SciComp with Py

Artificial Neural Networks (ANNs): Part 2

Vladimir Kulyukin
Department of Computer Science
Utah State University

Outline

Review

Designing and Building Neural Networks

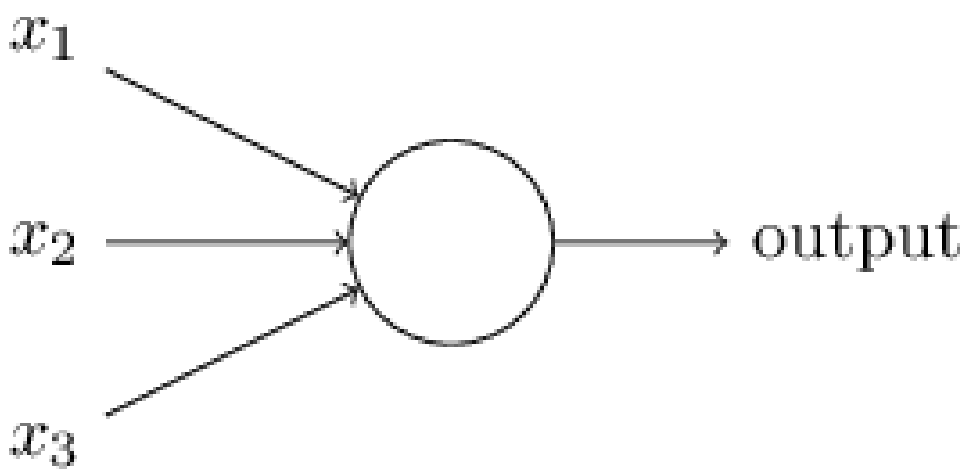
Training Neural Networks

Mathematics of Training

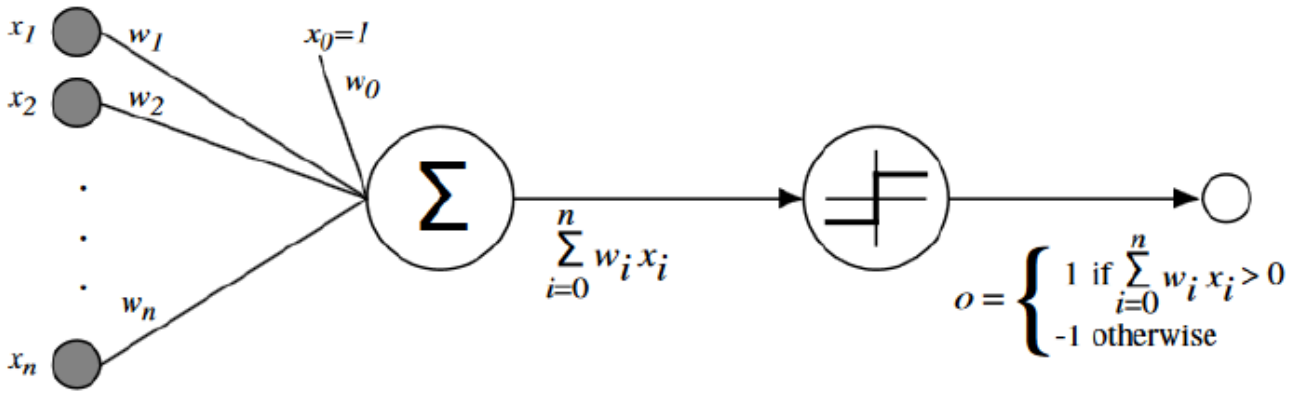
Perceptron

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



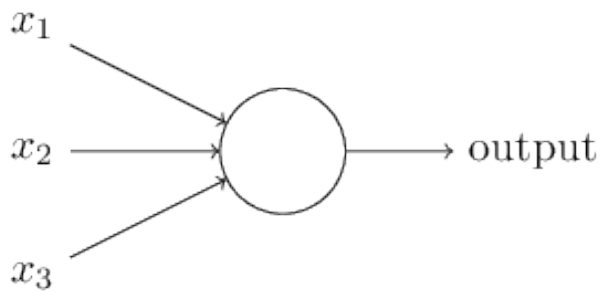
$$o(x_1, \ldots, x_n) = \begin{cases} 1 & \text{if } w_0 + w_1 x_1 + \cdots + w_n x_n > 0 \\ -1 & \text{otherwise.} \end{cases}$$

