Total Neural Network Code Components

```
import numpy as np

############################################ Nueron Layers
#weights: random weights of n inputs for every nueron
#bias: 0 bias for every neuron
#forward: sum of weights*inputs + bias as a dot product
class Layer_Dense:
#regularization penalizes weights for large weights and biases
#introduces a loss penalty

    def __init__(self, n_inputs, n_nuerons, w_reg_L1=0, //
                 b_reg_L1=0,w_reg_L2=0, b_reg_L2=0):
        # Inputs then weights makes it so there is no transpose needed
        self.weights = 0.1*np.random.randn(n_inputs, n_nuerons)
        self.biases = np.zeros((1, n_nuerons))
        self.w_reg_L1 = w_reg_L1
        self.w_reg_L2 = w_reg_L2
        self.b_reg_L1 = b_reg_L1
        self.b_reg_L2 = b_reg_L2
    def forward(self, inputs):
        self.inputs = inputs
        self.outputs = np.dot(inputs, self.weights) + self.biases
    def backward(self, dvalues):
        self.dweights = np.dot(self.inputs.T, dvalues)
        self.dbiases = np.sum(dvalues, axis=0, keepdims=True)

        if self.w_reg_L1 > 0:
            dL1 = np.ones_like(self.weights)
            dL1[self.weights < 0] = -1
            self.dweights += self.w_reg_L1*dL1
        if self.w_reg_L2 > 0:
            self.dweights += 2*self.w_reg_L2*self.weights
        if self.b_reg_L1 > 0:
            dL1 = np.ones_like(self.biases)
            dL1[self.biases < 0] = -1
            self.dbiases += self.b_reg_L1*dL1
        if self.b_reg_L2 > 0:
            self.dbiases += 2*self.b_reg_L2*self.biases

        self.dinputs = np.dot(dvalues, self.weights.T)

############################################### Dropout Layer
#"Turns off" neuron with a filter array that is the same shape as the layer
```

```
outputs
class Layer_Dropout:

    #Probabaility of outputs being a 1
    def __init__(self, rate):
        self.rate = 1 - rate

    def forward(self, inputs):
        self.inputs = inputs
        #Array of 0 and 1s with rate% being 1s
        self.binary_mask = np.random.binomial(1, self.rate, //
        size=inputs.shape) / self.rate
        #zeros selected random outputs
        self.outputs = inputs * self.binary_mask

    def backward(self, dvalues):
        #derivative of binomial array
        self.dinputs = dvalues * self.binary_mask

############################################ Activation function
#takes outputs and activates based on weight and bias
#forward: takes the outputs and if its negative it clips it (0) else its the
same
class Activation_ReLU:

    def forward(self, inputs):
        self.inputs = inputs
        #if value is greater than 0 outputs is input else the outputs is
        zero
        self.outputs = np.maximum(0, inputs) #Same as loop below
    def backward(self, dvalues):
        #derivative of ReLU
        self.dinputs = dvalues.copy()
        #uses copy of inputs and zeroes based on negative values
        self.dinputs[self.inputs <= 0] = 0

    def predictions(self, outputs):
        return (outputs > .5)*1

############################################ Exponentiation Activation
#Specific activation for final outputs
#forward: takes outputs and gets "probability" of each
class Activation_Softmax:

    def forward(self, inputs):
```

```
        #exponentaites values or probabilities but they are unnormalized
        #to prevent an "explosion" or overflow error subtract the max value
from
        #all values so max = 0 and other values are lower (no large numbers)
        self.inputs = inputs
        exp_values = np.exp(inputs - np.max(inputs, axis=1, keepdims=True))
        #Normalizes them         #axis adds rows or single batches, keepdim
keeps the dimensions
        probabilities = exp_values/np.sum(exp_values, axis=1, keepdims=True)
        self.outputs = probabilities

    def backward(self, dvalues):
        #derivative of softmax function
        #initializes array
        self.dinputs = np.empty_like(dvalues)
        #enumerates array
        for i, (single_output, single_dvalues) in \
                enumerate(zip(self.outputs, dvalues)):
            #flattens outputs
