**Why Learn Deep Learning?**

Get introduced to deep learning and learn why you should consider studying and/or pursuing it.

**We'll cover the following**

* [What is deep learning?](https://www.educative.io/module/page/qjv3oKCzn0m9nxLwv/10370001/5200716586549248/5339617817001984#What-is-deep-learning?)
* [Deep learning applications](https://www.educative.io/module/page/qjv3oKCzn0m9nxLwv/10370001/5200716586549248/5339617817001984#Deep-learning-applications)

**What is deep learning?**

Deep learning is considered a subfield of machine learning. Even though there are countless inspirations from real neurons, we will focus on modeling everything with formulas, intuitions, and theories that actually work.

In practice, deep learning is the scaling up of computational structures called neural networks.

Why do we take the time to develop such approaches?

Because it is the optimal solution when working with really large-scale data right now.

It is important to keep in mind that deep learning is all about learning **powerful** representations.

There is a huge shift from extracting features to learning features, and that is what deep learning is all about.

**Deep learning applications**

You will get a general perspective of a huge variety of problems that you can solve with deep learning.

First, you will learn to formulate problems in terms of machine and deep learning. That’s a crucial skill that you will use throughout your career and projects.

Secondly, you will learn the most basic components that tackle some of the following tasks:

* Image classification
* Time series prediction
* Image denoising and compression
* Image generation
* Machine translation
* Graph and node classification

Deep learning has already transformed a variety of **businesses** such as web search, augmented reality, social networks, automobiles, retail, cybersecurity, and manufacturing. But the most exciting thing is the potential novel applications that may appear in the future. These projects can radically transform every industry.

Some experts claim that **AI is the new electricity.** While this may be a disputable idea, what is for certain is that **deep learning is one of the most sought after and well-paid skills.**

So why stay behind?

# Linear Classifiers

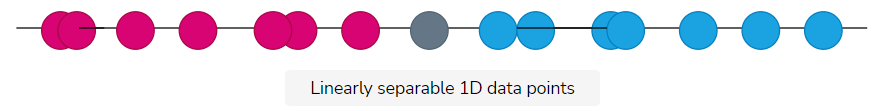
Explore linear classifiers, their principles, and their training process.

**We'll cover the following**

* [What is a linear classifier?](https://www.educative.io/module/page/qjv3oKCzn0m9nxLwv/10370001/5200716586549248/5568462289633280#What-is-a-linear-classifier?)
* [Training a classifier](https://www.educative.io/module/page/qjv3oKCzn0m9nxLwv/10370001/5200716586549248/5568462289633280#Training-a-classifier)
* [Loss function](https://www.educative.io/module/page/qjv3oKCzn0m9nxLwv/10370001/5200716586549248/5568462289633280#Loss-function)
* [Optimization and training process](https://www.educative.io/module/page/qjv3oKCzn0m9nxLwv/10370001/5200716586549248/5568462289633280#Optimization-and-training-process)

## What is a linear classifier?

Suppose we want to build a machine learning model to classify the following points into two categories based on their color. It is very easy to see that we can find a single point that separates them perfectly. The goal of our model is to find this point.

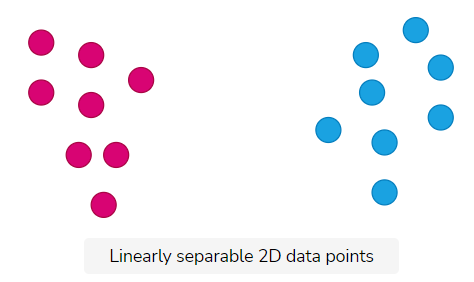


The easiest way to do that is to build a linear classifier. Our classifier has the form  (*x*,*w*)=*w*1​∗*x*1​+*w*2​. The purpose of *f*(*x*,*w*) will be to find the parameters *w*1​ and *w*2​, so that any corresponding scalar point (1D) can be distinguished perfectly. If *f*(*x*,*w*)>0, the point belongs to the blue category. Otherwise, it belongs to the red. Sounds easy?

Let’s extend this idea to 2D data points!

Each point will now be represented as (*x*1​,*x*2​).

For the 2D case, we need to find a line (instead of a point) that separates our 2D points, so our classifier will be *f*(*x*,*w*)=*w*1​∗*x*1​+*w*2​∗*x*2​+*w*3​. Again, the classifier should be trained to find the optimal *w*1​, *w*2​, and *w*3​.

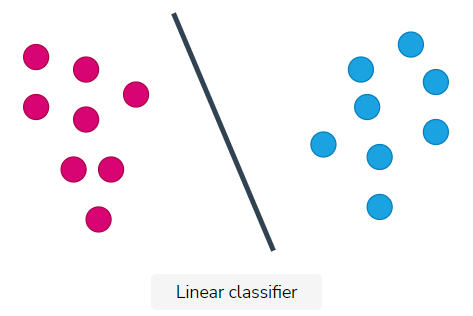


This idea can be naturally extended to arbitrary (N) dimensions. The line in the 2D space will be a plane in the 3D space and N-plane in higher dimensions. We call this the hyperplane of the N-dimensional space that can separate the space into 2 classes (red and blue).

