In [1]:

```
from keras.models import load_model
from keras.datasets import cifar10
from keras.utils import to_categorical
(trainX,trainY),(testX,testY) = cifar10.load_data()
trainY3 = ((trainY == 0)+(trainY ==1) +(trainY==2))
trainY3_class = trainY[trainY3]
trainX_class = trainX[trainY3[:,0],:,:,:]
testY3 = ((testY == 0)+(testY ==1) +(testY==2))
testY3_class = testY[testY3]
testX_class = testX[testY3[:,0],:,:,:]
```

```
Using TensorFlow backend.
```

In [2]:

```
print(len(trainX))
print(len(trainY))
print(len(testX))
print(len(testY))
print(len(trainY3_class))
print(len(trainX_class))
```

```
50000
50000
10000
10000
15000
15000
```

In [3]:
```
import numpy as np
import numpy.matlib

class nn_Sigmoid:
    def forward(self, x):
        return 1 / (1 + np.exp(-x))

class nn_Linear:
    def __init__(self, input_dim, output_dim):
        # Initialized with random numbers from a gaussian N(0, 0.001)
        self.weight = np.matlib.randn(input_dim, output_dim) * 0.001
        self.bias = np.matlib.randn((1, output_dim)) * 0.001

    # y = Wx + b
    def forward(self, x):
        return np.dot(x, self.weight) + self.bias

    def getParameters(self):
        return [self.weight, self.bias]

trainX = trainX_class.reshape(15000,32*3*32)
a1 = nn_Sigmoid().forward(nn_Linear(3072,3).forward(trainX))

# # Let's test the composition of the two functions (forward-propagation in th
e neural network).
# x1 = np.array([[1, 2, 2, 3]])
# a1 = nn_Sigmoid().forward(nn_Linear(4, 3).forward(x1))
# print('x[1] = '+ str(x1))
# print('a[1] = ' + str(a1))

# # Let's test the composition of the two functions (forward-propagation in th
e neural network).
# x2 = np.array([[4, 5, 2, 1]])
# a2 = nn_Sigmoid().forward(nn_Linear(4, 3).forward(x2))
# print('x[2] = '+ str(x2))
# print('a[2] = ' + str(a2))

# # We can also compute both at once, which could be more efficient since it r
equires a single matrix multiplication.
# x = np.concatenate((x1, x2), axis = 0)
# a = nn_Sigmoid().forward(nn_Linear(4, 3).forward(x))
# print('x = ' + str(x))
# print('a = ' + str(a))
```

In [4]:
```
class Cross_Category:  # MSE = mean squared error.
    def forward(self, predictions, labels):
        return np.sum(np.square(predictions - labels))
```

