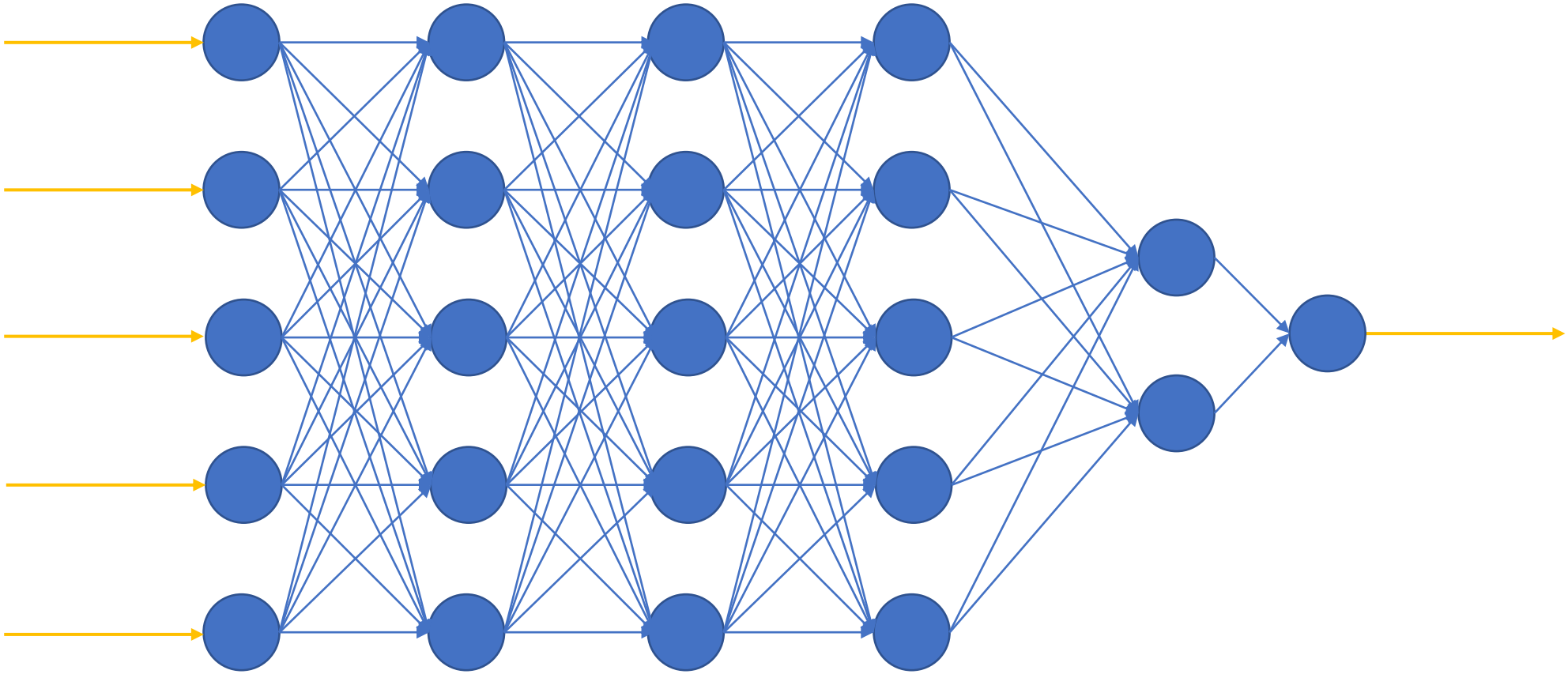
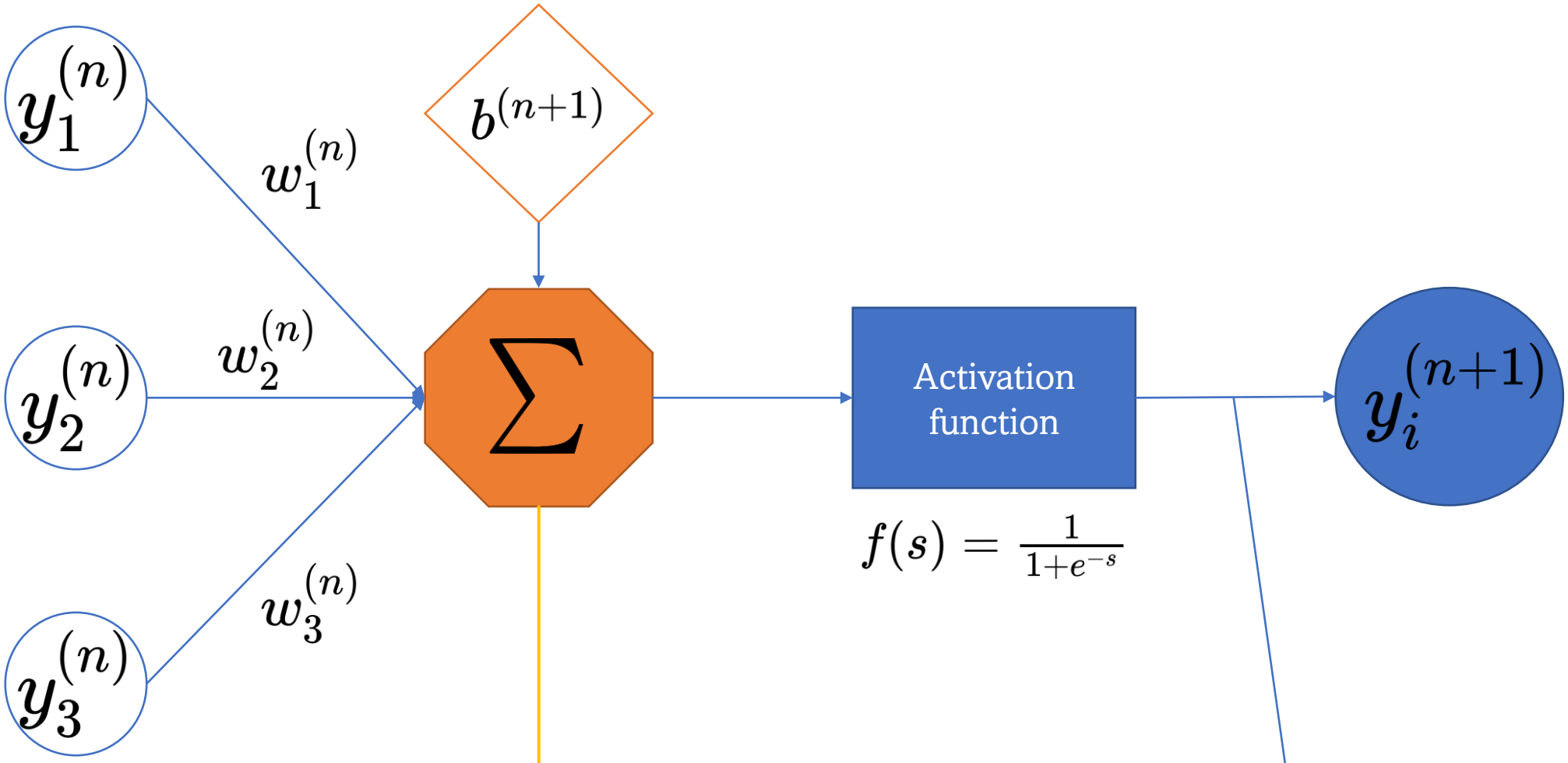
Artificial neural network: multilayer perceptron used as a classifier

