Aprendizagem Profunda

Lecture Notes

Chapter 1

Introduction and course overview

Course objectives and structure. AI and the origin of Artificial Neural Networks. Machine Learning. The power of nonlinear transformations. What deep learning offers

Note: for details on assignments, class schedules and assessment, please refer to the course page

1.1 Course objectives

The goal of this course is to give you an introduction to deep learning. Here we will cover the foundations of deep neural networks, looking at different architectures and their applications, activation functions, optimizers and how to train deep neural networks in different contexts of supervised, unsupervised and reinforcement learning.

We will also see how to implement these networks in practice and optimize the results using regularization, data augmentation ans similar techniques, and selecting the best architecture.

Finally, this course will give you some practical experience in implementing these models in Tensorflow 2.0 and Keras, and also cover some open problems in deep learning such as interpretation and biases.

1.2 AI: historical overview

The Dartmouth Summer Research Project on Artificial Intelligence, in 1956, was a seminal event in AI. The goal of this project, according to the proposal presented by John McCarthy, Marvin Minsky, Nathaniel Rochester and Claude Shannon, was to "proceed on the basis of the conjecture that every aspect of learning or any other feature of intelligence can in principle be so precisely described that a machine can be made to simulate it."[18] Although this included learning, the most successful approach in the beginning was to use the power of computers for symbolic operations to try to extract knowledge from rules. This was the motivation behind expert systems, logic programming and what is called traditional symbolic AI. One example of this approach is MYCIN[16], a rules-based classifier that predicted which bacteria were causing the symptoms and recommended a therapy. MYCIN relied on a set of six hundred if...then rules provided by experts in the field. For example:

```
If:
```

```
(1) the stain of the organism is gram positive, and
(2) the morphology of the organism is coccus, and
(3) the growth conformation of the organism is chains
Then :
there is suggestive evidence (0.7) that the identity of the organism is streptococc
```

This is an example of a classifier that does not learn automatically. All knowledge is fixed from the start. In other words, humans must decide which features to use and humans must program the rules the system will follow.

A different branch of AI was more focused on learning, and artificial neural networks were part of this branch right from the beginning, with the work of McCulloch and Pitts [9] and Rosenblatt's perceptron [13].

1.3 Perceptron

Figure 1.1 shows a neuron cell and a schematic representation of the neuron response. Neurons have a set of dendritic branches which can be stimulated by other cells. If the stimulus passes a threshold, then the neuron fires an impulse over the axon, consisting of a wave of membrane depolarization. This in turn leads to the release of neurotransmitters in the synaptic terminals.

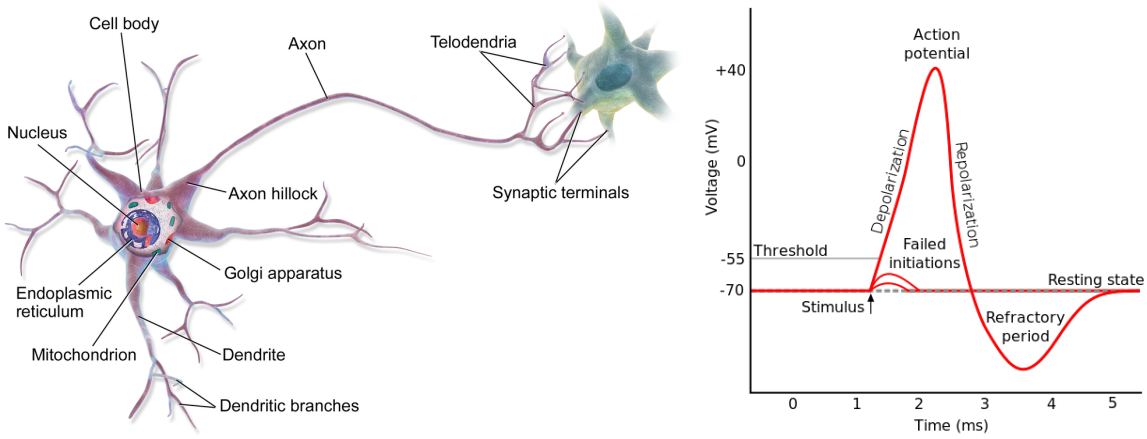

Figure 1.1: Neuron anatomy (BruceBlaus, CC-BY, source Wikipedia) and action potential response.

The neuron provides the inspiration for Rosenblatt's *perceptron*, a neuron model consisting of a linear combination of the inputs, plus a bias value, and a non-linear threshold response function:

$$y = \sum_{j=1}^{d} w_j x_j + w_0 \qquad s(y) = \begin{cases} 1, & y > 0 \\ 0, & y \leq 0 \end{cases}$$

Rosenblatt's algorithm for updating the weights of the perceptron made it possible to train the model by iteratively presenting it with examples and correcting mistakes until the best set of weights was found. The weights are updated according to the following rule:

$$w_i = w_i + \Delta w_i \qquad \Delta w_i = \eta(t - o)x_i$$

where t is the target label of the example, o the output of the *perceptron* for that example, x_i the input value for feature i and w_i the coefficient i of the perceptron. Since the output of the *perceptron* is either

0 or 1, as is the target class of each example, the training rule consists essentially of adjusting the weights of the *perceptron* for every example that is incorrectly classified. Figure 1.2 shows a schematic representation of the neuron model and the Rosenblatt perceptron.

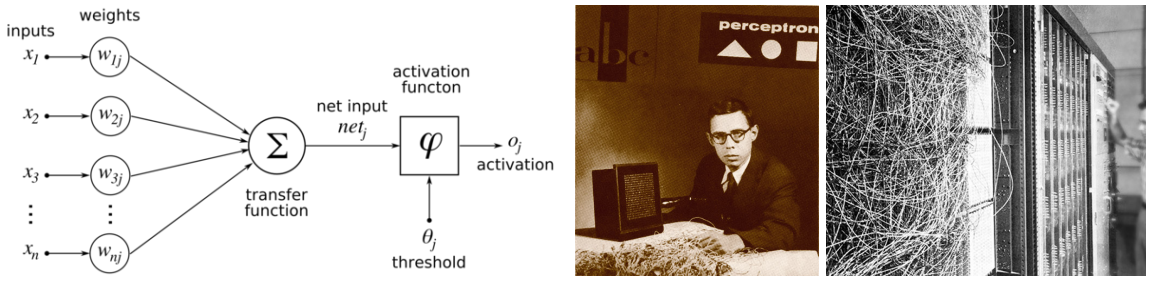

Figure 1.2: Neuron model, consisting of a weighted sum of the inputs, including a bias input of 1, and a non-linear activation function, such as the perceptron's threshold function. The middle and right panels show the original Rosenblatt perceptron (photos from the Arvin Calspan Advanced Technology Center and Hecht-Nielsen, R. Neurocomputing).

There were great hopes for this system capable of learning from examples. In 1958, the New York Times proclaimed it to be "the embryo of an electronic computer that [the Navy] expects will be able to walk, talk, see, write, reproduce itself and be conscious of its existence." Unfortunately, as it turned out, these initial neural models were equivalent to the generalized linear models already known to statisticians, such as logistic regression. Due to the limitations of the perceptron and the difficulties in training multi-layer networks, neural networks, and AI in general, went through a period of disappointment and skepticism between around 1960 and up to the mid 1980s, a period known as the "the AI winter". Although these systems could learn linear transformations of the features, the features needed to be chosen by the humans and only classes that could be linearly separated given those features could be adequately classified.

In 1986, Rumelhart, Hinton and Williams established that backpropagation as a method for training multi-layer neural networks [14]. Given that the activation functions at each layer provided a nonlinear transformation, training a stack of layers allowed the learning of sequences of nonlinear transformations. In the 1990s, AI research and applications shifted increasingly towards machine learning and data-driven applications. However, the practical limitations of fitting large neural networks meant that other machine learning approaches using nonlinear transformations were often more successful. Support vector machines using kernels, for example [1]. It also meant that, despite gaining the ability to solve nonlinear learning problems, it was still necessary for humans identify the right features.

In the late 1990s things started o improve for artificial neural networks. For example, by using convolution and multi-layered networks for processing checks [6] it was possible to automate reading handwritten digits. The modified National Institute of Standards and Technology (MNIST) database [8] pioneered the sharing of large datasets for machine learning research. In 2007, Hinton proposed a method for pre-training multi-layered networks one layer at a time [4], which helped solve the problems of backpropagation over multiple layers. In 2010, Hinton and Nair showed how Rectified Linear Units (ReLU) could improve training of neural networks, eventually overcoming the need for pre-training [4].

Furthermore, the improvement in hardware, with general-purpose graphics processing units (GPGPU)[11] and specialized processors such as the Tensor Processing Unit (TPU) from Google or the Nervana from Intel, keeps increasing the size of neural networks that can be trained. This created an important paradigm shift. Instead of using features selected by humans and learning a nonlinear transformation, deep learning involves fitting deep stacks of nonlinear transformations on the raw data,

dispensing with human intervention in the selection of features and making it easier to use unstructured data.

