MACHINE LEARNING

MODELING WITH NN

Key idea (supervised, deep learning)

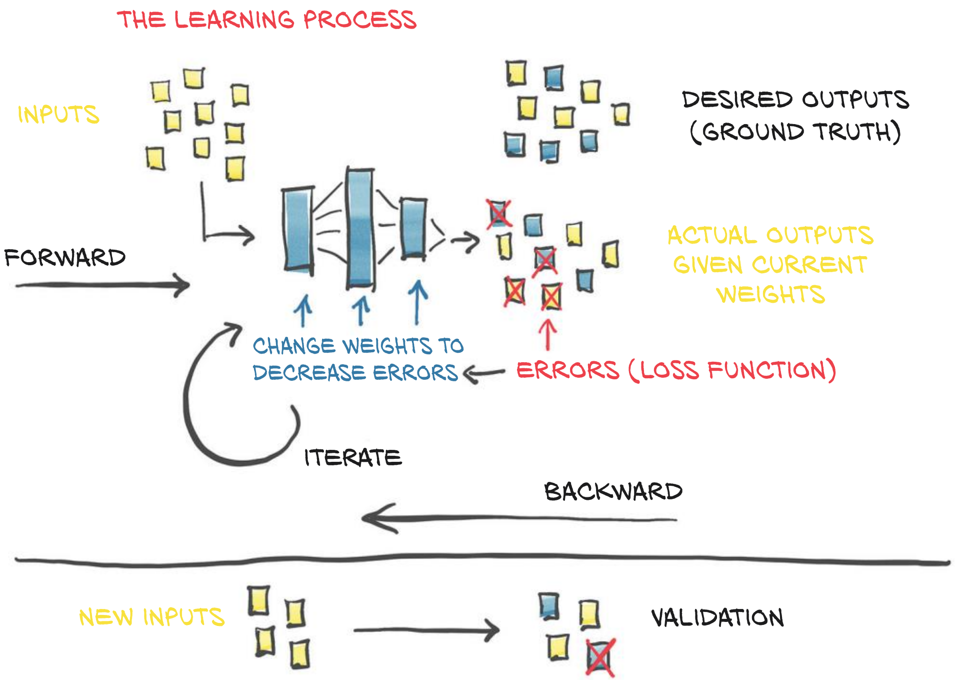
Universidad Francisco de Vitoria
UFV Madrid

Forward pass

Backward pass

Optimization

Repetition Forward pass



We've learnt the mathematical foundations that leads every single network in a powerful tool to generalize prediction models

Forward Pass: $Linear\ regression\ \circ Activation\ function = A\ neuron$

Backward Pass: $\mathcal{L}(w,b)\ , \nabla\mathcal{L}(w,b) \longrightarrow \mathrm{C}(w,b)\ , \nabla \mathrm{C}(w,b)$

Gradient Descent: $w_{n+1}, b_{n+1} = w_n, b_n - \Upsilon\, \nabla C\,(w_n, b_n)$

Repetition: $Just\ a\ loop\ with\ as\ many\ repetitions\ (epochs)\ as\ desired$

Most used frameworks

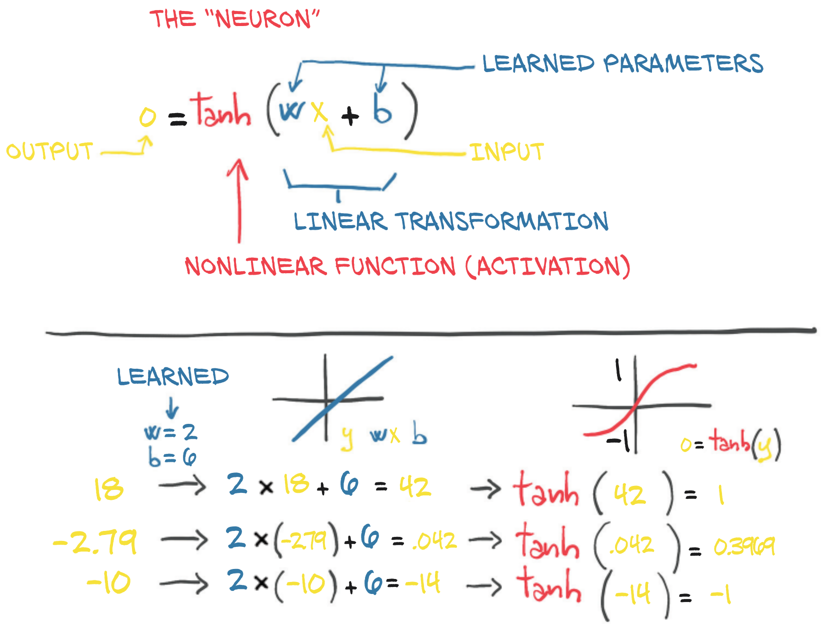
Tensorflow | **Pytorch** | **OpenAI**

Remembering a Neuron

At its core – Function Composition:

- a linear transformation of the input (multiplying the input by a number [the *weight*] and adding a constant [the *bias*]); followed by
- the application of a fixed nonlinear function (referred to as the *activation function*).

$$o = f(w \times x + b)$$



Composing a multilayer network

A multilayer neural network is made up of a composition of neurons

$x_1 = f(w_0 * x + b_0)$

$x_2 = f(w_1 * x_1 + b_1)$

...

$y = f(w_n * x_n + b_n)$

where the output of a layer of neurons is used as an input for the following layer.

Remember that w_0 here is a matrix, and x is a vector! w_0 holds an entire layer of neurons, not just a single weight.

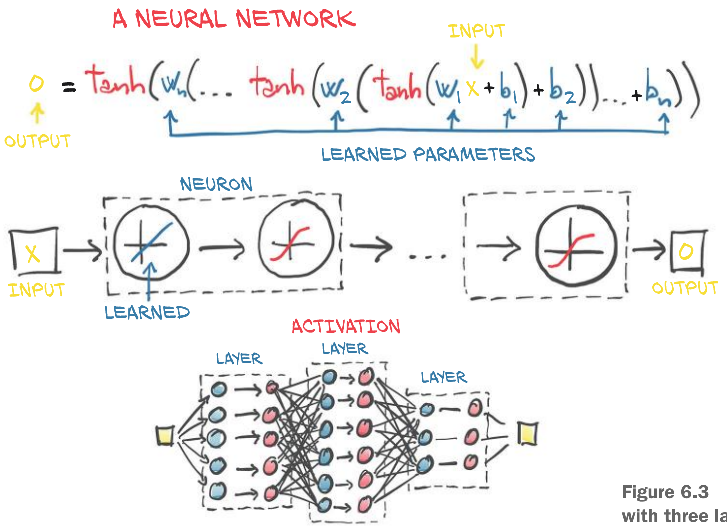


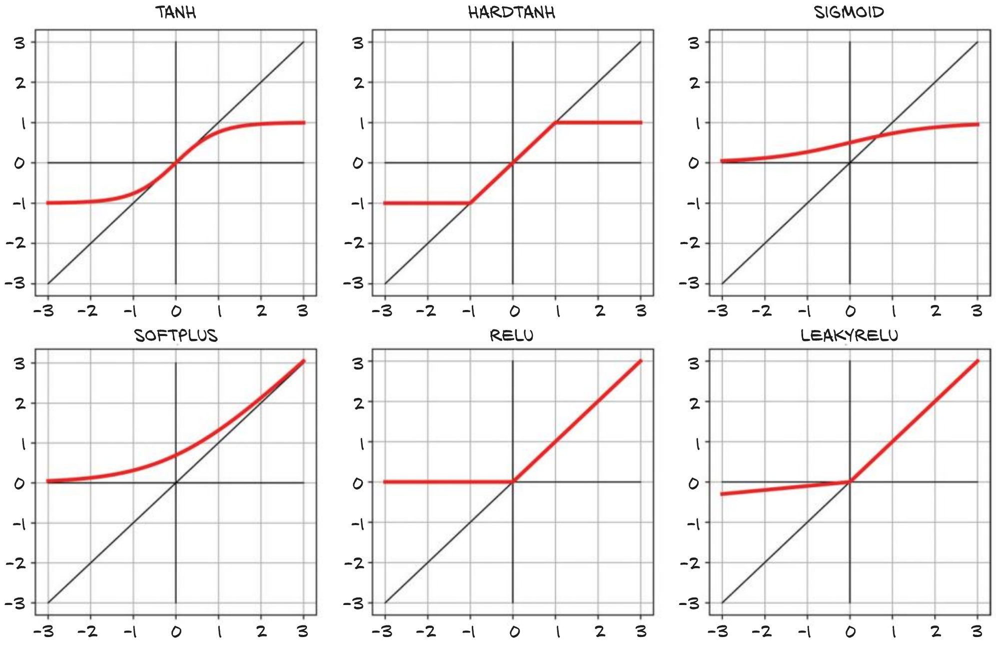
Figure 6.3 A neural network with three layers

Activation functions

Nonlinear, differentiable (point discontinuities OK).