            single_output = single_output.reshape(-1,1)
            #calcualtes jacobian matrix
            jacob_m = np.diagflat(single_output) - \
                      np.dot(single_output, single_output.T)
            #adds to array
            self.dinputs[i] = np.dot(jacob_m, single_dvalues)

    def predictions(self, outputs):
        return np.argmax(outputs, axis=1)

############################################# Sigmoid Function

class Activation_Sigmoid:

    def forward(self, inputs):
        self.inputs = inputs
        #gets input from sigmoid function
        self.ouput = 1/(1+np.exp(-inputs))

    def backward(self, dvalues):
        #derivative of sigmoid
         dvalues*(1-self.outputs)*self.outputs

    def predictions(self, outputs):
        return (outputs > .5)*1
```

```python
############################################### Linear Activation

class Activation_Linear:

    def forward(self, inputs):
        #input in equals input out
        self.inputs = inputs
        self.outputs = inputs

    def backward(self, dvalues):
        #derivative of line
        self.dinputs = dvalues.copy()

    def predictions(self, outputs):
        return outputs

########################################### Loss Calculation (Wrapper)
#Measures the error in the outcome
#sample_losses: calculates losses from each data point
#data_loss: calculates average loss per batch
class Loss:

    def regular_loss(self, layer):
        reg_loss = 0
        if layer.w_reg_L1 > 0:
            reg_loss += layer.w_reg_L1*np.sum(np.abs(layer.weights))
        if layer.b_reg_L1 > 0:
            reg_loss += layer.b_reg_L1*np.sum(np.abs(layer.biases))
        if layer.w_reg_L2 > 0:
            reg_loss += layer.w_reg_L2*np.sum(layer.weights*layer.weights)
        if layer.b_reg_L2 > 0:
            reg_loss += layer.b_reg_L2*np.sum(layer.biases*layer.biases)
        return reg_loss

    def remember_trainable_layers(self, trainable_layers):
        self.trainable_layers = trainable_layers

    def calculate(self, outputs, y):
        sample_losses = self.forward(outputs, y)
        data_loss = np.mean(sample_losses)
        return data_loss, self.regularization_loss()

########################################### Mean Sqaured Error Loss
#calculates loss using root mean square method used for linear activation
#classification (when a value is wanted)
```

```python
class Loss_MeanSquaredError(Loss):

    def forward(self, y_pred, t_true):
        #calculates the square root of differnce squared
        sample_losses = np.mean((y_true - y_pred)**2, axis=-1)
        return sample_losses

    def backward(self, dvalues, y_true):
        #derivative of root mean squared
        samples = len(dvalues)
        outputs = len(dvalues[0])
        self.dinputs = -2*(y_true - dvalues)/outputs
        self.dinputs = self.dinputs/samples

########################################### Mean Absolute Error loss
#calculates loss using absolute mean method used for linear activation
#classification (when a value is wanted)
class Loss_MeanAbsoluteError(Loss):

    def forward(self, y_pred, t_true):
        #absolute value of true value minus estimated value
        sample_losses = np.mean(np.abs(y_true - y_pred), axis=-1)
        return sample_losses

    def backward(self, dvalues, y_true):
        samples = len(dvalues)
        outputs = len(dvalues[0])
        self.dinputs = np.sign(y_true - dvalues)/outputs
        self.dinputs = self.dinputs/samples

########################################### Binary Loss Entropy
#calculates loss for binary or two classes
class Loss_BinaryCrossentropy(Loss):

    def forward(self, y_pred, y_true):
        y_pred_clipped = np.clip(y_pred, 1e-7, 1-(1e-7))
        sample_losses = -(y_true*np.log(y_pred_clipped) +
(1-y_true)*np.log(1-y_pred_clipped))
        samples_losses = np.mean(sample_losses, axis=-1)
        return sample_losses

    def backward(self, dvalues, y_true):
        samples = len(dvalues)
        outputs = len(dvalues[0])
        clipped_dvalues = np.clip(dvalues, 1e-7, 1-(1e-7))
```

```
        self.dinputs = -(y_true/clipped_dvalues -
