Deep learning & Neural Network

Mohamed Ali Ashfaq Ahamed[†]

[†] *Department of Mathematics and Statistics, Memorial University of Newfoundland, St. John's (NL) A1C 5S7, Canada*

E-mails: aamohamedali@mun.ca

Abstract

Neural networks are pivotal in artificial intelligence, emulating the brain's intricate structure to tackle diverse tasks. This paper elucidates neural network fundamentals, including neurons, layers, and activation functions. We explore how neural networks self-train through gradient-based optimization like stochastic gradient descent, facilitated by the backpropagation algorithm for efficient parameter updates. Additionally, regularization techniques like dropout curb overfitting. By unraveling neural network architecture and learning mechanisms, this paper offers a concise foundation for understanding this potent AI paradigm, propelling further research and practical applications.

1 Introduction

1.1 Background

Neural networks have emerged as a cornerstone of modern artificial intelligence, revolutionizing various domains including computer vision, natural language processing, and autonomous systems. Inspired by the intricate workings of the human brain, neural networks exhibit remarkable capabilities in learning complex patterns and relationships from data. In this paper, we embark on a journey to demystify neural networks, shedding light on their architecture and the mechanisms by which they autonomously learn. There are many variant of neural networks such as convolutional neural network and long short term memory network that is adequate for image recognition & speech recognition respectively.

At the heart of neural networks lie neurons, the basic computational units that process information. These neurons are organized into layers, forming interconnected networks capable of extracting hierarchical representations from raw input data. Activation functions introduce non-linearities to the network, enabling it to model complex relationships and capture intricate patterns.

However, the true power of neural networks lies not only in their architecture but also in their ability to self-train and adapt to diverse tasks. This process of learning is achieved

through iterative optimization of network parameters, a task facilitated by gradient-based optimization algorithms such as stochastic gradient descent. By iteratively adjusting the parameters based on the gradients of a chosen loss function, neural networks gradually converge towards optimal solutions.

Key to this learning process is the backpropagation algorithm, which efficiently computes gradients of the loss function with respect to each parameter in the network. This enables error signals to propagate backward through the network, guiding parameter updates to minimize prediction errors. Moreover, regularization techniques such as dropout and weight decay are employed to prevent overfitting and enhance generalization performance.

2 Analysis

What are neurons? and How are they connected? before you get an insight on neural network consider neurons as unit that holds a number.

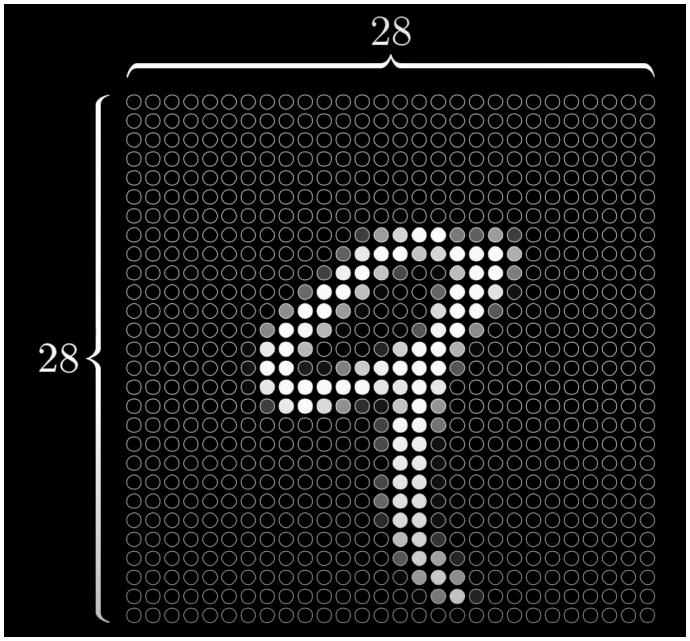

Figure 1: An image of a handwritten digit at a resolution of 28x28 pixel.

For an example consider a neural network that can learn to recognize handwritten digits. The network starts with a bunch of neurons corresponding to each of the 28x28 pixels of the input image, which is 784 neurons in total. Each one of these holds a number that represents the grayscale value of the corresponding pixel, ranging from 0 for black pixels up to 1 for white pixels. This number inside the neuron is called its activation. Each neuron is lit up when its activation is a high number.

