a beginner's guide to neural networks

matrices, classification & a whole lotta fun

a short e-book by katie zaback

what is this e-book?

well, first of all its not really a book. it's more of a guide to help *you*, wonderful reader, get your hands dirty with some good ol' fashioned machine learning.

often, the resources available online and in bookstores about machine learning algorithms and libraries can be very overwhelming. lots of complicated math, lots of symbols, lots of glossing over confusing theories, and lots of frustration.

my hope is that, with this guide, you will be able to dip your toes into the vast waters of machine learning and neural networks. this is by no means an exhaustive resource. we won't get into the depths of how libraries work, or the minute details of all the mathematical computations. this guide is simply to help you start your journey as a practitioner of machine learning.

my hope is that by the end of reading this, you will have implemented two different learning algorithms (a basic backpropagation neural network, and a convolutional network) and that you will understand conceptually how they function. we'll be solving the "hello world" of machine learning algorithms: the MNIST data classification problem.

don't worry -- it's gonna be great!

go through this guide slowly. lose yourself down twists and turns of different articles you may find online along the way, and look up different resources and algorithms on your own! be adventurous! the pages of this "book" will be there if you get overwhelmed and need to get back to the basics.

at the end of all this, you will have built your first neural net (and your second!). you'll get your computer to do things that are, in some ways, on the bleeding edge of our technological knowledge base.

but trust me, it's gonna be fun. let's get started!

part I: installing the basics

the start of your journey

before we dive into the nitty-gritty of deep learning, we're going to get the basic installations we will need you to have out of the way.

there are three main libraries that you will need:

- theano
- tensorflow
- keras

all of these libraries are created for python, so you will need that installed as well. in the upcoming pages, we will walk you through how to get all of these things on your computer.

if you get stuck, there are also resources and links on each page that should help.

get excited! your journey into deep learning is about to begin!

NOTE: the installation process described in the following sections worked for me, on my MacBook Pro (OS X 10.10.5)

since machines, hardware and software differ please read & follow these guidelines with a grain of salt.

in addition to the instructions, there are links to all of the libraries you will need to download. it is recommended that you start by looking at the documentation to see which installation process will work best for your specific OS environment.

cheers, and let's get learning!

(resources/links last updated 06 December 2016)

basic python & libraries

in order to begin coding up different learning algorithms, we will be using a few libraries available in python. as such, there's some housekeeping we need to do before we get started.

everyone has different versions & libraries for python installed on their machine. some of you may not even have python installed at all.

the easiest thing to do is to start by downloading **anaconda**, which is a distribution of python and R and includes a bunch of different packages (free for academic & personal use).

if you can, download the **MLK optimizations** option (this will help when installing other libraries later)

NOTE: if you think you already have all the necessary versions of basic python packages (i.e. python 3.3 or higher, the latest version of scipy & numpy) then you may be able to skip this step and move on.

download here:
https://www.continuum.io/downloads

what is tensorflow?

tensorflow is an open-source python library created by google. it uses “data flow graphs” that consist of nodes representing mathematical operations and edges that represent multi-dimensional arrays (termed “tensors”) which flow through the graph. it can function on both CPUs and GPUs, and performs mathematical computations quickly.

what is theano?

theano is an open-source python library that is popular for implementing deep learning algorithms, and was built specifically to allow for fast and efficient implementation of neural networks. it is extremely efficient at evaluating mathematical equations, and can be used on both CPUs and GPUs. it builds symbolic graphs that allow for fast differentiation and gradient computation.

installing theano

if you have already successfully installed anaconda, then it should be very easy to get theano on your machine.

you can simply type the following command into terminal, and theano should install:

```
$ pip install Theano
```

this method worked for me on my machine, but you may encounter some issues. if this happens, theano has a bunch of different recipes for successful installation on their website.

if you would rather look for a solution specific to your OS (or need help troubleshooting), go here:

http://deeplearning.net/software/theano/install.html

installing tensorflow

similarly, if you already have anaconda installed
getting tensorflow on your machine should be easy.

you can simply type the following command into
terminal, and tensorflow should install:

```
$ pip install tensorflow
```

again, different methods will work for different
machines. you can read over the installation methods
for different operating systems here:

https://www.tensorflow.org/versions/r0.12/get_started/os_setup.html

what is keras?

both theano and tensorflow (while powerful) are often difficult to use & debug, particularly for beginners. so, we are going to use a wrapper library.

keras runs either theano or tensorflow as its backend, and essentially just provides an easy-to-use interface. all of the hard labor is still done by either theano or tensorflow (which is good, because both of the libraries are powerful).

we will learn more about the specific structures and functions available in keras, but for now let's just get it installed so we can start building some models.

installing keras

after theano & tensorflow are installed, we are ready to install keras. this will be the library we use for all our coding.

type the following command into terminal:

```
$ sudo pip install keras
```

if you do not have both theano and tensorflow installed, you will most likely get errors when you try and run code later.

you can check which version of keras you are running with the following command:

```
$ python -c “import keras; print(keras.__version__)”
```

you will probably also get an error running that command if you don’t have both theano & tensorflow installed. if you need to update keras, use this:

```
$ sudo pip install --upgrade keras
```

for installation troubleshooting and documentation, visit the keras website:

https://keras.io/#installation

part II: defining the problem space

what is MNIST?

so now that we have our coding environment all set up, we're going to dive into the learning problem we'll be solving: **image classification.**

specifically, we're going to be building two different neural network (NN) models that solve the MNIST handwritten digit classification problem. this dataset consists of 70,000 images, each containing a single handwritten digit. The dataset is split up into two pools: 60,000 images for training, and 10,000 images for testing.

the MNIST dataset is very commonly used to test different machine learning algorithms, and has become a typical "hello world" learning task for newcomers to machine learning.

before we can start coding, let's talk about what exactly these images look like and how we might classify them.