In [5]:
```
# This is referred above as f(u).
class Cross_Category:
    def forward(self, predictions, labels):
        return np.sum(np.square(predictions - labels))

    def backward(self, predictions, labels):
        num_samples = labels.shape[0]
        return num_samples * 2 * (predictions - labels)

# This is referred above as g(v).
class nn_Sigmoid:
    def forward(self, x):
        return 1 / (1 + np.exp(-x))

    def backward(self, x, gradOutput):
        # It is usually a good idea to use gv from the forward pass and not re
compute it again here.
        gv = 1 / (1 + np.exp(-x))
        return np.multiply(np.multiply(gv, (1 - gv)), gradOutput)

# This is referred above as h(W, b)
class nn_Linear:
    def __init__(self, input_dim, output_dim):
        # Initialized with random numbers from a gaussian N(0, 0.001)
        self.weight = np.matlib.randn(input_dim, output_dim) * 0.01
        self.bias = np.matlib.randn((1, output_dim)) * 0.01
        self.gradWeight = np.zeros_like(self.weight)
        self.gradBias = np.zeros_like(self.bias)

    def forward(self, x):
        return np.dot(x, self.weight) + self.bias

    def backward(self, x, gradOutput):
        # dL/dw = dh/dw * dL/dv
        self.gradWeight = np.dot(x.T, gradOutput)
        # dL/db = dh/db * dL/dv
        self.gradBias = np.copy(gradOutput)
        # return dL/dx = dh/dx * dL/dv
        return np.dot(gradOutput, self.weight.T)

    def getParameters(self):
        params = [self.weight, self.bias]
        gradParams = [self.gradWeight, self.gradBias]
        return params, gradParams

trainX = trainX_class.reshape(15000,32*3*32)
a1 = nn_Sigmoid().forward(nn_Linear(3072,3).forward(trainX))
trainY = to_categorical(trainY3_class)

# # Let's test some dummy inputs for a full pass of forward and backward propa
gation.
# x1 = np.array([[1, 2, 2, 3]])
# y1 = np.array([[0.25, 0.25, 0.25]])

# Define the operations.
linear = nn_Linear(3072, 3)  # h(W, b)
```

```
sigmoid = nn_Sigmoid()  # g(v)
loss = nn_MSECriterion()  # f(u)

# Forward-propagation.
a0 = linear.forward(trainX)
a1 = sigmoid.forward(a0)
loss_val = loss.forward(a1, trainY) # Loss function.

# Backward-propagation.
da1 = loss.backward(a1, trainY)
da0 = sigmoid.backward(a0, da1)
dx1 = linear.backward(trainX, da0)

# Show parameters of the linear layer.
print('\nW = ' + str(linear.weight))
print('B = ' + str(linear.bias))

# Show the intermediate outputs in the forward pass.
print('\nx1 = '+ str(trainX))
print('a0 = ' + str(a0))
print('a1 = ' + str(a1))

print('\nloss = ' + str(loss_val))

# Show the intermediate gradients with respect to inputs in the backward pass.
print('\nda1 = ' + str(da1))
print('da0 = ' + str(da0))
print('dx1 = ' + str(dx1))

# Show the gradients with respect to parameters.
print('\ndW = ' + str(linear.gradWeight))
print('dB = ' + str(linear.gradBias))
```

```
W = [[-0.01314412 -0.00094724 -0.00427989]
 [ 0.00305823  0.01117694  0.00056257]
 [ 0.00102789  0.00480435  0.00208372]
 ...
 [ 0.00763975 -0.00621717 -0.00254085]
 [-0.00964656 -0.01512588  0.00163176]
 [-0.01553434  0.00151929 -0.00118366]]
B = [[ 0.00319039  0.00387922 -0.00148443]]

x1 = [[170 180 198 ...  73  77  80]
 [159 102 101 ... 182  57  19]
 [164 206  84 ... 122 170  44]
 ...
 [145 161 194 ...  37  39  54]
 [189 211 240 ... 195 190 171]
 [229 229 239 ... 163 163 161]]
a0 = [[ -16.16137337   10.66272905  -28.33179919]
 [ -57.08310076   22.15582951 -102.94878474]
 [ -74.28491613  -67.49568553  -43.58044166]
 ...
 [ -59.28341437  -17.07522754 -149.44208947]
 [-124.4865879   -44.92349987  -95.56421585]
 [ -39.87055799   11.18301258  -51.50317583]]
a1 = [[9.57645309e-08 9.99976599e-01 4.96199071e-13]
 [1.61854333e-25 1.00000000e+00 1.94944447e-45]
 [5.47609327e-33 4.86402974e-30 1.18373549e-19]
 ...
 [1.79283478e-26 3.83992625e-08 1.25350243e-65]
 [8.63300130e-55 3.09009549e-20 3.14042474e-42]
 [4.83544752e-18 9.99986092e-01 4.28997674e-23]]

loss = 15411.554419467193

da1 = [[ 2.87293593e-03 -7.02015505e-01  1.48859721e-08]
 [ 4.85562998e-21 -7.16088522e-06  5.84833342e-41]
 [ 1.64282798e-28  1.45920892e-25 -3.00000000e+04]
 ...
 [ 5.37850435e-22  1.15197788e-03 -3.00000000e+04]
 [ 2.58990039e-50 -3.00000000e+04  9.42127423e-38]
 [ 1.45063426e-13 -4.17248286e-01  1.28699302e-18]]
da0 = [[ 2.75125335e-010 -1.64271412e-005  7.38640556e-021]
 [ 7.85904750e-046 -1.70927590e-015  1.14010013e-085]
 [ 8.99627926e-061  7.09763560e-055 -3.55120646e-015]
 ...
 [ 9.64276968e-048  4.42350992e-011 -3.76050729e-061]
 [ 2.23586134e-104 -9.27028648e-016  2.95868027e-079]
 [ 7.01446582e-031 -5.80312368e-006  5.52117012e-041]]
dx1 = [[ 1.55567898e-08 -1.83604263e-07 -7.89214555e-08 ...  1.02132392e-07
   2.48472361e-07 -2.49618012e-08]
 [ 1.61909043e-18 -1.91044672e-17 -8.21195992e-18 ...  1.06268547e-17
   2.58543071e-17 -2.59687911e-18]
 [ 1.51987789e-17 -1.99779952e-18 -7.39971394e-18 ...  9.02308149e-18
  -5.79470841e-18  4.20343155e-18]
 ...
 [-4.19011500e-14  4.94412868e-13  2.12520905e-13 ... -2.75017025e-13
  -6.69094931e-13  6.72057712e-14]
 [ 8.78116407e-19 -1.03613398e-17 -4.45377021e-18 ...  5.76349245e-18
```

```
  1.40221268e-17 -1.40842174e-18]
 [ 5.49693704e-09 -6.48611413e-08 -2.78802380e-08 ...  3.60789924e-08
   8.77773692e-08 -8.81660512e-09]]

dW = [[1.24627119e+07 8.38445975e+07 6.83158060e+06]
 [1.07849444e+07 9.83281317e+07 7.98103903e+06]
 [7.39721086e+06 1.08302439e+08 8.99178028e+06]
 ...
 [2.54745240e+07 8.14794463e+07 9.18056290e+06]
 [2.38413398e+07 8.25715914e+07 9.65992575e+06]
 [2.07932340e+07 7.62229035e+07 1.01660618e+07]]
dB = [[ 2.75125335e-010 -1.64271412e-005  7.38640556e-021]
 [ 7.85904750e-046 -1.70927590e-015  1.14010013e-085]
 [ 8.99627926e-061  7.09763560e-055 -3.55120646e-015]
 ...
 [ 9.64276968e-048  4.42350992e-011 -3.76050729e-061]
 [ 2.23586134e-104 -9.27028648e-016  2.95868027e-079]
 [ 7.01446582e-031 -5.80312368e-006  5.52117012e-041]]
```