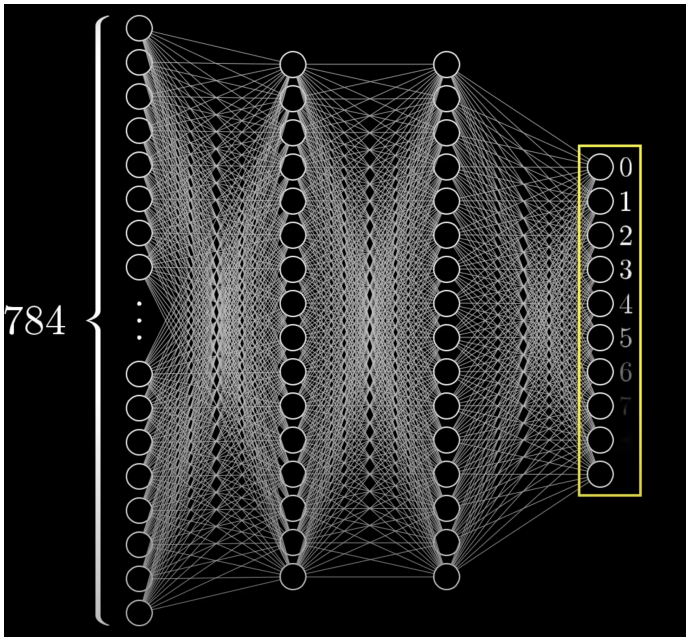

Figure 2: An abstract level of the neural network to recognize handwritten digits.

All of these 784 neurons make up the first layer of our network and the last layer has 10 neurons, each representing one of the digits. The activation in the last layer of neurons is number in between 0 and 1 that represents how much the system is certain that a given image corresponds with a digit.There is also couple of layers in between first layer and last layer which is called hidden layer.

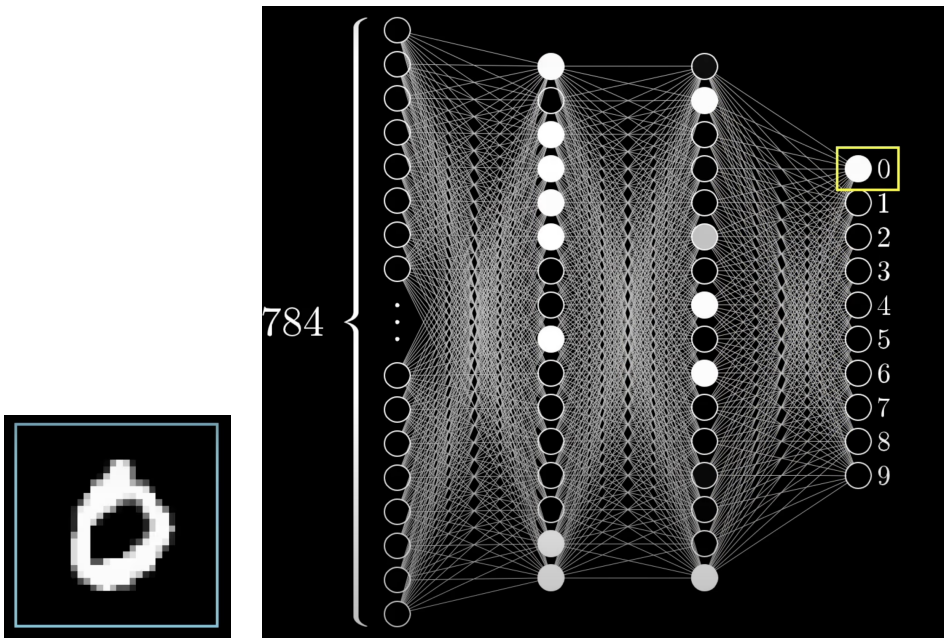

Figure 3: Feeding a image of zero triggers multiple layer in the hidden layer and activates a neuron in the last layer..