1.4 Machine Learning

Machine learning is the science of building systems that improve with data. This is a broad concept that includes instances ranging from self-driving cars to sorting images on a database and from recommendation systems for diagnosing diseases to fitting parameters in climate change models. The fundamental idea is that the system can use data to improve its performance at some task. Which immediately points us to the three basic elements of a well-posed machine learning problem:

1. The task that the system must perform.
2. The measure by which its performance can be evaluated.
3. The data that can be used to improve its performance.

Different tasks will determine different approaches. We may want to predict some continuous value, such as the price of apartments, which is a *Regression* problem. Or we may have a *Classification*, when we want to predict in which category, from a discrete set, each example belongs to. If we do this from a set of data containing the right answers, so we can then extrapolate to new examples, we are doing *Supervised Learning*.

There are other other types of problems that can be solved with machine learning, such as clustering, for example, which is an example of *Unsupervised Learning*. While *Supervised Learning* requires that all data be labelled, *Unsupervised Learning* uses unlabelled data. But it is possible to use data sets in which some data is labelled but the rest, usually most of the data, is not. In this case, we have *Semi-supervised Learning*. This approach has the advantage that, usually, unlabelled data is much easier to find than correctly labelled data. For example, it is possible to obtain from the World Wide Web many examples of English texts but to label correctly each grammatical element of each sentence would be very laborious. By combining clustering and classification it is possible to use unlabelled texts to improve the parsing and classification of elements from a set of labelled texts.

Regardless of the approach or the problem, the basic goal of machine learning is to create some representation of regularities in data.

Deep Learning

The term *deep learning* is used to refer to machine learning with models with multiple layers of nonlinear transformations. Though the idea may be more generic, this usually means neural networks with several layer that can learn different representations of the data. Figure 1.3 illustrates this for a generic classifier using deep learning, but the same idea could apply to regression or unsupervised learning such as with autoencoders.

A simple example is the solution of the exclusive or (XOR) function, which results in two classes that are not linearly separable, as Table 2.2 illustrates. As we will see in more detail later on, we can solve this using a neural network with two layers, with the neurons in the first layer transforming the data so that it can be linearly separated by the last layer. In other words, the hidden layer is learning a new representation of the data that allows it to be properly classified by the last neuron.

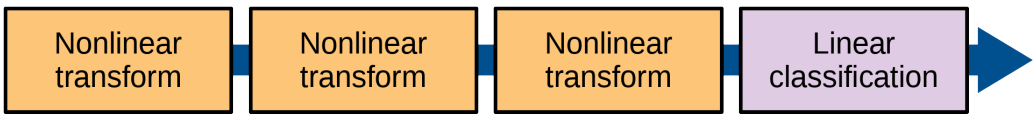

Figure 1.3: Schematic illustration of a deep learning model for classification. The last stage is a linear classifier operating on the result of multiple nonlinear transformations that change the representation of the original data.

Table 1.1: XOR

x_1	x_2	XOR
0	0	0
0	1	1
1	0	1
1	1	0

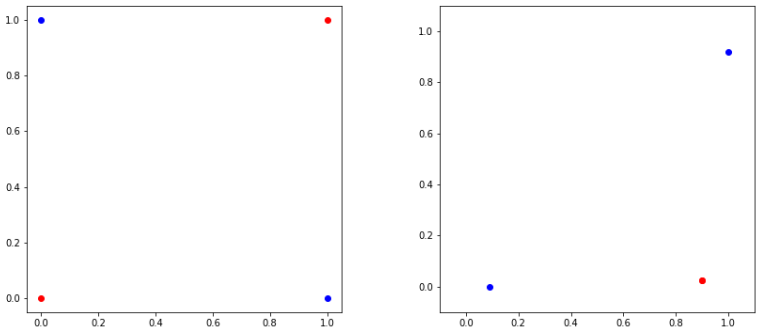

Figure 1.4: Original and transformed points from the OR function.

With more layers, more representations can be learned, each corresponding to a transformation of the representation in the previous layer. Figure 1.5 shows another data set and two transformations in the hidden layers, with 2 neurons each, resulting in a nearly linearly separable set of points that can be classified by the last neuron.

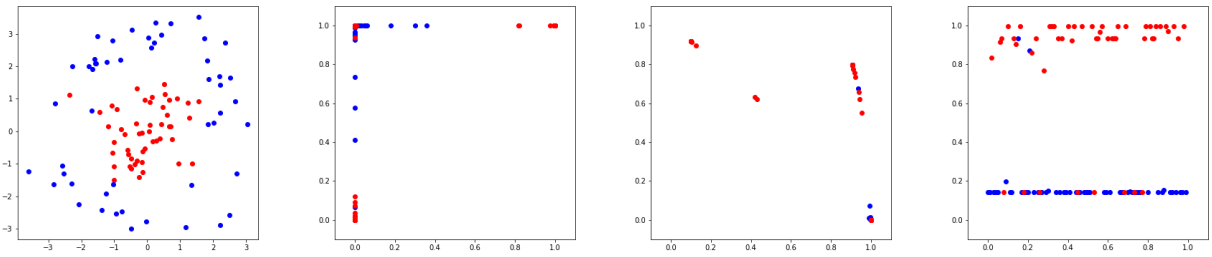

Figure 1.5: From left to right, the original data, the representation in the first hidden layer, in the second hidden layer and theclassification of each data point in the vertical axis, with the points sorted in the horizontal axis.

Deep learning can also be used in unsupervised learning in order to learn useful representations even without using class labels. The example shown in Figure 1.6 uses the banknote authentication data set from the UCI Machine Learning Repository [1]. This is a set of data from 1372 banknotes, divided in two classes (real and fake) and with each banknote described by four numerical features (variance, skewness and curtosis of wavelet transformed image and entropy). Figurerefbanknotes shows these four-dimensional data projected into the first two principal components using principal component analysis (PCA), the graph of an autoencoder with 6, 4, 2, 4, and 6 hidden layers trained to output values as close as possible to the input values and the representation of the data in the middle layer, with

[1] https://archive.ics.uci.edu/ml/datasets/banknote+authentication

2 neurons. We will revisit this later on, in more detail, both the computation graphs represented in Tensorboard and autoencoders, but at this point this serves to show how the representation learned by the deep network captures the structure of the data much better than classical methods such as PCA.

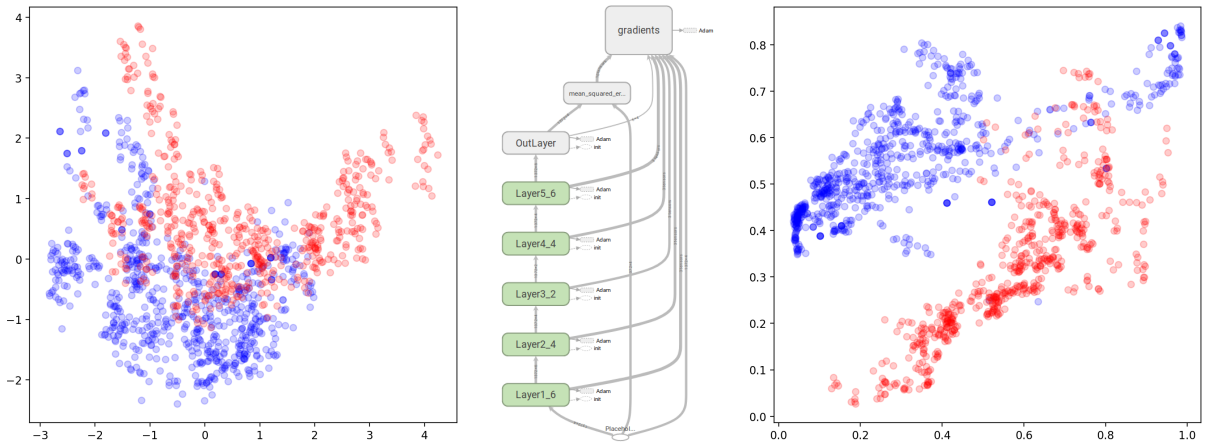

Figure 1.6: From left to right, a PCA projection of the original data using the first 2 components, the graph of the autoencoder and the representation learned in the middle hidden layer. The colors represent the different classes for the banknotes.

