Deep Learning HS19

Pascal Baumann
pascal.baumann@stud.hslu.ch

26. Januar 2024

For errors or improvement raise an issue or make a pull request on the github repository.

Contents

1 Introduction

Deep Learning is a sub-branch of Machine Learning. Machine Learning itself is at the convergence point of Mathematics, Algorithmics, Optimization, Signal Processing and Software Engineering, and "could be defined as a set of methods that automatically detect patterns in data, and then use the uncovered patterns to predict future data, or to perform other kinds of decision making under uncertainty."

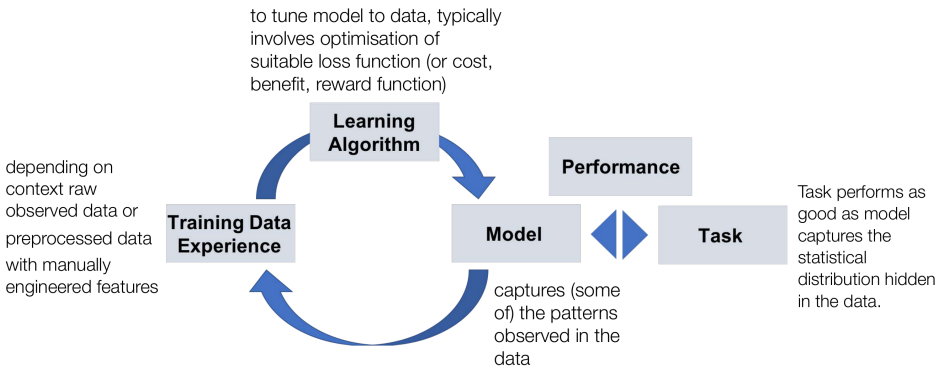


Figure 1.1: An overview of the Machine Learning workflow

When talking about Machine Learning there is a distinction between **supervised learning** and **unsupervised learning**. In supervised learning the goal is to extract some relevant features x from raw observation data o and to learn a mapping from inputs x to outputs y given a set of example data called the training set.

In contrast to unsupervised learning where the goal is to discover interesting structures from inputs x given a set of data called the training set. Or in other words, to determine characteristics or features that can be used to optimally define groups.

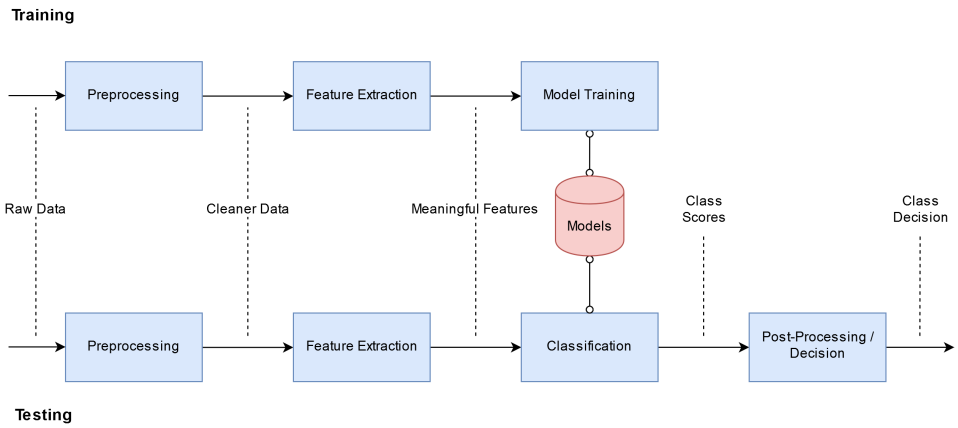


Figure 1.2: The machine learning process

As mentioned before, Deep Learning is a new subfield of Machine learning and at the convergence of larger quantities of data, new algorithms and better computer performance, even specialised computers for machine learning. In the core deep learning is a neuronal architecture with many layers and neurons, where less or no feature extraction is needed, as the model finds them automatically as part of the learning. The drawback is, that Deep Learning models need much more data than Machine Learning models.

2 Perceptron

The neuron in artificial neural networks is modelled after a neuron of the human brain, as it is able to send a signal after receiving enough activations until his own threshold is reached. The first artificial neuron model was the McCulloch-Pitts Neuron visualised in **??**. They showed that, in principle, any logical or arithmetic function can be computed with networks of such neurons.

