



Inteligencia Artificial

UTN – FRVM

5º Año Ing. en Sistemas de
Información

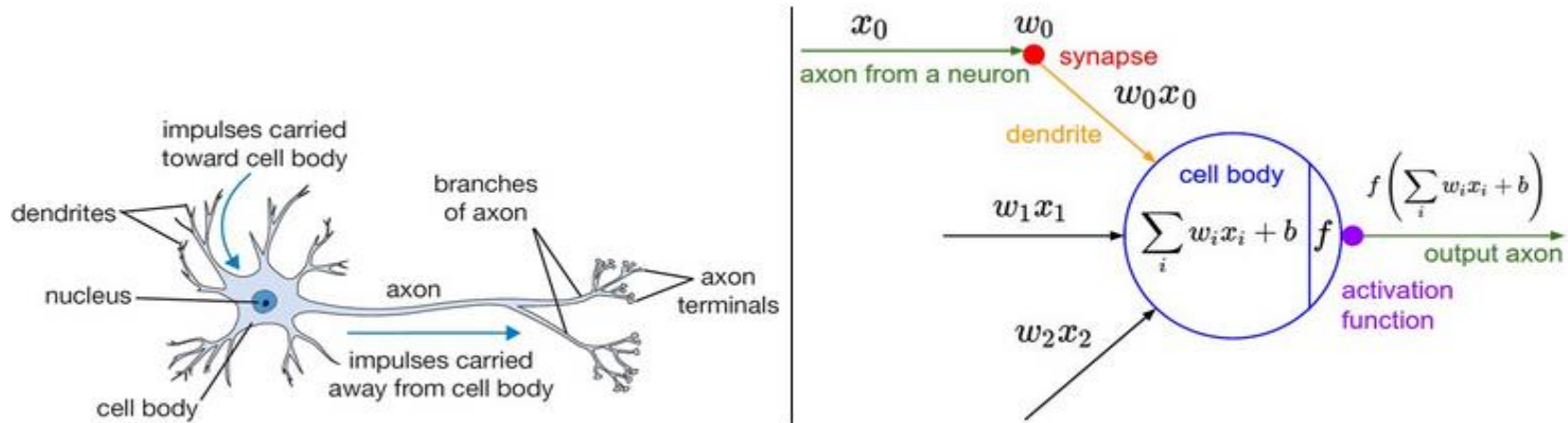


Agenda



- Neural Networks
 - Activation Functions
 - Architecture
 - Gradient Descent

Biological motivation and connections



A cartoon drawing of a biological neuron (left) and its mathematical model (right).

Each neuron performs a dot product with the input and its weights, adds the bias and applies the non-linearity (or activation function), in this case the sigmoid

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

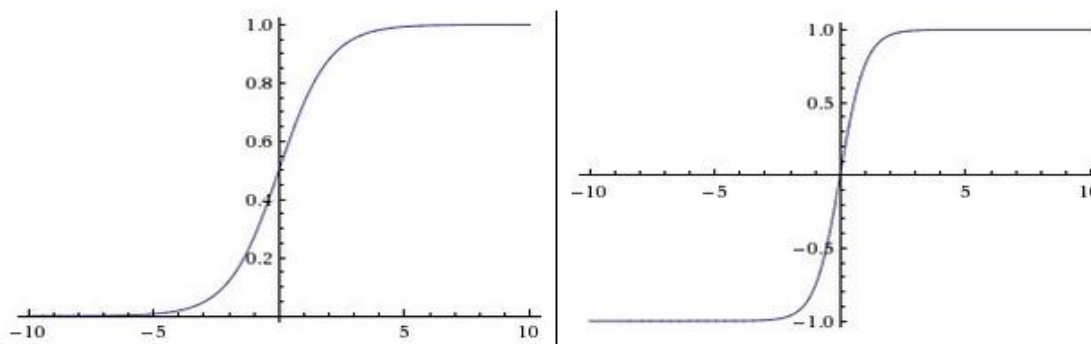
Coarse model?