1.5 Linear Classifiers

Linear classification has a long history in statistics and machine learning. Examples include linear discriminant analysis, logistic regression and the original perceptron and support vector machine algorithms. These classifiers are adequate only if the classes to be distinguished can be separated by a linear combination of the features. Logistic regression, for example, finds a hyperplane where the probability of a point belonging to each of two classes, estimated with the logistic function, is equal:

$$g(\vec{x}, \widetilde{w}) = P(C_1|\vec{x}) \qquad g(\vec{x}, \widetilde{w}) = \frac{1}{1 + e^{-(\vec{w}^T\vec{x} + w_0)}}$$

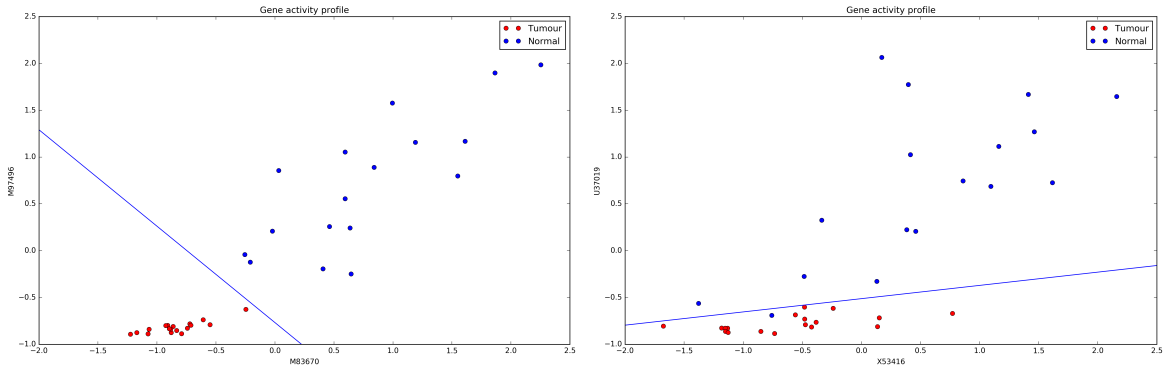

Figure 1.7: Linear classifiers are adequate for linearly separable classes but are unable to separate classes that are not linearly separable.

However, if the classes are not linearly separable, linear classifiers are inadequate. Figure 1.7 illustrates two data sets classified with logistic regression.

1.6 Shallow classifiers

One way of solving this problem is to expand the data with a non-linear transformation. Since the transformation is not linear, the data will be expanded into more dimensions and be spread out over a curved surface. With this transformed data, it may then be possible to separate the classes correcly with a linear classifier. Using the kernel trick, support vector machines can do this implicitly:

$$\arg\max_{\vec{\alpha}} \sum_{n=1}^{N} \alpha_n - \frac{1}{2} \sum_{n=1}^{N} \sum_{m=1}^{N} \alpha_n \alpha_m y_n y_m K(\vec{x}_n, \vec{x}_m)$$

Where $K(\vec{x}_n, \vec{x}_m)$, the kernel function, returns the dot product of some non-linear expansion ϕ of our original data. Figure 1.8 illustrates the difference between a linear classifier and a shallow, non-linear, classifier. This difference is due to a transformation of the selected features prior to classification. Generally, this involves expanding the features to a higher dimensional space (with more features) with a transformation that is not learned from the data by the classifier, but is fixed and chosen previously. In this example, the original features x_1 and x_2 were augmented by adding the computed features $x_1 x_2, x_1^2, x_2^2, x_1^3, x_2^3$.

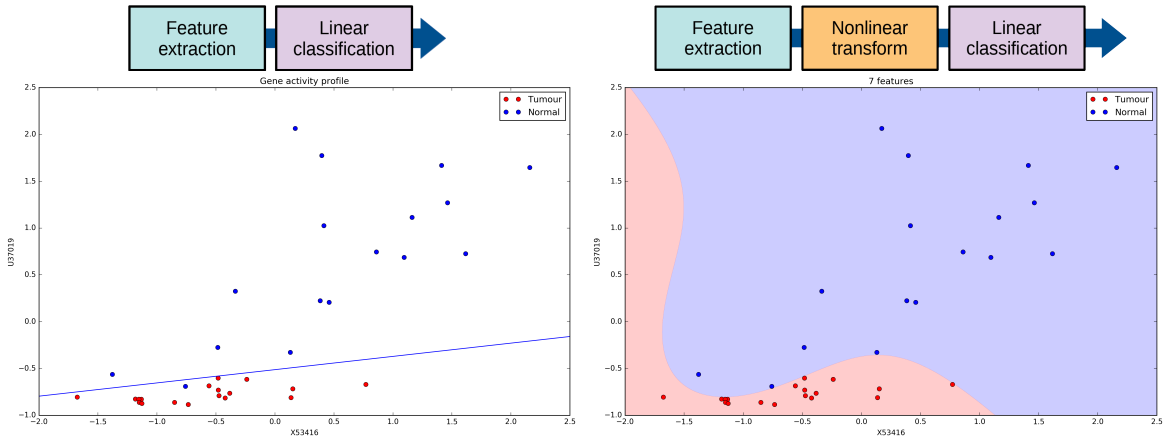

Figure 1.8: Linear classification without (left) or with (right) a non-linear transformation of the original features. In shallow classifiers, this transformation is generally controlled by hyperparameters that are fixed and not learned from the data. For example, the kernel function in a support vector machine.

1.7 Deep classifiers

Deep classifiers, such as deep neural networks, chain different non-linear transformations. This makes them highly non-linear, since each transformation operates on the result of the previous transformation, but also helps solve two problems that shallow classifiers have. Figure 1.9 illustrates this with the GoogLeNet network [19], which we will cover in a bit more detail in the next section.

This cascade of non-linear transformations in deep neural networks has advantages over the non-linear expansion typical of shallow classifiers. Since it is not a function of hyperparamaters that are fixed during training, but rather learned by the classifier, these transformations are adapted to the training data and result in more effective representations of different aspects of the data. The flexibility of these transformations and their stacking in different layers can also reduce the number of dimensions in which the data must be represented in order to be properly classified. For complex problems such as image or voice recognition, this can be a reason why shallow learning systems fail and deep learners

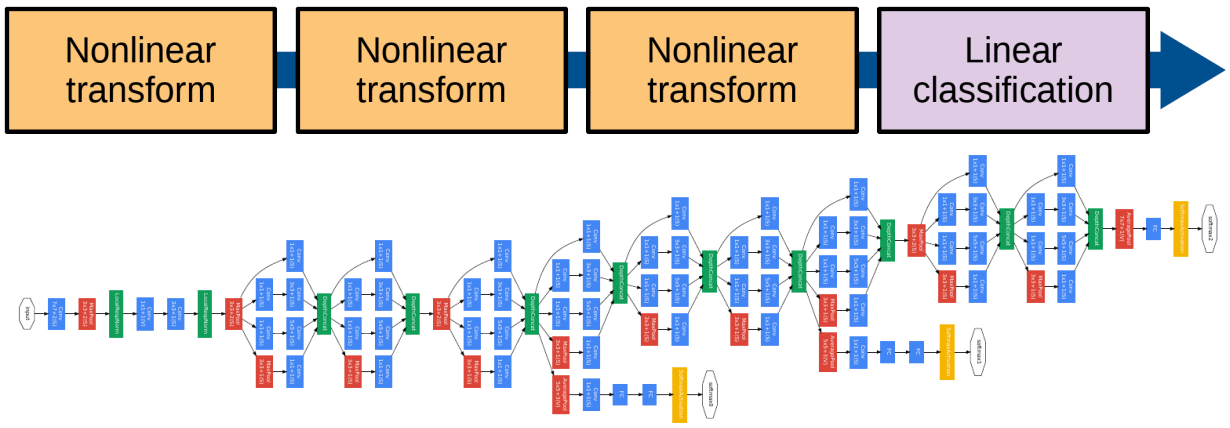

Figure 1.9: Example of a deep classifier, the GoogLeNet network, [19].

succeed. Figure 1.10 illustrates different representations at different layers of a deep neural network (a convolution network, in this case).

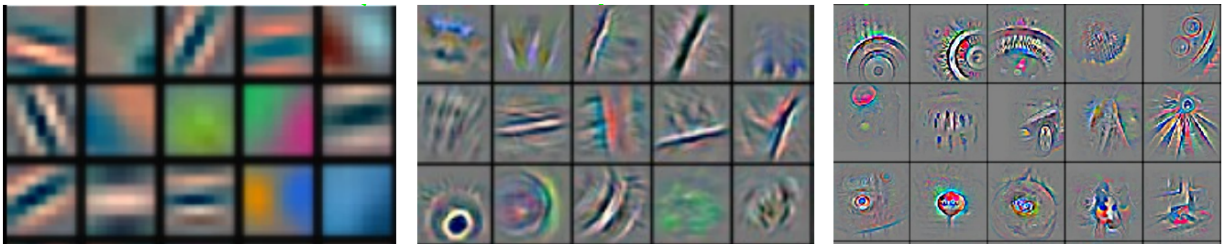

Figure 1.10: Example of representations in a deep convolution network, showing different aspects of the input images being detected at different layers, such as straight edges in the first layer, more complex edges in the second, complex shapes in the third and so on. Image from Zeitler 2014 [21].

No free lunch and overfitting

This is not to say that deep learning is, overall, better than shallow learning. In fact, there is no such thing as a better learning algorithm for the general cases. The "no-free-lunch" theorems prove that any learning algorithm that performs better than average for some type of problems will always pay for that performance by performing worse than average in other types of problems[20].

This also applies to the hypotheses obtained by instantiating any model. When we finish training a learning model, the set of parameters we obtained is especially suited to perform well over the training data but will perform poorly over data that is not similar to the training data. This is the problem of *overfitting*, which results from the model learning aspects of the training set that do not generalise to new data outside this set, thus resulting in a high true error despite a low training error. Figure 1.11 illustrates this, showing how the error measured with the training set or with data outside the training set (the test set) can diverge as we increase the power of the model to adjust to the training data.