(1-y_true)/(1-clipped_dvalues))/outputs
        self.dinputs = self.dinputs/samples

########################################### Categorical Loss Entropy
#forward: finds loss value depending on hot encode or categorical
#if elif for accepting both category or hot encode
class Loss_CategoricalCrossentropy(Loss):
#calculates loss for categorical cross-entropy

    def forward(self, y_pred, y_true):
        samples = len(y_pred) # len of samples
        y_pred_clipped = np.clip(y_pred, 1e-7, 1-1e-7)
        # avoid divis 0 error and negative loss so clip it by a small number
        if len(y_true.shape) == 1:
            correct_confidences = y_pred_clipped[
                range(samples),
                y_true
            ]
        elif len(y_true.shape) == 2:
            correct_confidences = np.sum(
                y_pred_clipped*y_true,
                axis=1
            )
        #losses
        negative_log_likelihoods = -np.log(correct_confidences)
        return negative_log_likelihoods

    def backward(self, dvalues, y_true):
        samples = len(dvalues)
        labels = len(dvalues[0])
        if len(y_true.shape) == 1:
            y_true = np.eye(labels)[y_true]
        self.dinputs = -y_true/dvalues
        self.dinputs = self.dinputs/samples

############################################ Classifier for faster backwards
movement
#Combines loss calculations and classification
class Activation_Softmax_Loss_CrossEntropy:

    def __init__(self):
        self.activation = Activation_Softmax()
        self.loss = Loss_CategoricalCrossentropy()
```

```python
    def forward(self, inputs, y_true):
        self.activation.forward(inputs)
        self.outputs = self.activation.outputs
        return self.loss.calculate(self.outputs, y_true)

    def backward(self, dvalues, y_true):
        samples = len(dvalues)
        if len(y_true.shape) == 2:
            y_true = np.argmax(y_true, axis=1)
        self.dinputs = dvalues.copy()
        self.dinputs[range(samples), y_true] -= 1
        self.dinputs= self.dinputs/samples

############################################# Adam (Optimizer being used)
class Optimizer_Adam:
#most current and popular optimizer. Combination of several optimization
features
#includes learning weight which changes rate of change for weights and biases
#decay which gradually lowers learning rate
#momentum gives a boost in areas of rapid change and slows in areas of low.
Takes average
#rate of change (beta_1)
#cache normalizes changes in weights so no weights get too big (uses epsilon
and beta_2)

    def __init__(self, learning_rate=.001, decay=0, epsilon=1e-7,
                 beta_1=0.9, beta_2=0.999):
        self.learning_rate = learning_rate
        self.current_LR = learning_rate
        self.decay = decay
        self.iter = 0
        self.epsilon = epsilon
        self.beta_1 = beta_1
        self.beta_2 = beta_2

    def pre_update(self):
        if self.decay:
            self.current_LR = self.learning_rate*(1./(1+self.decay*self.iter))

    def update_parameters(self, layer):
        if hasattr(layer, 'weight_momentums') == False:
            layer.weight_momentums = np.zeros_like(layer.weights)
            layer.weight_cache = np.zeros_like(layer.weights)
            layer.bias_momentums = np.zeros_like(layer.biases)
            layer.bias_cache = np.zeros_like(layer.biases)
```

```
        layer.weight_momentums = self.beta_1*layer.weight_momentums + (1 - 
self.beta_1)*layer.dweights
        layer.bias_momentums = self.beta_1*layer.bias_momentums + (1 - 
self.beta_1)*layer.dbiases

        layer.weight_cache = self.beta_2*layer.weight_cache + (1 - 
self.beta_2)*layer.dweights**2
        layer.bias_cache = self.beta_2*layer.bias_cache + (1 - 
self.beta_2)*layer.dbiases**2

        weight_momentums_corrected = layer.weight_momentums / (1 - 
self.beta_1**(self.iter + 1))
        bias_momentums_corrected = layer.bias_momentums / (1 - 
self.beta_1**(self.iter + 1))
        weight_cache_corrected = layer.weight_cache / (1 - 
self.beta_2**(self.iter + 1))
