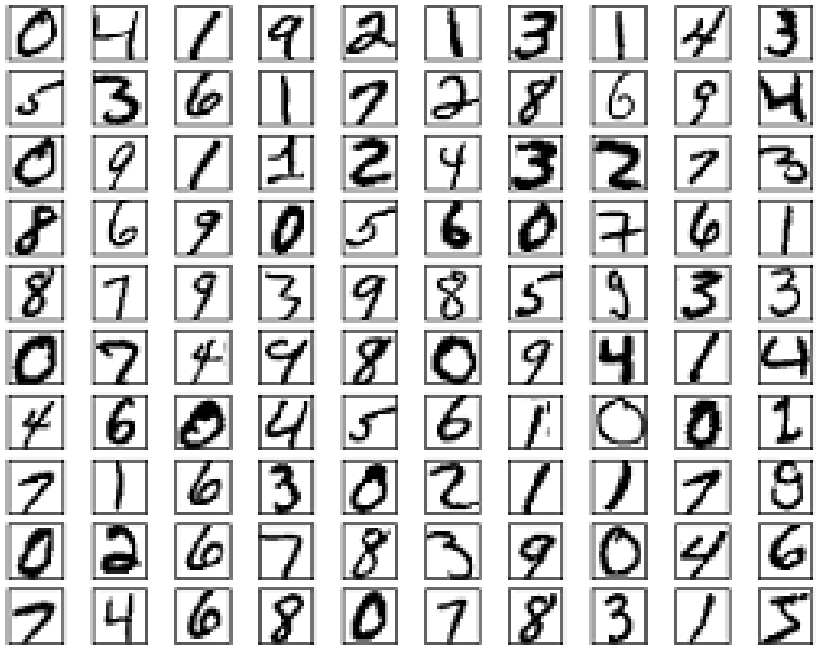
SciComp with Py

Artificial Neural Networks (ANNs): Part 1

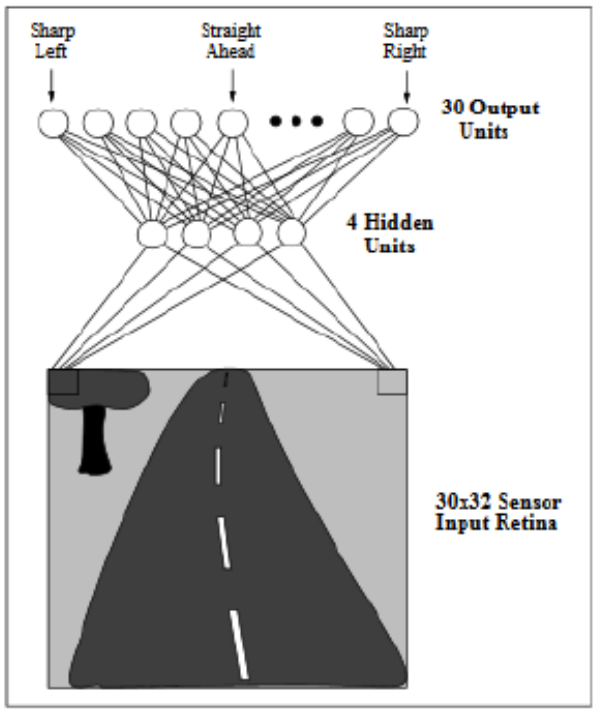
Vladimir Kulyukin
Department of Computer Science
Utah State University

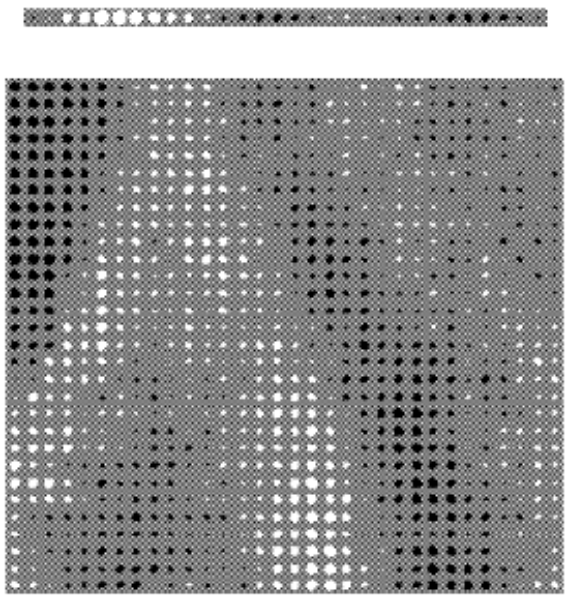
Outline

Motivation: Recognition of Handwritten Digits



Motivation: Self-Driving Cars





Outline

Brain's Visual Cortex

In each hemisphere of our brain, we have a primary visual cortex called V1.

V1 contains 140,000,000 neurons with tens of billions of connections between them.

Human vision involves not just V1 but also a series of visual cortices called V2, V3, V4, and V5, each of which does progressively more complex image processing.

Humans are very skillful at making sense of what their eyes show them, all of which is done unconsciously.

Biological Networks

Biological learning systems are built of very complex webs of interconnected neurons.

The human brain is estimated to contain a densely interconnected network of approximately 10^{11} neurons, each connected, on average, to 10^4 other neurons.

Neuron activity is excited/fired or inhibited through connections to other neurons.

Neuron Switching Speeds

The fastest human neuron switching time is approximately 10^{-3} seconds.