In this network i choose two hidden layers each layers with 16 neurons, based on how I want to motivate the structure of my network. The way the network operates is the

activation's in one layer determine the activation's of the next layer. When i feed in an image, according to the brightness of each pixel in the image, it lights up some neurons of the input layer, this pattern of activation's causes some very specific pattern in the next layer, which then causes some pattern in the one after it, which finally gives some pattern in the output layer. And the brightest neuron of that output layer is the network's choice.

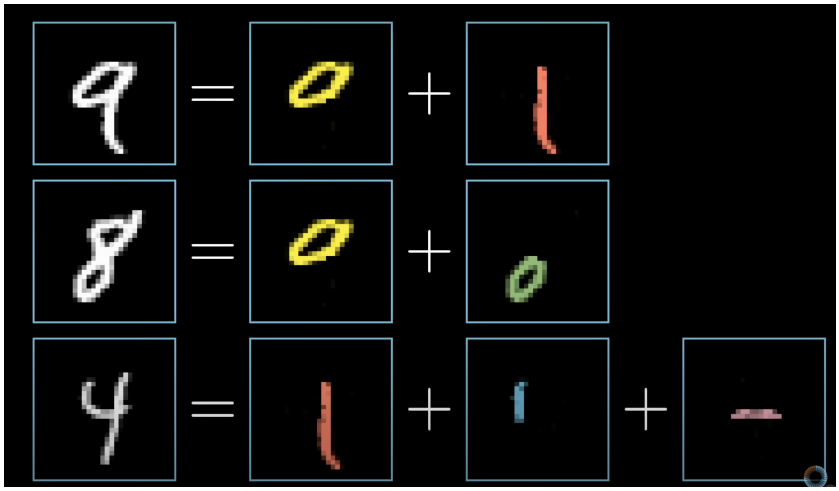

Figure 4: The required subcompnents of building a digit.

Why it's even reasonable to expect a layered structure to behave intelligently? When you recognize digits, we piece together various components. A 9 has a loop up top and a line on the right. An 8 also has a loop up top, but it's paired with another loop down bottom. A 4 basically breaks down into three specific lines. Each neuron in the second to last layer corresponds with one of these sub components that anytime you feed in an image with, a loop up top, like a 9 or an 8, there's some specific neuron whose activation close to 1. Any generally loopy pattern top sets off this neurons. Going from the third layer to the last layer requires learning which combination of sub components corresponds to which digits.

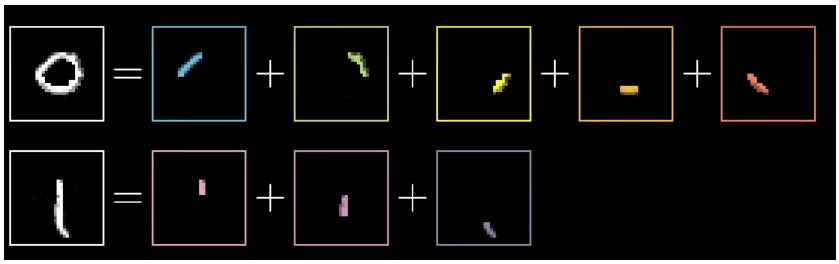

Figure 5: The required subcompnents of building patterns of digits.

Recognizing a loop can also break down into sub problems. One reasonable way to do this would be to first recognize the various little edges that make it up. Similarly, a long line, that you see in the digits 1 or 4 or 7, is a long edge, or certain pattern of several smaller edges. Each neuron in the second layer of the network corresponds with the various relevant little edges. When an image of 9 comes in, it lights up all of the neurons associated with around 8 to 10 specific little edges, which in turn lights up the

neurons associated with the upper loop and a long vertical line, and those light up the neuron associated with a 9. The goal is to have some mechanism that could conceivably combine pixels into edges, edges into patterns, and patterns into digits.

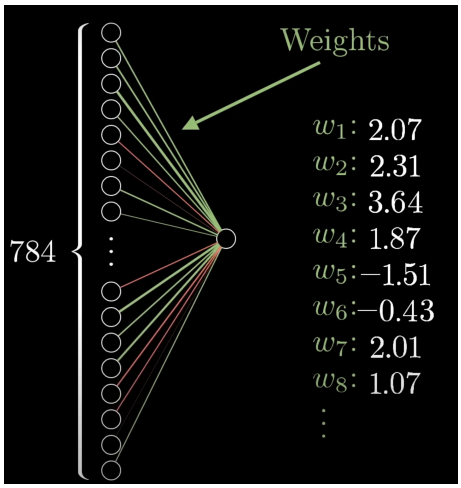

Figure 6: parameters that are used to activate a neuron from the previous layers.