Despite the advantages of deep learning models, these models always have a large number of parameters to be adjusted during training, and are extremely capable of adapting to the structure of the training data. This adaptability makes them better at solving complex problems but also more prone to overfitting if the training data is not representative of the universe of data where the learner is to be applied. In practice, this means that shallow learners can perform better than deep learners for simpler problems and when there are few examples available for training. It is only when the training

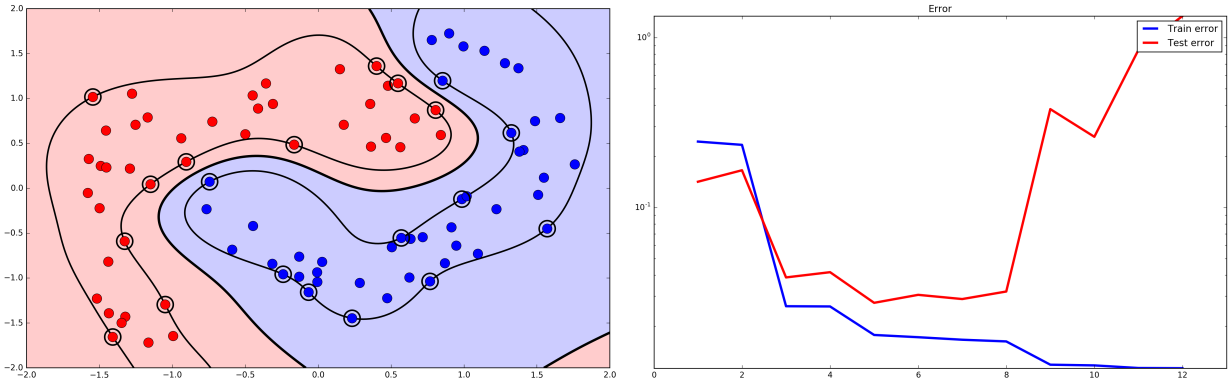

Figure 1.11: Example of overfitting. By adjusting too much to the training data, the resulting hypothesis may generalize poorly to data outside the training set (the test data in the right panel), leading to a greater true error, estimated by the test error, the greater the ability to adjust to the training data.

set is large enough to minimize overfitting that deep learning performs better, and in those cases the effectiveness of its representations enable this approach to solve problems that shallow learners cannot solve as effectively.

The availability of data, new techniques for training deep models and the development of efficient hardware for the computations is what made deep learning so successful in the last decade. For example, using the Caltech 101 images classification dataset, created in 2006 with 101 categories and an average of 50 images per category, classical computer vision algorithms and shallow classifiers performed best, with a 26% error rate. When the ImageNet database was created in 2012, with 1.2 million images in 1000 categories, deep convolution networks immediately dominated the competitions for image classification, beginning with the AlexNet, with a 15% error rate for top-5 predictions[5]. Over the years, results have been steadily improving but always with deep classifiers. For these problems and with such a large data set, shallow classifiers can no longer compete:

- AlexNet, Krizhevski et al. 2012, 15% top-5 error[5]
- OverFeat, Sermanet et al. 2013, 13.8% error[15]
- VGG Net, Simonyan, Zisserman 2014, 7.3% error[17]
- GoogLeNet, Szegedy et al. 2014 6.6% error[19]
- ResNet, He et al. 2015 5.7% error[12]

1.8 Examples

AlexNet was an early example of the power of deep networks for solving image classification problems. Used in 2012 for the ImageNet database, it was trained on two NVIDIA GeForce GTX 580 graphics cards and consisted in six convolution layers followed by three fully connected layers [5].

After this demonstration of de performance of deep models, the tendency was to increase the depth and complexity of the classifiers. In 2014, GoogLeNet reduced the top-5 classification error on ImageNet to 6.7% with 22 layers and 100 different processing blocks [19].

But deep models are not just for image classification. For example, in 2016 Hendricks et. al. [3] used a deep convolution networks to provide features for two long short term memory (LSTM) networks

so that, in addition to classifying images of birds, the system also learned to generate sentences to "explain" why the image was classified in that manner.

1.9 The promise of Deep Learning

There are several obstacles to solving a machine learning problem, even after we formulate the problem adequately by specifying what task we want the machine to solve, how to measure performance in that task and what data to use for training. In classical machine learning, the first step is to extract from the data the appropriate features that our model will use. This can be a demanding task, requiring human expertise in the domain of the problem. With deep learning this task can be greatly simplified because deep models have the ability to find good sequences of transformations that will extract the necessary features from the data.

We also need to choose the right model, and in classical machine learning this requires experimenting with fundamentally different approaches. A bayesian network, a support vector machine and a random forest classifier are very different algorithms, with different behaviors, requirements and computational methods. Deep neural networks provide a unified framework that can be used to create many different models that, despite being geared to different types of problems, all rest on the same basic principles.

1. Skansi, Introduction to Deep Learning, Chapter 1 [18]
2. Goodfellow *et. al.* Deep Learning [2], Chapters 1 and 5

Chapter 2

Deep learning, intro

Backpropagation. Stochastic Gradient Descent. Deep model architectures. Data and features. Why the success now? Some examples of deep learning with unstructured data

2.1 Backpropagation

As we saw, Rosenblatt's perceptron was only a linear classifier. However, if we can stack layers with non-linear responses we can go beyond linear classification. The problem with the original formulation of the perceptron is that the response function is discontinuous. This may be nearer to the biological features of the neuron but raises problems with the minimization of the error function if we stack several layers.

To solve this problem we can use a differentiable threshold function. One often used function is the *logistic* function, also called the *sigmoid* function:

$$s(y) = \frac{1}{1 + e^{-y}} = \frac{1}{1 + e^{-\vec{w}^T \vec{x}}}$$

But before we see how to solve the optimization problem for a neural network, we will start with a single neuron.

2.2 A Single Neuron

One possible way to train a logistic response neuron is by minimizing the squared error between the response of the neuron and the target class. So we minimize the error function:

$$E = \frac{1}{2} \sum_{j=1}^{N} (t^j - s^j)^2$$

We can do this in a way similar to the one used for the *perceptron*, by adjusting the weights of the neuron in small steps as a function of the error at each example j, $E^t = \frac{1}{2}(t^j - s^j)^2$, where t^j is the class of example j and s^j is the neuron's response for example j. To do this, we need to compute the derivative of the error as a function of the weights of the neuron in order to compute how to update the neuron weights. Since the error is a function of the activation of the neuron for example j (s^j), the

activation is a function of the weighted sum of the inputs (net^j) and this is, in turn, a function of the weights, we use the *chain rule* for the derivative of compositions of functions to obtain the gradient as a function of each weight:

$$-\frac{\delta E^j}{\delta w} = -\frac{\delta E^j}{\delta s^j}\frac{\delta s^j}{\delta net^j}\frac{\delta net^j}{\delta w}$$

where

$$s^t = \frac{1}{1+e^{-net^j}} \qquad net^j = w_0 + \sum_{i=1}^{M} w_i x_i$$

Since

$$\frac{\delta net^j}{\delta w} = x$$
$$\frac{\delta s^j}{\delta net^j} = s^j(1-s^j)$$
$$\frac{\delta E^j}{\delta s^j} = -(t^j - s^j)$$

We obtain the following update rule for the weight i of the neuron given example j:

$$\Delta w_i^j = -\eta\frac{\delta E^j}{\delta w_i} = \eta(t^j - s^j)s^j(1-s^j)x_i^j$$

Using this update function we descend the error surface in small steps in different directions according to each example presented to the net. With examples presented in random order, this is a *stochastic gradient descent*. Figure 2.1 illustrates this process of stochastically descending the error surface. The process of updating the weights at each example is called *online learning*. An alternative training schedule consists of summing the Δw_i^j updates for a set of examples and then updating the weights with the total change. This is called *batch learning*. These are examples of *stochastic gradient descent* because they are ways of descending along the gradient of the error function along random paths depending on the data. One pass through the whole training set is called an *epoch*.

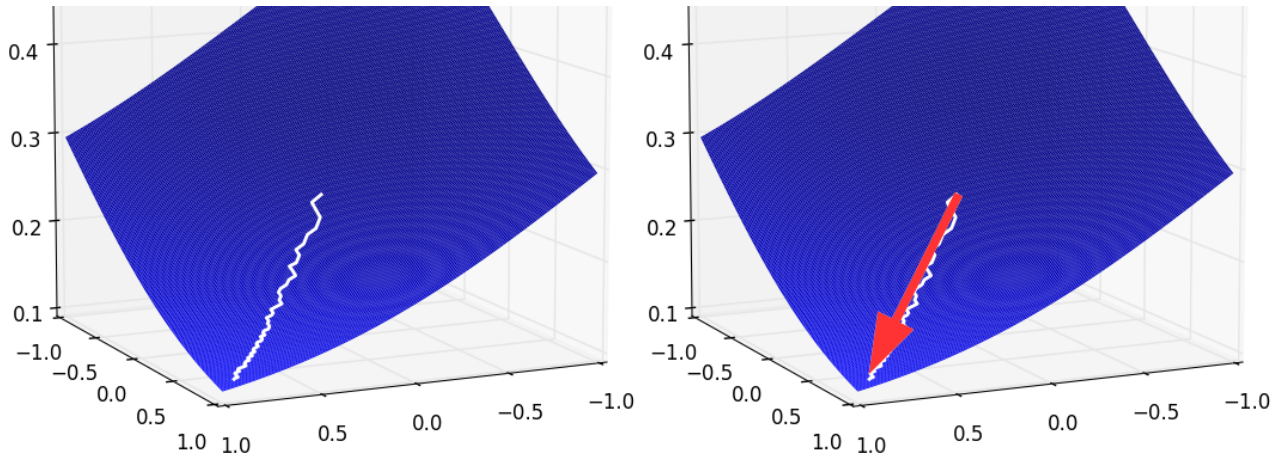

Figure 2.1: Stochastic gradient descent with online training (left panel) and batch training (right panel).

