CNN-Lytical : Lecture 4

Soham Joshi
210051004

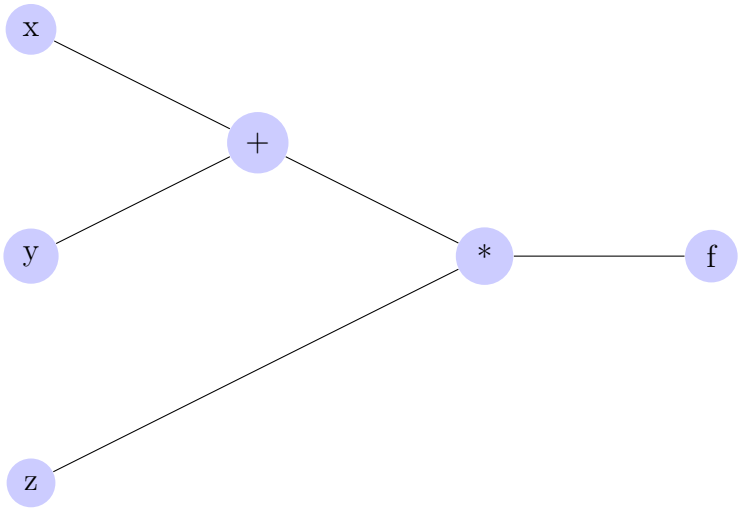
April 27, 2022

Contents

1 Backpropogation Basics

Now that we have loss functions, we need a way to calculate gradients analytically for the purpose of gradient descent. The lecture started with a simple example :

$$f(x, y, z) = (x + y)z \tag{1}$$



We saw calculation of gradients using chain rule :

$$q = x + y \tag{2}$$
$$f = qz \tag{3}$$
$$\frac{df}{dq} = z \tag{4}$$
$$\frac{df}{dz} = q \tag{5}$$
$$\frac{dq}{dx} = \frac{dq}{dy} = 1 \tag{6}$$

Hence, by chain rule :

$$\frac{df}{dx} = \frac{df}{dq}\frac{dq}{dx} = z \tag{7}$$

$$\frac{df}{dy} = \frac{df}{dq}\frac{dq}{dy} = z \tag{8}$$

$$\frac{df}{dz} = x + y \tag{9}$$

We have taken x, y, z as scalars for the purpose of demonstration but they can be scalars, vectors as well as tensors!

2 Neurons

A neuron or a node is the basic functional unit of a neural network. A single neuron performs the following tasks :

1. Input from previous neurons
2. Output to following neurons
3. Storing relevent parameters as cache for backpropogation
4. Take input gradient from following neurons and "pass" the derivatives

Neurons are used to calculate gradients as neurons can work collectively to mimic the chain rule but calculating derivatives using chain rule by hand is a rather difficult task. Hence the philosophy used is calculate derivates as far as possible using hand and chain together neurons into one unit, calculate further derivatives using collection of neurons.

Note : Gradients add at branches during backpropogation, and neural networks never have loops.

One such example of a multiplying neuron is :

```
class MultiplyGate :
def __init__(self):
    pass

def forward(self, x, y): # forward propogation
    z = x*y
    self.x = x
    self. y = y
    return z

def backward(self, dz): # backward propogation
    dx = self.y * dz
    dy = self.x * dz
    return [dx, dy]
```

3 Layers

Now, the implementation of full blown networks is done using forward/backward APIs in graph(or net) objects. There is a brief discussion regarding Torch and Caffe layers, each layer being an abstraction of a collection of neurons that enables the use of vectorized operations.

During these vectorized bacword propogation, the derivative sometimes turns out to be a jacobian matrix, in these cases often the properties of a jacobian matrix are used to pass on information rather than passing on a huge 4096x4096 matrix for instance.

Thus, a neural network can consist of many layers.

4 Assignment : Writing SVM/Softmax

In this assignment, the instructor has asked to stage forward computation using a vanilla setup (simplistic case)

```
import numpy as np

W = np.array([[3.2, 1.3, 2.2],[5.1, 4.9, 2.5],[-1.7, 2.0, -3.1]])
X = np.array([ [1, 0, 0], [0, 1, 0], [0, 0, 1] ])
cache = []
def SVM_loss(W, X):
    #forward pass
    scores = np.dot(W, X)
    examples = X.shape[1]
    columns = W.shape[0]
    scores_rows = np.zeros((examples))
    L_data = 0
    for i in range(examples):
        margin = 0
        for j in range(columns):
            loss = scores[j][i] - scores[i][i] + 1
            if (loss > 0 and loss > margin):
                margin = loss
                scores_rows[i] = j
        L_data += margin

    L_reg = 0
    for i in range(examples):
        for j in range(columns):
            L_reg += W[i][j] **2

    L_total = L_data/examples + L_reg

    #backward pass
    dscores = np.zeros(scores.shape)

    for i in range(examples) :
        dscores[int(scores_rows[i])][i] = 1
    dW = np.dot(dscores, X.T)
    cache.append((dW, dscores, W, X))

    return L_total
```