What parameters should the network have for one particular neuron in the second layer to pick up on whether or not the image has an edge? Assigning a weight to each one of the connections between a neuron in the second layer and the neurons from the first layer, then taking all of those activations from the first layer and compute their weighted sum according to those weights.

$$\hat{Y} = a_1w_1 + a_2w_2 + a_3w_3 + a_4w_4 + a_5w_5..... + a_{784}w_{784}$$

When you compute a weighted sum , you could come with any number, but for this network what we need is for activations to be some value between 0 and 1. So a common thing to do is to pump this weighted sum into some function that squishes the real number line into the range between 0 and 1. The common function that does this is called the sigmoid function, also known as a logistic curve. it's not that we want the neuron to light up when the weighted sum is bigger than 0. We only want it to be active when the sum is bigger than 10. . Just adding in some number like negative 10 to this weighted sum before plugging it through the sigmoid squishification function is a bias for it to be inactive. The additional number is called the bias.

$$\hat{Y} = a_1w_1 + a_2w_2 + a_3w_3 + a_4w_4 + a_5w_5..... + a_{784}w_{784} - 10$$

So the weights tell you what pixel pattern this neuron in the second layer is picking up on, and the bias tells you how high the weighted sum needs to be before the neuron starts getting meaningfully active.

3 Conclusion

How does neural learn? an algorithm where i show this network a whole bunch of training data, which comes in the form of a bunch of different images of handwritten digits, along

with labels for what they're supposed to be, and it'll adjust those 13,000 weights and biases so as to improve its performance on the training data. Finally it will generalizes to images beyond that training data.

Conceptually, we are certain each neuron as being connected to all the neurons in the previous layer, the weights in the weighted sum defining its activation as the strengths of those connections, and the bias is some indication of whether that neuron tends to be active or inactive. At the beginning initializing all of those weights and biases randomly, this network will perform horrible.

For instance you feed in an image of 3 and the output layer would just looks like a mess. now you define a cost function, a way of telling the computer, that output should have activation which are 0 for most neurons, but 1 for corresponding 3 digit neuron.

$$costfunction = (1stNeuronDifference)^2 ... + (lastNeuronDifference)^2$$

Mathematically, you add up the squares of the differences between each of those trash output activation and the value you want them to have, and this is what we are calling the cost of a single training example. Notice this sum is small when the network confidently classifies the image correctly, but it's large when the network doesn't know what it's doing. Considering the average cost over all of the tens of thousands of training examples in your training data set, the average cost is the measure of how awful the network is, and how bad the computer should feel.

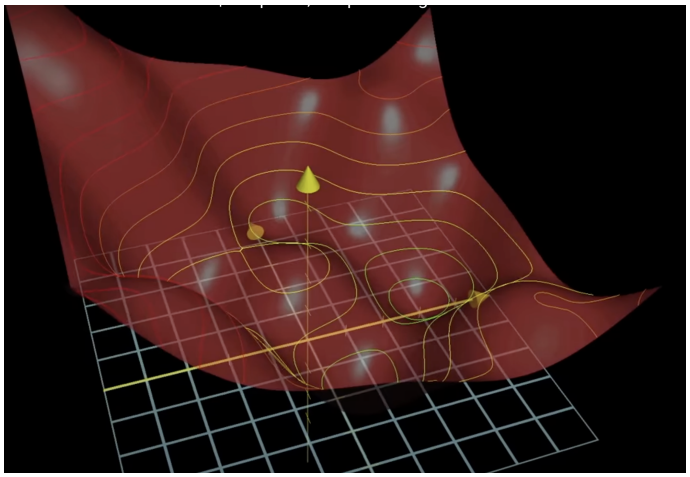

Figure 7: The plot drawn from the cost function determined from all the training data set.