By comparison, the computer switching time is 10^{-10} seconds.

Humans make surprisingly complex decisions suprisingly fast: it takes 10^{-1} seconds to visually recognize one's mother.

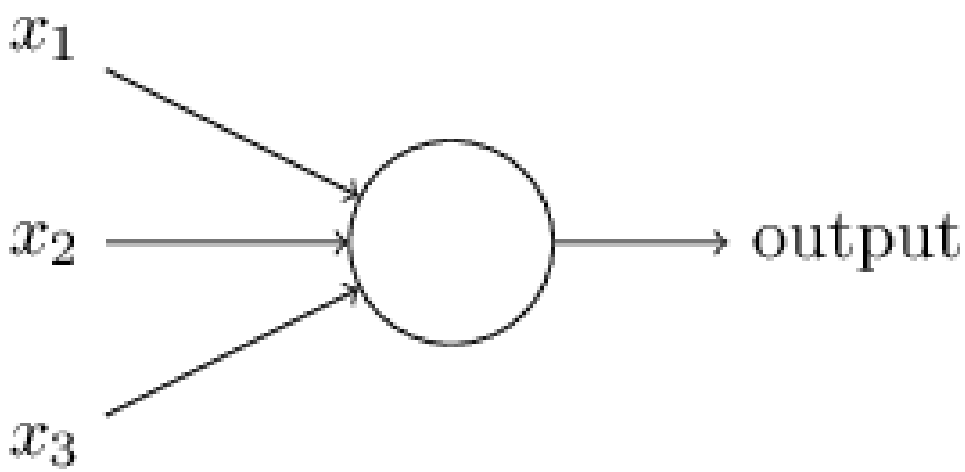
Some researchers speculate that information processing capabilities of biological neural systems are highly parallel and representations are distributed over many areas of the brain.

Outline

Perceptron

A perceptron is an artificial neuron.

How do perceptrons work? A perceptron takes several binary inputs x_1, x_2, ... x_n and produce a single binary output.



Perceptron's Output

Connections from the inputs to the perceptron are governed by weights w_1, w_2, ..., w_n.

Weights are real numbers expressing the importance of the respective inputs to the output.

The neuron's output, 0 or 1, is determined by whether the weighted sum $\sum_j w_j x_j$ is less than or greater than some threshold θ.

$$output = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \theta \\ 1 & \text{if } \sum_j w_j x_j > \theta. \end{cases}$$

Perceptron's Output Simplified

Let's simplify $\sum_j w_j x_j$ and write it as $w \cdot x$.

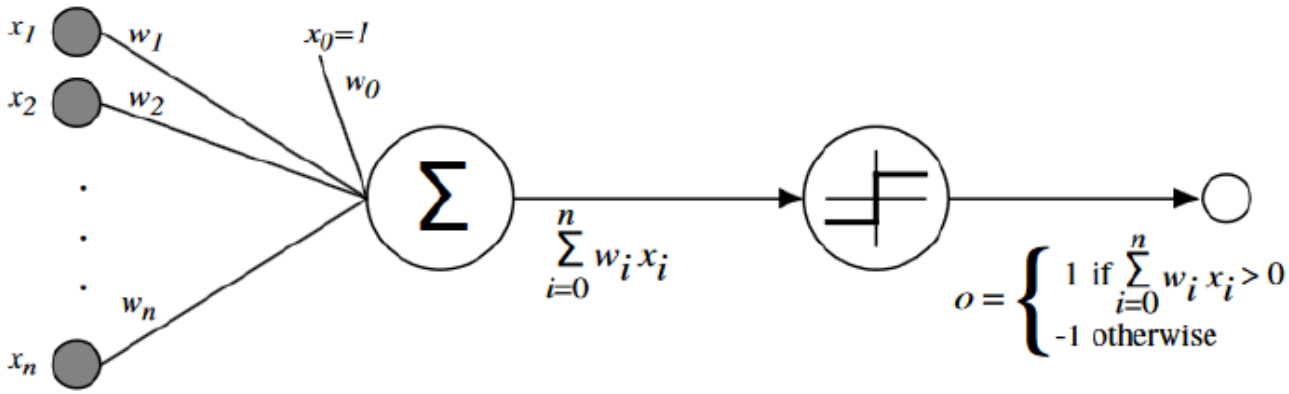
In other words, $\sum_j w_j x_j = w \cdot x$.

Another simplification is to move the threshold θ inside the inequalities and call it *basis* or b. If $b = -\theta$, then

$$output = \begin{cases} 0 & \text{if } \sum_j w_j x_j + b \leq 0 \\ 1 & \text{if } \sum_j w_j x_j + b > 0. \end{cases}$$

One can think of the bias b as a measure of how easy it is to get the perceptron to output 1. The bias is a measure of how easy it is to get the perceptron to *fire*: the more positive the bias, the easier for the perceptron to output 1 and vice versa.