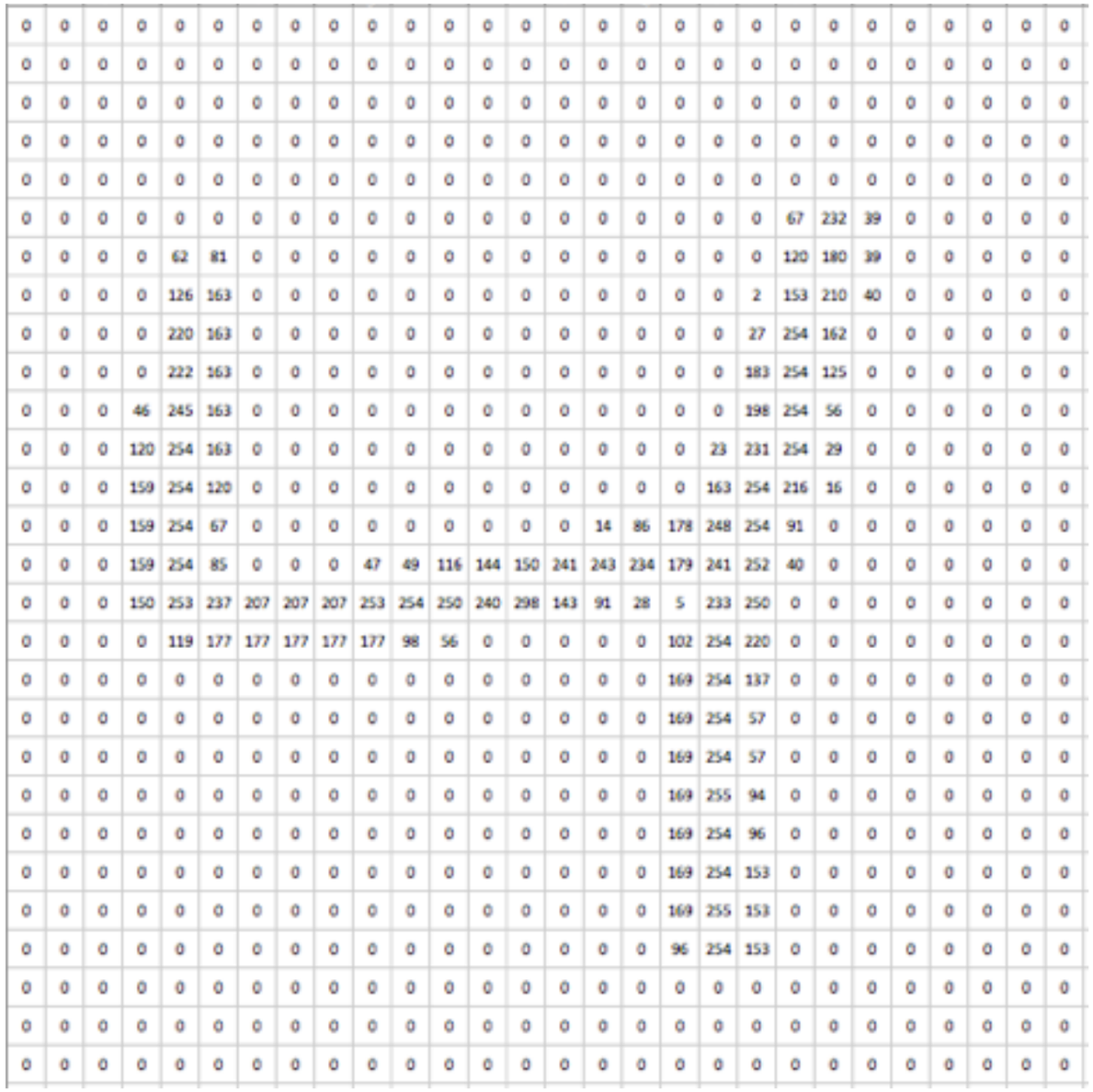
if you'd like to read up more about the MNIST dataset (or even download the files and peruse) go here:

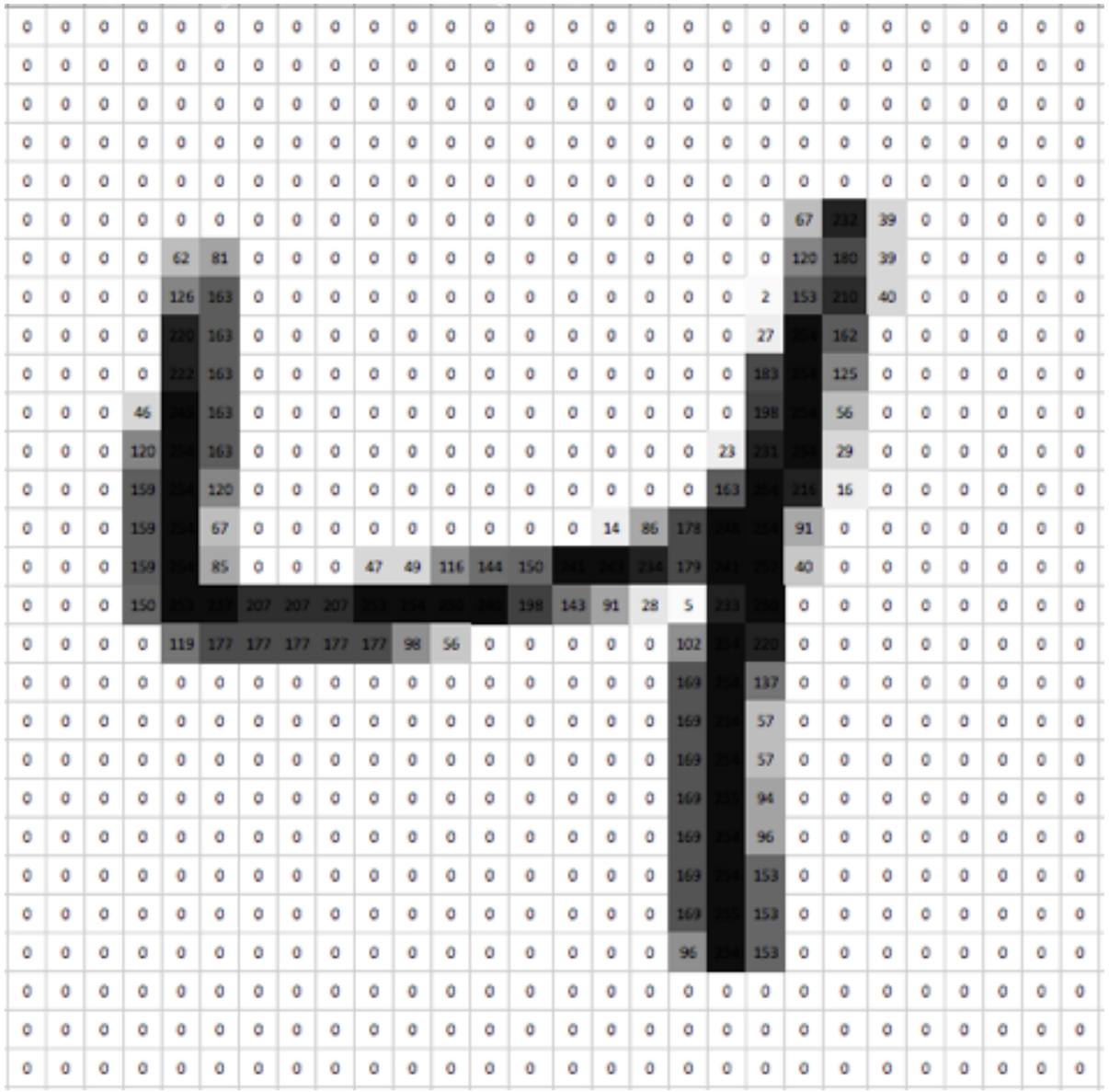
http://yann.lecun.com/exdb/mnist/

what we're working with

each sample in the MNIST dataset will consist of a greyscale image of a single handwritten digit. the image is 28px by 28px, and is stored as a two-dimensional matrix. because the image is greyscale, the color of each pixel is represented by an integer from the range 0-255 (where 0 = absolutely white and 255 = absolutely black).

we can visualize a single image as a 2-D matrix, with each index representing the color value of a single pixel.