(supervised learning)

Feed-forward neural network



Basic structure



Neural networks as function approximators

ANN can be seen as function approximators

$$F(x) \longrightarrow F_\theta(x) \qquad \theta = (\theta_1, \theta_2, \ldots, \theta_N)$$

We would like to measure the deviation between the network output and the function it is trying to approximate

$$C_x(\theta) = |F_\theta(x) - F(x)|^2$$

SAMPLE-SPECIFIC COST FUNCTION

But depending on your problem there are also other cost functions…

How the ANNs learn

We want to measure and minimize the “distance” between the output of the net and the result we would obtain

→<u>supervised learning</u>

During training the sample cost function will be averaged over all sets of training samples

$$C_x(\theta) = |F_\theta(x) - F(x)|^2 \qquad \rightarrow \qquad C(\theta) = \left\langle C_x(\theta) \right\rangle_x$$

How the ANNs learn: gradient descent

We can update the values of the parameters “sliding down the hill” that means updating parameters by steps

$$\delta\theta_k = -\epsilon \frac{\partial C(\theta)}{\partial \theta_k} \qquad \delta C = -\epsilon \left(\frac{\partial C(\theta)}{\partial \theta_k} \right)^2 + o\left(\epsilon^2\right)$$

where ϵ is what is called **learning rate**

Since the cost function is defined as the average over ALL possible inputs it can be too expensive to compute

Solution is **stochastic gradient descent**, or training over batches!

$$C(\theta) \sim \frac{1}{N} \sum_{j=1}^{N} C_{x_j}(\theta) \qquad \delta\theta_k = -\epsilon \left\langle \frac{\partial C_x(\theta)}{\partial \theta_k} \right\rangle_{batch}$$

How the ANNs learn: backpropagation

Since numerical differentiation is inefficient due to the large number of parameters we use the **chain rule**