Perceptron



$$o(x_1, \ldots, x_n) = \begin{cases} 1 & \text{if } w_0 + w_1 x_1 + \cdots + w_n x_n > 0 \\ -1 & \text{otherwise.} \end{cases}$$

Vector notation can also be used:

$$o(\vec{x}) = \begin{cases} 1 & \text{if } \vec{w} \cdot \vec{x} > 0 \\ -1 & \text{otherwise.} \end{cases}$$

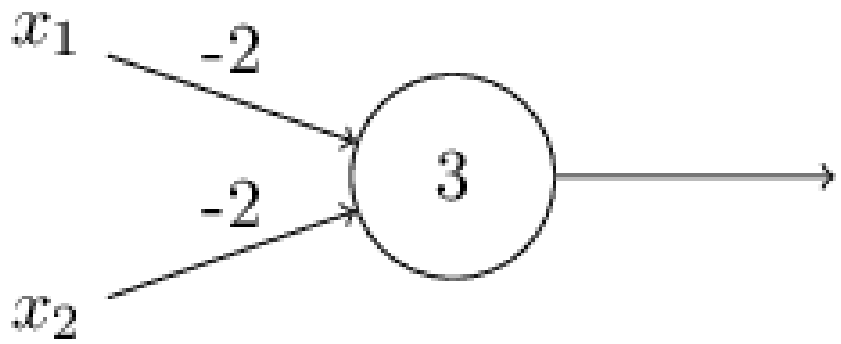
Another Notation for Perceptrons

$$o(\vec{x}) = sgn(\vec{w} \times \vec{x})$$

$$sgn(y) = \begin{cases} 1 & \text{if } y > 0 \\ -1 & \text{otherwise.} \end{cases}$$

Example: NAND

The perceptron below implements the NAND gate.



If $x_1 = 0$ and $x_2 = 0$, then the output is $-2 \cdot 0 + -2 \cdot 0 + 3 = 3$, so the output is 1.

If $x_1 = 0$ and $x_2 = 1$, then the output is $-2 \cdot 0 + -2 \cdot 1 + 3 = 1$, so the output is 1. Same output if $x_1 = 1$ and $x_2 = 0$.

If $x_1 = 1$ and $x_2 = 1$, then the output is $-2 \cdot 1 + -2 \cdot 1 + 3 = -1$, so the output is -1.

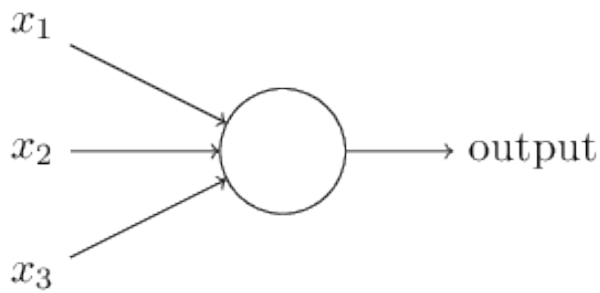
A Problem with Perceptrons

A small change in the weights or bias of a single perceptron in the network can sometimes cause the output of that perceptron to flip from 0 to 1.

This flip may cause the behavior of the rest of the network to change completely.

Sigmoid Neuron

A sigmoid neuron is also an artificial neuron like a perceptron, except its inputs x_1, x_2, ... x_n are real values between 0 and 1. The sigmoid neuron also has weights for each input (i.e., w_1, w_2, ..., w_n) and an overall bias b.



The output is not 0 or 1. Instead, it is $\sigma(w \cdot x + b)$, where σ is the sigmoid function defined as

$$\sigma(z) = \frac{1}{1 + e^{-z}},$$

where $e = \sum_{n=0}^{\infty} \frac{1}{n!} = \frac{1}{1} + \frac{1}{1} + \frac{1}{1 \cdot 2} + \frac{1}{1 \cdot 2 \cdot 3} + \ldots =$ 2.71828182845904523536028747135266249775724709369995.

Sigmoid Neuron's Output

The output of a sigmoid neuron with inputs the x_1, x_2, ... x_n, the weights w_1, w_2, ..., w_n, and the bias b is

$$\sigma(w \cdot x + b) = \frac{1}{1 + exp(-\sum_j w_j x_j - b)},$$

where $z = w \cdot x + b$.

Here is how you compute $exp(x) = e^x$ in numpy:

```
>>> import numpy as np
>>> np.exp(1)
2.7182818284590451
>>> np.exp(2)
7.3890560989306504
>>> np.exp(-1)
0.36787944117144233
```