Vector notation can also be used:

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

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

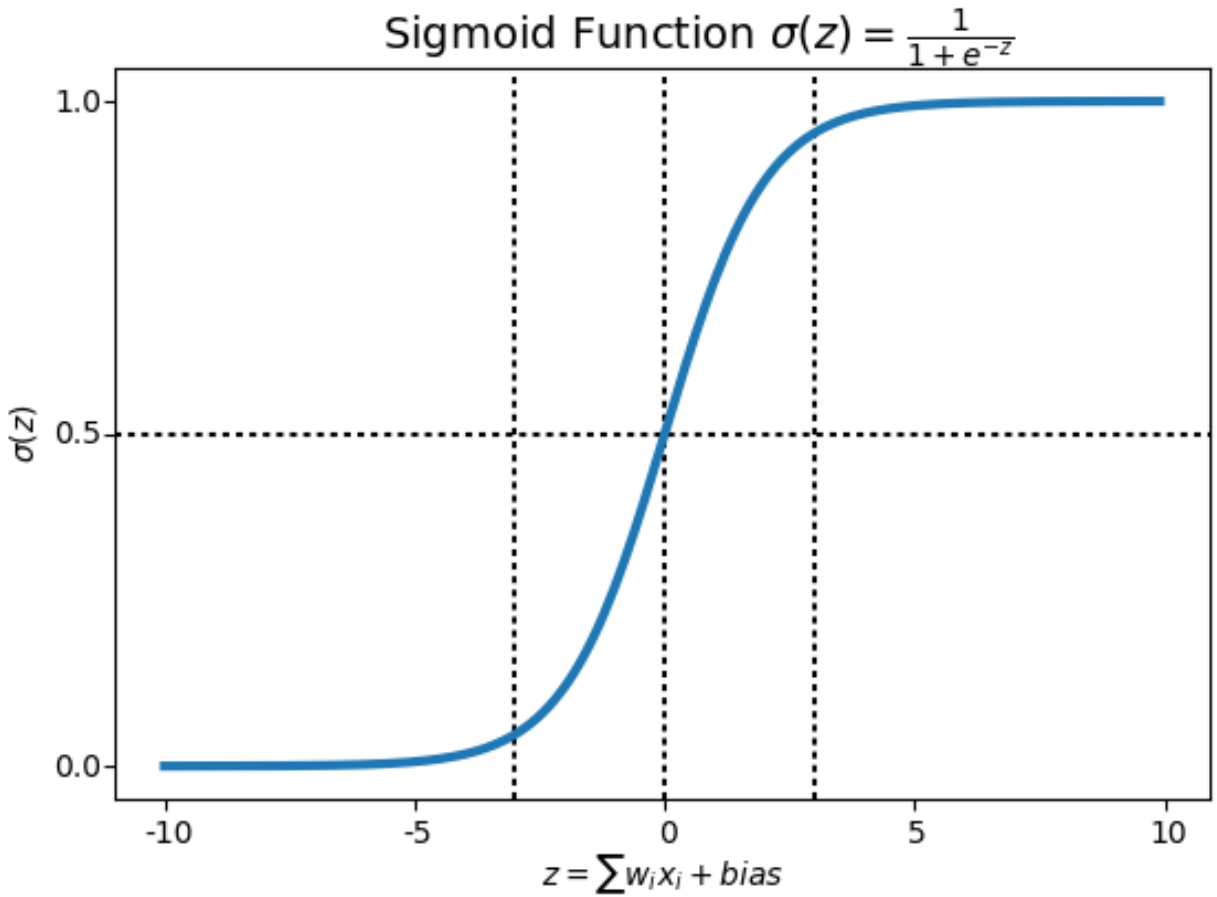
$$\sigma(w \cdot x + b) = \frac{1}{1 + exp(-\sum_j w_j x_j - b)},$$

