Artificial Neural Networks

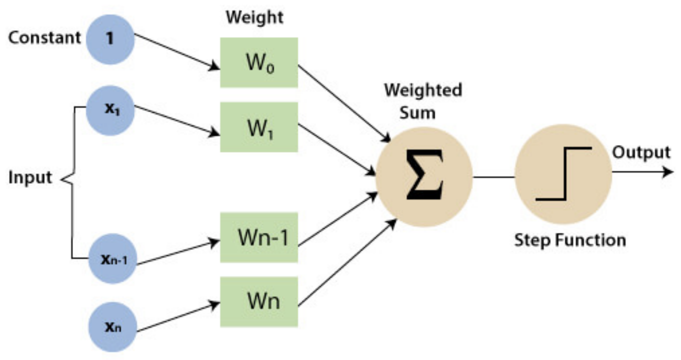
Introduction

As for other Machine Learnig techniques, the goal of Neural Networks is to find a function $\hat{y} : X \rightarrow Y$ that best approximates a target function. The core idea for solving this problem does not differ from the usual: we define a function $f(x, \theta)$ that takes as an input the input of our problems x and the weight of our model. In order to evaluate our model's predictions we will have to define an error function $E(\theta)$ and minimize it by tuning the parameters: $\theta^* = argmin_\theta E(\theta)$.

The main difference between neural networks and other classical machine learning techniques is that neural networks exploit non-linear functions not only with respect to the input space, but also with the parameters of the model itself. We say that the model is non-linear in θ.

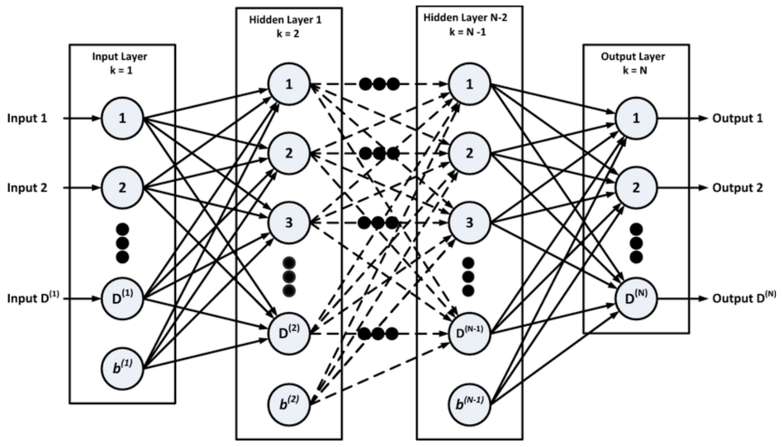
The perceptron

The intuitions for neural networks come from how biological neural networks work. The core unit of a neural network is a matematical abstraction called perceptron. A perceptron is fundamentally a pipeline that given some inputs $\{x_1, ..., x_n\}$ and weights $\{w_1, ..., w_n\}$ first computes a linear combination $\alpha = W^T X$ and then gives it as an input to the activation function that will compute the final output of the neuron.



Neural Networks

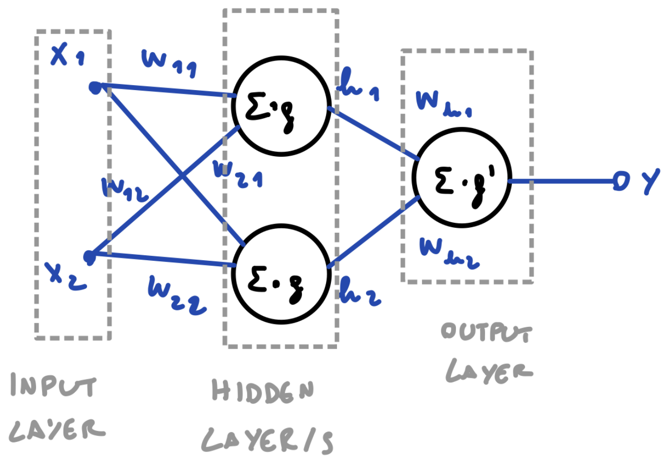
Starting from this fundamental unit, the idea is to realize a networks of interconnected perceptrons:



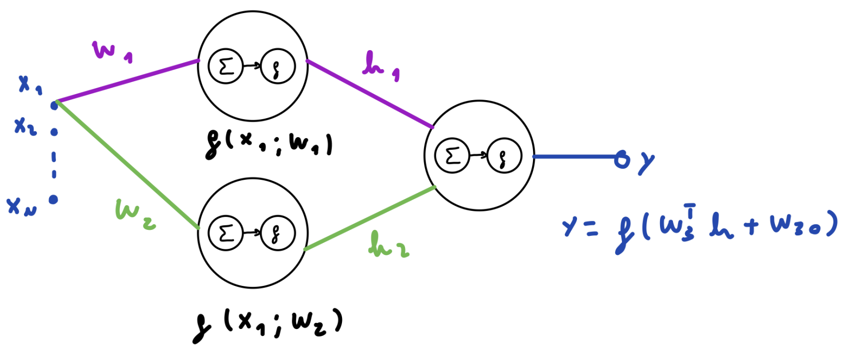
In particular, we will see Feed Forward Neural Neworks (FNNs). In this kind of networks the passage of informations from one layer to the other follows only one direction (from input to output) and does not have loops.

We have some standards denominations for layers of different nature:

- Input layer: it is that corresponds to the input features of our problem.
- Hidden layers: these are all the layers that lie between the input and the output layers.
- Output layer: the final layer of our model that will return the output.



Let’s start from a simple layout of network:



Here every input is connected to every unit of the first layer. This means that each of the units will output a linear combination of the same inputs, but with different weights and biases:

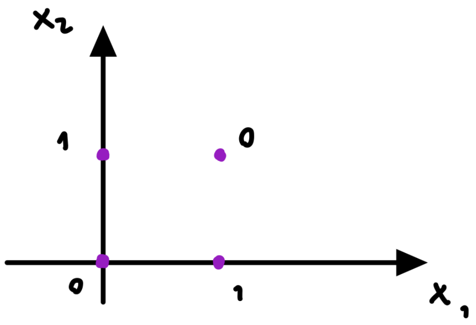
The final output will take as an input h_1, h_2 and subsequently apply a function to them. This means that in general the ouput of a neural network is actually a composition of functions:

$$h_3 = f^{(3)}(f^{(1)}(x, \theta^{(1)}), f^{(2)}(x, \theta^{(2)}), \theta^{(3)})$$

Or in an abbreviated form: $h_3 = f^{(3)}(f^{(2)}(f^{(1)}(x, \theta)))$

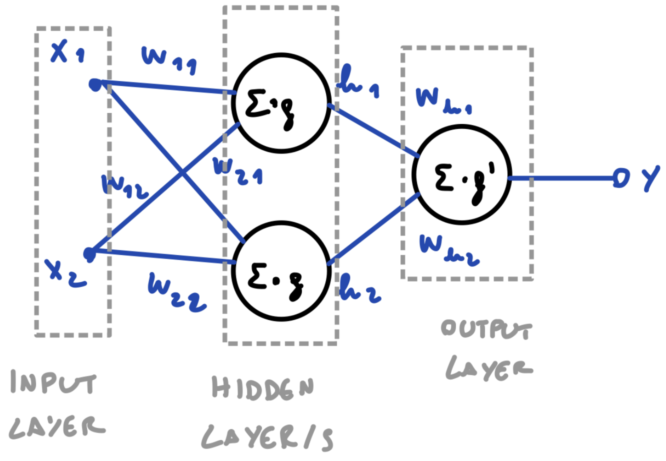
The XOR problem

We want to train a neural network on the representation of the XOR function:

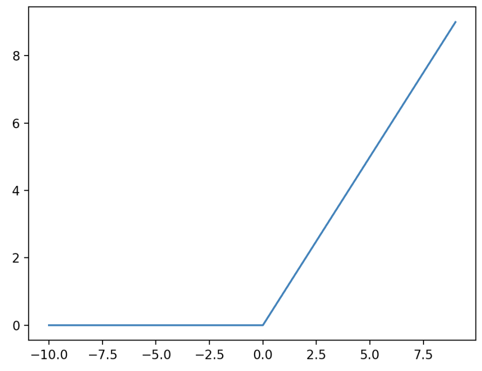


It’s clear that this problem is not solvable by a linear function. It could be solved by applying a kernel, but we will see that we don’t need it if we use a neural network.