For the cost function we defined before, we have

$$\frac{\partial C_x(\theta)}{\partial \theta_k} = 2\sum_s \left(\left(F_\theta(x)\right)_s - \left(F(x)\right)_s \right) \frac{\partial \left(F_\theta(x)\right)_s}{\partial \theta_k}$$

where $\left(F_\theta(x)\right)_s = y_s^{(N)}$ is the value of the s neuron of the output layer N and $\left(F(x)\right)_s$ the desired output for that neuron

How the ANNs learn: backpropagation

We have now to calculate the gradient of a neuron value with respect to any of the parameters

$$\frac{\partial y_s^{(n+1)}}{\partial \theta_k} = f'\left(z_s^{(n+1)}\right) \frac{\partial z_m^{(n+1)}}{\partial \theta_k} \longrightarrow \frac{\partial C_x(\theta)}{\partial \theta_k} = 2 \sum_s \left(y_s^{(N)} - (F(x))_s\right) f'\left(z_s^{(N)}\right) \frac{\partial z_s^{(N)}}{\partial \theta_k}$$

$$\frac{\partial z_s^{(n+1)}}{\partial \theta_k} = \sum_m w_{sm}^{(n+1,n)} \frac{\partial y_m^{(n)}}{\partial \theta_k}$$

It is quite evident the recursive structure. By defining the matrix element (in a total general way)

$$M_{sm}^{(n+1,n)} = w_{sm}^{(n+1,n)} f'\left(z_m^{(n)}\right)$$

it is possible to show that holds

$$\frac{\partial z_s^{(n)}}{\partial \theta_k} = \left[M^{(n,n-1)} M^{(n-1,n-2)} \dots M^{(\tilde{n}+1,\tilde{n})} \frac{\partial z^{(\tilde{n})}}{\partial \theta_k} \right]_s$$

where ñ is the layer where we want to update the parameter θ_k so where $\frac{\partial z_s^{(\tilde{n})}}{\partial w_{sm}^{(\tilde{n},\tilde{n}-1)}} = y_m^{(\tilde{n}-1)}$.

How the ANNs learn: backpropagation

The backpropagation algorithm can be summarized in:

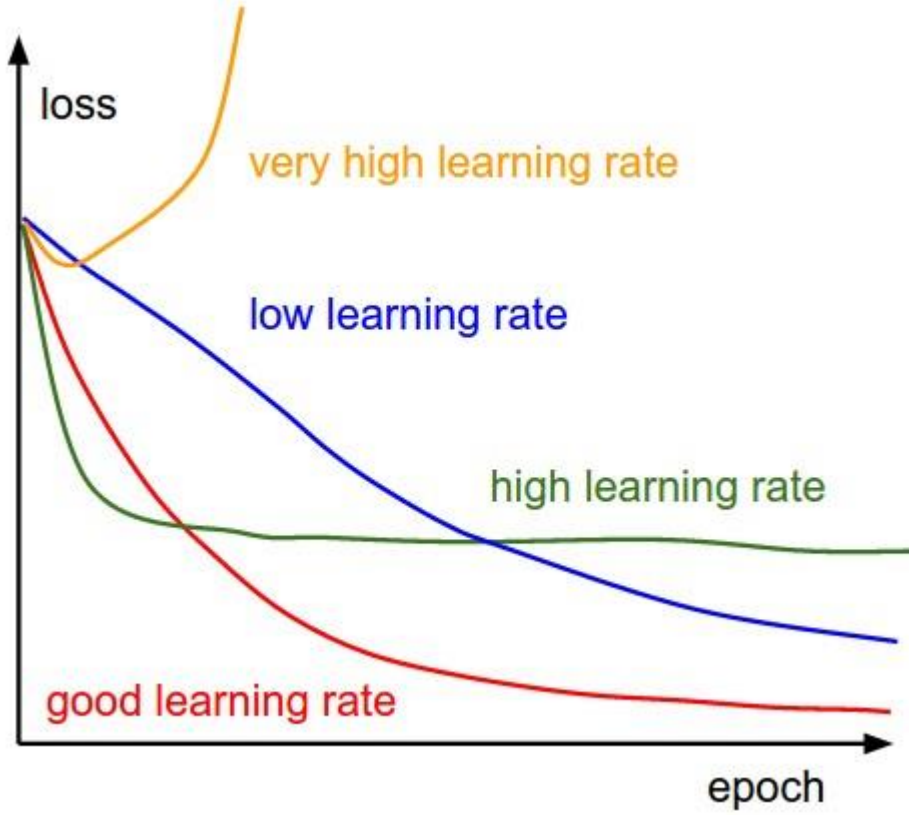
- Initialize a “deviation” vector at the output layer N $\Delta_j = \left(y_j^{(N)} - F(x)_j\right) f'\left(z_j^{(N)}\right)$ so proportional to the difference between the current *j*-th output and the expected *j*-th output
- For each layer starting at $n=N$ store the derivative with respect to the parameters at that layer: $\frac{\partial C_x(\theta)}{\partial \theta_k}$ for all θ_k explicitly occurring in $z_j^{(n)}$
- Step down to the next lower layer by setting $\Delta_k^{(new)} = \sum_j \Delta_j M_{jk}^{(n,n-1)}$ and repeat after storing derivatives

In this way all the derivatives will be known

How to choose the learning rate?

