Artificial Neural Network

This is a repository for the code used in my project to implement an artificial neural network in Python. The code is heavily based on the Keras source code.

The following is a summary of the code in this repository.

layers

class Reshape(Layer):

def **init**(self, input_shape, output_shape):

super(Reshape, self).**init**()

self.input_shape = input_shape

self.output_shape = output_shape

def forward(self, x):

"""

Input:

x: numpy array

Output:

reshaped_x: numpy array

"""

reshaped_x = x.reshape(self.output_shape)

return reshaped_x

def backward(self, output_gradient):

"""

Input:

output_gradient: numpy array

Output:

reshaped_output_gradient: numpy array

"""

reshaped_output_gradient = output_gradient.reshape(self.input_shape) return reshaped_output_gradient

def softmax(self, x):

""""

Input:

x: numpy array

Output:

softmax_x: numpy array

""""

softmax_x = np.exp(x - self.biases) / np.sum(np.exp(x - self.biases), axis=0)

return softmax_x

def sigmoid(self, x):

""""

Input:

x: numpy array

Output:

sigmoid_x: numpy array

""""

sigmoid_x = 1 / (1 + np.exp(-x))

return sigmoid_x

The activation functions are defined in the file `activations.py`. They include the ReLU, sigmoid, tanh, softmax, arctanh, and hyperbolic tangent activation functions. The implementation uses the following functions from the Keras backend for computing the activations:

```
T.nnet.sigmoid(x)
T.nnet.hard_sigmoid(x)
T.nnet.linear(x)
T.tanh(x)
T.sin(x)
T.cos(x)
T.exp(x)
T.log(x)
T.sqrt(x)
T.nnet.relu(x, alpha=0.0, max_value=None)
```

```
class ReLU(Layer):

def __init__(self, alpha=0.0, max_value=None):

super(ReLU, self).__init__()

self.alpha = alpha

self.max_value = max_value

def forward(self, x):

"""

Input:

x: numpy array

Output:

relu_x: numpy array of relu

"""

relu_x = T.nnet.relu(x, alpha=self.alpha, max_value=self.max_value)

return relu_x

def backward(self, output_gradient, relu_x):

"""

Input:

output_gradient: numpy array, gradient of the output

relu_x: numpy array

Output:

grad: numpy array of the gradient of the loss function with respect to the
output

"""

# Compute the gradient of the loss function with respect to the inputs

grad = output_gradient * self.grad(relu_x)

return grad
```

```
class Sigmoid(Layer):

def __init__(self, beta=1.0):

super(Sigmoid, self).__init__()

self.beta = beta

def forward(self, x):

"""

Input:

x: numpy array

Output:

sigmoid_x: numpy array

"""

sigmoid_x = T.nnet.sigmoid(x * self.beta)

return sigmoid_x

def backward(self, output_gradient):

"""

Input:

output_gradient: numpy array, gradient of the output

Output:

grad: numpy array of the gradient of the loss function with respect to the
output

"""

# Compute the gradient of the loss function with respect to the inputs

grad = output_gradient * self.grad(self.sigmoid_x)

return grad

class Tanh(Layer):

def __init__(self, beta=1.0):

super(Tanh, self).__init__()
```

```
self.beta = beta

def forward(self, x):

"""

Input:

x: numpy array

Output:

tanh_x: numpy array

"""

tanh_x = T.tanh(x * self.beta)

return tanh_x

def backward(self, output_gradient):

"""

Input:

output_gradient: numpy array, gradient of the output

Output:

grad: numpy array of the gradient of the loss function with respect to the
output

"""

# Compute the gradient of the loss function with respect to the inputs

grad = output_gradient * self.grad(self.tanh_x)

return grad
```

```
class Softmax(Layer):

def __init__(self, axis=-1):

super(Softmax, self).__init__()

self.axis = axis

def forward(self, x):
```

```
"""

Input:

x: numpy array

Output:

softmax_x: numpy array

"""

softmax_x = T.nnet.softmax(x, axis=-1)

return softmax_x

def backward(self, output_gradient):

"""

Input:

output_gradient: numpy array, gradient of the output

Output:

grad: numpy array of the gradient of the loss function with respect to the
output

"""

# Compute the gradient of the loss function with respect to the inputs

## layers

class Reshape(Layer):

def __init__(self, input_shape, output_shape):

super(Reshape, self).__init__()

self.input_shape = input_shape

self.output_shape = output_shape

def forward(self, x):

"""

Input:

x: numpy array

Output:
```

```
reshaped_x: numpy array

"""

reshaped_x = x.reshape(self.output_shape)

return reshaped_x

def backward(self, output_gradient):

"""

Input:

output_gradient: numpy array

Output:

reshaped_output_gradient: numpy array

"""

reshaped_output_gradient = output_gradient.reshape(self.input_shape)

return reshaped_output_gradient

def softmax(self, x):

"""

Input:

x: numpy array

Output:

softmax_x: numpy array

"""

softmax_x = np.exp(x - self.biases) / np.sum(np.exp(x - self.biases),
axis=0)

return softmax_x

def sigmoid(self, x):

"""

Input:

x: numpy array
```

```
Output:

sigmoid_x: numpy array

"""

sigmoid_x = 1 / (1 + np.exp(-x))

return sigmoid_x

class Dense(Layer):

def __init__(self, input_size, num_hidden_neurons, bias=0.0):

super(Dense, self).__init__()

self.num_hidden_neurons = num_hidden_neurons

# Initialize the W matrix

self.W = np.random.normal(0, math.sqrt(2.0 / self.num_hidden_neurons),
size=(num_hidden_neurons, input_size))

# Initialize the b vector as zero

self.b = np.zeros((self.num_hidden_neurons, 1))

self.bias = bias

def forward(self, x):

"""

Input:

x: numpy array of the
```