Carnegie Mellon University

MSE

Data Analytics for Materials Science

27-737

A.D. (Tony) Rollett, Amit Verma, R.A. LeSar (Iowa State Univ.)

Dept. Materials Sci. Eng., Carnegie Mellon University

Neural Nets, part 1

Lecture 12

Revised: 10th Apr. 2021

Recapitulation

To date, we have discussed:

Linear algebra
Linear regression: prediction
Multiple linear regression (MLR): prediction
Regular expressions
Principal component analysis (PCA)
Canonical Correlation Analysis (CCA)
Random Forest (RF)
Clustering

Resources

- Hastie et al. Elements of Statistical Learning:
  Neural Nets, starting on about p. 389
- https://victorzhou.com/blog/intro-to-neural-networks
- https://en.wikipedia.org/wiki/History_of_artificial_neural_networks
- https://scipy-lectures.org/advanced/mathematical_optimization/index.html
- http://neuralnetworksanddeeplearning.com/chap1.html
  seems to have nice simple explanations of, e.g., perceptrons, and what the adjustment of weights in the network accomplishes.
- http://cs231n.github.io/
  This appears to be a very complete set of notes on NNs, actually a complete course.
- https://machinelearningmastery.com/neural-network-models-for-combined-classification-and-regression/
- General description of NN: https://victorzhou.com/blog/intro-to-neural-networks
- For explaining forward and backward propagation: https://tech.trustpilot.com/forward-and-backward-propagation-5dc3c49c9a05

NEU Defect Database Example

To run the CNN example, look for the zipped package, neu_vgg16_example-master-Feb21.zip, which contains a Jupyter notebook called 1.0-ark_tutorial.ipynb and the (datasets) folder of images with NEU-CLS.zip.

From the website for the NEU example: "In the Northeastern University (NEU) surface defect database, six kinds of typical surface defects of the hot-rolled steel strip are collected, i.e., rolled-in scale (RS), patches (Pa), crazing (Cr), pitted surface (PS), inclusion (In) and scratches (Sc). The database includes 1,800 grayscale images: 300 samples each of six different kinds of typical surface defects."

NEU steel defect discussion, examples: https://akbarikevin.medium.com/neu-surface-defect-dataset-with-tensorflow-api-8753c85fe783

http://faculty.neu.edu.cn/yunhyan/NEU_surface_defect_database.html

- We would appreciate it if you cite our works when using the database:
  K. Song and Y. Yan, "*A noise robust method based on completed local binary patterns for hot-rolled steel strip surface defects*," ***Applied Surface Science***, vol. 285, pp. 858-864, Nov. 2013.(paper)
- Yu He, Kechen Song, Qinggang Meng, Yunhui Yan, "*An End-to-end Steel Surface Defect Detection Approach via Fusing Multiple Hierarchical Features*," ***IEEE Transactions on Instrumentation and Measurement***, 2020,69(4),1493-1504..(paper)
- Hongwen Dong, Kechen Song, Yu He, Jing Xu, Yunhui Yan, Qinggang Meng, "*PGA-Net: Pyramid Feature Fusion and Global Context Attention Network for Automated Surface Defect Detection*," ***IEEE Transactions on Industrial Informatics***, 2020.(paper)

Items of Interest

1. What the .yaml contains
2. Examine os and Path
3. Can someone figure out how to capture the current directory instead of specifying the absolute path?
4. Use of glob for finding files with a common pattern; here *.BMP
5. Use of os.path.basename[].split to find sets of images
6. Use random.seed instead of “seed=27737”
7. Use of sorted() to get a unique set of defect types
8. Saving an image as a file output
9. Re-sizing the greyscale images (200x200) to match VGG16’s 224x224 input with (3 channels of) RGB; use of helper function(s) in preprocessing.py

Neural Nets

- In the beginning … animals developed nervous systems and brains.
- Wikipedia: "The history of artificial neural networks (ANN) began with Warren McCulloch and Walter Pitts[1] (1943) who created a computational model for neural networks based on algorithms called threshold logic. This model paved the way for research to split into two approaches. One approach focused on biological processes while the other focused on the application of neural networks to artificial intelligence."
- McCulloch, Warren; Walter Pitts (1943). "A Logical Calculus of Ideas Immanent in Nervous Activity". Bulletin of Mathematical Biophysics. 5 (4): 115–133.
- 1940s: Hebbian learning hypothesis based on neural plasticity, which was a form of *unsupervised* learning. Turing invented the B-type (computing) machine. 1950s: Farley & Clark used calculators to simulate a Hebbian network. Rosenblatt created the *perceptron*, which is an algorithm for pattern recognition.
- 1960s: The first functional network with multiple layers was described by Ivakhnenko and Lapa in 1965. However, research slowed after Minsky & Papert pointed out that the perceptron lacked the capability to program an exclusive-OR gate. Also, it became clear that the computers of the time lacked the power to implement large neural networks. [Wikipedia]
- 1970s: Werbo published the *backpropagation algorithm* that enabled training of multi-layer networks by distributing the error back up through the layers via modification of weights at each node.
- 1980s: *Connectionism* arose as parallel distributed processing took hold for simulating neural processes, which contributed to the prediction of protein structures. Otherwise, however, support vector machines and linear classifiers developed in competition with NN.
- 1990s: Max-pooling was introduced in 1992 that increased tolerance to deformation and least shift invariance in 3D object recognition.
- 2000s and onwards: since 2010, GPUs have been increasingly used e.g., for backpropagation training via max-pooling. Some used only the sign of the error for learning to cope with the *vanishing gradient problem*. Hinton proposed in 2006 learning a high-level representation with successive layers of binary or real-valued variables with a restricted Boltzmann machine to model each layer. In 2012 Ng & Dean developed a network that was able to recognize high-level concepts such as the presence of a cat in an image in an unsupervised manner. Deployment of neural networks on a large scale for visual recognition problems became known as deep learning. [Wikipedia: https://en.wikipedia.org/wiki/History_of_artificial_neural_networks]
- Contests played a substantial role in accelerating the progress of deep learning and GPU-based implementations won many prizes for problems such as traffic sign recognition, speech transcription, and the ImageNet Competition, for example.

Neural Nets in Materials Science

- Engineers have used (artificial) neural networks (ANNs) for decades. They can be used to develop fits to data, just as we have seen for (multiple) linear regression, CCA, RF etc.
- ANNs developed a bad reputation precisely because the learning process, while often able to match experimental data precisely, results in a non-human interpretable model. In other words, it is a "black box".
- In recent years, however, we have "graduated" to convolutional neural nets (CNNs) in which the connections are sparse. Effectively, the learning process discovers features of images (especially) that give a strong signal. Thus, they act as feature selectors.
- Holm has published (Science 2019) literally a defense of the black box for cases where developing an explicit model is impossible. Which is often the case with microstructural images, i.e., the computer can be trained to do tasks that are far beyond what a human can do.

Neural Net – Details

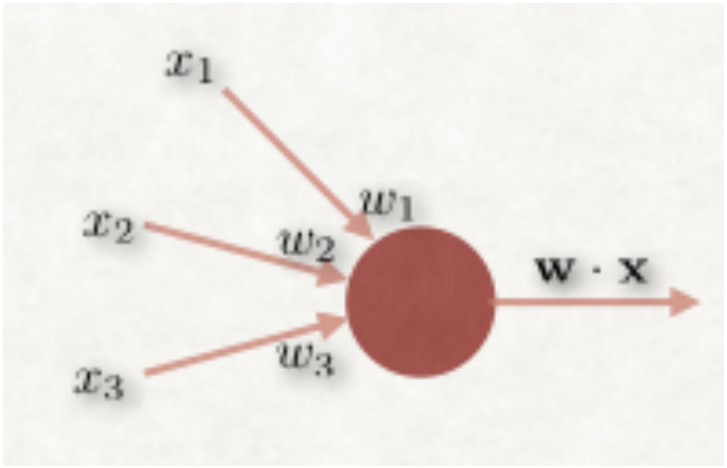
- The following slides are derived from lectures by Prof. Richard LeSar, Iowa State Univ.
- Other slides are taken from a lecture given by Prof. Marc De Graef in 2020, entitled "The Nuts and Bolts of a Dense, Fully Connected Neural Network".

A Neural Net

- In 1943, Warren McCulloch and Walter Pitts developed the first mathematical model of a neuron
- In the paper given below, they described a simple mathematical model for a neuron, representing a single cell of the neural system that takes inputs, processes those inputs, and returns an output.
- This model is known as the McCulloch-Pitts neural model.

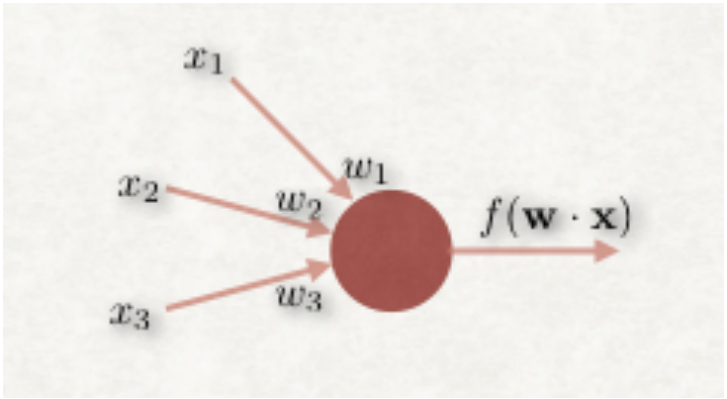
Slightly Less Basic Neuron

- Consider a neuron with 3 inputs, x_i, each with its own weight, w_i.
- If we use the weighted average of the inputs then the output of the neuron can only vary linearly. This is not very useful!

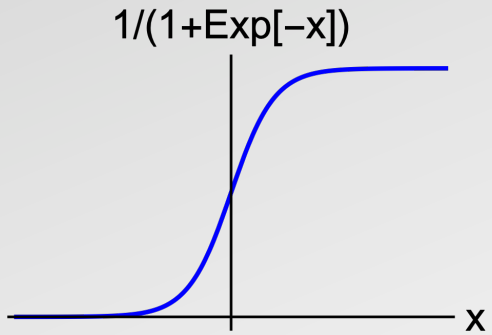


$$\mathbf{w} \cdot \mathbf{x} = \mathbf{w_1 x_1} + \mathbf{w_2 x_2} + \mathbf{w_3 x_3}$$

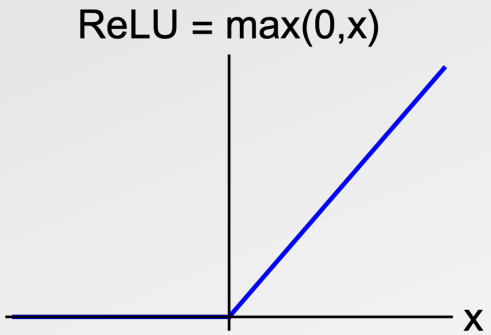
- So, the trick is to add an *activation function, f,* to insert non-linearity. Despite the infinite possibilities, only a very few are in common use.



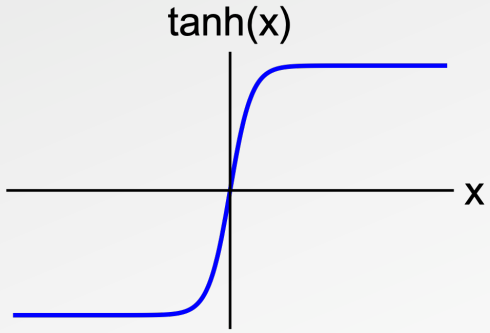
`sigmoid[x]`

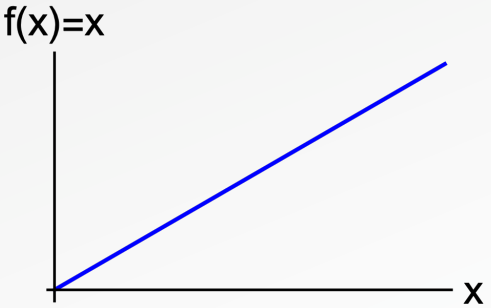


Note that the derivative of the sigmoid has a convenient form



ReLU = rectified linear unit.





Identity

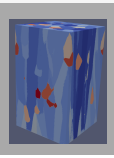
The activation function defines the output of a neuron in terms of a local induced field.

Activation functions give neural nets non-linearity and expressiveness.

There are many activation functions.

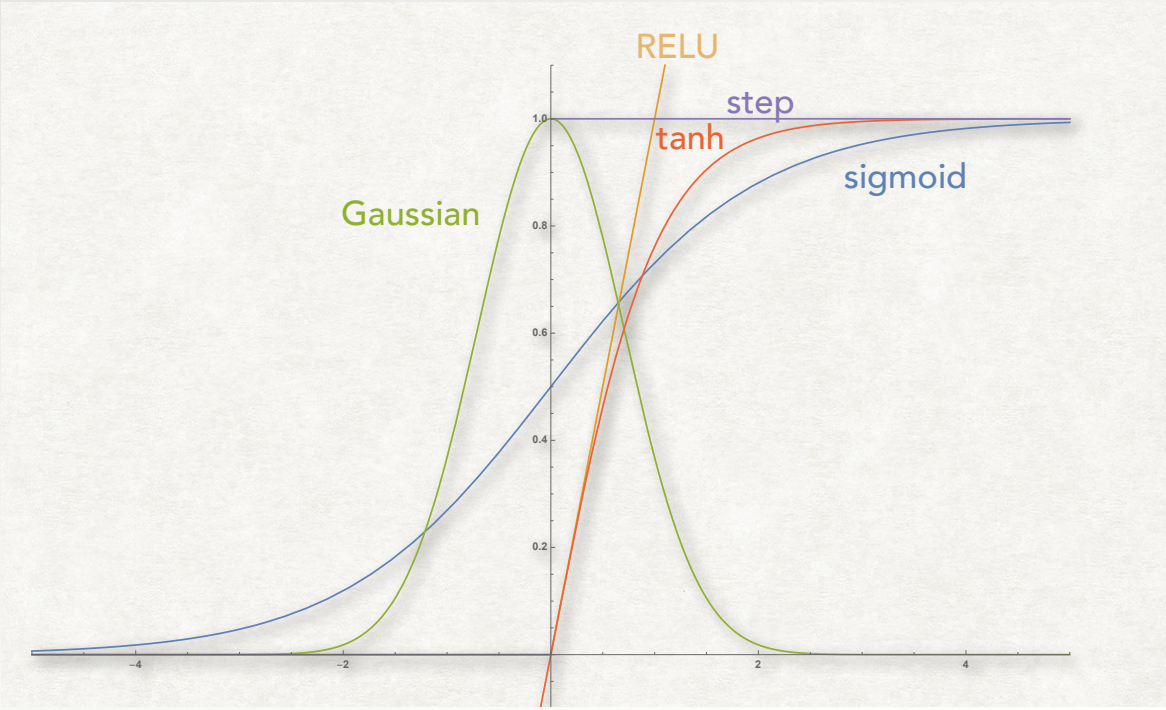
However, the sigmoid and the ReLU are very commonly used.

