chapter 1: using neural nets to recognize handwritten digits

- Until 2006 we didn't know how to train neural networks to surpass more traditional approaches, except for a few specialized problems. What changed in 2006 was the discovery of techniques for learning in so-called deep neural networks. (also, but not mentioned - availability of data and computing power is an oft cited reason although maybe not exactly in 2006 - but in recent years).
- this book is both about the underlying principles and about building a system that works => not a survey of all ideas in neural nets but we build a small neural net library that actually works!
- Michael Nielsen thinks we will learn best by building something, either our own project or working thru the problems in the book. His advice: Do all the exercises and a few of the problems.
- Neural networks learn from "sample" inputs and outputs, called training data. NNs get better with more "good" training data and better designed networks. We'll see what both of these mean.
- We're going to work with the sample problem of recognizing hand written digits - hits sweet spot between too easy and too difficult and is suitable to introduce concepts in deep learning after we're done talking about basic neural nets.
- in this chapter, we'll talk perceptrons and sigmoid neurons, stochastic gradient descent as the standard learning algorithm and an intuition for deep neural nets.
- The function used in the neuron is called the activation function.
- A perceptron (as opposed to sigmoid neuron) takes several binary inputs, $x_1, x_2..x_n$ and produces a single binary output. The perceptron's output, 0 or 1, is determined by whether the weighted sum $\sum_i w_i x_i$ is less than or greater than some threshold value. This is basically a linear combination of weights and inputs.
- By varying the weights and the threshold, we can get different models of decision-making from the perceptron.
- Each layer of perceptrons feeds its output as input into the next layer by connections. Each layer is making more complex, sophisticated, and more abstract decisions.
- Notationally, we can write the linear combination as a dot product, and bring the threshold to the same side of the equation. Threshold, bought to the same side, is called bias. So the equation is now $w^T x + b$
  - where w and x are vectors and b is the scalar bias. If this is > 0, then output is 1, else output is 0.
- Bigger the bias, easier it is to achieve an output of 1, obviously.
- Note how the output of a perceptron is a binary decision (vote) without any notion of magnitude of yes/no.
- We can build simple logical functions using perceptrons, or a network of them. For example, NAND OR and AND are all possible.
- In fact, the `NAND` example shows that we can use perceptrons to compute simple logical functions. In fact, we can use networks of perceptrons to compute any logical function at all. The reason is that the `NAND` gate is universal for computation, that is, we can build any computation up out of `NAND` gates. Adders can be built out of this. Convince yourself of this.
- Just like we could combine NAND gates to make an adder, we can construct an equivalent perceptron network to add two numbers.