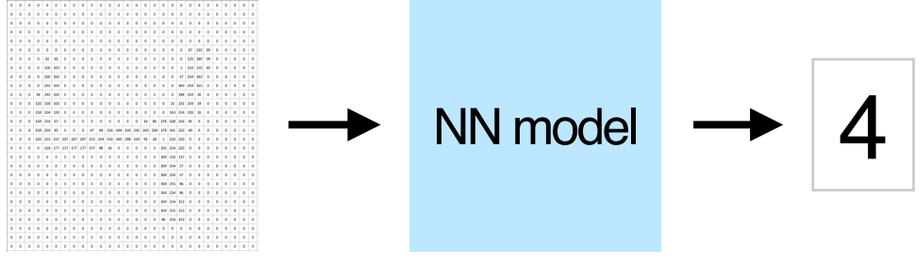
how computers “see”

the images above show what a MNIST sample looks like to a computer. notice how the grid is a two-dimensional matrix, with each slot representing a single pixel. you can see that the values stored for each pixel are a single integer. the higher the number, the darker the pixel. it will be helpful to keep this visual in mind when we start inputting our data for our model.

the goal of our system

what we would like to do is build a system that can take one of these samples as input (i.e. a single 2-D matrix representing a single handwritten digit) and output which digit it is [0-9].

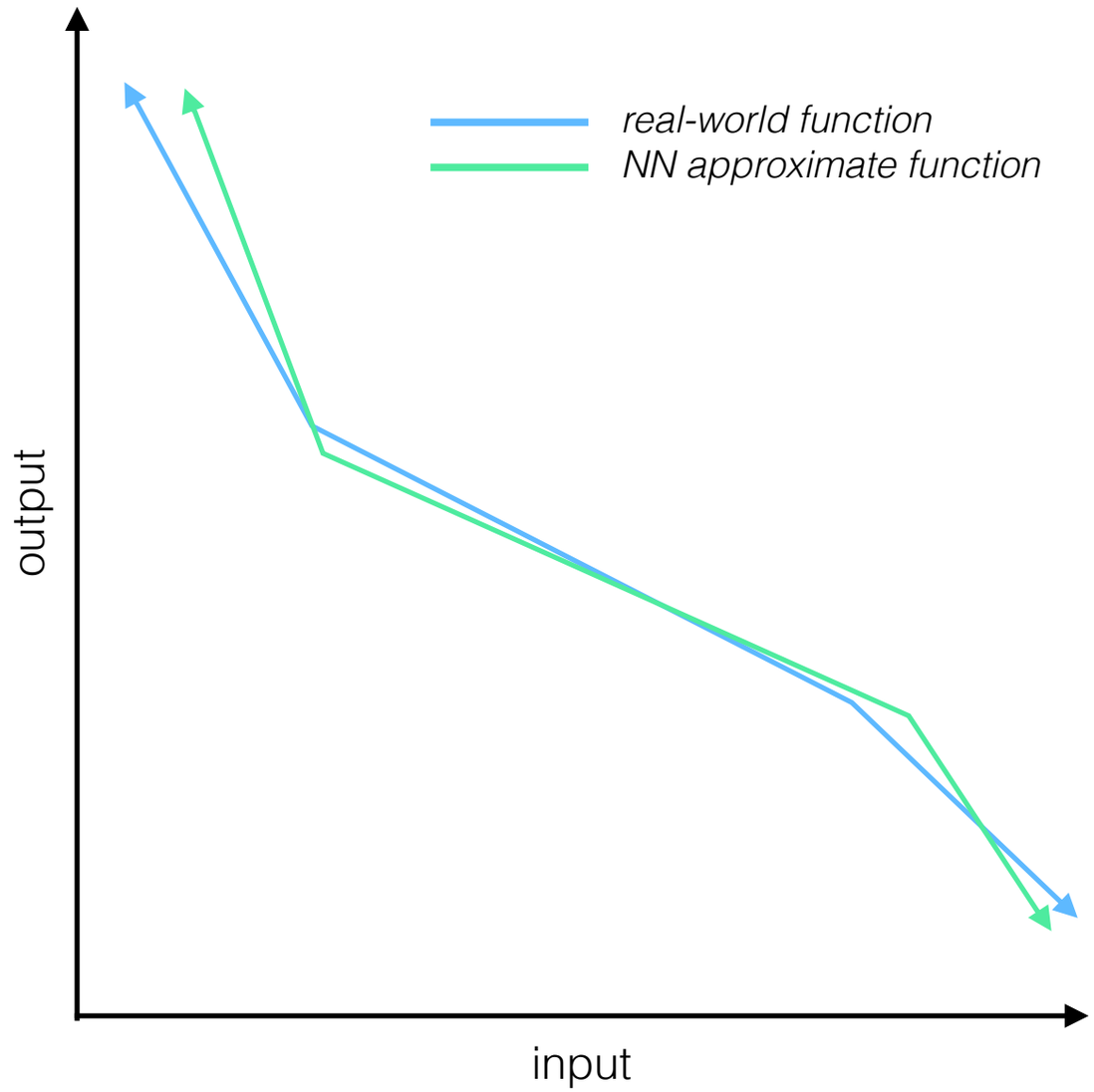
a very simplistic outline of our system design might look something like this:



so, as shown above, our input will be a sample from the MNIST dataset (in the form of a 2-D matrix) and our output should be the correct **classification** (or label) that should be assigned to that image.

we will be using two different types of neural networks (NNs) to solve this problem. before we build them in code, let's talk about the theory behind NNs and how they function.

part III: the basics of neural networks



function approximation

the neural network model is an excellent function approximator, meaning that if you provide a NN with some input, it will give you the correct output within some margin of error.

for example, imagine that the blue line of the graph on the left represents some **real-world function**. the green line then represents the output of a NN trained on that function. the smaller the margin of error of that net, the closer the green line gets to that real-world function.

as you can imagine, this is not particularly useful if you already *know* the real-world function. if you did, you wouldn't need a neural network. you would just code up the real-world function and get the output correct 100% of the time.

but neural nets do something for us that other models and systems cannot: they can approximate functions that we can't describe or even begin to know the shape of.

so while training a net to do simple addition makes no sense and wouldn't be useful, training a net to classify handwritten digits definitely would be.

the question is: how do they work?