<https://www.sciencedirect.com/science/article/pii/S0959438814000130>

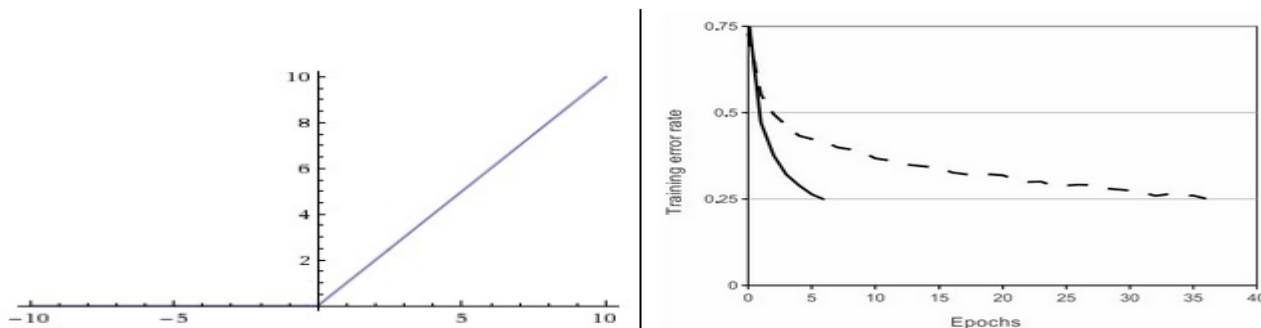
https://neuropsychics.ucsd.edu/courses/physics_171/annurev.neuro.28.061604.135703.pdf

Commonly used activation functions

- Every activation function (or *non-linearity*) takes a single number and performs a certain fixed mathematical operation on it. There are several activation functions in practice:



Left: Sigmoid non-linearity squashes real numbers to range between $[0,1]$ **Right:** The tanh non-linearity squashes real numbers to range between $[-1,1]$.



Left: Rectified Linear Unit (ReLU) activation function, which is zero when $x < 0$ and then linear with slope 1 when $x > 0$. **Right:** A plot from [Krizhevsky et al. \(pdf\)](#) paper indicating the 6x improvement in convergence with the ReLU unit compared to the tanh unit.



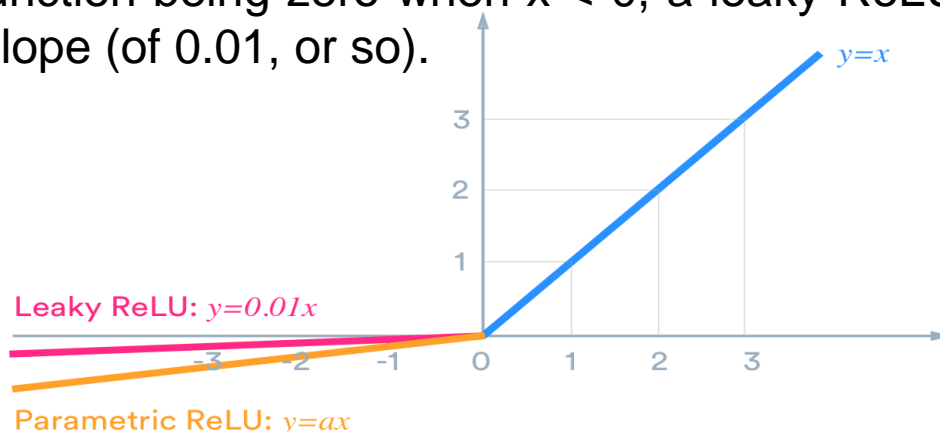
Activation functions

- **Sigmoid.** The sigmoid non-linearity has the mathematical form $\sigma(x) = 1/(1 + e^{-x})$
 - *Sigmoids saturate and kill gradients*
 - *Sigmoid outputs are not zero-centered*
- The **tanh** non-linearity squashes a real-valued number to the range $[-1, 1]$. Like the sigmoid neuron, its activations saturate, but unlike the sigmoid neuron its output is zero-centered. The tanh neuron is simply a scaled sigmoid neuron, in particular the following holds: $\tanh(x) = 2\sigma(2x) - 1$.
- **ReLU.** The Rectified Linear Unit has become very popular in the last few years. It computes the function $f(x) = \max(0, x)$. In other words, the activation is simply thresholded at zero. There are several pros and cons to using the ReLUs.