Just telling the computer what a lousy job it has done isn't very helpful. We have to tell it how to change those weights and biases so that it gets better. Imagine input space as the xy-plane, and the cost function as being graphed as a surface above it. Instead of examining the slope of the function we have to find which direction we should step. Naturally enough, taking the negative of the gradient gives us the direction that would decrease the function more quickly. The algorithm for minimizing the function is computing the gradient direction, then taking a small step downhill, and repeat that over and over. Changing the weights and biases in the cost function to decrease is by making the output of the network on each piece of training data looks a random array of

10 values, is an actual decision we want it to make.
It's important to remember, cost function involves an average over all of the training data, so if you minimize it, it means it's a better performance on all of those samples. The algorithm for computing this gradient efficiently, which is effectively the heart of how a neural network learns, is called back-propagation. The process of repeatedly nudging an input of a function by some multiple of the negative gradient is called gradient descent. It's a way to converge towards some local minimum of a cost function, basically a valley in the graph. Each component of the negative gradient tells us two things. The sign, tells us whether the corresponding component of the input vector should be nudged up or down and importantly, the relative magnitudes of all these components tells us which changes matter more.

4 Acknowledgement

Nielsen, M. (2019). Neural Networks and Deep Learning. [Online book]. Available from https://neuralnetworksanddeeplearning.com