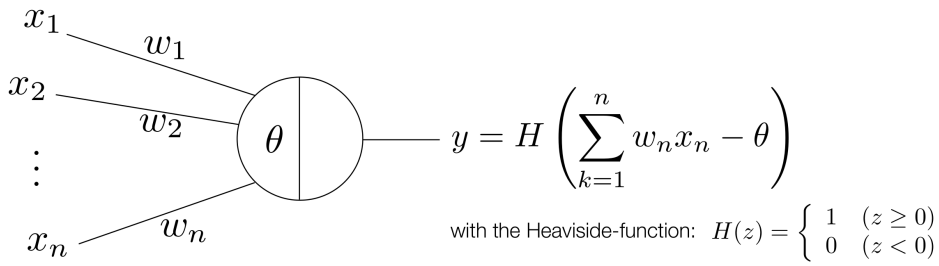


Figure 2.1: McCulloch-Pitts Neuron

In 1949 Donald Hebb postulated what is now known as Hebb's Rule: "Let us assume that the persistence or repetition of a reverberatory activity tends to induce lasting cellular changes that add to its stability. [...] When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased." This hypothesis is not proven biologically, but is taken as a principle for ANNs to adjust their connection weights. It should be noted, however, that this learning rule does not provide a stable learning.

In 1958 Rosenblatt presented a single perceptron, also referred as Linear Threshold Unit (LTU):

- Inputs and outputs are *real numbers*
- Each input connection is associated with a *real* weight
- The activity is computed by applying the Heaviside function to the weighted sum of the input while incorporating the activation threshold in form of a bias term b

$$y = H\left(\sum_{k=1}^{n} w_k x_k + b\right) \quad \text{for } x_k, w_k, b \in \mathbb{R}$$

Thus the output can take on two distinct values 0,1

$$H(x) = \begin{cases} 0 & x < 0 \\ 1 & x \geq 0 \end{cases}$$

Equation 1: The Heaviside step function

A single perceptron can be used for binary classification problems, where the decision boundary is given by

$$H_{w,b} = \left\{\mathbf{x} \in \mathbb{R}^D | \mathbf{w} \cdot \mathbf{x} + b = 0\right\} \quad \text{with } \mathbf{w} = \begin{pmatrix} w_1 \\ \vdots \\ w_D \end{pmatrix}, \mathbf{x} = \begin{pmatrix} x_1 \\ \vdots \\ x_D \end{pmatrix}$$

Therefore the perceptron is suited for **linearly separable binary classification problems**.

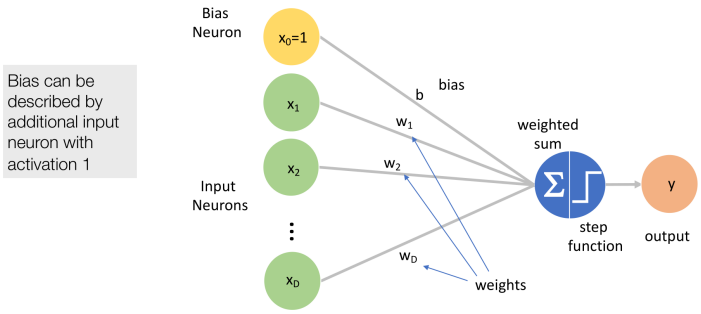


Figure 2.2: The Rosenblatt perceptron

2.1 Perceptron Learning Algorithm

Labelled Data: $(\mathbf{x}^{(i)}, y^{(i)})|i = 1, ..., N$ where $\mathbf{x}^{(i)}$ are d-vectors

1. Initialise parameters
   all zeroes or small random numbers
2. Iterate by updating weights according to
   A. Pick sample $\left(\mathbf{x}^{(i)}, y^{(i)}\right)$
   B. Compute predicted values $\hat{y}^{(i)} = H(\mathbf{w} \cdot \mathbf{x}^{(i)} + b)$
   C. Parameter update rule only update if $\hat{y}^{(i)} \neq y^{(i)}$

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \cdot \left(\hat{y}^{(i)} - y^{(i)}\right) \cdot \mathbf{x}^{(i)}$$
$$b \leftarrow b - \alpha \cdot \underbrace{\left(\hat{y}^{(i)} - y^{(i)}\right)}_{-1,0,1}$$

The learning rule searches for a weights vector that defines a hyperplane that separates the points associated with the two classes. This is only possible for linearly separable input sets. In case of wrongly classified points, the weights update rule leads to a correction of the weights such that the hyperplane is tilted to try to bring the wrongly classified points to the correct side.