(+) It was found to greatly accelerate the convergence of stochastic gradient descent compared to the sigmoid/tanh functions.
(+) Compared to tanh/sigmoid neurons that involve expensive operations (exponentials, etc.), the ReLU can be implemented by simply thresholding a matrix of activations at zero.
(-) Unfortunately, ReLU units can be fragile during training and can “die”.

Leaky ReLU/Maxout

- **Leaky ReLU.** Leaky ReLUs are one attempt to fix the “dying ReLU” problem. Instead of the function being zero when $x < 0$, a leaky ReLU will instead have a small negative slope (of 0.01, or so).



- **Maxout.** One relatively popular choice is the *Maxout* neuron that generalizes the ReLU and its leaky version. The Maxout neuron computes the function

$$\max(w_1^T x + b_1, w_2^T x + b_2).$$

- Notice that both ReLU and Leaky ReLU are a special case of this form (for example, for ReLU we have $w_1, b_1=0$). The Maxout neuron therefore enjoys all the benefits of a ReLU unit (linear regime of operation, no saturation) and does not have its drawbacks (dying ReLU). However, unlike the ReLU neurons it doubles the number of parameters for every single neuron, leading to a high total number of parameters.



ReLU vs. Other Functions

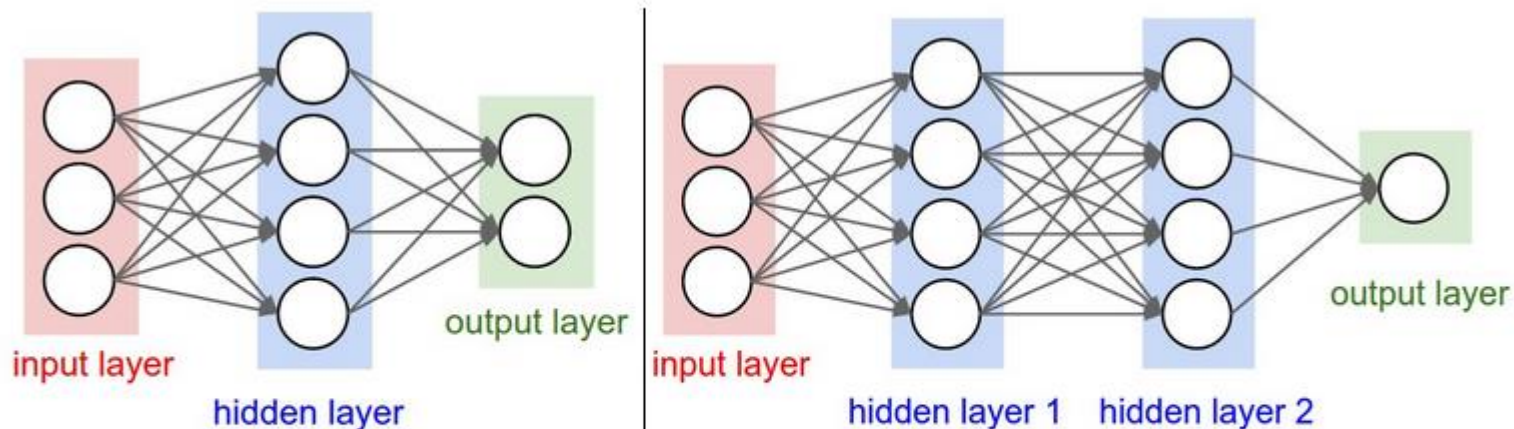
- Use the ReLU non-linearity, be careful with the learning rates and possibly monitor the fraction of “dead” units in a network. Then, give Leaky ReLU or Maxout a try. Never use sigmoid. Try tanh, but expect it to work worse than ReLU/Maxout.