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

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

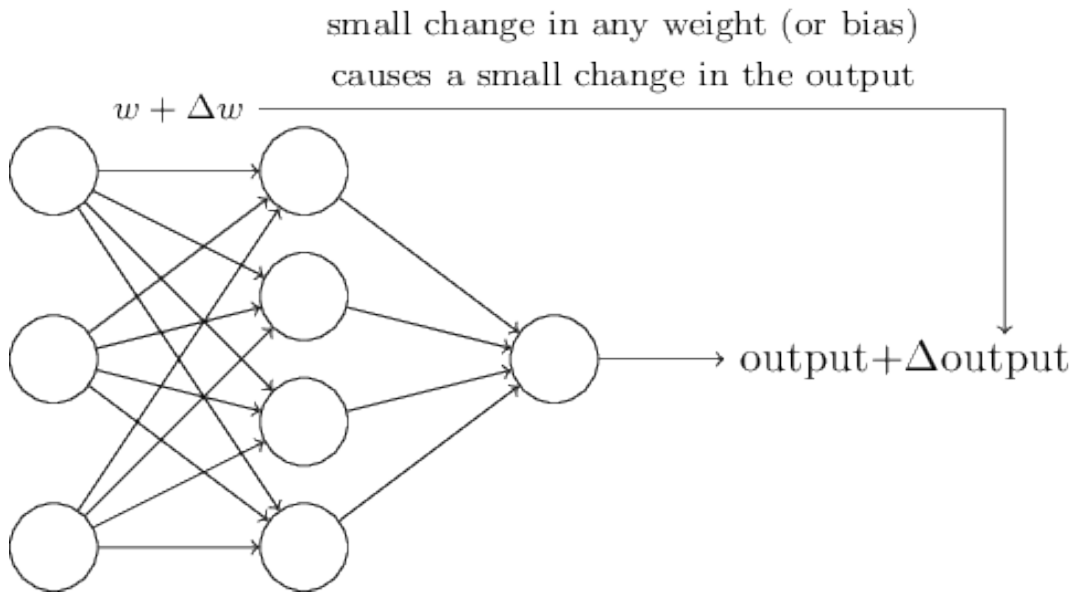
```
>>> import numpy as np
>>> np.exp(1)
2.7182818284590451
>>> np.exp(2)
7.3890560989306504
>>> np.exp(-1)
0.36787944117144233
```

Sigmoid Neuron's Output Function (aka Activation Function)



