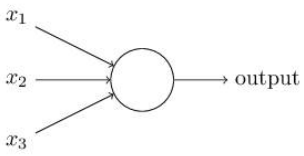
Neural Networks and Deep Learning- Michael Nielsen CH1 - original

- Intro
  - Learning algorithms
    - Suit for the secnarios where it's hard to make precise rules to make decisions
    - The idea is that the neural network uses the examples to automatically infer rules for recognizing handwritten digits.
    - Normally, the more the examples, the better NN learns
  - What to do
    - A handwriting digital number recognizer
    - we'll discuss how these ideas may be applied to other problems in computer vision, and also in speech, natural language processing, and other domains.
  - What are the key ideas about neural networks
    - two important types of artificial neuron
      - the perceptron
      - the sigmoid neuron
    - the standard learning algorithm for neural networks
      - stochastic gradient descent.
  - Throughout, I focus on explaining
    - why things are done the way they are,
    - and on building your neural networks intuition.
    - Amongst the payoffs, by the end of the chapter we'll be in position to understand what deep

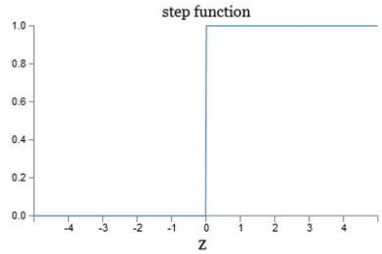
learning is, and why it matters.

- the perceptron
  - Structure



    - Inputs: searval binary
    - Output: one binary
    - Weight: real No., same count as inputs
    - Threshold: one real No.
  - How it works
    - Decesion rules·

$$\text{output} = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{ threshold} \\ 1 & \text{if } \sum_j w_j x_j > \text{ threshold} \end{cases} \quad (1)$$



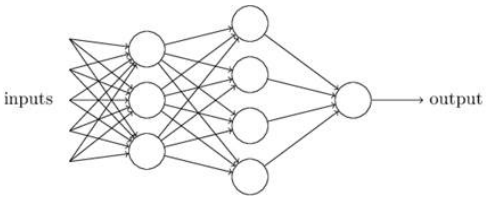
    - Simplified by b≡−threshold

$$\text{output} = \begin{cases} 0 & \text{if } w \cdot x + b \leq 0 \\ 1 & \text{if } w \cdot x + b > 0 \end{cases} \quad (2)$$

      - the bias as a measure of how easy it is to get the perceptron to output a 1. or in more biological terms, the bias is a measure of how easy it is to get the perceptron to fire.
    - A way you can think about the perceptron is that it's a device that makes decisions by weighing up evidence.

- By varying the weights and the threshold, we can get different models of decision-making.

- a complex network of perceptrons could make quite subtle decisions:



- The perceptrons in the later layer make decisions based on the outputs of the previous layer, which means the later layers can make decisions at a more complex and abstract level.

- Where perceptrons can be used
  - To make decisions
  - to compute the elementary logical functions , the underlying computation, e.g. AND, OR, and NAND
    - networks of perceptrons can compute any logical function
    - perceptrons are also universal for computation, because NAND gates are universal for computation

- What is unique with perceptron
  - we can devise learning algorithms which can automatically tune the weights and biases of a network of artificial neurons.
  - So that it can learn to solve problems

- the sigmoid neuron

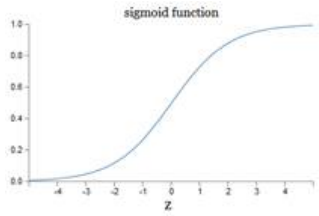
- Whay we need sigmoid instead of perceptron
  - What gives the ability to learn
    - small change in weight causes only a small corresponding change in the output. And then we can modify the weights and biases to get the desired output.
    - But in fact, a small change in the weights or bias of any single perceptron in the network can sometimes cause the output of that perceptron to completely flip, say from 0 to 1. besides, one change may influence the whole network in a complex way.
    - Here comes sigmoid neuron
- Intro
  - Sigmoid neurons are similar to perceptrons, but modified so that small changes in their weights and bias cause only a small change in their output.
- Structure
  - inputs, x1,x2,…. Real No.
  - Params
    - weights for each input, w1,w2,…,
    - an overall bias, b.
  - Output: $\sigma(w \cdot x+b)$ - Real No.·

$$\sigma(z) \equiv \frac{1}{1+e^{-z}}. \tag{3}$$

- Prop.
  - The smoothness of σ means that small changes Δwj in the weights and Δb in the bias will produce a small change Δoutput

$$\Delta \text{output} \approx \sum_j \frac{\partial \text{output}}{\partial w_j} \Delta w_j + \frac{\partial \text{output}}{\partial b} \Delta b, \tag{5}$$

