Técnicas de IA para Biologia (parte 1)

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 two important fields in modern AI, with special relevance to biology and bioinformatics: deep learning and ontologies. In this first part, 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 and unsupervised learning, with a special focus on seeing how to implement these networks in practice using Tensorflow 2.0 and Keras.

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." [35] 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 [33], 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
streptococcus
```

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

While initial expectations were high, accompanied by considerable research efforts funded by major agencies, and some expert systems reached commercial success in the 1970s, unavoidable problems became evident. First, what would work on a small scale was increasingly difficult to use on larger bodies of knowledge due to the inherent computational complexity involved when performing reasoning in general. Second, such rules are brittle in that they apply exactly in the way laid out, any minor deviations requiring the addition of further rules to adjust the system. Also, every time one would want to build a new system, the expert knowledge had to be gathered, curated and maintained, which is a time-consuming labor intensive task, known as the knowledge aquisition problem. This eventually lead to an AI winter where funding of research was cut back drastically.

A different branch of AI was more focused on learning automatically, and artificial neural networks were part of this branch right from the beginning, with the work of McCulloch and Pitts [22] and Rosenblatt's perceptron [29].

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.

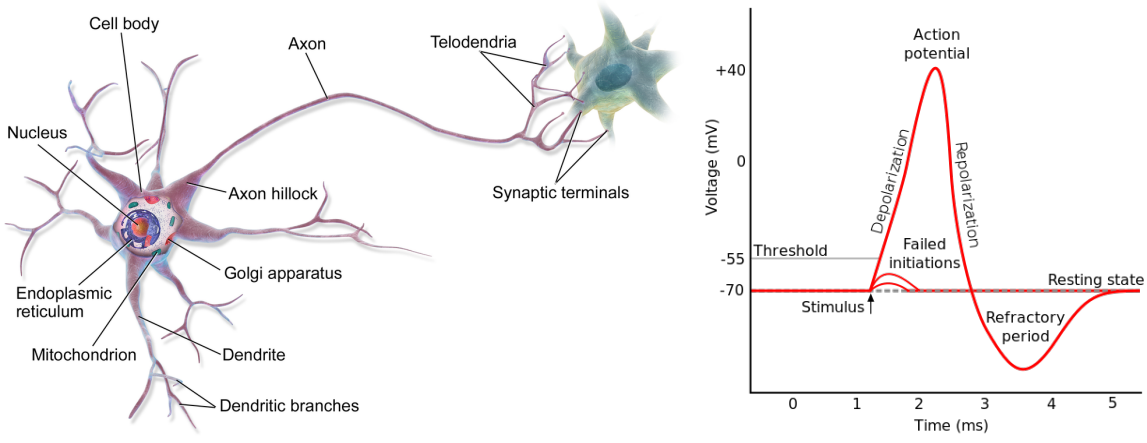


Figure 1.1: Neuron anatomy (BruceBlaus, CC-BY, source Wikipedia) and action potential response.

The neuron provides the inspiration for Rosenblatt's *perceptron*, a neuron model consisting of a linear combination of d 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, w_i the coefficient i of the perceptron, and η the learning rate. 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.

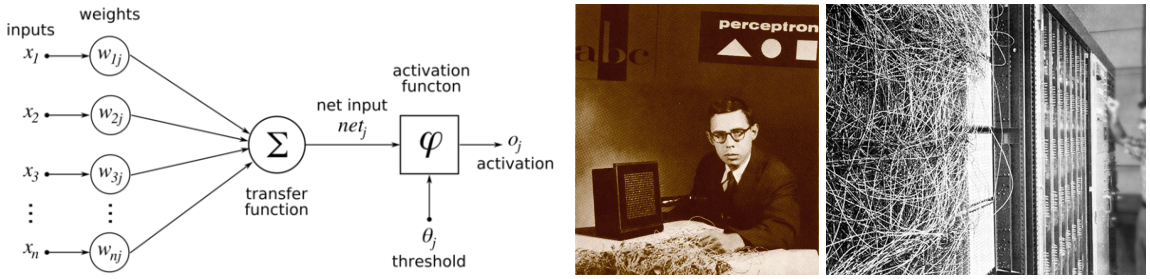


Figure 1.2: Neuron model, consisting of a weighted sum of the inputs, including a bias input of 1, and a non-linear activation function, such as the perceptron's threshold function. The middle and right panels show the original Rosenblatt perceptron (photos from the Arvin Calspan Advanced Technology 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 and neural networks, this work was disregarded for a long period of time, between around 1960 and up to the mid 1980s, a period known as the "the AI winter" (affected also by the lack of success of expert systems). Although these systems could learn linear transformations of the features, these needed to be chosen by 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 [30]. 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, such as support vector machines using kernels, for example [2]. It also meant that, despite gaining the ability to solve nonlinear learning problems, it was still necessary for humans to 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 [18], it was possible to automate reading handwritten digits. The modified National Institute of Standards and Technology (MNIST) database [19] 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 [12], 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 [12].

Furthermore, the improvement in hardware, with general-purpose Graphics Processing Units (GPU) [25] 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. This 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 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. Language models are trained with *Self-supervised Learning*, where the models are trained using the data as both features and labels, without requiring manual annotations.

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.

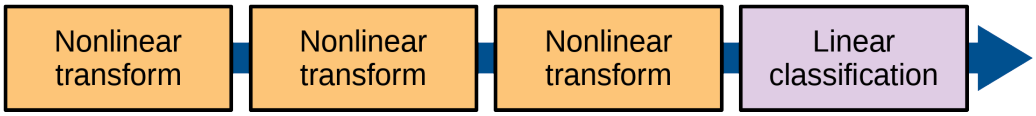


Figure 1.3: Schematic illustration of a deep learning model for classification. The last stage is a linear classifier operating on the result of multiple nonlinear transformations that change the representation of the original data.

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.

Table 1.1: XOR

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

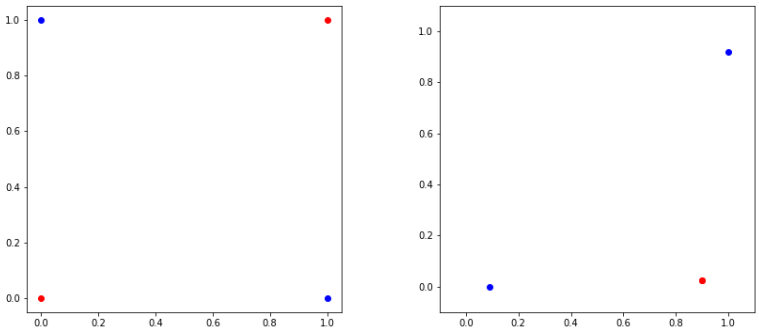


Figure 1.4: Original and transformed points from the OR function.

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.

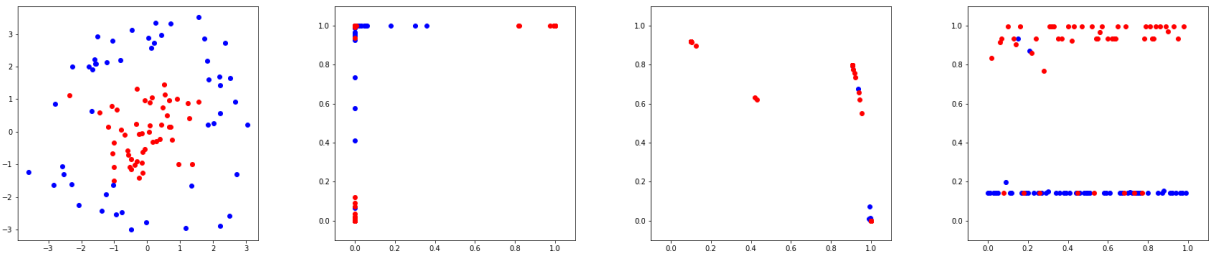


Figure 1.5: From left to right, the original data, the representation in the first hidden layer, in the second hidden layer and the classification of each data point in the vertical axis, with the points sorted in the horizontal axis.

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). Figure 1.6 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 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.

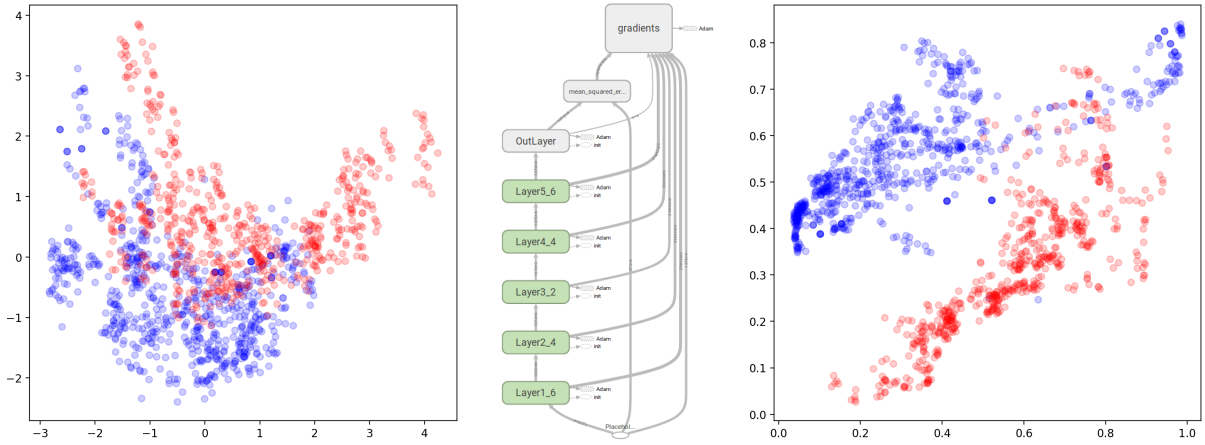


Figure 1.6: From left to right, a PCA projection of the original data using the first 2 components, the graph of the autoencoder and the representation learned in the middle hidden layer. The colors represent the different classes for the banknotes.

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:

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

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:

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

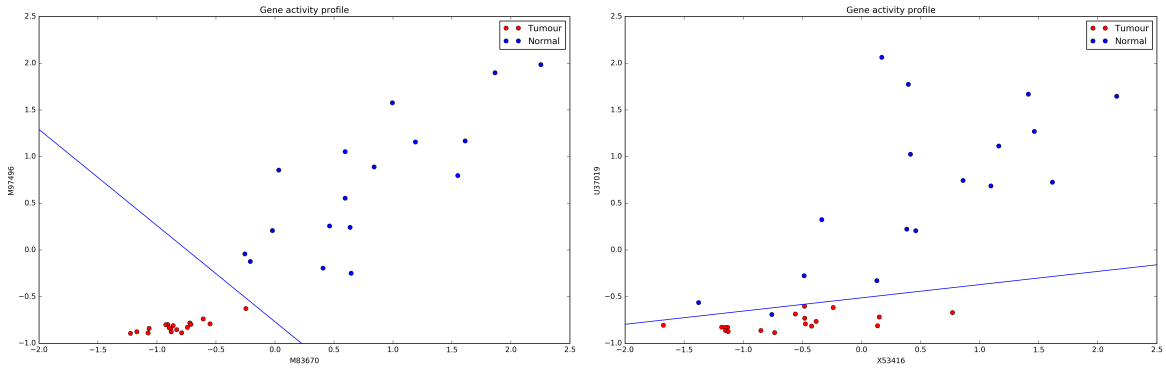


Figure 1.7: Linear classifiers are adequate for linearly separable classes but are unable to separate classes that are not linearly separable.

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

Here, $K(\vec{x}_n, \vec{x}_m)$, 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 $x_1 x_2, x_1^2, x_2^2, x_1^3, x_2^3$.

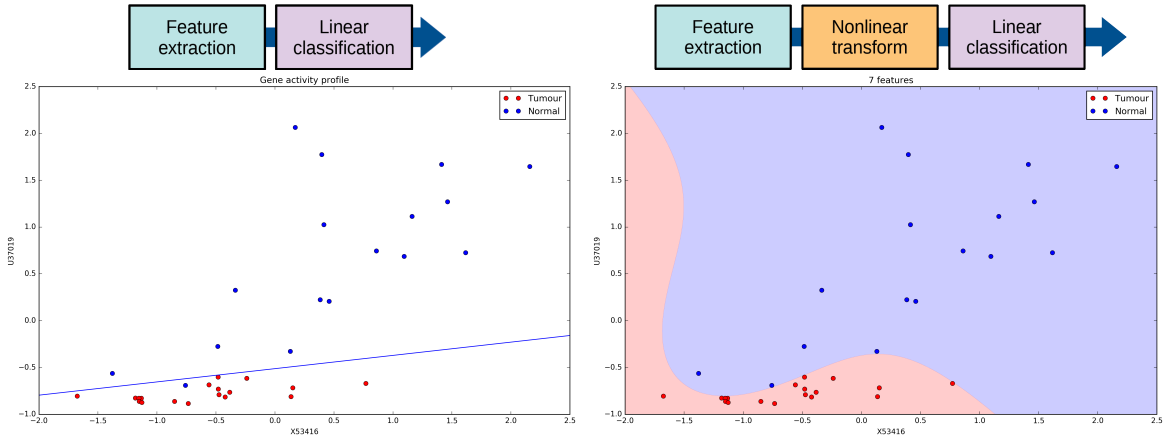


Figure 1.8: Linear classification without (left) or with (right) a non-linear transformation of the original features. In shallow classifiers, this transformation is generally controlled by hyperparameters that are fixed and not learned from the data. For example, the kernel function in a support vector 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 [38], which we will cover in a bit more detail in the next section.

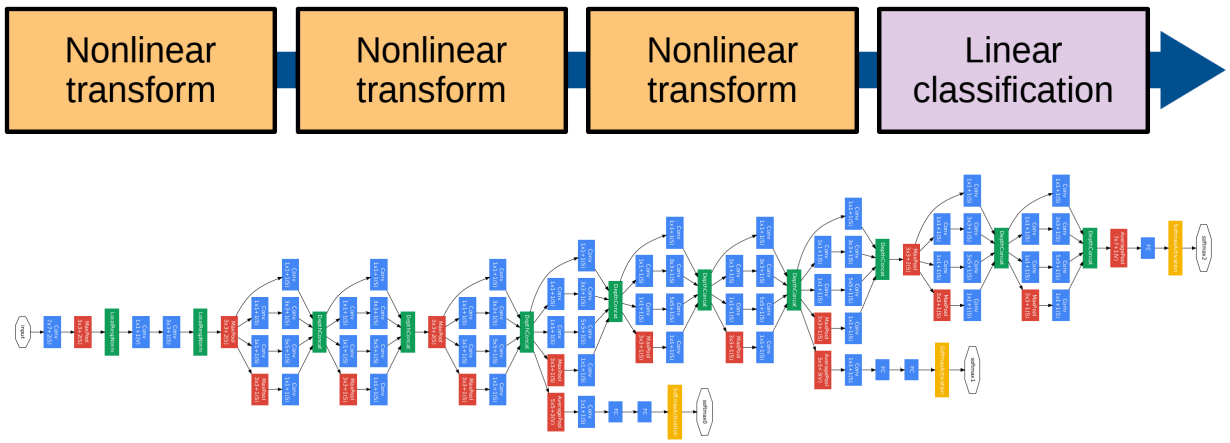


Figure 1.9: Example of a deep classifier, the GoogLeNet network, [38].

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

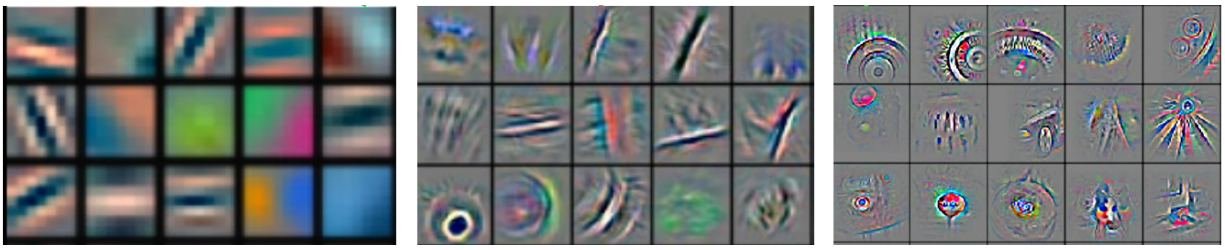


Figure 1.10: Example of representations in a deep convolution network, showing different aspects of the input images being detected at different layers, such as straight edges in the first layer, more complex edges in the second, complex shapes in the third and so on. Image from Zeitler 2014 [42].

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 [41].

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.

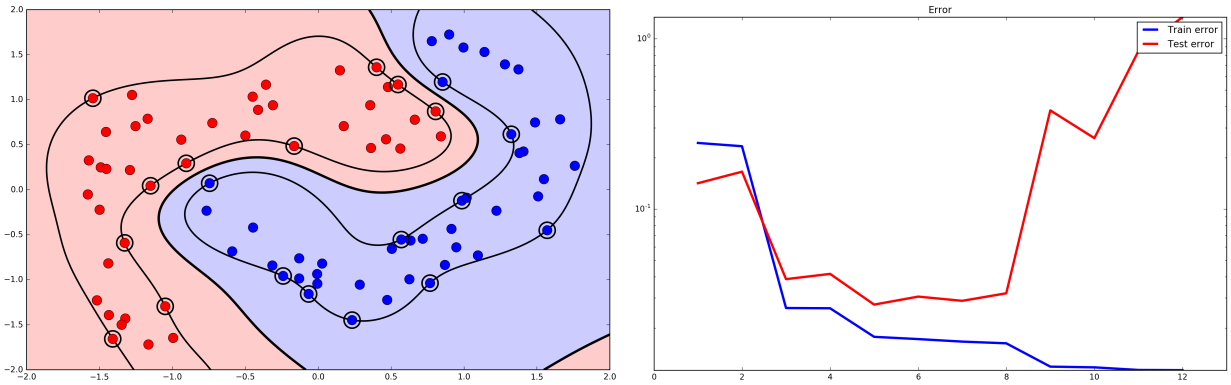


Figure 1.11: Example of overfitting. By adjusting too much to the training data, the resulting hypothesis may generalize poorly to data outside the training set (the test data in the right panel), leading to a greater true error, estimated by the test error, the greater the ability to adjust to the training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

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 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[17]. 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[17]
- OverFeat, Sermanet et al. 2013, 13.8% error[31]
- VGG Net, Simonyan, Zisserman 2014, 7.3% error[34]
- GoogLeNet, Szegedy et al. 2014 6.6% error[38]
- ResNet, He et al. 2015 5.7% error[27]

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 [17].

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 [38].

But deep models are not just for image classification. For example, in 2016 Hendricks et. al. [11] 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 [35]
2. Goodfellow *et. al.* Deep Learning [9], Chapters 1 and 5

Chapter 2

Training Neural Networks

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

2.1 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 2.1.

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

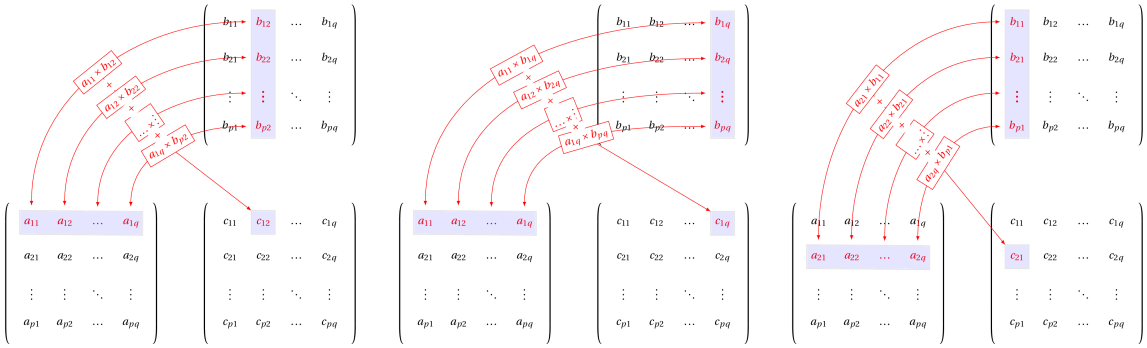


Figure 2.1: Matrix multiplication

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.

2.2 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.

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

2.3 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), which, in turn, is 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 2.2 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*.

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 2.1.

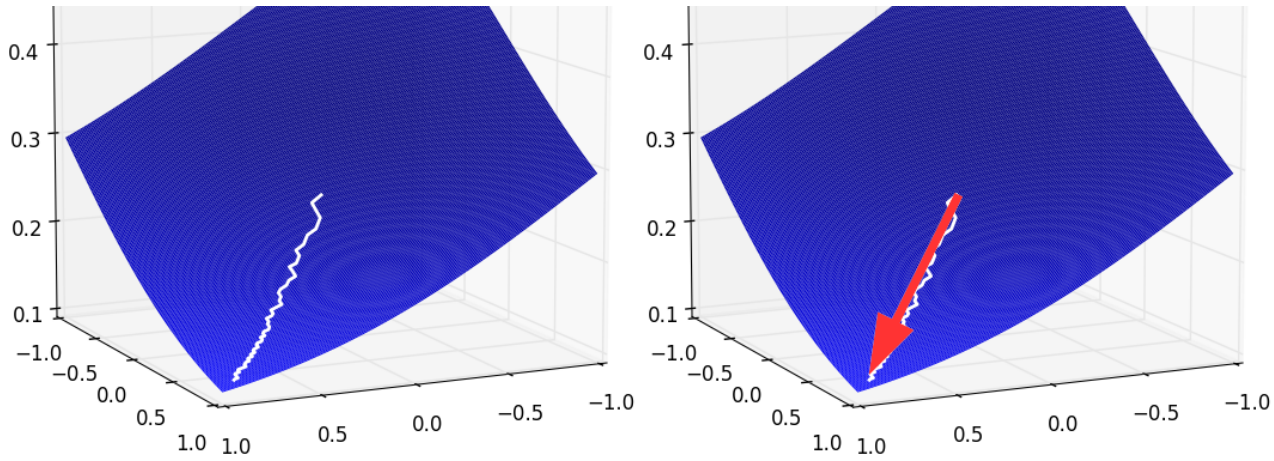


Figure 2.2: Stochastic gradient descent with online training (left panel) and batch training (right panel).

Table 2.1: The OR function

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

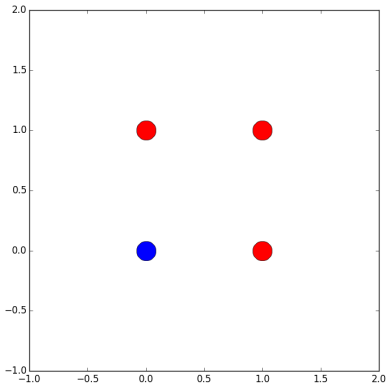


Figure 2.3: Set of points from the OR function.

Figure 2.4 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.

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 2.6.

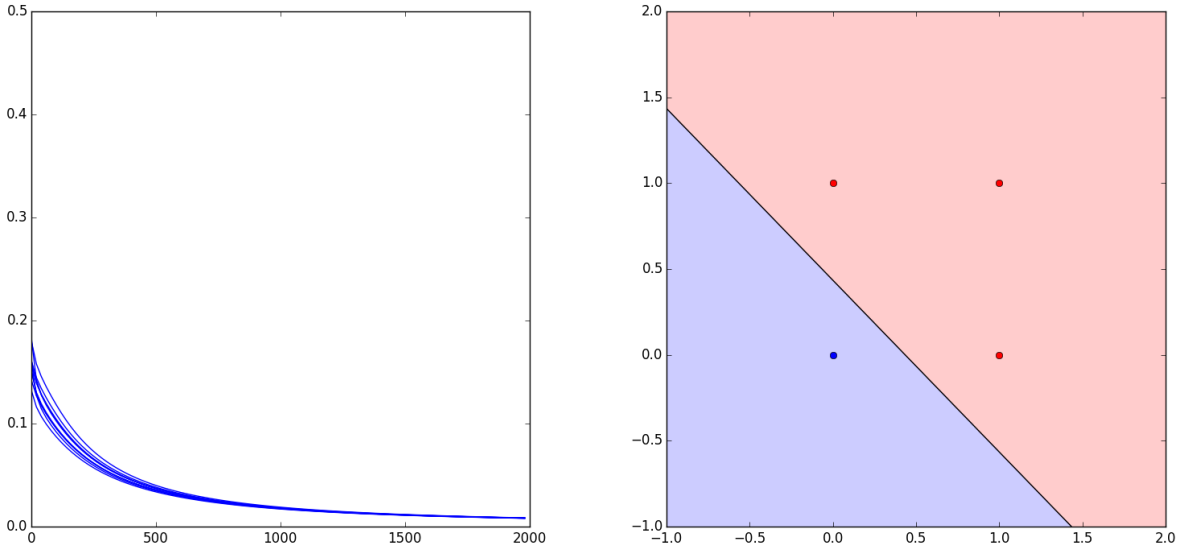


Figure 2.4: Training error and final classifier for one neuron trained to separate the classes in the OR function.

Table 2.2: The XOR function

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

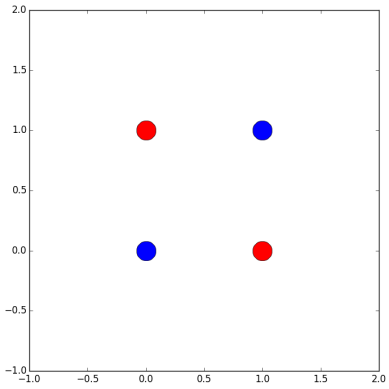


Figure 2.5: Set of points from the OR function.

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

2.4 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 2.7 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:

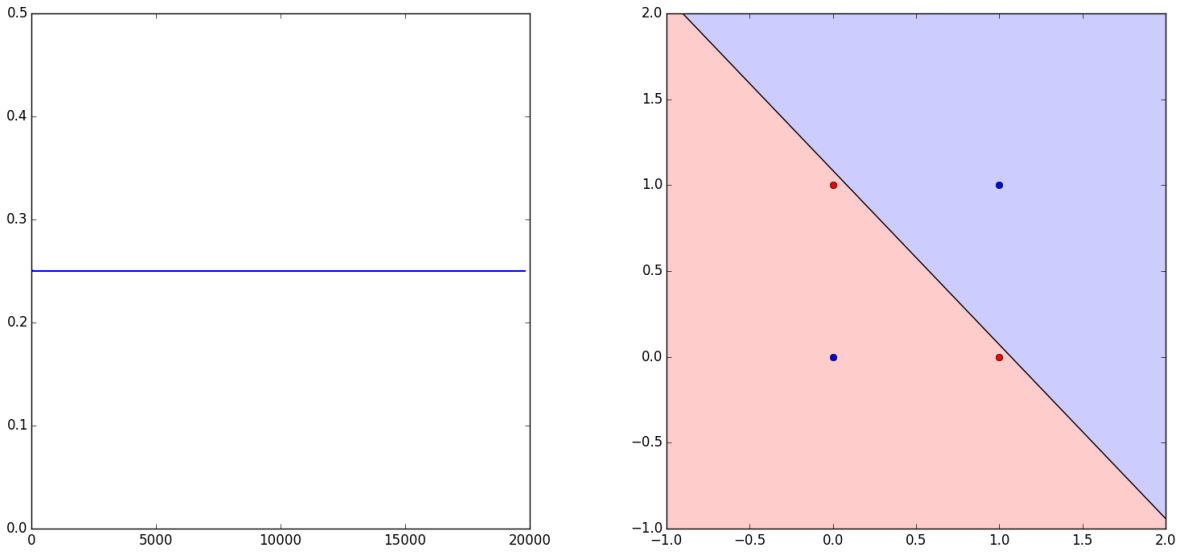


Figure 2.6: Training error and final classifier for one neuron trained to separate the classes in the OR function.

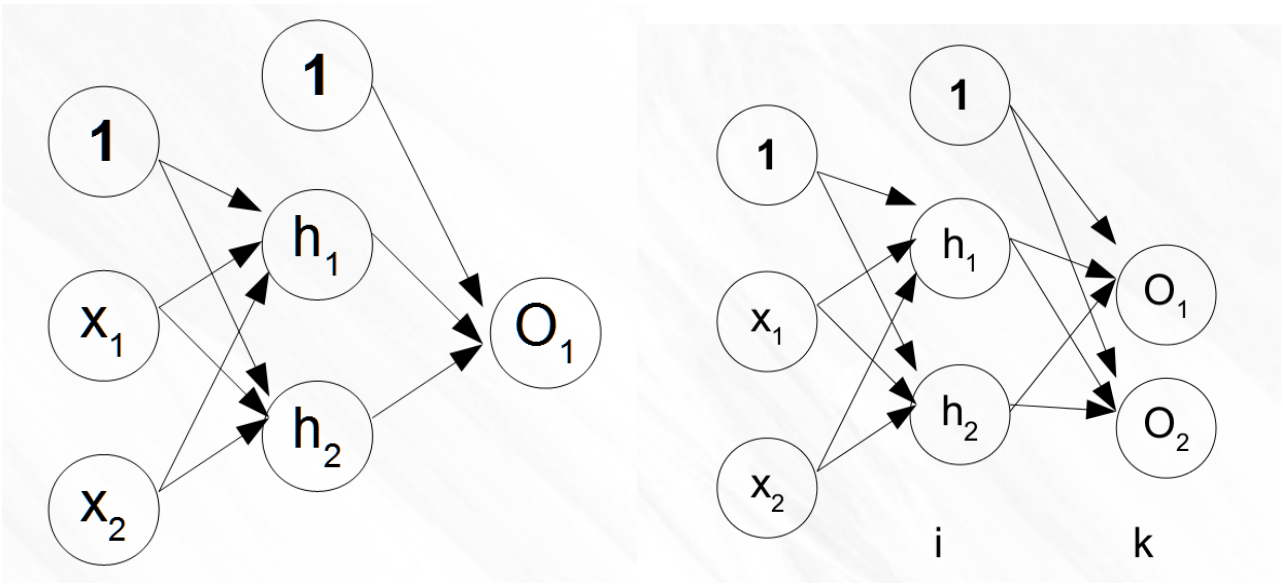


Figure 2.7: Two examples of multilayer perceptrons. Both have a hidden layer. The left panel shows a MLP with one output neuron, the right panel an MLP with two output neurons.

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

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:

$$\Delta w^j_{m,i,n} = -\eta \left(\sum_p \frac{\delta E^j_{k,p}}{\delta s^j_{k,p}} \frac{\delta s^j_{k,p}}{\delta net^j_{k,p}} \frac{\delta net^j_{k,p}}{\delta s^j_{i,n}} \right) \frac{\delta s^j_{i,n}}{\delta net^j_{i,n}} \frac{\delta net^j_{i,n}}{\delta w_{m,i,n}}$$

$$= \eta(\sum_p \delta_{kp} w_{m,k,p}) s^j_{in}(1 - s^j_{i,n}) x^j_i = \eta \delta_{i,n} x^j_i$$

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 update the weights.

With this algorithm and the MLP architecture shown on the left panel of Figure 2.7, 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.

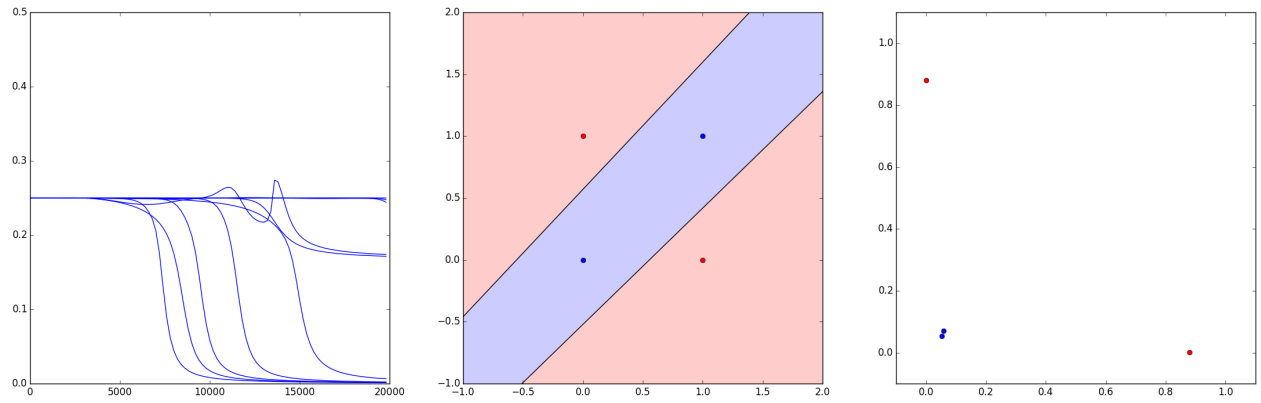


Figure 2.8: Training the MLP with one hidden layer for classifying the XOR function output. The first panel shows the training error over 10 training runs. Note that, due to the stochastic initialization and ordering of the examples presented, there are differences between different runs. The second 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.

2.5 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. 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, which also 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 2.9 shows.

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

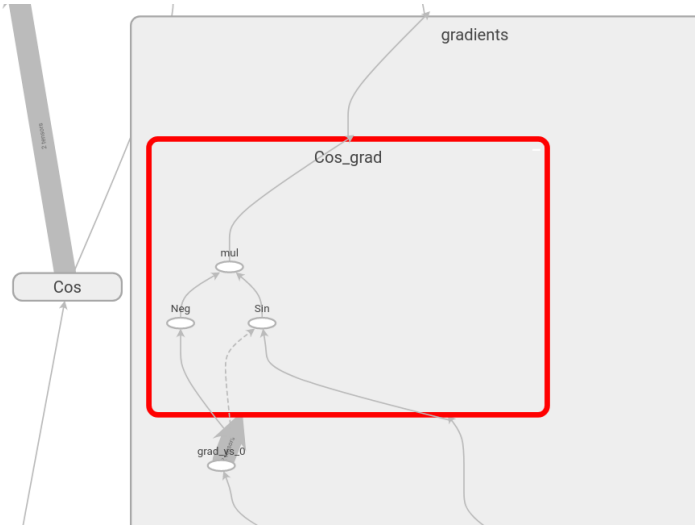


Figure 2.9: Matrix multiplication

2.6 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 [9] 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 sigmoid 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.

2.7 Introduction to the Keras Sequential API in Tensorflow

To build a model using the Keras Sequential API, you just need to create a `Sequential` object an then add the different layers. In this first tutorial we will only be using dense layers (fully connected to the previous layer) with sigmoid activation. Start by importing the necessary classes. For example:

```
from tensorflow.keras.optimizers import SGD
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
```

Now we create the model. To illustrate, we will add an initial layer of four neurons with as many input values as the dimension of our data vectors (in variable `inputs`, which has the value 2 with the date used in the tutorial below), then another layer with four neurons and finally a single output neuron. All neurons will have a sigmoid activation and for the deeper layers Keras automatically determines the number of input values for each neuron. The first layer needs the input shape specified if we want to compile it to see a summary before training it. Note that the batch size is omitted (will be specified later) and the shape corresponds to a single example.

```
model = Sequential()
model.add(Dense(4, activation = 'sigmoid',input_shape=(inputs,)))
model.add(Dense(4, activation = 'sigmoid'))
model.add(Dense(1, activation = 'sigmoid'))
```

To compile the model we need to select an optimizer and specify the loss function and metrics to register during the run. In this case, we choose the SGD optimizer and the loss function is the mean

squared error (`mse`) which is mean of the squared differences between the predicted and the target values. Note that this is not the ideal loss function for this example (binary classification) but we will take a better look at loss functions in future lectures. After compiling you can check the model with the `summary` method.

```
opt = SGD(lr=INIT_LR, momentum=0.9)
model.compile(loss="mse", optimizer=opt, metrics=["mse"])
model.summary()
```

In this case the summary of the model will be as follows:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 4)                 12
_________________________________________________________________
dense_01 (Dense)             (None, 4)                 20
_________________________________________________________________
dense_02 (Dense)             (None, 1)                 5
=================================================================
Total params: 37
Trainable params: 37
Non-trainable params: 0
_________________________________________________________________
```

Now we can train the model, calling the `fit` method to which we supply the training features and labels, as well as the batch size (how many examples to present at each update of the model parameters) and number of epochs (how many times to go through all the examples in the data set). The `fit` method returns a `history` object with a dictionary, also called `history`, with the record of all metrics collected during training.

```
H = model.fit(X, Y, batch_size=16, epochs=10000)
plt.plot(H.history['loss'])
plot_model(model,X,Y)
```

After training we can save our model as a JSON file and the weights in a hierarchical data format (HDF5) file:

```
model.save_weights('tutorial_1.weights.h5')
model_json = model.to_json()
with open("tutorial_1.json", "w") as json_file:
    json_file.write(model_json)
```

2.8 Exercise: Training a neural network

Load the data in the file `reordered_90.txt`. The first two columns contain the feature values, which are the input values to the model. The last column is the target class to predict (0 or 1). Separate these values and standardize the features (and not the class label). Here is an example of how to do this. Note that you need to import the `numpy` library.

```
data = np.loadtxt('reordered_90.txt')
X = data[:,:2]
Y = data[:,2]
means = np.mean(X,axis=0)
stds = np.std(X,axis=0)
X = (X-means)/stds
```

Now create a simple neural network as described in the previous section. The idea is to have a single output neuron with a sigmoid activation, which outputs values between 0 and 1, and train the network to make these values as close as possible to the target labels. To do this we can minimise the quadratic error (the square of the difference between the predicted value and the target value). This is not the best loss function for this application but we will see more on this later.