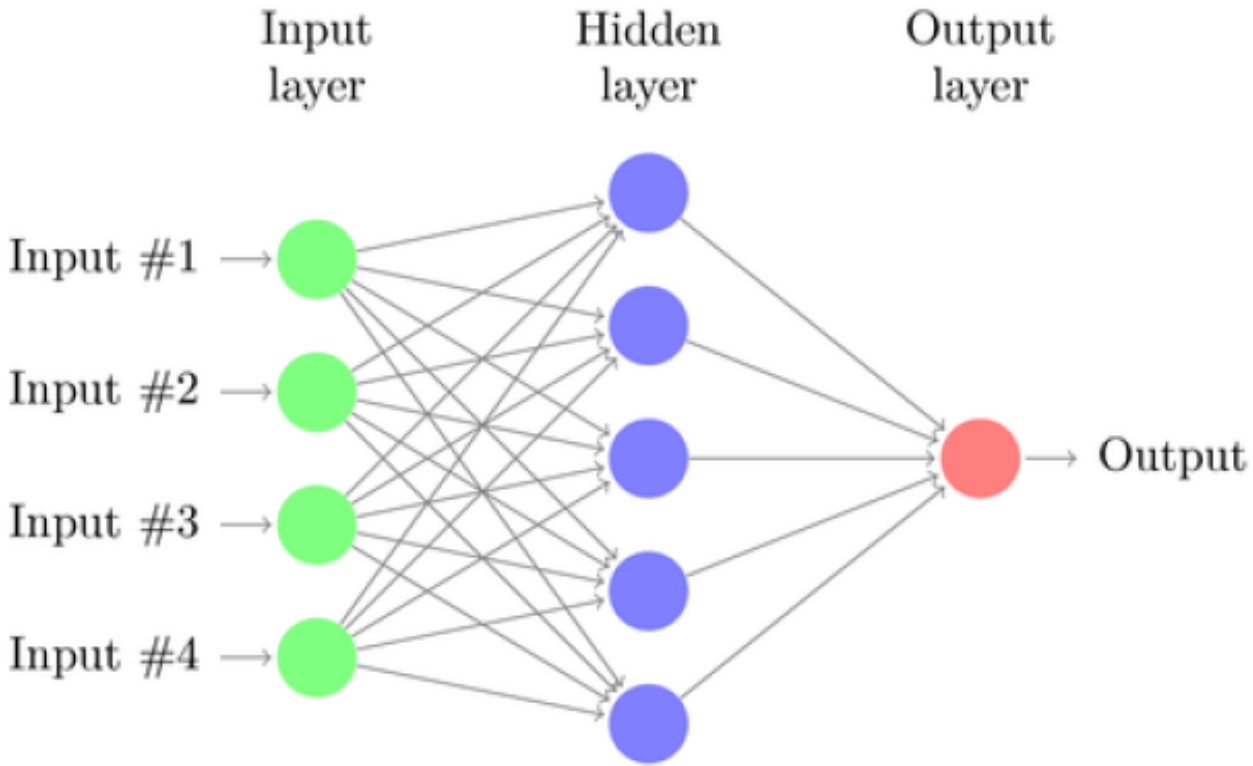
What Do Sigmoid Neurons Buy Us?

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.



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 and 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 neuron is to take the inputs from the synapses and apply its activation function to them.

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

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

Review

Designing and Building Neural Networks

Training Neural Networks

Mathematics of Training

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.