Roles:

- Output function can have different slopes and different values
- Compresses the output range (undersaturated, sensitive and oversaturated regions)
- Some of them bounded



MODELING WITH NN

- *From CG to NN*
- *The PyTorch nn.Module*

Our goal

Universidad Francisco de Vitoria
UFV Madrid

We will move our Cat Classifier model from “Old School” to Pytorch

1 (cat)

0 (cat)

209 images for training

50 images for validation

As many images as you want to test

The PyTorch nn module

- **Objective: replace our linear model with a neural network unit**
- **torch.nn: PyTorch submodule dedicated to neural networks.**
- **Contains the building blocks for all sorts of neural network architectures: modules in PyTorch parlance (layers in other frameworks).**
- **Multiple coded nn.modules.. you don't need to write them down again!**

https://pytorch.org/docs/stable/nn.html

The PyTorch nn module

A PyTorch **module is a Python class deriving from the nn.Module base class**.

- A module can have one or more Parameter instances as attributes, which are tensors whose values are optimized during the training process (think w and b in our linear model).
- A module can also have one or more submodules (subclasses of nn.Module) as attributes, and it will be able to track their parameters as well.

Pytorch provides two ways of defining modules

Sequential style

Subclass style

nn.Linear. - Sequential style

Universidad Francisco de Vitoria
UFV Madrid

CLASS torch.nn.Sequential(*args: Module) [SOURCE]

CLASS torch.nn.Sequential(arg: OrderedDict[str, Module])

A sequential container. Modules will be added to it in the order they are passed in the constructor. Alternatively, an `OrderedDict` of modules can be passed in. The `forward()` method of `Sequential` accepts any input and forwards it to the first module it contains. It then "chains" outputs to inputs sequentially for each subsequent module, finally returning the output of the last module.

The value a `Sequential` provides over manually calling a sequence of modules is that it allows treating the whole container as a single module, such that performing a transformation on the `Sequential` applies to each of the modules it stores (which are each a registered submodule of the `Sequential`).

What's the difference between a `Sequential` and a `torch.nn.ModuleList`? A `ModuleList` is exactly what it sounds like–a list for storing `Module` s! On the other hand, the layers in a `Sequential` are connected in a cascading way.

Example:

```
# Using Sequential to create a small model. When `model` is run,
# input will first be passed to `Conv2d(1,20,5)`. The output of
# `Conv2d(1,20,5)` will be used as the input to the first
# `ReLU`; the output of the first `ReLU` will become the input
# for `Conv2d(20,64,5)`. Finally, the output of
# `Conv2d(20,64,5)` will be used as input to the second `ReLU`
model = nn.Sequential(
          nn.Conv2d(1,20,5),
          nn.ReLU(),
          nn.Conv2d(20,64,5),
          nn.ReLU()
        )

# Using Sequential with OrderedDict. This is functionally the
# same as the above code
model = nn.Sequential(OrderedDict([
          ('conv1', nn.Conv2d(1,20,5)),
          ('relu1', nn.ReLU()),
          ('conv2', nn.Conv2d(20,64,5)),
          ('relu2', nn.ReLU())
        ]))
```

You should read the documentation …

nn.Sequential. - Sequential style, an example

Universidad Francisco de Vitoria
UFV Madrid

```
# Checking what every single dimension stands for
n_samples = x_train.shape[0]
n_channels = x_train.shape[1]
n_rows = x_train.shape[2]
n_columns = x_train.shape[3]


# Model definition, we use Sequential style
model = nn.Sequential(
    nn.Linear(n_samples * n_channels * n_rows * n_columns,1),
    nn.Sigmoid()
    )
```

nn.Module – Subclass style

MODULE

CLASS torch.nn.Module [SOURCE]

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing to nest them in a tree structure. You can assign the submodules as regular attributes:

```
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))
```

Submodules assigned in this way will be registered, and will have their parameters converted too when you call `to()`, etc.

nn.Module. - Subclass style, an example

```python
class Model(nn.Module):
    def __init__(self,
                 n_neurons: int):
        super().__init__()
        self.n_neurons = n_neurons

        self.Linear1 = nn.Linear(self.n_neurons,1)

    def forward(self,x):
        x = F.sigmoid(self.Linear1(x))
        return x
```