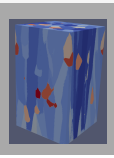
Soon, we will add "bias" to the output, which effectively allows us to offset the activation function w.r.t. to the argument



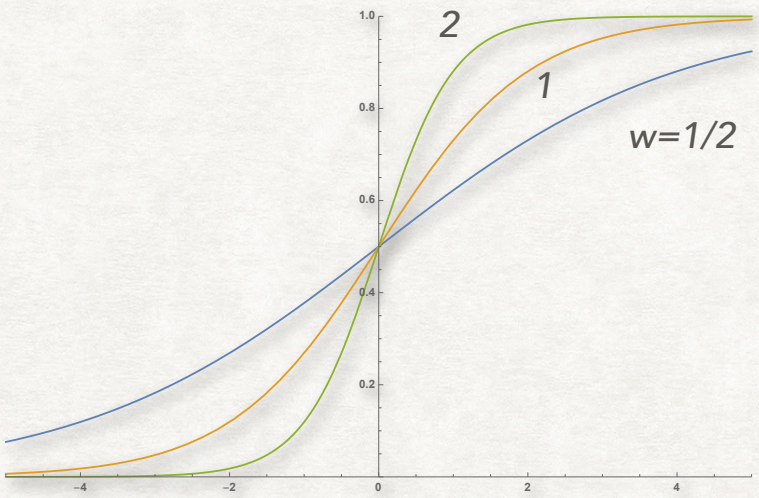
Neural net: activation functions

- sigmoid $f(z) = \dfrac{1}{1+e^{-z}}$
- Gaussian $f(z) = e^{-z^2}$
- RELU $f(z) = \max(0, z)$
- step $f(z) = \begin{cases} 1 & z > 0 \\ 0 & z \leq 0 \end{cases}$
- tanh $f(z) = \tanh(z)$



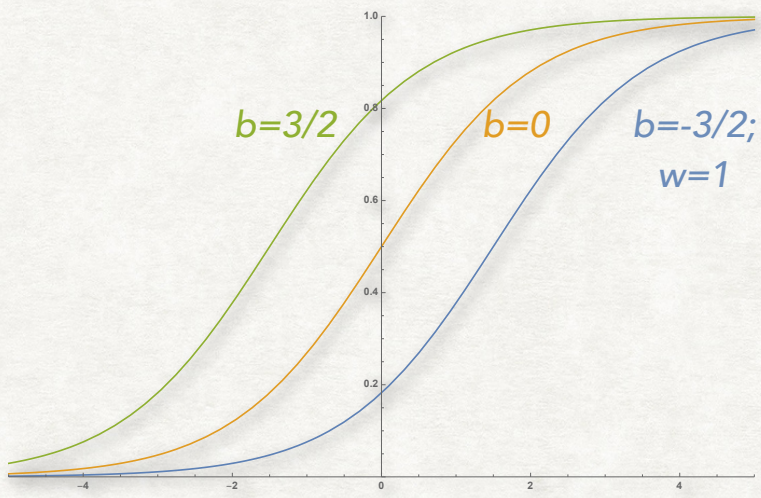


- Another view of typical *Activation Functions*



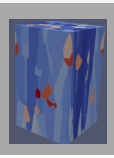
$$f(wx) = \frac{1}{1 + e^{-wx}}$$

$$f(0) = \frac{1}{2} \quad \ldots \text{ always } \ldots$$

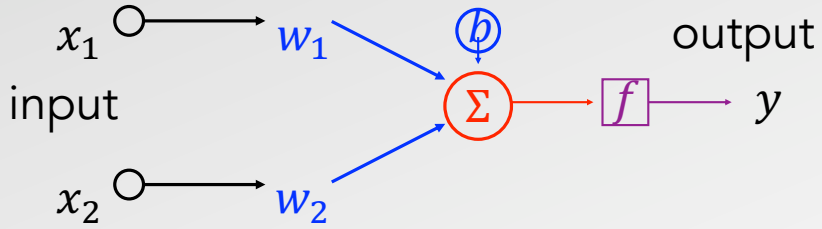


$$f(wx + b) = \frac{1}{1 + e^{-(wx+b)}}$$

Think of the bias as a sort of offset



- How the *bias* component helps, i.e., by varying how strong the input must be to get output from the neuron



1. Each input is multiplied by a weight:

$x_1 \rightarrow w_1 x_1$

$x_2 \rightarrow w_2 x_2$

2. Weighted inputs are summed and have a **bias** b added to them:

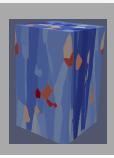
$\Sigma = w_1 x_1 + w_2 x_2 + b$

3. The sum is put through an "activation function" f yielding the output y

$f(w_1 x_1 + w_2 x_2 + b) \rightarrow y$

The neuron's behavior is defined by the parameters w_1, w_2 and b and the function f.

Learning means changing the weights & biases to achieve a desired result



A neural net: steps at a "neuron"

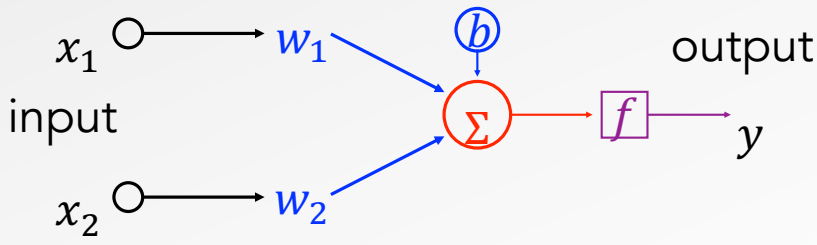
Taking f as the sigmoid function, we would have after 1 pass

(writing $\mathbf{x} = [x_1, x_2]$ and $\mathbf{w} = (w_1, w_2)$)

$$y = f(\mathbf{w} \cdot \mathbf{x} + b) = f(w_1 x_1 + w_2 x_2 + b)$$

$$= \frac{1}{1 + e^{-(\mathbf{w} \cdot \mathbf{x} + b)}}$$

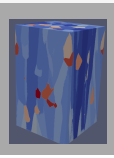
For example, let the "neuron" be defined with $\mathbf{w} = [1,1]$ and $b = 0$.



For $\mathbf{x} = [2,3]$, $y = 0.9933$

This neuron maps $[2,3] \rightarrow 0.9933$.

- Changing the weights and the bias will change the output of the neuron.
- Changing the activation function changes how the neuron *functions*.



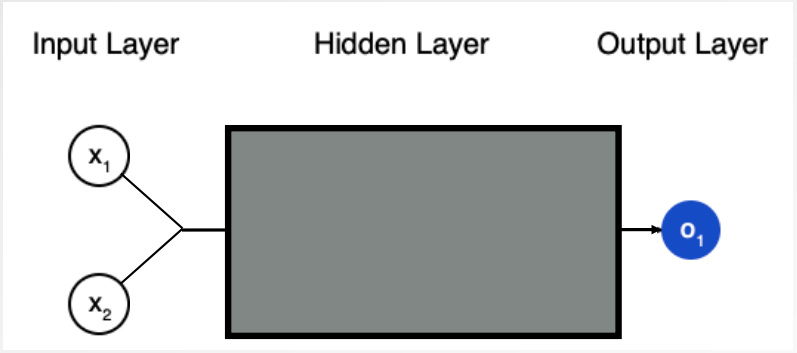
A neural net: simple example

We create a neural network by linking together a number of neurons.

If all the weights and biases are different, then we would have:

$h_1 = f(\mathbf{w}_1 \cdot [x_1, x_2] + b_1)$

$h_2 = f(\mathbf{w}_2 \cdot [x_1, x_2] + b_2)$




```
nr[xx_, yy_, ww1_, ww2_, bb_] := N[sigmoid[ww1 xx + ww2 yy + bb]]
h1 = nr[2, 3, 1, 1, 0]
h2 = nr[2, 3, 1, 1, 0]
o1 = nr[h1, h2, 1, 1, 0]
```

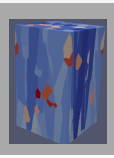
The output o_1 is given by: $o_1 = f(\mathbf{w}_o \cdot [h_1, h_2] + b_o)$

If all $\mathbf{w}_1 = \mathbf{w}_2 = \mathbf{w_o} = [1,1]$ and $b_1 = b_2 = b_o = 0$, then we find: $[2,3] \rightarrow 0.8794$

A hidden layer is **any** layer between the input layer (first) and the output layer (last). There can be many hidden layers.

This is an example of a feedforward operation.

https://victorzhou.com/blog/intro-to-neural-networks



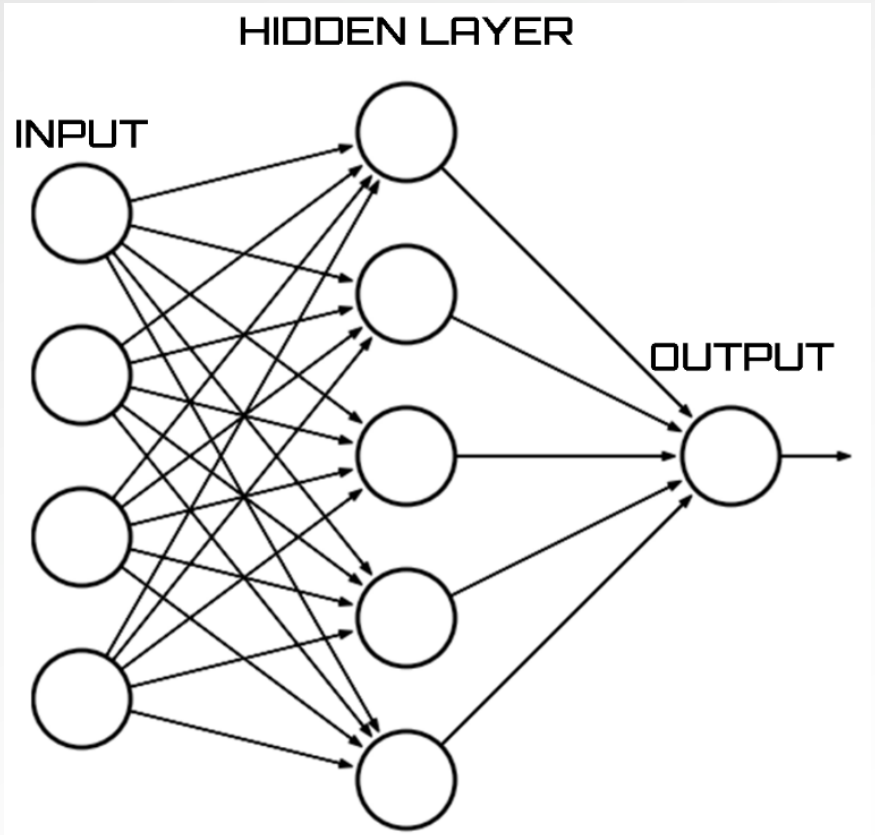
A neural net: hidden layers

The output of such a network can be written as:

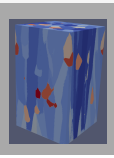
$$o(\mathbf{x}) = \sum_{j=1}^{N} \beta_j f(\mathbf{w}_j \cdot \mathbf{x} + b_j)$$

This is known as *forward propagation*: of course, one has to have a set of weights and biases.

In principle, any activation function can be used. So, e.g., one could use exp{i**w**·**x**} (with zero bias), which would give a Fourier series but not very efficiently!



https://static.packt-cdn.com/products/9781789808452/graphics/7a69eb62-1d09-442b-9e0a-0f38203981a8.png



- Another view of a (fully connected) hidden layer

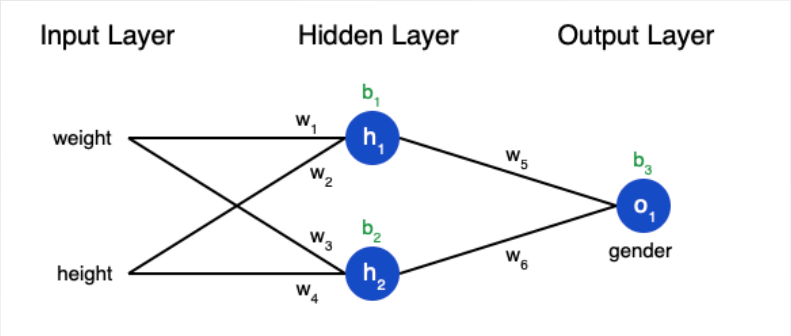
Consider the following dataset:

Name	Weight (lbs)	Height (in)	Gender
Alice	133	65	F
Bob	160	72	M
Charlie	152	70	M
Diana	120	60	F

(1) Subtract the mean of the weight and height from the appropriate columns.
(2) Use F=1 and M=0.