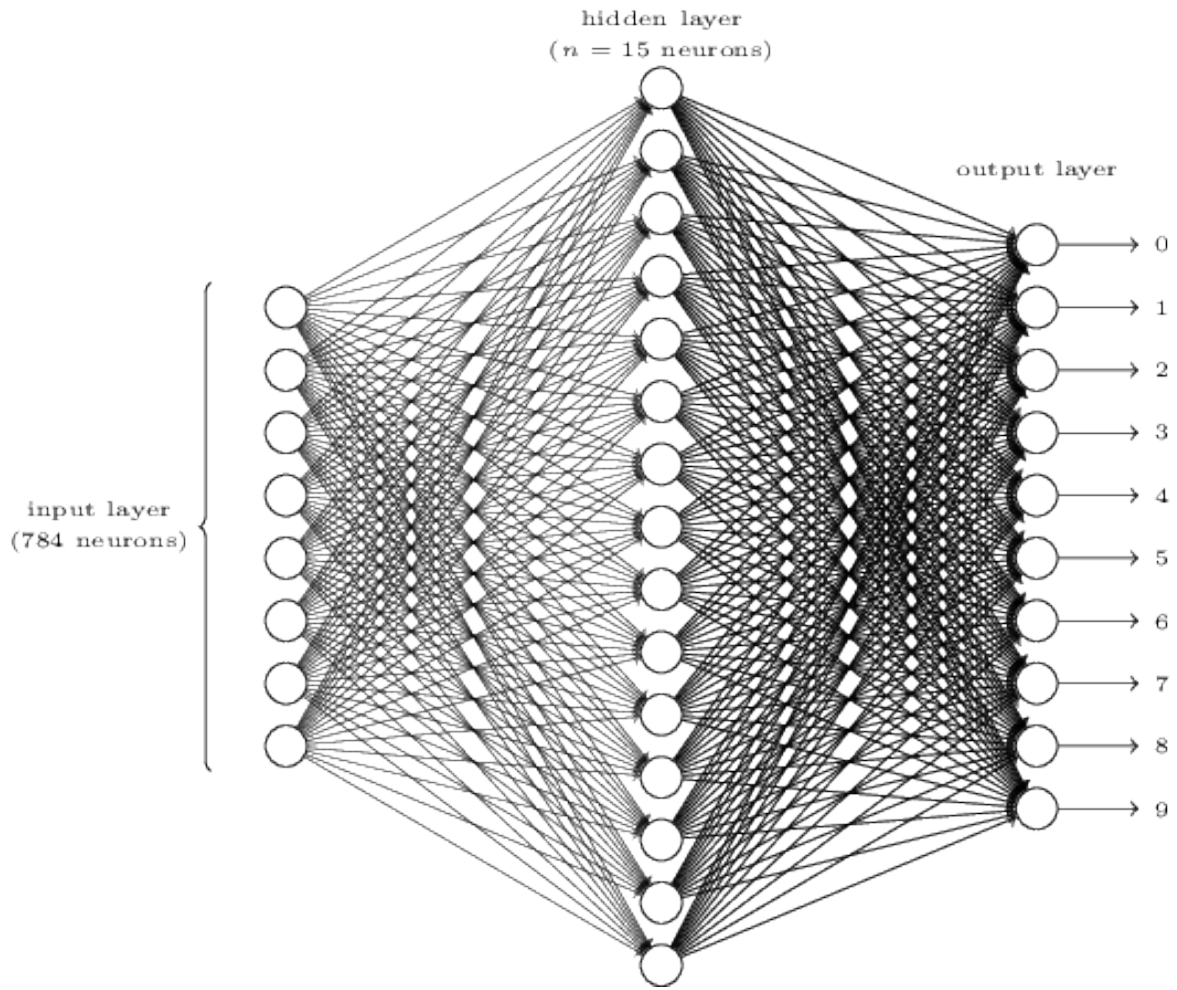
If you're recognizing images, it makes sense to have 10 output nodes - one for each digit. If you're recognizing the presence/absence of a car in an image - 1 or 2 output neurons are enough.

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. 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

def build_2_3_1_nn():
    # 1. seed random number generator
    np.random.seed(1)
    # 2. initiate 1st synapse matrix
    syn0 = np.random.randn(2, 3)
    # 3. initiate 2nd synapse matrix
    syn1 = np.random.randn(3, 1)
    # 4. return 2-tuple of synapse matrices
    return syn0, syn1
```

Sigmoid Activation Function and its Derivative

Here is the sigmoid function, its derivative and derivative approximation:

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

$$\frac{d}{dx}\sigma(x) = \sigma(x)(1 - \sigma(x)) = \frac{e^{-x}}{(1 + e^{-x})^2}.$$

$$\frac{d}{dx}\sigma(x) \approx x(1 - x).$$

Building an ANN: Sigmoid Activation Function

```
def sigmoid(x, deriv=False):
    if (deriv == True):
        return x * (1 - x)
        #return np.exp(-x)/((1 + np.exp(-x))**2)
    return 1 / (1 + np.exp(-x))
```

If you use the real derivative, beware of this warning.

```
Warning (from warnings module):
  File "ann.py", line 47
    return 1 / (1 + np.exp(-x))
RuntimeWarning: overflow encountered in exp
```

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. $Z^{(2)} = XW^{(1)}$; $Z^{(2)}$ is the input to layer 2;
2. $a^{(2)} = f(Z^{(2)})$; $a^{(2)}$ is the output of layer 2;
3. $Z^{(3)} = a^{(2)}W^{(2)}$; $Z^{(2)}$ is the input to output layer (layer 3);
4. $\widehat{Y} = f(Z^{(3)})$; $\widehat{Y}$ is the output of output layer (layer 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: Equations 1 and 2

Equations 1 and 2:

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

Here is Python:

```
Z2 = np.dot(X, W1)
a2 = sigmoid(Z2)
```

Building an ANN: Feedforward: Equations 3 and 4

Equations 3 and 4:

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

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

Here is Python:

```
Z3 = np.dot(a2, W2)
yHat = sigmoid(Z3)
```

Building an ANN: Complete Feedforward

```
## equation 1
Z2 = np.dot(X, W1)
## equation 2
a2 = sigmoid(Z2)
## equation 3
Z3 = np.dot(a2, W2)
## equation 4
yHat = sigmoid(Z3)
```

Outline

Review

Designing and Building Neural Networks

Training Neural Networks

Mathematics of Training

Error/Cost Function

We have to decide how far our output $\widehat{y}$ is from the ground truth y. Let's use this simple formula.

$$\widehat{y}_{err} = y - \widehat{y}.$$

If negative weights cause numerical instability, we can use

$$\widehat{y}_{err} = |y - \widehat{y}|$$

or

$$\widehat{y}_{err} = (y - \widehat{y})^2.$$

Curse of Dimensionality

We can change the weights of our NN by adjustring one weight at a time. Is this a good idea?

No, it is too expensive. This is known as the **curse of dimensionality**. We have to be smarter.

Backpropagation

Backpropagation is an algorithm of adjusting synapse weights on the basis of the output error, i.e., $\widehat{y}_{err}$.

In our network, we have to compute the adjustments for 2 layers, the output layer and the hidden layer.

The adjustment for the output layer is used to adjust $W^{(2)}$. The adjustment for the hidden layer is used to adjust $W^{(1)}$.

Backpropagation: Adjustment for Output Layer

The adjustment for the output layer is used to adjust $W^{(2)}$.

$$\widehat{y}_{err} = y - \widehat{y}$$

$$\widehat{y}_{\Delta} = \widehat{y}_{err} \cdot \sigma'(\widehat{y})$$

In Python:

```
yHat_error = y - yHat
yHat_delta = yHat_error * sigmoid(yHat, deriv=True)
```

Backpropagation: Adjustment for Hidden Layer

The adjustment for the hidden layer is used to adjust $W^{(1)}$.

$$a_{err}^{(2)} = \widehat{y}_{\Delta} \cdot (W^{(2)})^{T}$$

$$a_{\Delta}^{(2)} = \widehat{a}_{err}^{(2)} \cdot \sigma'(a^{(2)}).$$

In Python:

```
a2_error = yHat_delta.dot(W2.T)
a2_delta = a2_error * sigmoid(a2, deriv=True)
```

Backpropagation: Adjusting $W^{(1)}$ and $W^{(2)}$

$$W^{(2)} = (a^{(2)})^T \cdot \widehat{y}_\Delta$$

$$W^{(1)} = X^T \cdot a^{(2)}_\Delta.$$

In Python:

```
W2 += a2.T.dot(yHat_delta)
W1 += X.T.dot(a2_delta)
```

Backpropagation: Putting It All Together

```
def train_3_layer_nn(numIters, X, y, build):
  W1, W2 = build()
  for j in range(numIters):
     Z2 = np.dot(X, W1)
     a2 = sigmoid(Z2)
     Z3 = np.dot(a2, W2)
     yHat = sigmoid(Z3)

     yHat_error = y - yHat
     yHat_delta = yHat_error * sigmoid(yHat, deriv=True)
     a2_error = yHat_delta.dot(W2.T)
     a2_delta = a2_error * sigmoid(a2, deriv=True)
     W2 += a2.T.dot(yHat_delta)
     W1 += X.T.dot(a2_delta)