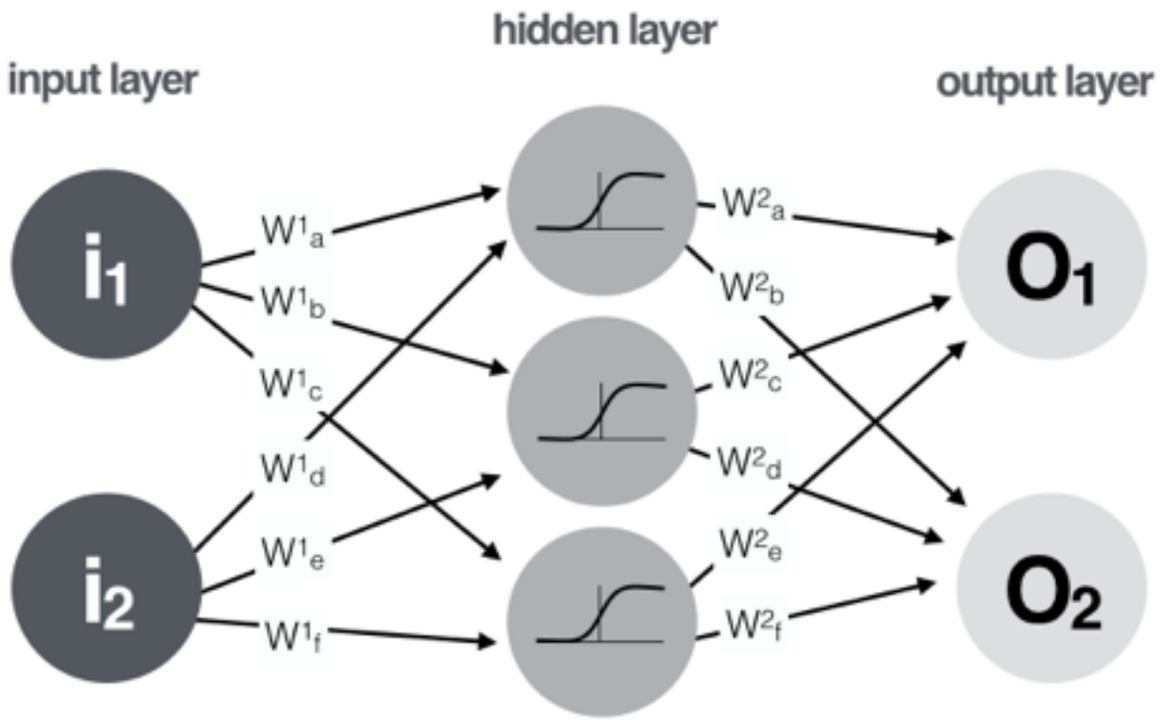
forward propagation

on the bottom right is a simple neural network model. we're going to start by going through the basic steps of a "pass" through the network.

the circles represent nodes of the network. each input node will represent a single input value (such as the color value of a pixel). the arrows represent weights, and each weight is a numeric value (the Ws on the model).

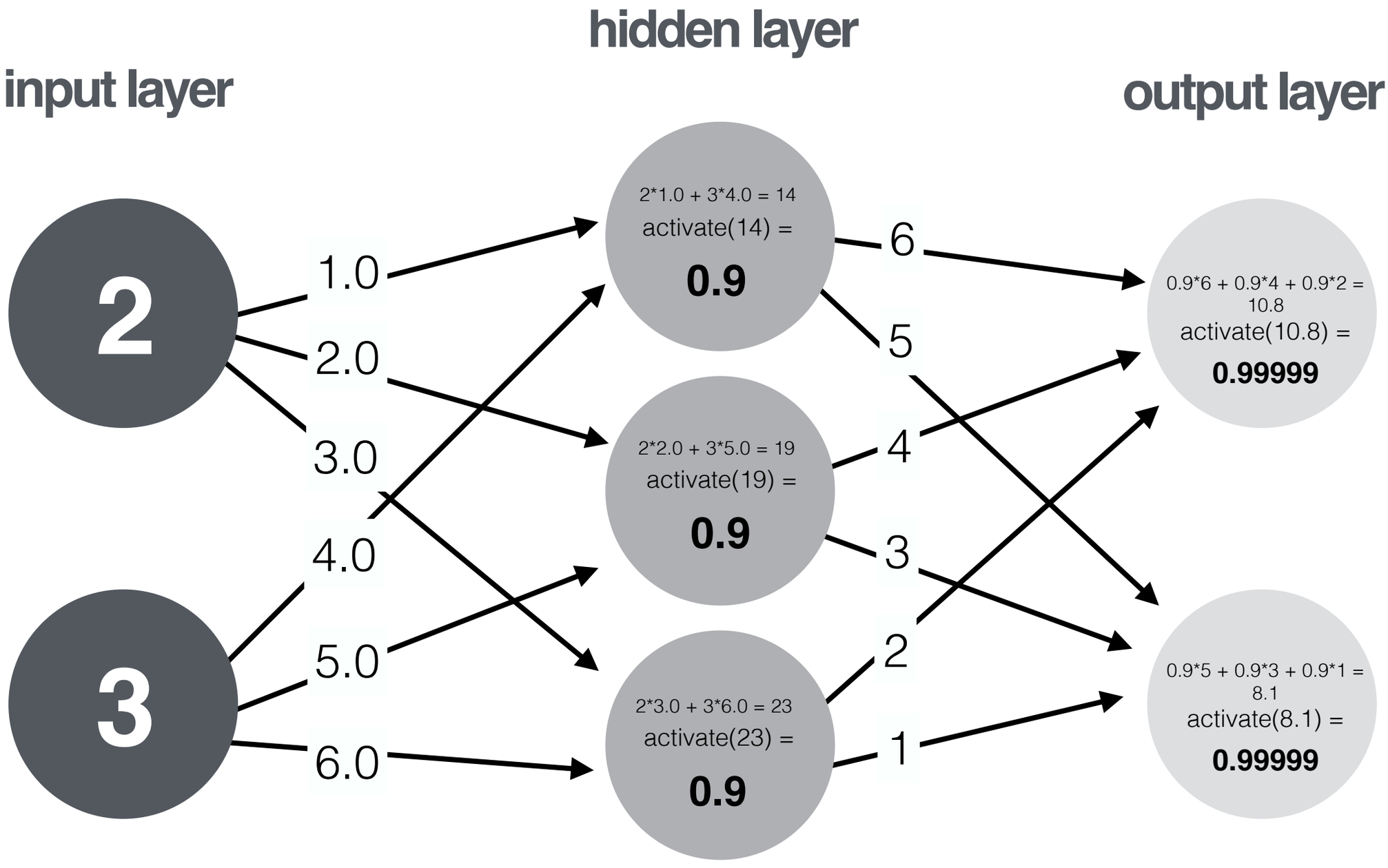
the network is essentially a form of numeric manipulation to transform the input values [i.e. i_1, i_2] into the desired output values [i.e. O_1, O_2]. the steps to get from input values to output values are as follows:

#1 multiply input values by the weight values
#2 each hidden layer node will add together all the (input * weight) values that are passed to it
#3 apply some sort of activation function to this hidden layer value
#4 each of these activation values will be multiplied by the weight values
#5 each output node will add together all the (hidden * weight) values that are passed to it
#6 apply the activation function to each of these output values



forward propagation

the steps described on the previous page are a process known as forward propagation. take a moment to walk through these steps with the example model below. we will eventually be implementing this process in code, using matrix multiplication. it will be helpful for you to be familiar with the details of this process. (don't worry about what an "activation function" is just yet — we'll get there in a bit! just focus on the basic forward propagation steps for now)



where are we getting these values?

now that you've stepped through forward propagation, you're probably wondering where all those numbers came from.

the input values will vary depending on the problem you're solving, but in this case each input node will hold the color value of a single pixel (0-255). the weight values are a bit more interesting.

when we initialize our neural network, we assign all the weight values to random values. this might seem weird, but it will make sense pretty soon.

it's important to see that, as of now, with our random weight values this neural network doesn't do much of anything at all. it kind of sucks, to be honest. we give it a bunch of input data, it multiplies some random numbers together and spits out an nonsensical output value. that doesn't do what we want at all!

and you're correct! we need to **teach** our model. and to do that, we need some new terminology and techniques.

supervised learning

you might remember earlier that we discussed how neural networks are very good at approximating functions that we can't define very well ourselves. let's elaborate on that a bit.

the function we are trying to approximate is a digit-recognition function. we want the input to be an image, and the function will output the correct classification of that digit.

now, if we sat down and tried to write that function on our own it would be very difficult. we could try describing "lines" and "intersections" and "curves" in code, but it would be very complicated. then we'd have to describe what a "one" or an "eight" looks like. this process gets hairy and tedious pretty quickly.

instead, we are going to use a neural network which is a **supervised learning algorithm**.

that means that, rather than describing the function, we are going to help the algorithm "learn" by providing it with a bunch of examples of different digits and their correct classification. over time (with some mathematical calculations) the net will learn how to correctly classify handwritten digits.

cool! let's see how that works.

warning:

the next section is going to discuss partial derivatives. if you are unfamiliar with this concept (or need to brush up) there are great videos on the topic at khan academy. it's highly recommended that you take a moment to get comfortable with these concepts before moving on.

what's a derivative?
https://www.khanacademy.org/math/calculus-home/taking-derivatives-calc

partial derivatives:
https://www.khanacademy.org/math/multivariable-calculus/multivariable-derivatives

don't worry! you don't need to know too much about the specifics of *how* to calculate derivatives.
just know the basic concepts.
(the chain rule in particular will be useful to you later on!)

defining a cost function

so, we have all of these randomly generated weight values. what we would like to do now is to make those weight values *better* somehow, so we can get the output we want.

in order to do this, we're going to define a **cost function** (sometimes called a **loss function**). this function is going to compare what the neural network outputs (known as its **prediction**) to what the correct classification should be.

the function will return a high number when the error is large (i.e. the prediction is way off) and a low number when the error is small (i.e. the prediction is close or even correct).

we will do this by subtracting the predicted output from what the correct value should be:

```
cost = (predicted value - correct value)
```

there are many different variations of this function which manipulate this basic formula (ex. making sure the error maps to some form of a convex curve, which will become important in a bit).

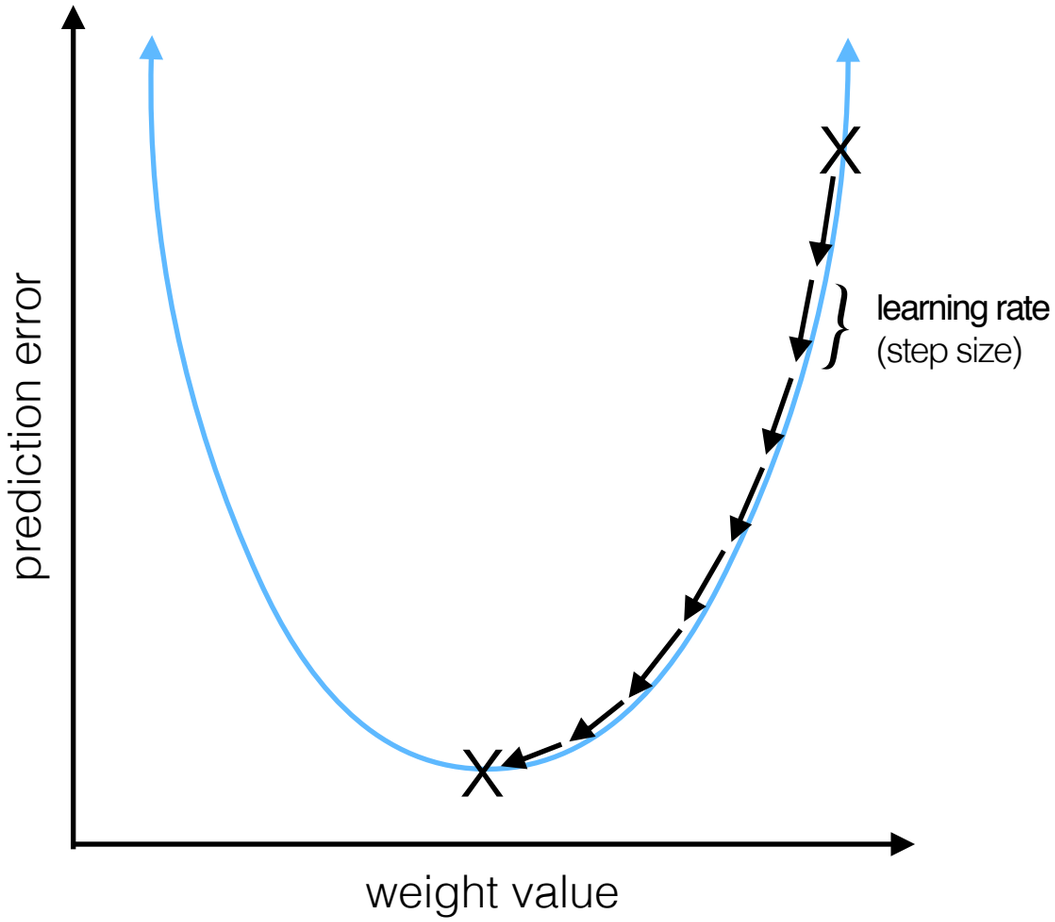
in particular, we want to make sure that both errors too-high and too-low will be represented as a high positive number, and absolutely correct values will return zero. our cost function should never return a negative value.