The choice of the learning rate depends from the shape of the cost (loss) function. The cost function give us an idea on how the net is actually learning and we have to manually adjust the learning rate to maximize learning.

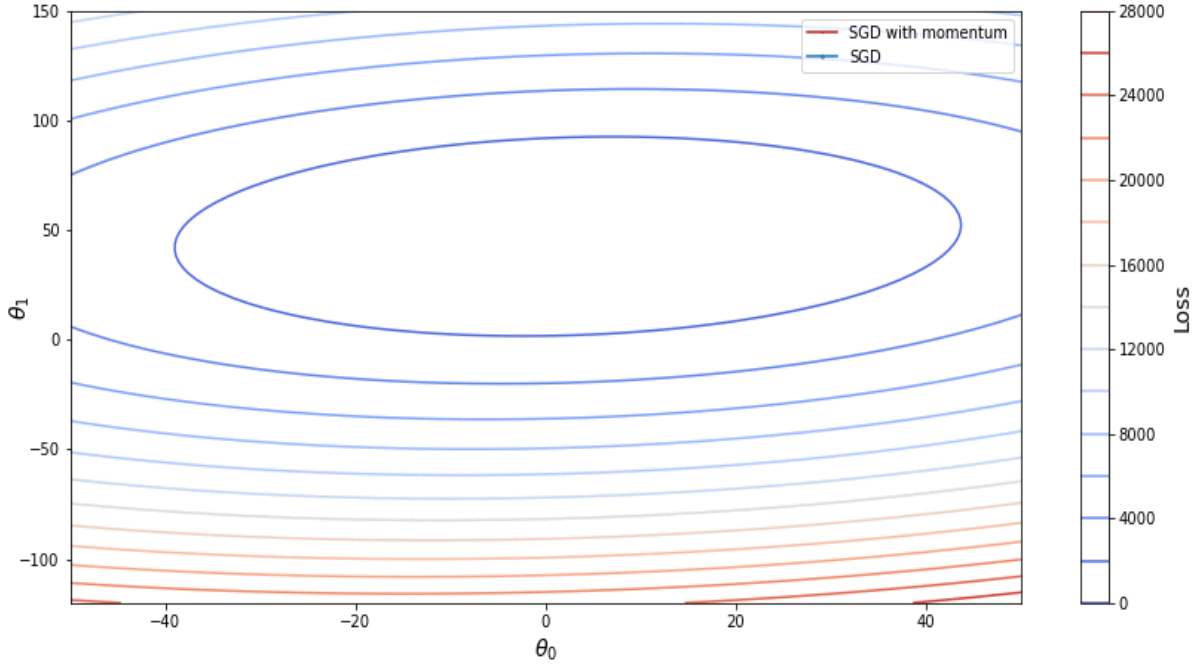
Usually $LR < 1$



Optimizers - SGD with momentum

Parameter update

$$\theta_k = \theta_k + \delta\theta_k^{t-1}$$



$$\delta\theta_k = \gamma\,\delta\theta_k^{t-1} - \epsilon\frac{\partial C(\theta)}{\partial\theta_k}$$

SGD with momentum

$$\delta\theta_k = \gamma\,\delta\theta_k^{t-1} - \epsilon\frac{\partial C\left(\theta+\gamma\,\delta\theta_k^{t-1}\right)}{\partial\theta_k}$$

Nesterov Accelerated Gradient (NAG)