        bias_cache_corrected = layer.bias_cache / (1 - self.beta_2**(self.iter 
+ 1))

        layer.weights += -self.current_LR*weight_momentums_corrected / 
(np.sqrt(weight_cache_corrected) + self.epsilon)
        layer.biases += -self.current_LR*bias_momentums_corrected / 
(np.sqrt(bias_cache_corrected) + self.epsilon)


    def post_update(self):
        self.iter += 1

############################################### AdaGrad Optmizer
#uses normalization of weights
class Optimizer_AdaGrad:

    def __init__(self, learning_rate=1, decay=0, epsilon=1e-7):
        self.learning_rate = learning_rate
        self.current_LR = learning_rate
        self.decay = decay
        self.iter = 0
        self.epsilon = epsilon

    def pre_update(self):
        if self.decay:
            self.current_LR = self.learning_rate*(1/(1+self.decay*self.iter))

    def update_parameters(self, layer):
```

```
        if hasattr(layer, 'weight_momentums') == False:
            layer.weight_cache = np.zeros_like(layer.weights)
            layer.bias_cache = np.zeros_like(layer.biases)

        layer.weight_cache += layer.dweights**2
        layer.bias_cache += layer.dbiases**2

        layer.weights +=
-self.current_LR*layer.dweights/(np.sqrt(layer.weight_cache) + self.epsilon)
        layer.biases +=
-self.current_LR*layer.dbiases/(np.sqrt(layer.bias_cache) + self.epsilon)

    def post_update(self):
        self.iter += 1

########################################## Optimizer for or SGD:
#Basic optimizer
class Optimizer_SGD:

    def __init__(self, learning_rate=1, decay=0, momentum=0):
        self.learning_rate = learning_rate
        self.current_LR = learning_rate
        self.decay = decay
        self.iter = 0
        self.momentum = momentum

    def pre_update(self):
        if self.decay:
            self.current_LR = self.learning_rate*(1/(1+self.decay*self.iter))

    def update_parameters(self, layer):
        if self.momentum:
            if hasattr(layer, 'weight_momentums') == False:
                layer.weight_momentums = np.zeros_like(layer.weights)
                layer.bias_momentums = np.zeros_like(layer.biases)

            weight_updates = self.momentum*layer.weight_momentums -
self.current_LR*layer.dweights
            layer.weight_momentums = weight_updates
            bias_updates = self.momentum*layer.bias_momentums -
self.current_LR*layer.dbiases
            layer.bias_momentums = bias_updates
        elif self.momentum == 0:
            weight_updates = -self.learning_rate*layer.dweights
            bias_updates = -self.learning_rate*layer.dbiases
```

```python
        layer.weights += weight_updates
        layer.biases += bias_updates

    def post_update(self):
        self.iter += 1




#################################################
#x = [0, 2, -1, 3.3, -2.7, 1.1, 2.2, -100]
#out = []
#for i in x:
#    if i > 0:             #for loop for activation ReLu Function
#        out.append(i)
#    elif i <= 0:
#        out.append(0)
#print(out)

############################################# Before Objects
#weights = [[0.2, 0.8, -0.5, 1],
#           [0.5, -0.91, 0.26, -0.5],
#           [-0.26, -0.27, 0.17, 0.87]]
#
#biases = [2, 3, 0.5]
#
#weights2 = [[0.1, -0.14, 0.5],
#           [-0.5, 0.12, -0.33],
#           [-0.44, 0.73, -0.13]]
#
#biases2 = [-1, 2, -0.5]
#
#layer1_outputs = np.dot(inputs, np.array(weights).T) + biases
#
#layer2_outputs = np.dot(layer1_outputs, np.array(weights2).T) + biases2
#
#print(layer2_outputs)
############################################## Layer Calculation
#neuron_layer = []
#for i in range(len(weights)):
#    n_output = 0
#    for j in range(len(weights[i])):
#        n_output += weights[i][j]*inputs[j]
#    neuron_layer.append(n_output + biases[i])
```

```
#print(neuron_layer)

############################################# Loss Calculation
# softmax_output = [.7, .1, .2]
# target_output = [1,0,0]
# loss = -(np.log(softmax_output[0])*target_output[0]+
#          np.log(softmax_output[1])*target_output[1]+
#          np.log(softmax_output[2])*target_output[2])
# # goes to:
# loss_s = -np.log(softmax_output[0])*target_output[0]
```