In [ ]:
```
# # We will compute derivatives with respect to a single data pair (x,y)
# x = np.array([[2.34, 3.8, 34.44, 5.33]])
# y = np.array([[3.2, 4.2, 5.3]])

# Define the operations.
linear = nn_Linear(3072, 3)
sigmoid = nn_Sigmoid()
criterion = Cross_Category()

# Forward-propagation.
a0 = linear.forward(trainX)
a1 = sigmoid.forward(a0)
loss = criterion.forward(a1, trainY) # Loss function.

# Backward-propagation.
da1 = criterion.backward(a1, trainY)
da0 = sigmoid.backward(a0, da1)
dx = linear.backward(trainX, da0)

gradWeight = linear.gradWeight
gradBias = linear.gradBias

approxGradWeight = np.zeros_like(linear.weight)
approxGradBias = np.zeros_like(linear.bias)

# We will verify here that gradWeights are correct and leave it as an excercis
e
# to verify the gradBias.
epsilon = 0.0001
for i in range(0, linear.weight.shape[0]):
    for j in range(0, linear.weight.shape[1]):
        # Compute f(w)
        fw = criterion.forward(sigmoid.forward(linear.forward(trainX)), trainY
) # Loss function.
        # Compute f(w + eps)
        shifted_weight = np.copy(linear.weight)
        shifted_weight[i, j] = shifted_weight[i, j] + epsilon
        shifted_linear = nn_Linear(3072, 3)
        shifted_linear.bias = linear.bias
        shifted_linear.weight = shifted_weight
        fw_epsilon = criterion.forward(sigmoid.forward(shifted_linear.forward(
trainX)), trainY) # Loss function
        # Compute (f(w + eps) - f(w)) / eps
        approxGradWeight[i, j] = (fw_epsilon - fw) / epsilon

# These two outputs should be similar up to some precision.
print('gradWeight: ' + str(gradWeight))
print('\napproxGradWeight: ' + str(approxGradWeight))
```

In [ ]:
```
dataset_size = 1000

# Generate random inputs within some range.
x = np.random.uniform(0, 6, (dataset_size, 4))
# Generate outputs based on the inputs using some function.
y1 = np.sin(x.sum(axis = 1))
y2 = np.sin(x[:, 1] * 6)
y3 = np.sin(x[:, 1] + x[:, 3])
y = np.array([y1, y2, y3]).T

print(x.shape)
print(y.shape)
```

In [ ]:
```
learningRate = 0.1

model = {}
model['linear'] = nn_Linear(3072, 3)
model['linear2'] = nn_Linear(3, 3)
model['sigmoid'] = nn_Sigmoid()
model['loss'] = Cross_Category()

for epoch in range(0, 100):
    loss = 0
    for i in range(0, dataset_size):
        xi = trainX[i:i+1, :]
        yi = trainY[i:i+1, :]

        # Forward.
        a0 = model['linear'].forward(xi)
        a1 = model['sigmoid'].forward(a0)
        a2 = model['linear2'].forward(a1)
        a3 = model['sigmoid'].forward(a2)
        a4 = model['linear2'].forward(a3)
        a5 = model['sigmoid'].forward(a4)
        loss += model['loss'].forward(a5, yi)

        # Backward.
        da1 = model['loss'].backward(a1, yi)
        da0 = model['sigmoid'].backward(a0, da1)
        model['linear'].backward(xi, da0)

        model['linear'].weight = model['linear'].weight - learningRate * model
['linear'].gradWeight
        model['linear'].bias = model['linear'].bias - learningRate * model['li
near'].gradBias

    if epoch % 10 == 0: print('epoch[%d] = %.8f' % (epoch, loss / dataset_size
))
```