Theorem 1 *Perceptron Learning Algorithm converges in a finite number of steps to a weights vector and bias that separates the two classes - provided that the two classes are linearly separable.*

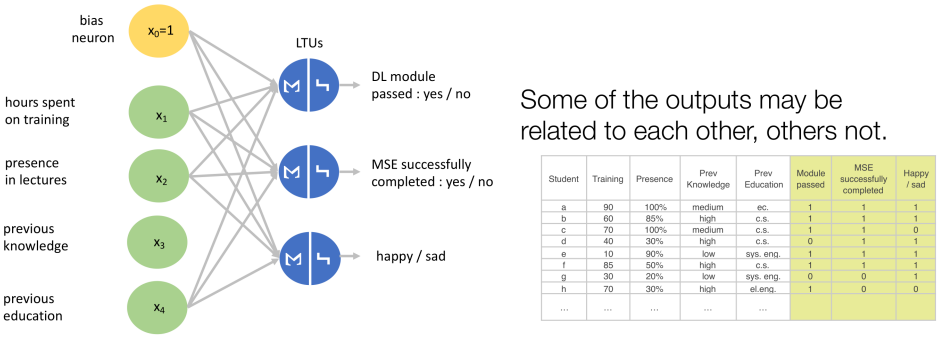
However, the solutions obtained for linearly separable inputs is not unique and not optimal. The optimal solution (with the widest separating corridor) is known as the linear support vector machine (SVM). The perceptron may be a reasonable model also for problems that are not linearly separable. However, the application of the Perceptron Learning Algorithm does not converge in this case.

2.2 Single Layer LTUs

The inputs for ANNs are often represented using special *passthrough* neurons called **input neurons**, if each neuron in the subsequent layer is connected to all the preceding neurons, the layer is **fully connected**. Generally, an extra bias feature $x_0 = 1$ is added, often represented with a **bias neuron** that just outputs one and whose influence is adjusted by the weight (see **??**).

2.2.1 Possible Applications

A single layer LTU with m units can classify instances simultaneously into $\boldsymbol{m}$ **different binary classes**, that is a multi-output classifier that can perform multiple tasks and is able to apply the learning algorithm independently to each output.



Student	Training	Presence	Prev Knowledge	Prev Education	Module passed	MSE successfully completed	Happy / sad
a	90	100%	medium	ec.	1	1	1
b	60	85%	high	c.s.	1	1	1
c	70	100%	medium	c.s.	1	1	0
d	40	30%	high	c.s.	0	1	1
e	10	90%	low	sys. eng.	1	1	1
f	85	50%	high	c.s.	1	1	1
g	30	20%	low	sys. eng.	0	0	1
h	70	30%	high	el.eng.	1	0	0
...	...	...	...	...			

Figure 2.3: An example for a multi-output classification with LTU

The same model might also be considered for classifying input data into m different exclusive classes. The output values, or labels, in the dataset would be prepared as one-hot vectors with only one non-zero entry. There are better ways for handling this task, like the Softmax algorithm, which will be introduced in a future section.

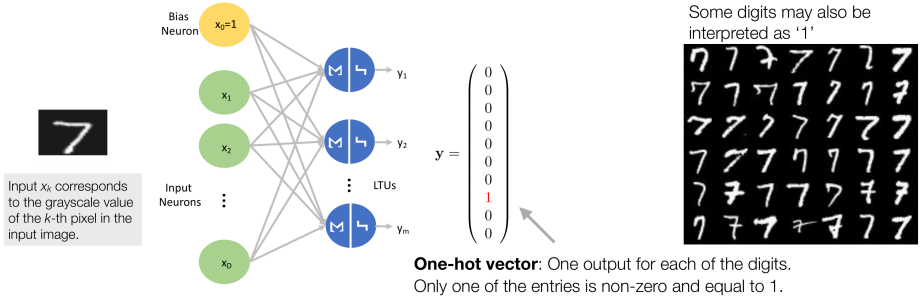
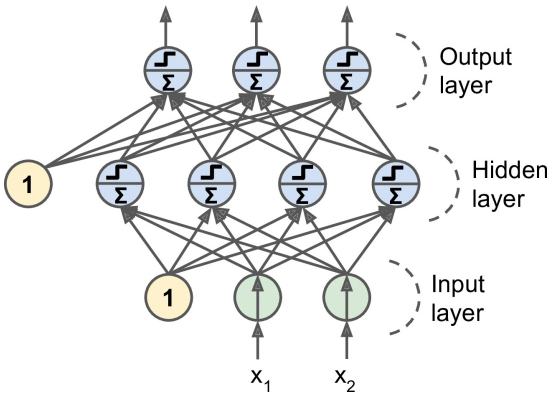