Neural Network architectures: Layer-wise organization

- **Neural Networks as neurons in graphs.** Neural Networks are modeled as collections of neurons that are connected in an acyclic graph.
- In other words, the outputs of some neurons can become inputs to other neurons. Cycles are not allowed since that would imply an infinite loop in the forward pass of a network.
- Instead of an amorphous blobs of connected neurons, Neural Network models are often organized into distinct layers of neurons.
- For regular neural networks, the most common layer type is the **fully-connected layer** in which neurons between two adjacent layers are fully pairwise connected, but neurons within a single layer share no connections.

NN Architectures



Left: A 2-layer Neural Network (one hidden layer of 4 neurons (or units) and one output layer with 2 neurons), and three inputs.

Right: A 3-layer neural network with three inputs, two hidden layers of 4 neurons each and one output layer. Notice that in both cases there are connections (synapses) between neurons across layers, but not within a layer.

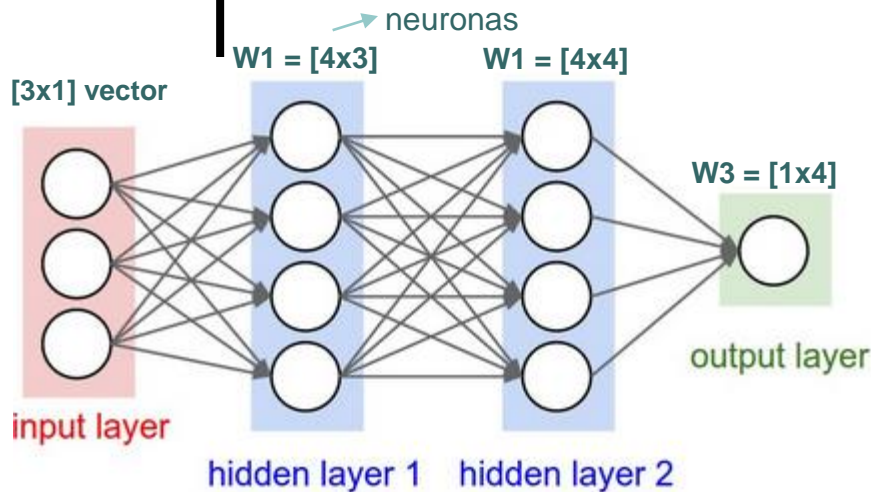
Output layer. Unlike all layers in a Neural Network, the output layer neurons most commonly do not have an activation function (or you can think of them as having a linear identity activation function). This is because the last output layer is usually taken to represent the class scores (e.g. in classification), which are arbitrary real-valued numbers, or some kind of real-valued target (e.g. in regression).



Sizing neural networks

- The two metrics commonly used to measure the size of neural networks are the number of neurons, or more commonly the number of parameters. Working with the two example networks:
- The first network (left) has $4 + 2 = 6$ neurons (not counting the inputs), $[3 \times 4] + [4 \times 2] = 20$ weights and $4 + 2 = 6$ biases, for a total of 26 learnable parameters.
- The second network (right) has $4 + 4 + 1 = 9$ neurons, $[3 \times 4] + [4 \times 4] + [4 \times 1] = 12 + 16 + 4 = 32$ weights and $4 + 4 + 1 = 9$ biases, for a total of 41 learnable parameters.
- To give context, modern Convolutional Networks contain on orders of 100 million parameters and are usually made up of approximately 10-20 layers (hence *deep learning*). However, as we will see the number of *effective* connections is significantly greater due to parameter sharing