gradient descent & backpropagation

our cost function is a crucial element in finding the best weight values for our network (and therefore having a net that returns the values we want it to). what we are going to do now is attempt to minimize this cost function by using a method called **gradient descent**.

the idea behind gradient descent is actually pretty simple. every time we complete a forward pass, we will calculate the cost (or error) of the net's prediction. from there, we will calculate the partial derivates for each of the weigh values w.r.t. the cost function. these derivatives will tell us how making small changes to the weight values will affect the cost function.

what we want to do is make a small update to the weight values after each forward pass so that the cost function will decrease. our goal is to get to the point where our neural network reaches a very low average error for our samples. this is sometimes referred to as **minimizing loss**.



learning rate

while calculating the partial derivates (or the **gradient**) is a neat trick, it turns out that if we simply adjusted the weight values by the partial derivatives the algorithm would take way too long to actually improve any significant amount.

because of this, there's something called a learning rate. this is a constant value that we multiply by our gradient values in order to speed up the learning process.

increasing this value will increase the speed at which your prediction error will reach some minimum (and conversely, decreasing it will make it slower).

notice (on the graph) how at each pass of backpropagation, the weight values are updated so that the cost (or error) is minimized. the learning rate is essentially the "step size" that the algorithm will take on that path to the minimum error.

part IV: building a simple neural network

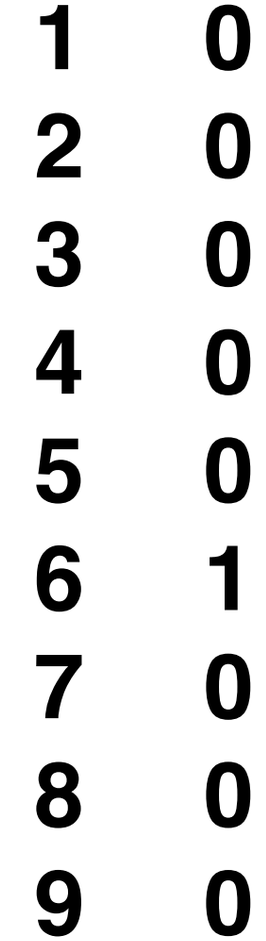
applying neural networks to MNIST

so now you hopefully have a good understanding of how a basic backpropagation neural network functions. but maybe you're wondering: how can we use this model to solve the MNIST classification problem?

well, our input is going to be a vector containing all the pixel information for a single image. we know that each MNIST sample is 28px * 28px, which means we will need 784 input nodes. that may sound like a lot, but remember: computers are really good at doing matrix multiplication very quickly.

we know that we want our output to be the classification of the sample [i.e. 0 - 9]. it turns out that an effective way to structure this output is to have ten output nodes (one for each digit), and do something called **one-hot encoding**.

this means that, rather than having the output for each node be [1] or [2] or [3], etc. we will have the correct output be a vector where 1 denotes belonging to a class, and 0 denotes NOT belonging to that class.



left: each of the possible classifications
right: a vector that would denote the classification '5'

let's get coding!

now that you have a basic idea of how a backpropagation neural network operates, and how we will apply it to the MNIST classification task: let's implement it in code!

the finished code is available at:
https://github.com/zabackka/nn_tutorials.git

use it to help guide your own construction, or even run it before starting just to get a feel for what the finished product will look like.

importing libraries

before we start, we need to import some different libraries. make sure that you include all of these libraries, or your code may not run correctly.

```
import numpy
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
from keras.utils import np_utils
```

don’t worry about what all these libraries are for just yet. just make sure you import them at the top of your code.

loading the data

our first task is going to be loading all of our MNIST data and storing it in the correct data structures. luckily for us, keras makes this super easy.

this imports the MNIST dataset and stores it into four different variables:

```
(X_train, y_train), (X_test, y_test) = mnist.load_data()
```

`X_train :`

3-D matrix that includes all the training samples [each index is a 2-D matrix that contains the pixel color values for a single sample]

`y_train :`

a vector that contains the correct classification for each training example [ex. index 0 stores the correct classification for training sample 0]

`X_test :`

this is structured just like the other training sample matrix, but will be used for testing rather than training

`y_train :`

this is structured just like the other correct classification vector, but corresponds to the testing (rather than training) data

a note about convention:

it is common for programmers to use X and Y when writing machine learning code. X usually denotes the input values, and the Y usually denotes the output (whether predicted or actual)

pixel value

sample #				
sample [0]	sample [0] pixel [0]	sample [0] pixel [1]	sample [0] pixel [2]	sample [0] pixel […]
sample [1]	sample [1] pixel [0]	sample [1] pixel [1]	sample [1] pixel [2]	sample [0] pixel […]
sample [2]	sample [2] pixel [0]	sample [2] pixel [1]	sample [2] pixel [2]	sample [0] pixel […]
sample [3]	sample [3] pixel [0]	sample [3] pixel [1]	sample [3] pixel [2]	sample [0] pixel […]
sample [4]	sample [4] pixel [0]	sample [4] pixel [1]	sample [4] pixel [2]	sample [0] pixel […]

restructuring input (X) data

in order to properly feed our network model, we need to restructure our data.

```
X_train = X_train.reshape(X_train.shape[0], n_pixels).astype('float32')
X_test = X_test.reshape(X_test.shape[0], n_pixels).astype('float32')
```

these two lines of code restructure the two X matrices so that each row is a single sample (therefore, there will be 60,000 rows — one row per sample) and each column represents a single pixel value (so there will be 784 columns — one column per pixel). we also cast the integer values of the input to floats, which will help us later on when we're doing mathematical calculations.

