{11}^{(l)} & \cdots & w_{1(K_{L-1})}^{(l)} & b_1 \\ w_{21}^{(l)} & \cdots & w_{2(K_{L-1})}^{(l)} & b_2 \\ \vdots & & & \\ w_{K_L 1}^{(l)} & \cdots & w_{K_L (K_{L-1})}^{(l)} & b_{K_L} \end{bmatrix}$$

$$\vec{z}^{(l)} = W^{(l)} \vec{a}^{(l-1)}$$

$$\vec{a}^{(l)} = \sigma(z^{(l)})$$

$$C = -\sum_{i=1}^{M} y_i \log(a_i^{(L)}) = -\log a_y^{(L)}$$

M = No. of classes to be classified

$$C = y_1 \log a_1^{(L)} + \cdots + y_M \log a_M^{(L)}$$

$$\frac{\partial C}{\partial a_1^{(L)}} = \frac{y_1}{a_1^{(L)}} \qquad \boxed{\frac{\partial C}{\partial a_y^{(L)}} = \frac{-1}{a_y^{(L)}}}$$

$$\cancel{\frac{\partial C}{\partial a_i^{(L)}} = \frac{y_i}{a_i^{(L)}} \cdots \mathrm{I}}$$

$$a_y^{(L)} = \sigma(z_y^{(L)})$$

$$\frac{\partial a_y^{(L)}}{\partial z_y^{(L)}} = \sigma'(z_y^{(L)}) \quad \cdots \mathrm{II}$$

a[l][y]

$$\vec{z}^{(L)} = W^{(L)} \vec{a}^{(L-1)}$$

$$z_y^{(L)} = W_{y1}^{(L)} a_1^{(L-1)} + \cdots + W_{yk_{L-1}} a_{k_{L-1}}^{(L-1)} + W_{i\,k_{L-1}+1}$$

$$\frac{\partial z_y^{(L)}}{\partial w_{yj}^{(L)}} = a_j^{(L-1)} \qquad j \in [1, k_{L-1}]$$

$$\frac{\partial z_i^{(L)}}{\partial w^{(L)}_{iK_{L-1}+1}} = 1$$

$$\frac{\partial z_i^{(L)}}{\partial w_i} = \begin{bmatrix} a_1^{(L-1)} \\ \vdots \\ \vdots \\ a_{K_{L-1}}^{(L-1)} \\ 1 \end{bmatrix} = \vec{a}^{(L-1)} \quad \ldots \text{III}$$

$$\frac{\partial C}{\partial w_{ij}^{(L)}} = \frac{\partial C}{\partial a_i^{(L)}} \cdot \frac{\partial a_i^{(L)}}{\partial z_i^{(L)}} \cdot \frac{\partial z_i^{(L)}}{\partial w_i^{(L)}}$$

$$\frac{\partial C}{\partial w_{ij}^{(L)}} = \frac{y_i}{a_i^{(L)}} \sigma'(z_i^{(L)}) \, a_j^{(L-1)} \qquad j \in [1, K_{L-1}]$$

$$= \frac{y_i}{a_i^{(L)}} \sigma'(z_i^{(L)}) \qquad j = K_{L-1}+1$$

$$\boxed{w_{ij}^{(L)} = w_{ij}^{(L)} - \eta \frac{\partial C}{\partial w_{ij}^{(L)}}}$$

No bias, 1 @ W array with 1 at end of activat

$$\frac{\partial C}{\partial w_{ij}^{(L)}} = \frac{y_i}{a_i^{(L)}} \sigma'(z_i^{(L)}) \, a_j^{(L-1)} \qquad 1 \le j \le K_{L-1}$$