We will utilize linear algebra and matrices to formulate it. To facilitate the readability, matrices will be denoted in bold.

As a result, we now have *f*(*x*,**W**)=**W**∗*x*+*b* , where *x* and *b* are N-dimensional vectors while *W* is an *N*×*N* matrix. We need to find the correct values of **W** and *b* to define a hyperplane. When we have those, we can receive the category *y* for any data-point *x*.

From now on, we’ll denote our classifier as *f*(*xi*​,**W**)



In Pytorch, we can build a linear classifier with 5 inputs and 10 outputs using just one line of code. The following code will initialize a trainable matrix **W** and a vector *b* and every time we use class instance classifier, it will perform the operation *y*=**W**∗*x*+*b*

## Basic imports

import torch

import torch.nn as nn

## initializes a matrix W and a vector b

classifier = nn.Linear(5, 10)

## Training a classifier

We know that we want to find the matrix **W** and the vector *b* in order to classify our examples.

But how?

First of all, we need training data. Training data are data-points (*x*) whose category (target class *t*) we are aware of. Thus, we can utilize it to “train” our classifier.

“Training” a classifier refers to the notion of trying to find the matrix **W** by feeding to its already known data points.

Because we know the “labels” (category) of the data, these training approaches are called supervised. The data are provided in pairs (*x*,*t*). We use the *x* as an input to the classifier and the labels *t* to compute the loss (distance). Note that *y* refers to the output of the classifier and will be equal with *y*=**W**∗*x*+*b*

Intuitively, we will push the randomly initialized model to learn this mapping from *x*−>*t*.

Before we describe the process of training, we need to describe two more concepts.

## Loss function

**Loss (or cost) is a measure of how good or bad a classification of a data-point is**. Alternatively, it can be defined as how far the classifier’s prediction *y* is, for the data-point *x*, from the actual class *t*. Let’s make that crystal clear:

Given a dataset (*xi*​,*ti*​) of N points where *xi*​ is an N-dimensional point in space and *ti*​ is an integer that defines the point’s category, loss is the distance between *f*(*xi*​,*W*) and *ti*​.*Ci*​(*f*(*xi*​,*W*),*ti*​) is the cost for a single example *xi*​.

The overall loss of the entire training data is simply the average of all the individual losses. However, in practice, we rarely average the loss over all data points.

Note that the choice of the loss function depends on the problem and the form of the data. In our case, from now on, we will use the mean squared error distance defined as:

2*C*=∑(*f*(*xi*​,*W*)−*ti*​)2

Notice that the sum is between the elements of the vector. Here is a code example:

import torch

import torch.nn as nn

model = nn.Linear(10,3)

loss = nn.MSELoss()

## dummy input x

input\_vector = torch.randn(10)

## class number 3, denoted as a vector with the class index to 1

target = torch.tensor([0,0,1])

## y in math

pred = model(input\_vector)

output = loss(pred, target)

print("Prediction: " ,pred)

print("Output: " , output)

It is important to understand that even though the target class is a scalar (3 in the example above), we convert it to a tensor. For three classes, you will have these possible target vectors *t*:

class 1 -> [1,0,0]  
class 2 -> [0,1,0]  
class 3 -> [0,0,1]

**Optimization and training process**

Optimization is the process of finding the weight matrix **W** that minimizes the loss function. In other words, it is the process of selecting the individual weights *wi*​ so that the classifier’s prediction *y* is as close as possible to the point’s real label *t*.

Mathematically this can be written as:

*w*′=*argminw*​(*C*(*w*))

For now, let’s keep in mind that optimization is an abstract concept that describes how we select the matrix. We will dive into it in the next lesson where we will talk about neural networks.

Now, we can describe the training algorithm in its entirety:

Given a set of training examples *xi*​ with their labels *ti*​, we need to:

* Initialize the classifier *f*(*xi*​,**W**) with random weight **W**.
* Feed a training example in the classifier and get the output *yi*​.
* Compute the loss between the prediction *yi*​ and *ti*​.
* Adjust the weights **W** according to the loss *Ci*​ (next lesson).
* Repeat for all training examples.

This is the core idea behind all deep learning models. In the end, we will have a trained classifier that can be generalized in previous unseen examples.

The only step that should be unclear now is how we adjust the weights. We will discuss this in the next lesson.