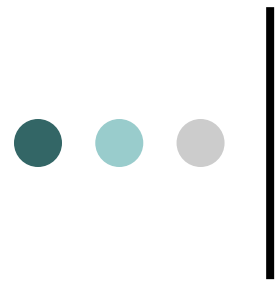
Example feed-forward computation



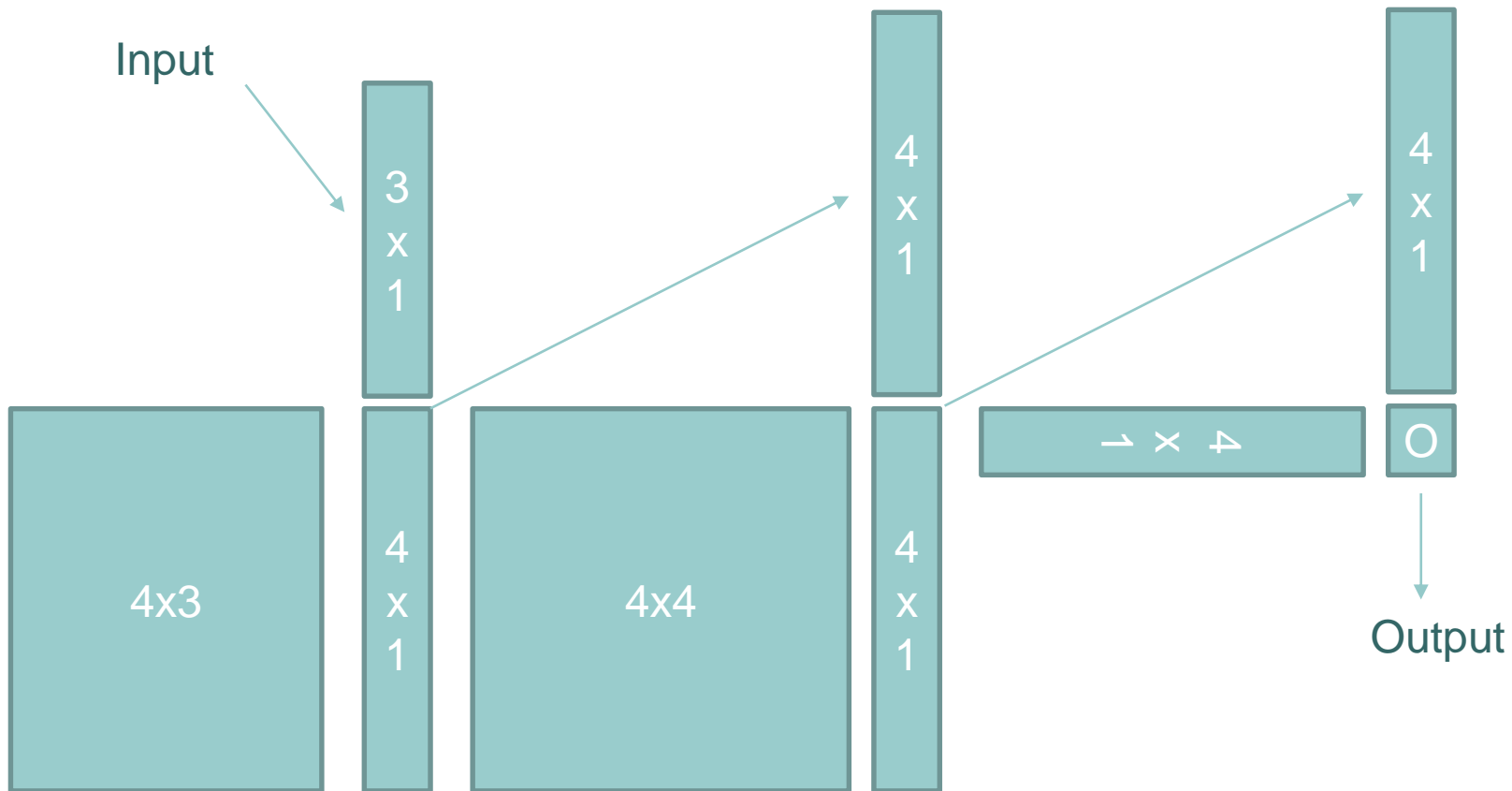
```
# forward-pass of a 3-layer neural network:  
f = lambda x: 1.0/(1.0 + np.exp(-x)) # activation function (use sigmoid)  
x = np.random.randn(3, 1) # random input vector of three numbers (3x1)  
h1 = f(np.dot(W1, x) + b1) # calculate first hidden layer activations (4x1)  
h2 = f(np.dot(W2, h1) + b2) # calculate second hidden layer activations (4x1)  
out = np.dot(W3, h2) + b3 # output neuron (1x1)
```