5 Neural Networks

We saw how short the algorithm of a neural network actually is using a piece of code of 11 lines provided by the instructor :

```
import numpy as np
X = np.array([ [0, 0, 1], [0, 1, 1], [1, 0, 1], [1, 1, 1] ])
Y = np.array([ [0, 0, 0, 0] ]).T
w1 = 2*np.random.random((3, 4)) - 1
w2 = 2*np.random.random((4, 1)) - 1
for j in range(10000) :
    #forward propogation
    l1 = 1/(1 + np.exp(-(np.dot(X, w1))))
    l2 = 1/(1 + np.exp(-(np.dot(l1, w2))))
    #backpropogation
    l2_delta = (Y-l2) * (l2 * (1-l2)) #logistic regression loss
    l1_delta = np.dot(l2_delta, w2.T) * l1 * (1 - l1)
    #update
    w2 += np.dot(l1.T, l2_delta)
    w1 += np.dot(X.T, l1_delta)
```

Sidenote : To be honest I don't understand the update of l1_delta and I think that the update done here might be wrong, as I got a different answer when I did this by hand.

5.1 Assignment : Writing 2 layer Net

In this assignment we have to stage forward/backward computation for a 2 layer neural network, i.e. one hidden layer

1. Inputs : W_1, W_2, b_1, b_2
2. Ouput : Activation of each layer, gradients

We can do this using the following code snippet :

```
import numpy as np

def sigmoid(v):
    return 1/(1+np.exp(-v))

def NeuralNet(W1, W2, b1, b2, X):
    #receive W1, W2, b1, b2 (weights/biases), X(data)
    #forward pass
    h1 = sigmoid(np.dot(W1, X) + b1)
    scores = sigmoid(np.dot(W2, h1) + b2)
    L_total = 0
    #softmax loss
    for i in range(X.shape[1]):
        denominator = np.sum(np.exp(scores), axis = 0,
        keepdims=True) #(row vector of denominators)
        loss = -np.log(np.exp(scores)[i][i]/denominator[i])
        L_total += loss

    #backward pass
    dscores = 0 #!! my math may be wrong.
    dW1 = dW2 = db1 = db2 = 0
```

5.2 Neural Network = Brain ?

Neural networks are often considered analogous to the brain (due media mainly :P). There is some merit to this due to a clever analogy.

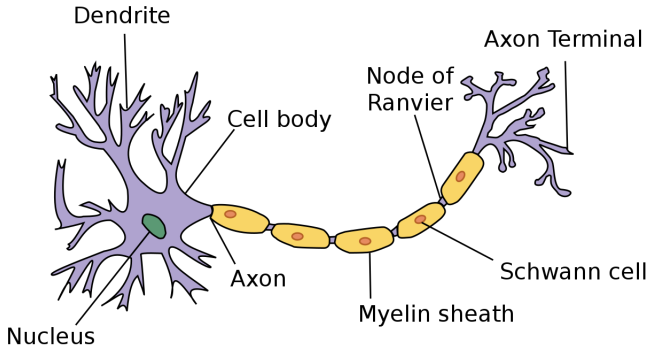


Figure 1: Neuron

The Dendrites can be treated as taking inputs from other neurons, the corresponding activation decides the firing of the signal through the axon and the terminals act as outputs to the next neuron. However this analogy is not to be pushed too much and is only for understanding purposes as :

1. Many different types of neurons exist
2. Dendrites can perform complex non-linear computations
3. Synapses are not a single weight but a complex non-linear dynamical system
4. Rate code may not be adequate

Hence, we cannot say that a neural network is equivalent to a human brain.

5.3 Activation Functions

The most common activation functions are

$$Sigmoid(x) = \frac{1}{1+e^{-x}} \tag{10}$$

$$tanh(x) = \frac{e^{x}+e^{-x}}{e^{x}-e^{-x}} \tag{11}$$

$$RELU = max(0, x) \tag{12}$$