Figure 2.4: An example for an exclusive multi-output classification with LTU

The limitation of a single layer perceptron is that it's incapable of solving problems with a non-linear decision boundary. This limitation can be overcome by stacking multiple layers of a perceptron into a so called Multi-Layer Perceptron (MLP). An MLP is composed of:

- Input layer: Inputs are passed by pass-through neurons
- Hidden layers: One or more layers of LTUs
- Output layer: Final layer with a single or multiple LTUs (depending on the problem)



Both the input layer and hidden layers of an MLP include a bias neuron, and are fully connected to the next layer. The Activation Function is a generalisation of the hard threshold (Heaviside Function) and replaced by smooth functions.

3 MNIST

MNIST is a public dataset with labelled data widely used for testing and illustration:

- 70000 images
- 28x28-pixel
- available for download from `mldata.org`

3.1 Load MNIST Images

4 Data Preparation

4.1 Training and Test Set

Split the data into training and test set, making sure that both sets have the same characteristics. It's a good idea to randomly shuffle the datasets before splitting, unless one is sure that the dataset is already sufficiently shuffled. It's of utmost importance to keep the two sets strictly separate while training, so no information contained in the test set is used to adjust parameters of the model.

	Small Datasets	**Large Datasets**
Train	70-80%	99 %
Test	30-20%	1%

Table 1: Recommended Train/Test split given a particular size of the dataset

4.1.1 Splitting using SciKit-Learn

4.2 Data Normalisation and Feature Scaling

The numeric values of features contained in the data may be on different scales, ranging over different order of magnitude or centred around different values on average. The data should thus be brought to similar scales, so called Feature Scaling, and approximately centred around zero, the so called Feature Centring. This is important as the *numerical stability* of the learning algorithm depends on it and it can focus on learning the importance of the features, which leads to *improved convergence properties.*

Two different methods are mostly used:
- z-Normalisation
- Min-Max-Rescaling or Min-Max Normalisation

4.2.1 z-Normalisation

Shifting and **rescaling** the data so that a zero mean and a unit-variance is obtained.

$$x'^{(i)}_k = \frac{x^{(i)}_k - \mu_k}{\sigma_k}$$

where the mean and variance for the k-th component are

$$\mu_k = \frac{i}{N} \sum_{k=1}^{N} x^{(i)}_k$$

$$\sigma^2_k = \frac{i}{N} \sum_{k=1}^{N} \left(x^{(i)}_k - \mu_k \right)^2$$

The parameters μ_k, σ_k should be **computed on the training set** only, as computing it on the whole data set is considered to falsify the training. When normalising image data the mean and standard deviation are calculated over all the pixels.

4.2.2 Min-Max-Rescaling and Normalisation

Min-Max-Rescaling

$$x'^{(i)}_k = \frac{x^{(i)}_k - \min_k}{\max_k - \min_k}$$

Features are rescaled to lie in the $[0, 1]$ range.

Min-Max-Normalisation

$$x'^{(i)}_k = 2 \cdot \frac{x^{(i)}_k - \min_k}{\max_k - \min_k} - 1$$

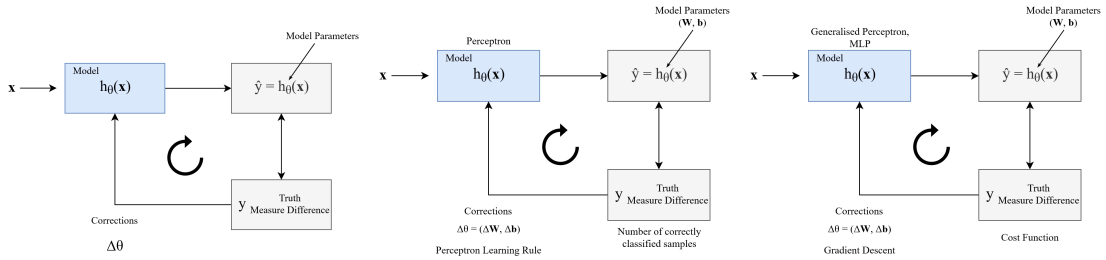
Features are rescaled to lie in the $[-1, 1]$ range. The same remarks as for the z-Normalisation hold true: min/max should be computed on the training set, and computed over all pixels when using image data.