- It is conventional to draw an extra layer of inputs, the input layer, that shows the inputs, as part of the neural net. These not real perceptrons that output computations. They just output the input.
- We are looking for perceptrons such that a small change in weight of one of the perceptrons causes a small change in output of the network, because that property would enable us to iteratively make small changes to the weights to get desired final output across a range of inputs. The problem is the linear model with bias doesn't have this property, so we can't easily train the network this way. That's why we use sigmoid neurons.
    - *Prove that they don't have this property:* Intuitively, this is because lots of small changes to a weight don't cause any change to that output and hence no change to output at all. Even if it causes a change to the output of that perceptron, it is not very likely to cause a change to the output of the perceptron it feeds into. Thus it is "hard to tune".
    - Prove that networks with sigmoid neurons have this property.
    - Are there other kinds of activation functions you can think of that might give us this property?
- Sigmoid neurons are similar to perceptrons, but modified so that small changes in their weights and bias cause only a small change in the network's output. That's the crucial fact which will allow a network of sigmoid neurons to learn.
- $sigmoid(z) = \frac{1}{(1+e^{-z})}$. Try replacing e with numbers from 1 to very big numbers. See how the shape of the curve changes from gradual to more binary-like.
    - e is used in part because it is great when differentiated. Leads to simpler algebra.
- To put it all a little more explicitly, the output of a sigmoid neuron with inputs x1,x2,...x1,x2,..., weights w1,w2,...w1,w2,..., and bias b is $\frac{1}{(1+e^{w^T x+b})}$
- Incidentally, σ (sigmoid function) is sometimes called the logistic function, and this new class of neurons called logistic neurons. It's useful to remember this terminology, since these terms are used by many people working with neural nets.
- It's only when $w^T x + b$ is of modest (not too positive or too negative) size that there's much deviation from the perceptron model. Very large or very small values lead to 0's and 1's, just as with perceptrons.
    - Doesn't this mean that if the initial weights are such that w⋅x+b is very large or very small, we'll run into the same problem as with perceptrons? The only difference is we will see a small difference in output, not zero difference. Should we initialize appropriately so we don't run into this problem?
- We can think of sigmoid as the smoothed out version of the perceptron.
- $\Delta O_k \approx \sum \frac{\partial O_k}{\partial w_j)} * \Delta w_j + \frac{\partial O_k}{\partial w_j} * \Delta b$
- Notice that for small changes Δw_j, Δoutput is a linear function of Δw_j. This linearity makes it easy to choose small changes in the weights and biases to achieve any desired small change in the output. So while sigmoid neurons have much of the same qualitative behavior as perceptrons, they make it much easier to figure out how changing the weights and biases will change the output.
- The main reason the activation function matters is because, of course, what does it actually compute and does that help us estimate stuff but also because can we have small changes in weights that lead to small changes in outputs, thus enabling us to train networks. Note how the sigmoid is better than the perceptron in this regard, even though they are kinda similar in other ways.
- There are three parts in a neural net: input layer, hidden layer(s) and output layer.
- The design of input and output layers is often straightforward. Hidden layer design is an art.
- Somewhat confusingly, and for historical reasons, multiple layer networks are sometimes called multilayer perceptrons or MLPs, despite being made up of sigmoid neurons, not perceptrons.
- networks where the output from one layer is used as input to the next layer are called feedforward neural networks. there are other models of artificial neural networks in which feedback loops are possible. These models are called recurrent neural nets.

- The idea in these models is to have neurons which fire for some limited duration of time, before becoming quiescent. That firing can stimulate other neurons, which may fire a little while later, also for a limited duration. That causes still more neurons to fire, and so over time we get a cascade of neurons firing. Loops don't cause problems in such a model, since a neuron's output only affects its input at some later time, not instantaneously.
- Recurrent neural nets have been less influential than feedforward networks, in part because the learning algorithms for recurrent nets are (at least when Michael wrote this book) less powerful.
  - why does backprop not work (well?) for RNNs?
- Onehot outputs: One design is to have as many output neurons as the number of possible labels (10 for digit classification). This “onehot” style design empirically works better than, maybe encoding the output as 4 bits.
  - Design a net where these 10 neurons are then connected to a 4 neuron output and see how that does.
- One reason for why this might be is that the neuron gets a stronger indication of what the answer is. Its either harder to get to an assignment of weights, or maybe such an assignment doesn't exist where a output neuron would only light up if it was a 1 bit (if not using one-hot encoding). My guess is more likely that its hard to get to such an assignment even though such assignments probably exist for many use cases.
- MNIST dataset is scans of tens of thousands of handwritten digits, along with their correct classification.
- *x* usually represents an input vector. In this case, its an image so its a m x n vector of numbers. Notice how we just call it a vector. Its implied that its a multidimensional matrix. y will be the output vector. In this case, of size 10.
- *T* here is the transpose operation, turning a row vector into an ordinary (column) vector.
- What we'd like is an algorithm which lets us find weights and biases so that the output from the network approximates *y*(*x*) for all training inputs *x*.
  - Among all the assignments of weights and biases that work well for training data, which is the right assignment that works well outside the training data? How do we answer this question for neural networks so that they generalize well?
- cost function == loss function == objective function. they all mean the same thing.
- If the cost function is framed as a sum of errors (or some measure that is related to number of training examples), it has to be normalized by the number of training examples, so that error isn't bigger simply cuz the training set is bigger.
- $C(w,b) = \frac{1}{2n}\sum ||y(x) - a||^2$ is the quadratic cost function, or mean squared error function. We'll minimize this using an algorithm called gradient descent.
- [possible question you might have] Michael says *“For the most part, making small changes to the weights and biases won't cause any change at all in the number of training images classified correctly. That makes it difficult to figure out how to change the weights and biases to get improved performance. If we instead use a smooth cost function like the quadratic cost it turns out to be easy to figure out how to make small changes in the weights and biases so as to get an improvement in the cost.”*
  - [question] How the hell is the quadratic cost function smooth? The cost function basically still measures the number of incorrect classifications.
  - answer: Note that the output isn't a vector with a single 1 and the rest 0's. The output is a vector of activations of the output layer, which is a bunch of floating point numbers. Eventually, we'd like for this to be so that one of them is close to a 1 and the rest are all very close to 0's, but that may not be the case when we start training.