With a single neuron it is possible to learn to classify any linearly separable set of classes. One classical example is the `OR` function, as shown in Table **??**.

Table 2.1: The OR function

x_1	x_2	OR
0	0	0
0	1	1
1	0	1
1	1	1

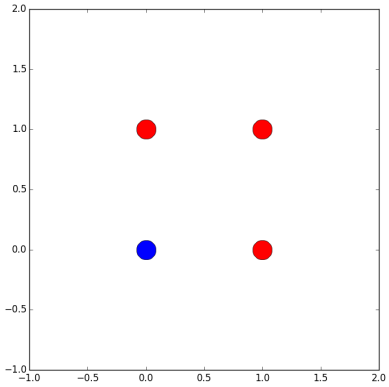

Figure 2.2: Set of points from the OR function.

Figureřef8-neuro-or shows the training error for one neuron being presented the four examples of the OR function and the final classifier, separating the two classes. The frontier corresponds to the line where the response of the neuron is 0.5.

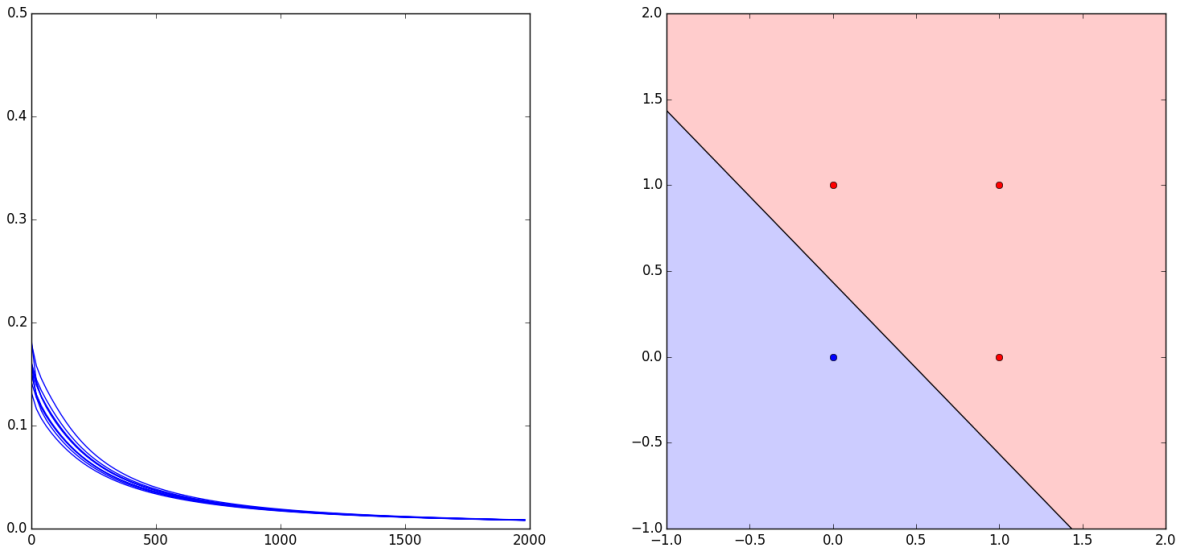

Figure 2.3: Training error and final classifier for one neuron trained to separate the classes in the OR function.

However, if the sets are not linearly separable, a single neuron cannot be trained to classify them correctly. This is because the neuron defines a hyperplane separating the two classes. For example, the exclusive or (XOR) function results in two classes that are not linearly separable, as Table 2.2 illustrates. So, if we try to train a neuron to separate these classes there is no reduction in the training error nor does the final classifier manage to separate the classes, as shown in Figure 2.5.

Table 2.2: The XOR function

x_1	x_2	XOR
0	0	0
0	1	1
1	0	1
1	1	0

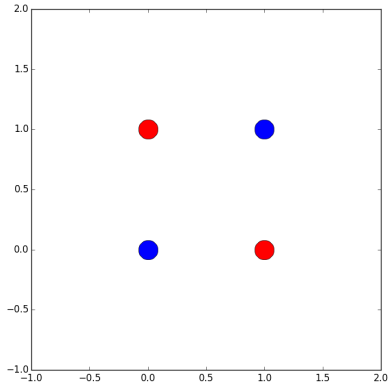

Figure 2.4: Set of points from the OR function.

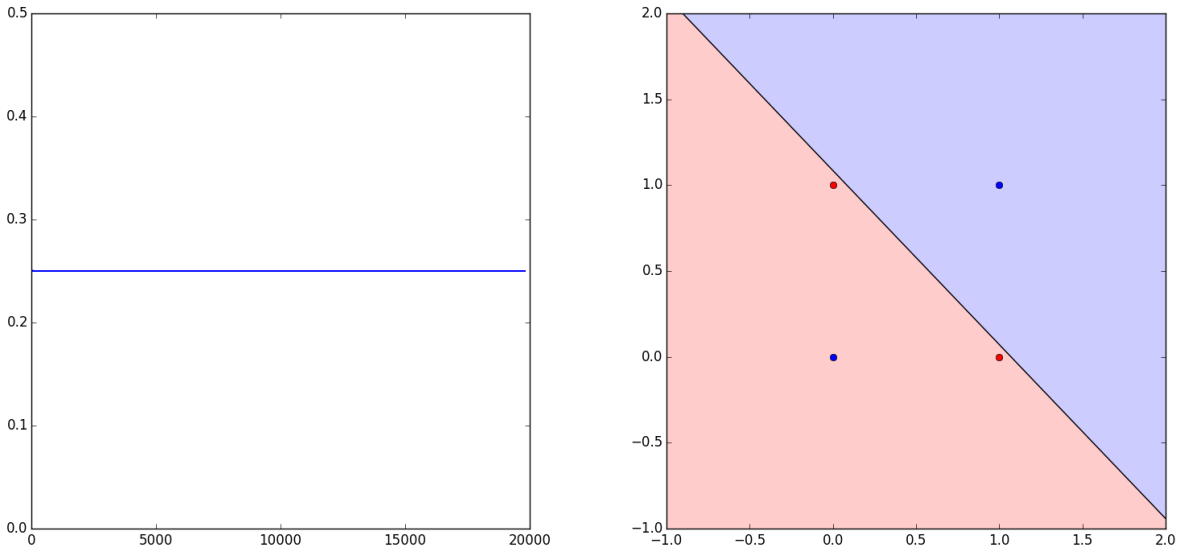

Figure 2.5: Training error and final classifier for one neuron trained to separate the classes in the OR function.

The solution for this problem is to add more neurons in sequence.

2.3 Multilayer Perceptron and Backpropagation

The *multilayer perceptron* is a fully connected, feedforward neural network. This means that each neuron of one layer receives as input the output of all neurons of the layer immediately before. Figure 2.6 shows two examples of multilayer perceptrons (MLP).

To update the coefficients of the output neurons, we derive the same update rule as for the single neuron with the only difference that the input value is not the value of an example feature but rather the value of the output of the neuron from the previous layer. Thus, the update rule for weight m of neuron n in layer k is:

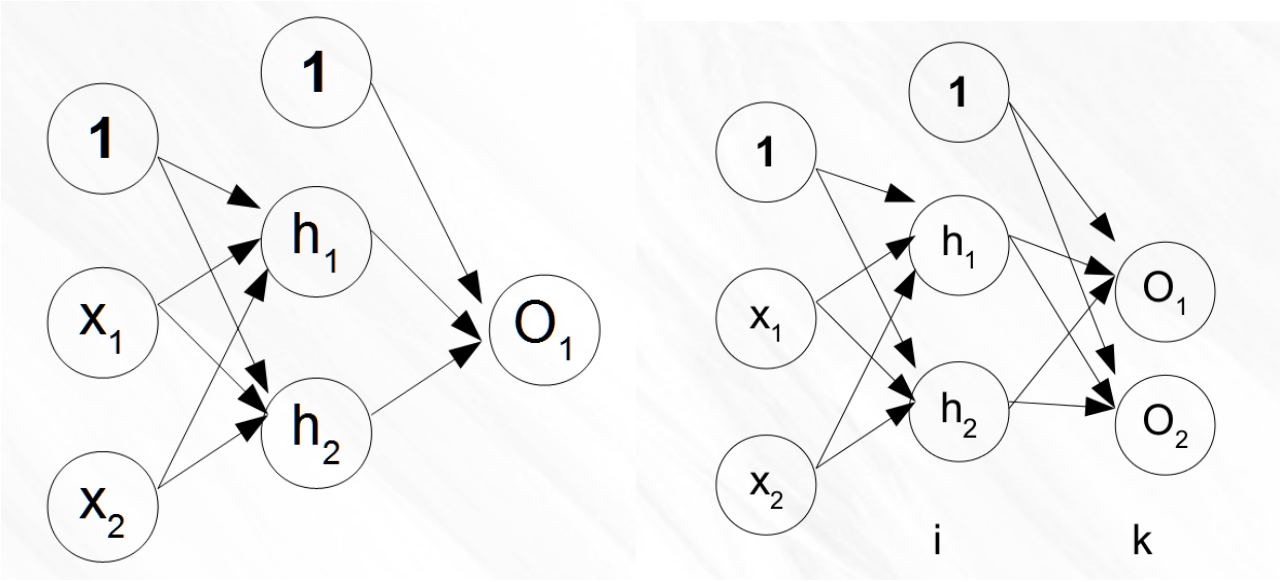

Figure 2.6: Two examples of multilayer perceptrons. Both have a hidden layer. The left panel shows a MLP with one output neuron, the right panel an MLP with two output neurons.

$$\begin{aligned}\Delta w^j_{m,k,n} &= -\eta \frac{\delta E^j_{k,n}}{\delta s^j_{k,n}} \frac{\delta s^j_{k,n}}{\delta net^j_{k,n}} \frac{\delta net^j_{k,n}}{\delta w_{m,k,n}} \\ &= \eta(t^j - s^j_{k,n}) s^j_{k,n}(1 - s^j_{k,n}) s^j_{i,n} = \eta \delta_{k,n} s^j_{k-1,n}\end{aligned}$$

Where $s^j_{k-1,n}$ is the output from neuron n of layer $k-1$.

For neurons in hidden layers, we need to backpropagate the error through the layers in front:

$$\begin{aligned}\Delta w^j_{m,i,n} &= -\eta \left(\sum_p \frac{\delta E^j_{k,p}}{\delta s^j_{k,p}} \frac{\delta s^j_{k,p}}{\delta net^j_{k,p}} \frac{\delta net^j_{k,p}}{\delta s^j_{i,n}} \right) \frac{\delta s^j_{i,n}}{\delta net^j_{i,n}} \frac{\delta net^j_{i,n}}{\delta w_{m,i,n}} \\ &= \eta(\sum_p \delta_{kp} w_{m,k,p}) s^j_{in}(1 - s^j_{i,n}) x^j_i = \eta \delta_{i,n} x^j_i\end{aligned}$$

The intuition for this is that the neuron in the hidden layer will contribute its output to several neurons in the layer ahead. Thus, we need to sum the errors from the neurons of the front layer, propagated through the respective coefficients of those front neurons.

This is the *backpropagation algorithm*:

- Present the example to the MLP and activate all neurons, propagating the activation forward through the network.
- Compute the $\delta_{n,k}$ for each neuron n of layer k, starting from the output layer and then backpropagating the error through to the first layer.
- With the $\delta_{n,k}$ values .

With this algorithm and the MLP architecture shown on the left panel of Figure 2.6, we can train the network to classify the XOR function output. During training the two neurons on the hidden layer learn to transform the training set so that their outputs result in a linearly separable set that the neuron on the output layer can then separate.

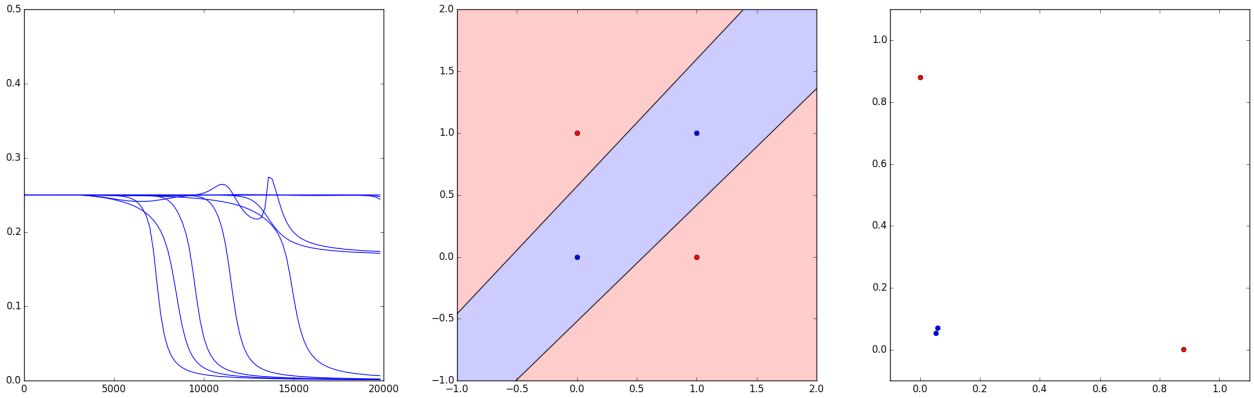

Figure 2.7: Training the MLP with one hidden layer for classifying the XOR function output. The first panel shows the training error over 10 training runs. Note that, due to the stochastic initialization and ordering of the examples presented, there are differences between different runs. The second panel shows the resulting classifier, successfully separating the classes. The third panel shows the output of the two neurons in the hidden layer of the network. This layer transforms the features of the training set making it linearly separable.

2.4 Data and features

Classical machine learning models work best with structured data, where all examples have the same set of features and each feature has a specific meaning. Each example differs in the values of its features, but the features are all the same. Thus, structured data can be represented as a table or matrix of features and examples. This is the best case for most machine learning algorithms because each feature of each example will provide a value in the input vector and these vectors will all be the same, both in size and in what each value represents. For example, we can characterize different patients by measuring temperature, blood pressure, glucose levels, height, weight and so forth, and each entry in the vector describing each patient would always correspond to one feature. This can be done either with numerical features or categorical features. The requirement for the data to be structured is simply that the features are fixed and the same for all examples.

Unstructured data is data that does not meet these requirements, and it is likely that the majority of useful data is stored without structure. Examples include text documents, phone conversations, video and photographs and social media posts and comments. In these cases, there is no predetermined relation between each value and features with some semantic significance. We can look at the brightness of the pixel at some position in all images but this pixel may be part of an eye, or hand, or wheel depending on the image.

Classical methods of dealing with unstructured data require a first step of feature extraction to create a structured representation of the data. Since this explicit step is guided by humans, using their knowledge of the domain, this is a labor-intensive. But, now, deep learning is changing the approach to unstructured data. Deep neural networks can be used to learn, automatically how to extract useful features from unstructured data, and so deep learning is making available a large volume of data that would otherwise be left unusable.

2.5 The Triumph of Deep Learning

(work in progress...)

2.6 Further Reading

1. LeCun, Bengio and Hinton, Deep learning, Nature 2015[7]
2. Goodfellow et. al., Chapter 5 and beginning of Chapter 6 (sections 6.1 and 6.2)[2]
3. Patterson and Gibson, Deep learning: A practitioner's approach, introduction to chapter 3 (pp 81-91)[10]

Chapter 3

Training Neural Networks

Algebra (revisions), The computational graph and AutoDiff, Training with Stochastic Gradient Descent.

3.1 Algebra

To understand how to implement artificial neural networks, and how to use `tensorflow`, we will start by reviewing some concepts of algebra. We will use the following terms:

- Scalar: a single number
- Vector: a one-dimensional array of numbers
- Matrix: a two-dimensional array of numbers
- Tensor: formally any relation between sets of algebraic objects, but we will use this term to refer to n-dimensional arrays of numbers

In particular, we will be using tensors. Hence the name `tensorflow`, which is fundamentally a library to apply operations to tensors. And that is why an important attribute of tensors is their shape, since we must take care how tensors fit with the operations.

One example of this is the algorithm for multiplying two matrices, or two 2D tensors. For the product $C = AB$, the matrix C is obtained by the sum of the element-wise products of each row of A with each column of B, as illustrated in Figure 3.1.

This is a very useful operation in artificial neural networks because, if matrix A contains a batch of examples with one example per row and one feature per column, and matrix B has the weights of the neurons of one layer in our network, with each neuron as a column, then the product C will have, for each row, the sum of the products of the features of each example by the weights of the neurons, with each neuron in each column and each example in each row. This means that we can simply add the bias vectors to matrix C and then apply the activation function for these neurons and we obtain a matrix that we can feed into the next layer, repeating these operations.