Experiment with a single neuron, with 2 layers (several neurons followed by a single neuron), with 3 layers, with more or fewer neurons. Also check the history for the progression of the loss function during training. You can also use the `plot_model` function provided to see how your model is classifying points in the data space.

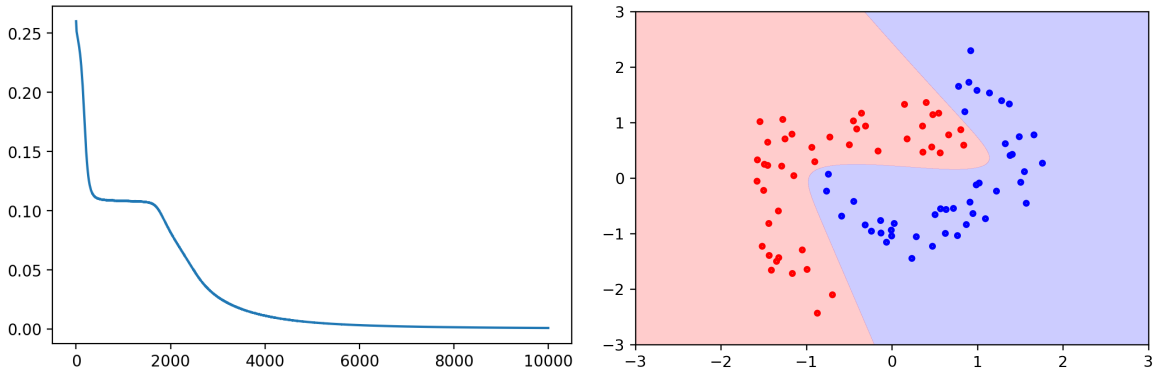


Figure 2.10: The training loss and final result of a 4x4x1 network. The training loss plot is useful to check if the loss function levelled out or if it was still worth it to continue training. We will also see in future how to plot the validation loss (or other metrics) to check for overfitting.

2.9 Optional: other activation functions

Check the Tensorflow Playground site here: https://playground.tensorflow.org. Experiment with the different data sets and different architecture. Note how ReLU activations are much more efficient than sigmoid for hidden layer neurons, taking fewer epochs to learn. You can try this on your own network by replacing the sigmoid with `relu` activations (but only on the hidden layers, not on the output neuron which we want to output values between 0 an 1). You can get similar results with far fewer epochs. We will see why in the next lectures.

2.10 Further Reading

1. Goodfellow et. al., Chapter 2, 4 and 8 [9]

Chapter 3

Activations, Loss Functions and Optimizing Networks

Wide and deep networks. The vanishing gradient problem. Rectified Linear Units. Choosing activation functions.Optimizers. Learning rates. Weight initialization. Batch normalization. Model selection, overfitting, bias and variance. Regularization. Choosing hyperparameters

3.1 Wide vs Deep

The universal approximation theorem, proven by Cybenko in 1989 [4], implies that a single hidden layer is sufficient to approximate any function, to arbitrary precision, within a finite region. More specifically, given a function ϕ that is not constant, is bounded and continuous and the unit volume $I_m = [0,1]^m$ for any function f continuous in I_m there are N constants v_i, b_i and a $\vec{w}_i$ such that:

$$F(\vec{x}) = \sum_{i=1}^{N} v_i \phi(\vec{w_i}^T \vec{x} + b_i) \qquad |F(\vec{x}) - f(\vec{x})| < \epsilon$$

for all $\vec{x} \in I_m$, with $\epsilon > 0$.

However, this approximation may require a very large number of neurons.

Yet, it has still not been demonstrated whether deep (increasing the number of layers) or wide (increasing the number of neurons per layer) networks are preferable [20]. Telgarsky [39] demonstrated that deep networks are more capable of approximating oscillating functions with fewer elements. This is important not only for the ability to approximate functions in general but also because this is related to the *Vapnik-Chervonenkis dimension* (VC dimension) of a classifier. The VC dimension of a classifier is the dimension of the largest set of examples a model shatters. A model shatters a set of examples if it can provide hypotheses that classify all those examples correctly whatever the class each example belongs to. For example, a linear classifier in two dimensions can shatter a set of 3 points forming a triangle, as shown in Figure 3.1. However, with 4 examples this is not true, as we saw in the case of the XOR function previously.

In general, functions that are more capable of oscillations can be used as classifiers with higher VC dimension, suggesting that deep neural networks have a larger classification power than shallow

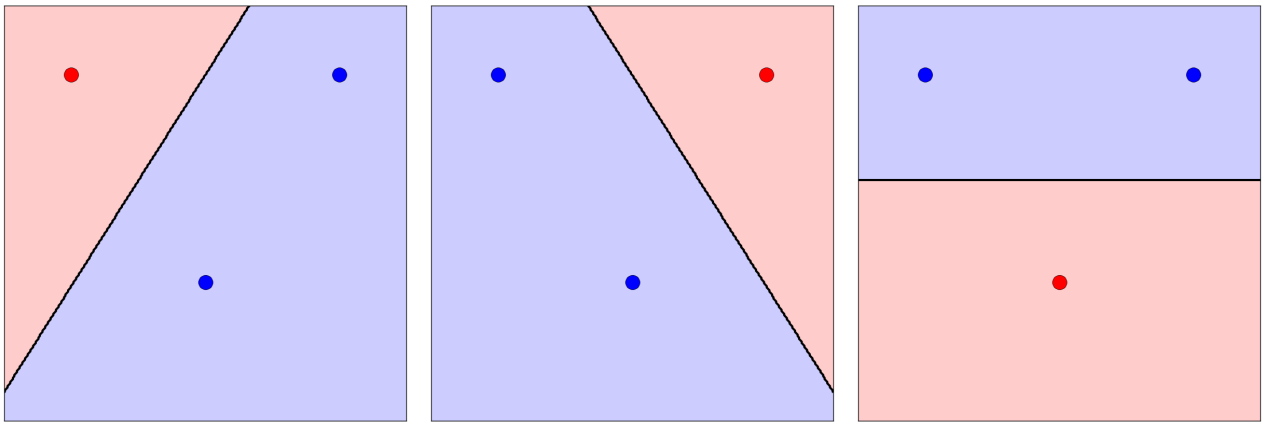


Figure 3.1: A linear classifier in two dimensions can shatter this set of 3 points by correctly classifying them whatever their labels. Note that two other cases, where all points belong to the same class, were omitted for being trivial.

and wide neural networks with the same number of neurons. However, this does not mean that deep networks will be best in all cases.

3.2 No Free Lunch

"... for any two learning algorithms A and B, [...] there are just as many situations (appropriately weighted) in which algorithm A is superior to algorithm B as vice versa." (David Wolpert [41])

The "no free lunch" theorems are a set of theorems in statistics, machine learning and optimization that show that any approach that performs better at solving one problem will perform worse on other problems. Thus, considering the universe of all theoretically possible problems, no algorithm is better than any other. In the case of machine learning, Wolpert demonstrated that test error, which measures the capacity for generalization outside of the training data, will be the same for all machine learning algorithms when averaged over all possible distributions of the data.

However, in real life data does not come from all mathematically possible distributions. Rather, for each problem, data follows some particular distribution and it is solely that distribution that is relevant. Thus, in real applications, there are algorithms and models that perform better than others. It is just a matter of what the data is like.

In general, shallow models composed of linear combinations combined with a non-linear transformation are easier to interpret and good at memorizing simple interactions between given features. For example, which other products were purchased by customers who purchased some product. However, these models are not suited to generalize more complex properties unless given engineered features for that purpose.

Deep models are more powerful and better at generalizing and extracting relevant features, although this extra power comes at a greater risk of overfitting. For example, [10] tested a number of deep neural networks for face recognition and found that their performance degraded substantially if the image quality fell below what the networks were trained with.

In short, deep neural networks may not be the best choice in all cases, not only because of the need for a large training set to reduce overfitting but also because of how difficult it is to interpret the trained model. Figure 3.2 shows some popular examples of images that can be easily confused. It is possible to train a deep neural network to distinguish even between categories as similar as these, but it is not

possible to determine by examining a trained deep neural network if it learned the correct features because the features are extracted by the network itself and not easily intelligible, as is the case with shallow models [1].

Figure 3.2: Three popular examples of confusing image datasets. Given a deep model, only by testing can we determine if it can distinguish between such examples since the resulting classifier is hard to interpret. Shallow models are easier to interpret but do not perform well in this type of problems. Image credits: teenybiscuit, Twitter.

3.3 Vanishing Gradients

One reason why deep neural networks were not popular for many years, even after backpropagation was introduced, was the difficulty in training such networks with classical activation functions, such as the sigmoid or the hyperbolic tangent functions. The reason for this is the chain rule in backpropagation combined with the shape of these functions. To propagate the error gradient backwards through the network we need to multiply the gradients at each step. For example, for the output neuron, we multiply the gradient of the error function, the activation function and the weighted sum of the inputs (from the previous layer) by the weights. For output neuron n of layer k receiving input from m from layer i through weight j, this is:

$$\Delta w^j_{mkn} = -\eta \frac{\delta E^j_{kn}}{\delta s^j_{kn}} \frac{\delta s^j_{kn}}{\delta net^j_{kn}} \frac{\delta net^j_{kn}}{\delta w_{mkn}} = \eta (t^j - s^j_{kn}) s^j_{kn} (1 - s^j_{kn}) s^j_{im} = \eta \delta_{kn} s^j_{im}$$

For a weight m on hidden layer i, we must continue the multiplications to propagate the output error backwards from all neurons ahead:

$$\Delta w^j_{min} = -\eta \left(\sum_p \frac{\delta E^j_{kp}}{\delta s^j_{kp}} \frac{\delta s^j_{kp}}{\delta net^j_{kp}} \frac{\delta net^j_{kp}}{\delta s^j_{in}} \right) \frac{\delta s^j_{in}}{\delta net^j_{in}} \frac{\delta net^j_{in}}{\delta w_{min}}$$

When using functions like sigmoid or hyperbolic tangent (see Figure 3.3), when the neuron is activated the output levels off at one. This makes the derivative of the activation function, δs, approach zero. Since we are multiplying all the derivatives, the result will tend to zero. The problem with this is that, although the neuron is active for a particular example, all derivatives computed through that neuron will be close to zero and thus will not inform the gradient descent to reduce the cost function.

[1] See also performance tests of popular image recognition network on the chihuahua muffin classification problem at freeCodeCamp:"Chihuahua or muffin? My search for the best computer vision API", by Maryia Yao.

With networks formed of multiple layers of neurons this makes training so slow that backpropagation becomes impractical.

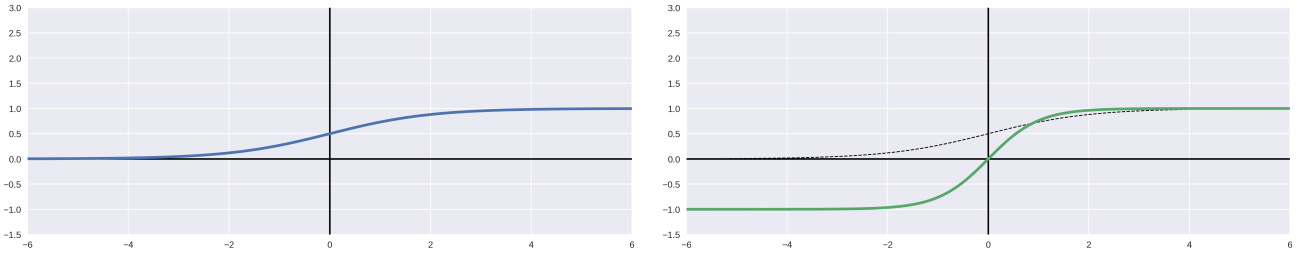


Figure 3.3: Two formerly widely used neuron activation functions: sigmoid and hyperbolic tangent. Note how the gradient drops to zero at high input values as the function saturates.

Until 2010, pre-training methods were used to help train deep networks. But with Rectified Linear Units (ReLU) the problem of vanishing gradients disappeared.

3.4 Rectified Linear Units

Rectified Linear Units (ReLU) were first used in 2009 for object recognition [14], then for restricted Boltzmann machines [24] and deep neural networks in 2011 [8]. The ReLU is a piecewise linear function that is zero for negative x_i inputs and equal to x_i for positive values of the input x_i. This is enough to provide the non-linearity necessary for the network while, at the same time, iit keeps the derivative constant and non zero when the neuron is active. Figure 3.4 shows several variants of ReLU. The leaky ReLU has a value of $a_i x_i$ for the output for negative values of x_i, with a fixed a_i value. The Randomized leaky ReLU is similar but a_i is random during training, drawn from a given probability distribution, and after training is fixed to the expected value of that distribution. In another variant, the Parametric ReLU, the a_i value is also learned during training. The Exponential Linear Unit is similar to a ReLU in the effect of avoiding the vanishing gradients for active neurons, but has a continuous derivative, with the activation value equal to $a(e^{x_i} - 1)$ for negative inputs [3]. These variants solve a problem with the original ReLU which is the "death" of neurons that become inactive over all the training set, since the gradients for these neurons in the original ReLU becomes zero and their weights will thus no longer change.

Concatenated ReLU is another ReLU variant that combines linear activations, with opposing slopes, for positive and negative inputs [32].

3.5 Choosing activation functions

In 2015, Xu *et al* compared the performance of different ReLU variants in deep convolution networks on the CIFAR image datasets, concluding that a non-zero slope in the negative inputs improves performance over the original ReLU. Thus, leaky variants of ReLU are generally used in the hidden layers of a deep network.

For the output layer, the activation function depends on the type of problem we are solving. For regression problems, the output neuron should not have an activation function, outputting only the weighted sum of the inputs multiplied by the neuron weights. This makes it possible for the neuron to output any desired value for the regression function.

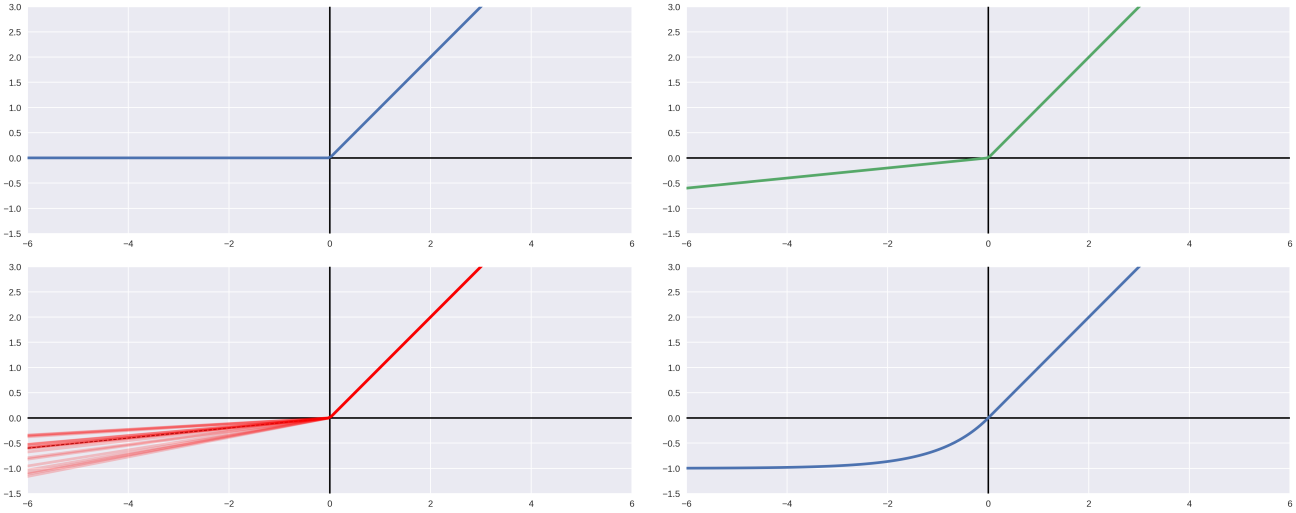


Figure 3.4: Rectified Linear Units and similar variants: ReLU; Leaky ReLU, with a non zero slope for negative values of x; Randomized Leaky ReLU, where the slope for negative inputs is random; and the Exponential Linear Unit, with a continuous derivative.

For binary classification problems, it is useful to have an output value that can give us a probability of the example belonging to one of the two classes. One useful output function in this case is the sigmoid function.

For n-ary classification problems, the most used activation function is the *softmax* function:

$$\sigma : \mathbb{R}^K \rightarrow [0,1]^K \qquad \sigma(\vec{x})_j = \frac{e^{x_j}}{\sum_{k=1}^{K} e^{x_k}}$$

This function returns a vector of values where $\sigma_j \in [0,1]$ and $\sum_{k=1}^{K} \sigma_k = 1$, which means that σ_j can be used to represent the probability of the example belonging to class j of k.

3.6 Loss and Likelihood

We train neural networks by minimizing a loss (or cost) function with the objective of finding the best parameters. But how do we decide which function to miminize? One approach is to choose a function that corresponds to a *maximum likelihood* solution, meaning that we instantiate our model with parameters that maximize the probability of the data being distributed as we find it to be. This is a common approach to machine learning and neural networks [2]. Let us consider a simple problem of linear regression.

A *linear regression* is a regression in which the hypothesis class corresponds to the model $y = \theta_1 x_1 + \theta_2 x_2 + ... + \theta_{n+1}$, where each x_n is one dimension of the input space. Suppose, to simplify, that our input space has only one dimension and we have a set of (x, y) points and want to find the best way to predict the y value of each point given the x value assuming that the best fit is some straight line $y = \theta_1 x + \theta_2$. Figure 3.5 shows an example of a data set of points and possible lines from our *hypothesis class*, obtained by instantiating the model with different values of θ_1 and θ_2.