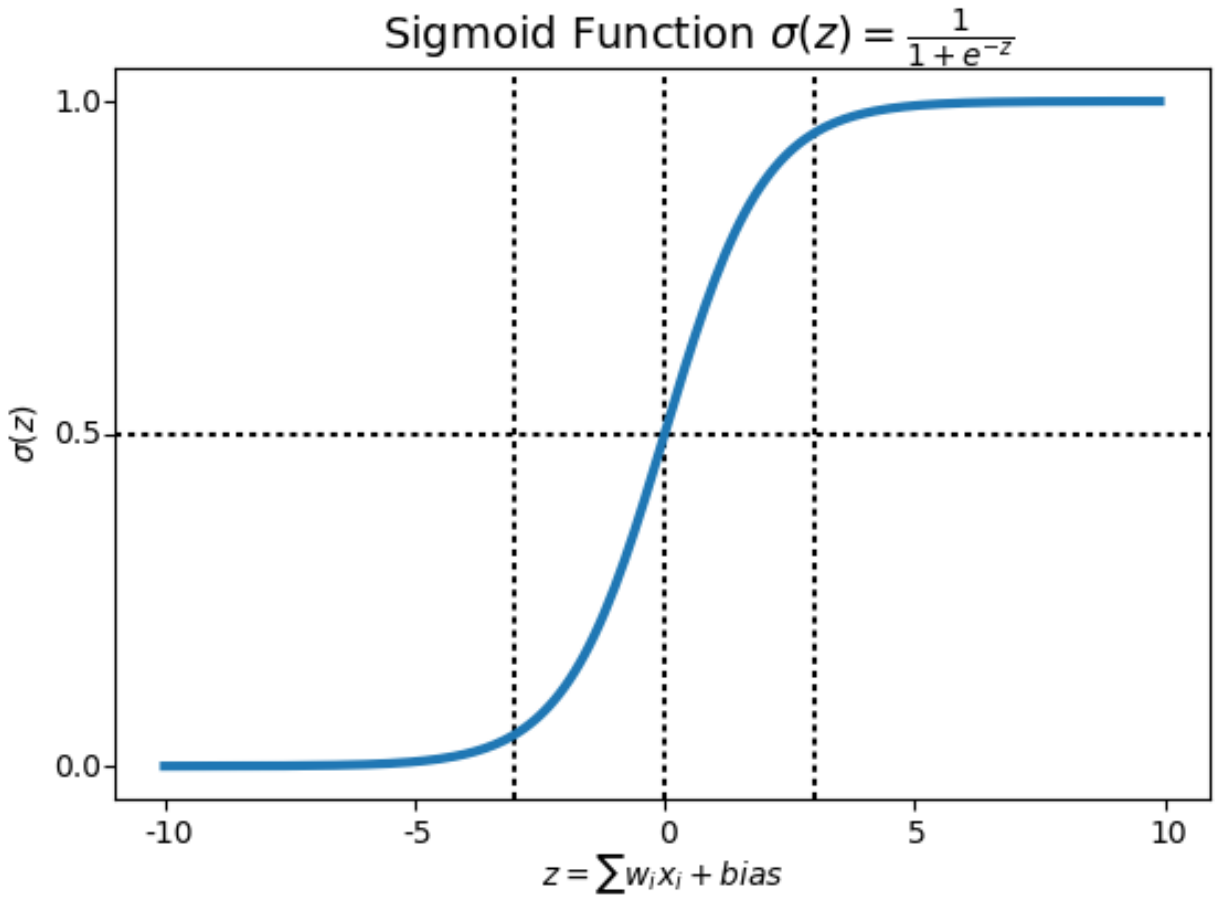
Sigmoid Neuron's Output

If $z = w \cdot x + b$ is a large positive number, then $e^{-z} \approx 0$ and $\sigma(z) \approx 1$.

If $z = w \cdot x + b$ is a large negative number, then $e^{-z} \approx 0$ and $\sigma(z) \approx 0$.

When $z = w \cdot x + b$ is a number of moderate size, then the sigmoid neuron's output is different from the perceptron's output.

Sigmoid Neuron's Output Function (aka Activation Function)



Interpreting Sigmoid Neuron's Output

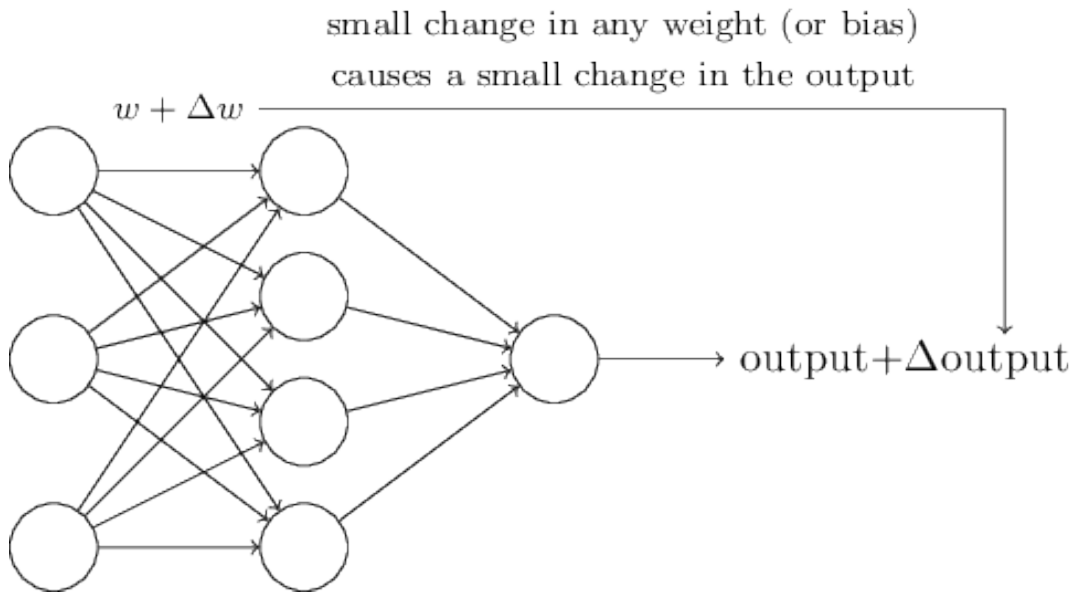
The main difference between the perceptron and the sigmoid neuron is in the output: the perceptron outputs either 0 or 1; the sigmoid neuron can output any real number between 0 and 1.

How do we interpret sigmoid outputs such as 0.5, 0.256, 0.9789, etc.? For example, if we are training our neural network to recognize digits and the output sigmoid neurons, when the network is given an image of 7, it outputs 0.7754. How do we interpret it?