5 Appendix

```
#Author: Nielsen, M
# Standard library
import cPickle
import gzip

# Third-party libraries
import numpy as np
import theano
import theano.tensor as T
from theano.tensor.nnet import conv
from theano.tensor.nnet import softmax
from theano.tensor import shared_randomstreams
from theano.tensor.signal import downsample

# Activation functions for neurons
def linear(z): return z
def ReLU(z): return T.maximum(0.0, z)
from theano.tensor.nnet import sigmoid
from theano.tensor import tanh


#### Constants
GPU = True
if GPU:
    print "Trying to run under a GPU.  If this is not desired, then modify "+\
        "network3.py\nto set the GPU flag to False."
    try: theano.config.device = 'gpu'
    except: pass # it's already set
```

```
    theano.config.floatX = 'float32'
else:
    print "Running with a CPU.  If this is not desired, then the modify "+\
        "network3.py to set\nthe GPU flag to True."

#### Load the MNIST data
def load_data_shared(filename="../data/mnist.pkl.gz"):
    f = gzip.open(filename, 'rb')
    training_data, validation_data, test_data = cPickle.load(f)
    f.close()
    def shared(data):
        """Place the data into shared variables.  This allows Theano to copy
        the data to the GPU, if one is available.

        """
        shared_x = theano.shared(
            np.asarray(data[0], dtype=theano.config.floatX), borrow=True)
        shared_y = theano.shared(
            np.asarray(data[1], dtype=theano.config.floatX), borrow=True)
        return shared_x, T.cast(shared_y, "int32")
    return [shared(training_data), shared(validation_data), shared(test_data)]

#### Main class used to construct and train networks
class Network(object):

    def __init__(self, layers, mini_batch_size):
        """Takes a list of `layers`, describing the network architecture, and
        a value for the `mini_batch_size` to be used during training
        by stochastic gradient descent.

        """
        self.layers = layers
        self.mini_batch_size = mini_batch_size
        self.params = [param for layer in self.layers for param in layer.params]
        self.x = T.matrix("x")
        self.y = T.ivector("y")
        init_layer = self.layers[0]
        init_layer.set_inpt(self.x, self.x, self.mini_batch_size)
        for j in xrange(1, len(self.layers)):
            prev_layer, layer  = self.layers[j-1], self.layers[j]
            layer.set_inpt(
                prev_layer.output, prev_layer.output_dropout, self.mini_batch_size)
        self.output = self.layers[-1].output
        self.output_dropout = self.layers[-1].output_dropout

    def SGD(self, training_data, epochs, mini_batch_size, eta,
            validation_data, test_data, lmbda=0.0):
        """Train the network using mini-batch stochastic gradient descent."""
```

```
training_x, training_y = training_data
validation_x, validation_y = validation_data
test_x, test_y = test_data

# compute number of minibatches for training, validation and testing
num_training_batches = size(training_data)/mini_batch_size
num_validation_batches = size(validation_data)/mini_batch_size
num_test_batches = size(test_data)/mini_batch_size

# define the (regularized) cost function, symbolic gradients, and updates
l2_norm_squared = sum([(layer.w**2).sum() for layer in self.layers])
cost = self.layers[-1].cost(self)+\
       0.5*lmbda*l2_norm_squared/num_training_batches
grads = T.grad(cost, self.params)
updates = [(param, param-eta*grad)
           for param, grad in zip(self.params, grads)]

# define functions to train a mini-batch, and to compute the
# accuracy in validation and test mini-batches.
i = T.lscalar() # mini-batch index
train_mb = theano.function(
    [i], cost, updates=updates,
    givens={
        self.x:
        training_x[i*self.mini_batch_size: (i+1)*self.mini_batch_size],
        self.y:
        training_y[i*self.mini_batch_size: (i+1)*self.mini_batch_size]
    })
validate_mb_accuracy = theano.function(
    [i], self.layers[-1].accuracy(self.y),
    givens={
        self.x:
        validation_x[i*self.mini_batch_size: (i+1)*self.mini_batch_size],
        self.y:
        validation_y[i*self.mini_batch_size: (i+1)*self.mini_batch_size]
    })
test_mb_accuracy = theano.function(
    [i], self.layers[-1].accuracy(self.y),
    givens={
        self.x:
        test_x[i*self.mini_batch_size: (i+1)*self.mini_batch_size],
        self.y:
        test_y[i*self.mini_batch_size: (i+1)*self.mini_batch_size]
    })
self.test_mb_predictions = theano.function(
    [i], self.layers[-1].y_out,
    givens={
        self.x:
```

```
                test_x[i*self.mini_batch_size: (i+1)*self.mini_batch_size]
            })
        # Do the actual training
        best_validation_accuracy = 0.0
        for epoch in xrange(epochs):
            for minibatch_index in xrange(num_training_batches):
                iteration = num_training_batches*epoch+minibatch_index
                if iteration % 1000 == 0:
                    print("Training mini-batch number {0}".format(iteration))
                cost_ij = train_mb(minibatch_index)
                if (iteration+1) % num_training_batches == 0:
                    validation_accuracy = np.mean(
                        [validate_mb_accuracy(j) for j in xrange(num_validation_batch
                    print("Epoch {0}: validation accuracy {1:.2%}".format(
                        epoch, validation_accuracy))
                    if validation_accuracy >= best_validation_accuracy:
                        print("This is the best validation accuracy to date.")
                        best_validation_accuracy = validation_accuracy
                        best_iteration = iteration
                        if test_data:
                            test_accuracy = np.mean(