We start from a two-layer NN:



The non-linear function denoted as g are called activation functions. As a standard, neurons of the same layer share the same activation function, while the output layer uses a different one. A common choice is using ReLu (Rectified Linear Unit) for the hidden layers and the identity function for the output layer. The ReLu function maps all the negative values to 0, while all of the non negative values stay the same:



The neural network will perform the following computations:

$$h_1 = ReLu(\begin{bmatrix} w_{11} & w_{12} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + w_{10})$$
$$h_2 = ReLu(\begin{bmatrix} w_{21} & w_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + w_{20})$$
$$y = \begin{bmatrix} w_{h_1} & w_{h_2} \end{bmatrix} \begin{bmatrix} h_1 \\ h_2 \end{bmatrix} + b = W^T h + b$$

The expression of the full model is:

$$y(x;\theta) = W^T ReLu(W^T X + c) + b$$

With $\theta =< W, c, w, b >$.

Let’s define now an error function. We choose Mean Squared Error (MSE) as a loss function:

$$J(\theta) = \frac{1}{N} \sum_{n=1}^{N} (t_n - y(x_n, \theta))^2$$

This function measures the squared distances between our prediction and the target value.

The formulation that we have studied will be capable of dealing with the XOR problem.

Design choices

When designing a neural network we have to face the following choices:

- How many layers? The **Depth** of the network.
- How many units for each layer? The **Width** of our network.
- Which kind of units? The **Activation Functions**.
- Which kind of cost functions? The **Loss Function**.

The universal approximation theorem

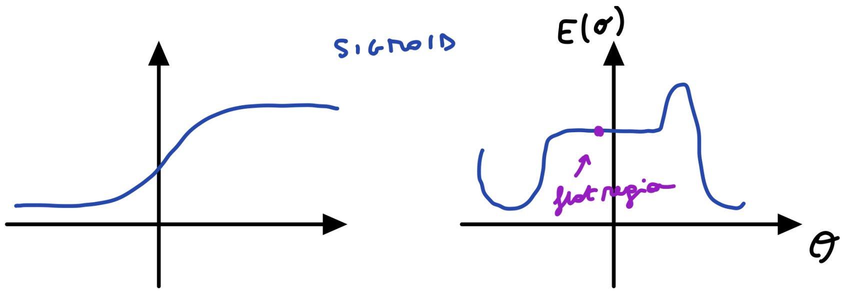
A feed forward neural network with one linear output layer and at least one hidden layer with any “squashing” function is capable of approximating any Borel measurable function with any desired amount of error, provided that enough hidden units are used.

But how many units?

We can't know the perfect amount of hidden units to use a priori. In theory a network with one very wide hidden layer could approximate any function, but in practice this results in a very bad approach. Instead, we normally use many hidden layers to create deep and narrow networks that make computations easier. Hence, the concept of **Deep Learning**.

Combining activation and loss functions

A problem with activation functions that "squish" the input space is that they present flat regions. Therefore, a bad combination of activation and loss functions could result in dealing flat regions with the loss function, which makes performing gradient descent very difficoult:



Thus, we will have to make the proper choices for these functions. We will deal with this in the next chapters.

Defining the cost function

We define the cost function based on the concept of likelihood. In particular, we are interested in the probability of obtaining the correct prediction given the input x and the weights of the neural network: $P(t|x,\theta)$.

The error is defined as the negative log-likelihood:

$$J(\theta) = E_{x,t\in D}[-ln(P(T|x,\theta))]$$

Similarly to the linear model case, we start by assuming that the likelihood's distribution is modeled by some Gaussian noise:

$$J(\theta) = E_{x,t\in D}(\frac{1}{2}||t - f(x;\theta)||^2)$$

Where $||t - f(x;\theta)||^2$ is the mean squared error.

Finding the maximum likelihood estimation corresponds to minimizing the mean squared error.

We choose the proper cost function by identifying the nature of our problem:

- Regression
- Binary classification
- Multi-class classification

Regression

When dealing with regression we choose a linear model for our last layer that implements an identity activation function: $y = W^T h + b$. We assume that the likelihood follows a Gaussian noise distribution $P(t|x) = \mathcal{N}(t|\gamma, \beta^{-1})$.

We choose mean squared error for the cost function.

Binary classification

Here we assume that our two classes are encoded as 0 and 1. We can use the Sigmoid for our activation function because of it property of mapping every possible output into values between 0 and 1: $y = \sigma(W^T h + b)$.

