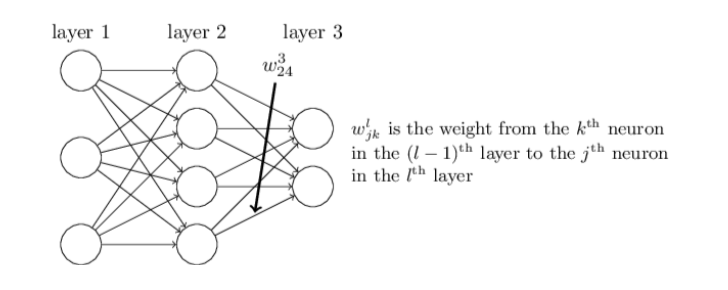
**Backpropagation Algorithm**

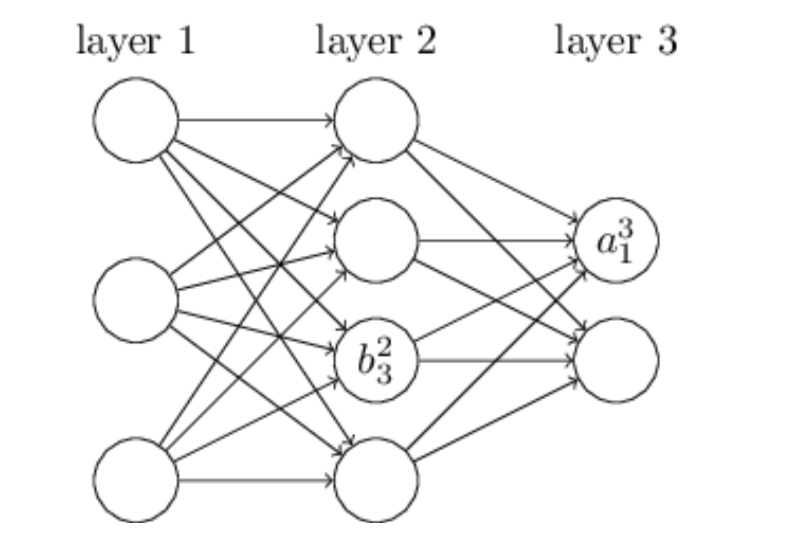
Backpropagation, an abbreviation for "backward propagation of errors", is a common method of training artificial neural networks used in conjunction with an optimization method such as gradient descent. The method calculates the gradient of a loss function with respect to all the weights in the network. The gradient is fed to the optimization method which in turn uses it to update the weights, in an attempt to minimize the loss function.

**2.1. A fast matrix-based approach to computing the output from a neural network**

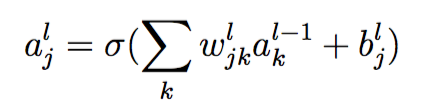


We’ll use Wjkl to denote the weight for the connection from the kth neuron in the (l−1)th layer to the jth neuron in the lth layer. So, for example, the diagram above shows the weight on a connection from the fourth neuron in the second layer to the second neuron in the third layer of a network.

We use a similar notation for the network’s biases and activations. Explicitly, we use bjl for the bias of the jth neuron in the lth layer. And we use ajl for the activation of the jth neuron in the lth layer. The following diagram shows examples of these notations in use:

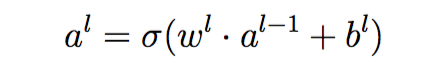


The activation ajj of the jth neuron in the lth layer is related to the activations in the (l − 1)th layer by the equation