parameters

- One of the primary reasons that Neural Networks are organized into layers is that this structure makes it very simple and efficient to evaluate Neural Networks using matrix vector operations.
- Working with the example above, the input would be a **$[3 \times 1]$ vector**. All connection strengths for a layer can be stored in a single matrix.
- The first hidden layer's weights **$W1$ would be of size $[4 \times 3]$** , and the biases for all units would be in the vector **$b1$** , of size **$[4 \times 1]$** . Here, every single neuron has its weights in a row of **$W1$** , so the matrix vector multiplication $\text{np.dot}(W1, x)$ evaluates the activations of all neurons in that layer. Similarly, **$W2$** would be a **$[4 \times 4]$ matrix** that stores the connections of the second hidden layer, and **$W3$** a **$[1 \times 4]$ matrix** for the last (output) layer. The full **forward pass** of this 3-layer neural network is then simply three matrix multiplications, interwoven with the application of the activation function:



Input



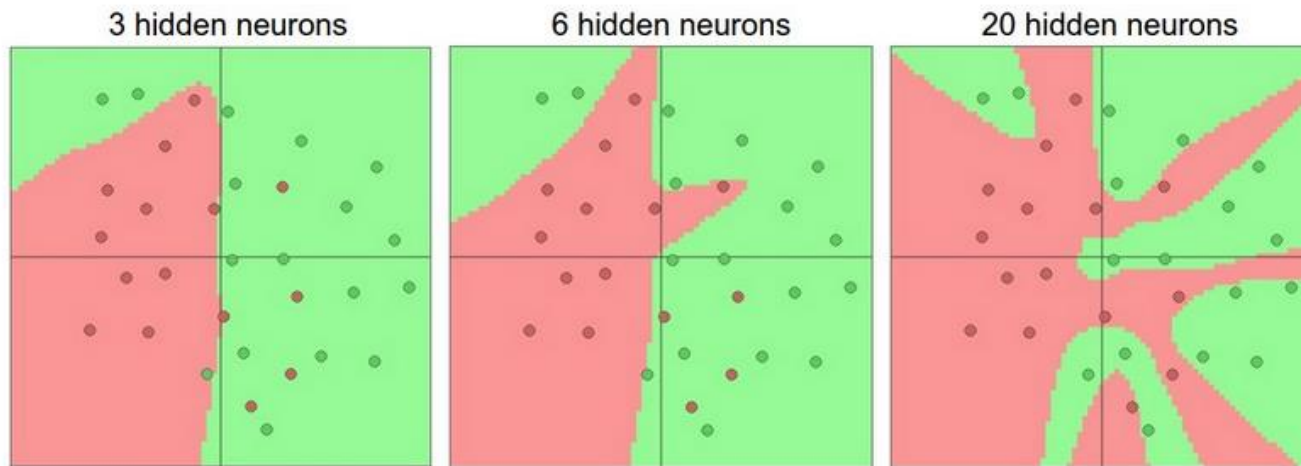


Representational power

- Neural Networks with fully-connected layers define a family of functions that are parameterized by the weights of the network.
- Neural Networks with at least one hidden layer are universal approximators.
- It can be shown (e.g. see Approximation by Superpositions of Sigmoidal Function from 1989), that given any continuous function $f(x)$ and some $\epsilon > 0$, there exists a Neural Network $g(x)$ with one hidden layer (with a reasonable choice of non-linearity, e.g. sigmoid) such that $\forall x, |f(x) - g(x)| < \epsilon$. In other words, the neural network can approximate any continuous function.

Setting number of layers and their sizes

- How do we decide on what architecture to use when faced with a practical problem? Should we use no hidden layers? One hidden layer? Two hidden layers? How large should each layer be?
- First, note that as we increase the size and number of layers in a Neural Network, the **capacity** of the network increases. That is, the space of representable functions grows since the neurons can collaborate to express many different functions.
- For example, suppose we had a binary classification problem in two dimensions. We could train three separate neural networks, each with one hidden layer of some size and obtain the following classifiers:



Larger Neural Networks can represent more complicated functions. The data are shown as circles colored by their class, and the decision regions by a trained neural network are shown underneath.