  - Very suitable to learn
    - it's the shape of σ which really matters, and not its exact form



    - The smoothness, the continuity enables ability to learn.
    - The main thing that changes when we use a different activation function is that the particular values for the partial derivatives in Equation (5) change. Which influence the learning speed only.
  - The output can be regarded as a probability or confildence

- Exercise
  - No change
  - Output = 0 is the decision boundary.

- The architecture of neural networks
  - Basic
    - Input layer/neuron
    - Output layer/neuron
    - Hidden layer: means nothing more than "not an input or an output"
    - multiple layer networks
    - these are sometimes called multilayer perceptrons or MLPs, despite being made up of sigmoid neurons, not perceptrons.
  - Design of NN
    - Input/output layer: #input/output
    - Hidden layer requires art, normally guided by some heuristics for hidden layer design [research]
      - e.g. trade-off between #neurons & training time
    - Types of NN
      - feedforward neural networks
        - No loops, information is always fed forward, never fed back. Loops are not allowed
      - recurrent neural networks
        - feedback loops are possible
        - 按时间，one N fires another for a limited duration，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 of the less powerful learning algorithms (at least to date). But they're much closer in spirit to how our brains work. And it's possible that recurrent network can solve more complex problems

---

- A simple network to classify handwritten digits
    - Procedure
        - First, split the digits, i.e. breaking an image containing many digits into a sequence of

separate images

- classifying individual digits with NN

- Architecture
  - feedforward NN with 3 layers
    - Input layer: #input: 784=28×28 neurons
      - The input pixels are greyscale, with a value of 0.0 representing white, a value of 1.0representing black, and in between values representing gradually darkening shades of grey.
    - output layer = #output : 10 neurons (0,1,2,…,9)
      - the neuron that has the highest activation value determine the output
      - why 10 instead of 4(binary representation)
        - the performance of 10

is better than 4

- reasons
    - Output N makes the decision by weighing up evidence from the hidden layer of neurons. Here the evidence is the shape patch/feature one hidden neuron learns

- 10 neurons separates each shape of 10 digits, while 4 neurons can't, e.g. for 1 & 3 (001 011), they have not much common shape at all.

- Hidden layer: n=15 neurons
  - need to tune
    - learning algorithm
      - the standard learning algorithm for neural networks, known as stochastic gradient descent

---

Learning with gradient descent

- MNIST data set

    - 50,000 images to be used as training data.
    - 10,000 images to be used as validation data
        - to tune *hyper-parameters*
    - 10,000 images to be used as test data
        - taken from a *different* set of 250 people than the original training data
- values in layers
    - input: [0, 1.0] x 784 neurons
    - output: y(x)=(0,0,0,0,0,0,1,0,0,0)'
- To quantify how well we're achieving this goal we define a *cost function*
    - *mean squared error* or just *MSE*

$$C(w,b) \equiv \frac{1}{2n} \sum_x \|y(x) - a\|^2. \tag{6}$$

    - 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**. only after that will we examine the classification accuracy.
- *gradient descent - the learning algorithm*
    - the aim of our training algorithm
        - will be to minimize the cost C(w,b) as a function of the weights and biases, i.e. we want to find a set of weights and biases which make the cost as small as possible.
    - the basic idea

$$v \to v' = v - \eta \nabla C. \tag{11}$$

$$w_k \to w_k' = w_k - \eta \frac{\partial C}{\partial w_k} \tag{16}$$

$$b_l \to b_l' = b_l - \eta \frac{\partial C}{\partial b_l}. \tag{17}$$

        - Summing up, the way the gradient descent algorithm

works is to repeatedly compute the gradient ∇C, and then to move in the*opposite* direction, "falling down" the slope of the valley.

