Architecture

```
model = Architecture()
```

Arguments
This function doesn't take any arguments

Add Input Layer

```
add_inputLayer (
        Input_units , normalization = False,
        Weight_initialization= "randomize",random_initializer = 0.01
)
```

Arguments	Description
Input_units	Number of neurons required by the input data to pass the data to the other layers of the network
normalization	Optional True of False. If the input data has to be normalized then set this value to True

	Default value is False
weight_initialization	Optional default set to randomize.
random_initializer	If randomization weight initialization is used then this is the variance of weights.

Add Hidden Layer

```
add_hiddenLayer (
        hidden_units, activation_function = "sigmoid",
        dropout = 1, regularizer = None, regularizer_lambda=0.1
)
```

Arguments	Description
hidden_units	Number of neurons for this layer
activation_function	Optional default is sigmoid. Activation equation of the particular layer.
dropout	Optional default is set to 1 for no dropout. Used for dropping a few weights of this particular layer.
regularizer	Optional default set to None. Used for regularizing the particular layer.
regularizer_lambda	Optional default set to 0.1. This parameter can be used for all types of regularization.

Compile

```
compile (
          cost_function = "crossEntropy", optimizer = 'gradient',
           learning_rate = 0.01, momentum_beta = 0.9,
           rms_beta = 0.999, adam_beta1=0.9,
           adam_beta2=0.999,batch_normalization = False
)
```

Arguments	Description
cost_function	Optional default set to CrossEntropy. Used for predicting the cost after training the model.
optimizer	Optional default set to gradient. Used as an algorithm for enhancing performance of updating parameters.
learning_rate	Optional default is set to 0.01 . Rate at which the weight and biases should be updated during back propagation.
momentum_beta	Optional default set to 0.9 . Used when the optimizer is momentum.

rms_beta	Optional default set to 0.999 . Used when the optimizer is rmsProp.
adam_beta1	Optional default set to 0.9. Used when the optimizer is adam.
adam_beta2	Optional default set to 0.999. Used when the optimizer is adam.
batch_normalization	Optional default set to False. If the output activation of each layer has to be normalized then this is used.

Train

```
train (
        X, Y, number_of_epoch = 1000,
        batch_size = 0, print_every = 100
)
```

Arguments	Description
X	Training dataset for the neural network
Y	Labels of the given training dataset.
number_of_epoch	Optional default is set to 1000. Number of iterations on the training dataset
batch_size	Optional default set to 0. To divide the training dataset into mini batches based upon the size of every batch.

print_every	Optional default set to 100. Used for printing the loss of the model after every print_every iteration.

Example

```
model = Architecture()
model.add_inputLayer(X.shape[0],normalization = False,weight_initialization="xavier")
model.add_hiddenLayer(4,"tanh", dropout = 0.8)
model.add_hiddenLayer(4,"relu", regularizer = “L1” , regularizer_lambda = 0.01)
model.add_hiddenLayer(Y.shape[0],"sigmoid")
model.compile(cost_function = "meanSquare", optimizer = 'gradient', learning_rate = 1.2)
model.train(X,Y,number_of_epoch = 10000)
```

Forward Propagation

This method is called when model.train() is called, during this phase the training data is passed through the hidden layer where it's linear calculation Z [L] = W[L] * A[L-1] + b[L] , where L indicates the current layer. Activation function calculation A [L] = Activation_function (Z[L]).

```
forward_propagation (
        X, total_layers, parameters,
        activations, dropout_layers
)
```

Arguments	Description
X	Training dataset for the neural network
total_layers	Number of layers of the neural network
parameters	Weights and biases of each layer stored in a dictionary.
activations	Activation functions stored as a dictionary for each layer.
dropout_layers	Dropout value of each layer stored in a dictionary.

Cost Function

This method is invoked during model.train(). This function calculates the loss or the error of the neural network after going through the whole neural network. The loss is calculated based upon the loss functions specified.

```
compute_cost (
        Y_pred, Y, cost_function,
        parameters, regularizer_layers
)
```

Arguments	Description
Y_Pred	Output produced by the neural network , the activation of the last layer.
Y	The expected outcome of the training dataset.
cost_function	Computes the cost of the network based upon the function specified.
parameters	Weights and biases of each layer stored in a dictionary.

regularizer_layers	Regularizations and its lambda values of each layer stored in a dictionary.

Backward Propagation

This function is invoked during method.train(). During this phase of the neural network, the gradient of weight and biases are calculated according to the loss for improving the neural network to train upon the given training dataset.

dZ [L] = Activation_function_derivative(dA[L])

dW [L] = (1/m) * dZ[L] * A[L-1]

dA [L] = (1/m) * ☐ dZ[L]

dA [L-1] = W[L] * dZ[L]

```
backward_propagation(
        Y, cost_function, parameters, cache, activations,
        total_layers, regularizer_layers, dropout_layers
)
```

Arguments	Description
Y	The expected outcome of the training dataset..
cost_function	Computes the derivative of the cost function for dA of the last layer.
parameters	Weights and biases of each layer stored in a dictionary.
cache	A and Z of each layer stored in a dictionary.

activations	Activation of each layer stored in a dictionary.
total_layers	Total number of layers on the network.
regularizer_layers	Regularizations and its lambda values of each layer stored in a dictionary.
dropout_layers	Dropout of each layer stored in a dictionary.