Other optimization algorithms

Adagrad

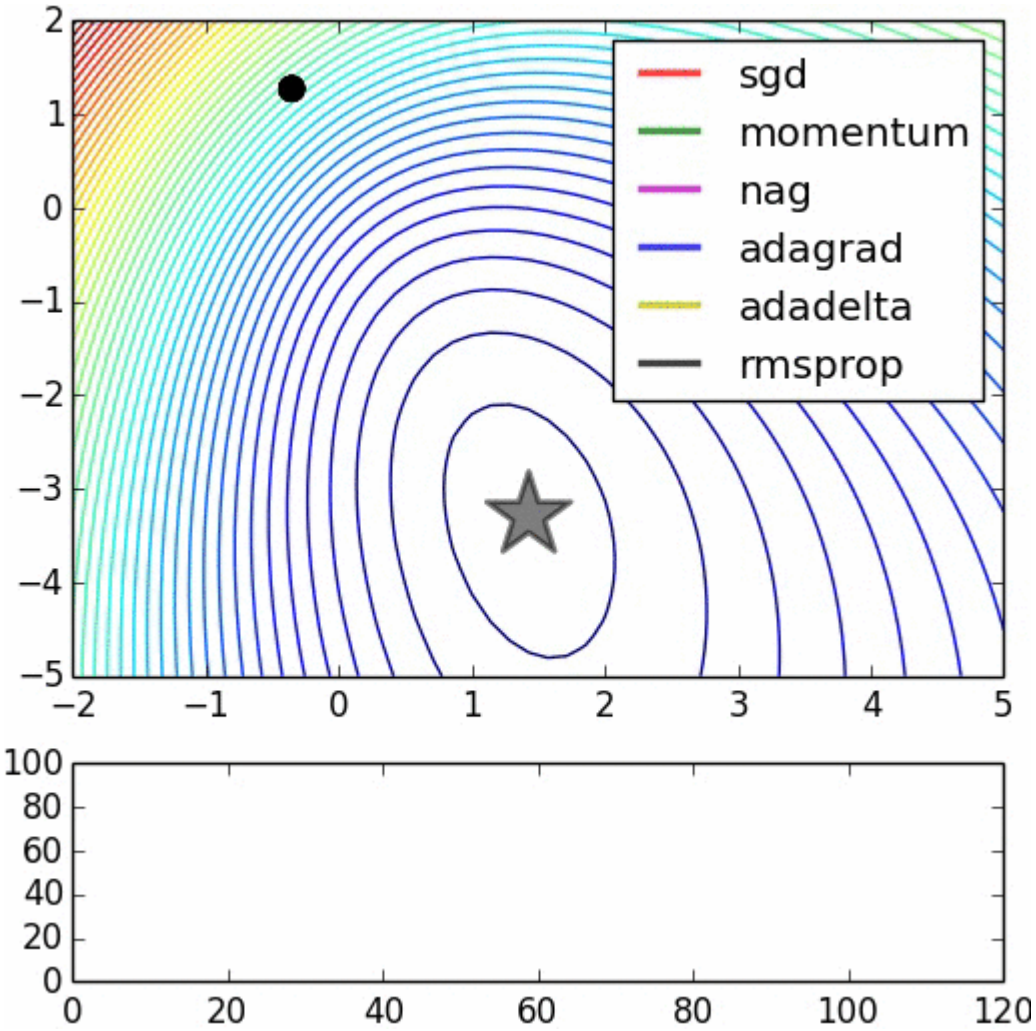
$$\begin{cases} \vec{s} \rightarrow \vec{s} + \frac{\partial C(\theta)}{\partial \theta} \times \frac{\partial C(\theta)}{\partial \theta} \\ \theta \rightarrow \theta - \epsilon \frac{\partial C(\theta)}{\partial \theta} / \sqrt{\vec{s} + \sigma} \end{cases}$$

RMSprop

$$\begin{cases} \vec{s} \rightarrow \gamma \vec{s} + (1-\gamma) \frac{\partial C(\theta)}{\partial \theta} \times \frac{\partial C(\theta)}{\partial \theta} \quad \text{with } \gamma = 0.9 \\ \theta \rightarrow \theta - \epsilon \frac{\partial C(\theta)}{\partial \theta} / \sqrt{\vec{s} + \sigma} \end{cases}$$

Adam

$$\begin{cases} \vec{m} \rightarrow \gamma_1 \vec{m} - (1-\gamma_1) \frac{\partial C(\theta)}{\partial \theta} \\ \vec{s} \rightarrow \gamma_2 \vec{s} + (1-\gamma_2) \frac{\partial C(\theta)}{\partial \theta} \times \frac{\partial C(\theta)}{\partial \theta} \\ \hat{m} \rightarrow \vec{m} / (1-\gamma_1^s) \\ \hat{s} \rightarrow \vec{s} / (1-\gamma_2^s) \\ \theta \rightarrow \theta + \epsilon \hat{m} / \sqrt{\hat{s} + \sigma} \end{cases}$$



The choice of the $C(\theta)$

There are several cost function:

- Mean Squared Error
- Cross Entropy (and Binary)
- Kullback–Leibler divergence
- Hellinger distance

$$\mathrm{BCE}(t,p) = -(t\,\log_2(p) + (1-t)\log_2(1-p))$$

$$\mathrm{KL}(P||Q) = \sum_i P(x_i)\log_2\left(P(x_i)/Q(x_i)\right)$$

$$\mathrm{HD} = \frac{1}{\sqrt{2}}\sum_i \left(\sqrt{y_\theta(x_i)} - \sqrt{y(x_i)}\right)^2$$