- a shortcoming of GD
  - each update requires to compute $\nabla C=1/n\sum_x \nabla C_x$. Unfortunately, when the number of training inputs is very large this can take a long time, and learning thus occurs slowly.
  - here comes *stochastic gradient descent* which can be used to speed up learning
- misc
  - finding alternatives to gradient descent is an active area of investigation

- *stochastic gradient descent*
  - *the core idea*
    - to estimate the gradient ∇C by computing $\nabla C_x$for a small sample of randomly chosen training inputs, called mini-batch .
    - Provided the sample size m is large enough we expect that the average value of the ∇C_{X_j} will be roughly equal to the average over all ∇C_x
  - so the update rule

$$w_k \to w_k' = w_k - \frac{\eta}{m}\sum_j \frac{\partial C_{X_j}}{\partial w_k} \tag{20}$$

$$b_l \to b_l' = b_l - \frac{\eta}{m}\sum_j \frac{\partial C_{X_j}}{\partial b_l}, \tag{21}$$

  - pro & con
    - it speeds up learning compared to GD

        - but 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.
    - special case
        - People sometimes omit the 1/n, summing over the costs of individual training examples instead of averaging. This is particularly useful when the total number of training examples isn't known in advance, e.g. real time data
    - *online*, *on-line*, or *incremental* learning - An extreme version of GD
        - update our weights and biases on each training input
    - misc
        - epoch: one span of the whole data set
        - we used an algebraic (rather than visual) representation to visualize high dimensions
            - this discussion of some of the techniques professional mathematicians use to think in high dimensions.

Implementing our network to classify digits

- MNIST data set
    - 50,000 images to be used as training data.
    - 10,000 images to be used as validation data

        - to tune *hyper-parameters*
    - 10,000 images to be used as test data
        - taken from a *different* set of 250 people than the original training data
- Numpy, for doing fast linear algebra.
- `Network` class used to initialize a `Network` object
    -
```
class Network(object):


    def __init__(self, sizes):
        self.num_layers = len(sizes)
        self.sizes = sizes
        self.biases = [np.random.randn(y, 1) for y in sizes[1:]]
        self.weights = [np.random.randn(y, x)
                        for x, y in zip(sizes[:-1], sizes[1:])]
```
        - Code explaination
            - the list `sizes` contains the number of neurons in the respective layers
                - `net = Network([2, 3, 1])`
            - `np.random.randn` function to generate Gaussian distributions with mean 0 and standard deviation 1.
            - Note that the first layer of neurons is an input layer, and omits to set any biases for those neurons, since biases are only ever used in computing the outputs from later layers.
            - `net.weights[1]` is a Numpy matrix

storing the weights connecting the second and third layers of neurons.

- w_{jk} is the weight for the connection between the k^{th} neuron in the second layer, and the j^{th} neuron in the third layer because it's easy to write code computing the output from a `Network` instance in this way.

- ```
  def sigmoid(z):
      return 1.0/(1.0+np.exp(-z))
  ```
  - vectorized form

- ```
  def feedforward(self, a):
      """Return the output of the network if "a" is input."""
      for b, w in zip(self.biases, self.weights):
          a = sigmoid(np.dot(w, a)+b)
      return a
  ```
  - It is assumed that the input a is an (n, 1)Numpy ndarray, not a (n,) vector. using an (n, 1)ndarray makes it particularly easy to modify the code to feedforward multiple inputs at once

- ```
  def SGD(self, training_data, epochs, mini_batch_size, eta,
          test_data=None):
      """Train the neural network using mini-batch stochastic
      gradient descent.  The "training_data" is a list of tuples
      "(x, y)" representing the training inputs and the desired
      outputs.  The other non-optional parameters are
      self-explanatory.  If "test_data" is provided then the
      network will be evaluated against the test data after each
      epoch, and partial progress printed out.  This is useful for
      tracking progress, but slows things down substantially."""
      if test_data: n_test = len(test_data)
      n = len(training_data)
      for j in xrange(epochs):
  ```
  ```

```
            random.shuffle(training_data)            #shuffled in each epoch
            mini_batches = [
                training_data[k:k+mini_batch_size]
                for k in xrange(0, n, mini_batch_size)]
            for mini_batch in mini_batches:
                self.update_mini_batch(mini_batch, eta)
            if test_data:
                print "Epoch {0}: {1} / {2}".format(
                    j, self.evaluate(test_data), n_test)
            else:
                print "Epoch {0} complete".format(j)
```

- code
  - The `training_data` is a list of tuples `(x, y)` representing the training inputs and corresponding desired outputs.
  - `eta`is the learning rate, η.
  - If the optional argument `test_data` is supplied, then the program will evaluate the network after each epoch of training, and print out partial progress. This is useful for tracking progress, but slows things down substantially.