[2] Another good approach is Bayesian learning, but we will not cover that in this course

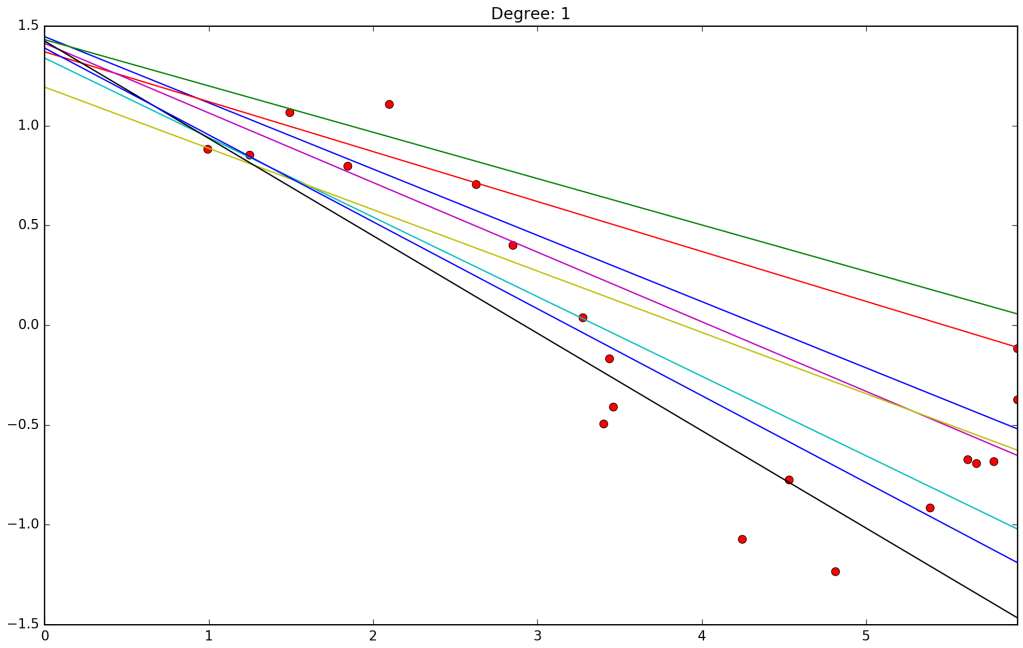


Figure 3.5: Example of lines for predicting the y values in these data.

How can we determine the best line? Let us assume that the dependent variable y is some (unknown) function of the independent variable x plus some error:

$$y = F(x) + \epsilon$$

We want to approximate $F(x)$ with a model $g(x, \theta_1, \theta_2)$. Assuming that the error is random and normally distributed:

$$\epsilon \sim N(0, \sigma^2)$$

then, if $g(x, \theta_1, \theta_2)$ is a good approximation of the true function $F(x)$, the probability of having a particular y value given some x value can be computed from our function $g(x, \theta_1, \theta_2)$ as:

$$p(y|x) \sim \mathcal{N}(g(x, \theta_1, \theta_2), \sigma^2)$$

This allows us to estimate the probability of the data coming out with the distribution we observe in our data set given any hypothesis instantiating θ, representing the vector of all $\theta_1, ..., \theta_n$ parameters (in this case, θ_1, θ_2). The probability of the data given the hypothesis is the *likelihood* of the hypothesis. Note that we cannot assume a probability for the hypothesis, at least in a frequentist sense, because the hypothesis is not a random variable. What we assume to be random here is the sampling of data that resulted in obtaining this dataset from the universe of all possible data.

Thus, given our dataset $\mathcal{X} = \{x^t, y^t\}_{t=1}^N$ and knowing that $p(x, y) = p(y|x)p(x)$, then the likelihood of the set of parameters θ is

$$l(\theta|\mathcal{X}) = \prod_{t=1}^{n} p(x^t, y^t) = \prod_{t=1}^{n} p(y^t|x^t) \times \prod_{t=1}^{n} p(x^t)$$

Now we know how to choose the best hypothesis: we pick the one with the *maximum likelihood*. In other words, we pick the hypothesis that estimates the largest probability of obtaining the data we have. To simplify, let us change the expression by noting that the hypothesis that maximizes the likelihood also maximizes the logarithm of the likelihood, so we can focus on the logarithm of the likelihood, $\mathcal{L}$, instead of the likelihood l:

$$\mathcal{L}(\theta|\mathcal{X}) = log \prod_{t=1}^{n} p(y^t|x^t) + log \prod_{t=1}^{n} p(x^t)$$

We can also ignore the $p(x)$ term since this corresponds to the probability of drawing those x values in our data from the universe of possible values and this is the same for all hypotheses (all values of θ) we are considering.

$$\mathcal{L}(\theta|\mathcal{X}) \propto log \prod_{t=1}^{n} p(y^t|x^t)$$

Since we assume that the probability of obtaining some y value given some x is approximately normally distributed around our prediction, we can replace that term with the corresponding distribution:

$$p(y|x) \sim \mathcal{N}(g(x,\theta), \sigma^2)$$

and then replace it with the expression for the normal distribution:

$$\mathcal{N}(z,\mu,\sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(z-\mu)^2/2\sigma^2}$$

leaving:

$$\mathcal{L}(\theta|\mathcal{X}) \propto log \prod_{t=1}^{n} \frac{1}{\sigma\sqrt{2\pi}} e^{-[y^t - g(x^t|\theta)]^2/2\sigma^2}$$

which can be simplified to:

$$\mathcal{L}(\theta|\mathcal{X}) \propto log \prod_{t=1}^{n} e^{-[y^t - g(x^t|\theta)]^2}$$

$$\mathcal{L}(\theta|\mathcal{X}) \propto -\sum_{t=1}^{n} [y^t - g(x^t|\theta)]^2$$

But this is the expression of the square of the training error:

$$E(\theta|\mathcal{X}) = \sum_{t=1}^{n} [y^t - g(x^t|\theta)]^2$$

So, basically, to find the hypothesis with the *maximum likelihood*, we need (under our assumptions) to find the hypothesis with the *minimum squared error* on our training set. This problem is called a *Least Mean Squares minimization*.

Note that the squared error is often represented by this expression:

$$E(\theta|\mathcal{X}) = \frac{1}{2} \sum_{t=1}^{n} [y^t - g(x^t|\theta)]^2$$

The reason for this is that, when computing the derivative of this error as a function of the parameters, the square power cancels the 2 in the denominator, simplifying the algebra. However, the values obtained for the parameters minimizing the squared error or one half the squared error are the same. This is merely an algebraic convenience.

Now we can do gradient descent on this quadratic error curve to find the best set of parameters, as Figure 3.6 illustrates

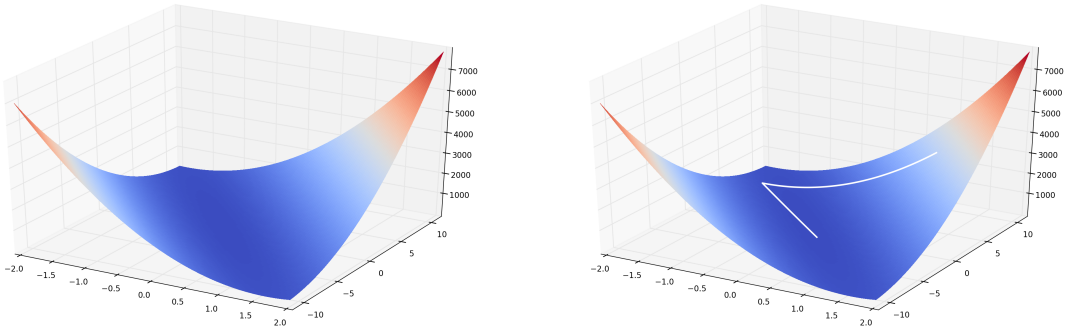


Figure 3.6: Gradient descent on the squared error surface.

3.7 Maximum Likelihood

For a general case, suppose we have a set of examples $\mathbb{X} = \{x^1, ..., x^m\}$ drawn randomly from the population with some probability distribution. We also have a family of probability distributions $p_{model}(x; \theta)$ which tell us the probability of x as a function of θ. The maximum likelihood estimator for θ is the vector that maximizes the joint probability of all the examples being what they are:

$$\theta_{ML} = \arg\max_{\theta} p_{model}(x; \theta) = \arg\max_{\theta} \prod_{i=1}^{m} p_{model}(x^i; \theta)$$

Since multiplying many small numbers will often lead to numerical underflow, it is best to use logarithms:

$$\arg\max_{\theta} \prod_{i=1}^{m} p_{model}(x^i; \theta) = \arg\max_{\theta} \sum_{i=1}^{m} \log\, p_{model}(x^i; \theta)$$

Since the parameters corresponding to the maximum value do not change by a simple rescaling of the function, we can divide by m the expectation function of the log-probabilities of an x according to our model considering the empirical distribution of examples in our data. In other words, we are maximizing the probability of any x given by our model weighted by the probability of that x being drawn from the distribution probability of examples.

$$\arg\max_{\theta} \mathbb{E}_{x \sim \hat{p}_{data}} \log\, p_{model}(x; \theta)$$

This value is maximized when $p_{model}(x; \theta)$ is as similar as possible to the distribution of examples $\hat{p}_{data}(x)$. This can be measured by the Kullback–Leibler divergence, a measure of the dissimilarity between different distributions and which is the expectation of the log-probability differences between them. In our case:

$$D_{KL}(\hat{p}_{data}, p_{model}) = \mathbb{E}_{x \sim \hat{p}_{data}} [\log \hat{p}_{data} - \log p_{model}]$$

Since $\hat{p}_{data}$ does not depend on θ, minimizing the KL divergence is the same as minimizing the expectation of the $-\log p_{model}$, which is equivalent to the maximization we had before:

$$\arg\min_{\theta} \mathbb{E}_{x \sim \hat{p}_{data}} - \log\, p_{model}(x; \theta) = \arg\max_{\theta} \mathbb{E}_{x \sim \hat{p}_{data}} \log\, p_{model}(x; \theta)$$

This means that maximizing likelihood is equivalent to minimizing the divergence between the probability distribution of the data we draw and the probability distribution of the data given by your model. This corresponds to minimizing cross-entropy between the two distributions.

In supervised learning, we want to adjust our parameters θ to predict some set of target values or labels Y from the features X. So in this case the maximum likelihood solution can be written as

$$\theta_{ML} = \arg\max_{\theta} P(Y|X;\theta) = \arg\max_{\theta} \sum_{i=1}^{m} \log P(y^i|\vec{x}^i;\theta)$$

For linear regression, as we saw before, we assumed that any y is given by our function plus a normally distributed error around that value, $p(y|x) \sim \mathcal{N}(g(x,\theta),\sigma^2)$, and in this case the cost function (cross-entropy) is the squared error:

$$E(\theta|\mathcal{X}) = \sum_{t=1}^{n} [y^t - g(x^t|\theta)]^2$$

3.8 Binary Classification

If we have a binary classification problem using a sigmoid activation function to predict the probability of each example belonging to one class based on the θ parameters and the features x, then if $g(\vec{x},\theta) = P(t_n = 1|\vec{x})$ and $t_n \in \{0,1\}$, the maximum likelihood solution corresponds to:

$$\mathcal{L}(\theta|X) = \prod_{n=1}^{N} \left[g_n^{t_n}(1-g_n)^{1-t_n}\right] \quad l(\theta|X) = \sum_{n=1}^{N} [t_n \ln g_n + (1-t_n)\ln(1-g_n)]$$

where $g(\vec{x},\theta)$ is the sigmoid activation of our output neuron. In this case, the maximum likelihood solution corresponds to minimizing the logistic loss:

$$E(\widetilde{w}) = -\frac{1}{N}\sum_{n=1}^{N} [t_n \ln g_n + (1-t_n)\ln(1-g_n)] \quad g_n = \frac{1}{1+e^{-(\vec{w}^T\vec{x_n}+w_0)}}$$

3.9 Multi-class Classification

If we have a multi-class classification problem, then we want to use the softmax function to predict the probability of each example belonging to each class, as we saw in the previous chapter, and the corresponding maximum likelihood solution is the softmax cross entropy:

$$-\sum_{c=1}^{C} y_c \log \frac{e^{a_c}}{\sum_{k=1}^{C} e^{a_k}}$$

3.10 Optimization

Numerical optimization algorithms fall outside the scope of this course. However, it is useful to have an idea of how effective the different options are and how they work. The basic gradient descent algorithm, implemented on Tensorflow in `tf.keras.optimizers.GradientDescentOptimizer`, simply computes the gradient of the loss function and then updates the parameters in that direction

with a step proportional to the gradient and a single learning rate. Stochastic gradient descent uses the gradient computed at each example, selected at random, and mini-batch gradient descent applies the update after computing the total gradient from a batch of randomly selected examples. This randomness helps avoid local minima and improves the chances of converging to better solutions.

One problem with basic gradient descent is that the learning rate is constant, preventing the optimizer from adapting to different conditions along the minimization. This is especially problematic if the gradient varies differently along different dimensions, which may cause the gradient descent algorithm to oscillate and fail to find the best path towards minimization. The use of momentum helps solve this problem by also accumulating previous gradient directions. This causes oscillations to cancel out and the correct directions to become reinforced, speeding up optimization as we saw previously.

However, this still has the problem of using the same learning rate for all parameters, and different parameters may benefit from different update rates. The AdaGrad algorithm [6] (`tf.keras.optimizers.Adagrad`) divides the learning rate of each parameter by the sum of past (squared) gradient values for that parameter. This way the algorithm increases the learning rate for parameters with smaller gradients and reduces the learning rate for those with large gradients, speeding up learning with less risk of stepping too far.

RMSprop is similar, but keeps a moving average of the squared gradients and divides the gradient by the square root of the mean squares (hence the "RMS"). This evens out the size of the updates for all parameters. RMSprop is famous by being widely used and yet never officially published, having been proposed by Geoff Hinton in a machine learning course. RMSprop is implemented in Tensorflow in the class `tf.keras.optimizers.RMSprop`.

The Adaptive Moment Estimation (Adam) [16] algorithm combines momentum and different learning rates for different parameters using an exponentially decaying average over the previous gradients. Adam seems to perform better than previous optimization algorithms on average, although it can have some convergence problems [26]. In Tensorflow, Adam is implemented in the (`tf.keras.optimizers.Adam`). Adam can be a good first choice to experiment with your models but pay attention to the error values because it may not converge well.

Choosing the best optimizer for a particular problem may require some experimentation.

3.11 Learning Rate

The choice of optimizer and learning rate can have a significant impact not only on training times but also on the performance of your trained network. A low training rate makes training take longer, because the optimization steps are smaller. But if the training rate is too high, the optimizer may fail to converge to the best solution, or even a good solution. The actual values to use depend on the problem and the optimizer, but you can check for this problem also by looking at the error profiles. Figure 3.7 shows the same model trained with a higher and lower training rate. The panel on the left shows that the optimizer is having difficulty converging to a solution, jumping too much over different values. When this happens in the training error it is often a bad sign. Although the training error drops faster in the beginning, with a higher leaning rate, the end result may not be good.

Figure 3.7: Examples of a model trained with high and lower training rates.

3.12 Weight Initialization

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 [9] 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.

Keras offers a choice of initializers, including random uniform and normal distributions, truncated normal distributions, as well as more sophisticated initializers. The default initializer for dense layers, such as those used in multilayer perceptrons, is normalized initialization, also known as the Glorot uniform initializer:

$$W_{i,j} \sim U\left(-\sqrt{\frac{6}{fan_{in}+fan_{out}}}, \sqrt{\frac{6}{fan_{in}+fan_{out}}}\right)$$

In this initialization method, the weight distribution scales in proportion to the number of inputs into the neuron (fan_{in}) and the number of neurons in the next layer receiving the output of this neuron (fan_{out}).

3.13 Batch Normalization

We always normalize (or standardize) the inputs to a neural network, as it is easier for the network to learn by avoiding extreme values and by having the data centred on the origin. With a deep network, we can imagine that each layer is receiving the activations of the previous layer as inputs. It will thus benefit from normalizing these activations. Furthermore, during training the activations keep changing as the weights are updated. This causes the inputs to each layer to shift in mean and variance constantly,

forcing each layer to readjust to these shifts caused by changes in the previous layers. These processes can hinder training.

Batch normalization [13] solves these problems by standardizing the activations at each layer using running statistics for mean and variance collected during training. This not only makes the input more "well behaved" for the next layer but also eliminates the constant changes in mean and variance of the activations during training. Since backpropagation can work through these rescaling operations, these can be inserted into the network as if it was an additional layer, and that is precisely how we can do it with Keras.

3.14 Overfitting

Now that we now which loss function to choose for our problem and how to minimize it, we need to consider the danger of fitting our data so much that our network performs poorly when applied to new examples. This is called overfitting, and basically consists in having an error rate outside the training set that is much larger than the error rate inside the training set. We should select a network architecture (and other aspects of training, such as regularization, which we will see later) that minimizes overfitting.

In order to monitor overfitting, we must measure our network's performance outside the training set. To do this there are two possibilities. One is to split our data set in training and validation sets and use only the former for training the network. This leaves us with one set of data to monitor how the network is classifying examples outside the training set.

A more computationally demanding approach, but giving better results, is to use cross-validation. To do cross-validation, we partition our training data into k disjoint *folds*. For example, if we have 50.000 examples and want to use 5-fold cross-validation, we place 10.000 examples into each fold. Then we train our model with all folds but one and validate on the fold that was left out. Then we repeat training leaving another fold out, and do this k times, averaging the validation error. This gives us a better estimate of the true error that, on average, instances of our model will have when trained on such data. Figure 3.8 shows an example of 5-fold cross validation using the gene expression data with a linear classifier. Each panel shows an hypothesis obtained by fitting the model to four of the folds (indicated by the smaller points) and then validating using the fold left out.

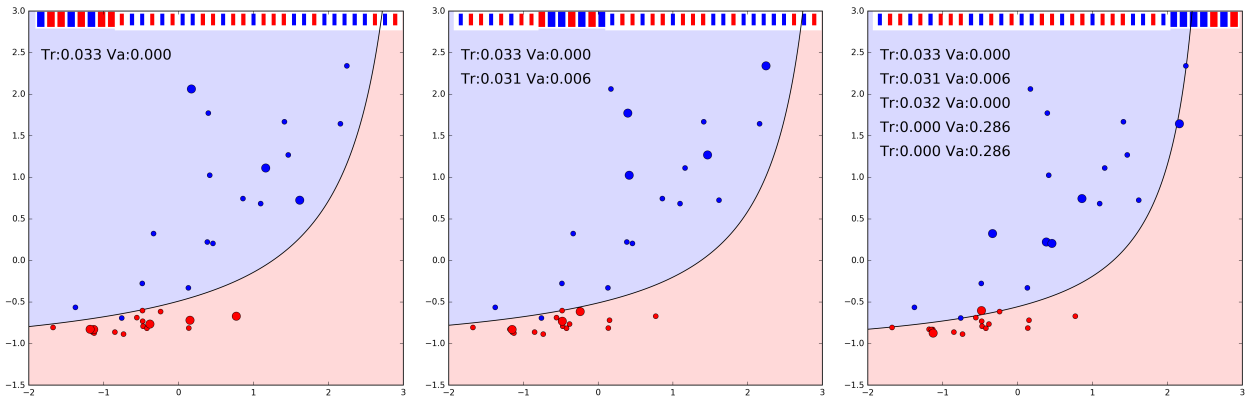


Figure 3.8: Example of 5-fold cross validation, showing the plots for folds 1, 2 and 5. In each panel, one of the folds is left out for validation, the other folds are used for training. The larger points are those used for validation in each fold. The training and validation errors are kept for each fold and then averaged in the end.