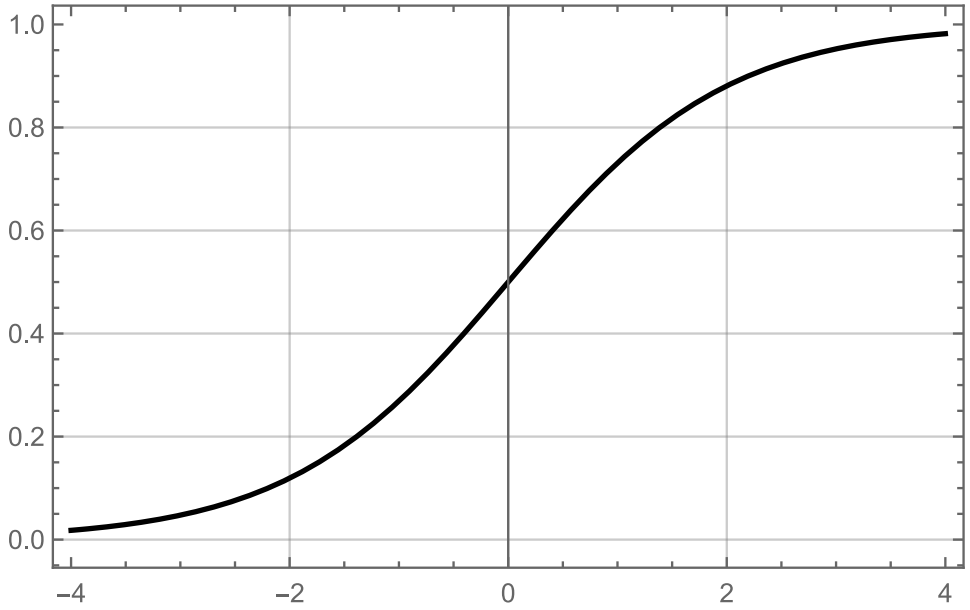
There is a lot of theory behind cost functions and each one would perform better on a problem with respect to another

The choice of the activation function

Sigmoid $\frac{1}{1+e^{-x}}$

Useful in classification problem together with BCE. Used in the output layer.

Smooth gradient, preventing “jumps” in output values. Output values bound between 0 and 1, normalizing the output of each neuron.

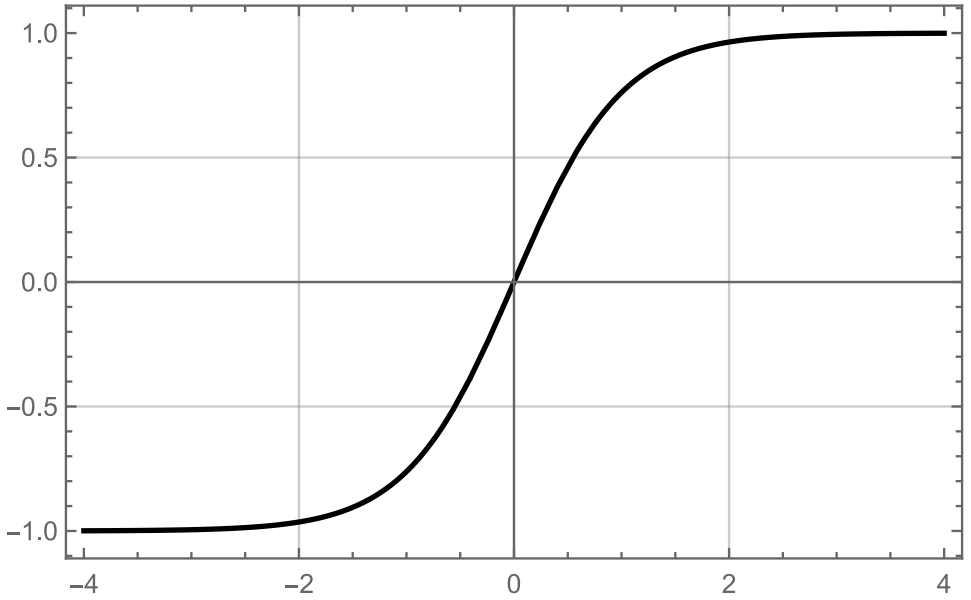


Disadvantages Vanishing gradient - for very high or very low values of the z, there is almost no change to the prediction, causing a vanishing gradient problem. This can result in the network refusing to learn further, or being too slow to reach an accurate prediction. Moreover it is computationally expensive.

The choice of the activation function

Tanh $\tanh(x)$

Zero centered: make it easier to model inputs that have strongly negative, neutral, and strongly positive values. Otherwise like the Sigmoid function.