- Note that designing a neural net system involves choice of network architecture, choice of activation function, choice of cost function and choice of dataset - these are the big ones. Gradient descent doesn't help us make these choices. Instead given a choice of architecture, cost function, activation function and dataset, gradient descent helps us find a good assignments of weights and biases for this set of choices.
- The problem here is to basically find the values for *w* and *b* such that the cost function has the least value. We could use calculus to find extrema but note that with larger networks, we'll have networks whose outputs depend on billions of weights in a complicated way! So the calculus way is infeasible.
- Recall the definition of differentiability of a multivariable function. from that definition, we will locally linearize a function in a small neighborhood of its gradient.
- From the definition of differentiability, we can estimate change by linear approximation:
  - $\Delta f = \sum_i (\frac{\partial f}{\partial x_i} * \Delta x_i)$
  - Or, written as dot products, $\Delta C \approx \nabla C \cdot \Delta v$.
  - A big assumption here is that the function has local linearity → which follows from differentiability in a small enough neighborhood.
- ∇C (gradient vector) is fixed at a given spot. To choose Δv so that ΔC is negative, we can set Δv = -η • ∇C. This way, we can just choose η and we are guaranteed that ΔC is negative (for appropriately small η).
- η is often varied so that Equation (9) remains a good approximation, but the algorithm isn't too slow.
- given these assumptions, several things can go wrong in making gradient descent work and hence it always doesn't work. when it does work, it works well. we'll revisit this in later chapters.
- It can be proved that the choice of Δv which minimizes ∇C·Δv is Δv = −η∇C, where η = ϵ/‖∇C‖is determined by the size constraint ‖Δv‖= ϵ. So gradient descent can be viewed as a way of taking small steps in the direction which does the most to immediately decrease C.
  - To see why the most decrease is in the direction of the gradient, refer to the notes on multivariable calculus.
- the idea of gradient approach is a popular way to train neural nets, although alternatives to gradient descent is a popular area of research.
- The cost function is a function of weights *w,* bias *b* and input *x*. C is averaged over all the inputs, but the weights and biases are same for all the inputs (at a given state during learning). So when we write the weight change equation:
  - w → w′ = w − η • (∂C/∂w), the partial derivative is different for different x's.
  - to compute the gradient ∇C we need to compute the gradients separately for each training input, and then average them. this takes a lot of time when the input training data is really big and learning goes slower.
- with *stochastic gradient descent* the idea is to estimate the gradient ∇C by computing ∇Cx for a small sample of randomly chosen training inputs. the chosen inputs to estimate gradient are called the mini-batch. this makes gradient descent and hence the learning process go faster.
- we pick a different set of *m* every time. the idea is to make use of all the training data. we go thru the training set in mini-batches until we exhaust all the training data. then we start over again. each such iteration is called an *epoch.*
  - this could, in theory, result in some iterations where the cost function does not decrease, since the average may not be a good approximation of overall average.
- "*Of course, the estimate won't be perfect - there will be statistical fluctuations - but it doesn't need to be perfect: all we really care about is moving in a general direction that will help decrease C, and that means we don't need an exact computation of the gradient. In practice, stochastic gradient descent is a*

commonly used and powerful technique for learning in neural networks, and it's the basis for most of the learning techniques we'll develop in this book."