one “slot” for each classification

0 1 2 3 4 5 6 7 8 9

0 → [**1** 0 0 0 0 0 0 0 0 0]
1 → [0 **1** 0 0 0 0 0 0 0 0]
2 → [0 0 **1** 0 0 0 0 0 0 0]
3 → [0 0 0 **1** 0 0 0 0 0 0]
4 → [0 0 0 0 **1** 0 0 0 0 0]
5 → [0 0 0 0 0 **1** 0 0 0 0]
6 → [0 0 0 0 0 0 **1** 0 0 0]
7 → [0 0 0 0 0 0 0 **1** 0 0]
8 → [0 0 0 0 0 0 0 0 **1** 0]
9 → [0 0 0 0 0 0 0 0 0 **1**]

restructuring output (y) data

now, we are going to set up the one-hot encoding we discussed earlier:

```
y_train = np_utils.to_categorical(y_train)
y_test = np_utils.to_categorical(y_test)
```

this converts each single-digit classification from the ‘y’ data to a vector that uses one-hot encoding.

normalizing input values

a common machine learning practice is to normalize the input values. this usually speeds up learning and gets better NN results. we can do that easily with the following lines of code:

```
X_train = X_train / 255
X_test = X_test / 255
```

because the pixel values are between 0 and 255, we can divide by 255 to get a value between 0 and 1.

setting up the network

we will start by creating an instance of a sequential model in keras, which is essentially a linear stack of layers. we will build our input layer, our hidden layer, and our output layer using this model.

note that the two layers we add to the stack are dense, which in keras means a **fully-connected neural network layer.** we provide the function with the dimensions of each layer, and an activation function to use.

the first line of code configures a layer that take in our input values, does the necessary mathematical operations an applies the **rectifier activation function**. the second line will take the values from the previous layer, do the necessary weight multiplication and apply the **softmax activation function**. notice how the second line of code specifies that we want the number of output nodes to be equal to the number of classes we have. this will give us output that matches the one-hot encoded vectors we set up earlier.

```
// input -> hidden layer
model.add(Dense(n_pixels, input_dim=n_pixels, init='normal', activation='relu'))

// hidden -> output layer
model.add(Dense(n_classes, init='normal', activation='softmax'))

// compiling the model
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

to learn more about the keras sequential model (or for code troubleshooting), go here:
https://keras.io/getting-started/sequential-model-guide/

build & fit the model

so far, we haven't actually *done* any learning. all we've accomplished is setting up our network in a way so that it *can* learn. now, we're going to build our model and have it "fit" itself to our data (i.e. learn).

```
model.fit(X_train, y_train, validation_data=(X_test, y_test), nb_epoch=10, batch_size=200, verbose=2)
```

we provide the fit function with all the necessary information the network will need to learn. we give it the input data and output data we would like the model to use to train. we also give it the testing data to use for **validation** (meaning, it will pass the data forward through the model to see the error but it will not update the weights based on that error).

once this function completes, the model will have learned the from the data. now, we're going to test it to see how well it did.

test the model

after fitting the, we can test how well it can correctly classify new data it has never seen before. we will use our test data matrices for this:

```
error = model.evaluate(X_test, y_test, verbose=0)
```

the evaluate function will return a tuple: the percentage loss and the success (in decimal form) of the system on the test data.

to print the error as a percentage, use the following code:

```
print("error: %.2f%%" % (100-error[1]*100))
```

this is the number we care about: the error/success of our system. we want to keep our error low to so we have a system that should perform well on new data.

run your code!

now that you have your model built, run your code! see what error rate you get and how fast it runs.

try using different activation functions or different numbers of epochs. see if you can achieve a lower error rate, or get your code to run faster. if you need any help with modifying or running your code, there are plenty of resources online.

here's one in particular that's helpful:
http://swanintelligence.com/first-steps-with-neural-nets-in-keras.html

improving scalability & error rate

you probably achieved a pretty decent error rate running your code. but, the error rate can always be better (read: smaller) and you might have noticed that this code would not scale well to larger images.

it turns out that the network we built (called a **backpropagation neural network**) is only one many different kinds of networks and architectures.

next, we're going to improve our model by turning it into a CNN, or **convolutional neural network**. these networks are particularly good at processing images. let's take a look at the theory and concepts behind them to get a better idea of how they work!

part V: image classification with CNNs

how images are different

one of the main motivations behind creating a neural network designed to process images is the *size* of most images. our example with the MNIST data worked well because the images were relatively small (only 28px * 28px). however, most images we might want to process are not that small. some images are as big as 5000px * 5000px (that's *25 million* input nodes, all with multiple weight calculations!)

so, we needed a better way to process images. the CNN model is based on the idea of relative importance and local connectivity. simply put, we're not going to have all our layers be fully-connected (like our previous model). we know that the pixels closer to each other are more related than those far away.

we will capitalize on this aspect of images and build some different kinds of layers to boost our speed and lower our error rate. let's check it out!

receptive fields

so, we know that the pixels closest to each other are the most closely related to each other. because of this, we're going to create something called a **receptive field**. this is going to be a smaller portion of the image, and will get its own hidden layer neuron.

all of these hidden layer neurons are going to hold the same weight values, however. the goal of this architecture is to create a filter that will look for some feature, regardless of position or scale in the image.

in keras, we can specify the size of the receptive field as well as how many filters to use. we can do this by creating the following layer:

```
model.add(Convolution2D(32, 5, 5, input_shape=(1, 28, 28), activation='relu'))
```

this code creates 32 convolution filters, and has a 5px * 5px receptive field. we also specify the input shape (a 28px * 28px image, with only one color value [as opposed to RGB]).

this model is better suited for images because not only does it take into account the relative importance of a pixel in relation to space (via the receptive field) but also reduces the amount of connections/weights that need calculating (which makes this model scale better).

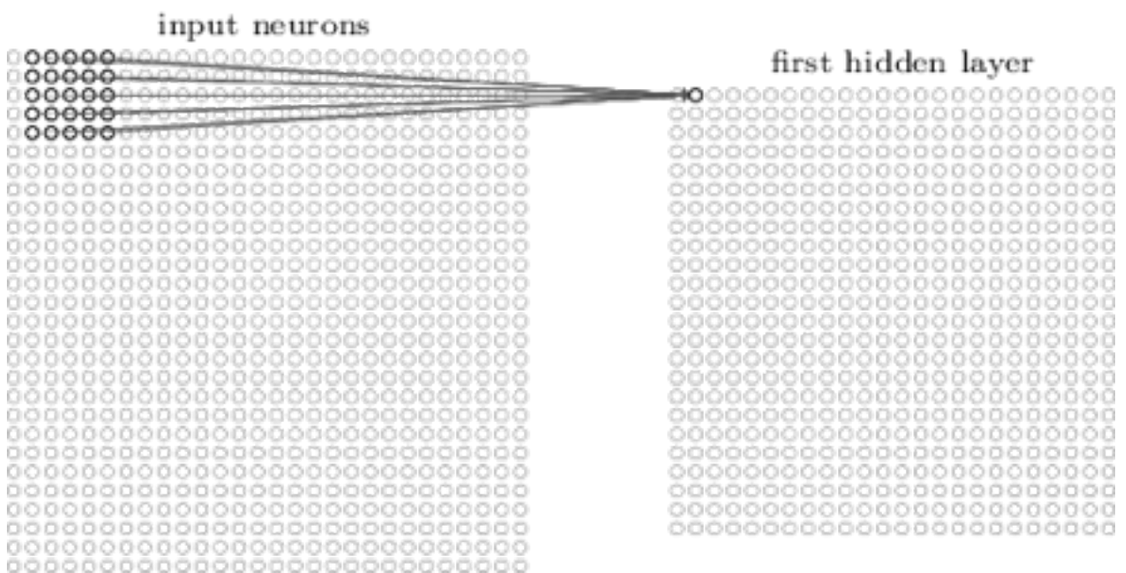


image courtesy of http://neuralnetworksanddeeplearning.com/chap6.html

max-pooling

now that we have our input nodes reduced to a subset of values (each representing the predicted value, or importance, of that receptive field), we are going to further reduce the amount of values we have with **max-pooling**.

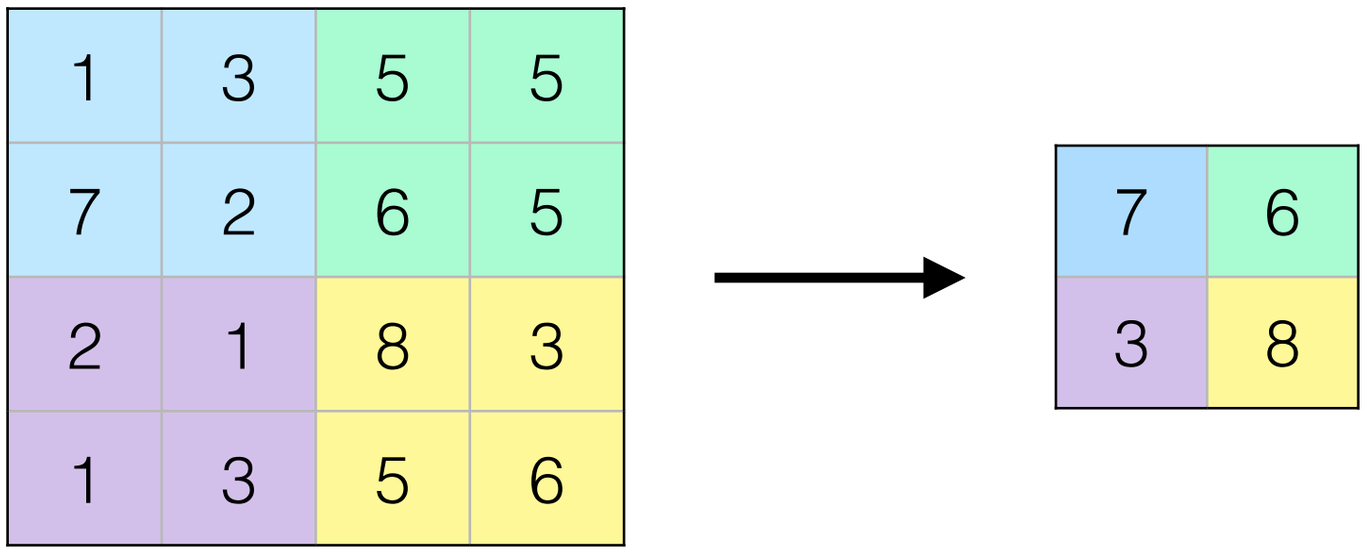
this will take a subset of our convolutional filter values, and only choose the largest value. you can see how this works with the diagram below.

this allows us to further reduce the amount of computations we need to perform, and also helps to not **overfit the data.**

the following code will create a max-pool layer:

```
model.add(MaxPooling2D(pool_size=(2, 2)))
```

max-pooling with a 2px * 2px pool size



dropout & flattening

next, to further prevent overfitting we are going to add a dropout layer. this simply drops (or excludes) some percentage of max-pool output from our final calculations. this may seem strange, but remember: images are incredibly complex, and our net is just looking at pixel values. we want to reduce our input down to as little as possible, so our net can focus on the most important bits.

we're also going to flatten our data before pushing it through the final layer. the output values of our max-pool layer are currently stored in a 2-D matrix, and we need it in one long vector to feed into our final layer.

we can do both of those things with the following lines of code:

```
model.add(Dropout(0.2))

model.add(Flatten())
```

now we're finally ready for the final layers of our CNN. and they should be pretty familiar to you!

back to the basics

now that we've scaled down the amount of values we have to work with, we are going to feed those values into the fully-connected layers we created earlier in our basic neural network.

after all that convolution, pooling and dropout we are left with the 128 different values. we will input all of those into a fully connected layer. our output layer will look exactly the same as before, because we still want the net to classify the same digits!

the last few lines of code look like this:

```
model.add(Dense(128, activation='relu'))
model.add(Dense(num_classes, activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

these should look pretty familiar to you, because
they're almost identical to the code we created earlier!

run your code!

ok, we've built our convolutional neural network! go ahead and run you code, and see what it outputs!

you might notice that the CNN runs slower than the basic neural net on your machine. this is because, while CNNs scale much better to larger images there are more mathematical calculations to perform. so, it might take ten minutes or so for you code to run. that's ok!

just have patience, and check out your results when it finishes. how did the CNN compare to our basic model? try tweaking different numbers, combining different layers, or changing the number of epochs and see what results you get. see if you can get that error number even lower!

as always, feel free to download and refer to my example code if you get stuck. look up resources online, and be adventurous!

wrapping up

it's so exciting that you've completed the first part of your journey into the world of machine learning. this guide barely scratched the surface on all the theory, models and techniques that are out there.

hopefully, you will take this as a jumping off point and dive head first into more machine learning!

here's a few free resources you might look into if you'd like to start now:

http://deeplearning.net/tutorial/lenet.html
http://neuralnetworksanddeeplearning.com/
https://www.youtube.com/watch?v=bxe2T-V8XRs

keep trying, keep exploring and keep learning!

cheers!
— katie

the code provided in this book was heavily adapted from
Deep Learning with Python: Develop Deep Learning Models on Theano and Tensorflow Using Keras
by Jason Brownlee

you can view more of his work & publications by visiting
http://machinelearningmastery.com/

thanks for reading!