Cross validation gives us a better estimate of the true error (the average error over all possible data)

Figure 3.9: Examples of a model that is overfitting (left) and one that is not (right).

than simple training and validation because we are averaging k estimates of the true error. However, this requires training the network k times. This is why cross validation, although widely used in classic machine learning, is often discarded in favour of simple training and validation in deep learning, where models take much longer to train. In any case, it is important to evaluate how our networks perform outside the training data, otherwise we are likely to overfit the data and get a much poorer performance than the training results suggest.

Finally, it may be useful to have an unbiased estimate of the true error of our final network, so we can judge how it will perform. This should not be done with validation or cross validation errors if we have been fine-tuning the network and other hyperparameters, as is generally the case. The reason for this is that, while selecting hyperparameters, we choose those that minimize the validation (or cross validation) error, causing the error value obtained in the end to be biased towards underestimating the true error. The solution for this problem is to leave out a subset of our initial data, the test set, that was never used to choose any model or parameter and can thus serve to obtain an unbiased estimate of the true error of the final network.

In machine learning, we want to use the data available to train a model so that the hypothesis (the trained instance) we obtain can be applied to new data. This is especially true for supervised learning, although sometimes in unsupervised learning we just want to make sense of the data that we have. In these cases, we must consider the performance of our networks outside the training data, as we saw previously.

3.15 Model Selection

If our model is able to learn irrelevant details about the training set, that do not generalize, then the resulting hypothesis may perform poorly on new data. This is called *overfitting* and we can check for this effect using cross-validation or with a validation set, as we saw in the previous chapter. This way we can select the best model to use for our problem. Figure 3.9 illustrates how we can easily notice this problem when plotting the training and validation errors. Even though the model on the left seems better at adjusting to the training set, it fares poorly on the validation set, and thus it is best to use the one on the right.

3.16 Bias and Variance

When considering how to best solve a machine learning problem, it is good to separate two sources of error, at least conceptually, to better understand how we can optimize the results. Statistically, the

bias is the difference between the expected value of an estimator and the true value being estimated. Thus, the *bias* of a model at some point is the difference between the true value we are trying to predict and what, on average, the model predicts at that point after training with some set of data. Note that this average is not only for one specific training set but for all possible training sets. If we had many training sets drawn from the same universe of possible data, the *bias* for the model in one point could be estimated as the quadratic error between the true value and the average of what our model predicts for that value. Averaging this for all possible values would give us the expected bias of the model:

$$bias_n = (\bar{y}(x_n) - t_n)^2 \qquad bias = \frac{1}{N}\sum_{n=1}^{N}(\bar{y}(x_n) - t_n)^2$$

Figure 3.10 shows an example of a model that cannot adequately fit the data. The estimates for the point marked as a large blue circle are all tendentiously above the true value and thus there is a difference between the average and the true value.

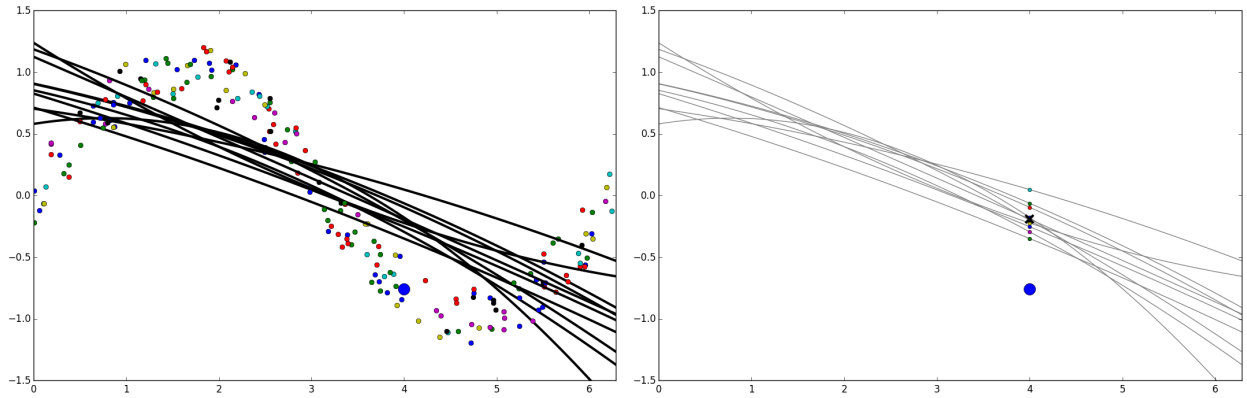


Figure 3.10: This model cannot adjust to the data and thus has a large *bias* in some points.

Variance is a measure of the dispersion of values. Applying this concept to a regression model, the *variance* of the model at some point is the expected variance of the predicted values for that point when the model is trained over any data set. The *variance* for the model is the average of the variances for all points. To estimate the variance of a point and on N points of a model trained on M data sets, we compute:

$$\frac{1}{M}\sum(\bar{y}(x_n) - y_m(x_n)) \qquad var = \frac{1}{NM}\sum_{n=1}^{N}\sum_{m=1}^{M}(\bar{y}(x_n) - y_m(x_n))^2$$

where $\bar{y}(x_n)$ is the average of the predictions for point x_n. Figure 3.11 shows a model that overfits the data, which results in a large *variance*, showing that, for the point marked as a large circle, the predictions of individual hypotheses are spread in a broad range around their average.

These are the two fundamental sources of error that we have control over. There is also an additional possible source of error, which is that associated with our data, but there is nothing we can do about that. With respect to our models, we can consider that the error we get outside the training set (the true error, over all possible data) will be due to a combination of the error due to the model not being able to perfectly adapt to the shape of the data, which is the *bias*, and the model varying its predictions too much as a function of details in the training set, which is the *variance*.

This distinction is important because bias can only be mitigated if we use a better model or if we combine simpler models so that we get a greater power to adjust to the data. Variance, however, can be

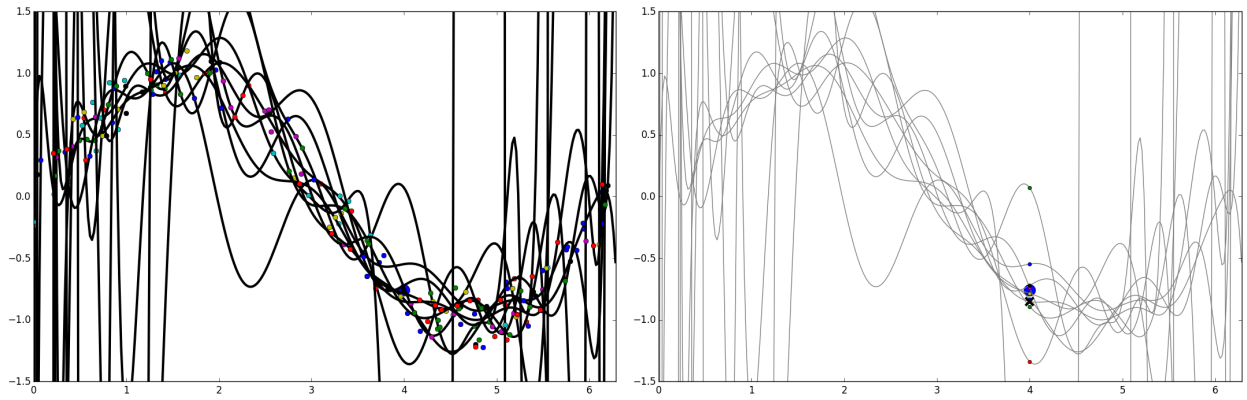


Figure 3.11: This model overfits the data and thus has a large *variance* in some points.

mitigated either by opting for simpler models, which will generally increase bias, or by changing the way the model is trained in order to prevent it from varying so much in its predictions. Thus, it is often the case that we deliberately choose a model in overfitting and then mitigate that problem by reducing its variance with regularization.

3.17 Regularization

Regularization is any way of changing the training of the model in order to decrease its variance. This may increase bias, but if the model is overfitting and the bias increase is modest, regularization can greatly improve results. There are several ways of regularizing neural networks.

Parameter Norm Penalties

Penalizing the norm of the parameters vector is a traditional regularization method in statistics and machine learning. In brief, we can imagine that the set of parameters in our model represents a vector and we want our training algorithm to try not only to reduce the loss function but also some measure of the "length" of this vector. Thus we change our loss function to also include a penalty term as a function of the parameters themselves:

$$\tilde{J}(\theta; X, y) = J(\theta; X, y) + \alpha\Omega(\theta)$$

where X, y is our data, θ the parameters and $J(\theta; X, y)$ our original loss function. α is a regularization parameter that we can adjust to give greater or lesser weight to the regularization.

With L^2 regularization, also known as *ridge regression* or *Tikhonov regularization*, the penalty function is the square of the modulus of the parameter vector, $||\theta||^2$. This forces the overall magnitude of the parameters to go closer to zero. In neural networks, this can also be implemented in the parameter update rule as *weight decay*: whenever a weight on a neuron is updated, a fraction of its previous weight is subtracted:

$$w_{i+1} = w_i - \eta\frac{dJ}{dw} - 2\alpha w_i$$

where w is some weight, η the learning rate, J the loss function and α the regularization weight. Note the factor of 2 multiplying α, which results from computing the derivative of the quadratic penalty function applied to the weight.

Another used norm penalty function is L^1 regularization, which penalizes the sum of the absolute values of the parameters, $\sum_i |\theta_i|$. While L^2 regularization drives the whole vector θ to be smaller, L^1 regularization makes some weight fall to zero, resulting in a sparser solution, where weights are completely "turned off"

Although in general, L^2 and L^1 regularization apply the penalty to all parameters in the model, in neural networks these penalties are usually applied only to the neuron weights and not the bias values. This is because the bias values, unlike the weights, do not contribute to the sensitivity of the model to slight variations in the data.

Dataset Augmentation, Noise and Semi-supervised Learning

We can always reduce overfitting by using more data for training. Although data is generally not easy to come by, sometimes we can augment our data set with some simple operations. For example, with images we can rotate or mirror images in our dataset to create variants that are different but retain the same labels, if appropriate. But some care must be taken, depending on the data itself, for while a mirror image of a cat is still a cat, the mirror image of `b` is `d`.

Another regularization method involves injecting random noise in the features whenever an example is presented to the neural network. This can be considered a form of data augmentation, providing the network with a greater diversity of examples which can be useful as long as the noise injected is not excessive.

Noise can also be applied to the weights of the network, which is equivalent to penalizing the gradient of the error and leading the optimizer to find sets of weights in more stable regions of the loss function.

In classification problems, *label smoothing* may be useful. This involves making the target labels, such as in a one-hot encoding or binary classification, not hard values of 0 and 1 probability but softer values close to zero and one. For example, $\frac{\epsilon}{k-1}$ instead of zero and $1-\epsilon$ instead of one, with a small ϵ. This prevents the optimizer from trying to drive the sigmoid or softmax functions to exactly zero and one, which is impossible, and is equivalent to adding some noise to the class labels, with a small probability of an example being presented as belonging to the wrong class.

Semi-supervised learning uses unlabelled data to enrich the data set used for supervised learning. One way of doing this is to use an unsupervised learning algorithm to cluster all data, both labelled and unlabelled, and assume that unlabelled examples close to labelled examples belong to the same class. Other approaches involve combining labelled and unlabelled examples to learn the probability distributions of features, which can then inform the training of the classifiers on the labelled data.

Early Stopping

Another way of regularizing neural networks is to stop training to prevent overfitting from getting worse. As the optimizer moves the parameter vector away from the initial values (close to 0) it can improve the fit in the training set while degrading performance outside this set. Early stopping keeps the optimization closer to the initial values, thus restricting this walk into more extreme values that worsen overfitting. Figure 3.12 illustrates this. Even though the training error continues to fall, the validation error starts to rise after about 15 thousand epochs, showing that overfitting is becoming worse and the training should be stopped at that point.

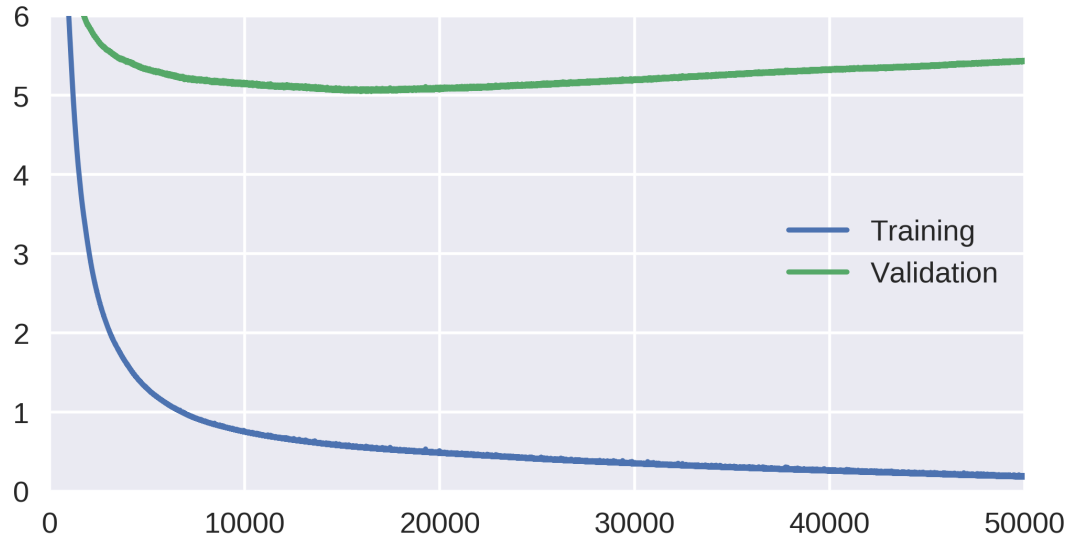


Figure 3.12: This model overfits the data and the problem gets worse after about 15 thousand epochs.

Ensembles and Dropout

One commonly used method for reducing variance in machine learning is to group together several instances of the model, or even several models, trained in different subsets of the training data. *Bagging* is an example of this, using randomly drawn subsets of the training data to train the model into different hypotheses and then classifying new data with the average of all hipotheses.

However, this is not practical for deep neural networks because of how long it takes to train them or to use an ensemble of networks to classify new examples. *Dropout* [37] is a better way of doing this with neural networks. During training, at each example or mini-batch, neurons are "dropped" randomly by ignoring their weights and activation. Typically, the probability of doing this for any hidden neuron at any iteration is 0.5, or 0.2 for input neurons (output neurons are not dropped). Thus, we are training an ensemble of different networks given by the combination of neurons that are "alive" at each iteration. Figure 3.13 shows two iterations of the same network with different neurons dropped off, at random.

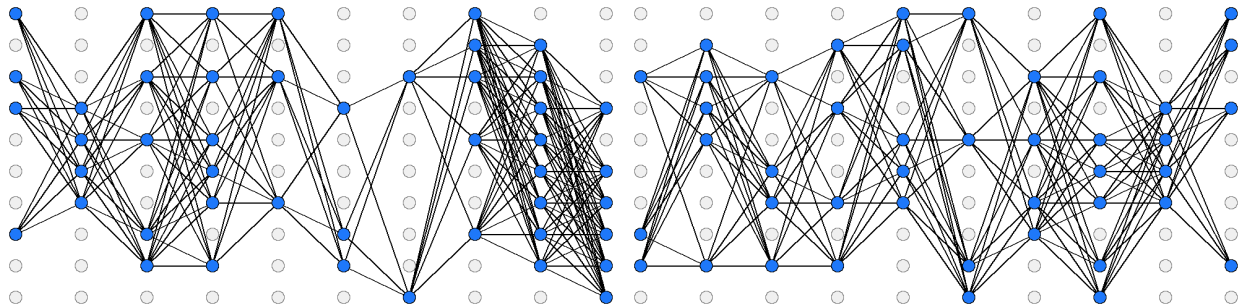


Figure 3.13: Dropout: neurons are ignored at random at different iterations (examples or mini-batches.

After training, the activations of neurons (or weights in the forward neurons) can be rescaled by the dropout probability to retain the same expected activation. Alternatively, as is implemented in Tensorflow, we can use *inverse dropout* during training. In this case, neurons that are not dropped will have their activations multiplied by $1/p$ so that expected activations will then be the same on test.

3.18 Selecting Hyperparameters

With so many options to choose from, in network architecture, optimizers, regularization and so forth, selecting the right combination of hyperparameters can be difficult. One option is to try different combinations manually, try to understand what effects they have and gradually narrow down a good combination.

Alternatively, it is possible to automate the search for the best hyperparameters. A *grid search* systematically runs through combinations of parameters within a given range and with a given step. Given the large number of possibilities, making it unfeasible to systematically run through all combinations, a *random search* may be best. Simply pick hyperparameter values at random and repeat, keeping the best combination. Finally, hyperparameter choice can be seen as an optimization problem. One approach is to use a Bayesian model to estimate the validation error for each combination based on combinations tested so far and use that to decide what new combinations to test to improve results.

3.19 Further Reading

1. Goodfellow et. al., Chapters 5, 6, 7 and 11, and sections 8.4, 8.5 and 8.7.1 [9]
2. Tensorflow, activation functions and linked relevant papers: https://www.tensorflow.org/api_guides/python/nn#Activation_Functions

Chapter 4

Convolutional Networks

Convolution. Convolution layers and networks. Pooling. Classification with convolutional networks.

4.1 Convolution

Mathematically, the convolution of two functions f and g at t, represented $(f * g)(t)$, is the integral of the product of the two functions with one being reversed and then shifted by a parameter t. Convolution is commutative:

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau = \int_{-\infty}^{\infty} f(t-\tau)g(\tau)d\tau$$

This seems a strange operation but it crops up in many situations. For example, the probability density function of the sum of two continuous random variables is equal to the convolution of the probability density functions of the two variables. More easy to understand is the modelling of linear time invariant systems. These are systems that respond to inputs with some response function such that the response is linear with respect to the inputs and depends only on the timing of the inputs. Let's consider a spring with a dampener, responding to an impulse with a function $g(t)$ in the form of a decaying sine wave, for example. We hit the spring with a function $f(t)$ and want to know how the fork reacts.

In the context of this course, we refer to the first argument as the input and the second argument as the kernel. The output of this convolution is refered to as the feature map, since it represents the new features obtained by applying the kernel to the input.