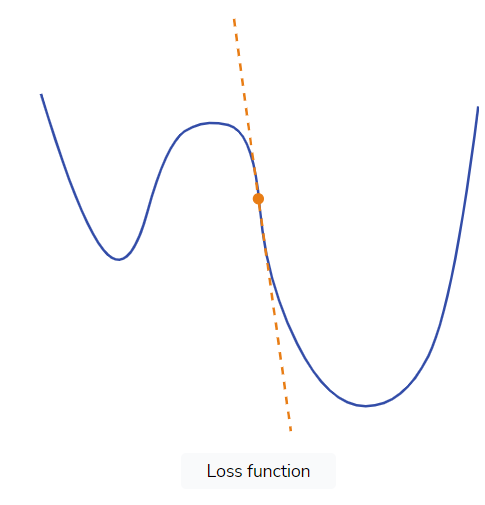
# Optimization and Gradient Descent

Learn about the fundamental algorithm behind machine learning training: gradient descent.

**We'll cover the following**

* [Slope: the derivative of the loss function](https://www.educative.io/module/page/qjv3oKCzn0m9nxLwv/10370001/5200716586549248/4611497874948096#Slope:-the-derivative-of-the-loss-function)
* [Computing the gradient of a loss function](https://www.educative.io/module/page/qjv3oKCzn0m9nxLwv/10370001/5200716586549248/4611497874948096#Computing-the-gradient-of-a-loss-function)
* [Summing up the training scheme](https://www.educative.io/module/page/qjv3oKCzn0m9nxLwv/10370001/5200716586549248/4611497874948096#Summing-up-the-training-scheme)

In our 2D example, the loss function can be thought of as a parabolic-shaped function that reaches its minimum on a certain pair of *w*1​ and *w*2​. Visually, we have:



To find these weights, the core idea is to simply follow the slope of the curve. Although we don’t know the actual shape of the loss, we can calculate the slope in a point and then move towards the downhill direction.

You can think of the loss function as a mountain. The current loss gives us information about the local slope.

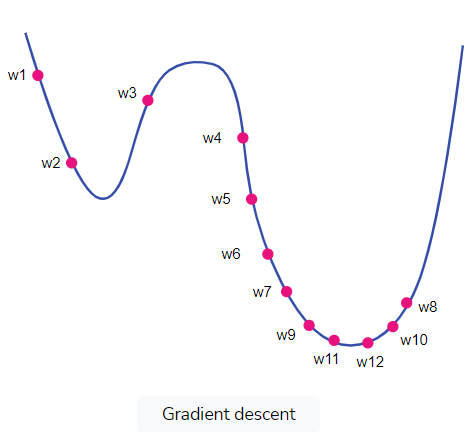
But what is the slope?

## Slope: the derivative of the loss function

In calculus, the slope is the derivative of the function at this point and is denoted as ∂*x/*∂*w*​. The ultimate goal would be to find the global min. The minimums, local or global, have a nearly zero derivative, which indicates that we are located at the minimum of the curve.

For now, suppose that we want to minimize the loss function *C*. By calculating the derivative, we will take small steps along the slope in an iterative fashion. In this way, we can gradually reach the minimum of the curve.

The same principle can be extended into many dimensions *N*. Despite the fact this is very difficult to visualize, maths is here to help us.



**Computing the gradient of a loss function**

The question is how do we compute the derivative (or gradient) with respect to the weights? In simple cases, such as the two-dimensional one, we can compute the analytical form with calculus.

Since our loss function is �=(�(��,�)−��)2*C*=(*f*(*xi*​,**W**)−*yi*​)2, where the classifier �*f* is �=�1∗�+�2*f*=*w*1​∗*x*+*w*2​, we can easily prove that:

∂�∂�1=2(�1∗�+�2−�)�∂*w*1​∂*C*​=2(*w*1​∗*x*+*w*2−*y*)*x*

∂�∂�2=2(�1∗�+�2−�)∂*w*2​∂*C*​=2(*w*1​∗*x*+*w*2−*y*)

This is nothing more than the partial derivatives with respect to our 2 weights. In complex cases, such as neural networks, the chain rule will come to the rescue.

Now that we have our gradients, let’s adjust our weights to go downhill:

�1�=�1�−1−�∗∂�∂�1*w*1*t*​=*w*1*t*−1​−*λ*∗∂*w*1​∂*C*​

�2�=�2�−1−�∗∂�∂�2*w*2*t*​=*w*2*t*−1​−*λ*∗∂*w*2​∂*C*​

where �*λ* is a small constant called **learning rate**. The learning rate �*λ* is usually between 10−310−3 and 10−610−6 and defines how quickly we move down towards the direction of the gradient.

The negative sign intuitively means that we are going downhill! We follow the negative slope of the curve.

That’s all? Yes and no.

Yes, because this principle will come in handy all the time. No, because we will not calculate the derivatives for every single neural network that we will use. Don’t worry!

However, we will analyze many more aspects of optimization as it is the heart of machine learning.

Ok, we found the gradient! How do we change the parameter?

This is the so-called update rule:

The update rule for iteration �*j* of a scalar weight �*w* is as follows:

��=��−1−�×∂�∂��*wj*=*wj*−1−*λ*×∂*wj*∂*C*​

The index �*j* shows the iteration step.