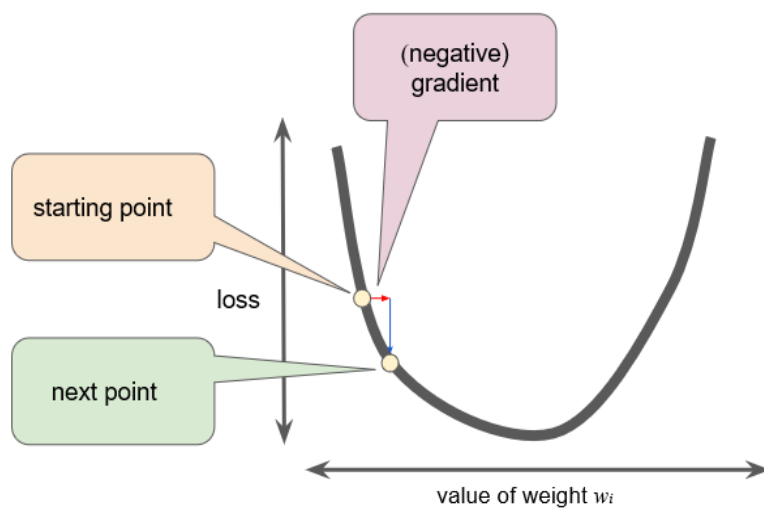
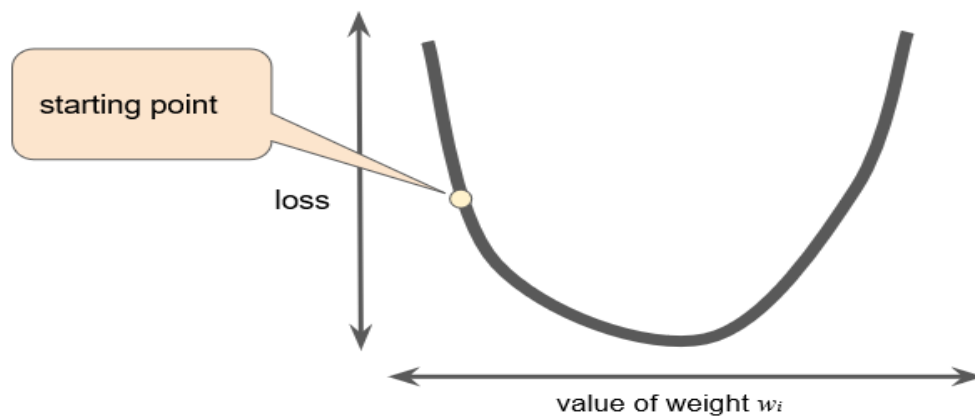
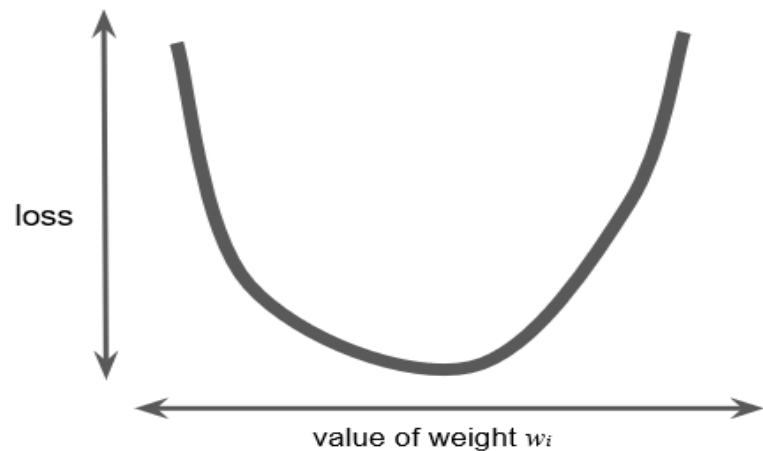
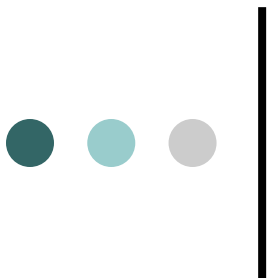
Neural Network Learning as Optimization