Since we will consider only discrete data points in t, we can rewrite the previous formula as a discrete convolution:

$$(f * g)(t) = \sum_{\tau=-\infty}^{\infty} f(\tau)g(t-\tau) = \sum_{\tau=-\infty}^{\infty} f(t-\tau)g(\tau)$$

In practice, for CNN, we consider that the values outside of the finite set of points we have in our input have value 0. The parameters of the kernel will be learned during training, as these are parameters of the model, and the convolution operation is differentiable.

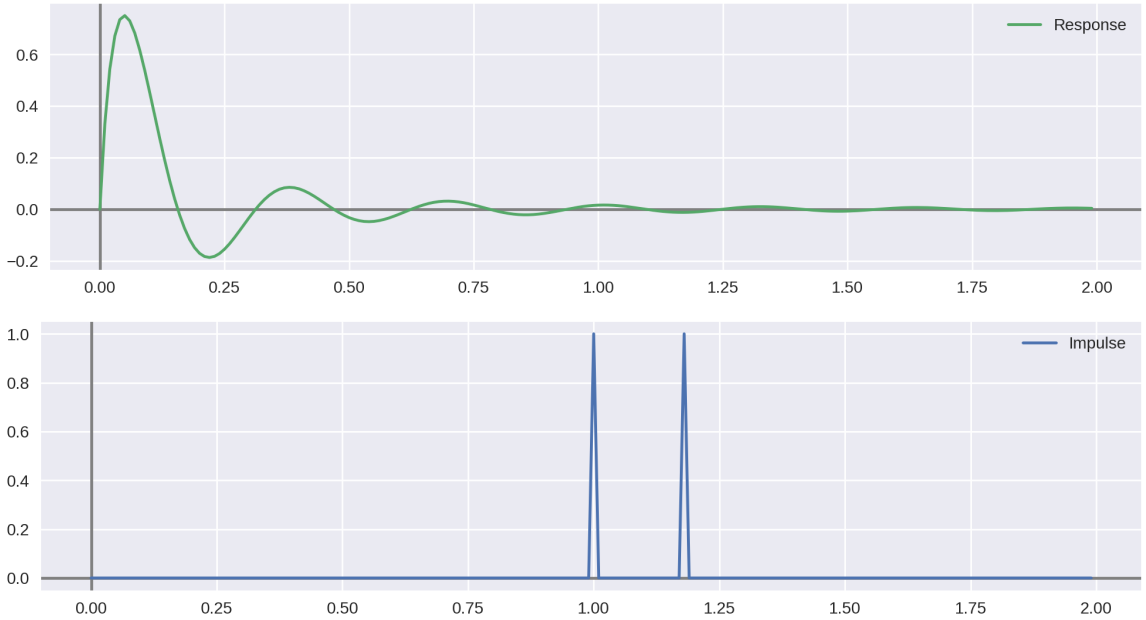


Figure 4.1: Response (top) and impulse (bottom) functions for an hypothetical dampened spring.

We often consider 2D convolutions, across two dimensions:

$$S(i,j) = (I*K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n) = (I*K)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n)$$

4.2 Convolution Networks

4.3 Pooling

4.4 Convolution Networks with the Keras Sequential API in Tensorflow

To build a model using the Keras Sequential API, you just need to create a `Sequential` object an then add the different layers. We will use the Fashion MNIST exercise (below) as an example. First, we import the necessary classes:

```
from tensorflow.keras.optimizers import SGD
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import BatchNormalization,Conv2D,MaxPooling2D
from tensorflow.keras.layers import Activation, Flatten, Dropout, Dense
```

Now we create the model. To illustrate, we will add a 2D convolution layer with a 3×3 kernel and 32 filters as the first layer. The input shape corresponds to the image shapes in the Fashion MNIST dataset. Then will add a ReLU activation and a batch normalization layer. The first layer needs the input shape specified as it will receive the data. Note that the batch size is omitted and the shape corresponds to a single example. The shape of the subsequent layers is determined automatically.

```
model = Sequential()
model.add(Conv2D(32, (3, 3), padding="same", input_shape=(28,28,1)))
```

```
model.add(Activation("relu"))
```

To add pooling, dropout and dense layers, we just use the appropriate classes. The parameters are mostly self-evident, but for more details please look up the documentation online. Note the `Flatten` layer before the dense layers. This is needed to flatten the input into a single dimension for the features, which is what the dense layer expects.

```
...
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(64, (3, 3), padding="same"))
model.add(Flatten())
model.add(Dense(512))
model.add(Activation("relu"))
model.add(BatchNormalization())
model.add(Dropout(0.5))
model.add(Dense(10))
model.add(Activation("softmax"))
```

To compile the model we need to select an optimizer and specify the loss function and metrics to register during the run.

```
opt = SGD(lr=INIT_LR, momentum=0.9, decay=INIT_LR / NUM_EPOCHS)
model = create_model()
model.compile(loss="categorical_crossentropy", optimizer=opt,
    metrics=["accuracy"])
```

Now we can train the model, calling the `fit` method to which we supply the training features and labels, as well as the validation data if we wish to record validation metrics during training. We can also set the batch size and epochs. The `fit` method returns a `history` object with a dictionary, also called `history`, with the record of all metrics collected during training.

```
history = model.fit(trainX, trainY, validation_data=(testX, testY),
                    batch_size=BS, epochs=NUM_EPOCHS)
```

After training we can save our model as a JSON file and the weights in a hierarchical data format (HDF5) file:

```
model.save_weights('fashion_model.weights.h5')
model_json = model.to_json()
with open("fashion_model.json", "w") as json_file:
    json_file.write(model_json)
```

4.5 Exercise: Exploring Hyperparameters

In this example, we will create a regression multilayer perceptron to predict the fuel consumption of cars using a modified version the Auto MPG Data Set, available at the UCI repository [5]. You'll find the data set in the class materials, and it was modified by removing a text column and some rows with missing values. You can read the file provided using `loadtxt` from the Numpy library but skipping the first row (the headers row).

First, we must preprocess the data adequately. We will shuffle the data before splitting into training and validation sets and use the validation set to monitor for overfitting.

The first column, MPG (miles per gallon), is what we want to predict, so separate this as an array of target values. But first standardize the data[1]. In classification problems we only standardize the features, since the class labels are discrete. But for regression, it is better to standardize everything to prevent very large values resulting in convergence problems, and also to centre the output of the model on 0. Remember to keep the rescaling parameters not only because they need to be applied to the features of future examples but also because we need to rescale the predictions again to get meaningful results if we apply our trained model.

This dataset has only 392 examples. We will use 300 for training and 92 for validation.

```
data = np.loadtxt('AutoMPG.tsv',skiprows=1)
np.random.shuffle(data)
means = np.mean(data,axis=0)
stds = np.std(data,axis=0)
data = (data-means)/stds

valid_Y = data[300:,0]
valid_X = data[300:,1:]

Y = data[:300,0]
X = data[:300,1:]
```

Since this is a regression problem, we will use a linear output for the last neuron (it should be only one neuron), since we do not want to constrain the output with an activation function, and we will use mean squared error loss function for training. Also, to speed up training, we will use leaky ReLU activations in the hidden layers.

Experiment with different architectures for this problem using the Keras API as we saw previously. See https://keras.io/losses/ to find the appropriate loss function for this model. Explore different hyperparameters (network, optimizer, learning rate and so on) to and try improve results on the validation set.

4.6 Exercise: Fashion MNIST with a convolution network.

In this exercise you will create a CNN to classify the Fashion MNIST data set[2].

First, we need to load the data set and format it adequately. The loader will check if the dataset is already available locally and download it if not. Then we will reshape the features to $28 \times 28 \times 1$ to fit our convolution layers and convert the image data from integer to float.

For the class labels we will need one-hot encoding. We can convert them with the `to_categorical` function in `keras.utils`.

```
1 from tensorflow import keras
2
3 ((trainX, trainY), (testX, testY)) = keras.datasets.fashion_mnist.load_data()
4 trainX = trainX.reshape((trainX.shape[0], 28, 28, 1))
```

[1] Standardization is subtracting the mean and dividing by the standard deviation, so that all feature distributions have mean of 0 and standard deviation of 1

[2] Based on a PyImageSearch tutorial by Adrian Rosenbock, https://www.pyimagesearch.com/2019/02/11/fashion-mnist-with-keras-and-deep-learning/

```
testX = testX.reshape((testX.shape[0], 28, 28, 1))

trainX = trainX.astype("float32") / 255.0
testX = testX.astype("float32") / 255.0

# one-hot encode the training and testing labels
trainY = keras.utils.to_categorical(trainY, 10)
testY = keras.utils.to_categorical(testY, 10)
```

Now create this model with the sequential API provided by Keras:

- Two convolution layers with a 3×3 kernel, "same" padding, 32 filters, ReLU activation and batch normalization. Note that batch normalization uses the last axis as default direction to normalize. If the channels are at the end, this is correct.
- Max pooling of size 2×2 and the same stride (note that omitting the stride will use the pool size for stride too). Optionally, you can try a dropout layer with 25% dropout probability, but dropout in convolution layers is generally not very useful.
- Two more convolution layers like the ones before, followed by the same pooling, but use 64 filters in the convolution layers.
- Finally, a dense layer of 512 neurons also with ReLU activation, batch normalization and dropout of 50% followed by a softmax layer with 10 neurons.

Use the optimizer shown before. Train the model for about 25 epochs (or fewer, if this is too slow) and then save the weights. We will be using these later on for transfer learning.

4.7 Further Reading

1. Goodfellow et. al., Chapter (9) [9]

Chapter 5

Autoencoders

Autoencoders. Undercomplete autoencoders. Regularization in autoencoders.

5.1 Autoencoders

An autoencoder is an artificial neural network that is trained to recreate the input in the output. In other words, it is trained to approximate a function F so that $F(x) = x$. The motivation for this is to obtain from the autoencoder some useful representation in the hidden layers. Figure 5.1 illustrates an undercomplete autoencoder trained to recreate cat images. The designation of undercomplete refers to the smaller size of the hidden layer at the middle of the autoencoder.

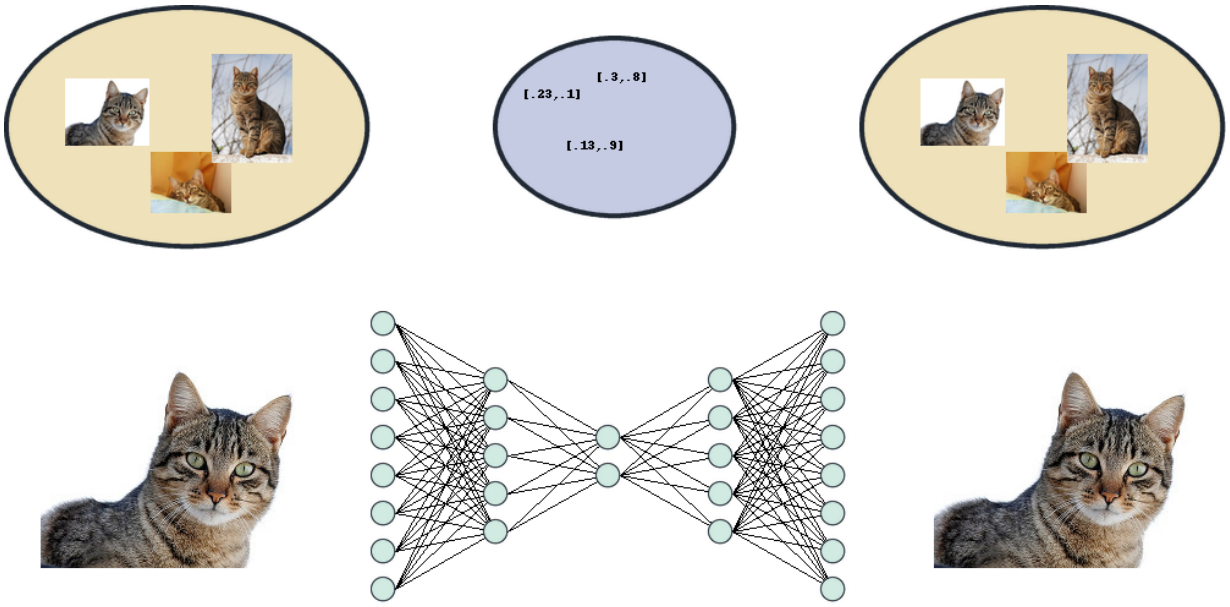


Figure 5.1: Representation of an undercomplete autoencoder. Cat images by Joaquim Alves Gaspar CC-SA.

We can consider that the first half of the autoencoder, the encoder part, will transform the input into a representation at the middle of the autoencoder. Then the second half, the decoder, will convert this representation into the input vector again. In other words, we are training the network to learn two functions, f and g, corresponding to the encoder (f) and decoder (g) such that $g(f(x)) = x$. If we

constrain the encoder adequately, the encoding function $h = f(x)$ may give us a useful representation of x. And an advantage of the autoencoder architecture is that we do not need any additional labels. This means that, even though the network is trained by minimizing a loss function with backpropagation, as usual, in practice this works as unsupervised learning in that we do not need labelled data to train the autoencoder.

The requirement is simply that the network be constrained, by its architecture or by regularization, so that the encoder needs to transform the input x into a useful latent representation h. One simple way of doing this is to force the dimension of h to be smaller than the dimension of x, which is the case of an undercomplete autoencoder. But there are other options, as we will see below. But first we can take a look at one intuitive application of autoencoders: manifold learning.

Manifold Learning

Mathematically, an n-dimensional manifold, or n-manifold, is a set of points such that each point and its neighbours form approximately an Euclidean space. For example, seismic events at the surface of the Earth form a two-dimensional surface in three-dimensional space. Figure 5.2 shows two examples of manifolds.

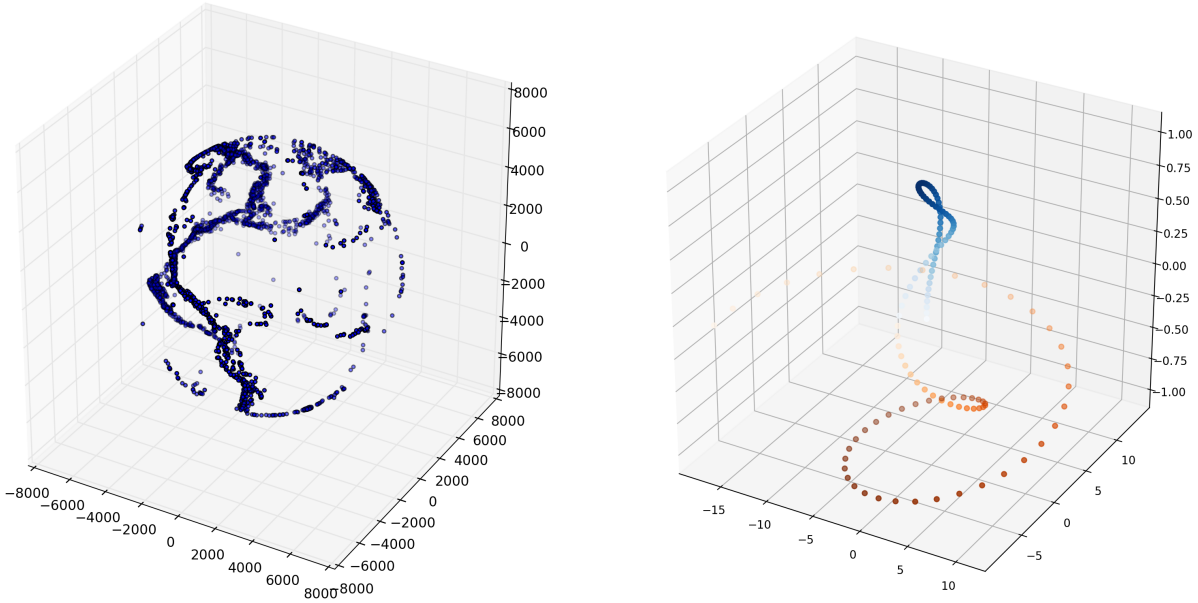


Figure 5.2: Examples of two manifolds in three-dimensional space: the 2-dimensional manifold of seismic events on the surface of the Earth and a 1-dimensional manifold of hypothetical data that follows a line in 3D.

If our data follows a lower-dimensional manifold, then it is possible to compress it into a lower-dimensional representation without losing much information. In other words, we can train an undercomplete autoencoder to find a representation h with a dimension lower than x while retaining the ability to reconstruct the data outside the training set. This last point is important: overfitting the autoencoder to the training data can give good reconstructions but the latent representation may become useless.

5.2 Regularized Autoencoders

We do not need to reduce the dimension of the latent representation in order to force the autoencoder to give us a useful representation. In theory, we could even expand the input data into a larger dimensional space. This would be an overcomplete autoencoder, as illustrated in Figure 5.3. As long as we restrict the ability of the autoencoder to prevent it from simply copying the input to the output, we can force it to learn a useful representation. And we can do this with regularization, by changing the training of the autoencoder. In particular, by changing the loss function.

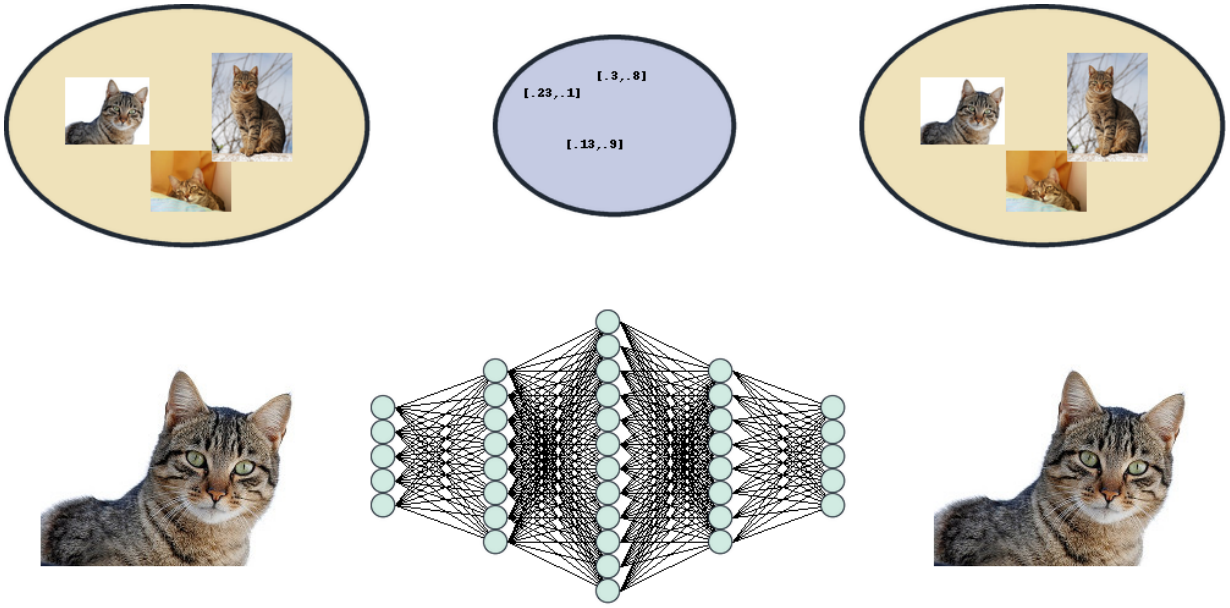


Figure 5.3: Representation of an overcomplete autoencoder. Cat images by Joaquim Alves Gaspar CC-SA.