                                [test_mb_accuracy(j) for j in xrange(num_test_batches
                            print('The corresponding test accuracy is {0:.2%}'.format
                                test_accuracy))
        print("Finished training network.")
        print("Best validation accuracy of {0:.2%} obtained at iteration {1}".format(
            best_validation_accuracy, best_iteration))
        print("Corresponding test accuracy of {0:.2%}".format(test_accuracy))

#### Define layer types

class ConvPoolLayer(object):
    """Used to create a combination of a convolutional and a max-pooling
    layer.  A more sophisticated implementation would separate the
    two, but for our purposes we'll always use them together, and it
    simplifies the code, so it makes sense to combine them.

    """

    def __init__(self, filter_shape, image_shape, poolsize=(2, 2),
                 activation_fn=sigmoid):
        """`filter_shape` is a tuple of length 4, whose entries are the number
        of filters, the number of input feature maps, the filter height, and the
        filter width.

        `image_shape` is a tuple of length 4, whose entries are the
        mini-batch size, the number of input feature maps, the image
        height, and the image width.
```

```
        `poolsize` is a tuple of length 2, whose entries are the y and
        x pooling sizes.

        """
        self.filter_shape = filter_shape
        self.image_shape = image_shape
        self.poolsize = poolsize
        self.activation_fn=activation_fn
        # initialize weights and biases
        n_out = (filter_shape[0]*np.prod(filter_shape[2:])/np.prod(poolsize))
        self.w = theano.shared(
            np.asarray(
                np.random.normal(loc=0, scale=np.sqrt(1.0/n_out), size=filter_shape),
                dtype=theano.config.floatX),
            borrow=True)
        self.b = theano.shared(
            np.asarray(
                np.random.normal(loc=0, scale=1.0, size=(filter_shape[0],)),
                dtype=theano.config.floatX),
            borrow=True)
        self.params = [self.w, self.b]

    def set_inpt(self, inpt, inpt_dropout, mini_batch_size):
        self.inpt = inpt.reshape(self.image_shape)
        conv_out = conv.conv2d(
            input=self.inpt, filters=self.w, filter_shape=self.filter_shape,
            image_shape=self.image_shape)
        pooled_out = downsample.max_pool_2d(
            input=conv_out, ds=self.poolsize, ignore_border=True)
        self.output = self.activation_fn(
            pooled_out + self.b.dimshuffle('x', 0, 'x', 'x'))
        self.output_dropout = self.output # no dropout in the convolutional layers

class FullyConnectedLayer(object):

    def __init__(self, n_in, n_out, activation_fn=sigmoid, p_dropout=0.0):
        self.n_in = n_in
        self.n_out = n_out
        self.activation_fn = activation_fn
        self.p_dropout = p_dropout
        # Initialize weights and biases
        self.w = theano.shared(
            np.asarray(
                np.random.normal(
                    loc=0.0, scale=np.sqrt(1.0/n_out), size=(n_in, n_out)),
                dtype=theano.config.floatX),
            name='w', borrow=True)
```

```
        self.b = theano.shared(
            np.asarray(np.random.normal(loc=0.0, scale=1.0, size=(n_out,)),
                       dtype=theano.config.floatX),
            name='b', borrow=True)
        self.params = [self.w, self.b]

    def set_inpt(self, inpt, inpt_dropout, mini_batch_size):
        self.inpt = inpt.reshape((mini_batch_size, self.n_in))
        self.output = self.activation_fn(
            (1-self.p_dropout)*T.dot(self.inpt, self.w) + self.b)
        self.y_out = T.argmax(self.output, axis=1)
        self.inpt_dropout = dropout_layer(
            inpt_dropout.reshape((mini_batch_size, self.n_in)), self.p_dropout)
        self.output_dropout = self.activation_fn(
            T.dot(self.inpt_dropout, self.w) + self.b)

    def accuracy(self, y):
        "Return the accuracy for the mini-batch."
        return T.mean(T.eq(y, self.y_out))

class SoftmaxLayer(object):

    def __init__(self, n_in, n_out, p_dropout=0.0):
        self.n_in = n_in
        self.n_out = n_out
        self.p_dropout = p_dropout
        # Initialize weights and biases
        self.w = theano.shared(
            np.zeros((n_in, n_out), dtype=theano.config.floatX),
            name='w', borrow=True)
        self.b = theano.shared(
            np.zeros((n_out,), dtype=theano.config.floatX),
            name='b', borrow=True)
        self.params = [self.w, self.b]

    def set_inpt(self, inpt, inpt_dropout, mini_batch_size):
        self.inpt = inpt.reshape((mini_batch_size, self.n_in))
        self.output = softmax((1-self.p_dropout)*T.dot(self.inpt, self.w) + self.b)
        self.y_out = T.argmax(self.output, axis=1)
        self.inpt_dropout = dropout_layer(
            inpt_dropout.reshape((mini_batch_size, self.n_in)), self.p_dropout)
        self.output_dropout = softmax(T.dot(self.inpt_dropout, self.w) + self.b)

    def cost(self, net):
        "Return the log-likelihood cost."
        return -T.mean(T.log(self.output_dropout)[T.arange(net.y.shape[0]), net.y])

    def accuracy(self, y):
```

```
        "Return the accuracy for the mini-batch."
        return T.mean(T.eq(y, self.y_out))


#### Miscellanea
def size(data):
    "Return the size of the dataset `data`."
    return data[0].get_value(borrow=True).shape[0]

def dropout_layer(layer, p_dropout):
    srng = shared_randomstreams.RandomStreams(
        np.random.RandomState(0).randint(999999))
    mask = srng.binomial(n=1, p=1-p_dropout, size=layer.shape)
    return layer*T.cast(mask, theano.config.floatX)
```