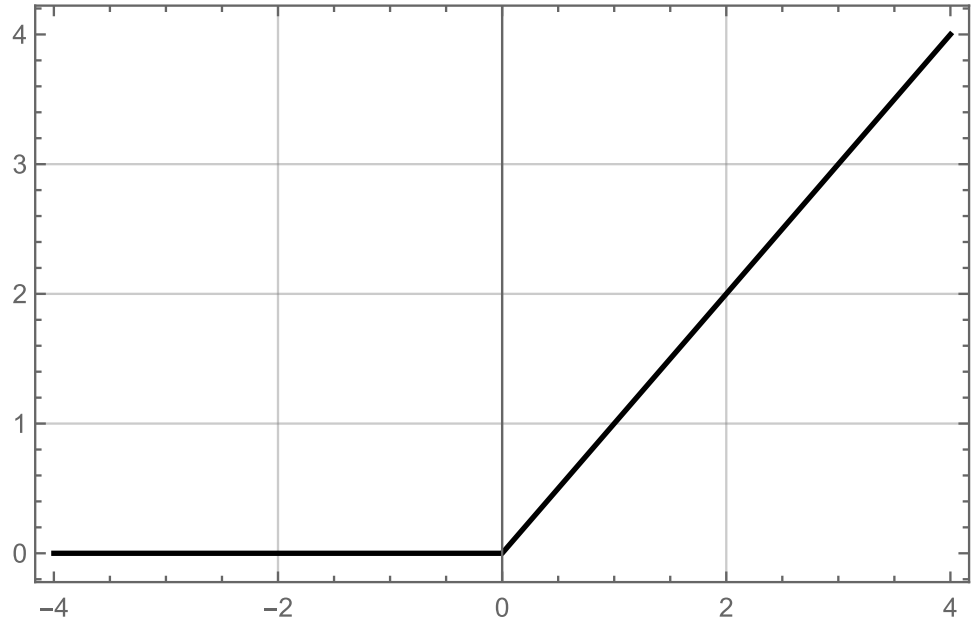


Disadvantages Same as sigmoid function.

The choice of the activation function

ReLU $\begin{cases} x & x \geq 0 \\ 0 & x < 0 \end{cases}$

Computationally efficient: allows the network to converge very quickly. Used widely in feed-forward networks. It has better performance with respect to other activation functions.

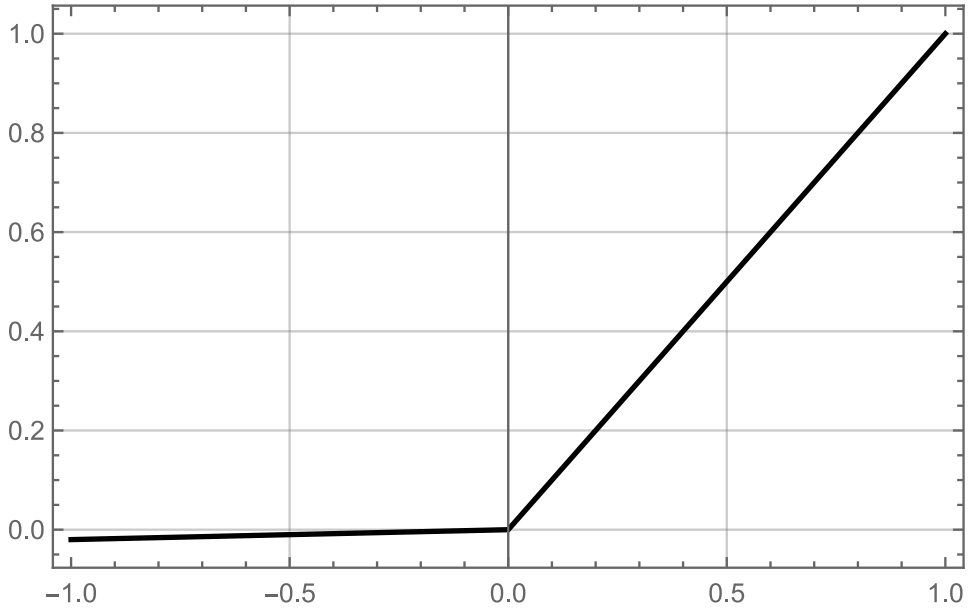


Disadvantages The Dying ReLU problem: when inputs approach zero, or are negative, the gradient of the function becomes zero, the network cannot perform backpropagation and cannot learn.

The choice of the activation function

Leaky ReLU $\begin{cases} x & x \geq 0 \\ 0.01\,x & x < 0 \end{cases}$

Prevents dying ReLU problem. This variation of ReLU has a small positive slope in the negative area, so it does enable backpropagation, even for negative input values