5.3 Sparse Autoencoders

In a sparse autoencoder, we want the neurons in the latent representation layer to be mostly inactive for each example. We can specify this by choosing a target probability for the average activation of each neuron in the middle hidden layer of the autoencoder. This can be estimated by the average activation over a set of examples:

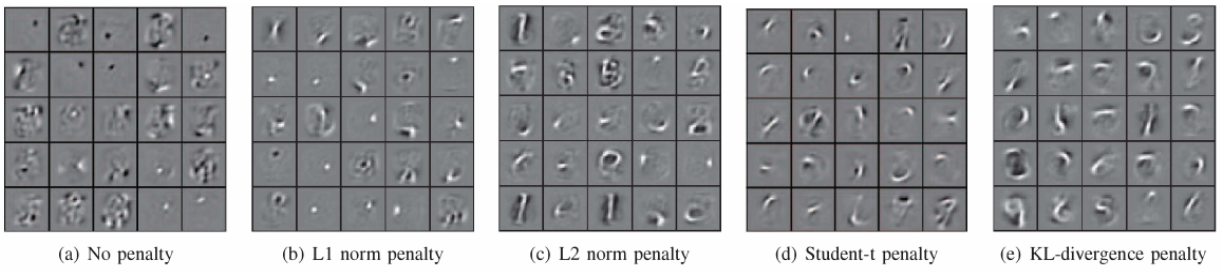
$$\hat{p}_i = \frac{1}{m} \sum_{j=1}^{m} h_i(x_j)$$

If we want the probability of h_i firing to be p_i, we can include in the loss function a penalization term corresponding to the Kullback-Leibler divergence between the activations of this neuron and a binomial distribution with probability p_i:

$$L(x, g(f(x))) + \lambda \sum_i \left(p \log \frac{p}{\hat{p}_i} + (1-p) \log \frac{1-p}{1-\hat{p}_i} \right)$$

By forcing only a few neurons to fire for each example, using a small activation probability (*e.g.* 0.05) these neurons will specialize in identifying particular features that distinguish the examples.

Other regularization options for sparse autoencoders include L1 regularization applied to the activations instead of the weights, L2 regularization, and others. Jiang and others show a comparison of difference sparseness penalties using the MNIST dataset [15]. Some results are illustrated in Figure 5.4.



(a) No penalty (b) L1 norm penalty (c) L2 norm penalty (d) Student-t penalty (e) KL-divergence penalty

Figure 5.4: Normalized input maps that maximize activation of latent neurons in sparse autoencoders, using different sparseness penalties. Image from [15].

5.4 Denoising Autoencoders

We saw previously that noise injection can be used for regularization in neural networks, since it forces the network to rely less on individual input values and more on larger patterns. This also works for autoencoders. In a denoising autoencoder, the network is trained to reconstruct the original x from a corrupted version $\tilde{x}$, obtained by adding some noise to x, typically by randomly setting values to zero. Figure 5.5 illustrates this.

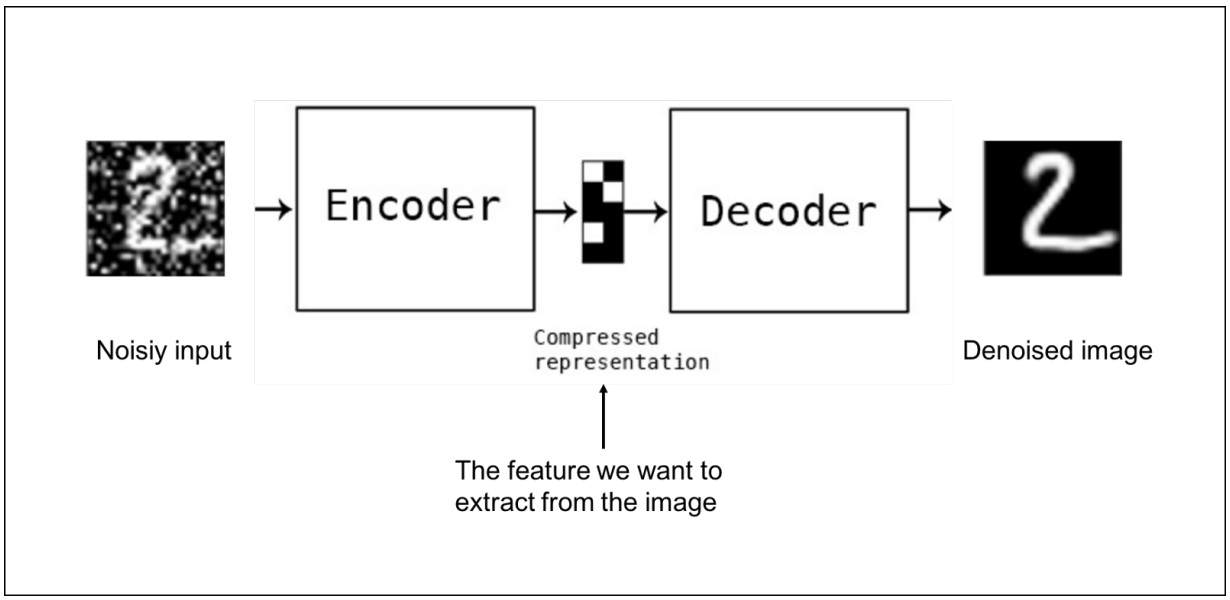


Figure 5.5: Normalized input maps that maximize activation of latent neurons in sparse autoencoders, using different sparseness penalties. Image by Adil Baaj.

5.5 Contractive Autoencoders

Rifai and others proposed adding to the loss function a regularization term that reduces the autoencoder's sensitivity to small variations in the input [28], in the form of a penalization proportional to the square of the gradients of the activations in the hidden layer with respect to the inputs:

$$L(x, g(f(x))) + \lambda \sum_i ||\nabla_x h_i||^2$$

This makes the encoding more robust and groups similar examples closer together in the manifold of the latent representation h (hence the name of contractive autoencoder). Nevertheless, the autoencoder also needs to reduce the reconstruction loss for the input, so it must distinguish between more or less penalized directions when moving from each point. We can visualize these directions and compare the result of a contractive autoencoder and other dimensionality reduction techniques, like PCA. This is what Rifa *et. al.* did for the manifold tangent classifier [28], based on stacking contractive autoencoders, as shown in Figure 5.6

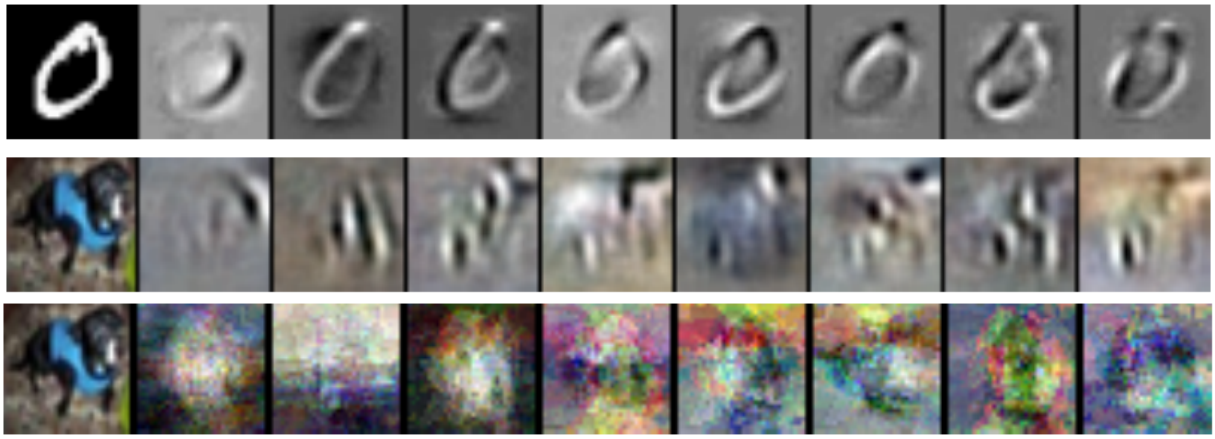


Figure 5.6: Principal singular values of the Jacobian matrix, showing tangents of each point in a contractive autoencoder, and compared to PCA. From [28].

The effect is similar to that of a denoising autoencoder, and a contractive autoencoder can be more expensive to train due to the need to compute the gradients with respect to the inputs.

5.6 Stochastic Autoencoders

5.7 Further Reading

1. Goodfellow et. al., Chapter 14 [9]

Chapter 6

Representation Learning

Learning adequate features. Unsupervised pretraining.

6.1 Learning Features

Representation learning, or feature learning, is the automated extraction of useful features from data. The motivation for this is that having adequate features may make problems much easier to solve. For example, representing numbers with Arabic numerals makes algebraic operations much easier to perform than using Roman numerals. Think of the difference between adding 58 and 12 or adding LVIII and XII.

In classical machine learning, features are usually extracted by human intervention. This not only requires domain knowledge but the design of procedures for converting the original data in a more useful representation. But with deep neural networks we can rely on the model finding good representations of the data for whatever target we select. There are several reasons for assuming this to be the case, besides empirical confirmation that it is so in several fields. Here are some examples from Bengio and others [1]:

Manifolds Real data is not uniformly distributed over the whole space of the original features. Rather, it usually spans a lower dimensional manifold in which we can project it without losing much information while simplifying the representation. In addition, after projecting into the manifold it is likely that any function of the data we may want to learn becomes smoother.

Smoothness In simple terms, this is a property of a function that has a similar outputs for similar inputs. In general, when we use data with a large number of dimensions, this property does not hold for functions we may want to learn (for classification or regression, for example). However, if we obtain a better representation of the features, these functions tend to be smoother.

Disentanglement Useful features tend to be able to vary independently, because relations between features lead to redundancy and may make learning harder. A good representation will disentangle such features so that the features that are extracted are more independent.

Hierarchy Useful features are generally organizable in hierarchies, with different levels of representations. For example, in image processing lower level features may include edges of different

orientations, contrast and so on, while higher level features may be patterns such as eyes or wheels. Deep models are good for learning these hierarchical representations, and such hierarchies can help reuse parts of these models to extract features at relevant levels in different problems. We saw this previously in the transfer learning example by using the convolutional layers of a pretrained network, which is related to the shared factors aspect we cover next.

Shared factors A related aspect of adequate features is that they can be useful for different problems and across similar datasets. This means not only that adequate features can be reused in other problems but also that we can learn these features in different datasets and then use the ability to extract them in the problem we are interested in solving. This is why transfer learning is often useful.

Sparsity When considering useful features, for most examples only a few features will be useful. We can have a network that detects teeth, eyes, wings, feathers, fur and so on and for each example only some of these features will be relevant. This means that the representation will be less sensitive to small changes in the input vector.

Semi-supervised learning In general, the same features that can help us model the probability distribution of the set of input vectors, $P(X)$, can also be useful to model the conditional probability of some target given these vectors, $P(Y|X)$. This means that we can use unlabelled data to help us extract useful features for some supervised learning task, and since unlabelled data is generally more abundant this can be very helpful.

6.2 Unsupervised Pretraining

Autoencoders, covered in the Chapter 5, are one type of model that can help us extract features from unlabelled data. We train a neural network to output the same vector as the input but with some constraint to force a hidden layer to extract useful features. This can be used for a greedy pretraining of individual layers and was an important technique before the use of ReLU activations mitigated the vanishing gradient problem.

We can start with an autoencoder with a single hidden layer and train it to reconstruct the inputs after adding noise, for example. This is the denoising autoencoder we saw in 5.4. If after training we discard the decoder part of the autoencoder and retain only the encoder, we now have a layer capable of creating this representation of the data. We can use this transformed data to train a new denoising autoencoder and repeat the process. This is called a stacked denoising autoencoder and can be used to pretrain the network with unsupervised learning before fine tuning the complete network with labeled data.

This pretraining technique is now rarely used, since modern activation functions and optimizers allow us to train deep networks all at once, which is more practical, but as Erhan and others showed in 2010, this pretraining acts as a form of regularization by compressing the training trajectories in a specific region of the parameter space [7], as shown in Figure 6.1. This figure represents training trajectories of two groups of 50 instances of the same model, one with pretraining and the other without, projected using t-SNE and ISOMAP. We can see that the different groups are spread in different regions of the parameter space and, using the ISOMAP projection that preserves distances at a larger scale, we can see that the pretrained networks span a much smaller region than with random initialization.

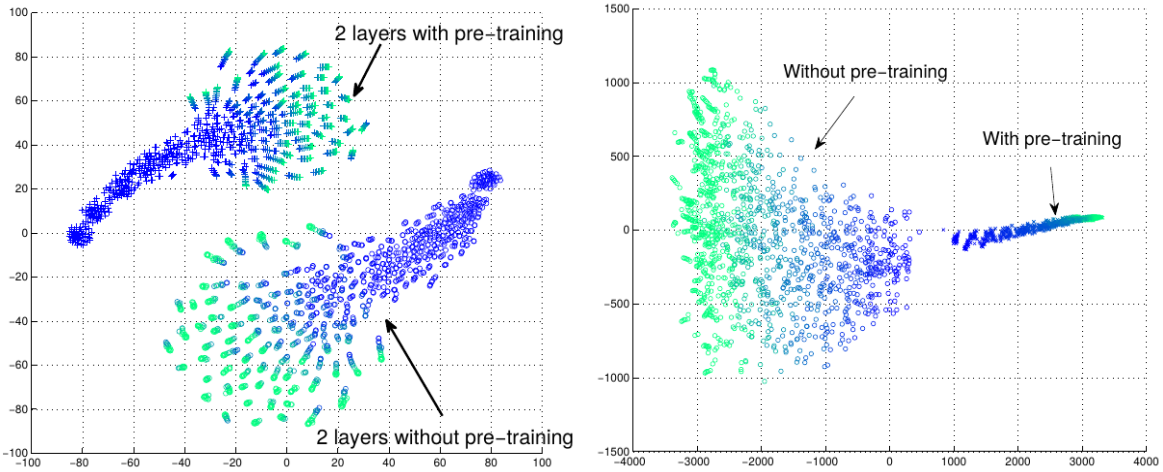


Figure 6.1: Projection with t-SNE[21] and ISOMAP[40], from Erhan et. al [7].

Although pretraining with stacked denoising autoencoders is now seldom used, unsupervised pretraining of layers to obtain desirable features is relevant in applications where the input data is highly dimensional, very sparse or has other undesireable attributes and when unlabelled data is abundant. For example word embedding in natural language processing.

One example of this is feature extraction from very sparse and high-dimensional data from motion sensors to classify human movements [23]. The original data has a dimension of 5000 but for each example only about 2% of the features are not zero. Using a denoising autoencoder followed by a PCA projection for visualization different groups of motions are apparent, as shown in Figure 6.2

6.3 Transfer learning

Good representations for a particular problem do not need to be learned specifically for that problem and dataset. We can use the same transformations to extract useful features in different problems or different data. This is called transfer learning. There are different situations where reusing the same transformations or starting from previously trained networks can be of benefit to solving some problem.

In the case of *domain adaptation* we model the same mapping from input to extracted features or output but with a different data set. For example, we trained a model to classify the sentiment of movie reviews but now want to classify the sentiment of reviews of items sold on an electronics store. Although the review texts are different, it is likely that the relation between the words and whether the person was happy or angry is similar.

In the case of *concept drift* the problem arises because either the mapping from input to output is gradually changing or the data distribution is changing. In either case, this change is gradual so we may not need to retrain our model from zero and it may suffice to just refresh the model regularly by training with small sets of recent data starting from the parameters of the previous instance of the model.

Zero-shot learning is an extreme case of transfer learning where we can use exactly the same instance to new data outside the distribution of the data used to train the model. Socher and others [36] give an example of this in cross-modal learning, where the models relate images and text. After learning the manifolds of images and words, obtaining representations of each modality of examples using unsupervised learning, and learning a supervised mapping between the image manifold and the word manifold. This means that even if new images do not correspond to any of the classes in the

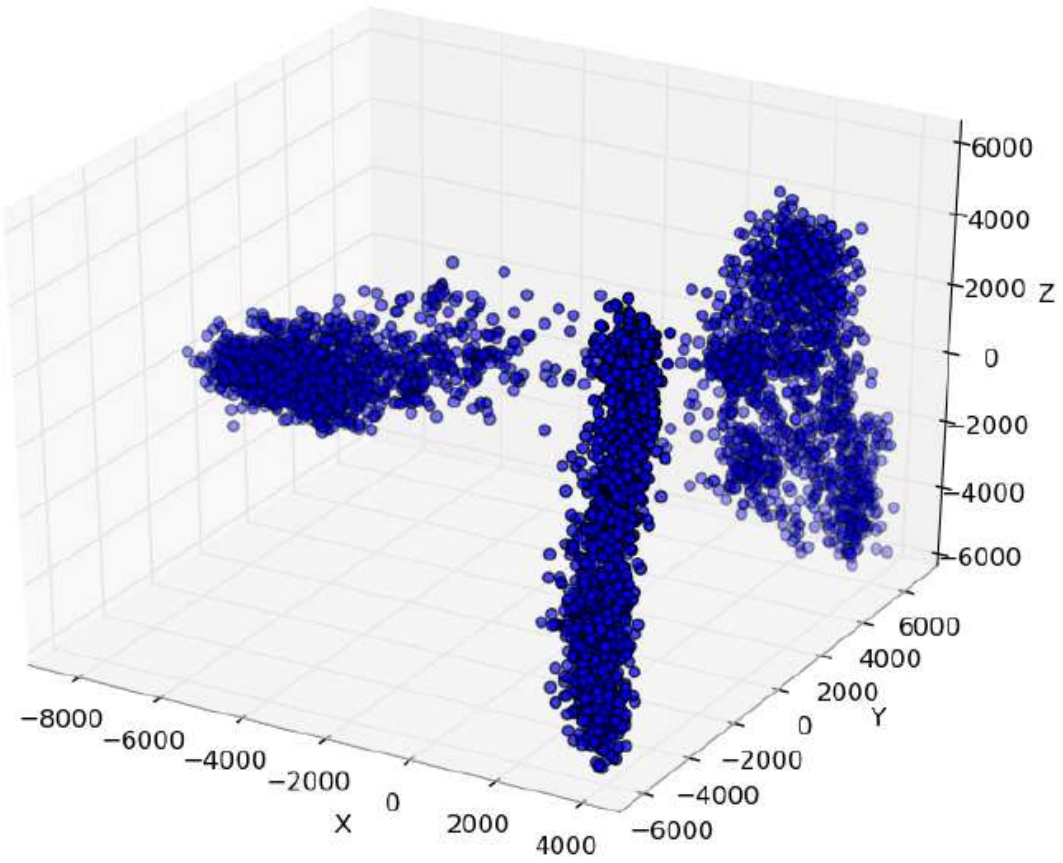


Figure 6.2: PCA from denoising autoencoder hidden layer [23]

training set, it may still be possible to map them into the word manifold and assign them novel classes. Figure 6.3 illustrates this for classes “truck” and “cat”.