**Summing up the training scheme**

To recap, the training algorithm, known as gradient descent, can be formulated like this for the N-dimensional case:

* Initialize the classifier *f*(*xi*​,**W**) with random weights **W**.
* Feed a training example *xi*​ (vector) with corresponding target vector *ti*​ in the classifier, and compute the output *yi*​=*f*(*xi*​,**W**).
* Compute the loss between the prediction *yi*​ and target vector *ti*​. The mean squared error loss is commonly used 2*C*=∑(*yi*​−*ti*​)2.
* Compute the gradients for the loss with respect to the weights/parameters.
* Adjust the weights **W** based on the rule *wit*​=*wit*−1​−*l*∗∂*wi*​∂*C*​. Note that ∂*wi*​∂*C*​ is the gradient of the parameter and *λ* the learning rate.
* Repeat for all training examples.

In Pytorch, the entire algorithm can again be developed with a few lines of code. In the following snippet, we have a simple linear classifier that is trained using gradient descent and the mean squared error loss. It accepts a four-sized vector and outputs a single value.

Feel free to play around with the following code by trying different inputs and inspect the output and the gradient of the model. But don’t try to dive too deep into the code as we will discuss it in detail in the upcoming lessons.

import torch

import torch.nn as nn

def train():

model = nn.Linear(4,2)

print(model)

print('-------------')

criterion = torch.nn.MSELoss()

optimizer = torch.optim.SGD(model.parameters(), lr=0.1)

for epoch in range(10):

# Converting inputs and labels to Variable

inputs = torch.Tensor([0.8,0.4,0.4,0.2])

labels = torch.Tensor([1,0])

# Clear gradient buffers because we don't want any gradient from previous epoch to carry forward

optimizer.zero\_grad()

# get output from the model, given the inputs

outputs = model(inputs)

# get loss for the predicted output

loss = criterion(outputs, labels)

print(loss)

# get gradients w.r.t to parameters

loss.backward()

# update parameters

optimizer.step()

print('epoch {}, loss {}'.format(epoch, loss.item()))

if \_\_name\_\_ == "\_\_main\_\_":

train()

# Neural Networks

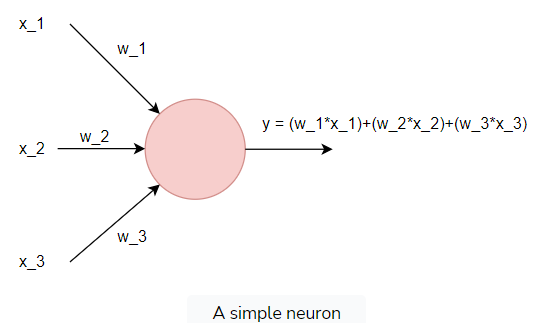
Understand the neural network, how it is formulated, and why it works.

**We'll cover the following**

* [What is a neuron?](https://www.educative.io/module/page/qjv3oKCzn0m9nxLwv/10370001/5200716586549248/6163653060984832#What-is-a-neuron?)
* [Multilayer perceptron](https://www.educative.io/module/page/qjv3oKCzn0m9nxLwv/10370001/5200716586549248/6163653060984832#Multilayer-perceptron)
* [Universal approximation theorem](https://www.educative.io/module/page/qjv3oKCzn0m9nxLwv/10370001/5200716586549248/6163653060984832#Universal-approximation-theorem)
* [Deep neural networks as feature extractors](https://www.educative.io/module/page/qjv3oKCzn0m9nxLwv/10370001/5200716586549248/6163653060984832#Deep-neural-networks-as-feature-extractors)

## What is a neuron?

In the world of Neural Networks (NN), the basic building block is the neuron. NNs are nothing more than a collection of neurons organized in layers, with information passing from one layer to the other. So to understand NN, we first need to understand the neuron: the basic computing unit.



Mathematically, we have:

y=*w*1​∗*x*1​+*w*2​∗*x*2​+*w*3​∗*x*3​

A neuron is simply a linear classifier with a single output.

import torch.nn as nn  
  
neuron = nn.Linear(3,1, bias=False)

Looks familiar? In most applications, we also add a bias �*b* to shift the position of the boundary line that separates the data points. This is the infamous Perceptron.

To extend this idea, we also pass this weighted average through a non-linear function �*σ* that will give us the decision boundary.

Why?

Because with non-linear functions between linear layers, we can model much more complex representations with less linear layers.

Non-linearities is a key component that makes NN very rich function approximators.

Putting it all together, we have:

*y*=*σ*(*w*1​∗*x*1​+*w*2​∗*x*2​+*w*3​∗*x*3​+*b*)=*σ*(*w*∗*x*+*b*)

We organize neurons in layers with nn.Linear(in\_features, out\_features) and stack layers in sequential order. The layer’s stacking combined with the non-linear activation function gives us the ability to distinguish non-linearly separable data.

There are multiple names in the literature for stacking linear layers with non-linear activations: Multi-Layer Perceptron (MLP), artificial Neural Network, and feedforward module. All these terms mean the same thing.

In practice, for three input features and two classes, our model will be like this:

import torch.nn as nn

## 20 is the hidden dimension. arbitary choice

model = nn.Sequential(

nn.Linear(3,20), # 3 for the input features x1,x2,x3

nn.ReLU(),

nn.Linear(20,2)) # 2 for the classes

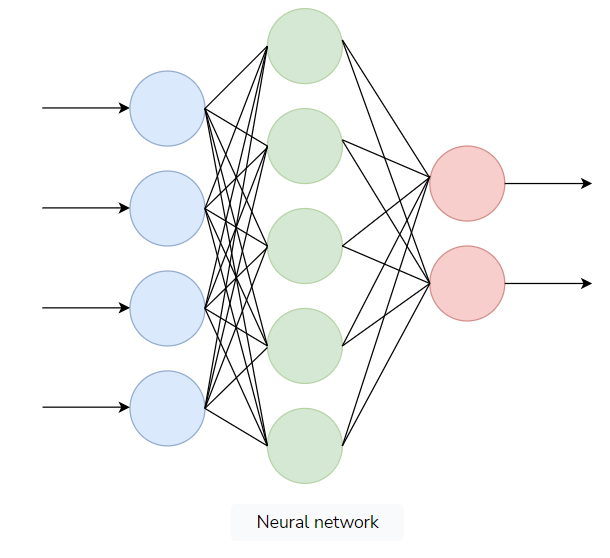
print(model)

Output

5.43s

Sequential( (0): Linear(in\_features=3, out\_features=20, bias=True) (1): ReLU() (2): Linear(in\_features=20, out\_features=2, bias=True) )

The activation function *σ* is nn.ReLU().



A good practice for you is to write the depicted image in Pytorch.

**Hint:** It is not that different from the illustrated code

## Multilayer perceptron

But why are MLPs able to find non-linear functions? Let’s have another look at NNs from a mathematical perspective. Remember that each neuron is represented with:

�=�(�1∗�1+�2∗�2+�3∗�3+�)*y*=*σ*(*w*1​∗*x*1​+*w*2​∗*x*2​+*w*3​∗*x*3​+*b*)

So for two neurons nn.Linear(3,2), we would have:

�1=�(�1∗�1+�2∗�2+�3∗�3+�)*y*1​=*σ*(*w*1​∗*x*1​+*w*2​∗*x*2​+*w*3​∗*x*3​+*b*)

�2=�(�4∗�1+�5∗�2+�6∗�3+�)*y*2​=*σ*(*w*4​∗*x*1​+*w*5​∗*x*2​+*w*6​∗*x*3​+*b*)

This can be transformed using linear algebra to �=�(�∗�+�)*y*=*σ*(**W**∗*x*+*b*) , where:

�=[�1�2]*y*=[*y*1​*y*2​​]

�=[�1�2�3�4�5�6]*W*=[*w*1​*w*4​​*w*2​*w*5​​*w*3​*w*6​​]

�=[�1�2�3]*x*=⎣⎡​*x*1​*x*2​*x*3​​⎦⎤​

�=[�1�2]*b*=[*b*1​*b*2​​]

If we assume that �*x* is our input and �*y* is our output, we can see that we have a non-linear expression with respect to �*x*. Note that in the absence of an activation function, we will end up with a linear classifier.

If we add a second layer to make our model more expressive, we will have:

*y*2=*f*(*W*2∗*z*+*b*2)

where *z*=*W*1*x*+*b*1.

This can be written in a more general format for current layer *L* and previous layer *L*−1 as:

*yL*=*f*(*WL*⋅*σ*(*WL*−1*xL*−2+*bL*−1)+*bL*)

The inner activation is denoted with �*σ*, which is the symbol we use for non-linear activation functions.

## Universal approximation theorem

According to the universal approximation theorem, given enough neurons and the correct set of weights, a multi-layer NN can approximate any function. Learning this function is increasingly hard, and we have no guarantee that our data are enough to do so.

Admittedly, that doesn’t mean we should only use NNs.

In fact, we will learn about other models and how we can make a NN more compact, wider, or deeper to learn very rich data representations.

Why is that even useful?

Because NNs hide another secret besides being very good function approximators. They are also very good feature extractors.

## Deep neural networks as feature extractors

Feature extraction can be seen as the transformation of the input data points from the input space to the feature space where classification is much easier.

Here is an intuitive and oversimplified example:

Imagine that each data point has 70 dimensions. Finding the correct 70-dimensional function to distinguish the data into two categories is very difficult and time consuming.

Instead, we transform our input to a three-dimensional space where a classifier can approximate the decision boundary more easily. If we transform the 3D decision boundary back to the 70-dimensional space, we will see that it corresponds to a 70-dimensional decision boundary.

The transformed space does not always need to be low-dimensional, but high-dimensional spaces do not guarantee better results either.

Think of the 70-dim example: if one of these input dimensions refers to the label, it would be enough to have 100% accuracy.

In any case, this is the main reason Deep Neural Networks (DNNs) exist: to transform the input data into a “better” space. Better because we can classify the data more easily after we transform them!

In fact, in most real-life applications, only the last one or two layers of a neural network performs the actual classification. The rest account for feature extraction and learning representations.

# Backpropagation Algorithm

Take a look at the mathematics of the backpropagation algorithm.

**We'll cover the following**

* [Notations](https://www.educative.io/module/page/qjv3oKCzn0m9nxLwv/10370001/5200716586549248/6677290011852800#Notations)
* [Forward pass](https://www.educative.io/module/page/qjv3oKCzn0m9nxLwv/10370001/5200716586549248/6677290011852800#Forward-pass)
* [Backward pass](https://www.educative.io/module/page/qjv3oKCzn0m9nxLwv/10370001/5200716586549248/6677290011852800#Backward-pass)
* [The chain rule for the backward pass](https://www.educative.io/module/page/qjv3oKCzn0m9nxLwv/10370001/5200716586549248/6677290011852800#The-chain-rule-for-the-backward-pass)

Neural Networks (NN) are non-linear classifiers that can be formulated as a series of matrix multiplications. Just like linear classifiers, they can be trained using the same principles we followed before, namely the gradient descent algorithm. The difficulty arises in computing the gradients.

But first things first.

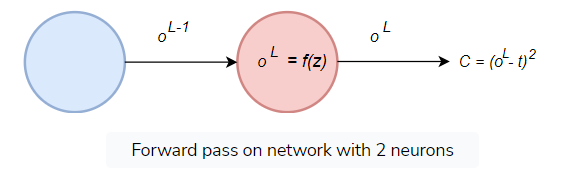
Let’s start with a straightforward example of a two-layered NN, with each layer containing just one neuron.

## Notations

* The superscript defines the layer that we are in.
* ��*oL* denotes the activation of layer L.
* ��*wL* is a scalar weight of the layer L.
* ��*bL* is the bias term of layer L.
* �*C* is the cost function, �*t* is our target class, and �*f* is the activation function.

## Forward pass

Our lovely model would look something like this in a simple sketch:



We can write the output of a neuron at layer �*L* as:

��=�(����−1+��)*oL*=*f*(*wLoL*−1+*bL*)

To simplify things, let’s define:

��=����−1+��*zL*=*wLoL*−1+*bL*

so that our basic equation will become:

��=�(��)*oL*=*f*(*zl*)

We also know that our loss function is:

�=(��−�)2*C*=(*oL*−*t*)2

This is the so-called forward pass. We take some input and pass it through the network. From the output of the network, we can compute the loss �*C*.

## Backward pass

Backward pass is the process of adjusting the weights �*w* in all the layers to minimize the loss �*C*.

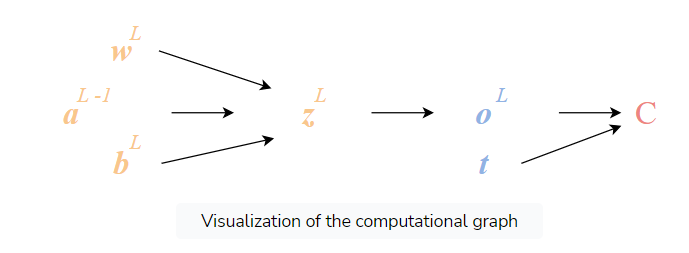
To adjust the weights based on the training example, we can use our known **update rule**:

���=��−1�−�∗∂�∂��*wtL*​=*wt*−1*L*​−*λ*∗∂*wL*∂*C*​

where �*λ* is the learning rate that scales down the gradient.

It should be clear by now that the only thing left to compute is the gradient ∂�∂��∂*wL*∂*C*​ (the derivative of the loss with respect to the weight).

One way to think about computing ∂�∂��∂*wL*∂*C*​ is through the following diagram, which is called ***computational graph***:



We summarize the performed operation in this way. To convert this into math, we need to revisit the chain rule.

## The chain rule for the backward pass

To compute the gradient ∂�∂��∂*wL*∂*C*​, our most useful tool is calculus and the chain rule. Using both, we can write:

∂�∂��=∂��∂��∗∂��∂��∗∂�∂��∂*wL*∂*C*​=∂*wL*∂*zL*​∗∂*zL*∂*oL*​∗∂*oL*∂*C*​

It is evident that the final gradient is affected by the gradients of the previous neuron, which in turn is affected by the gradients of the one before. You can see that in order to compute the gradient, we need to go back (through the chain rule) all the way to the beginning of the network.

In other terms, we need to propagate the error backwards. This is how the backpropagation algorithm got its name.

To find the gradients, let’s compute all the subgradients. By using basic calculus, we get:

�=(��−�)2−>∂�∂��=2(��−�)*C*=(*oL*−*t*)2−>∂*oL*∂*C*​=2(*oL*−*t*)

��=�(����−1+��)=�(��)−>∂��∂��=�′(��)*oL*=*f*(*wLoL*−1+*bL*)=*f*(*zL*)−>∂*zL*∂*oL*​=*f*′(*oL*)

��=����−1+��−>∂��∂��=��−1*zL*=*wLoL*−1+*bL*−>∂*wL*∂*zL*​=*oL*−1

Combining them all together, we acquire our final gradient:

∂�∂��=��−1∗�′(��)∗2(��−�)∂*wL*∂*C*​=*oL*−1∗*f*′(*oL*)∗2(*oL*−*t*)

Similar equations can be derived for the biases. Instead of ∂��∂��∂*wL*∂*zL*​ , we would have:

��=����−1+��−>∂��∂��=1*zL*=*wLoL*−1+*bL*−>∂*bL*∂*zL*​=1

For completion, if we do the math, we get:

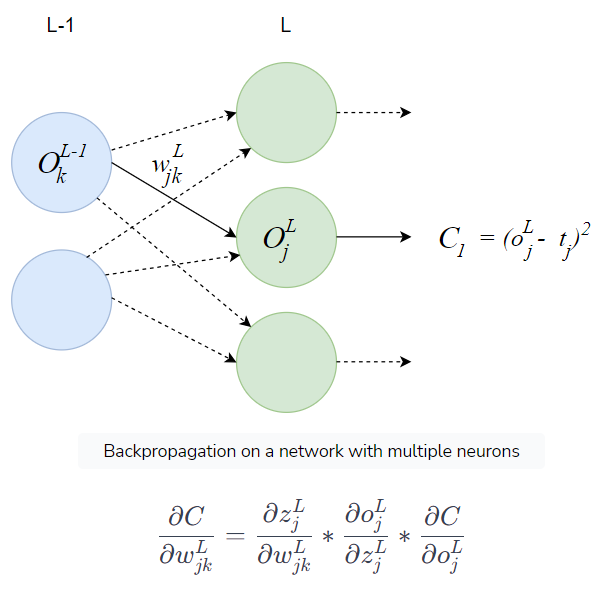
∂�∂��=�′(��)∗2(��−�)∂*bL*∂*C*​=*f*′(*oL*)∗2(*oL*−*t*)

Now, we can adjust the weight and biases based on a single training example based on the update rule:

���=��−1�−�∗∂�∂��*wtL*​=*wt*−1*L*​−*λ*∗∂*wL*∂*C*​

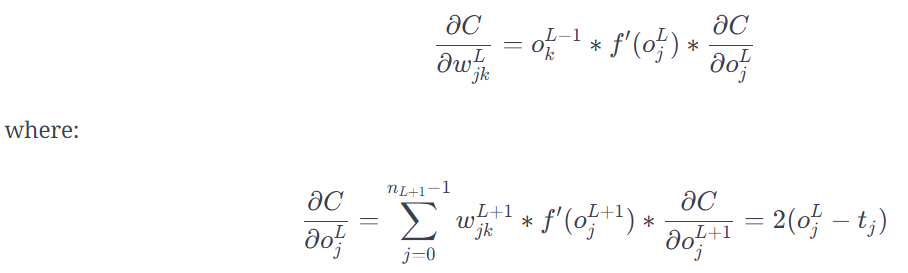
Next, we’ll feed the next example and readjust, and repeat. This is the infamous backpropagation.

You might argue that this is oversimplified because we only have 1 neuron. To be honest, not much will change if we add more neurons per layer. We will essentially conclude to the same equation.



Here �*j* is a neuron in the layer �*L*, while k a neuron in the layer �−1*L*−1.

And if we want to present the derivative in its final form, we have:



Two final things to note here:

* The derivative with respect to the activation is a summation due to the fact that the activation of a neuron now depends on the activations of all the neurons on the previous layer.
* The same derivative also depends on the derivatives of the next layer’s activation (backpropagation of the error).

You now have a sense of how NNs learn, and that is no easy task.

**Important note**: We will not be computing gradients in every network that we define. The gradients are computed automatically in modern frameworks such as PyTorch.

No more partial derivatives!

# Build a Neural Network With Pytorch

Implement a vanilla neural network from scratch using Pytorch.

**We'll cover the following**

* [PyTorch basics](https://www.educative.io/module/page/qjv3oKCzn0m9nxLwv/10370001/5200716586549248/6212991766233088#PyTorch-basics)
* [Build a neural network](https://www.educative.io/module/page/qjv3oKCzn0m9nxLwv/10370001/5200716586549248/6212991766233088#Build-a-neural-network)
* [Program your own neural network](https://www.educative.io/module/page/qjv3oKCzn0m9nxLwv/10370001/5200716586549248/6212991766233088#Program-your-own-neural-network)

## PyTorch basics

In the previous lessons, we looked at some Pytorch code, but this time we will take a closer look at it.

[Pytorch](https://pytorch.org/) is an open-source Python deep learning framework that enables us to build and train neural networks. Pytorch is a library that helps you perform mathematical operations between matrices in their basic form because, as you will realize, deep learning is just simple linear algebra.

The fundamental building block of a Pytorch is the **tensor**. A tensor is an N-dimensional array. We can have an 1d array (or a vector) x=[1,2,3,4,5], a 2d-array y=[[1,2],[3,4]], and so on.

In Pytorch, these can be defined as:

X= torch.tensor([ 1,2,3,4,5])   
Y= torch.tensor([1,2],[3,4]])

From there we can define almost all mathematical operations between tensors.

Z = torch.add(X,Y)   
Z = torch.matmul(X,Y)   
Z = 1 / (1+torch.exp(x))

Let’s revisit the neuron’s equation:  *a*=*f*(*w*1​∗*a*1​+*w*2​∗*a*2​+*w*3​∗*a*3​+*bo*​)

The above equation can be easily transformed into tensor operations (remember everything in deep learning can be represented as tensors).

[a\_4] = f([ w1, w2, w3] \* [a1, a2,a3 ] +[b\_o])

All we did here is this:

* Gather together all activations into a 1-d vector and all weights into another 1-d vector.
* Multiply them.
* Add the bias.
* Apply the sigmoid function in the result.

Note that individual numbers can also be seen as 0d tensors.

Let’s proceed with our first exercise. Using everything that you learned just now, try to program your first neuron from scratch. All necessary information and commands have already been mentioned, so all you have to do is reconstruct the above equation using Pytorch.

As a first exercise, try to code a simple neuron with three inputs in Pytorch. Initialize the weights as [0.5,0.5,0.5], the bias as 0.5, and return the output.

import torch

def neuron(input):

  wts=torch.tensor([0.5,0.5,0.5])

  b=torch.tensor([0.5])

  return torch.add(torch.matmul(input,wts),b)

if \_\_name\_\_ == "\_\_main\_\_":

  input=torch.tensor([1.0,2.0,3.0])

  output=neuron(input)

  print('output --> ',output)

## Build a neural network

Luckily, for us, Pytorch provides lots of ready functions so that we don’t have to build each neuron from scratch. For example, if we want to declare a layer of neurons, we can use the premade function as follows:

linear1 = nn.Linear(5, 20)

The above command constructs a layer that inputs a 5-sized vector and outputs a 20-sized vector.

To develop a neural network, we can use the following function, which defines a sequential order of individual layers:

nn.Sequential(   
        nn.Linear(2, 3),   
        nn.Sigmoid(),   
        nn.Linear(3, 2),   
        nn.Sigmoid()   
    )

Using five lines of code, we build a neural network that has two inputs, a hidden layer with three neurons and two outputs. Note that the linear layer does not contain the activation function, so we have to explicitly declare them as well.

## Program your own neural network

Now that you are more familiar with building a neural network in Pytorch, let’s see what’s going on under the hood. We will come back to the 2-3-2 NN for simplicity.

The following is another way to define a NN in Pytorch:

class Model(nn.Module):   
    def \_\_init\_\_(self):   
        super(Model, self).\_\_init\_\_()   
        self.linear1 = nn.Linear(2, 3)   
        self.linear2 = nn.Linear(3, 2)   
    
    def forward(self, x):   
        h = torch.sigmoid(self.linear1(x))   
        o = torch.sigmoid(self.linear2(h))   
        return o

Many people prefer it because it gives more control and explainability over the network. To run a forward propagation, we can create a random input, initialize the model, and pass the input as an argument as follows:

model= Model()   
X = torch.randn((1, 5))   
Y = model(X)

After placing print statements between all layers, we can inspect the state of our NN.

Our input is:

tensor([0.5000, 0.5000])

While the first layer’s weights are:

tensor([[-0.3580, -0.4130],   
        [ 0.5652,  0.6722],   
        [ 0.4894,  0.4164]]

Lastly, the first layer’s output is:

tensor([0.5555, 0.4797, 0.5787])

Now, it is your turn to build a neural network. Let’s have it receive a vector with 10 numbers, have 3 linear layers with dimensions 128, 64, 2, and two RELU layers in between.

import torch

import torch.nn as nn

seed = 172

torch.manual\_seed(seed)

torch.cuda.manual\_seed(seed)

def fnn(input):

    ### WRITE YOUR ANSWER HERE

    model = nn.Sequential(nn.Linear(10, 128),

                        nn.ReLU(),

                        nn.Linear(128, 64),

                        nn.ReLU(),

                        nn.Linear(64, 2)

                    )

    return model(input)

if \_\_name\_\_ == "\_\_main\_\_":

    input=torch.randn((1, 10))

    out = fnn(input)

    print(out)