- A deep learning neural network learns to map a set of inputs to a set of outputs from training data.
- We cannot calculate the perfect weights for a neural network; Instead, the problem of learning is cast as a search or optimization problem
- An algorithm is used to navigate the space of possible sets of weights the model may use in order to make good or good enough predictions.



Neural Network Learning as Optimization


- Typically, a neural network model is trained using the stochastic **gradient descent optimization** algorithm and weights are updated using the backpropagation of error algorithm.
- The “*gradient*” in gradient descent refers to an error gradient. The model with a given set of weights is used to make predictions and **the error for those predictions is calculated**.
- The **gradient descent algorithm seeks to change the weights so that the next evaluation reduces the error**, meaning the optimization algorithm is navigating down the gradient (or slope) of error.



Gradient Descent

- Computing the gradient numerically with finite differences

```
def eval_numerical_gradient(f, x):  
    """  
    a naive implementation of numerical gradient of f at x  
    - f should be a function that takes a single argument  
    - x is the point (numpy array) to evaluate the gradient at  
    """  
  
    fx = f(x) # evaluate function value at original point  
    grad = np.zeros(x.shape)  
    h = 0.00001  
  
    # iterate over all indexes in x  
    it = np.nditer(x, flags=['multi_index'], op_flags=['readwrite'])  
    while not it.finished:  
  
        # evaluate function at x+h  
        ix = it.multi_index  
        old_value = x[ix]  
        x[ix] = old_value + h # increment by h  
        fxh = f(x) # evaluate f(x + h)  
        x[ix] = old_value # restore to previous value (very important!)  
  
        # compute the partial derivative  
        grad[ix] = (fxh - fx) / h # the slope  
        it.iternext() # step to next dimension  
  
    return grad
```


$$\frac{df(x)}{dx} = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}$$



Gradient Descent

- Following the gradient formula, the code above iterates over all dimensions one by one, makes a small change h along that dimension and calculates the partial derivative of the loss function along that dimension by seeing how much the function changed. The variable `grad` holds the full gradient in the end.

Step Size

- **Effect of step size.** The gradient tells us the direction in which the function has the steepest rate of increase, but it does not tell us how far along this direction we should step.
- Choosing the step size (also called the **learning rate**) will become one of the most important (and most headache-inducing) hyperparameter settings in training a neural network.
- **Very small progress** (this corresponds to having a small step size) VS **descend faster**, but this may not pay off.

```
def CIFAR10_loss_fun(W):
    return L(X_train, Y_train, W)

W = np.random.rand(10, 3073) * 0.001 # random weight vector
df = eval_numerical_gradient(CIFAR10_loss_fun, W) # get the gradient

loss_original = CIFAR10_loss_fun(W) # the original loss
print 'original loss: %f' % (loss_original, )

# lets see the effect of multiple step sizes
for step_size_log in [-10, -9, -8, -7, -6, -5, -4, -3, -2, -1]:
    step_size = 10 ** step_size_log
    W_new = W - step_size * df # new position in the weight space
    loss_new = CIFAR10_loss_fun(W_new)
    print 'for step size %f new loss: %f' % (step_size, loss_new)

# prints:
# original loss: 2.200718
# for step size 1.000000e-10 new loss: 2.200652
# for step size 1.000000e-09 new loss: 2.200057
# for step size 1.000000e-08 new loss: 2.194116
# for step size 1.000000e-06 new loss: 1.647802
# for step size 1.000000e-05 new loss: 2.844355
# for step size 1.000000e-04 new loss: 25.558142
# for step size 1.000000e-03 new loss: 254.086573
# for step size 1.000000e-02 new loss: 2539.370888
# for step size 1.000000e-01 new loss: 25392.214036
```