  return W1, W2
```

Outline

Review

Designing and Building Neural Networks

Training Neural Networks

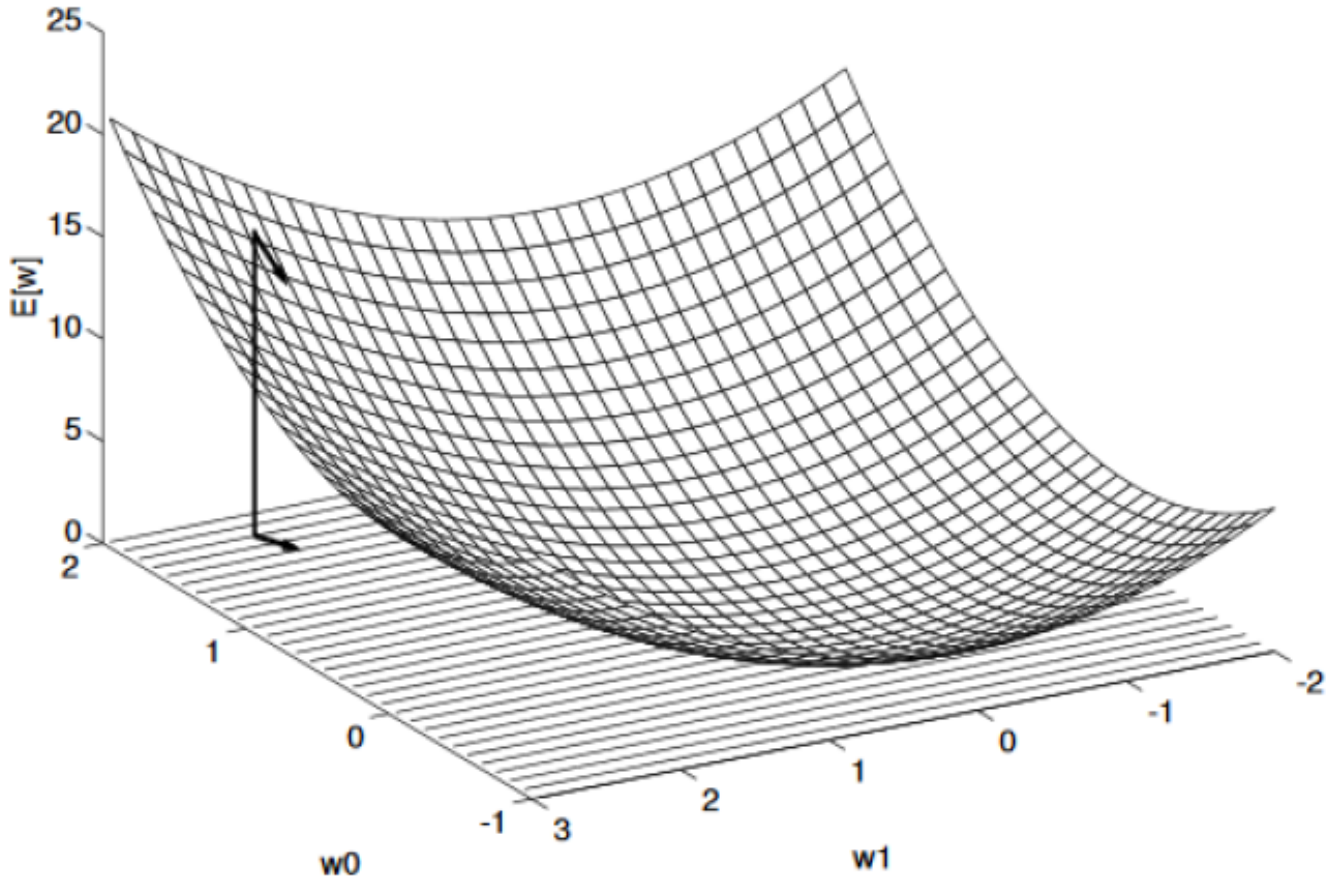
Mathematics of Training

Mathematics of Training

The slides in this section are optional. They expose some mathematics behind of ANN training.

You can read them for your general background and come back to them if and when you want to learn more about ANNs.

Hypothesis Search Space



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.,

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

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*.