This depends on a specific problem at hand. We can, for example, set up a convention that any output value below 0.5 is 0, and any output value that is at least 0.5 is 1. When you are desiging an ANN or using a third-party ANN, make sure that you know what this convention is.

What Do Sigmoid Neurons Buy Us?

The use of sigmoid neurons makes it much more likely that a small change in a weight/bias causes only a small change in output. We can use this fact in modify the weights/biases to get our network to behave more smoothly.



What Do Sigmoid Neurons Buy Us?

The smoothness of σ means that the small weight changes Δw_j and the bias Δb produce a small change in $\Delta output$ from any given neuron.

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

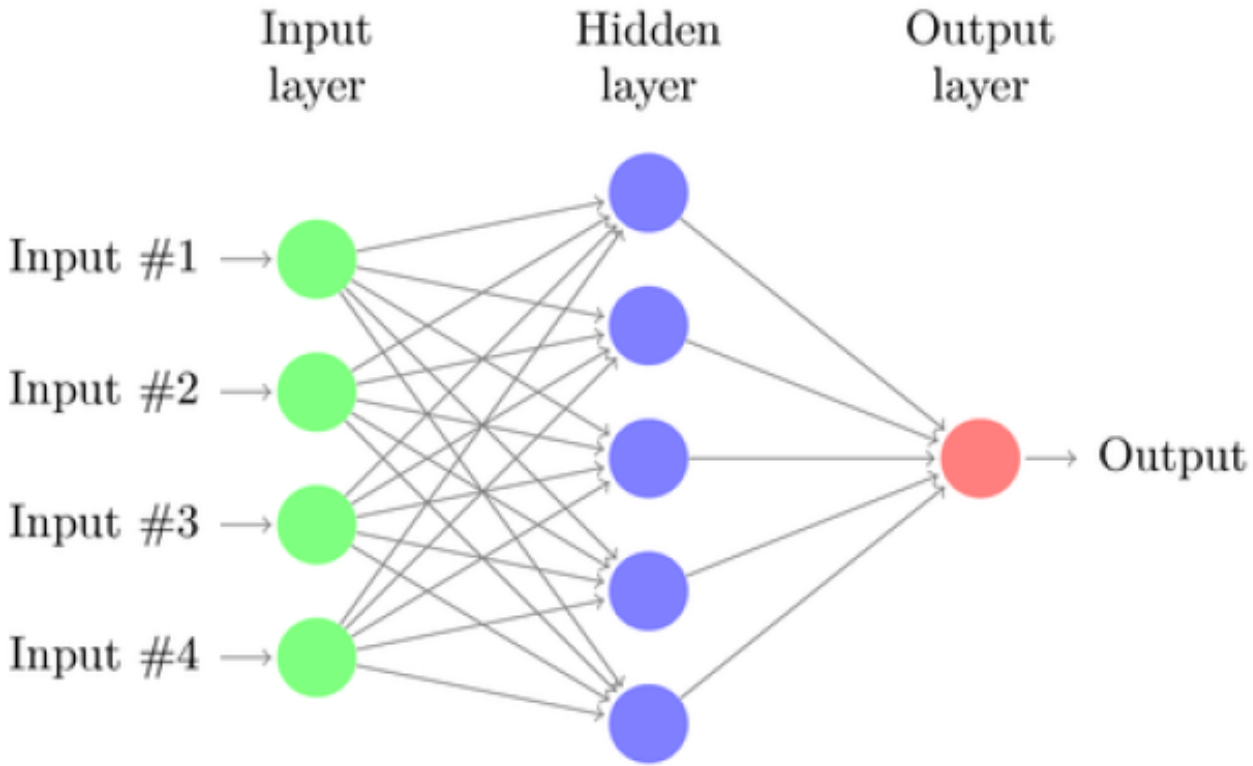
where the sum is over all the weights w_j and $\frac{\partial output}{\partial w_j}$ and $\frac{\partial output}{\partial b}$ are partial derivatives of the output with respect to w_j and b, respectively.

If you don't know calculus, don't worry! What this apporixmate equality tells us is that $\Delta output$ is a linear function of the change in weight Δw_j and the change in bias Δb.

Outline

Layers

Neural networks consists of layers. Each layer consists of neurons (either perceptrons or sigmoids). There can be arbitrarily many layers stacked on each other. The leftmost layer is called the *input* layer. The righmost layer is called the *output* layer. The layers in between the input layer and the output layer, if there are any, are called the *hidden* layers.



Synapses vs. Neurons

The links between neurons, sometimes called *synapses*, indicate which neurons input their outputs to which other neurons. These links have weights.

The job of a synapse is pretty simple: take the value from the input neuron, multiply the value by its weight, and output the result into the output neuron.

The job of a neuron is to take the inputs from the synapses and apply its activation function to them.

Parameters vs. Hyperparameters

The neural network differ from each other in terms of their layers, types of neurons, synapses, and activation functions. These are called *hyperparameters*. Hyperparameters cannot be changed.

The parameters of a neural network are weights and biases that can be manipulated and/or learned.

The neural networks also differ from each other by their learning procedures.

Outline

Designing Neural Networks

The design of input and output layers is often straightforward. For example, if you're classifying 32×32 grayscale images, then it makes sense to have $32 \times 32 = 1024$ input neurons, each of which is a number between 0 and 255. If you're classifying 64×64 grayscale images, you should have $64 \times 64 = 4,096$ neurons.