Since we are dealing with binary outputs, we can express the likelihood by using a probability distribution. Therefore our loss function $J(\theta) = E_{x,t \in D}\left[-ln(P(T|x,\theta))\right]$ will implement the following probability:

$$-ln(P(t|x)) = -ln[\sigma((\alpha)^t(1-\alpha)^{1-t})] = -ln[\sigma((2t-1)-\alpha)] = \\ = softplus((1-2t)\alpha)$$

Remember that $\alpha = W^T X + b$. Softplus is the name for our error function.

Multi-class classification

In this case we choose the softmax activation function defined as:

$$softmax(\alpha_i) = \frac{e^{\alpha_i}}{\sum_j e^{\alpha_j}}$$

The softmax function normalizes the output for each class. The best prediction will correspond to the class with that will return the closest value to 1.

Here the likelihood is expressed as a multinomial distribution, with J_i corresponding to the error for the i-th class:

$$J_i(\theta) = E_{x,t \in D}\left[-ln(softmax(\alpha_i)\right]$$

Activation functions for the hidden units

The premise for the choice of the activation function is that there is no theoretical better one. Empirically, it has been shown that ReLus are the best choice expecially if we don't have any particular information about the dataset. Being non-linear function makes them robust to the phenomenon of flat regions. Their only drawback is that ReLus are not differential in 0, but in practice this won't be a problem.

The formal definition of a ReLu is:

$$g(\alpha) = max(0, \alpha)$$

Computing the gradient

When implementing a neural network it's fundamental to choose an efficient way for computing the gradient. The default algorithm is **Backpropagation**.

The general idea for backpropagation is having an iterative approach, where each iteration has two steps: the first step (*forward step*) is about computing the output value for each layer until we get to the final output y. The second step (*backward step*) computes the loss between the network's output and the corresponding target value and adjusts the values for each layer based on the gradient.

How do we compute the gradient?

In order to get to the actual computation of the gradient we must start by noticing that our final output is the result of a composition of function. Thus, the gradient for the loss function $\nabla_\theta J(\theta)$ will be computed by exploiting the chain rule.

The general formula for computing the derivative of a composition of function is the product of each partial derivative for each function. For example let $y = g(x)$ and $z = f(g(x)) = f(y)$:

$$\frac{dz}{dx} = \frac{df}{dy}\frac{dy}{dx}$$

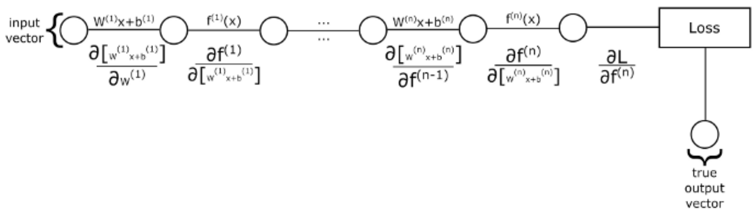
In general we will have to deal with vector functions $g : \mathbb{R}^m \rightarrow \mathbb{R}^n$ and functions $f : \mathbb{R}^n \rightarrow \mathbb{R}$.

The corresponding gradient will be:

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i}$$

Or, in vector notation: $\nabla_{x^z} = (\frac{\partial y}{\partial x})^T \nabla_{y^z}$.

This concept adapts well to the case of neural networks because the function at layer n is the result of a composition of functions for each layer from 0 to n-1. This means that in order to express its derivative we will have to apply the chain rule and multiply the derivatives for each of the n layers.