Define a class

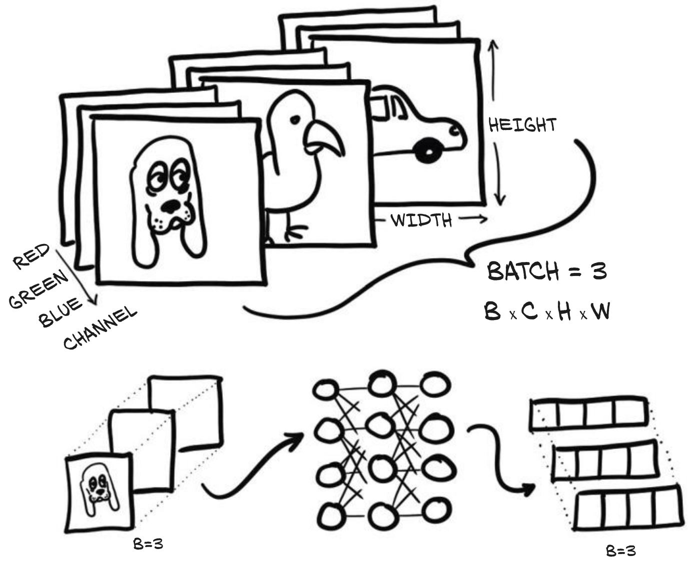
```python
# Model definition, we use Subclass style
model = Model(n_neurons = n_neurons)
```

Instance and object

Batching inputs

Any module in nn is written to produce outputs for a *batch* of multiple inputs at the same time.

Input tensor of size $B \times Nin$, where B is the size of the batch and Nin is the number of input features runs it once through the model.



MODELING WITH NN

- *From CG to NN*
- *The PyTorch nn.Module*
- *First linear module*

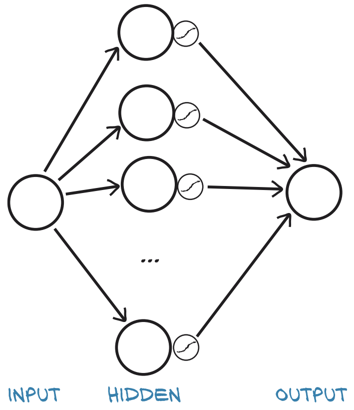
Neural Network with a hidden layer

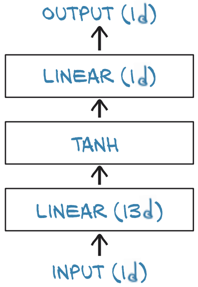
Redefinition of the model from linear to a neural net.

Simplest possible neural network:

- a linear module; followed by
- an activation function; feeding into
- another linear module.

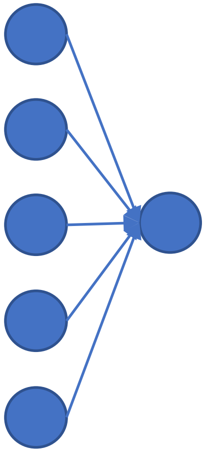
The first linear + activation layer is commonly referred to as a hidden layer for historical reasons



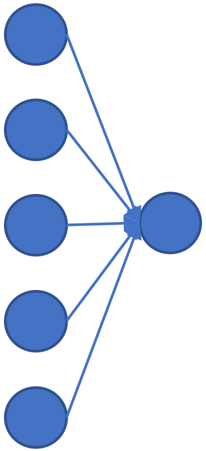


Neural Network Classification

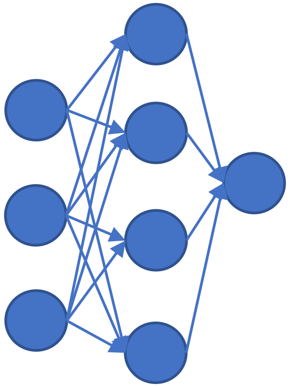
Therefore, we have four types of Neural Networks depending on the number of Layers



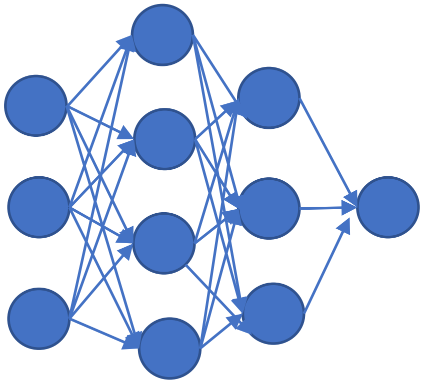
Linear Regression



Logistic Regression



Shallow Neural Network



Deep Neural Network

nn.Sequential

```
class Model(nn.Module):
    def __init__(self,
                 n_neurons: int):
        super().__init__()
        self.n_neurons = n_neurons

        self.Linear1 = nn.Linear(self.n_neurons,1)

    def forward(self,x):
        x = F.sigmoid(self.Linear1(x))
        return x
```

Returns a container, submodule of nn.Module. Modules added to it in the order they are passed in the constructor.