Name	Weight (lbs)	Height (in)	Gender
Alice	-8.25	-1.75	1
Bob	18.75	5.25	0
Charlie	10.75	3.25	0
Diana	-21.25	-6.75	1

$\overline{weight} = 141.25 lb \quad \overline{height} = 66.75 in$

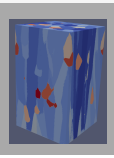


Can we train a neural net as shown above to be able to **predict** the gender based on height and weight?

The input will be the weights and heights for 4 people along with their respective gender.

The output will be a prediction for the gender of the four people and a measure of error. In this case, the desired outcome is to **minimize the error**.

https://victorzhou.com/blog/intro-to-neural-networks



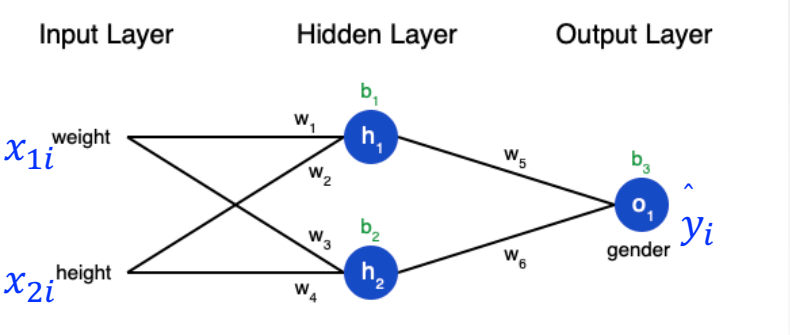
Equations for each pair of weight and height:

$h_{1i} = f(\mathbf{w}_1 \cdot [x_{1i}, x_{2i}] + b_1) i = 1 \ldots 4, \quad h_{2i} = f(\mathbf{w}_2 \cdot [x_{1i}, x_{2i}] + b_2) i = 1 \ldots 4,$

$\hat{y}_i = f(\mathbf{w}_3 \cdot [h_{1i}, h_{2i}] + b_3) i = 1 \ldots 4$

with

$\mathbf{w}_1 = [w_1, w_2], \ \mathbf{w}_2 = [w_3, w_4], \mathbf{w}_o = [w_5, w_6]$



We will quantify the neural net by calculating the square error: (y_i is the actual gender, $\hat{y}_i$ is predicted.)

$$L = \sum_{i=1}^{4} (y_i - \hat{y}_i)^2$$

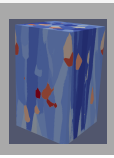
Training is thus a minimization problem: vary $\{w_1, w_2, w_3, w_4, w_5, w_6, b_1, b_2, b_3\}$ to minimize L.
Minimizing the Mean Square Error (MSE) would be equivalent.

Name	Weight (lbs)	Height (in)	Gender
Alice	-8.25	-1.75	1
Bob	18.75	5.25	0
Charlie	10.75	3.25	0
Diana	-21.25	-6.75	1

$\overline{weight} = 141.25 lb$

$\overline{height} = 66.75 in$

https://victorzhou.com/blog/intro-to-neural-networks



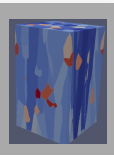
A neural net: training a neural net

```
In[1]:= sigmoid[x_] := 1 / (1 + Exp[-x])

In[•]:= sigmoid[x]

Out[•]= 1 / (1 + e^-x)

nr[xx_, yy_, ww1_, ww2_, bb_] := Chop[ N[sigmoid[ww1 xx + ww2 yy + bb]]]

In[232]:= dat1 // MatrixForm

Out[232]//MatrixForm=
( Alice   -8.25  -1.75  1. )
(   Bob    18.75   5.25  0. )
( Charlie  10.75   3.25  0. )
(  Diana  -21.25  -6.75  1. )

In[313]:=
GF[ww1_, ww2_, ww3_, ww4_, ww5_, ww6_, bb1_, bb2_, bb3_] :=
 Module[{w1 = ww1, w2 = ww2, w3 = ww3, w4 = ww4, w5 = ww5, w6 = ww6, b1 = bb1,
   b2 = bb2, b3 = bb3},
  h1 = Table[nr[dat1[[i, 2]], dat1[[i, 3]], w1, w2, b1], {i, 4}];
  h2 = Table[nr[dat1[[i, 2]], dat1[[i, 3]], w3, w4, b2], {i, 4}];
  o1 = Table[nr[h1[[i]], h2[[i]], w5, w6, b3], {i, 4}];
  mse = Sum[(o1[[i]] - dat1[[i, 4]])^2, {i, 4}];
  mse]
```

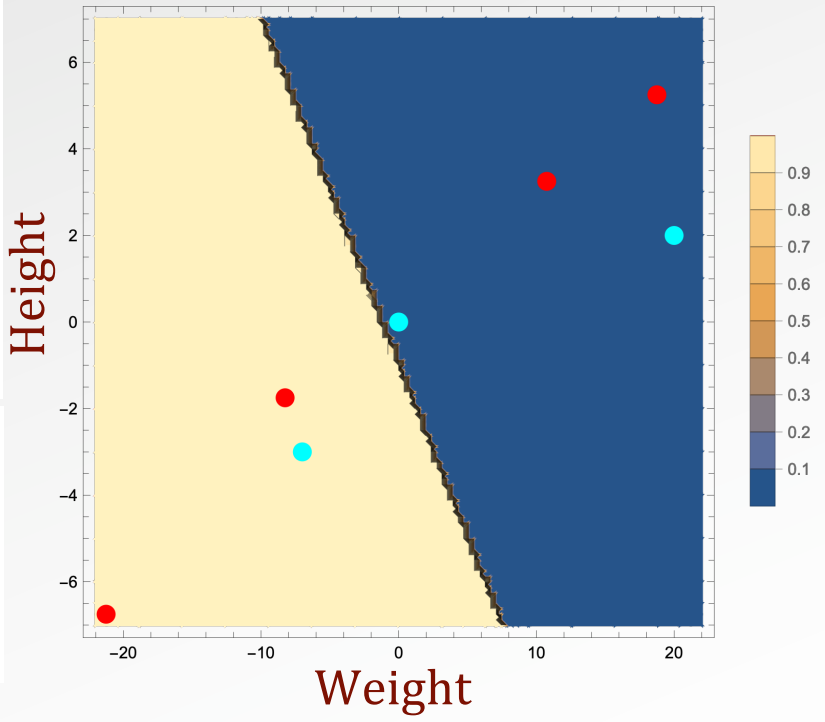
```
ww1 = 1;
ww2 = 1;
ww3 = 1;
ww4 = 1;
ww5 = 1;
ww6 = 1;
b1 = 0;
b2 = 0;
b3 = 0;
Chop[Minimize[GF[xw1, xw2, xw3, xw4, xw5, xw6, xb1, xb2, xb3],
  {xw1, xw2, xw3, xw4, xw5, xw6, xb1, xb2, xb3}]]
{0, {xw1 → -89.1732, xw2 → -111.147, xw3 → 161.528, xw4 → -154.281,
  xw5 → 152.525, xw6 → 49.7479, xb1 → -104.725, xb2 → 164.886, xb3 → -109.99}}

GF[-89.1732, -111.147`, 161.528`, -154.281, 152.525, 49.7479 , -104.725 ,
 164.886, -109.99]
o1

0.

{1., 0, 0, 1.}
```

$L \rightarrow$

LeSar trained this NN using a brute force minimization in Mathematica. The weights and biases were varied until the loss function $L = 0$. The results for those parameters (indicated "w" and "b" values enclosed by the red box) can now be used to see how well this NN works for predicting gender.

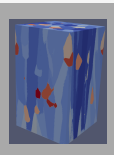


```
In[196]:= test[bx1_, bx2_, ww1_, ww2_, ww3_, ww4_, ww5_, ww6_, bb1_, bb2_, bb3_] :=
           Module[{x1 = bx1, x2 = bx2, w1 = ww1, w2 = ww2, w3 = ww3, w4 = ww4, w5 = ww5,
             w6 = ww6, b1 = bb1, b2 = bb2, b3 = bb3},
            h1 = nr[x1, x2, w1, w2, b1];
            h2 = nr[x1, x2, w3, w4, b2];
            o1 = nr[h1, h2, w5, w6, b3];
            o1]

In[197]:= test[-7, -3, -89.17315160846633`, -111.14689665817994`, 161.5282636801397`,
           -154.28119811687068`, 152.52523997462785`, 49.747919848755814`, -104.72488085879272`,
           164.88552760518735`, -109.98970807864066]

          General: Exp[-852.928] is too small to represent as a normalized machine number; precision may be lost.

Out[197]= 1.

In[199]:= Chop[test[20, 2, -89.17315160846633`, -111.14689665817994`, 161.5282636801397`,
             -154.28119811687068`, 152.52523997462785`, 49.747919848755814`,
             -104.72488085879272`, 164.88552760518735`, -109.98970807864066]]

          General: 2.68807125552270160725590311112.630208178148958*^-917 is too small to represent as a normalized machine
          number; precision may be lost.

          General: Exp[-3086.89] is too small to represent as a normalized machine number; precision may be lost.

Out[199]= 0
```



$\overline{weight} = 141.25\,lb$ $\overline{height} = 66.75\,in$

This is a stupid way to solve this problem — we need a better way to find a minimum for the large NNs we may need to create.

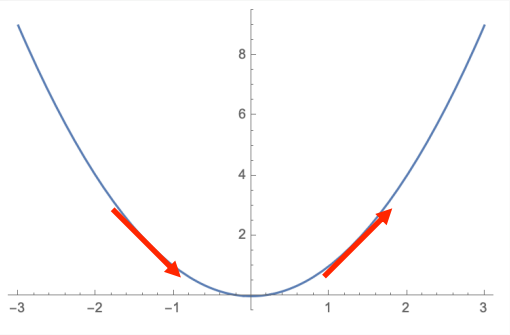


A neural net: training a neural net by brute force

Computational optimization is a very large field — whole courses are taught on this subject.

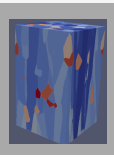
- We will discuss here gradient methods for optimization.
- Suppose we have a loss function defined as: $L(w) = (1/n)\sum_{i=1}^{n} L_i(w)$, where w is a parameter
- Standard gradient descent methods would take an iterative approach using

  $w \leftarrow w - \eta \frac{\partial L}{\partial w} = w - \frac{\eta}{n}\sum_{i=1}^{n} \frac{\partial L_i}{\partial w}$. η is the learning rate (in ML).
- If $\partial L/\partial w > 0$, w will decrease
- If $\partial L/\partial w < 0$, w will increase
- Doing this iteration many many times, the system will slowly approach the minimum of $L(w)$



The problem is that even with the hierarchical nature of the derivatives, in large NNs, there are so many derivatives to calculate that it would be prohibitive because computational time would explode.

One can think of the *learning rate* as a fraction of the correction that we apply in each iteration. Too small and we never get there: too large and we oscillate around the solution.



Optimization

https://victorzhou.com/blog/intro-to-neural-networks
https://scipy-lectures.org/advanced/mathematical_optimization/index.html

We need to find an efficient and accurate way to vary the weights and biases to minimize the loss function $L(\{w\})$

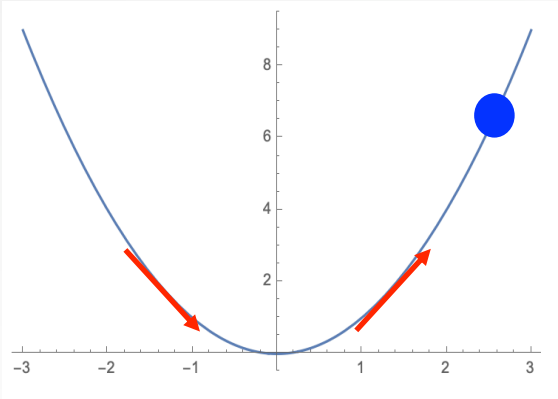
This is an example of a computational optimization, which is a very large field. Indeed, whole courses are taught on the subject.

We focus on an approach that depends on using the *gradients* of the $L(\{w\})$.

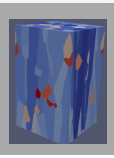
Consider the plot at the right, which is the function $f(x) = x^2$. Suppose the system is at $x = x_o$ (the blue dot). The easiest computational way to find the x that minimizes $_f$ is to start at $x = x_o$ and use an iterative procedure:

$x \leftarrow x - \eta \frac{df}{dx}$,

where η controls the "step size". Stop the procedure when the change in x in the iteration is less than some prescribed value.



If $df/dx > 0$, x will decrease
If $df/dx < 0$, x will increase



Suppose we have a loss function defined as

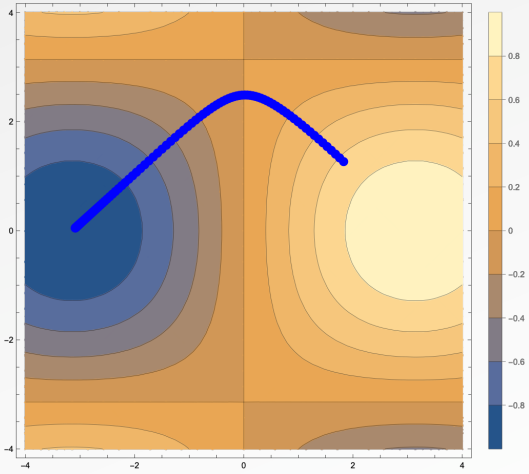
$$L(w) = (1/n)\sum_{i=1}^{n} L_i(w),$$

where w is one of the parameters.