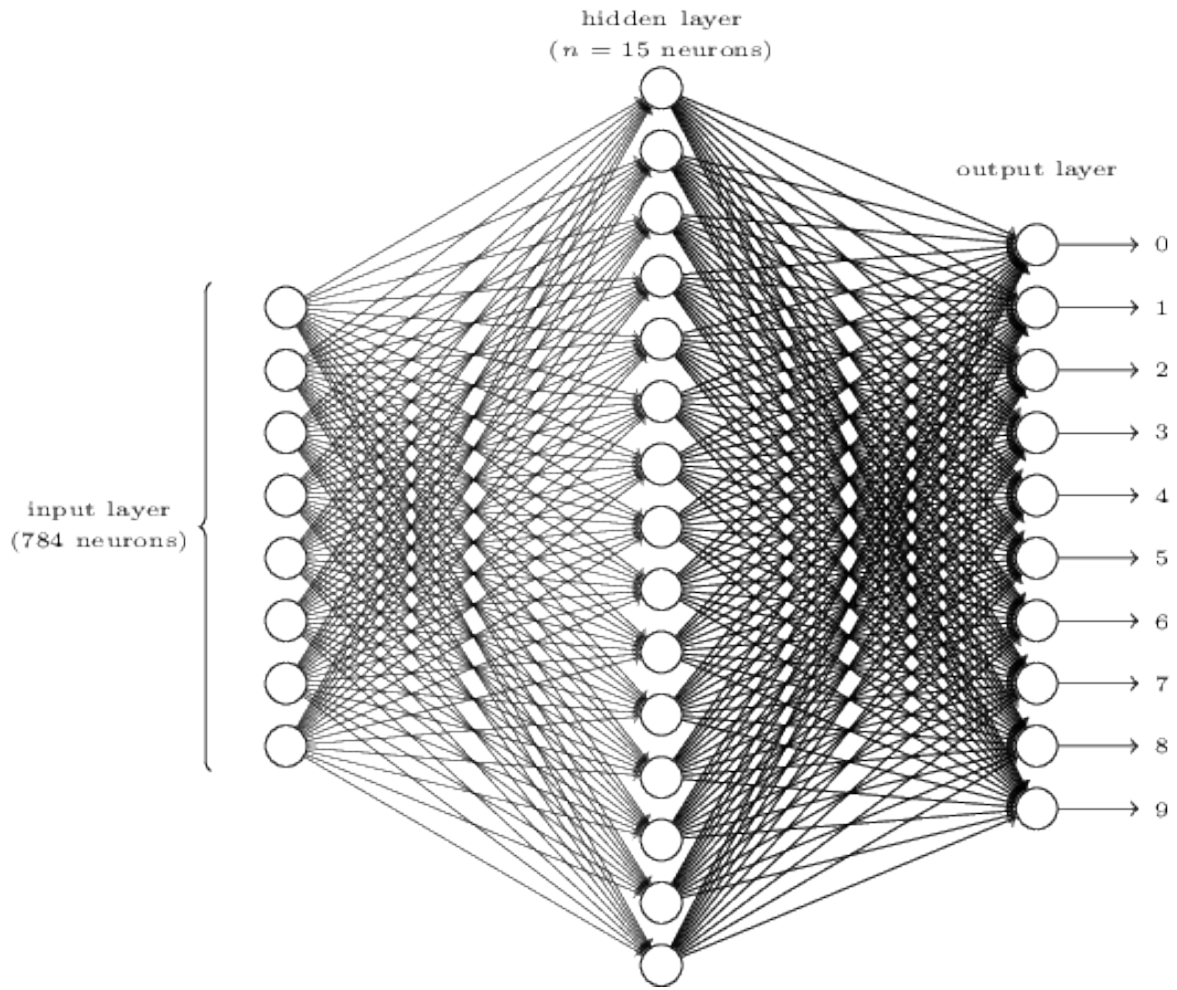
If you're recognizing images, it makes sense to have 10 output nodes - one for each digit. If you're recognizing the presence/absence of a car in an image - 1 or 2 output neurons are enough.

There are no accepted rules for designing hidden layers. There are some heuristics that ANN researchers use but none of them are universally accepted.

The term *deep learning* refers to neural networks with many hidden layers.

Example

Suppose that we want to design and train an ANN to recognize handwritten digits in 28×28 grayscale images. Then here is one possible ANN.



Building an ANN

Let's suppose that we want to predict how the years of study and years of job experience correlate with our salaries.

We've decided that we'll build a $2 \times 3 \times 1$ sigmoid network. In other words, 2 sigmoid neurons in the input layer (years of study and years of job experience), 3 sigmoid neurons in the hidden layers, and 1 sigmoid neuron in the output layer.

Let's suppose that we have lots and lots of 3-tuples (x_1, x_2, x_3), where x_1 is a person's years of study, x_2 is the person's years of experience, and x_3 is the person's salary.

Building an ANN

The input to the neural network will be (x_1, x_2) of each (x_1, x_2, x_3). Let's call it X.

The output to the neural network is $\widehat{Y}$.

For each $X = (x_1, x_2)$, we want to compare how far the output of our network, $\widehat{Y}$, is from x_3.

Building an ANN: Initializing Weights

There is one more question we need to answer before coding - how do we initialize weights?