5 Training a Model

5.1 Generalised Perceptron

The first generalisation from Rosenblatt's Perceptron is the replacement of the Heaviside function as the activation function through a smooth Sigmoid function:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$



(a) General training (b) Training a perceptron (c) Training a MLP

This allows for a probabilistic interpretation of the output. The sigmoid is continuous and thus differentiable. With this, Optimisation techniques can be used.

The second generalisation is the use of the Optimisation technique Gradient Descent for Learning, this technique is also applicable for problems that are not linearly separable and for MLPs.

5.1.1 Model for Binary MNIST Classification

$$\hat{y} = h_\theta(\mathbf{x}) = \sigma(\mathbf{w} \cdot \mathbf{x} + b)$$

Where $\sigma(z)$ is the sigmoid function, and the model parameters $\theta = (\mathbf{w}, b)$. The output is no longer binary, but a numeric value $\hat{y} \in]0, 1[$ and can be transformed to a class label by the rule:

$$\hat{y} = \begin{cases} 1 & \text{if } h_\theta(\mathbf{x}) > 0.5 \\ 0 & \text{if } h_\theta(\mathbf{x}) \leq 0.5 \end{cases}$$

5.2 Cost Function

Choose the initial parameters such that the predicted values $\hat{y}$ are in some notion of distance close to the true outcomes y. This notion of distance is expressed in terms of a cost function, whereas the parameters are chosen in such a way that this function is minimised. Different cost functions will lead to different solutions.

5.2.1 Mean Square Error (MSE) Cost Function

The MSE cost function is based on the principle of the mean quadratic distance between y and $\hat{y}$.

$$J_{\text{MSE}}(\theta) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2 \tag{1}$$

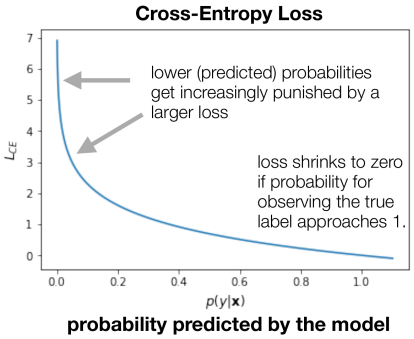
By minimising $J_{\text{MSE}}(\theta)$ with respect to the parameters θ we adjust the model to best represent the mapping $\mathbf{x} \to y$ seen in the training data.

5.2.2 Cross Entropy (CE) Cost Function

The CE cost function is based on the Cross-Entropy Loss defined by

$$L_{\text{CE}}\left((\mathbf{x}, y), \theta\right) = -\log\left(p_\theta(y|\mathbf{x})\right)$$

This loss function is suited for classification problems, and based on the model predicting the probability for observing class y given $\mathbf{x}$ denoted by $p_\theta(y|\mathbf{x})$.



The cross entropy cost function is then the averaged cross entropy loss

$$J_{\text{CE}} = -\frac{1}{m}\sum_{i=1}^{m} \log\left(p\left(y^{(i)}|\mathbf{x}^{(i)},\theta\right)\right) \tag{2}$$

For the binary classification problem

$$J_{\text{CE}} = -\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}\log(h_\theta(\mathbf{x}^{(i)})) + (1-y^{(i)})log(1-h_\theta(\mathbf{x}^{(i)}))\right) \tag{3}$$

In the given single layer perceptron problem the cross-entropy cost function is a convex function. As a result, the function has a single local minimum (a single critical point) which is the global minimum. As long as the chosen learning rate is sufficiently small, the gradient descent will always converge to a global minimum.