Standard gradient descent methods would take an iterative approach using

$$w \leftarrow w - \eta \frac{\partial L}{\partial w} = w - \frac{\eta}{n}\sum_{i=1}^{n} \frac{\partial L_i}{\partial w}.$$

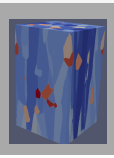
- η is the (step size) learning rate (in ML).
- by doing the iteration many many times, we hope that the system will approach the minimum of $L(w)$ (as opposed to a local minimum)
- There are many variants of this method (steepest descent, conjugate gradient, etc.)



$f[x, y] = \sin[x/2]\cos[y/2]$

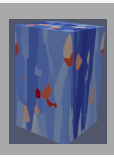
$x \leftarrow x - \eta df/dx$

$y \leftarrow y - \eta df/dy$

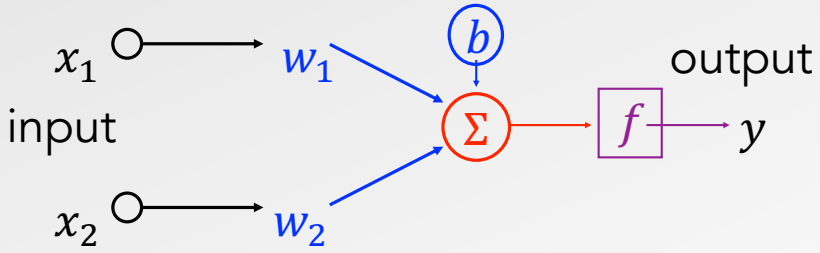


Optimization: Gradient descent methods

Derivatives of L with respect to weights and biases



Suppose we have data $\{x_1, x_2, y\}$ and we want to find the parameters $\{w_1, w_2, b\}$ that minimize the loss function $L(w_1, w_2, b) = (y - \hat{y})^2$, in which $\hat{y}$ is the output of the neuron.



We need the derivative of L with respect to the parameters $q \in \{w_1, w_2, b\}$. We invoke the chain rule: $\frac{\partial L}{\partial q} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial q}$, where $\hat{y} = f(w_1 x_1 + w_2 x_2 + b)$.

(1) $\frac{\partial L}{\partial \hat{y}} = -2(y - \hat{y})$.

(2) $\frac{\partial \hat{y}}{\partial w_1} = x_1 f'(w_1 x_1 + w_2 x_2 + b) = x_1 \hat{y}(1 - \hat{y})$

$\frac{\partial \hat{y}}{\partial w_2} = x_2 f'(w_1 x_1 + w_2 x_2 + b) = x_2 \hat{y}(1 - \hat{y})$

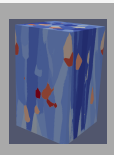
$\frac{\partial \hat{y}}{\partial b} = f'(w_1 x_1 + w_2 x_2 + b) = \hat{y}(1 - \hat{y})$

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

$$f'(x) = \frac{df}{dx} = \frac{e^{-x}}{(1 + e^{-x})^2} = f(x)(1 - f(x))$$

Since $\hat{y} = f(w_1 x_1 + w_2 x_2 + b)$

$$f'(w_1 x_1 + w_2 x_2 + b) = \hat{y}(1 - \hat{y})$$



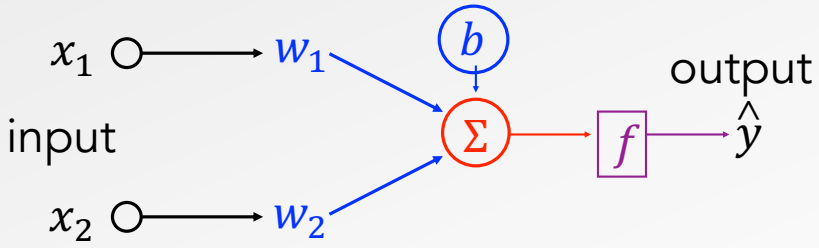
Derivatives of the loss function: 1 neuron

output layer input layer

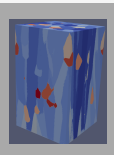
(1) $\frac{\partial L}{\partial w_1} = (-2(y-\hat{y})) \times (x_1 \hat{y}(1-\hat{y}))$

(2) $\frac{\partial L}{\partial w_2} = (-2(y-\hat{y})) \times (x_2 \hat{y}(1-\hat{y}))$

(3) $\frac{\partial L}{\partial b} = (-2(y-\hat{y})) \times (\hat{y}(1-\hat{y}))$



At the end of each pass, we calculated the derivatives by *back propagation* (i.e., working backwards from the outputs via the gradients to adjust the weights) and iterate using $w \leftarrow w - \eta \frac{\partial L}{\partial w}$.



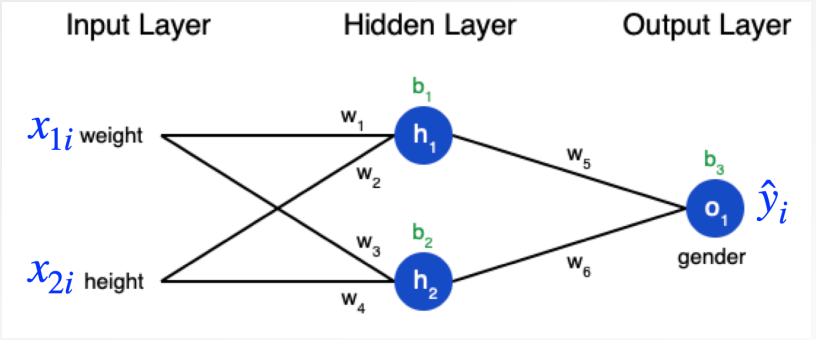
Derivatives of the loss function: 1 neuron

$$L = L(w_1, w_2, w_3, w_4, w_5, w_6, b_1, b_2, b_3) = \sum_{i=1}^{n} L_i = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 \text{ for } n \text{ samples } \{x_{1i}, x_{2i}, y_i\}.$$

We need $\frac{\partial L}{\partial q_j} = \sum_{i=1}^{n} \frac{\partial L_i}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial q_j}$

with $q_j \in \{w_1, w_2, w_3, w_4, w_5, w_6, b_1, b_2, b_3\}$.

Simplest case: $q_j \in \{w_5, w_6, b_3\}$.



(1) $\frac{\partial L_i}{\partial \hat{y}_i} = -2(y_i - \hat{y}_i)$.

(2) $\hat{y}_i = f(w_5 h_{1i} + w_6 h_{2i} + b_3)$:

(3) $\frac{\partial \hat{y}_i}{\partial w_5} = h_{1i}\hat{y}_i(1 - \hat{y}_i)$

$\frac{\partial \hat{y}_i}{\partial w_6} = h_{2i}\hat{y}_i(1 - \hat{y}_i)$

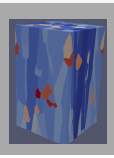
$\frac{\partial \hat{y}_i}{\partial b_3} = \hat{y}_i(1 - \hat{y}_i)$

output layer hidden layer

(1) $\frac{\partial L_i}{\partial w_5} = (-2(y_i - \hat{y}_i)) \times (h_{1i}\hat{y}_i(1 - \hat{y}_i))$

(2) $\frac{\partial L_i}{\partial w_6} = (-2(y_i - \hat{y}_i)) \times (h_{2i}\hat{y}_i(1 - \hat{y}_i))$

(3) $\frac{\partial L_i}{\partial b_3} = (-2(y_i - \hat{y}_i)) \times (\hat{y}_i(1 - \hat{y}_i))$

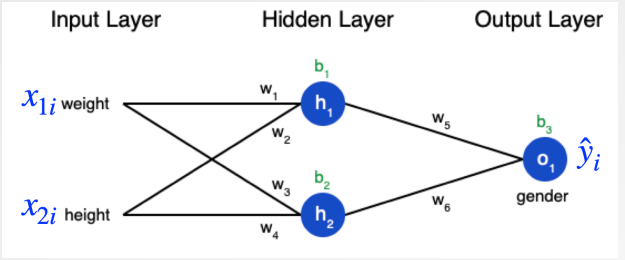


Derivatives of the loss function: hidden layers

$$\frac{\partial L}{\partial q_i} = \sum_{i=1}^{n} \frac{\partial L}{\partial \hat{y}_i}\frac{\partial \hat{y}_i}{\partial q_i} \quad q_i \in \{w_1, w_2, w_3, w_4, b_1, b_2\}\text{: input layer}$$

$$\frac{\partial L}{\partial \hat{y}_i} = -2(y_i - \hat{y}_i) \text{ and}$$

$$h_{1i} = f(w_1 x_{i1} + w_2 x_{2i} + b_1),$$
$$h_{2i} = f(w_3 x_{1i} + w_4 x_{2i} + b_2),$$
$$\hat{y}_i = f(w_5 h_{1i} + w_6 w_{2i} + b_3)$$



If $q_j \in \{w_1, w_2, b_1\}$, $\frac{\partial \hat{y}_i}{\partial q_j} = \frac{\partial \hat{y}_i}{\partial h_{1i}}\frac{\partial h_{1i}}{\partial q_j} \Rightarrow \frac{\partial \hat{y}_i}{\partial h_{1i}} = w_5 \hat{y}_i(1 - \hat{y}_i)$

$\frac{\partial h_{1i}}{\partial w_1} = x_{1i} h_{1i}(1 - h_{1i})$, $\frac{\partial h_{1i}}{\partial w_2} = x_{2i} h_{1i}(1 - h_{1i})$; $\frac{\partial h_{1i}}{\partial b_1} = h_{1i}(1 - h_{1i})$

If $q_i \in \{w_3, w_4, b_2\}$, $\frac{\partial \hat{y}_i}{\partial q_j} = \frac{\partial \hat{y}_i}{\partial h_{2i}}\frac{\partial h_{2i}}{\partial q_j} \Rightarrow \frac{\partial \hat{y}_i}{\partial h_{2i}} = w_6 \hat{y}_i(1 - \hat{y}_i)$

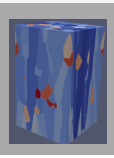
$\frac{\partial h_{2i}}{\partial w_3} = x_{1i} h_{2i}(1 - h_{2i})$, $\frac{\partial h_{2i}}{\partial w_4} = x_{2i} h_{2i}(1 - h_{2i})$; $\frac{\partial h_{2i}}{\partial b_2} = h_{2i}(1 - h_{2i})$

output layer hidden layer input layer

$$\frac{\partial L_i}{\partial w_1} = (-2(y_i - \hat{y}_i))(w_5 \hat{y}_i(1 - \hat{y}_i))(x_{1i} h_{1i}(1 - h_{1i}))$$
$$\frac{\partial L_i}{\partial w_2} = (-2(y_i - \hat{y}_i))(w_5 \hat{y}_i(1 - \hat{y}_i))(x_{2i} h_{1i}(1 - h_{1i}))$$
$$\frac{\partial L_i}{\partial b_1} = (-2(y_i - \hat{y}_i))(w_5 \hat{y}_i(1 - \hat{y}_i))(h_{1i}(1 - h_{1i}))$$

output layer hidden layer input layer

$$\frac{\partial L_i}{\partial w_3} = (-2(y_i - \hat{y}_i))(w_6 \hat{y}_i(1 - \hat{y}_i))(x_{1i} h_{2i}(1 - h_{2i}))$$
$$\frac{\partial L_i}{\partial w_4} = (-2(y_i - \hat{y}_i))(w_6 \hat{y}_i(1 - \hat{y}_i))(x_{2i} h_{2i}(1 - h_{2i}))$$
$$\frac{\partial L_i}{\partial b_2} = (-2(y_i - \hat{y}_i))(w_6 \hat{y}_i(1 - \hat{y}_i))(h_{2i}(1 - h_{2i}))$$



Derivatives of the loss function: hidden layers

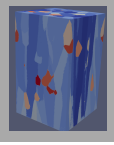
$$\frac{\partial L}{\partial w_1} = \sum_{i=1}^{n} \underbrace{\left(-2(y_i - \hat{y}_i)\right)}_{\text{output layer}} \underbrace{[(w_5 \hat{y}_i (1 - \hat{y}_i)) \ldots]}_{\text{hidden layers}} \underbrace{(x_{1i} h_{1i} (1 - h_{1i}))}_{\text{input layer}}$$

This is called "backpropagation" — we work backwards in the network to calculate the derivatives — the derivatives reflect the network at all layers of the NN.

However, since the needed information has already been calculated, from a computer's perspective, this just requires a certain amount of bookkeeping, and a lot of calculations.

However:

- All hidden layers are included in the derivative — there could be many.
- The number of data points n could be very large.
- Doing a full gradient minimization with all these derivatives might (depending on the number of hidden layers and n) be computationally prohibitive.

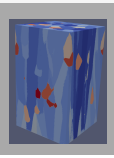


Derivatives of the loss function: hidden layers

It is useful to consider a simpler example of how *forward* and *backward propagation* actually work.

Briefly, NNs use an iterative process to optimize the weight and bias values. We use the training set (of data) so that we have some known "answers" to compare against. *Forward propagation* uses the inputs to calculate (with the current weight & bias values) a new set of estimated outputs. We compute the error signal (e.g., squared error or MSE). We then compute the gradients (discussed in the preceding slides), i.e., rate of change of each weight/bias value with respect to the error value. *Backward propagation* uses the gradient values to update all the weight and bias values.

Link: https://tech.trustpilot.com/forward-and-backward-propagation-5dc3c49c9a05



Paraphrasing from the blogpost by Balász Tóth:

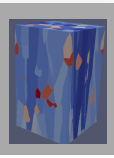
In the post, he walks us through a simple neural network example and illustrate how forward and backward propagation work. His neural network example predicts the outcome of the logical conjunction.

The logical conjunction (AND operator) takes two inputs and returns one output. The function only returns true, if both of its inputs are true. The truth table of it looks like the figure in top right.