We can use `numpy.random.rand(x, y)` that creates x, y numpy arrays of normally distributed random numbers with a mean of 0 and a variance of 1.

```
 >>> import numpy as np
 >>> np.random.randn(2, 2)
array([[ 0.35120796,  0.19578908],
       [-0.20131598,  0.06917832]])
 >>> np.random.rand(2, 3)
array([[ 0.92847923,  0.10751104,  0.64730707],
       [ 0.81078793,  0.52196304,  0.59270607]])
```

Building an ANN: Initializing Network

```
import numpy as np

class NeuralNet(object):
    def __init__(self):
        #HyperParameters
        self.inputLayerSize  = 2
        self.outputLayerSize = 1
        self.hiddenLayerSize = 3

        #Weights on Synapses
        self.W1 = np.random.randn(self.inputLayerSize,
                                  self.hiddenLayerSize)
        self.W2 = np.random.randn(self.hiddenLayerSize,
                                  self.outputLayerSize)
```

Building an ANN: the Sigmoid Activation Function

Here is the sigmoid function we defined a few slides ago:

$$\sigma(z) = \frac{1}{1 + e^{-z}}.$$

Here is the Python equivalent:

```
import numpy as np

class NeuralNet(object):
    def sigmoid(self, z):
        return 1/(1+np.exp(-z))
```

Building an ANN: From Input to Output

How do we compute $\widehat{Y}$ from $X = (x_1, x_2)$? This process of pushing the input through the network from the input to the output layer is called *feedforward*.

Feedforward in our $2 \times 3 \times 1$ network is defined by the following four equations:

1. $Z^{(2)} = XW^{(1)}$;
2. $a^{(2)} = f(Z^{(2)})$;
3. $Z^{(3)} = a^{(2)}W^{(2)}$;
4. $\widehat{Y} = f(Z^{(3)})$,

where $W^{(1)}$ is the matrix of weights on the synapses connecting layer 1 (input) to layer 2 (hidden), $f(z)$ is the activation function, and $W^{(3)}$ is the matrix of weight on the synapses connecting layer 2 (hidden) to layer 3 (output).

Building an ANN: Feedforward: Equation 1

Equation 1:

$$Z^{(2)} = XW^{(1)}.$$

Here is equation 1 in Python:

```
def forward(self, X):
    self.z2 = np.dot(X, self.W1) # eq. 1
```

Building an ANN: Feedforward: Equation 2

Equation 2:

$$a^{(2)} = f(Z^{(2)}).$$

Here is equation 2 in Python:

```
def forward(self, X):
    self.z2 = np.dot(X, self.W1)    # eq. 1
    self.a2 = self.sigmoid(self.z2) # eq. 2
```

Building an ANN: Feedforward: Equation 3

Equation 3:

$$Z^{(3)} = a^{(2)} W^{(2)}.$$

Here is equation 3 in Python:

```
def forward(self, X):
    self.z2 = np.dot(X, self.W1)        # eq. 1
    self.a2 = self.sigmoid(self.z2)     # eq. 2
    self.z3 = np.dot(self.a2, self.W2) # eq. 3
```

Building an ANN: Feedforward: Equation 4

Equation 4:

$$\widehat{Y} = f(Z^{(3)}).$$

Here is equation 4 in Python:

```
def forward(self, X):
    self.z2 = np.dot(X, self.W1)        # eq. 1
    self.a2 = self.sigmoid(self.z2)     # eq. 2
    self.z3 = np.dot(self.a2, self.W2) # eq. 3
    yHat = self.sigmoid(self.z3)        # eq. 4
    return yHat
```

Outline

Perceptron Training Rule

$$w_i \leftarrow w_i + \Delta w_i$$

$$\Delta w_i = \eta(t - o)x_i$$

where

- $t = c(\vec{x})$ is target value
- o is perceptron output
- η is small constant (e.g., 0.1) called *learning rate*

Gradient Descent

Consider a simple *linear unit*, where

$$o = w_0 + w_1 x_1 + \cdots + w_n x_n$$

We will learn w_i's that minimize the squared error

$$E[\vec{w}] \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

where D is set of training examples.

Gradient Descent Rule

Gradient:

$$\nabla E[\vec{w}] \equiv \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \cdots \frac{\partial E}{\partial w_n} \right]$$