`tensorflow` even helps us do this for batches of matrices using higher dimensional tensors. If we use the `matmul` function we can perform matrix multiplication on the 2D matrices defined by the two

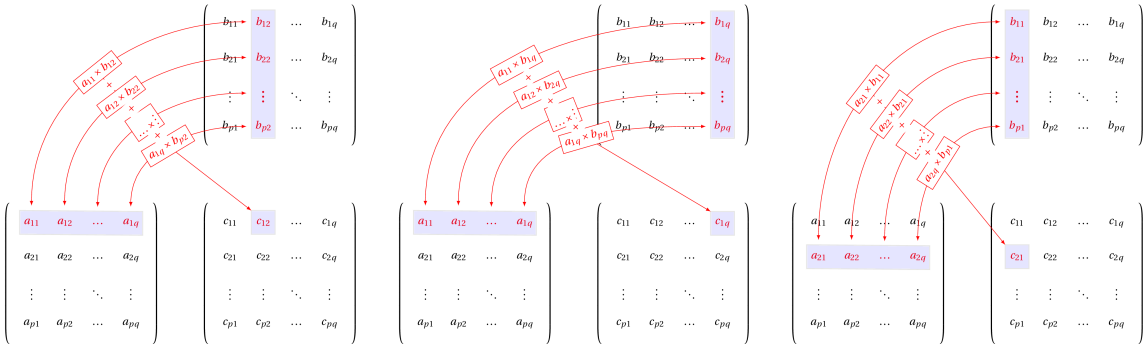

Figure 3.1: Matrix multiplication

last dimensions of the tensors, broadcast through the other dimensions using the normal broadcasting rules like in `numpy`. The code below illustrates this, by first creating two constant tensors of shapes (2,2,3) and (2,3,2) and then applying matrix multiplication between the two 2D matrices in each tensor. Note that the shapes of theses matrices, (2,3) and (3,2), match the requirements for matrix multiplication using algebra rules. The remaining dimensions of these tensors must allow broadcasting [1]

```
In : a = tf.constant(np.arange(1, 13, dtype=np.int32), shape=[2, 2, 3])
In : b = tf.constant(np.arange(13, 25, dtype=np.int32), shape=[2, 3, 2])
In : c = tf.matmul(a, b) # or a * b
Out: <tf.Tensor: id=676487, shape=(2, 2, 2), dtype=int32, numpy=
array([[[ 94, 100],
        [229, 244]],
        [[508, 532],
        [697, 730]]], dtype=int32)>
```

Further along this course we will be using the Keras API, so we will not generally have to worry with these low level operations. However, the shape of the tensors is always something to which we must pay attention.

3.2 Automatic Differentiation and the computational graph

As we saw previously, neural networks are trained through backpropagation, which requires computing the derivative of the loss function with respect to each adjustable parameter in the network. In previous examples, we solved this problem by manually computing the analytical expression of the derivatives. For example, for updating the weights of a single neuron with a quadratic loss function and sigmoid activation, we obtained this expression:

$$\Delta w_i^j = -\eta \frac{\delta E^j}{\delta w_i} = \eta (t^j - s^j) s^j (1 - s^j) x_i^j$$

However, it is easy to see how this would not be practical for large networks with different architectures. Symbolic derivation of all expressions would require a lot of work before we could train any network, since we would need the expressions for all adjustable parameters.

A generic approach to differentiation, when we cannot derive the analytical expressions, is to use numerical differentiation. This approximates derivatives by computing function values over small steps.

[1] For more details on broadcasting, see the documentation here: https://www.tensorflow.org/xla/broadcasting

The problem is that this would be too inefficient for training neural networks, since we would have to do numerical differentiation at each step during training, and has other problems such as accuracy and convergence conditions.

To solve these problems, `tensorflow` uses automatic differentiation, more specifically reverse-mode automatic differentiation, of which backpropagation is a particular case. Basically, `tensorflow` creates a computational graph where all the nodes are operations and the arcs are the tensors with the data being operated upon. This graph includes not only the operations we specify but also the corresponding derivatives, which are analytically defined for all operations `tensorflow` supports. During the forward pass through the computational graph, `tensorflow` records all operations and stores the intermediate outputs. Then in the backwards pass it computes the gradients by chaining the corresponding derivatives. This is done using a `GradientTape` object as we will see later. The details of reverse-mode automatic differentiation are complex and outside the scope of this course, but you can see the derivatives in the computation graph generated by `tensorflow`. For example, suppose we wanted to find the value of x that minimizes the cosine function. Associated with the cosine function, we also include the derivative in the computation graph as part of the gradients, as Figure 3.2 shows.

$$\underset{x}{\operatorname{argmin}}\,(\cos x) \qquad \frac{d\cos x}{dx} = -\sin x$$

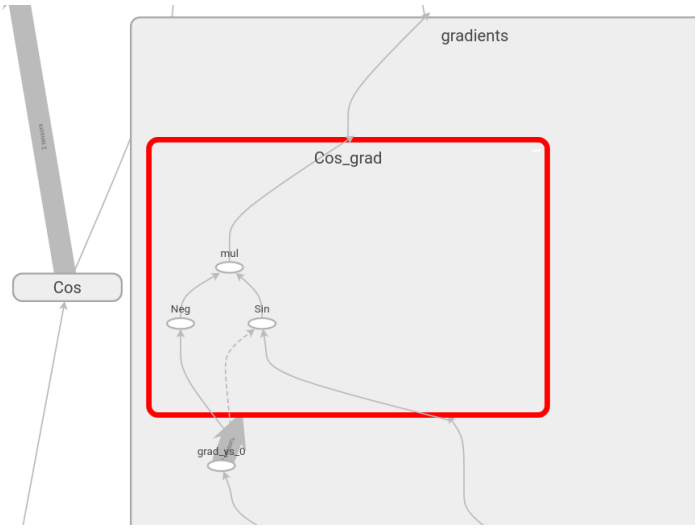

Figure 3.2: Matrix multiplication

3.3 Training with Stochastic Gradient Descent

To train the MLP it is important to start with small, random weights, close to 0. Initializing the network weights properly can be important but, unfortunately, there is no good understanding of precisely what the best way to do this may be. But it is clear that weights cannot be the same for all neurons, otherwise the gradient will be the same for all parameters and all neurons will be optimized in the same way. This symmetry must be broken from the start. It is not a problem to start with bias values at zero, but the weights must be randomized to guarantee that different neurons start at different combinations of parameters.

There are other considerations, depending on the networks. Recurrent networks are more susceptible to instability if the weights start too large and, in any case, large initial weights may saturate activations

or cause other numerical problems. On the other hand, larger weights are better at breaking neuron symmetry and "spread out" the network more widely from the start.

One standard way of choosing initial weights is to simply draw them at random from a Gaussian distribution with mean zero and variance 1. Other initialization schemes include taking into account the number of neurons in each layer, or the number of inputs, and other factors. See section 8.4 of [2] for more details, but bear in mind that weight initialization heuristics may not always give the best results when compared to a simple normal distribution.

This is because the sigmoidal activation functions saturate away from zero. It is also important to run the training process several times, since the training is not always exactly the same. Normalizing or standardizing the inputs is also important, since input features at different scales will force the network to adjust weights at different rates.

3.4 Further Reading

1. Goodfellow et. al., Chapter 2, 4 and 8 citeGoodfellow-et-al-2016

Chapter 4

Introduction to Tensorflow

Tensorflow basics: tensors ans computation graphs. Getting started with tensorflow: simple examples

4.1 Tensorflow

Tensorflow is a library for computing with tensors. A tensor is a multidimensional matrix, and Tensorflow allows us to specify a graph of operations that input and output tensors. This computation graph can then be compiled into CUDA for running on GPU and decomposed into several processors for more efficient execution. So to work with Tensorflow we must first define the computation graph and then execute it. This is done within a Tensorflow session. The thing to remember is that, unlike it is usual in imperative programming, the operations are not executed with the interpreter encounters them. Rather, they merely create the computation graph to be executed when the computations are run.

For example, to compute $a + b$ assigning 3 to a and 4 to b we can use the following code:

```
1 import tensorflow as tf
2 a = tf.constant(3.0, dtype=tf.float32, name='a')
3 b = tf.constant(4.0, name='b')
4 total = tf.add(a,b, name='total')
```

Fist we import the `tensorflow` library. In Tensorflow 2 the default execution mode is eager execution, meaning that each operation is executed by Tensorflow as it is interpreted by the Python interpreter. We will see another execution mode, graph execution, in which the whole computation graph is created first and then executed (this was the default in previous versions of Tensorflow).

We create the two constants `a` and `b`. The type of value can be specified in `dtype`; otherwise, it will be set according to the value of the constant. Then we use the addition operation that will output a tensor with the sum

4.2 Logistic Regression with Tensorflow

In a logistic regression, we assume that there is a function $g(\vec{x}, \widetilde{w})$ that, given an example $\vec{x}$ and some parameters $\widetilde{w}$ approximates the probability of the example belonging to some class. We also assume that the function has this shape:

$$g(\vec{x}, \widetilde{w}) = \frac{1}{1 + e^{-(\vec{w}^T \vec{x} + w_0)}}$$

The function

$$f(x) = \frac{1}{1 + e^{-k(x - x_0)}}$$

is called the *logistic function* and has the useful feature of varying from 0 to 1 around a threshold value but being nearly constant away from this threshold. This is what we need to find a hyperplane separating the two classes. The *logistic regression* is a regression model; when fitting the model we are trying to approximate this continuous probability function. However, because we also choose a cut-off value where we separate the two classes — where $P(C_1|\vec{x}) = P(C_0|\vec{x})$ — we turn this regression model into a classifier.