The neural network has 2 inputs, p and q, and one output, the prediction of p & q. The training set includes 2 examples out of the 4 possible examples.

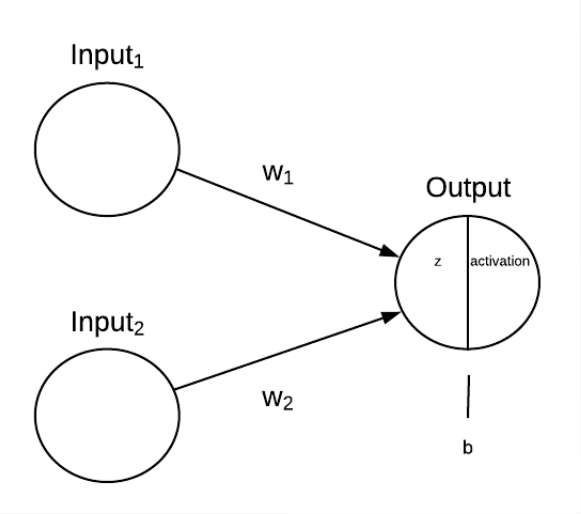
p	q	p & q
1	1	1
0	1	0
0	0	0
1	0	0

$$X_1 = [1,1]$$
$$X_2 = [0,1]$$



- Logical AND example, P1

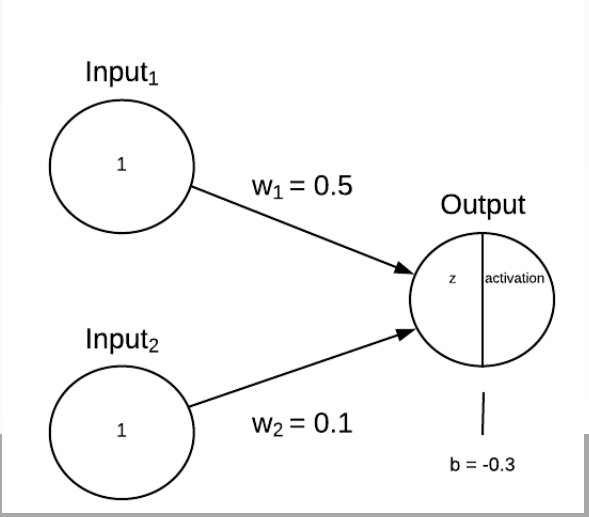
For the first training example, the desired output of the neural network is 1, and for the second training example, it is 0. The neural network has 2 neurons in the input layer and 1 neuron on the output layer. The structure looks like this on the right.

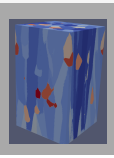


Forward propagation:

In the forward propagation, we check what the neural network predicts for the first training example with initial weights and bias. First, we initialize the weights and bias randomly.

Then we calculate z, the weighted sum of activation and bias:





- Logical AND example, P2

Then we calculate z, the weighted sum of activation and bias: diagram on the right

After we have z, we can apply the activation function to it:

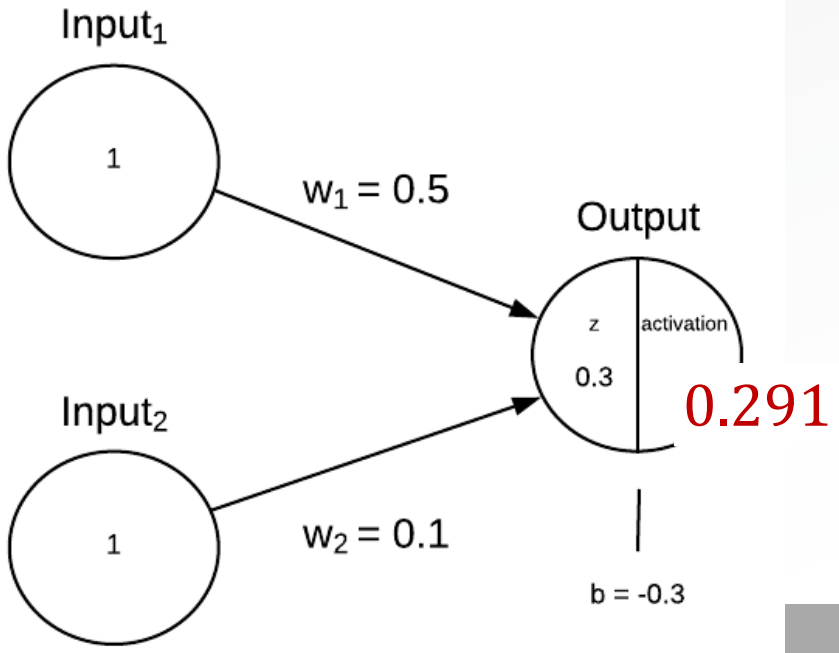
$$\text{Output} = \sigma(z)$$

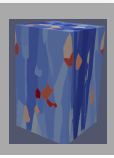
σ is the activation function. The most common activation functions are *relu*, sigmoid and tanh. In this example, we are going to use *tanh*.

$$\text{Output} = \tanh(0.3) = 0.291$$

$$z = Input_1 \times w_1 + Input_2 \times w_2 + b$$

$$z = 1 \times 0.5 + 1 \times 0.1 \ -0.3 = 0.3$$

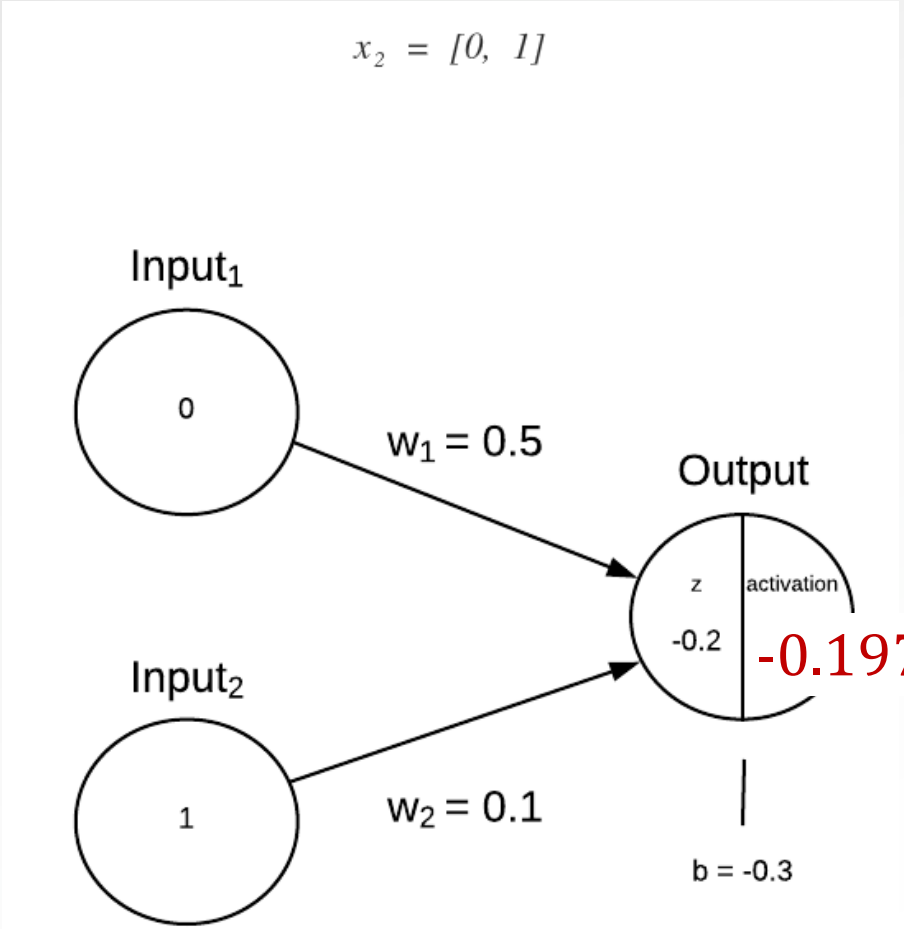


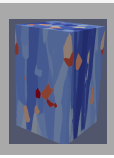


- Logical AND example, P3

For the first training example, our neural network predicted the outcome 0.291. Our desired outcome is 1. The neural network can improve with the learning process of backward propagation. Before we continue with the backward propagation, we calculate the prediction for the second training example, $x=[0,1]$.

These actions give us the two results for the *forward propagation* step.





- Logical AND example, P4

We can define a cost function that measures how good our neural network performs. For an input, x, and desired output, y, we can calculate the cost of a specific training example as the square of the difference between the network's output and the desired output, that is,

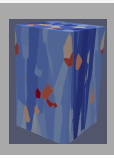
$$C_k = (\text{Output} - y)^2$$

where *k* stands for the training example and the output is assumed to be the activation of the output neuron, and *y* is the actual desired output. For our training examples, the costs are the following:

$$C_1 = (0.291 - 1)^2 = 0.502; \quad C_2 = (-0.197 - 0)^2 = 0.038$$

The total cost of a training set is the average of the individual cost functions of the data in the training set, i.e., *MSE*. Here, *N* stands for the number of training examples.

$$C = \frac{1}{N} \sum C_k$$



- Logical AND example, P5

In our training set the *MSE* looks like this:

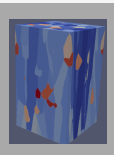
$$C = \frac{0.502 + 0.038}{2} = 0.27$$

We want to improve the performance of the neural network on the training examples, so that we can change the weights and bias, and hopefully, lower the total cost. We want to know how much the specific weights and bias affect the total cost, so we need to calculate the partial derivatives of the total cost with respect to the weights and bias. To do this, we can apply the chain rule:

$$\frac{\partial C_k}{\partial w_1} = \frac{\partial C_k}{\partial Output} \frac{\partial Output}{\partial z} \frac{\partial z}{\partial w_1}$$

$$\frac{\partial C_k}{\partial w_2} = \frac{\partial C_k}{\partial Output} \frac{\partial Output}{\partial z} \frac{\partial z}{\partial w_2}$$

$$\frac{\partial C_k}{\partial b} = \frac{\partial C_k}{\partial Output} \frac{\partial Output}{\partial z} \frac{\partial z}{\partial b}$$



- Logical AND example, P6

After simplification, the parts look like this:

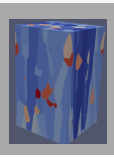
$$\frac{\partial C_k}{\partial Output} = 2(Output - y)$$

$$\frac{\partial Output}{\partial z} = \sigma'(z)$$

$$\frac{\partial z}{\partial w_1} = Input_1$$

$$\frac{\partial z}{\partial w_2} = Input_2$$

$$\frac{\partial z}{\partial b} = 1$$



- Logical AND example, P7

The calculation of the partial derivatives for the first training example

$$\frac{\partial C_1}{\partial Output} = 2(0.291 - 1) = -1.418$$

$$\frac{\partial Output}{\partial z} = \frac{1}{\cosh^2 0.3} = 0.915$$

$$\frac{\partial z}{\partial w_1} = 1$$

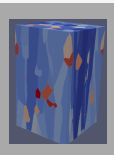
$$\frac{\partial z}{\partial w_2} = 1$$

$$\frac{\partial z}{\partial b} = 1$$

$$\frac{\partial C_1}{\partial w_1} = (-1.418) \times 0.915 \times 1 = -1.297$$

$$\frac{\partial C_1}{\partial w_2} = (-1.418) \times 0.915 \times 1 = -1.297$$

$$\frac{\partial C_1}{\partial b} = (-1.418) \times 0.915 \times 1 = -1.297$$



- Logical AND example, P8

The partial derivatives for the second training example:

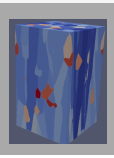
$$\frac{\partial C_2}{\partial w_1} = 0$$

$$\frac{\partial C_2}{\partial w_2} = -0.379$$

$$\frac{\partial C_2}{\partial b} = -0.379$$

Now, we calculate the partial derivatives with respect to the total cost. Consider the first weight (w_1). The partial derivative of the total cost with respect to w_1 is the average of all the partial derivatives of the individual cost functions with respect to w_1:

$$\frac{\partial C}{\partial w_1} = \frac{1}{N} \sum_{k=0}^{N-1} \frac{\partial C_k}{\partial w_1}$$



- Logical AND example, P9

We do the same calculation for the other (2nd) weight and bias:

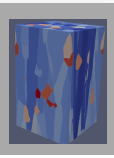
$$\frac{\partial C}{\partial w_2} = \frac{(-1.297) + (-0.379)}{2} = -0.838$$

$$\frac{\partial C}{\partial b} = \frac{(-1.297) + (-0.379)}{2} = -0.838$$

Then we update the weights and bias: we multiply the partial derivatives with some learning rate and subtract the results from the weights and bias.
Let's use a value of the learning rate $\alpha = 0.6$

$$w_1^+ = w_1 - (\alpha \times \frac{\partial C}{\partial w_1})$$

$$w_1^+ = 0.5 - (0.6 \times (-0.648)) = 0.888$$



- Logical AND example, P10

Repeating this calculation with the other weight and the bias:

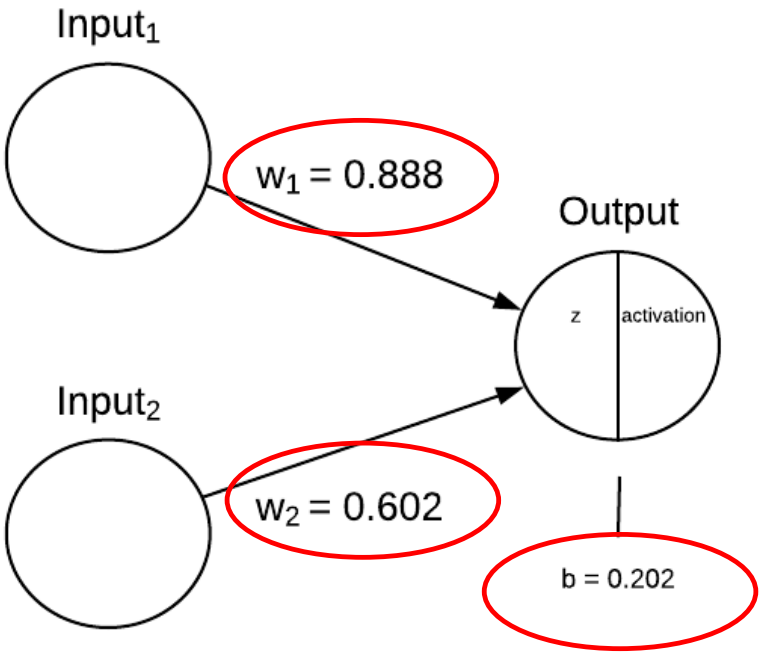
$$w_2^+ = 0.602$$

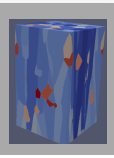
$$b^+ = 0.202$$

After updating the weights and bias, our neural network looks like this:

Notice the new values of weights and bias

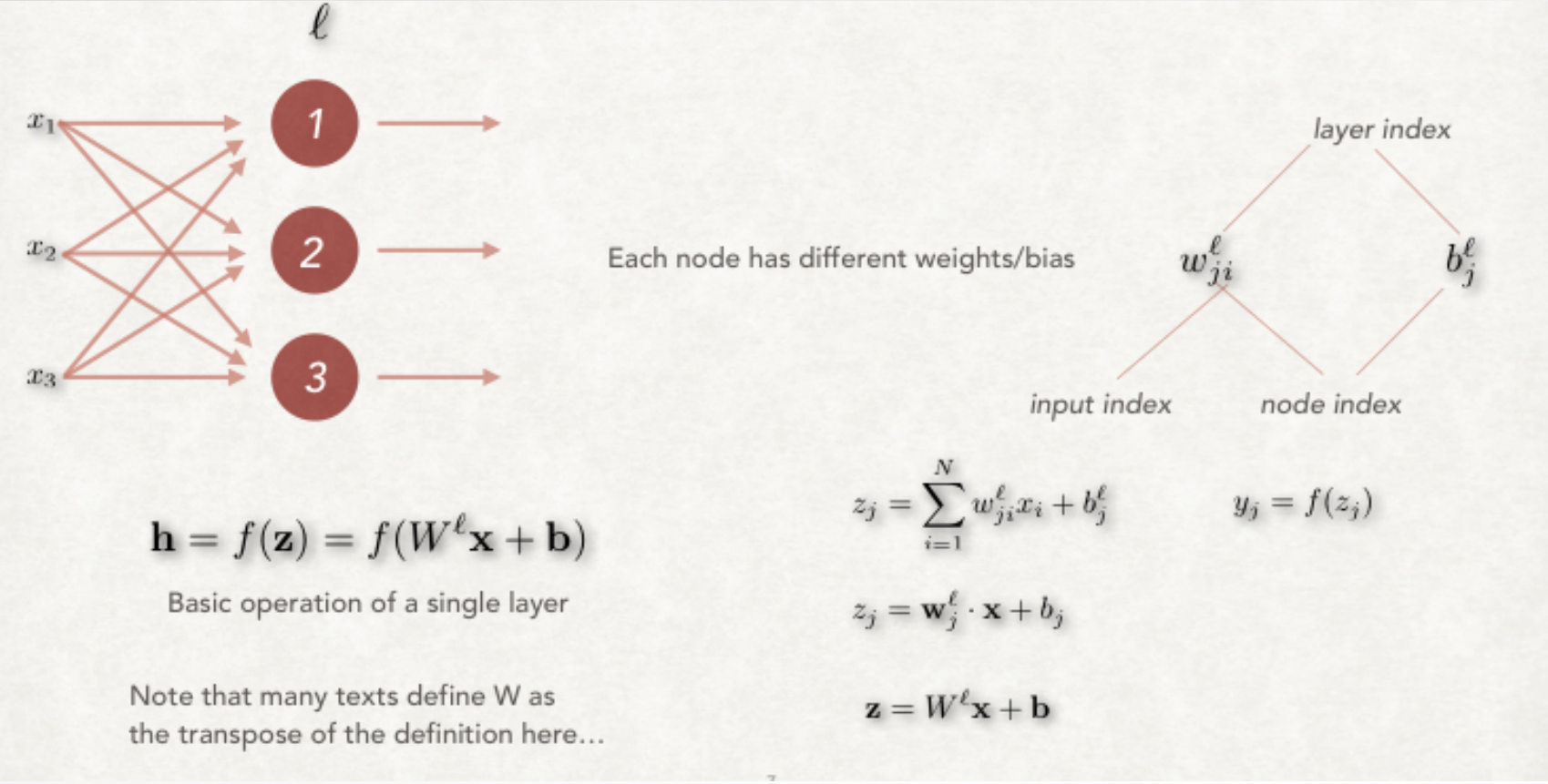
This is the end of the first iteration of the backward propagation. We could continue with the forward propagation, calculate the cost, and then go back to the backward propagation again.

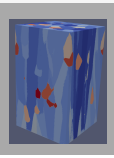




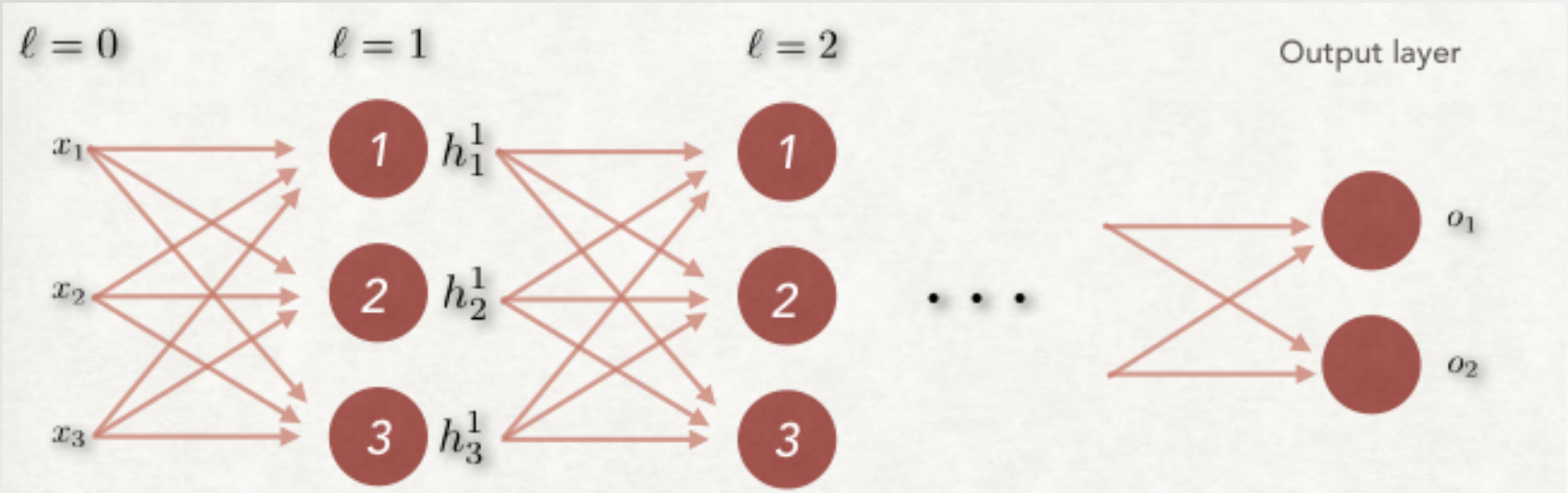
- Logical AND example, P11

Now we have to consider the possibility of multiple layers:

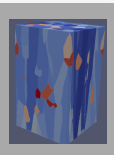




- Multiple Layers



- the activation of one layer is the input for the next layer;
- within each layer, the same activation function is used;
- often, all but the last layer have the same activation function;
- in a fully connected network, all nodes of one layer are connected to each of the nodes of the next layer;
- for a training data set, both input and output are known; all other layers have initially unknown weights and biases, "hidden layers"



- Neural Network as a Set of Connected Layers

- Weights and biases should be initialized with random numbers, not zeroes.
- Each node should have different initial values, otherwise all nodes will continue to have identical weights and biases
- Standard practice is to sample values from a normal distribution centered on 0, with a small variance:

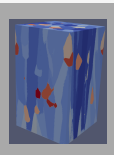
$$\mathcal{N}[\mu, \sigma^2] \rightarrow \mathcal{N}[0, 0.1]$$

Take two random numbers, r and s, uniformly on $[0, 1]$

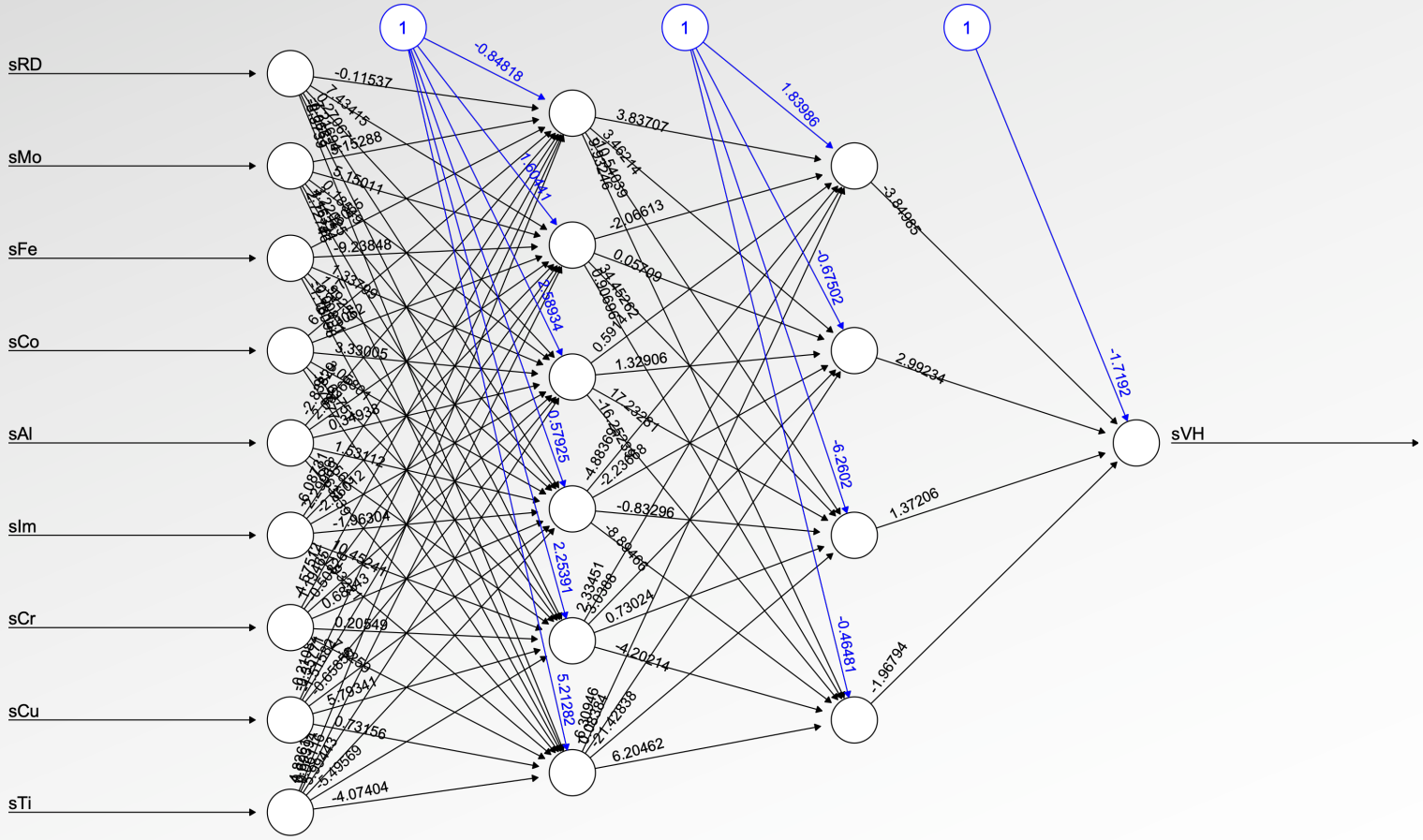
$\rightarrow \sigma\sqrt{-2\ln(r)}\cos(2\pi s)$ is normally distributed around 0 with variance σ^2

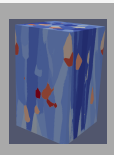
Related to the Box-Muller Transform

$$\text{initial } w^\ell \sim \mathcal{N}\left(0, \sqrt{\frac{2}{n^\ell + n^{\ell-1}}}\right) \qquad \text{Xavier/He initialization}$$