$$= \frac{y_i}{a_i^{(L)}} \sigma'(z_i^{(L)}) \qquad j = K_{L-1}+1$$

Implementation Screenshots

```
def __init__(self, layers):
    self.layers = layers
    self.L = len(layers)

    self.lin_coeffs = {}
    prev_next = 0
    for l, layer in enumerate(layers):
        prev, next, activation = layer

        if l > 0 and prev_next != prev:
            raise Exception("The no. of layers in this layer is inconsistent."
                            + "\nThe last layer mentioned no. of layers in "
                            + str(l) + "th layer is " + str(prev_next)
                            + ".\nBut this layer mentions the no. of layers to be "
                            + str(prev) + ".")

        if activation == 'linear':
            self.lin_coeffs[l] = (np.random.rand(), np.random.rand())

        prev_next = next

    self.wts = [np.random.randn(next, prev + 1) for prev, next, activation in layers[:-1]]
    self.wts.append(np.random.randn(layers[-1][1], layers[-1][0] + 1))
    self.wts.insert(0, [])

    self.act = [[] for i in range(self.L + 1)]
    self.z = [[] for i in range(self.L + 1)]
```

```
def _forward_propogation(self, x):
    x = np.append(x, self.BIAS)
    self.act[0] = x

    for l in range(1, self.L + 1):
        self.z[l] = self.wts[l] @ self.act[l - 1]
        activation = self.layers[l - 1][2]

        if activation == 'linear':
            m, c = self.lin_coeffs[l]
            self.act[l] = np.append(self.ACTIVATION[activation](self.z[l], m, c), 1)
        else:
            self.act[l] = np.append(self.ACTIVATION[activation](self.z[l]), 1)

    return self.act[self.L]
```

```
def _backward_propogation(self, y):
    for l in range(1, self.L):
        for i in range(len(self.wts[l])):
            for j in range(len(self.wts[l][0])):
                if i == y:
                    activation = self.layers[l - 1][2]

                    if activation == 'linear':
                        m = self.lin_coeffs[l][0]
                        derivative = self.DERIVATIVE[activation](m)
                    else:
                        derivative = self.DERIVATIVE[activation](self.z[l][y])

                    # print("ZERO ERROR", float(self.act[l][y]))
                    scalers = self.alpha * self.act[l - 1][j] / float(self.act[l][y] + self.CORRECTION)

                    self.wts[l][i][j] -= scalers * derivative
```

```python
def fit_once(self, X, Y, alpha):
    n_cols = X.shape[1]

    if n_cols != self.layers[0][0]:
        raise Exception("The no. of neurons in the first layer should be the same as the no. of columns in X "
                        + "\nNo. of columns in X " + str(n_cols)
                        + "\nNo. of neurons in first layer " + str(self.layers[0][0]))

    if max(Y) + 1 != self.layers[-1][1]:
        raise Exception("The no. of neurons in the last layer should be the same as the no. of classes in Y"
                        + "\nNo. of classes in Y " + str(max(Y) + 1)
                        + "\nNo. of neurons in the last layer " + str(self.layers[-1][1]))
    self.alpha = alpha

    for x, y in list(zip(X, Y)):
        self._forward_propogation(x)
        self._backward_propogation(y)
```

```python
def predict(self, x):
    n_cols = x.shape[1]

    if n_cols != self.layers[0][0]:
        raise Exception("The no. of neurons in the first layer should be the same as the no. of columns in X "
                        + "\nNo. of columns in X " + str(n_cols)
                        + "\nNo. of neurons in first layer " + str(self.layers[0][0]))

    return [self._forward_propogation(x_vec) for x_vec in x]

def categorical_cross_entropy_loss(self, y, yhat):
    if max(y) + 1 != self.layers[-1][1]:
        raise Exception("The no. of neurons in the last layer should be the same as the no. of classes in Y"
                        + "\nNo. of classes in Y " + str(max(y) + 1)
                        + "\nNo. of neurons in the last layer " + str(self.layers[-1][1]))
    error = 0
    for i, y_label in enumerate(y):
        error += -log(yhat[i][y_label] + self.CORRECTION)
    return error
```

Results

```
/opt/conda/lib/python3.7/site-packages/ipykernel_launcher.py:9: RuntimeWarning: overflow encountered in exp
  if __name__ == '__main__':
/opt/conda/lib/python3.7/site-packages/ipykernel_launcher.py:9: RuntimeWarning: invalid value encountered in true_divide
  if __name__ == '__main__':
(670,) 670
Curr loss: nan
 Test Accuracy:  0.14545454545454545
 Train Accuracy:  0.15970149253731344
List of losses: [nan]
```

I achieved an accuracy of 0.15 on my local test set owing to the high amount of time my implementation was taking to run. I faced errors like decimal overflow while using exponential, zero division errors etc. I tackled these errors by adding a very small correction of 1e-7.