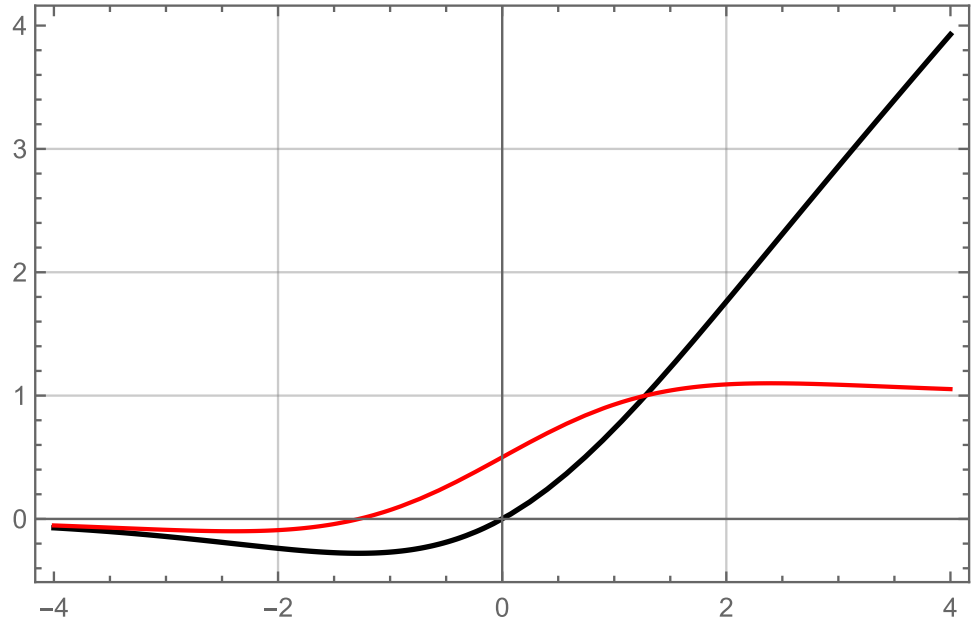


Disadvantages Results not consistent - leaky ReLU does not provide consistent predictions for negative input values.

The choice of the activation function

Swish or SiLU $x \cdot sigmoid(x) = \frac{x}{1+e^{-x}}$

Activation function discovered by researchers at Google. According to their paper, it performs better than ReLU with a similar level of computational efficiency. In experiments on image datasets perform ~ 1% better than ReLU. For batch training it seems to be faster than ReLU!



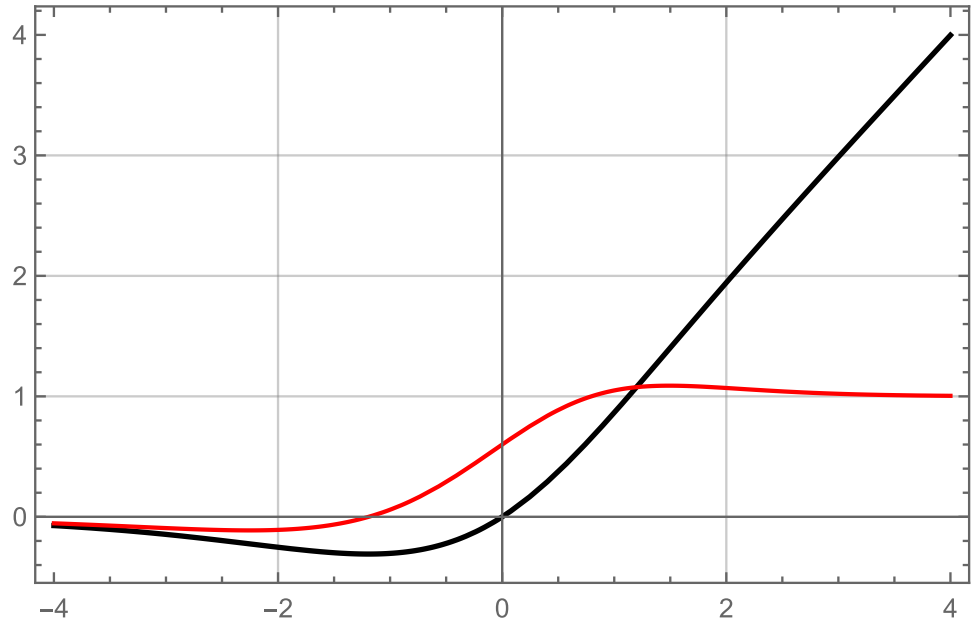
Disadvantages as much as I searched on various paper I found no know disadvantages. ReLU is much known.

Authors claim: "[…] this simple, suboptimal procedure resulted in Swish consistently outperforming ReLU and other activation functions. […] The simplicity of Swish and its similarity to ReLU means that replacing ReLUs in any network is just a simple one line code change." https://arxiv.org/abs/1710.05941 (2017)

The choice of the activation function

Mish $x \cdot \tanh\left(\ln\left(1+e^{x}\right)\right)$

Very similar to Swish but it is found that it performs slightly better.



Disadvantages same as Swish

https://arxiv.org/abs/1908.08681 (2019!)

Further NN improvements

Dropout is a regularization techniques If well tuned it prevents overfitting, increasing performances. The network could need more epochs to be trained properly but it will be more able to generalize!