The forward() method of Sequential accepts any input and forwards it to the first module it contains. It then "chains" outputs to inputs sequentially for each subsequent module, returning the output of the last module.

Parameters:

[param.shape for param in seq_model.parameters()]

[torch.Size([13, 1]), torch.Size([13]), torch.Size([1, 13]), torch.Size([1])]

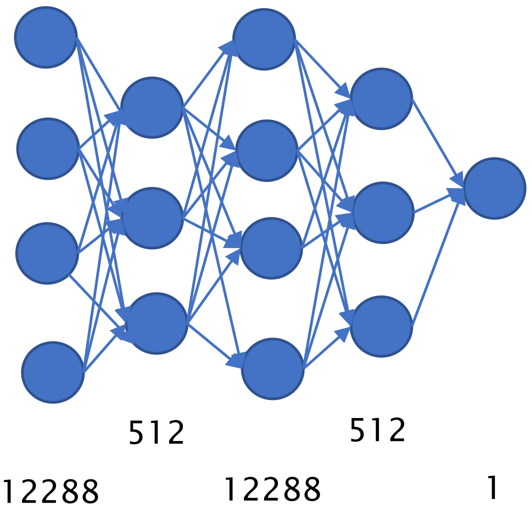
Deep Neural Network Example

```python
class Model(nn.Module):
    def __init__(self):
        super().__init__()

        self.Linear1 = nn.Linear(12288,512)
        self.Linear2 = nn.Linear(512,12288)
        self.Linear3 = nn.Linear(12288,512)
        self.Linear4 = nn.Linear(512,1)

    def forward(self,x):
        x = F.sigmoid(self.Linear1(x))
        x = F.sigmoid(self.Linear2(x))
        x = F.sigmoid(self.Linear3(x))
        x = F.sigmoid(self.Linear4(x))

        return x
```



Loss functions in nn Module

The nn comes also with several common loss functions, among them nn.BCE, which is exactly what we defined earlier as our loss_fn.

Loss functions in *nn* are still subclasses of *nn.Module*: create an instance and call it as a function.

```
# Define the Cost Function (Binary Cross Entropy)
loss_fn = nn.BCELoss()
```

BCELOSS

CLASS torch.nn.BCELoss(*weight=None*, *size_average=None*, *reduce=None*, *reduction='mean'*) [SOURCE]

Creates a criterion that measures the Binary Cross Entropy between the target and the input probabilities:

The unreduced (i.e. with `reduction` set to `'none'`) loss can be described as:

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = -w_n \left[y_n \cdot \log x_n + (1 - y_n) \cdot \log(1 - x_n) \right],$$

where N is the batch size. If `reduction` is not `'none'` (default `'mean'`), then

$$\ell(x, y) = \begin{cases} \operatorname{mean}(L), & \text{if reduction} = \text{'mean';} \\ \operatorname{sum}(L), & \text{if reduction} = \text{'sum'.} \end{cases}$$

This is used for measuring the error of a reconstruction in for example an auto-encoder. Note that the targets y should be numbers between 0 and 1.

Notice that if x_n is either 0 or 1, one of the log terms would be mathematically undefined in the above loss equation. PyTorch chooses to set $\log(0) = -\infty$, since $\lim_{x \to 0} \log(x) = -\infty$. However, an infinite term in the loss equation is not desirable for several reasons.

For one, if either $y_n = 0$ or $(1 - y_n) = 0$, then we would be multiplying 0 with infinity. Secondly, if we have an infinite loss value, then we would also have an infinite term in our gradient, since $\lim_{x \to 0} \frac{d}{dx} \log(x) = \infty$. This would make BCELoss's backward method nonlinear with respect to x_n, and using it for things like linear regression would not be straight-forward.

Our solution is that BCELoss clamps its log function outputs to be greater than or equal to -100. This way, we can always have a finite loss value and a linear backward method.