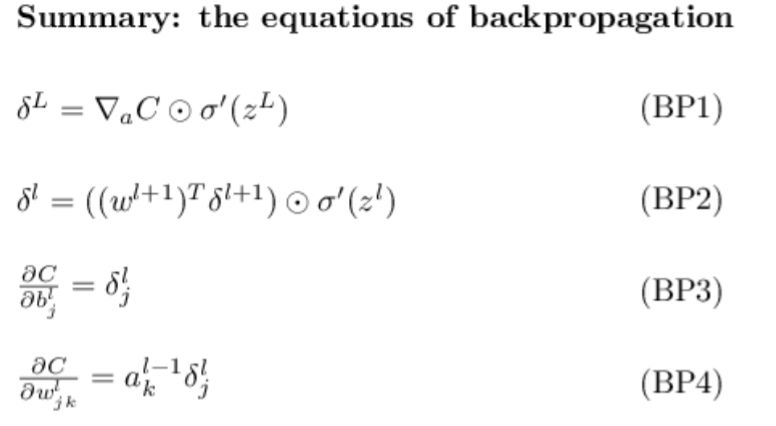


Where the sum is over all neurons k in the (l − 1)th layer. Equation can be

rewritten in the beautiful and compact vectorized form



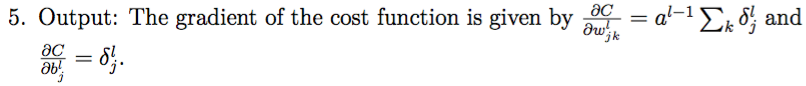
**2.2 The four fundamental equations behind backpropagation**

****

**2.3 The Backpropogation Algorithm**

The backpropagation equations provide us that computing the gradient of the cost function.

1. Input x: Set the corresponding activation a1 for the input layer.
2. Feedforward: For each l = 2,3,...,L compute zl = wlal−1 + bl and al =  σ(zl).
3. Output error δL: Compute the vector δL = ∇aC ⊙ σ′(zL).
4. Backpropagate the error: For each l = L − 1,L − 2,...,2 compute δl = ((wl+1)T δl+1) ⊙ σ′(zl).



import matplotlib.pylab as plt

import numpy as np

class QuadraticCost():

def cost(self, y, out):

return 0.5 \* ((y - out) \*\* 2.)

def diff(self, y, out):

return (out - y)

class Linear():

def activate(self, z):

return z

def diff(self, z):

return 1

class ReLU():

def activate(self, z):

return max(0, z)

def diff(self, z):

if z > 0:

return 1

else:

return 0

def compareTrainCost(\*args):

costs = []

for index, arg in enumerate(args):

(x, y) = arg[0] # input and expected output

(weight, bias) = arg[1] # initial weight and bias

costFunction = arg[2]

activationFunction = arg[3]

eta = arg[4] # learning rate

llambda = arg[5] # regularization constant

epoch = arg[6]

costs.append([])

for i in range(epoch):

z = x \* weight + bias

out = activationFunction.activate(z)

weight = weight - eta \* llambda \* weight - eta \* costFunction.diff(y, out) \* activationFunction.diff(z) \* out

bias = bias - eta \* costFunction.diff(y, out) \* activationFunction.diff(z)

costs[index].append(costFunction.cost(y, out))

# Last out value with updated weights and biases.

z = x \* weight + bias

out = activationFunction.activate(z)

costs[index].append(costFunction.cost(y, out))

# Draw train cost vs. epoch

for index, cost in enumerate(costs):

plt.plot(range(epoch + 1), cost, label="model {}".format(index + 1))

plt.xlabel('epochs')

plt.ylabel('cost')

plt.legend()

plt.show()

def compareWeightBias(\*args):

biases = []

weights = []

for index, arg in enumerate(args):

(x, y) = arg[0]

(weight, bias) = arg[1]

costFunction = arg[2]

activationFunction = arg[3]

eta = arg[4]

llambda = arg[5]

epoch = arg[6]

biases.append([])

weights.append([])

for i in range(epoch):

z = x \* weight + bias

out = activationFunction.activate(z)

weight = weight - eta \* llambda \* weight - eta \* costFunction.diff(y, out) \* activationFunction.diff(z) \* out

bias = bias - eta \* costFunction.diff(y, out) \* activationFunction.diff(z)

biases[index].append(bias)

weights[index].append(weight)

for index, weight in enumerate(weights):

plt.plot(range(epoch), weight, label="model {}".format(index + 1))

plt.xlabel('epochs')

plt.ylabel('weight')

plt.legend()

plt.show()

for index, bias in enumerate(biases):

plt.plot(range(epoch), bias, label="model {}".format(index + 1))

plt.xlabel('epochs')

plt.ylabel('bias')

plt.legend()

plt.show()