- The approach of using all the training data in each iteration is called batch gradient descent (as opposed to stochastic gradient descent). in practice, batch gradient descent is simply too expensive and the stochastic approach helps us go down the hill anyway, so its a lot faster and a lot more practical.
- An extreme version of the stochastic approach is to use just one input at a time. this is called online learning or incremental learning.
- Its definitely not abnormal to find it hard to visualize higher dimensions. Its hard for everybody! There are no supermathematicians that can. They just use other techniques to understand behavior of high dimensional functions. Some techniques here: http://mathoverflow.net/questions/25983/intuitive-crutches-for-higher-dimensional-thinking.
- Hyper parameters of a neural network are parameters that describe the network and not the weights and biases, which are more just parameters. hyper parameters are things like the learning rate, etc.
- The weights from one layer of a neural net to the next can be represented as a matrix. Each inner list of the matrix represents the weights from the outgoing neurons of the previous layer to a single neuron in the current layer.
- The activation of neurons of a layer are computed by this equation:
- $a' = \sigma(wa+b)$
- Applying the sigmoid function to all elements of a vector, as done above is called vectorizing the function.
- Look at the network python class here and understand everything except the backpropagation() method (and the helpers it uses). https://github.com/mnielsen/neural-networks-and-deep-learning/blob/master/src/network.py
- running the above linked network with 30 epochs and mini batch size of 10 gives us about 94% accuracy. michael nielsen saw that accuracy go up when the number of hidden layer neurons was increased from 30 to 100.
- however, other readers got different results, some considerably worse. Using the techniques introduced in chapter 3 will greatly reduce the variation in performance across different training runs for our networks.
- its quite hard to debug neural nets. luckily michael nielsen got a good rate by starting at a good set of hyperparameters: learning rate, hidden layer neuron count, mini batch size, epochs and the architecture itself. we would wonder what is wrong and have no idea. maybe training data is bad/less?
- *You need to learn that art of debugging in order to get good results from neural networks. More generally, we need to develop heuristics for choosing good hyper-parameters and a good architecture. We'll discuss all these at length through the book, including how I chose the hyper-parameters above.* => Sounds super exciting!!!
- You can come up with some of your own rules like darkness of image and its correlation with digit. While these techniques seem to be less accurate, they are also less robust and rely on assumptions in the data.
- Simple SVMs that comes out of the box with scikit-learn got a 94% score. Pretty good! The current best MNIST solver are neural nets though.
- Usually, when programming we believe that solving a complicated problem like recognizing the MNIST digits requires a sophisticated algorithm.
- But really, even the current best approach to this problem is simple algorithm + lots of good training data. so the complexity is all learnt, not hand coded.

- In some sense, the moral of both our results and those in more sophisticated papers, is that for some problems:
- sophisticated algorithm $<<$ simple learning algorithm + good training data.
- Notice the emphasis on some.
- Each “higher” layer of a neural net might be answering questions that are increasingly complicated. (by complicated, I mean more abstract concepts and less closer to the pixels). eg. what's the pixel color here => does that look like the eyeball => is there an eye here => is there a face here, etc. Might be.
- *The end result is a network which breaks down a very complicated question - does this image show a face or not - into very simple questions answerable at the level of single pixels.*
- Networks with this kind of many-layer structure - two or more hidden layers - are called *deep neural networks*.
- Deep neural nets since 2006 vs neural networks earlier than that: “The reason, of course, is the ability of deep nets to build up a complex hierarchy of concepts.”
    - Delve more into this statement. What is some evidence for this? Is there evidence that suggests other reasons.