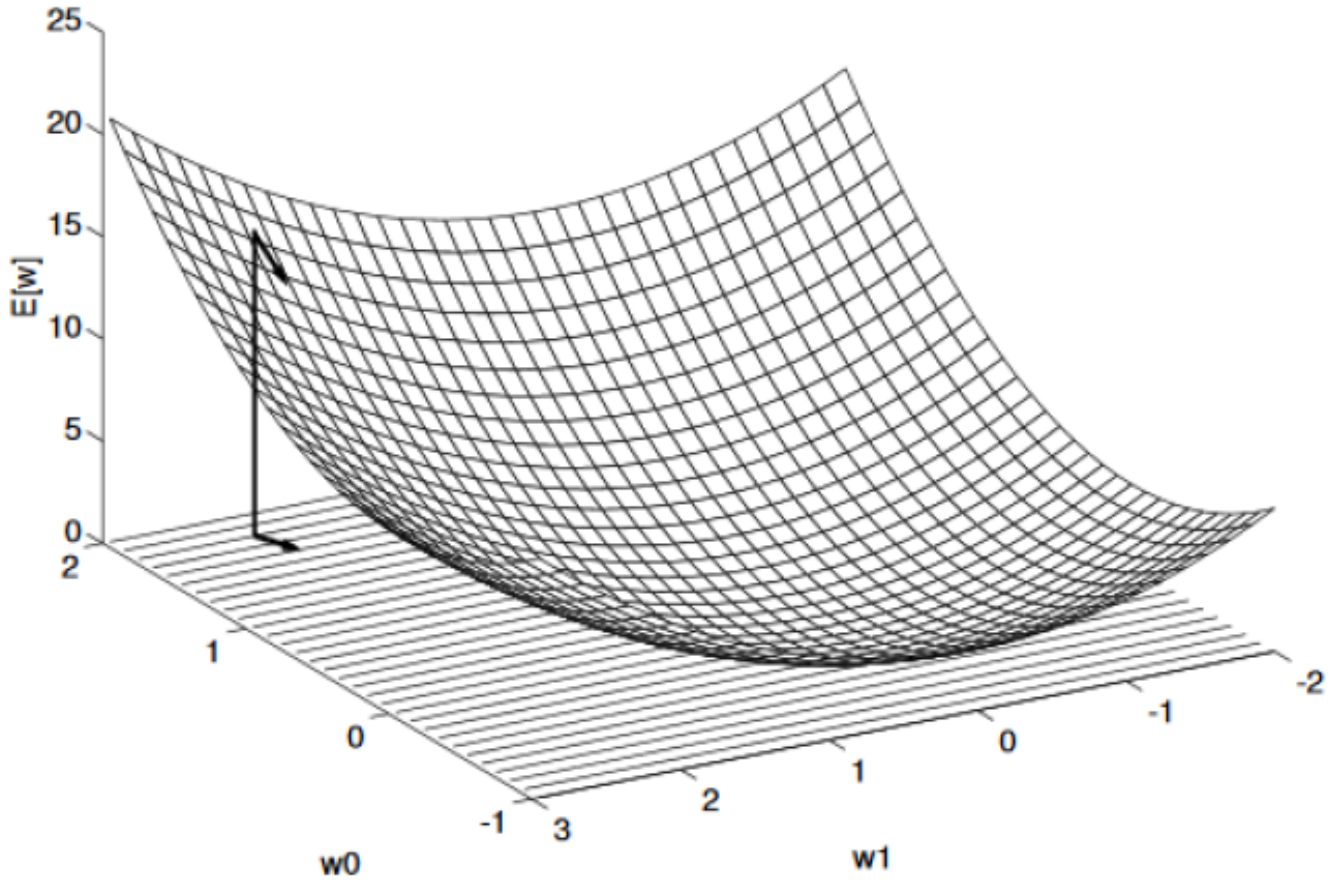
Training rule:

$$\Delta \vec{w} = -\eta \nabla E[\vec{w}]$$

i.e.,

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$

Hypothesis Search Space



Derivation of Gradient Descent

$$
\begin{aligned}
\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i}\frac{1}{2}\sum_d (t_d - o_d)^2 \\
&= \frac{1}{2}\sum_d \frac{\partial}{\partial w_i}(t_d - o_d)^2 \\
&= \frac{1}{2}\sum_d 2(t_d - o_d)\frac{\partial}{\partial w_i}(t_d - o_d) \\
&= \sum_d (t_d - o_d)\frac{\partial}{\partial w_i}(t_d - \vec{w}\cdot\vec{x_d}) \\
\frac{\partial E}{\partial w_i} &= \sum_d (t_d - o_d)(-x_{i,d})
\end{aligned}
$$

where $x_{i,d}$ is the input component x_i for training example d.

Derivation of Gradient Descent

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i} = \eta \sum_{d \in D} (t_d - o_d) x_{i,d}$$

Gradient Descent Algorithm

GRADIENT-DESCENT($training\_examples, \eta$)

Each training example is a pair of the form $\langle \vec{x}, t \rangle$, *where* $\vec{x}$ *is the vector of input values, and* t *is the target output value.* η *is the learning rate (e.g., .05).*

- Initialize each w_i to some small random value
- Until the termination condition is met, Do
  - Initialize each Δw_i to zero.
  - For each $\langle \vec{x}, t \rangle$ in *training_examples*, Do
    - Input the instance $\vec{x}$ to the unit and compute the output o
    - For each linear unit weight w_i, Do

$$\Delta w_i \leftarrow \Delta w_i + \eta(t - o)x_i$$

  - For each linear unit weight w_i, Do

$$w_i \leftarrow w_i + \Delta w_i$$

Incremental Gradient Descent Algorithm

Batch mode Gradient Descent:
Do until satisfied

1. Compute the gradient $\nabla E_D[\vec{w}]$
2. $\vec{w} \leftarrow \vec{w} - \eta \nabla E_D[\vec{w}]$

Incremental mode Gradient Descent:
Do until satisfied

- For each training example d in D
  1. Compute the gradient $\nabla E_d[\vec{w}]$
  2. $\vec{w} \leftarrow \vec{w} - \eta \nabla E_d[\vec{w}]$

$$E_D[\vec{w}] \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

$$E_d[\vec{w}] \equiv \frac{1}{2} (t_d - o_d)^2$$

Incremental Gradient Descent can approximate *Batch Gradient Descent* arbitrarily closely if η made small enough

Incremental Gradient Descent Training Rule

$$w_i = w_i + \eta(t - o)x_i$$

References

1. T.M. Mitchell. *Machine Learning*.
2. M. Neilsen. *Neural Networks and Deep Learning*.