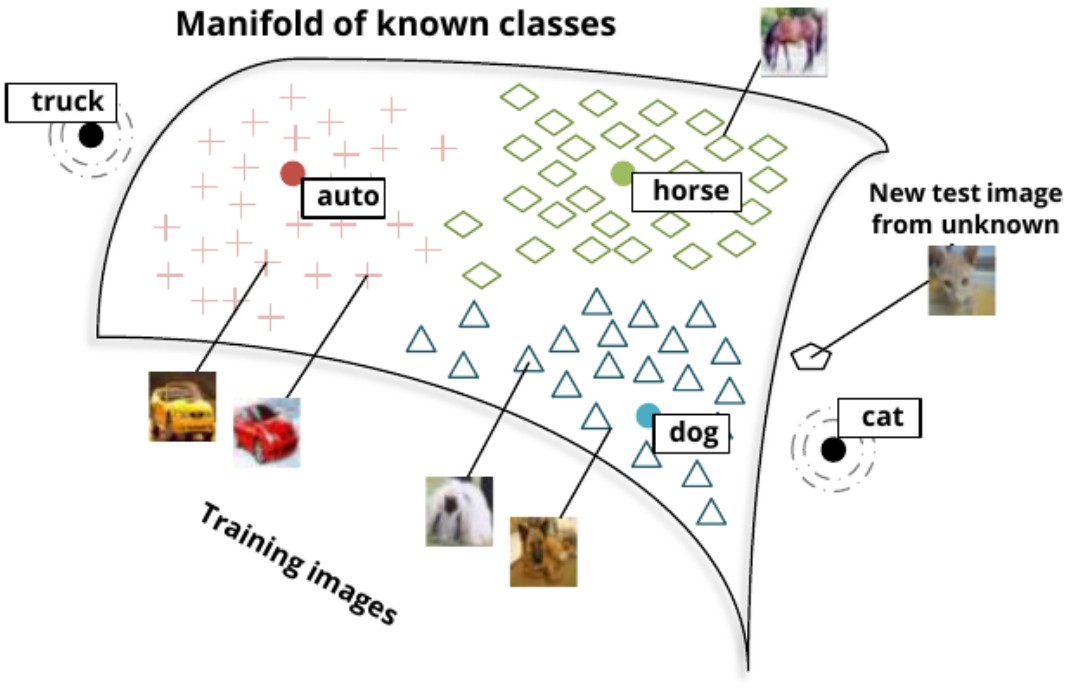


Figure 6.3: Zero-shot learning

6.4 Exercise: transference learning with the Keras functional API

The Keras functional API is more versatile than the sequential API. Layer instances in Keras are all callable, as Python functions. Each layer instance, when called, receives a tensor as argument and returns a tensor. This way we can chain layers in sequence. For example, here is a code fragment to create a convolution layer similar to the tutorial on Chapter 4. Note that we are now loading the MNIST data set instead of the Fashion MNIST which was used for training the previous model.

```
from tensorflow import keras
from tensorflow.keras.optimizers import SGD
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, BatchNormalization, Conv2D, Dense
from tensorflow.keras.layers import MaxPooling2D, Activation, Flatten, Dropout

((trainX, trainY), (testX, testY)) = keras.datasets.mnist.load_data()
trainX = trainX.reshape((trainX.shape[0], 28, 28, 1))
testX = testX.reshape((testX.shape[0], 28, 28, 1))
trainX = trainX.astype("float32") / 255.0
testX = testX.astype("float32") / 255.0

inputs = Input(shape=(28,28,1),name='inputs')
layer = Conv2D(32, (3, 3), padding="same", input_shape=(28,28,1))(inputs)
layer = Activation("relu")(layer)
layer = BatchNormalization(axis=-1)(layer)
layer = Conv2D(32, (3, 3), padding="same")(layer)
layer = Activation("relu")(layer)
layer = BatchNormalization(axis=-1)(layer)
layer = MaxPooling2D(pool_size=(2, 2))(layer)
...
```

As we feed the output tensor of each layer into the next, this creates a chain of operations and tensors in the computation graph. In this way, the functional API is similar too the sequential API. However, since we can decide which tensors go into which layers, we can create more complex graphs as necessary.

Note also that we can give names to specific layers. For example, the input layer is named `inputs` in this model. This makes it easier to find the right layers if we want to connect them in different ways. In this example, we want to recreate the CNN model from Chapter 4[1] to reuse the trained weights for the convolutional part of the network. This network was trained on the Fashion MNIST dataset and, instead of retraining the whole network, we will retain the feature extraction of the convolutional part and merely retrain the dense part of the network.

Once the layers are chained in the same way as the original model, we can create a model instance and compile it. To create the model we specify the inputs and outputs of the chain of tensors and operations.

```
#continuing the model
...
features = Flatten(name='features')(layer)
layer = Dense(512)(features)
layer = Activation("relu")(layer)
```

[1] Based on a PyImageSearch tutorial by Adrian Rosenbock, https://www.pyimagesearch.com/2019/02/11/fashion-mnist-with-keras-and-deep-learning/

```
layer = BatchNormalization()(layer)
layer = Dropout(0.5)(layer)

layer = Dense(10)(layer)
layer = Activation("softmax")(layer)
old_model = Model(inputs = inputs, outputs = layer)
```

Now that we have the model instance, we compile the model and load the weights. We will also disable training for all layers of the old model. This in order to use this model for training new layers.

```
old_model.compile(optimizer=SGD(), loss='mse')
old_model.load_weights('fashion_model')
for layer in old_model.layers:
    layer.trainable = False
```

Now we create a new model. First, we chain dense layers starting from the output of the convolutional net in the old model. This is why we named the `Flatten` layer as `features`. This makes it easier to get the correct layer from the old model. The `output` attribute of a layer is the tensor that the layer outputs, and this is the tensor we use as input for a new dense layer.

```
layer = Dense(512)(old_model.get_layer('features').output)
layer = Activation("relu")(layer)
layer = BatchNormalization()(layer)
layer = Dropout(0.5)(layer)
layer = Dense(10)(layer)
layer = Activation("softmax")(layer)
model = Model(inputs = old_model.get_layer('inputs').output, outputs = layer)
```

After creating a new chain of layers, we create the new model whose input is the input tensor to the old model and the output is the last layer in the new chain. This way we are using the convolutional part of the old model and, since we disabled training for this part, we can now fit the new model by training only the new part.

```
opt = SGD(lr=1e-2, momentum=0.9)
model.compile(loss="categorical_crossentropy", optimizer=opt, metrics=["accuracy"])
model.summary()
NUM_EPOCHS = 5
BS=512
H = model.fit(trainX, trainY,validation_data=(testX, testY),
              batch_size=BS, epochs=NUM_EPOCHS)
```

The `model.summary()` gives us a summary of the model. You should be able to see the different layers and, at the end, the indication of the trainable and non trainable parameters. These include some non-trainable layers, like batch normalization layers, but also all parameters in the convolution layers, which were frozen after loading. You should see something like this in the summary:

```
...
features (Flatten)           (None, 3136)              0
_________________________________________________________________
dense_2 (Dense)              (None, 512)               1606144
_________________________________________________________________
```

```
activation_6 (Activation)    (None, 512)               0
_________________________________________________________________
batch_normalization_5 (Batch (None, 512)               2048
_________________________________________________________________
dropout_3 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_3 (Dense)              (None, 10)                5130
_________________________________________________________________
activation_7 (Activation)    (None, 10)                0
=================================================================
Total params: 1,679,082
Trainable params: 1,612,298
Non-trainable params: 66,784
_________________________________________________________________
```

Using the pretrained convolution network to extract useful features, just a few epochs are enough to reach around 97% validation accuracy.

6.5 Exercise: convolutional autoencoder for MNIST

In this exercise you will implement a convolutional autoencoder to learn a 2D representation of the MNIST dataset. The encoder part will consist of a series of convolutional and pooling layers, then a dense layer with 2 linear neurons for the encoding. Then we add another dense layer, reshape and feed into a sequence of convolution and upsampling.

Loading the data for MNIST is similar to the Fashion MNIST example, but we will need some additional layers for our model because we will be implementing a convolutional autoencoder, which needs not only pooling but also upsampling layers. Here are the imports you will need and how to load the MNIST dataset:

```
from tensorflow.keras.optimizers import SGD
from tensorflow.keras.models import Model
from tensorflow.keras.layers import UpSampling2D,Reshape,Input
from tensorflow.keras.layers import BatchNormalization,Conv2D,MaxPooling2D
from tensorflow.keras.layers import Activation, Flatten, Dropout, Dense

((trainX, trainY), (testX, testY)) = keras.datasets.mnist.load_data()
trainX = trainX.reshape((trainX.shape[0], 28, 28, 1))
testX = testX.reshape((testX.shape[0], 28, 28, 1))

trainX = trainX.astype("float32") / 255.0
testX = testX.astype("float32") / 255.0
```

Now we create the autoencoder using the functional API from Keras. This is a sample of the code just to show how to create the encoder and decoder. First, the encoder will compress the inputs into increasingly smaller tensors until we reach the two neurons that create the 2D representation. This part is similar to a convolution network for classification, with pooling and flattening to feed into the dense layers:

```
def autoencoder():
    inputs = Input(shape=(28,28,1),name='inputs')
    layer = Conv2D(32, (3, 3), padding="same", input_shape=(28,28,1))(inputs)
    layer = Activation("relu")(layer)
    layer = BatchNormalization(axis=-1)(layer)
    [...]
    layer = MaxPooling2D(pool_size=(2, 2))(layer)
    [...]
    layer = Conv2D(8, (3, 3), padding="same")(layer)
    layer = Activation("relu")(layer)
    layer = BatchNormalization(axis=-1)(layer)

    layer = Flatten()(layer)
    features = Dense(2,name='features')(layer)
```

We will leave the neurons with the encoded representation as linear neurons and put this layer in a different variable called `features` so we can use it for separating the encoder from the full autoencoder.

Now we can apply batch normalization create a dense layer with the same number of neurons as the last feature map, and reshape it to feed into the convolution layers:

```
layer = BatchNormalization()(features)
layer = Dense(8*7*7,activation="relu")(features)
layer = Reshape((7,7,8))(layer)
layer = Conv2D(8, (3, 3), padding="same")(layer)
layer = Activation("relu")(layer)
```

If you start with a 28 by 28 input and do one 2 by 2 pooling, you reduce the feature map to 14 by 14. Another pooling and the feature map will be 7 by 7. This is what we did in the encoder before compressing to 2 neurons. Since the last convolution layer in the encoder had 8 filters, we now create a dense layer of 7*7*8 neurons in order to recreate the same tensor shape and feed into a similar convolution.

Now we replicate in the decoder the inverse of the operations we did in the encoder, using the `UpSampling2D` instead of pooling to double the resolution of the feature maps.

```
layer = BatchNormalization(axis=-1)(layer)
layer = Conv2D(16, (3, 3), padding="same")(layer)
layer = Activation("relu")(layer)
layer = BatchNormalization(axis=-1)(layer)
layer = UpSampling2D(size=(2,2))(layer)
[...]
layer = Conv2D(32, (3, 3), padding="same")(layer)
layer = Activation("relu")(layer)
layer = BatchNormalization(axis=-1)(layer)
layer = Conv2D(1, (3, 3), padding="same")(layer)
layer = Activation("sigmoid")(layer)
```

After we reach the 28 by 28 size again, we add a convolution layer with a single filter and a sigmoid activation to obtain a 28 by 28 feature map with values ranging from 0 through 1, corresponding to a grayscale image. This is what the autoencoder will learn to reproduce the original images input into the network.

All we need to do now is to create two models, one for the full autoencoder, from the inputs to the final layer, and one just for the encoder, from the inputs to the features layer where we encode the new features:

```
    autoencoder = Model(inputs = inputs, outputs = layer)
    encoder = Model(inputs=inputs,outputs=features)

    return autoencoder,encoder
```

You can experiment with different architectures, but I suggest starting from something like this:

- Convolution layer with 32 filters, followed by pooling
- Convolution layer with 32 filters, then convolution layer with 16 filters, then pooling
- Convolution layer with 16 filters, then convolution layer with 8 filters
- Dense layer with 2 neurons and linear output, then dense layer with 8*7*7 neurons (relu) and reshape
- Convolutions in reverse (8, 16, 16, 32 and 32 filters), and upsampling instead of pooling
- A final convolution of 1 filter for the (sigmoid) output.

Now you can train this autoencoder using the MNIST data set. Use the training data for training and the test data for validation, since we will not need to estimate the true error. The validation is useful in autoencoders because there is a danger of overfitting if the autoencoder cannot reconstruct the input for examples outside the training set. Remember to compile your autoencoder model with an optimizer and, after fitting, save the weights to a file.

Note that only a small part of the reconstruction error corresponds to the details, so you should let you autoencoder train for a while (e.g. 40 epochs).

Now you can use the encoder part to project the MNIST data into 2 dimensions. This is an example of how to create the model, load the weights and use the encoder to plot the representation in 2D:

```
def plot_representation():
    ae,enc = autoencoder()
    ae.load_weights('mnist_autoencoder.h5')
    encoding = enc.predict(testX)
    plt.figure(figsize=(8,8))
    for cl in np.unique(testY):
        mask = testY == cl
        plt.plot(encoding[mask,0],encoding[mask,1],'.',label=str(cl))
    plt.legend()
```

Note that the weights are loaded for the complete autoencoder, but only the encoder part (the second object returned by the `autoencoder` function) is used for creating the representation.

For plotting, we iterate through each class and plot the encoding of the images corresponding to that class with a different series.

You can also check the quality of the image reconstruction with this function:

```
from skimage.io import imsave

def check_images():
    ae,enc = autoencoder()
    ae.load_weights('mnist_autoencoder.h5')
    imgs = ae.predict(testX[:10])
    for ix in range(10):
        imsave(f'T03_{ix}_original.png',testX[ix])
        imsave(f'T03_{ix}_restored.png',imgs[ix])
```

This loads the autoencoder and uses the `predict` method to reconstruct the images. The original and reconstructed images are saved using the `imsave` function from the Scikit Image library.

6.6 Further Reading

1. Goodfellow et. al., Chapter 15[9]

Bibliography

[1] Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence*, 35(8):1798–1828, 2013.

[2] 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.

[3] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). *arXiv preprint arXiv:1511.07289*, 2015.

[4] George Cybenko. Approximations by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2:183–192, 1989.

[5] Dheeru Dua and Casey Graff. UCI machine learning repository, 2017.

[6] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

[7] Dumitru Erhan, Yoshua Bengio, Aaron Courville, Pierre-Antoine Manzagol, Pascal Vincent, and Samy Bengio. Why does unsupervised pre-training help deep learning? *Journal of Machine Learning Research*, 11(Feb):625–660, 2010.

[8] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, page 315–323, 2011.

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

[10] Klemen Grm, Vitomir Štruc, Anais Artiges, Matthieu Caron, and Hazım K Ekenel. Strengths and weaknesses of deep learning models for face recognition against image degradations. *IET Biometrics*, 7(1):81–89, 2017.

[11] 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.

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

[13] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[14] Kevin Jarrett, Koray Kavukcuoglu, Yann LeCun, et al. What is the best multi-stage architecture for object recognition? In *2009 IEEE 12th International Conference on Computer Vision (ICCV)*, page 2146–2153. IEEE, 2009.

[15] Nan Jiang, Wenge Rong, Baolin Peng, Yifan Nie, and Zhang Xiong. An empirical analysis of different sparse penalties for autoencoder in unsupervised feature learning. In *2015 international joint conference on neural networks (IJCNN)*, page 1–8. IEEE, 2015.

[16] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[17] 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.

[18] 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.

[19] 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.

[20] Zhou Lu, Hongming Pu, Feicheng Wang, Zhiqiang Hu, and Liwei Wang. The expressive power of neural networks: A view from the width. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, page 6231–6239. Curran Associates, Inc., 2017.

[21] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(Nov):2579–2605, 2008.

[22] 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.

[23] Grégoire Mesnil, Yann Dauphin, Xavier Glorot, Salah Rifai, Yoshua Bengio, Ian Goodfellow, Erick Lavoie, Xavier Muller, Guillaume Desjardins, David Warde-Farley, et al. Unsupervised and transfer learning challenge: a deep learning approach. In *Proceedings of the 2011 International Conference on Unsupervised and Transfer Learning workshop-Volume 27*, page 97–111. JMLR. org, 2011.

[24] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, page 807–814, 2010.

[25] 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.

[26] Sashank J Reddi, Satyen Kale, and Sanjiv Kumar. On the convergence of adam and beyond. 2018.

[27] 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.

[28] Salah Rifai, Yann N Dauphin, Pascal Vincent, Yoshua Bengio, and Xavier Muller. The manifold tangent classifier. In *Advances in Neural Information Processing Systems*, page 2294–2302, 2011.

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

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

[31] 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.

[32] Wenling Shang, Kihyuk Sohn, Diogo Almeida, and Honglak Lee. Understanding and improving convolutional neural networks via concatenated rectified linear units. In *international conference on machine learning*, page 2217–2225, 2016.

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

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

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

[36] Richard Socher, Milind Ganjoo, Christopher D Manning, and Andrew Ng. Zero-shot learning through cross-modal transfer. In *Advances in neural information processing systems*, page 935–943, 2013.

[37] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[38] 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.

[39] Matus Telgarsky. Benefits of depth in neural networks. *arXiv preprint arXiv:1602.04485*, 2016.

[40] Joshua B Tenenbaum, Vin De Silva, and John C Langford. A global geometric framework for nonlinear dimensionality reduction. *science*, 290(5500):2319–2323, 2000.

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

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