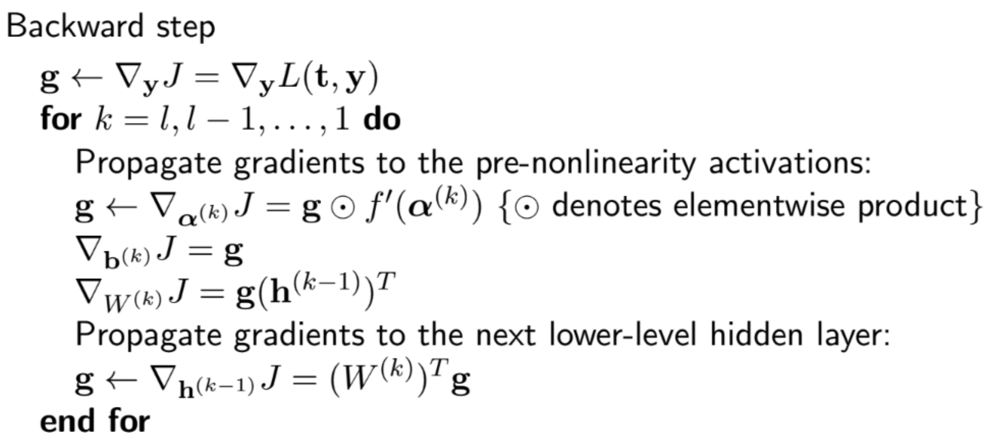
Forward step

```
Forward step
**Require:** Network depth $l$
**Require:** $W^{(i)}, i \in \{1, \dots, l\}$ weight matrices
**Require:** $\mathbf{b}^{(i)}, i \in \{1, \dots, l\}$ bias parameters
**Require:** $\mathbf{x}$ input value
**Require:** $\mathbf{t}$ target value
  $h^{(0)} = \mathbf{x}$
  **for** $k = 1, \dots, l$ **do**
    $\boldsymbol{\alpha}^{(k)} = \mathbf{b}^{(k)} + W^{(k)}\mathbf{h}^{(k-1)}$
    $\mathbf{h}^{(k)} = f(\boldsymbol{\alpha}^{(k)})$
  **end for**
  $\mathbf{y} = \mathbf{h}^{(l)}$
  $J = L(\mathbf{t}, \mathbf{y})$
```

For each layer of the function we compute the linear combination $\alpha^{(k)}$ taking as input the output of the previous layer $h^{(k-1)}$. Then we apply the activation function $h^{(k)}$ to $\alpha^{(k)}$ whose result will be the input for the next layer.

The final output y will be the output of the last layer $h^{(l)}$. The final step will be computing the loss between the target and the output.

Backward step



```
Backward step
  $\mathbf{g} \leftarrow \nabla_{\mathbf{y}} J = \nabla_{\mathbf{y}} L(\mathbf{t}, \mathbf{y})$
  for $k = l, l-1, \dots, 1$ do
    Propagate gradients to the pre-nonlinearity activations:
    $\mathbf{g} \leftarrow \nabla_{\boldsymbol{\alpha}^{(k)}} J = \mathbf{g} \odot f'(\boldsymbol{\alpha}^{(k)})$ {$\odot$ denotes elementwise product}
    $\nabla_{\mathbf{b}^{(k)}} J = \mathbf{g}$
    $\nabla_{W^{(k)}} J = \mathbf{g}(\mathbf{h}^{(k-1)})^T$
    Propagate gradients to the next lower-level hidden layer:
    $\mathbf{g} \leftarrow \nabla_{\mathbf{h}^{(k-1)}} J = (W^{(k)})^T \mathbf{g}$
  end for
```

Notice that in this algorithm the variable g represents the gradient and works as an accumulator. Each time that the arrow appears we are updating g.

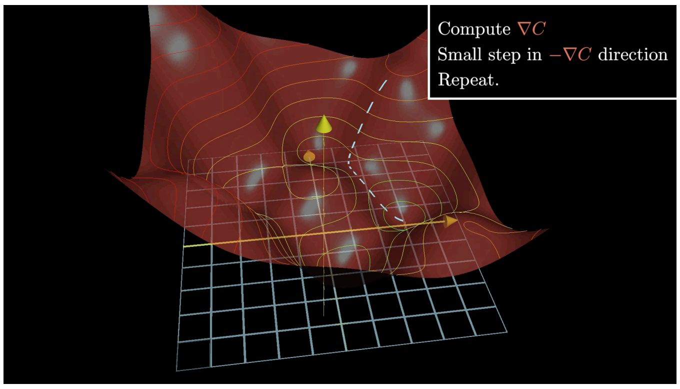
The first step consists in computing the gradient for the loss function $\nabla_y J = \nabla_y L(t, y)$ and assigning it to g.

Then for each previous layer (going from l to 1) we compute the gradient of the loss with respect to the k-th layer $\nabla_{\alpha^{(k)}} J$ as the product between the current value of g (which represents the gradient of the layer k+1) and the derivative of the non-linear function applied to the linear combination of the current layer $f'(\alpha^{(k)})$.

Rember that ultimately our goal is to compute the gradient of the loss function with respect to the weights and biases of each layer, in order to tune them properly and lower the error at the next iteration. In order to do so, we compute the gradient of the bias for layer k with respect to the loss function $\nabla_{b^{(k)}} J = g$ and the gradient of the weights for layer k with respect to the loss function -$\nabla_{W^{(k)}} J = g(h^{k-1})^T$.