Now, given this function

$$g(\vec{x}, \widetilde{w}) = P(C_1|\vec{x})$$

the maximum likelihood solution for the problem of finding the parameters $\widetilde{w}$ is the minimum of this cost function:

$$E(\widetilde{w}) = -\sum_{n=1}^{N} [t_n \ln g_n + (1 - t_n) \ln(1 - g_n)]$$

with

$$g_n = \frac{1}{1 + e^{-(\vec{w}^T \vec{x_n} + w_0)}}$$

We will now see how to implement a logistic regression classifier with Tensorflow. We will be using the gene activity data shown previously. We will start by importing the data and standardizing it. The data is in a text file with one example per row and the two features values followed by the class label, separated by tabs. This is the beginning of our script:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Logistic regression demo
"""
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt

mat = np.loadtxt('gene_data.txt',delimiter='\t')

Ys = mat[:,-1]
Xs = mat[:,:-1]
means = np.mean(Xs,0)
stdevs = np.std(Xs,0)
Xs = (Xs-means)/stdevs
```

Standardization consists in subtracting the average and dividing by the standard deviation of all values in order to convert a distribution of values into a distribution with a mean of zero and a standard deviation of one. This is useful to prevent our parameters from varying over a too broad range of values. Now we create our model, which consists of the parameters and the prediction function, which simply applies the network operations.

```
18 weights = tf.Variable(tf.random.normal((2,1)), name="weights")
19 bias = tf.Variable(0.0, name="bias")
20
21 def prediction(X):
22     t_X = tf.constant(X.astype(np.float32))
23     net = tf.add(tf.matmul(t_X, weights), bias, name="net")
24     return tf.reshape(tf.nn.sigmoid(net, name="output"),[-1])
```

For the parameters of our model, we create two variables. Tensorflow variables are persistent objects, like constants, but they can be changed during the computation. Specifically, variables are what the optimizers will adjust in order to minimize a cost function. So our variables are the two weights that will be multiplied by the features and the bias value that will be added before the sigmoid activation. This is implemented in the `prediction` function, using the `sigmoid` activation from Tensorflow. The `prediction` function multiplies the inputs by the weights of the neuron, then adds the bias value and finally applies the sigmoid activation function. One important note: after the matrix operations, we have a two-dimensional matrix. This makes the output of the function be a two-dimensional matrix with a single column with the predictions for the batch of examples. However, since this is just a single column with the predictions, we want to reshape it into a one-dimensional vector so we can compare it with the vector of labels `Ys`. It is always important to match tensor shapes.

Now that we have our model we create the loss function, which will be minimized, and the function for computing the gradients of the loss function:

```
25 def logistic_loss(predicted,y):
26     t_y = tf.constant(y.astype(np.float32))
27     cost = -tf.reduce_mean(t_y * tf.math.log(predicted) \
28            + (1-t_y) * (tf.math.log(1-predicted)))
29     return cost
30
31 def grad(X, y):
32     with tf.GradientTape() as tape:
33         predicted = prediction(X)
34         loss_val = logistic_loss(predicted,y)
35     return tape.gradient(loss_val, [weights, bias]),[weights,bias]
```

The `logistic_loss` functions computes the logistic loss, which will be minimized by using a `GradientDescentOptimizer` object. To do this, the `grad` function uses a `tf.GradientTape` object to trace all computations and compute the derivatives. Within the context of this object, `tensorflow` keeps track of all operations and backtraces them to compute the gradients of the given tensor with respect to the specified variables. In this case, we want the gradients of the loss with respect to the weights and biases.

With everything setup, we create our training loop:

```
1 optimizer = tf.optimizers.SGD(learning_rate=0.1)
2
3 batch_size = 1
4 batches_per_epoch = Xs.shape[0]//batch_size
5 epochs=20
6
7 def run():
8     for epoch in range(epochs):
```

```
9         shuffled = np.arange(len(Ys))
10        np.random.shuffle(shuffled)
11        for batch_num in range(batches_per_epoch):
12            start = batch_num*batch_size
13            batch_xs = Xs[shuffled[start:start+batch_size],:]
14            batch_ys = Ys[shuffled[start:start+batch_size]]
15            gradients,variables = grad(batch_xs, batch_ys)
16            optimizer.apply_gradients(zip(gradients, variables))
17        y_pred =prediction(Xs)
18        loss = logistic_loss(y_pred,Ys)
19        print(f"Epoch {epoch}, loss {loss}")
```

In this loop, we take a batch of examples (by default it is a batch of 1 example only, but we can change that) after shuffling all examples, so that they are presented in a random order, and we pass that batch to the `grad` function, which computes the activations and the derivatives. Then we update the weights using the optimizer. The optimizer receives a list of tuples with each gradient and corresponding variable and updates the variables according to the update algorithm of the optimizer. In this case, if we use a simple SGD optimizer without momentum, this will just involve adding to the weights a fraction of the derivatives determined by the learning rate.

4.3 Solving the OR problem

We can use this same neuron to solve the OR problem. All we need to to is to change the data to examples of the OR function:

```
10 Xs = np.array([(0,0),(0,1),(1,0),(1,1)])
11 Ys = np.array([0,1,1,1])
```

We also need more epochs to train the neuron in this case, since each epoch consists of only four examples. You can also change the loss function to a quadratic loss:

```
10 def mse_loss(predicted,y):
11     t_y = tf.constant(y.astype(np.float32))
12     diffs= tf.math.square(t_y-predicted)
13     cost = tf.reduce_mean(diffs)
14     return cost
```

But the rest of the code should work unchanged.

4.4 Exercises

Start by implementing the examples shown above in this lecture. Make sure you understand the operations, the shapes of the resulting tensors and how everything fits together.

To solve the XOR problem, we need more than a neuron. Create a network with two neurons on the first layer (the hidden layer) and one output neuron. Note that you need to include this extra layer in the prediction function and also obtain the gradients to all the variable tensors you use for your network. Use these data for the XOR problem examples:

```
10 Xs = np.array([(0,0),(0,1),(1,0),(1,1)])
```

```
11 Ys = np.array([0,1,1,0])
```

Try varying the learning rate and use a larger number of epochs. You can also try using momentum. When you create the `SGD` optimizer you can add a `momentum` argument (*e.g.* 0.9). Then you can also apply this network to the data sets provided in the zip file and, if you want, make the network more complex by adding more neurons or even an additional layer. Note, however, that if you add more layers training may become significantly slower. We will see why in future lessons.

4.5 Questions

- If the batch of examples is presented as a matrix with examples in rows and features in columns, why do we encode the layers with the weights of each neuron in a column?
- What is the purpose of the `GradientTape` object used in the `grad` function?
- What does the optimizer do when you call the `apply_gradients` method?

Bibliography

[1] Bernhard E Boser, Isabelle M Guyon, and Vladimir N Vapnik. A training algorithm for optimal margin classifiers. In *Proceedings of the fifth annual workshop on Computational learning theory*, page 144–152. ACM, 1992.

[2] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[3] Lisa Anne Hendricks, Zeynep Akata, Marcus Rohrbach, Jeff Donahue, Bernt Schiele, and Trevor Darrell. Generating visual explanations. In *European Conference on Computer Vision*, page 3–19. Springer, 2016.

[4] Geoffrey E Hinton. Learning multiple layers of representation. *Trends in cognitive sciences*, 11(10):428–434, 2007.

[5] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, page 1097–1105, 2012.

[6] Yann Le Cun, Leon Bottou, and Yoshua Bengio. Reading checks with multilayer graph transformer networks. In *1997 IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 1, page 151–154. IEEE, 1997.

[7] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[8] Yann LeCun, Léon Bottou, Yoshua Bengio, Patrick Haffner, et al. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[9] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[10] Josh Patterson and Adam Gibson. *Deep learning: A practitioner's approach*. " O'Reilly Media, Inc.", 2017.

[11] John D. Owens Purcell, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron Lefohn, and Timothy J. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[12] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In *Advances in neural information processing systems*, page 91–99, 2015.

[13] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[14] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533, 1986.

[15] Pierre Sermanet, David Eigen, Xiang Zhang, Michaël Mathieu, Rob Fergus, and Yann LeCun. Overfeat: Integrated recognition, localization and detection using convolutional networks. *arXiv preprint arXiv:1312.6229*, 2013.

[16] Edward H Shortliffe and Bruce G Buchanan. A model of inexact reasoning in medicine. *Mathematical biosciences*, 23(3-4):351–379, 1975.

[17] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[18] Sandro Skansi. *Introduction to Deep Learning: From Logical Calculus to Artificial Intelligence*. Springer, 2018.

[19] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, page 1–9, 2015.

[20] David H Wolpert, William G Macready, et al. No free lunch theorems for optimization. *IEEE transactions on evolutionary computation*, 1(1):67–82, 1997.

[21] Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *European conference on computer vision*, page 818–833. Springer, 2014.