- Network Initialization





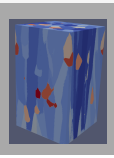
A neural net for regression

```
In[131]:= deriv[ww1_, ww2_, ww3_, ww4_, ww5_, ww6_, bb1_, bb2_, bb3_] :=
  Module[{w1 = ww1, w2 = ww2, w3 = ww3, w4 = ww4, w5 = ww5, w6 = ww6, b1 = bb1,
    b2 = bb2, b3 = bb3},
   h1 = Table[nr[dat1[[i, 2]], dat1[[i, 3]], w1, w2, b1], {i, 4}];
   h2 = Table[nr[dat1[[i, 2]], dat1[[i, 3]], w3, w4, b2], {i, 4}];
   yhat = Table[nr[h1[[i]], h2[[i]], w5, w6, b3], {i, 4}];
   mse = Sum[(yhat[[i]] - dat1[[i, 4]])^2, {i, 4}];
   (* all 9 derivatives *)
   (* dL/dy *)
   df1 = -2 Table[dat1[[j, 4]] - yhat[[j]], {j, 4}];
   (* w5, w6, b3 *)
   dydw5 = Table[h1[[j]] * yhat[[j]] (1 - yhat[[j]]), {j, 4}];
   dydw6 = Table[h2[[j]] * yhat[[j]] (1 - yhat[[j]]), {j, 4}];
   dydb3 = Table[yhat[[j]] (1 - yhat[[j]]), {j, 4}];
   (* w1, w2, b1 *)
   dydh1 = w5 Table[yhat[[j]] (1 - yhat[[j]]), {j, 4}];
   dh1dw1 = Table[dat1[[j, 2]] * h1[[j]] (1 - h1[[j]]), {j, 4}];
   dh1dw2 = Table[dat1[[j, 3]] * h1[[j]] (1 - h1[[j]]), {j, 4}];
   dh1db1 = Table[h1[[j]] (1 - h1[[j]]), {j, 4}];
   (* w3, w4, b2 *)
   dydh2 = w6 Table[yhat[[j]] (1 - yhat[[j]]), {j, 4}];
   dh2dw3 = Table[dat1[[j, 2]] * h2[[j]] (1 - h2[[j]]), {j, 4}];
   dh2dw4 = Table[dat1[[j, 3]] * h2[[j]] (1 - h2[[j]]), {j, 4}];
   dh2db2 = Table[h2[[j]] (1 - h2[[j]]), {j, 4}];
   (* put it together *)
   dfw1 = Sum[df1[[j]] * dydh1[[j]] * dh1dw1[[j]], {j, 4}];
   dfw2 = Sum[df1[[j]] * dydh1[[j]] * dh1dw2[[j]], {j, 4}];
   dfb1 = Sum[df1[[j]] * dydh1[[j]] * dh1db1[[j]], {j, 4}];
   dfw3 = Sum[df1[[j]] * dydh2[[j]] * dh2dw3[[j]], {j, 4}];
   dfw4 = Sum[df1[[j]] * dydh2[[j]] * dh2dw4[[j]], {j, 4}];
   dfb2 = Sum[df1[[j]] * dydh2[[j]] * dh2db2[[j]], {j, 4}];
   dfw5 = Sum[df1[[j]] * dydw5[[j]], {j, 4}];
   dfw6 = Sum[df1[[j]] * dydw6[[j]], {j, 4}];
   dfb3 = Sum[df1[[j]] * dydb3[[j]], {j, 4}];
   {dfw1, dfw2, dfb1, dfw3, dfw4, dfb2, dfw5, dfw6, dfb3}
  ]
```

```
deriv[1, 1, 1, 1, 1, 1, 0, 0, 0]

{0.000095278, 0.0000203596, -0.0000111946, 0.000095278,
 0.0000203596, -0.0000111946, 0.369901, 0.369901, -0.130076}

deriv[-89.17315160846633`, -111.14689665817994`, 161.5282636801397`,
 -154.28119811687068`, 152.52523997462785`, 49.747919848755814`,
 -104.72488085879272`, 164.88552760518735`, -109.98970807864066]

General: Exp[-825.461] is too small to represent as a normalized machine number; precision may be lost.

General: 9.11229352743894396967268482232203 9`12.581633110615407*^-1026 is too small to represent as a normalized machine number; precision may be lost.

General: 2.088607914473337201384117675188010 77`12.800907904196126*^-619 is too small to represent as a normalized machine number; precision may be lost.

General: Further output of General::munfl will be suppressed during this calculation.

{0., 0., 0., 0., 0., 0., 0., 1.89107×10^-52, 1.89107×10^-52}
```

A neural net: derivatives

Numerical optimization



In the stochastic gradient descent method, rather than do **all** the derivatives (referred to as *batch gradient descent*), the true gradient on $L(w)$ is approximated by a gradient taken for a specific data point,

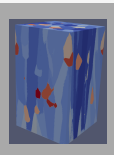
$$w \leftarrow w - \frac{\eta}{n}\sum_{i=1}^{n}\frac{\partial L_i}{\partial w} \Rightarrow w \leftarrow w - \eta\frac{\partial L_i}{\partial w}$$

It will see all parameters in the NN, but restricts the number of derivatives to 1 out of n samples.

As the algorithm sweeps through the training set, it performs the above update for each training example. Several passes can be made over the training set until the algorithm converges. If this is done, the data can be shuffled for each pass to prevent cycles.

- Choose an initial vector of parameters w and learning rate η.
- Repeat until an approximate minimum is obtained:
  - Randomly shuffle examples in the training set.
  - For $i = 1, 2, \ldots, n$, do:
    - $w := w - \eta \nabla Q_i(w)$.

While fast, the gradients are very poorly captured this way.



https://en.wikipedia.org/wiki/Stochastic_gradient_descent

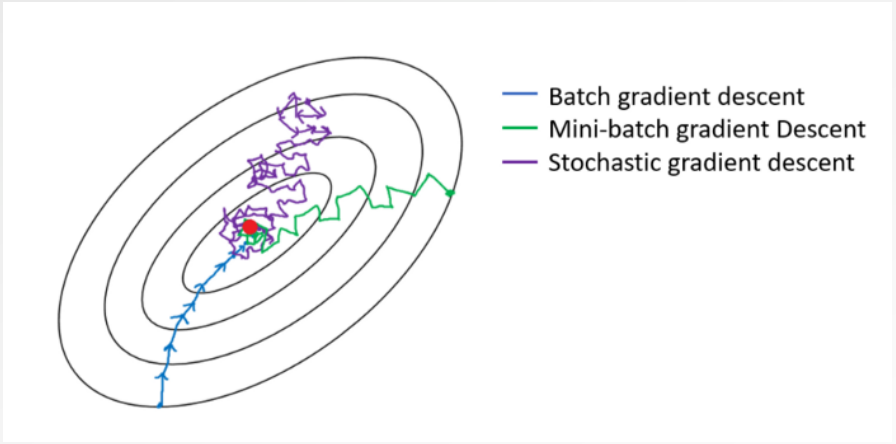
Optimization: Stochastic gradient descent (SGD)

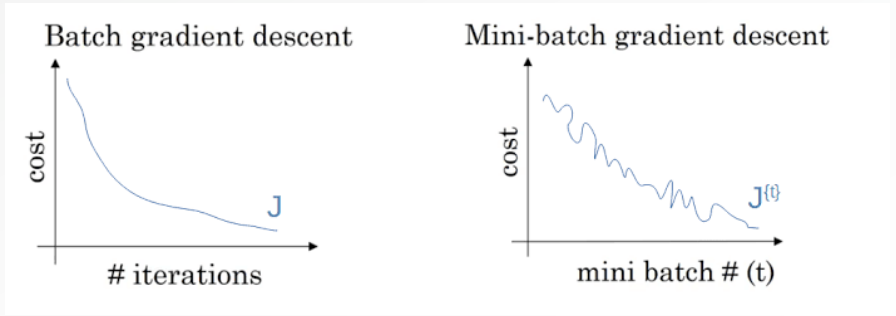
A compromise between computing the true gradient and the gradient at a single example is to compute the gradient against more than one training example (called a "mini-batch") at each step.

For example, if we used a mini-batch based on m samples (data points),

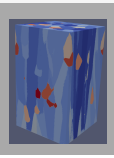
$$w \leftarrow w - \frac{\eta}{n}\sum_{i=1}^{n}\frac{\partial L_i}{\partial w} \Rightarrow w \leftarrow w - \eta\frac{\partial L_i}{\partial w} \Rightarrow w \leftarrow w - \frac{\eta}{m}\sum_{i\in n}^{m}\frac{\partial L_i}{\partial w}$$

Advantage: Typically networks train faster with mini-batches. That's because we update the weights after each propagation.



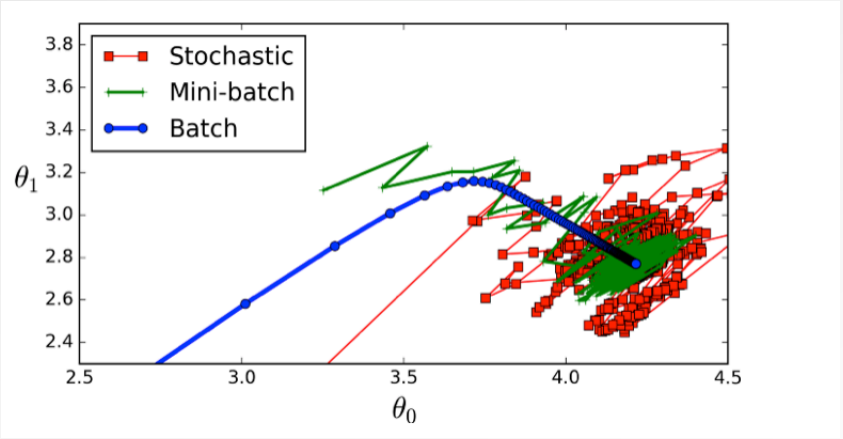


https://medium.com/@sweta.nit/batch-mini-batch-and-stochastic-gradient-descent-e9bc4cacd461

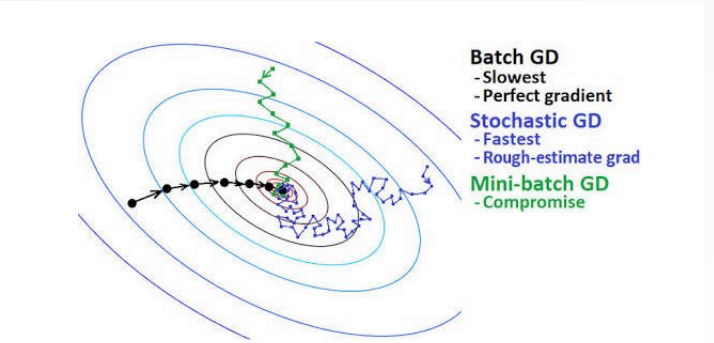


Optimization: Stochastic gradient descent (SGD)

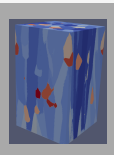
In the figure, the direction of the mini-batch gradient (green color) fluctuates much more in comparison to the direction of the full batch gradient (blue color). The stochastic approach ($m = 1$) leads to gradients that change more often than a mini-batch approach.



"If we compare all three optimizer[s], then every optimizer has its own advantages and disadvantages, [and] we can't come to conclusions [about] which optimizer is best, it totally depends on datasets."

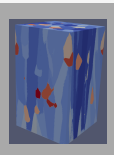


https://medium.com/@sweta.nit/batch-mini-batch-and-stochastic-gradient-descent-e9bc4cacd461



Optimization: Stochastic gradient descent (SGD)

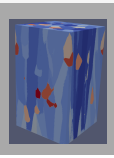
Epochs, Training, Error



An *epoch* is one pass through the network to update the *weights* and *biases*, i.e., a single update of the ANN. It is not feasible to predict how many epochs are required for training. In fact, we have exactly the same problem with ANNs as with any other fitting procedure in the sense of the risk of over-fitting. A compromise must be made between fitting error (*accuracy*) and bias *(validation loss)* in the result.

A reasonable way to proceed is to continue to train (i.e., accumulate *epochs*) until the error no longer decreases. You should split the data into a *training set* and a *test set* (or *validation set*). Cross-validation means repeating the fitting with different splits of the data. *Early stopping* means using the option to stop training once the validation loss starts to increase. *Patience* is a hyperparameter that sets the number of epochs to continue beyond early stopping point. Setting the *ModelCheckpoint callback* (or *restore_best_weights,* which is a boolean value) ensures that the best fit result is retained at the end of the "patience" set of epochs.

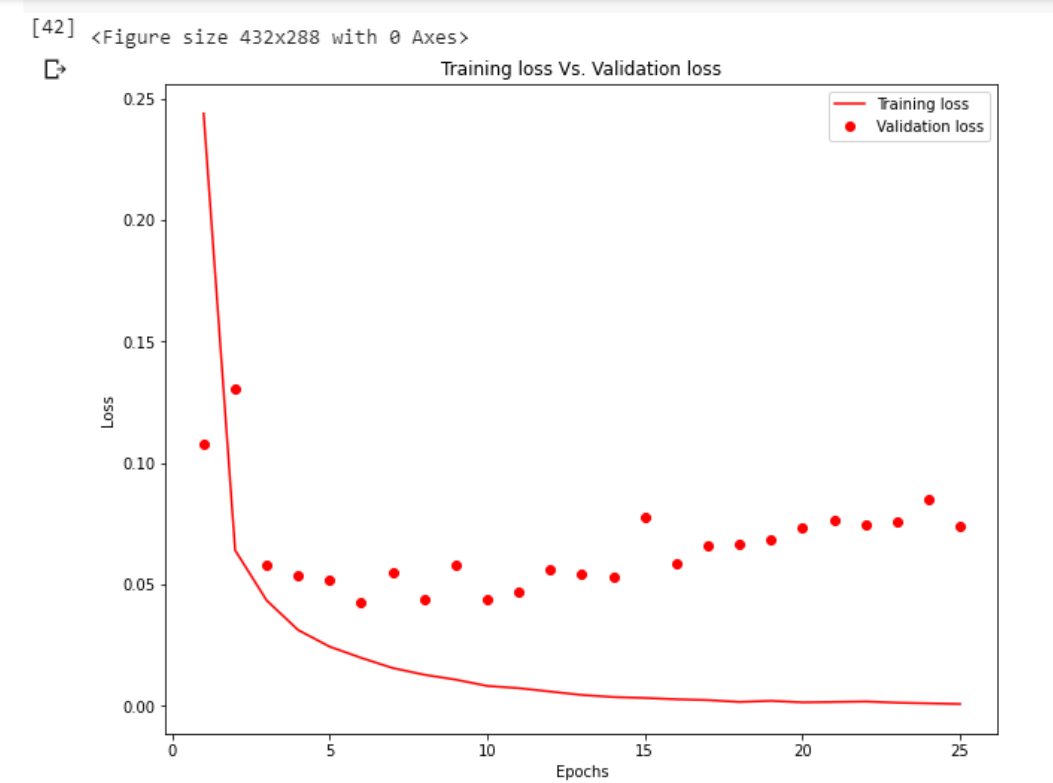
https://www.codespeedy.com/how-to-choose-number-of-epochs-to-train-a-neural-network-in-keras/



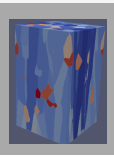
- **Epochs, Early Stopping & Patience**

This illustrates a typical (or hoped-for!) sequence during training.