The last step is computing the gradient of the loss function with respect to the output of the k-1 layer $\nabla_{h^{k-1}} J = (W^{(k)})^T g$.

This is an example of what gradient descent does in practice:



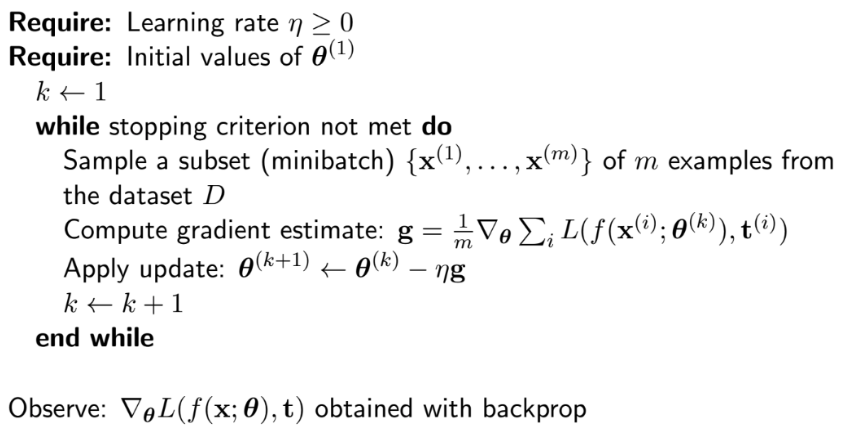
Notice that the backpropagation algorithm will be repeated many times during the training of our network. This results in a serious problem from a computational cost standpoint, meaning that even if this technique works in theory, in practice we might never reach a result in reasonable time. This is why several tricks have been developed in order to make the computation lighter:

- Use dynamic programming in order to avoid the same computation multiple times.
- Extract the symbolic form of the gradient.

Stochastic Gradient Descent (SGD)

In practice, we have seen that each step of backpropagation can be really demanding as the input size increases. A great workaround technique is stochastic gradient descent. The core concept here is to compute backpropagation with respect to a mini-batch for each iteration. A mini-batch is a small subset of the training set, made of $\{x_1, ..., x_m\}$ random samples (where m is the size of the mini batch).

At each iteration, we will compute the loss function considering only the current mini-batch, then we compute the average for the loss of each mini-batch.



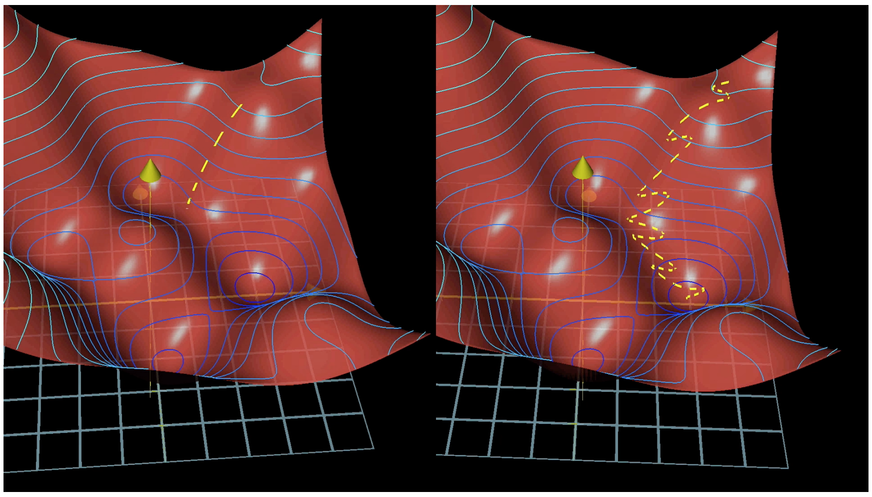
```
**Require:** Learning rate $\eta \geq 0$
**Require:** Initial values of $\boldsymbol{\theta}^{(1)}$
  $k \leftarrow 1$
  **while** stopping criterion not met **do**
    Sample a subset (minibatch) $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ of $m$ examples from
    the dataset $D$
    Compute gradient estimate: $\mathbf{g} = \frac{1}{m}\nabla_{\boldsymbol{\theta}} \sum_i L(f(\mathbf{x}^{(i)}; \boldsymbol{\theta}^{(k)}), \mathbf{t}^{(i)})$
    Apply update: $\boldsymbol{\theta}^{(k+1)} \leftarrow \boldsymbol{\theta}^{(k)} - \eta\mathbf{g}$
    $k \leftarrow k + 1$
  **end while**

Observe: $\nabla_{\boldsymbol{\theta}} L(f(\mathbf{x}; \boldsymbol{\theta}), \mathbf{t})$ obtained with backprop
```

Notice that in the algorithm appears the term η, the learning rate. We already encountered it and, as stated before, this coefficient regulates by how much we move in the direction of the gradient for each step. When deciding the value of η we can encounter two main problems. If its value is too big, we risk to skip the optimum and start oscillating back and forth until divergence. In contrast, if the learning rate is too low, we risk to move too slowly and never reach the local optimum.

The following is a comparison of what happens in classical vs stochastic gradient descent. Notice how we will be able to reach an optimum with both approaches, but

SGD require more steps in order to do it. The important thing is that each of the steps of the SGD takes substantially less time to be performed.



A possible solution to properly tune the learning rate is an adaptive approach. We start from a big value of η and we keep decreasing it at each iteration, until a certain step. This is a clever approach, since it's more likely that we start far from the optimum, and then get closer to the solution as the number of iteration increases. A formal representation of adaptive learning rate is:

Until iteration $\tau(k \leq \tau)$:

$$\eta^{(k)} = (1 - \frac{k}{\tau})\eta^{(k)} + (\frac{k}{\tau})\eta(\tau)$$

After iteration τ:

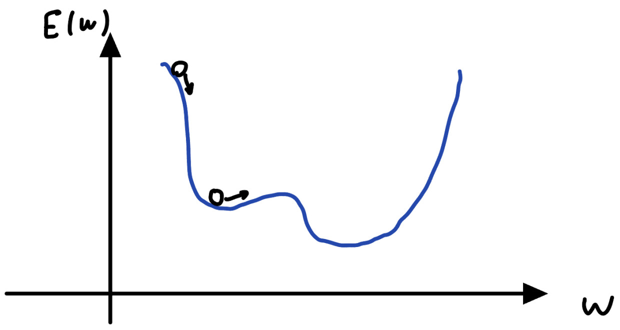
$$\eta^{(k)} = \eta^{(\tau)}$$

The value for τ will be another parameter that we'll have to properly tune.

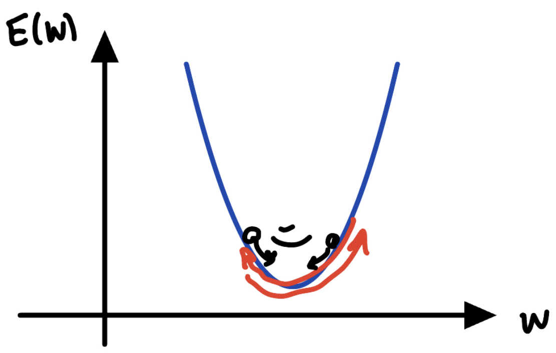
SGD with momentum

The intuition for this approach comes from the physics concept of momentum and potential energy.

When we let a ball roll from a cliff, it will still have a tendency to move forward even when it reached a plain surface due to the accumulation of momentum. The same concept can be applied to the computation of the gradient.



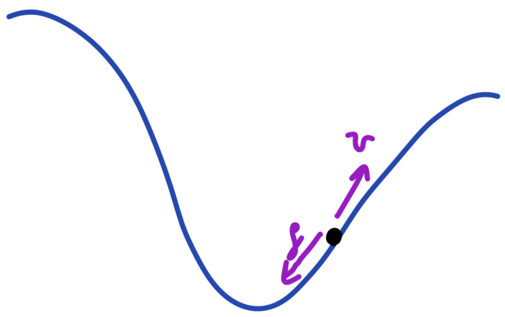
In cases like the image above, it is immediate to understand that applying momentum could make us discover better local optimums. In cases like the image shown below, the ball will keep oscillating until it eventually stabilizes due to friction:



When using SGD with momentum we have to care about two parameters:

- The learning rate $\eta \geq 0$
- The momentum $\mu \geq 0$

The momentum will regulate how much we will keep moving even if we are still. We start form the momentum in order to compute the velocity as $v^{(k+1)} \leftarrow \mu v^{(k)} - \eta g$, meaning that we will keep moving even when gradient g is non-negative. Notice that at a certain point when $v = g$ we will hit a breakpoint where the component of g will be higher than momentum and we will change direction.



Nesterov momentum

This is a technique that uses the same concept of momentum, but chooses to apply momentum before the computation of the gradient (sometimes it works better).

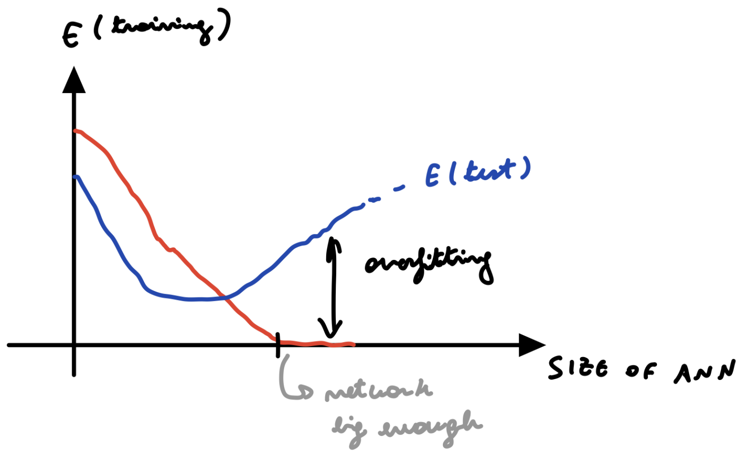
Tuning the parameters

We have seen how to compute the gradient of the loss with respect to each parameter of the network. Optimizers are the specific function that take as an input the loss function and use it to tune the parameters.

Popular optimizers are: AdaGrad, RMSProp, Adam…

Regularization

We have seen how in theory increasing the size of a neural network or the number of training iteration makes our model more performant. In reality we have to take into account that during training we are fitting our network just on a subset of samples. This will cause to hit a certain point where while the error on the training keeps decreasing, the error on the test set will start increasing. When this happens, we are facing overfitting.



We have a set of possible methods to avoid overfitting named regularization methods:

Parameter norm penalties

A common cause for overfitting is having parameters that are too high. In order to face it, we add a regularization term to the cost function that will penalize parameters with high values:

$$E_{reg}(\theta) = \sum_j |\theta_j|^q$$

The resulting cost function will be:

$$\bar{J}(\theta) = J(\theta) + \lambda E_{reg}(\theta)$$

Where λ is named regularization term, and will tune the effect of regularization.

Dataset augmentation

We know that the cause for overfitting is that our model will fit too precisely the test set, resulting in bad performances when generalizing the prediction.

A possible solution is to add noise to the training set, by producing samples that present small variation such as:

- Adding noise
- Image rotation, scaling, illumination etc.

Early stopping

We analyze the performance of the network during training over time by comparing train and test loss. We will stop when the train loss keeps dropping, but test loss starts increasing or remains stable.

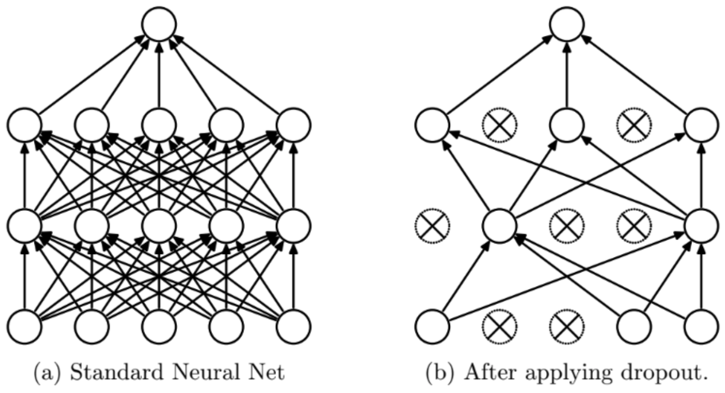
Parameters sharing

We now that a big number of parameters can increase the performance of a network, but this increases also the probability of overfitting. A possible solution here is to keep the numbers of parameters of the architecture, but introduce a constrainy such that some parameters must share the same value. This means that for each iteration we will have a certain number of different parameters (we call them trainable parameters), while all the other will be shared.

Dropout

As well as for parameters sharing, dropout is another method to deal with big numbers of parameters.

The concept here is to keep the same structure, but at each step we randomly choose a certain percentage connection that will be trained, will the others will stay the same.


(a) Standard Neural Net
(b) After applying dropout.