You all already know this formula

Optimization

Universidad Francisco de Vitoria
UFV Madrid

The **nn comes** also with several common Gradient Descent Algorithms, among them *nn.SGD*, that stands for Stochastic Gradient Descent

```
# Define the learning rate
learning_rate = 1e-3

# Define the Optimizer Algorithm (Stochastic Gradient Descent)
optimizer = optim.SGD(model.parameters(), lr=learning_rate)
```

Optimization

Docs > torch.optim

`Adam`	Implements Adam algorithm.
`AdamW`	Implements AdamW algorithm.
`SparseAdam`	Implements lazy version of Adam algorithm suitable for sparse tensors.
`Adamax`	Implements Adamax algorithm (a variant of Adam based on infinity norm).
`ASGD`	Implements Averaged Stochastic Gradient Descent.
`LBFGS`	Implements L-BFGS algorithm, heavily inspired by minFunc.
`NAdam`	Implements NAdam algorithm.
`RAdam`	Implements RAdam algorithm.
`RMSprop`	Implements RMSprop algorithm.
`Rprop`	Implements the resilient backpropagation algorithm.
`SGD`	Implements stochastic gradient descent (optionally with momentum).

Pytorch implements a lot of Optimization Algorithms.

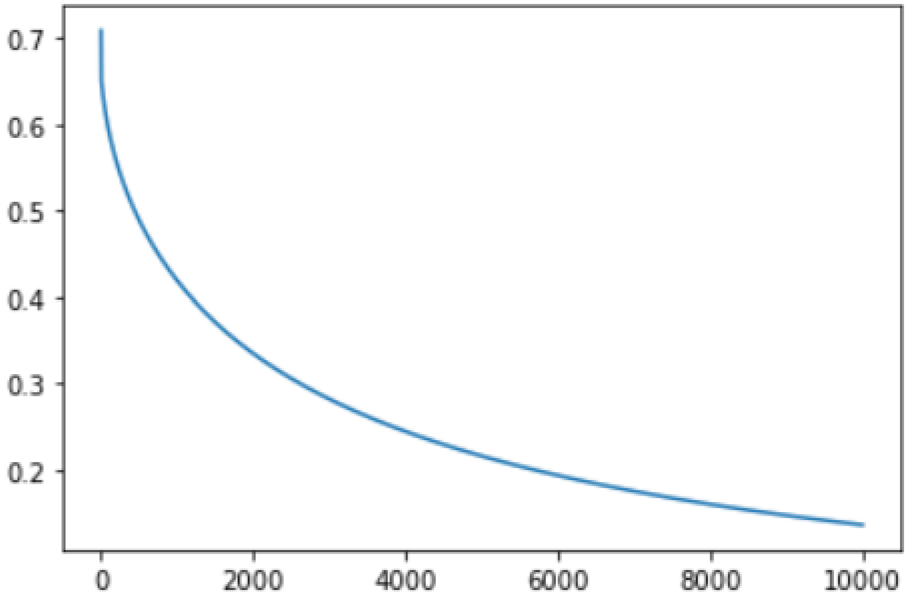
You can implement your own one if you consider neccesary

We follow our well-known four-step loop to train the model

```
losses=[]
for epoch in range(n_epochs):
        # Forward pass
        outputs = model(x_train)
        # Backward pass
        loss = loss_fn(outputs, y_train)
        # Optimizer
        optimizer.zero_grad()
        # Backward pass
        loss.backward()
        # Next step
        optimizer.step()
        print("Epoch: %d, Loss: %f" % (epoch, float(loss)))
        losses.append(loss.item())
```

Training the model

```
Epoch 1, Loss 0.664143
Epoch 2, Loss 0.654279
Epoch 3, Loss 0.651381
Epoch 10, Loss 0.644706
Epoch 11, Loss 0.643880
Epoch 99, Loss 0.592929
Epoch 100, Loss 0.592499
Epoch 1000, Loss 0.418612
Epoch 4000, Loss 0.244261
Epoch 5000, Loss 0.215819
Epoch 6000, Loss 0.193340
Epoch 7000, Loss 0.175053
Epoch 8000, Loss 0.159856
```



A neural net with more capacity

Can you increase the accuracy retrieved in this first approach?

1. Play with different architectures (more hidden layers, learning rates)

2. Data augmentation, try your own cat dataset

3. Fine tune hyperparameters (neurons, learning rate, epochs)...