```
def update_mini_batch(self, mini_batch, eta):
        """Update the network's weights and biases by applying
        gradient descent using backpropagation to a single mini batch.
        The "mini_batch" is a list of tuples "(x, y)", and "eta"
        is the learning rate."""
        nabla_b = [np.zeros(b.shape) for b in self.biases]
        nabla_w = [np.zeros(w.shape) for w in self.weights]
        for x, y in mini_batch:
            delta_nabla_b, delta_nabla_w = self.backprop(x, y)
            nabla_b = [nb+dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
            nabla_w = [nw+dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]
```

```
self.weights = [w-(eta/len(mini_batch))*nw
                for w, nw in zip(self.weights, nabla_w)]
self.biases = [b-(eta/len(mini_batch))*nb
               for b, nb in zip(self.biases, nabla_b)]
```

- Questions
  - How biases and weights are passed into the function
    - through the class varibale self.biases & self.weights
  - why not use element-wise matrix computation, e.g. nabla_b+delta_nabla_b
    - they're lists rather than array!!!

- Tune the hyperparameters
  - when the network doesn't work well for the first time, there're lots of possible causes. So we need to develop heuristics for choosing good hyper-parameters and a good architecture.
  - Detail is in later chapters
- Data structure used for NN
  - tuples and lists of Numpy `ndarray` objects
    - `training_data = zip(training_inputs, training_results)`
      - `training_inputs = [np.reshape(x, (784, 1)) for x in tr_d[0]]  # list of ndarray, each ndarray is an image`
      - `training_results #  a 10-ndarray with only one 1`

- validation_data = zip(validation_inputs, va_d[1])
    - va_d[1] # a list of integers
  - baseline tests to compare against
    - random guess 10%
    - darkness guess 22.5%
    - SVM 94.35%
    - human performance
  - Comparison
    - sophisticated algorithm ≤ simple learning algorithm + good training data.

- ## Toward deep learning
  - 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. It does this through a series of many layers, with early layers answering very simple and specific questions about the input image, and later layers building up a hierarchy of ever more complex and abstract concepts.

- Sub
  - what can be a prototype
    - As a prototype it hits a sweet spot: it's challenging - it's no small feat to recognize handwritten digits - but it's not so difficult as to require an extremely complicated

solution, or tremendous computational power.

- Furthermore, it's a great way to develop more advanced techniques, such as deep learning.

- the NAND gate is universal for computation, that is, we can build any computation up out of NAND gates
-

- why things are done the way they are, and on building your neural networks intuition.
-

- LAN
  - doing progressively more complex image processing.
  - Most people **effortlessly** recognize those digits as 504192. That ease is **deceptive.**
  - We carry in our heads a supercomputer, tuned by evolution over hundreds of millions of years, and superbly adapted to understand the visual world.
  - turn out to be not so simple to express algorithmically.
  - you quickly get lost in a **morass of exceptions and caveats** and special cases.
  - But this short program can recognize digits with an accuracy over 96 percent, **without human intervention.**
  - loathe bad weather
  - leftmost, bottommost
  - The computational universality of perceptrons is **simultaneously reassuring and disappointing.** It's reassuring because it tells us that networks of perceptrons can be as powerful as any other computing device. But it's also disappointing, because it makes it seem as though

perceptrons are merely a new type of NAND gate. That's **hardly big news!**

- Conventional vs. traditional
- This tuning happens in response to external stimuli, without direct intervention by a programmer.
- Radically vs. completely, totally
- Terrific
- the behaviour of the network on all the other images is likely to have completely changed in some hard-to-control way.
- the algebraic form of the sigmoid function turns out to be more of a technical detail than a true barrier to understanding.