#print len(weights[0])

#print len(biases[0])

neuron1 = ((1.0, 0.0),

(0.8, 1.3),

(QuadraticCost()),

(Linear()),

0.15,

0.0,

30)

neuron2 = ((1.0, 0.0),

(0.2, 1.3),

(QuadraticCost()),

(ReLU()),

0.15,

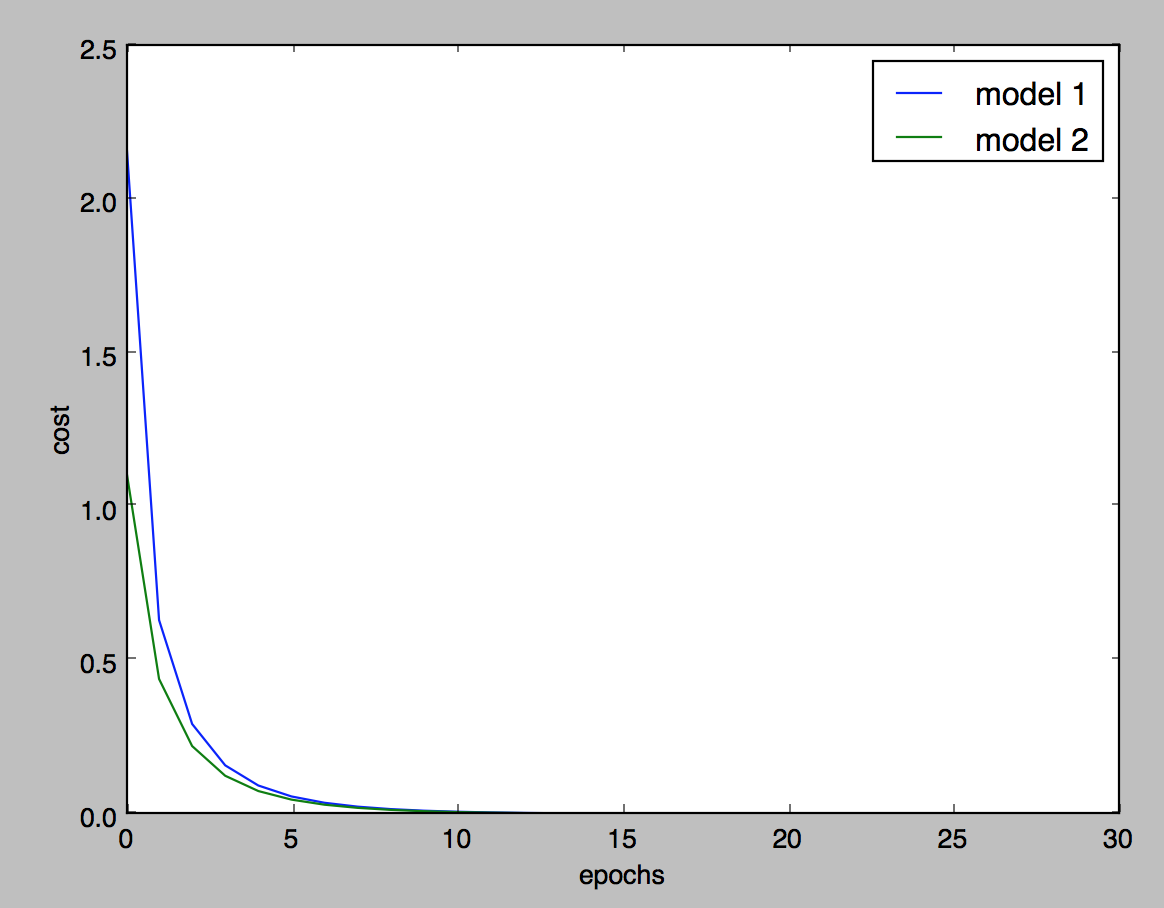
0.0,

30)

print compareTrainCost(neuron1, neuron2)

print compareWeightBias(neuron1, neuron2)

Change of the Cost. Model 1 is the neuron 1 and Model 2 is the neuron 2.



And the change of weights and biases of trained neurons:

