Deep Learning and Convolutional Neural Networks

efore we begin this chapter, we need to talk a bit about AI and **machine learning** (**ML**) and how those two components fit together. The term "artificial" refers to something that is not real or natural, whereas "intelligence" refers to something capable of understanding, learning, or able to solve problems (and, in extreme cases, being self-aware).

Officially, artificial intelligence research began at the Dartmouth Conference of 1956 where AI and its mission were defined. In the following years, everyone was optimistic as machines were able to solve algebra problems and learn English, and the first robot was constructed in 1972. However in the 1970s, due to overpromising but under delivering, there was a so-called AI winter where AI research was limited and underfunded. After this though AI was reborn through expert systems, that could display human-level analytical skills. Afterwards, a second AI winter machine learning got recognized as a separate field in the 1990s when probability theories and statistics started to be utilized.

Increases in computational power and the determination to solve specific problems led to the development of IBM’s Deep Blue that beat the world chess champion in 1997 . Fast forward and nowadays the AI landscape encompasses many fields including Machine Learning, Computer Vision, Natural Language Processing, Planning Scheduling, and Optimization, Reasoning/Expert systems, and Robotics.

During the past 10 years, we have witnessed a huge transformation in what ML, and AI in general, is capable of. Thanks mainly to Deep Learning.

In this chapter, we are going to cover the following topics:

- A general explanation of the concepts of AI and ML
- Artificial neural networks and Deep Learning
- **Convolutional neural networks** (**CNN**s) and their main building blocks
- Using TensorFlow to build a CNN model to recognize images of digits
- An introduction to Tensorboard

AI and ML

For the purpose of this book, consider **artificial intelligence** (**AI**) as the field of computer science responsible for making agents (software/robots) that act to solve a specific problem. In this case, "intelligent" means that the agent is flexible and it perceives its environment through sensors and will take actions that maximize its chances to succeed at some particular goal.

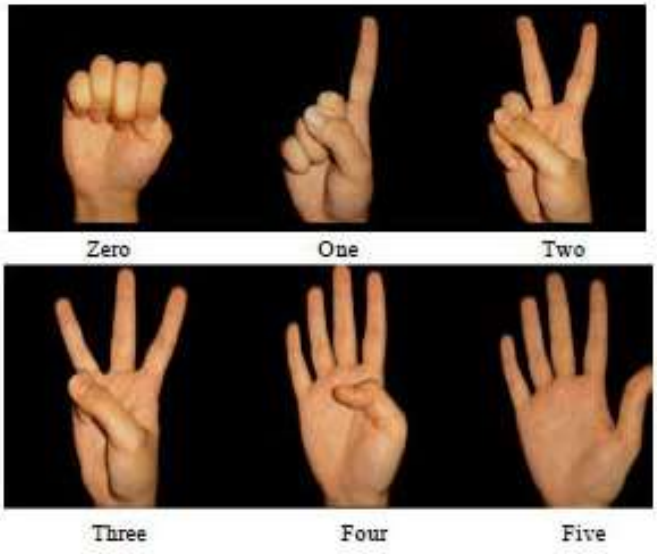
We want an AI to maximize something that is named **Expected Utility** or the probability of getting some sort of satisfaction by doing an action. An easy to understand example of this is by going to school, you will maximize your expected utility of getting a job.

AI aspires to replace the error-prone human intelligence involved in completing tedious everyday tasks. Some central components of human intelligence that AI aims to mimic (and an intelligent agent should have) are:

- **Natural Language Processing** (**NLP**): Give the ability to understand spoken or written human language and give natural response to questions. Some example NLP tasks include automated narration, machine translation, or text summarization.
- **Knowledge and Reasoning**: Develop and maintain an updated knowledge of the world around the agent. Follow human reasoning and decision-making to solve specific problems and react to changes in its environment.
- **Planning and problem solving**: Making predictions about possible actions and choosing the one that maximizes some expected utility, in other words, choosing the best action for that situation.
- **Perception**: The sensors that the agent is equipped with provides it with information about the world that the agent lives in. The sensors could be something as simple as an infrared sensor or more complicated such as a microphone for speech recognition or a camera to enable machine vision.
- **Learning**: For the agent to develop knowledge of the world, it must use perception to learn through observation. Learning is a way of knowledge acquisition that will be used to reason and make decisions. The subfield of AI that deals with algorithms that learn from data without some explicit programming is named Machine Learning.

ML uses *tools* such as statistical analysis, probabilistic models, decision trees, and neural networks to process efficiently large amounts of data instead of humans.

As an example, let's consider the following problem of gesture recognition. In this example, we want our machine to identify what hand gesture is being shown. The inputs to the system are hand images, as shown in the following image, and the output is the digits that they represent. The system that would solve this problem needs to use perception in the form of vision.



Giving just raw images as input to our machine would not produce a reasonable result. Therefore, the images should be preprocessed to extract some kind of interpretable abstraction. In our particular case, the simplest approach would be to segment the hand based on color and make a vertical projection summing the non-zero values on the *x*-axis. If the width of the image is 100 pixels, then the vertical projection forms a vector 100-elements long (100-dimensional), with the highest values at the location of the unfolded fingers. We can call any vector of features, that we extract, a **feature vector**.

Let’s say that for our hand data, we have 1000 different images and we have now processed them to extract feature vectors for each. In the machine learning stage, all the feature vectors will be given to a machine learning system that creates a model. We hope that this model can generalize and is able to predict the digit for any future images given to the system that it wasn’t trained on.

An integral part of an ML system is evaluation. When we evaluate our model, we see how well our model has done in a particular task. In our example, we would look at how accurately it can predict the digit from the image. Accuracy of 90% would mean that 90 out of 100 given images were correctly predicted. In the chapters that follow, we will discuss in more detail the machine training and evaluation process.

Types of ML

ML problems can be separated into three major groups depending on what kind of data is available to us and what we want to accomplish:

Supervised learning: Both the input and desired output or label are available to us. Hand gesture classification, where we are given images of hand gestures and corresponding labels, is an example of a supervised learning problem. We want to create a model that is able to output the correct label given an input hand image.

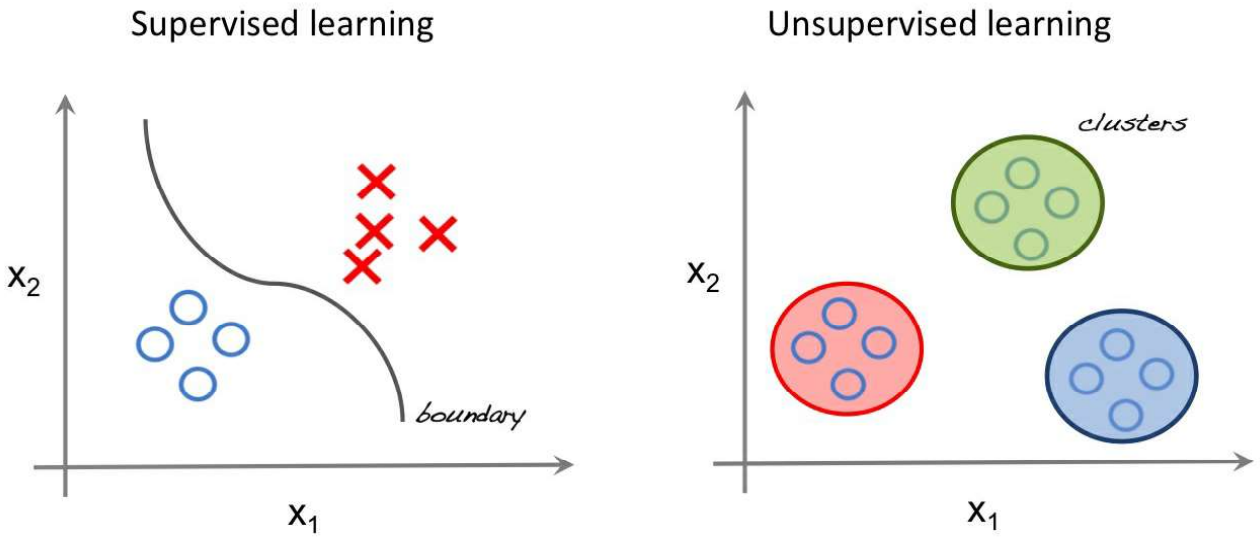
Supervised techniques include SVM, LDA, neural networks, CNN, K-NN, the decision tree, and so on.

Unsupervised learning: Only the inputs are available with no labels, and we don't necessarily know what we want our model to do. An example would be if we are given a large dataset containing pictures of hands, but no labels. In this case, we might know that there is some structure or relationships in the data, but we leave it up to an algorithm to try and find them within our data for us. We might want our algorithm to find clusters of similar hand gestures in our data, so we don't have to manually label them.

Another use of unsupervised learning is finding ways of reducing the dimension of the data we are using, again by finding important features in our data and discarding unimportant ones.

Unsupervised techniques include PCA, t-SNE, K-means, autoencoders, deep autoencoders, and so on.

The following image illustrates the difference between classification and clustering (when we need to find structure on unsupervised data).



Reinforcement learning: The third kind is all about training an agent to perform some action in an

environment. We know the desired outcome, but not how to get to it. Rather than having labeled data, we give the agent feedback telling it how good or bad it is at accomplishing the task. Reinforcement learning is out of the scope of this book.

Old versus new ML

The typical flow that an ML engineer might follow to develop a prediction model is as follows:

1. Gather data
2. Extract relevant features from the data
3. Choose an ML architecture (CNN, ANN, SVM, decision trees, and so on)
4. Train the model
5. Evaluate the model and repeat steps 3 to 5 until they find a satisfying solution
6. Test the model in the field

As mentioned in the previous section, the idea of ML is to have an algorithm that is flexible enough to learn the underlying process behind the data. This being said, many classic methods of ML are not strong enough to learn directly from data; they need to somehow prepare the data before using those algorithms.

We briefly mentioned it before, but this process of preparing the data is often called feature extraction, where some specialist filters out all the details of the data that we believe are relevant to its underlying process. This process makes the classification problems easier for the selected classifier as it doesn’t have to work with irrelevant variables in the data that it might otherwise see as important.

The single coolest feature that new deep learning methods of ML have is they don't need (or need less of) the feature extraction phase. Instead, using large enough datasets, the model itself is capable of learning what are the best features to represent the data, directly from the data itself! The examples of these new methods are as follows:

- Deep CNNs
- Deep AutoEncoders
- **Generative Adversarial Networks** (**GAN**s)

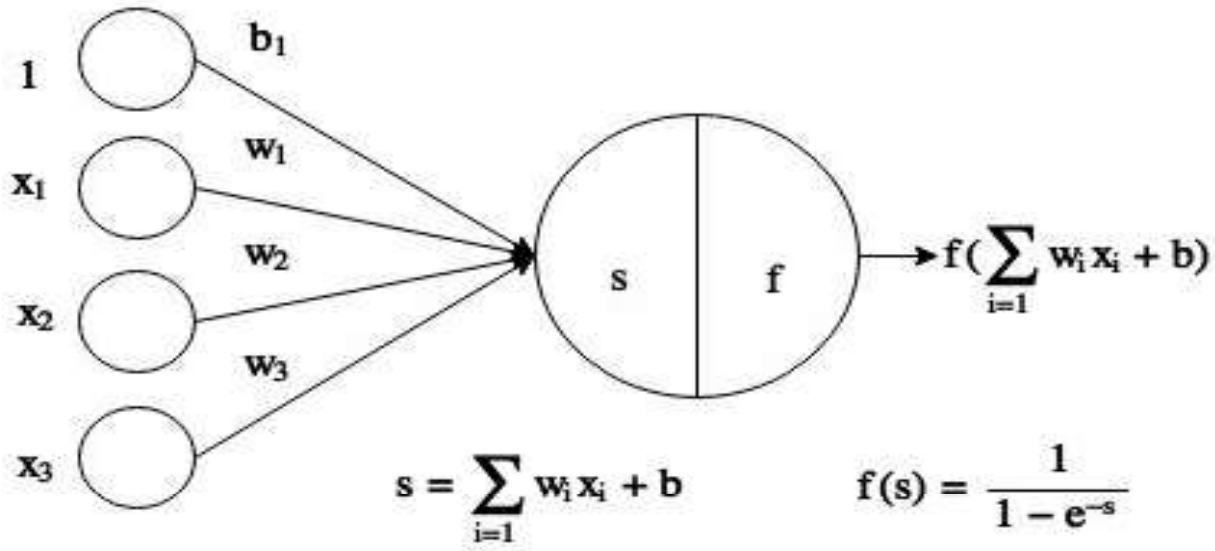
All these methods are a part of the deep learning process where vast amounts of data are exposed to multilayer neural networks. However, the benefits of these new methods come at a cost. All these new algorithms require much more computing resources (CPU and GPU) and could take much longer to train than traditional methods.

Artificial neural networks

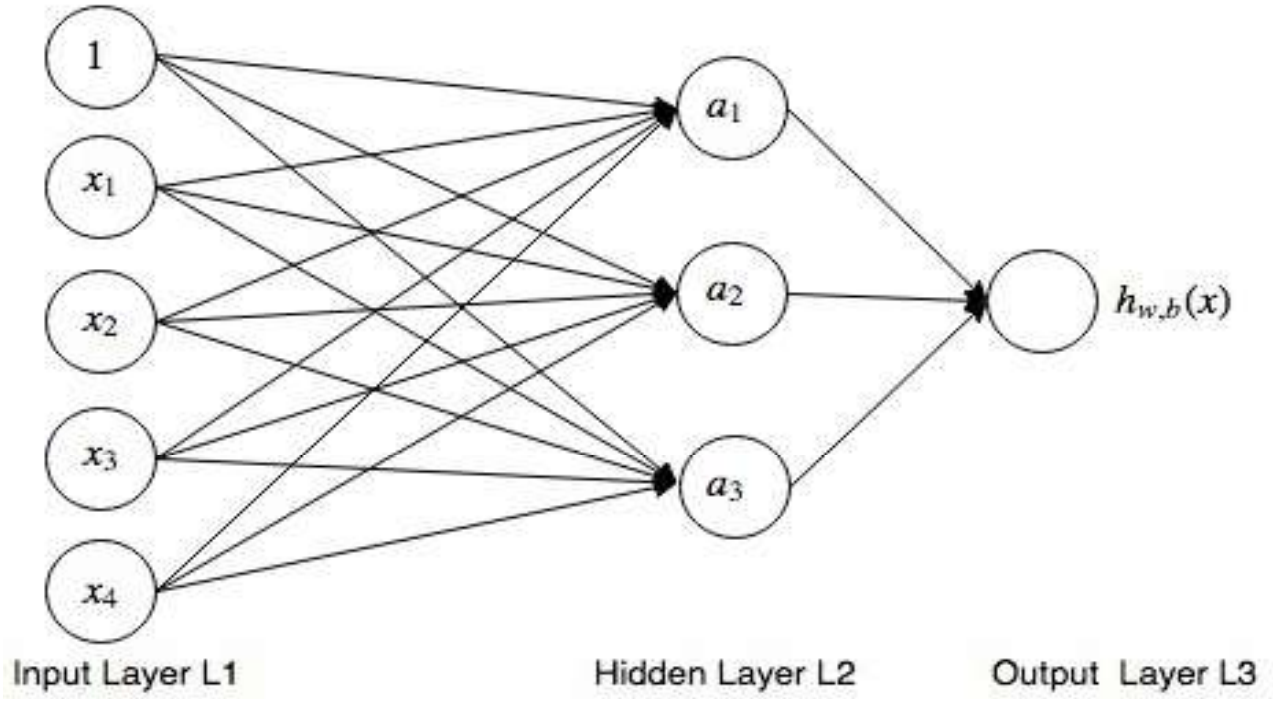
Very vaguely inspired by the biological network of neurons residing in our brain, **artificial neural networks** (**ANNs**) are made up of a collection of units named **artificial neurons** that are organized into the following three types of layers:

- Input layer
- Hidden layer
- Output layer

The basic artificial neuron works (see the following image) by calculating a dot product between an input and its internal *weights*, and the results is then passed to a nonlinear activation function *f* (sigmoid, in this example). These artificial neurons are then connected together to form a network. During the training of this network, the aim is to find the proper set of weights that will help with whatever task we want our network to do:



Next, we have an example of a 2-layer feed forward artificial neural network. Imagine that the connections between neurons are the weights that will be learned during training. In this example, Layer *L1* will be the input layer, *L2* the hidden layer, and *L3* the output layer. By convention, when counting the number of layers, we only include layers that have learnable weights; therefore, we do not include the input layer. This is why, it is only a 2-layer network:



Neural networks with more than one layer are examples of nonlinear hypothesis, where the model can learn to classify much more complex relations than linear classifiers can. In fact, they are actually universal approximators capable of approximating any continuous function.

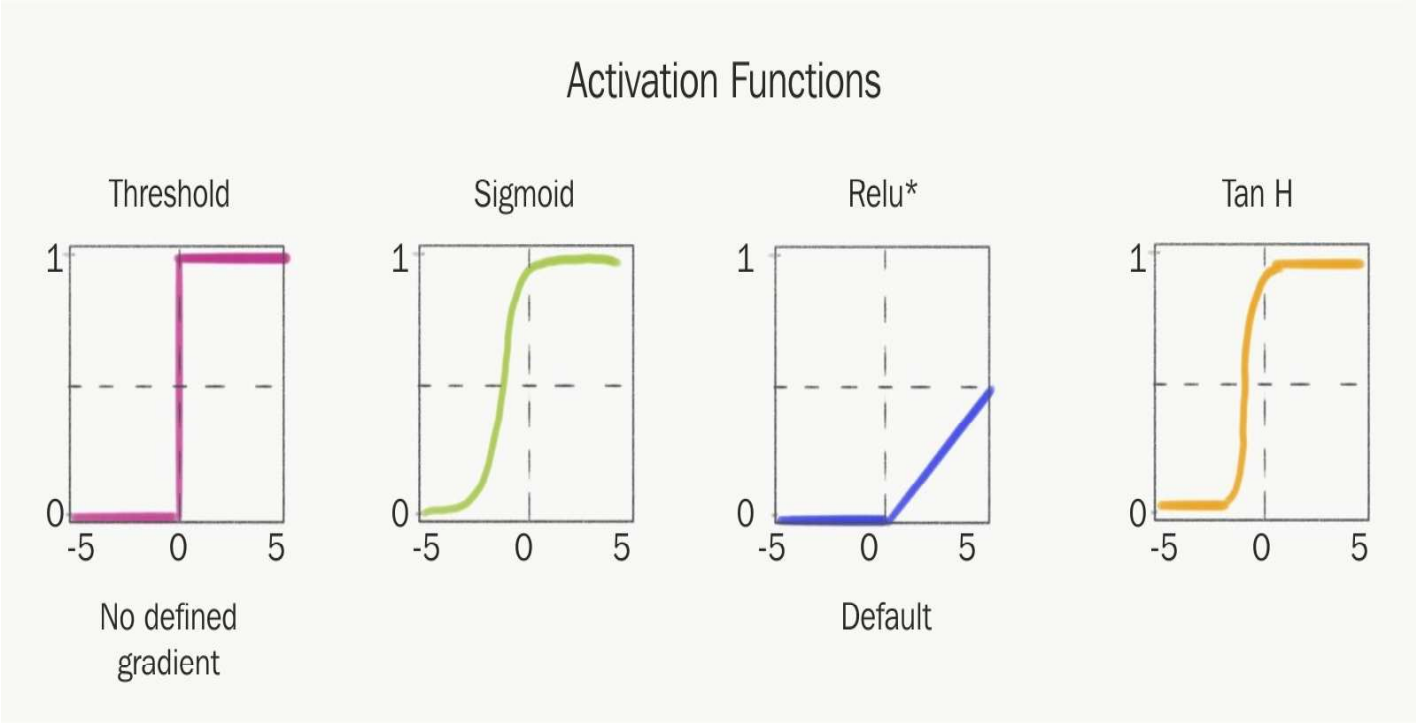
Activation functions

In order to allow the ANN models to be able to tackle more complex problems, we need to add a nonlinear block just after the neuron dot product. If we then cascade these nonlinear layers, it allows the network to compose different concepts together, making complex problems easier to solve.

The use of nonlinear activations in our neurons is very important. If we didn't use nonlinear activation functions, then no matter how many layers we cascaded we would only ever have something that behaves like a linear model. This is because any linear combination of linear functions collapses down to be a linear function.

There are a wide variety of different activation functions that can be used in our neurons, and some are shown here; the only important thing is that the functions are nonlinear. Each activation function has its own benefits and disadvantages.

Historically, it was Sigmoid and *TanH* that were the activation functions of choice for neural networks. However, these functions turned out to be bad for reliably training neural networks as they have the undesirable property that their values saturate at either end. This causes the gradients to be zero at these points, which we will find out later, and it is not a good thing when training a neural network.



As a result, one of the more popular activation functions is the ReLU activation or **Rectified Linear Unit**. ReLU is simply a max operation between an input and 0 - max(x,0). It has the desirable property that gradients (at least at one end) will not become zero, which greatly helps the speed of convergence

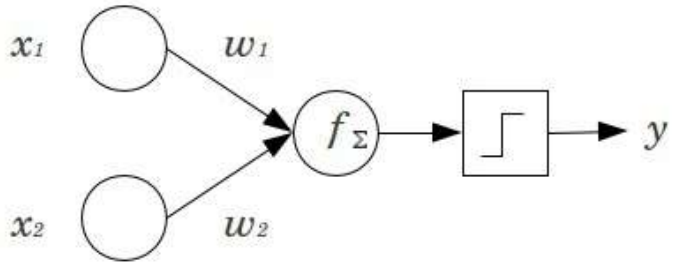
for neural network training.

This activation function gained popularity after it was used to help train deep CNNs. It's simplicity and effectiveness make it generally the go-to activation function to use.

The XOR problem

To explain the importance of depth in an ANN, we will look at a very simple problem that an ANN is able to solve because it has more than one layer.

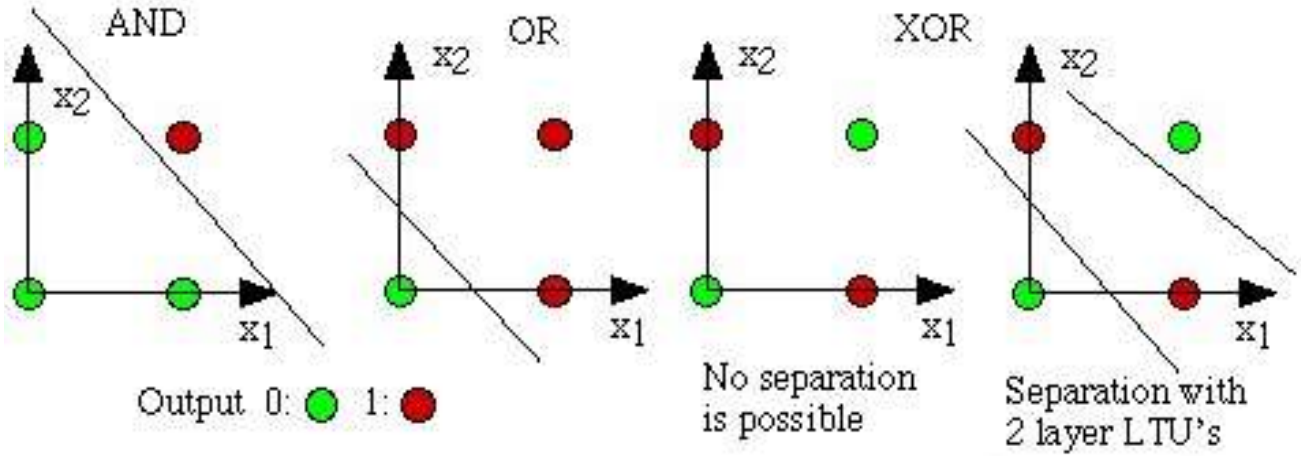
In the early days of working with artificial neurons, people did not cascade layers together like we do in ANNs, so we ended up with a single layer that was named a perceptron:



The perceptron is effectively just a dot product between an input and a set of learned weights, which means that it is actually just a linear classifier.

It was around the time of the first AI winter that people realized the weaknesses of the perceptron. As it is just a linear classifier, it is not able to solve simple nonlinear classification problems such as the Boolean exclusive-or (XOR) problem. To solve this issue, we needed to go deeper.

In this image, we see some different boolean logic problems. A linear classifier can solve the AND and OR problems but is not able to solve the XOR:



This led people to have the idea of cascading together layers of neurons that use nonlinear activations. A layer could create nonlinear concepts based on the output of the previous layer. This “composition of concepts” allows networks to be more powerful and to represent more difficult functions, and consequently, they are able to tackle nonlinear classification problems.

Training neural networks

So how do we go about setting the values of the weights and biases in our neural network that will best solve our problem? Well this is done in something called the training phase. During this phase, we want to make our neural network “learn” from a training dataset. The training dataset consists of a set of inputs (normally denoted as X) along with corresponding desired outputs or labels (normally denoted as Y).

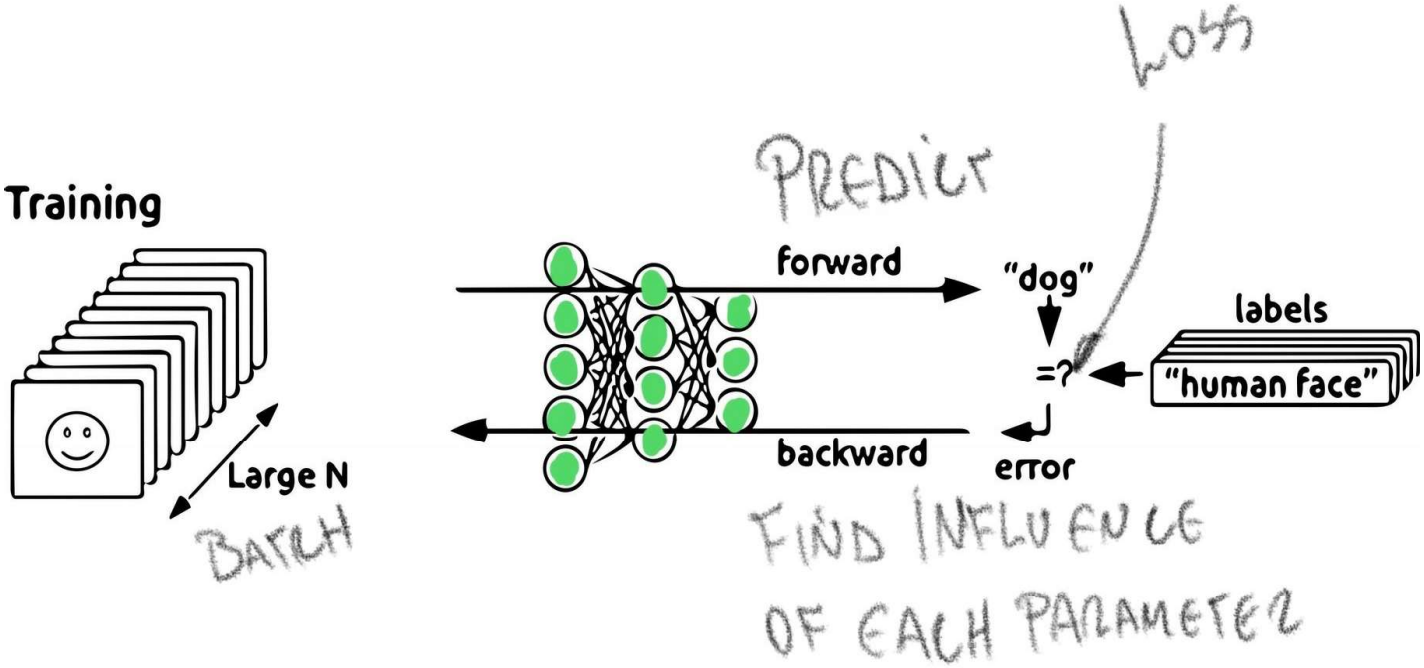
When we say the network learns, all that is happening is the network parameters get updated in such a way that the network should be able to output the correct Y for every X in the training dataset. The expectation is that after the network is trained, it will be able to generalize and perform well for new inputs not seen during training. However, in order for this to happen, you must have a dataset that is representative enough to capture what you want to output. For example, if you want to classify cars, you need to have a dataset with different types, colors, illumination, and so on.

One common mistake when training machine learning models in general occurs when we do not have enough data or our model is not complex enough to capture the complexity of the data. These mistakes can lead to the problems of overfitting and underfitting. You will learn how to deal in practice with these problems in future chapters.

During training, the network is *executed* in two distinct modes:

- **Forward propagation**: We work forward through the network producing an output result for a current given input from the dataset. A loss function is then evaluated that tells us how well the network did at predicting the correct outputs.
- **Backward propagation**: We work backward through the network calculating the impact each weight had on producing the current loss of the network.

This image shows the two different way the network is run when training.



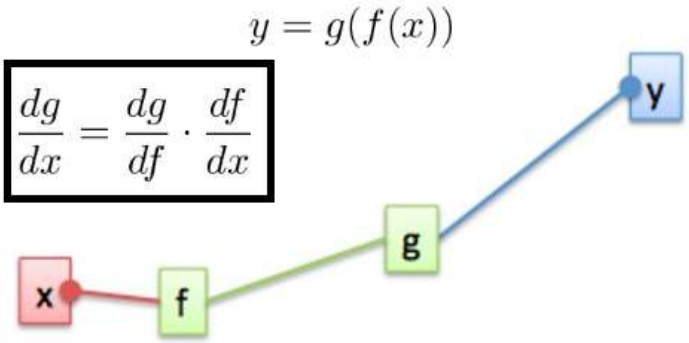
Currently, the workhorse for making neural networks "learn" is the backpropagation algorithm combined with a gradient-based optimizer like gradient descent.

Backpropagation is used to calculate gradients that tell us what effect each weight had on producing the current loss. After gradients are found an optimization technique such as gradient descent uses them to update the weights in such a way that we can minimize the value of the loss function.

Just a closing remark: ML libraries such as TensorFlow, PyTorch, Caffe, or CNTK will provide the backpropagation, optimizers, and everything else needed to represent and train neural networks without the need to rewrite all of this code yourself.

Backpropagation and the chain rule

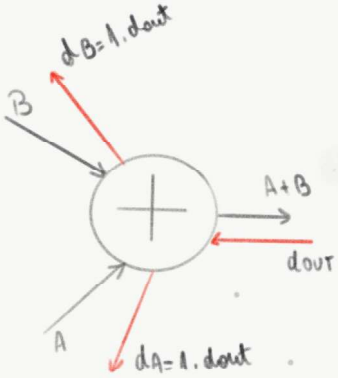
The backpropagation algorithm is really just an example of the trusty chain-rule from calculus. It states how to find the influence of a certain input, on systems that are composed of multiple functions. So for example in the image below, if you want to know the influence of *x* on the function *g*, we just multiply the influence of *f* on g by the influence of *x* on *f*:

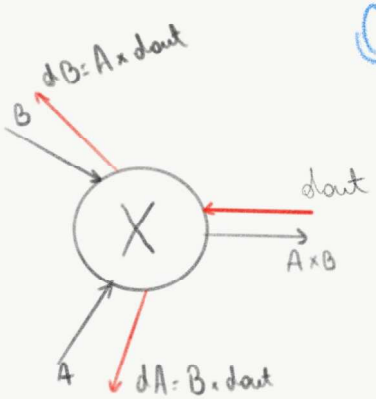


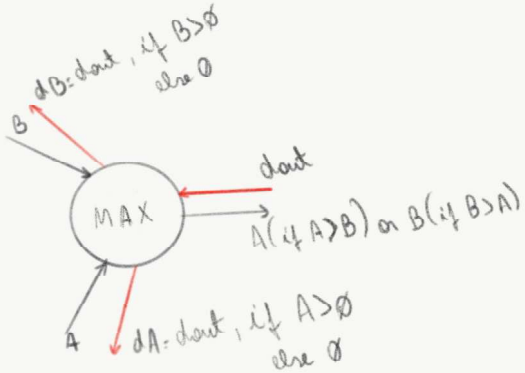
Also, this means that if we would like to implement our own deep learning library, we need to define the layers normal computation (forward propagation) and also the influence (derivative) of this computation block relative to its inputs.

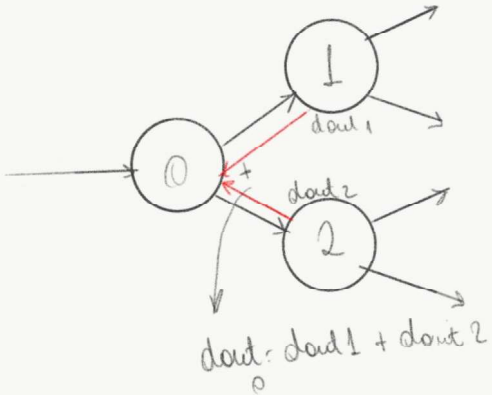
Below we give some common neural network operations and what their gradients are.

OPS GRADIENTS









```python
class MultiplyGate(object):
    # Implement Multiply gate
    def forward(self,x,y):
        z = x*y
        self.x = x;
        self.y = y;
        return z

    # Observe that we return a gradient for each input
    def backward(self,dz):
        dx = self.y * dz
        dy = self.x * dz
        return [dx,dy]

class AddGate(object):
    # Implement Add gate
    def forward(self,x,y):
        z = x+y
        self.x = x;
        self.y = y;
        return z

    # Observe that we return a gradient for each input
    def backward(self,dz):
        dx = 1*dz
        dy = 1*dz
        return [dx,dy]
```

```
In [2]: import gatesUtils as gates

In [3]: gateMul = gates.MultiplyGate(); gateAdd = gates.AddGate();

In [4]: gateAdd.forward(5,6)
Out[4]: 11

In [5]: gateMul.forward(2,3)
Out[5]: 6

In [6]: gateMul.backward(2)
Out[6]: [6, 4]

In [7]: gateAdd.backward(3)
Out[7]: [3, 3]
```

Batches

The idea of having the whole dataset in memory to train networks, as the example in `Chapter 1`, *Setup and Introduction to TensorFlow*, is intractable for large datasets. What people do in practice is, during training, they divide the dataset into small pieces, named mini batches (or commonly just batches). Then, in turn, each mini batch is loaded and fed to the network where the backpropagation and gradient descent algorithms will be calculated and weights then updated. This is then repeated for each mini batch until you have gone through the dataset completely.

The gradient calculated for a mini-batch is a noisy estimate of the true gradient of the whole training set, but by repeatedly getting these small noisy updates, we will still eventually converge close enough to a good minimum of the loss function.

Bigger batch sizes give a better estimate of the true gradient. Using a larger batch size will allow for a larger learning rate. The trade-off is that more memory is required to hold this batch while training.

When the model has seen your entire dataset then we say that an epoch has been completed. Due to the stochastic nature of training you will want to train your models for multiple epochs as it is unlikely for your model to have converged in only one epoch.

Loss functions

During the training phase, we need to correctly predict our training set with our current set of weights; this process consists of evaluating our training set inputs X and comparing with the desired output Y. Some sort of mechanism is needed to quantify (return a scalar number) on how good our current set of weights are in terms of correctly predicting our desired outputs. This mechanism is named the **loss function.**

The backpropagation algorithm should return the derivative of each parameter with respect to the loss function. This means we find out how changing each parameter will affect the value of the loss function. It is then the job of the optimization algorithm to minimize the loss function, in other words, make the training error smaller as we train.

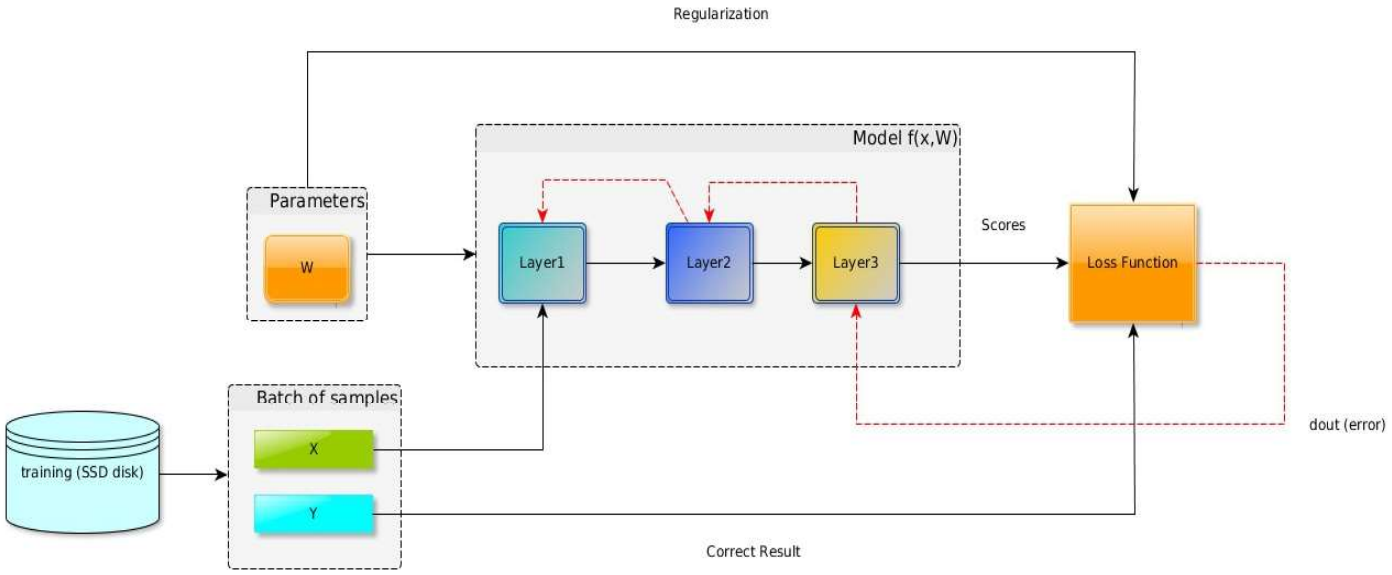
One important aspect is to choose the right loss function for the job. Some of the most common loss functions and what tasks they are used for are given here:

- **Log Loss -** Classification tasks (returning a label from a finite set) with only two possible outcomes
- **Cross-Entropy Loss -** Classification tasks (returning a label from a finite set) with more than two outcomes
- **L1 Loss -** Regression tasks (returning a real valued number)
- **L2 Loss -** Regression tasks (returning a real valued number)
- **Huber Loss -** Regression tasks (returning a real valued number)

We will see different examples of loss functions in action throughout this book.

Another important aspect of loss functions is that they need to be differentiable; otherwise, we cannot use them with backpropagation. This is because backpropagation requires us to be able to take the derivative of our loss function.

In the following diagram, you can observe that the loss function is connected at the end of the neural network (model output) and basically depends on the output of the model and the dataset desired targets.



This is also shown in the following line of code in TensorFlow as the loss only needs labels and outputs (called logits here).

```
loss = tf.losses.sparse_softmax_cross_entropy(labels=labels, logits=logits)
```

You probably noticed a third arrow also connected to the loss function. This is related to something named regularization, which will be explored in Chapter 3, *Image Classification in TensorFlow*; so for now, you can safely ignore it.

The optimizer and its hyperparameters

As mentioned before, the job of the optimizer is to update the network weights in a way that is going to minimize the training loss error. In all deep learning libraries such as TensorFlow, there is only really one family of optimizer used and that is the gradient descent family of optimizers.

The most basic of these is simply called gradient descent (sometimes called vanilla gradient descent), but more complex ones that try to improve on it have been developed. Some popular ones are:

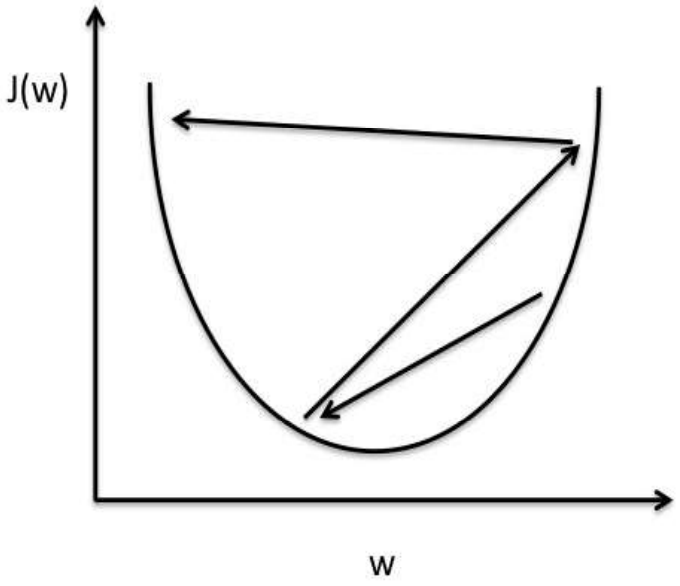
- Gradient descent with momentum
- RMSProp
- Adam

All of TensorFlow's different optimizers can be found in the `tf.train` class. For example the Adam optimizer can be used by calling `tf.train.AdamOptimizer()`.

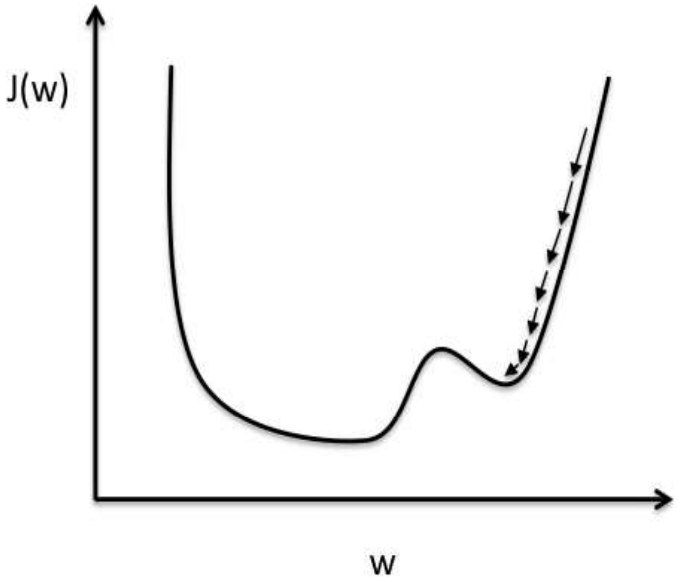
As you may suspect, they all have configurable parameters that control how they work, but usually the most important one to pay attention to and change is as follows:

- Learning rate: Control how quickly your optimizer tries to minimize the loss function. Set it too high and you will have problems converging to a minimum. Set it too small and it will take forever to converge or get trapped in a bad local minimum.

The following image shows the problems of having a badly chosen learning rate can have:



Large learning rate: Overshooting.



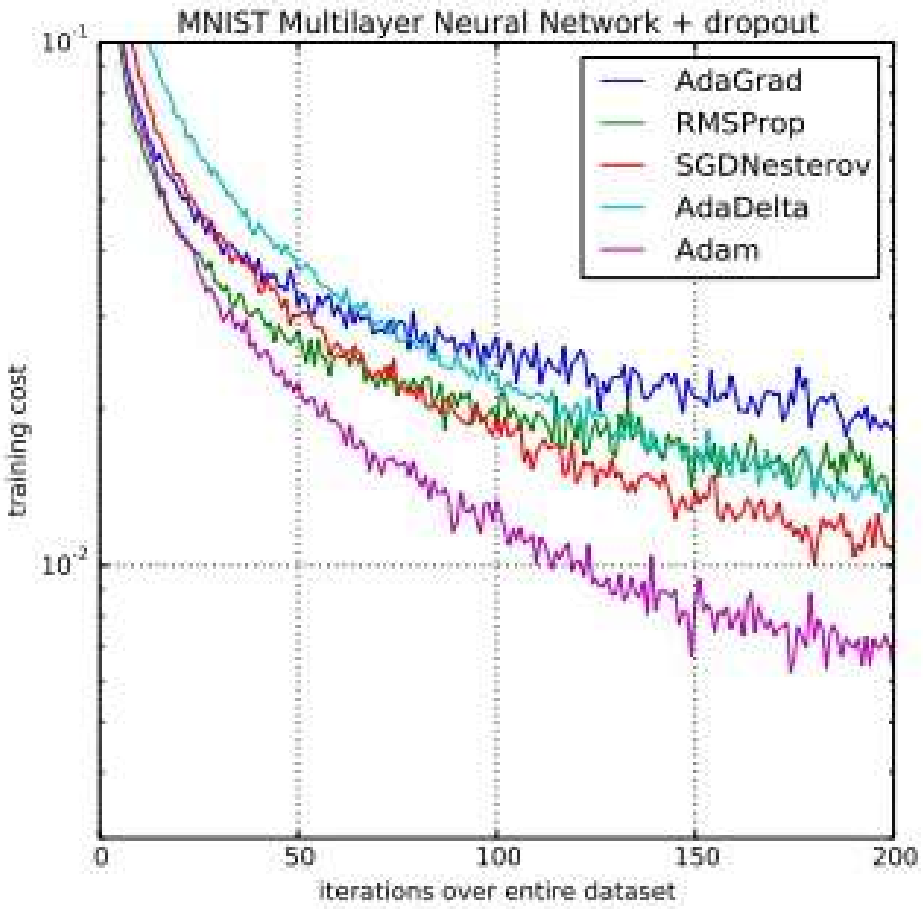
Small learning rate: Many iterations until convergence and trapping in local minima.

Another important aspect of the learning rate is that as your training progresses and the error drops, the learning rate value that you chose at the beginning of the training might become too big, and so you may start to overshoot the minimum.

To solve this issue, you may schedule a learning rate decay that from time to time decrease the learning rate as you train. This process is called **learning rate scheduling**, and there are several popular approaches that we will discuss in detail in the next chapter.

An alternative solution is to use one of the adaptive optimizers such as Adam or RMSProp. These optimizers have been designed so that they automatically adjust and decay the learning rates for all your model parameters as you train. This means that in theory you shouldn't have to worry about scheduling your own learning rate decay.

Ultimately you want to choose the optimizer that will train your network fastest and to the best accuracy. The following image shows how the choice of optimizer can affect the speed at which your network converges. There can be quite a gap between different optimizers, and this might change for different problems, so ideally if you can you should try out all of them and find what works best for your problem.



However, if you don't have time to do this, then the next best approach is to first try Adam as an optimizer as it generally works very well with little tuning. Then, if you have time, try SGD with Momentum; this one will take a bit more tuning of parameters such as learning rate, but it will generally produce very good results when well tuned.

Underfitting versus overfitting

When designing a neural network to solve a specific problem, we may have lots of moving parts, and have to take care of many things at the same time such as:

- Preparing your dataset
- Choosing the number of layers/number of neurons
- Choosing optimizer hyper-parameters

If we focus on the second point, it leads us to learn about two problems that might occur when choosing or designing a neural network architecture/structure.

The first of these problems is if your model is too big for the amount, or complexity, of your training data. As the model has so many parameters, it can easily just learn exactly what it sees in your training set even down to the noise that is present in the data. This is a problem because when the network is presented with data that is not exactly like the training set, it will not perform well because it has learned too precisely what the data looks like and has missed the bigger picture behind it. This issue is called **overfitting** or having **high-variance**.

On the other hand, you might choose a network that is not big enough to capture the data complexity. We now have the opposite problem, and your model is unable to capture the underlying structure behind your dataset well enough as it doesn't have the capacity (parameters) to fully learn. The network will again not be able to perform well on new data. This issue is called **underfitting** or having **high-bias**.

As you may suspect, you will always be looking for the right balance when it comes to your model complexity to avoid these issues.

In later chapters, we will see how to detect, avoid, and remedy those problems, but just for the sake of introduction, these are some of the classic ways to solve these issues:

- Getting more data
- Stopping when you detect that the error on the test data starts to grow (early-stopping)
- Starting the model design as simple as possible and only adding complexity when you detect underfitting

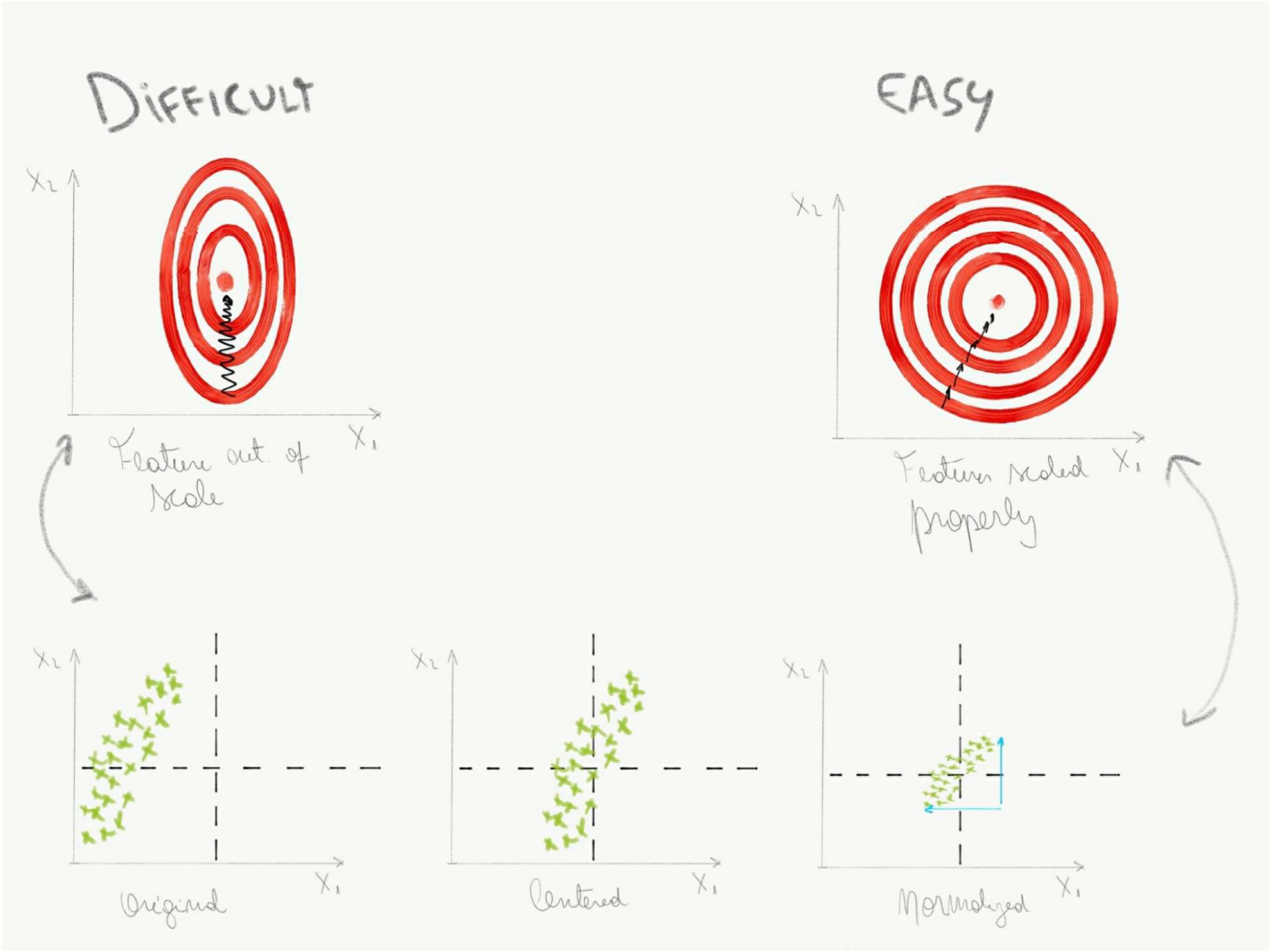
Feature scaling

In order to make the life of the optimizer algorithms easier, there are some techniques that can and should be applied to your data as an initial step before training and testing.

If the values on different dimensions of your input vector are out of scale with each other, your loss space will be somehow stretched. This will make it harder for the gradient descent algorithm to converge or at least make it slower to converge.

This normally happens when the features of your dataset are out of scale. For example, a dataset about houses might have "number of rooms" as one feature in your input vector that could have values between 1 and 4, whereas another feature might be "house area", and this could be between 1000 and 10000. Obviously, these are hugely out of scale of each other and this can make learning difficult.

In the following picture, we see a simple example of what our loss function might look like if our input features are not all in scale with each other and what it might look like when they are properly scaled. Gradient descent has a harder time reaching the minimum of the loss function when data is badly scaled.



Normally, you would do some standardization of the data, such as subtract the mean and divide by the standard deviation of your dataset before using it. In the case of RGB images, it's usually enough to just subtract 128 from each pixel value to center the data around zero. However, a better approach would be to calculate the mean pixel value for each image channel in your dataset. You now have three values, one for each image channel, which you now take away from your input images. We don't really need to worry about scaling when dealing with images as all features have the same scale (0-255) to start with.

Very very important to remember - if you do some preprocessing of your data at train time, you must do this exact same preprocessing at test time otherwise expect to get some bad results!

Fully connected layers

The layers of neurons that make up the ANNs that we saw earlier are commonly called densely connected layers, or **fully connected** (**FC**) layers or simply just linear layers. Some deep learning libraries such as Caffe would actually consider them just as the dot product operation that might or might not be followed by a nonlinearity layer. Its main parameter will be the output size, which will be basically the number of neurons in its output.

In `Chapter 1`, *Setup and Introduction to TensorFlow*, we created our own dense layer, but you can create it in an easier way using `tf.layers`, as follows:

```
dense_layer = tf.layers.dense(inputs=some_input_layer, units=1024, activation=tf.nn.relu)
```

Here, we defined a fully connected layer with 1,024 outputs, and it will be followed by a ReLU activation.

It is important to note that the input of this layer has to have just two dimensions, so if your input is a spatial tensor for example an image of shape [28*28*3] you will have to reshape it into a vector before inputting it :

```
reshaped_input_to_dense_layer = tf.reshape(spatial_tensor_in, [-1, 28 * 28 * 3])
```