The analyst should make their own choice as to where to stop. In this example, the *training loss* continues to decrease but ever more slowly. The *validation loss* first drops sharply and then slowly increases. A reasonable compromise is found at about 10 epochs in this example.



https://www.geeksforgeeks.org/choose-optimal-number-of-epochs-to-train-a-neural-network-in-keras/, referencing:
https://keras.io/callbacks/
https://keras.io/datasets/



• Graph of Training vs Validation Error

In earlier slides, we discussed in general terms the concepts of steepest descent and the use of batches.

Provided that you have enough data (not always the case!), you can divide the training data into batches.

The method is known as *mini-batch* when the training set is divided up into subsets of a fixed size. The batch size is commonly chosen to be 32, 64, 128 etc., depending (as always) on the particular problem.

Batch gradient descent is the same as the base method except that (to speed up the process), each batch (sample) is evaluated separately and the update is an average of the ensemble.

Stochastic gradient descent adds noise as it proceeds (just as we have seen).

Mini-batch gradient descent is the same as the above but with a fixed (small) batch size.

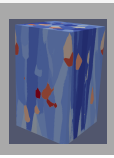
For big data problems, these approaches are forced on us by memory limitations and compute times.

```
Pseudo-code:

model = initialization(...)
n_epochs = ...
train_data = ...
for i in n_epochs:
    train_data = shuffle(train_data)
    X, y = split(train_data)
    predictions = predict(X, model)
    error = calculate_error(y, predictions)
    model = update_model(model, error)
```

This website has a nice example, with code, of these concepts applied to the MNIST dataset.

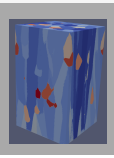
https://medium.com/@elimu.michael9/understanding-epochs-and-batches-23120a04b3cb

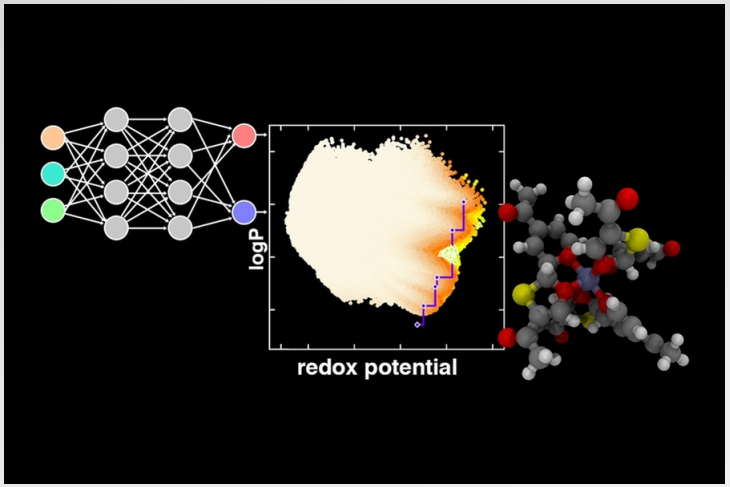
https://machinelearningmastery.com/difference-between-a-batch-and-an-epoch/
https://machinelearningmastery.com/gentle-introduction-mini-batch-gradient-descent-configure-batch-size/



• Batch Size

Final comments for the intro to NN





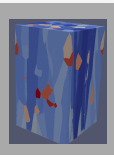
An iterative, multi-step process for training a neural network, as depicted at top left, leads to an assessment of the tradeoffs between two competing qualities, as depicted in graph at center. The blue line represents a so-called Pareto front, defining the cases beyond which the materials selection cannot be further improved. This makes it possible to identify specific categories of promising new materials, such as the one depicted by the molecular diagram at right.

Neural networks facilitate optimization in the search for new materials

Sorting through millions of possibilities, a search for battery materials delivered results in five weeks instead of 50 years.

"As a demonstration, the team arrived at a set of the eight most promising materials, out of nearly 3 million candidates, for an energy storage system called a flow battery. This culling process would have taken 50 years by conventional analytical methods, they say, but they accomplished it in five weeks."

https://news.mit.edu/2020/neural-networks-optimize-materials-search-0326

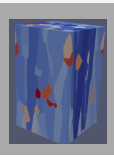


Neural nets in materials: an example

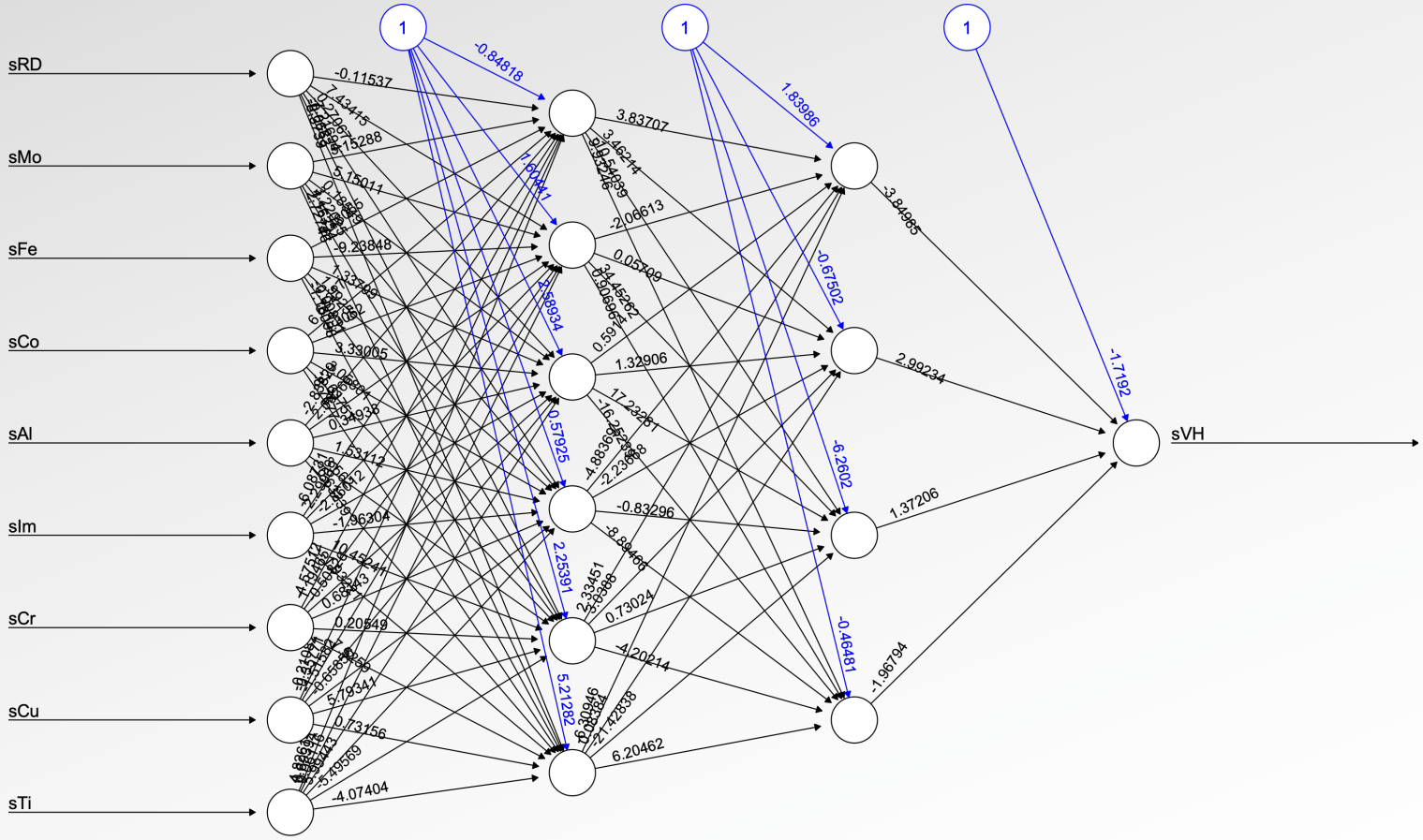
"Neural networks are more flexible and can be used with both regression and classification problems. Neural networks are good for the nonlinear dataset with a large number of inputs such as images. Neural networks can work with any number of inputs and layers. Neural networks have the numerical strength that can perform jobs in parallel."

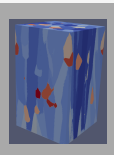
"There are more alternative algorithms such as SVM, Decision Tree and Regression are available that are simple, fast, easy to train, and provide better performance. ***Neural networks are much more of a black box, require more time for development and more computation power.*** Neural Networks requires more data than other Machine Learning algorithms. NNs can be used only with numerical inputs and non-missing value datasets. **A well-known neural network researcher said "A neural network is the second best way to solve any problem. The best way is to actually understand the problem."**

https://www.datacamp.com/community/tutorials/neural-network-models-r



Neural nets: comments

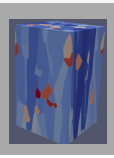




A neural net for regression

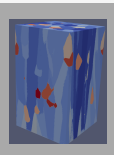
Homework 7

Use of ANNs, Comparison with Random Forest

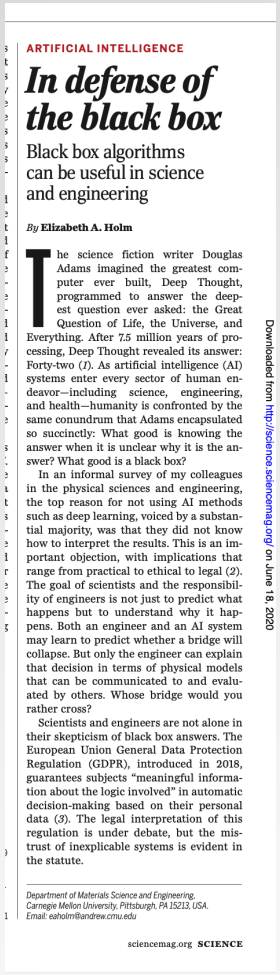


The last (!) homework (HW7) will be an exercise on implementing an ANN for regression

- Answer general questions about neural nets
- Check that your python/conda/anaconda setup works properly for the ANN Notebook provided based on this website: https://machinelearningmastery.com/neural-network-models-for-combined-classification-and-regression/
- Optimize the regression ANN to minimize MSE
- Re-run the regression with a different error measure; compare to the MSE method
- Re-run the regression with a different activation function; compare to the first result
- Compare both ANN results to your own best Random Forest result, both graphically and in terms of the error measure
- Without having a copy of the Notebook available, follow the guidance of the website and implement your own classification ANN
- Also compare your classification result with your own clustering analysis.



Neural nets: comments



ARTIFICIAL INTELLIGENCE

In defense of the black box

Black box algorithms can be useful in science and engineering

By **Elizabeth A. Holm**

The science fiction writer Douglas Adams imagined the greatest computer ever built, Deep Thought, programmed to answer the deepest question ever asked: the Great Question of Life, the Universe, and Everything. After 7.5 million years of processing, Deep Thought revealed its answer: Forty-two (*1*). As artificial intelligence (AI) systems enter every sector of human endeavor—including science, engineering, and health—humanity is confronted by the same conundrum that Adams encapsulated so succinctly: What good is knowing the answer when it is unclear why it is the answer? What good is a black box?

In an informal survey of my colleagues in the physical sciences and engineering, the top reason for not using AI methods such as deep learning, voiced by a substantial majority, was that they did not know how to interpret the results. This is an important objection, with implications that range from practical to ethical to legal (*2*). The goal of scientists and the responsibility of engineers is not just to predict what happens but to understand why it happens. Both an engineer and an AI system may learn to predict whether a bridge will collapse. But only the engineer can explain that decision in terms of physical models that can be communicated to and evaluated by others. Whose bridge would you rather cross?

Scientists and engineers are not alone in their skepticism of black box answers. The European Union General Data Protection Regulation (GDPR), introduced in 2018, guarantees subjects "meaningful information about the logic involved" in automatic decision-making based on their personal data (*3*). The legal interpretation of this regulation is under debate, but the mistrust of inexplicable systems is evident in the statute.

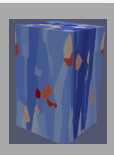
Department of Materials Science and Engineering, Carnegie Mellon University, Pittsburgh, PA 15213, USA. Email: eaholm@andrew.cmu.edu

"The science fiction writer Douglas Adams imagined the greatest computer ever built, Deep Thought, programmed to answer the deepest question ever asked: the Great Question of Life, the Universe, and Everything. After 7.5 million years of processing, Deep Thought revealed its answer:" **42**

"The first and most obvious case for using a black box is when the cost of a wrong answer is low relative to the value of a correct answer. …" Her example is image segmentation for which AI is good, but not perfect. "Perfection is not, however, necessary to make this system useful because the cost of a few disputed pixels is low compared with saving the time and sanity of belabored graduate students."

"The second case for the black box is equally obvious but more fraught. A black box can and should be used when it produces the best results." Her example is that AI enhances the ability of radiologists at detecting cancers in medical images. While the consequences of a misidentification are high, the "black-box" still offers the best solution (and are checked by a radiologist).

Holm, "In defense of the black box," Science 364, 26 (2019) (on Canvas)



Interesting paper

Questions?