Update Parameters

This method is invoked during model.train(). This function calculates the new weights and biases for the neural network based upon the gradients calculated during the back propagation.

W [updated] = W[current] - learning_rate*dW[current]

B [updated] = b[current] - learning_rate*dW[current]

```
update_parameters(
        parameters, v,s , t,
        grads , learning_rate, total_layers, optimizer,
        momentum_beta,rms_beta,adam_beta1,adam_beta2
)
```

Arguments	Description
parameters	Weights and biases of each layer stored in a dictionary.

v	Weight average decay when using momentum.
s	Weight average decay when using rmsProp.
t	Parameter required by adam optimizer.
grads	dW and db of each layer stored in a dictionary.
learning_rate	Rate at which the model has to update the parameters.
total_layers	Total number of layers of the network.
optimizer	Optimization function used for the network.
momentum_beta	Parameter required by momentum optimizer.
rms_beta	Parameter required by the rmsProp optimizer.
adam_beta1	Parameter required by the adam optimizer.
adam_beta2	Parameter required by the adam optimizer.

Activation Functions

Activation functions are the output of a particular layer ad these values decide whether its fired or not for the current training data example .

sigmoid(Z)

tanh(Z)

relu(Z)

```
softmax(Z)

relu_backward(dA, cache)

sigmoid_backward(dA, cache):

tanh_backward(dA, cache)

softmax_backward(dA,cache)
```

Arguments	Description
Z	Linear computation of the current layer.
dA	Gradient of activation computation of the given layer.
cache	Linear computation of the current layer. Just an alias name used but similar to Z.

Weight Initializations

Initializing the weights of the network based upon different activation functions mostly used by your network can greatly impact the final accuracy if the weights initialized are also taken into consideration instead of being completely random.

```
weight_initialize_randomize(curr_layer,prev_layer,random_initializer)
```

```
weight_initialize_xavier(curr_layer,prev_layer)
```

```
weight_initialize_he(curr_layer,prev_layer)
```

Arguments	Description
curr_layer	Number of neurons of the current layer.
prev_layer	Number of neurons of the previous layer
random_initializer	When using randomize initialization this is the variance of the weight array

Regularizations

Regularizations are used when the neural network overfits the training data, to reduce this different types of regularizations are used which impact the parameters during their update.

```
cost_L1(W,reg_lambda)

cost_L2(W,reg_lambda)

compute_regularizer_cost(m,parameters,regularizer_layers)

backward_propagation_L1(W,m,reg_lambda)
```

```
backward_propagation_L2(W,m,reg_lambda)
```

```
backward_propagation_regularizer(m,parameters,regularizer_layers)
```

Arguments	Description
W	Weight of the current layer in the network.
reg_lambda	Lambda value used by the regularization function of the current layer.
parameters	Weights and biases of all layers of the neural network.
regularizer_layers	Regularizer function and its lambda parameter stored in a dictionary.
m	Total number of training examples present in the input data.

DropOut

Dropout is used when the neural network is overfitting the data, and to overcome dropout randomly cuts off a few neurons of a particular layer specified for a particular iteration such that the neuron output won't affect while training for the particular iteration and aren't updated as well.

```
inverted_dropout_cache(A,dropout)

inverted_dropout_grads(dA,D,dropout)
```

Arguments	Description
A	Activation computation of the current layer.
dA	Gradient of the activation computation of the current layer.
D	Dropout mask matrix of the current layer used for shutting down few neurons.
dropout	Dropout value of the current layer, used for inverted dropout method.

Optimizers

Optimizers are algorithms or methods used to change the attributes of your neural network such as weights and learning rate in order to reduce the losses.

```
initialize_momentum(parameters,total_layers)

initialize_rmsProp(parameters,total_layers)
```

```
initialize_adam(parameters,total_layers)
```

```
gradient(parameters,total_layers,learning_rate,grads)

momentum(parameters,v,total_layers,learning_rate,grads,beta)

rmsProp(parameters,s,total_layers,learning_rate,grads,beta)

adam(parameters,v,s,t,total_layers,learning_rate,grads,beta1,beta2)
```

Arguments	Description
parameters	Weights and biases of all layers
total_layers	Total number of layers
grads	Gradients of weight and biases.
v	Weight average decay of momentum optimizer
s	Weighted average decay of rmsProp.
beta1,beta2,t	Parameters required by the adam optimizer.

Cost Functions

A cost function is a measure of "how good" a neural network did with respect to it's given training sample and the expected output.

```
crossEntropy_cost(Y_pred,Y,m)
```

```
meanSquare_cost(Y_pred,Y,m)
```

```
crossEntropy_derivative(Y_pred,Y)

meanSquare_derivative(Y_pred,Y)

categorical_crossEntropy_cost(Y_pred,Y,m)

categorical_crossEntropy_derivative(Y_pred,Y)
```

Arguments	Description
Y_pred	Prediction made by the neural network for the given training dataset.
Y	Expected output of the given training dataset.
m	Total Number of training examples

Generate Mini Batches

This function takes the training dataset and divides them into smaller batches known as mini batches. Mini batches are proved to have a significant increase in the accuracy for few models and are used at times when the training dataset is too big to pass in one iteration.

```
generate_mini_batches(
        X, Y, batch_size = 64
)
```

Arguments	Description
X	Training dataset
Y	Expected output of the given training dataset.
batch_size	Size of each batch used for dividing up the training dataset accordingly.