5.3 Gradient Descent

The basic principle of the Gradient Descent is the minimisation of some cost function $J(\theta)$:

1. Start with some initial value for the parameter vector: θ_0
2. Iteratively update the parameter vector by
   i. Compute the gradient of the cost function $J(\theta)$ at the last position reached (θ_t)
   ii. Step in the negative gradient direction according to

$$\theta_{t+1} = \theta_t - \alpha \cdot \nabla_\theta J(\theta_t)$$

   where α is the learning rate
3. Stop when the change in parameter vector is small

The learning principle works locally (by using local function properties and sufficiently small step sizes), but can get stuck in critical points were the gradient is zero. Without additional information the learning schema cannot discern between local minima, maxima or saddle point. Because the learning schema is iterative, it iteratively approaches a critical point and may fluctuate around it.

5.3.1 Computing Gradients

The Gradient is a **partial derivative** of the function $J(\theta) = J(\theta_1, \theta_2, ..., \theta_n)$ with regard to θ_k

$$\frac{\partial J}{\partial \theta_k} = \lim_{\epsilon \to 0}\left(J(\theta) = J(\theta_1, ..., \theta_k + \Delta\theta_k, ..., \theta_n) - J(\theta_1, ..., \theta_k, ..., \theta_n)\right)$$

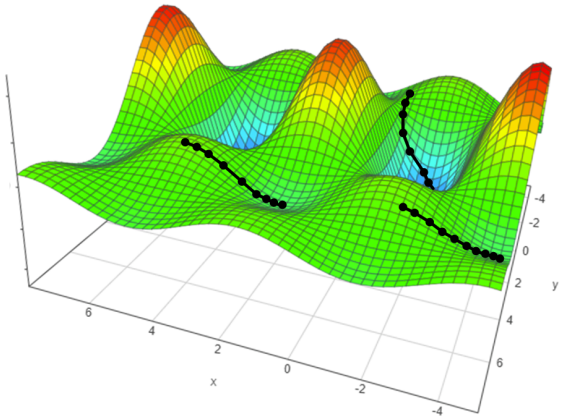


Figure 5.2: Illustration of a gradient descent, different initialisation and optimisers lead to different outcomes

And the **gradient**:

$$\nabla_\theta J = \frac{\partial J}{\partial \theta} = \begin{pmatrix} \frac{\partial J}{\partial \theta_1} \\ \vdots \\ \frac{\partial J}{\partial \theta_n} \end{pmatrix}$$

Linearisation (Taylor expansion): $J(\theta_0 + \Delta\theta) \approx J(\theta_0) + \Delta\theta \cdot \nabla_\theta J(\theta_0)$
Steepest Descent in direction of $\Delta\theta = -\alpha \nabla_\theta J(\theta_0)$ for sufficiently small α:

$$J\left(\theta_0 - \alpha \nabla_\theta J(\theta_0)\right) \approx J(\theta_0) - \alpha \|\nabla_\theta J(\theta_0)\|^2$$

5.3.2 Gradient of Mean Square Error Cost Function

$$J_{\text{MSE}}(\theta) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)}\right)^2$$

$$\nabla_{\mathbf{w}} J_{\text{MSE}}(\mathbf{w}, b) = \frac{1}{m} \sum_{i=1}^{m} \hat{y}^{(i)}(1 - \hat{y}^{(i)})(\hat{y}^{(i)} - y^{(i)})\mathbf{x}^{(i)}$$

$$\nabla_b J_{\text{MSE}}(\mathbf{w}, b) = \frac{1}{m} \sum_{i=1}^{m} \hat{y}^{(i)}(1 - \hat{y}^{(i)})(\hat{y}^{(i)} - y^{(i)})$$

5.3.3 Mean Square Error Update Rules

In vector notation: $\theta \leftarrow \theta - \alpha \nabla_\theta J(\theta)$
In coordinates: $\theta_k \leftarrow \theta_k - \alpha \frac{\delta J(\theta)}{\delta \theta_k}$

5.3.4 Issues with MSE Cost and Alternatives

The gradient of the MSE cost contains in each summand a factor $\hat{y}^{(i)}(1 - \hat{y}^{(i)}) \leq \frac{1}{4}$. As the model output gets closer to either 0 or 1 (where the model is very confident in its prediction) this expression and the gradient can get very small. In result, the change in parameters can get very small and training can get stalled or stuck. Thus for *classification tasks* the **cross entropy cost function** is better suited and its gradient has more preferable properties.

5.3.5 Gradient of the Cross-Entropy Cost

The calculation of the update rule for the cross-entropy cost function is more complicated. Here it is computed for the binary classification case. When calculating the gradient for the CE cost one starts with:

$$\nabla h_\theta(\mathbf{x}) = h_\theta(\mathbf{x}) \cdot (1 - h_\theta(\mathbf{x})) \begin{pmatrix} \mathbf{x} \\ 1 \end{pmatrix} \tag{4}$$

With this the gradient of the CE Loss may be computed:

$$\begin{aligned} -\nabla L_{\text{CE}}(\theta) &= \nabla \left(y \log(h_\theta(\mathbf{x})) + (1-y) \log(1 - h_\theta(\mathbf{x})) \right) \\ &= \frac{y}{h_\theta(\mathbf{x})} \nabla h_\theta(\mathbf{x}) - \frac{1-y}{1-h_\theta(\mathbf{x})} \nabla h_\theta(\mathbf{x}) \\ &= \left(\frac{y}{h_\theta(\mathbf{x})} - \frac{1-y}{1-h_\theta(\mathbf{x})} \right) \nabla h_\theta(\mathbf{x}) \\ &\overset{??}{=} h_\theta(\mathbf{x}) \cdot (1 - h_\theta(\mathbf{x})) \begin{pmatrix} \mathbf{x} \\ 1 \end{pmatrix} \\ &= (y - h_\theta(\mathbf{x})) \begin{pmatrix} \mathbf{x} \\ 1 \end{pmatrix} \end{aligned}$$

The gradient of the cost function is the gradient per sample ('loss') summed over all training samples

$$\nabla J_{\text{CE}}(\theta) = \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(\mathbf{x}^{(i)}) - y^{(i)} \right) \begin{pmatrix} \mathbf{x}^{(i)} \\ 1 \end{pmatrix} \tag{5}$$

The **Error Signal** contributes more for samples with a mismatch and consists of:

$h_\theta(\mathbf{x}^{(i)})$	Predicted probability for seeing label i
$y^{(i)}$	Label i

5.3.6 Cross Entropy Update Rules

These are the formulas for the update rules for a single layer perceptron

$$\mathbf{w} \leftarrow \mathbf{w} - \frac{\alpha}{m} \sum_{i=1}^{m} \left(h_\theta(\mathbf{x}^{(i)}) - y^{(i)} \right) \mathbf{x}^{(i)} \tag{6}$$

$$b \leftarrow b - \frac{\alpha}{m} \sum_{i=1}^{m} \left(h_\theta(\mathbf{x}^{(i)}) - y^{(i)} \right) \tag{7}$$

The cross entropy cost function for the generalised perceptron is a convex function, therefore the gradient descent is guaranteed to find a global minimum given a sufficient small learning rate.

6 Stochastic Gradient Descent

In normal gradient descent the cost function on the whole training set is computed, this is colloquially referred to as Batch Gradient Descent. This computation can be costly, as the evaluation of the model and its derivatives is done for the all the samples in the training set. This leads to the concept used in Stochastic Gradient Descent and Mini-Batch Gradient Descent: The assumption that the approximate update direction is obtained already with a few samples.

- **Batch GD**: Averaging over all training samples

- **Mini-Batch GD**: Averaging only over a subset of training samples

- **Stochastic GD**: No averaging, a single training sample is selected at random

Sometimes both mini-batch and stochastic gradient descent are called Stochastic Gradient Descent.

6.1 Characteristics of Stochastic Gradient Descent (SGD)

General Characteristics

- Tends to move in direction of the global minimum, but not always
- Never converges like batch gradient descent does, but ends up fluctuating around the global minimum.
- Allows for escaping local minima. Regularising effect
- Learning principle is generalisable to many other "hypothesis families"

Advantages

- Needs less epochs since parameters are updated for each training sample
- Can handle very large sets of data (**out-of-core learning**)
- Allows for incremental learning (**online learning**), that is on-the-fly adjustment of the model parameters on new incoming data

Disadvantages

- Not easily parallelised

6.1.1 Stochastic Gradient Descent for Simple Binary Classification with Cross Entropy Loss

1. Start with some inital $\theta = (b, \mathbf{w})$
2. Select **one** training sample $(\mathbf{x}^{(i)}, y^{(i)})$ at random and perform the update of the θ with this sample

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \left(h_\theta(\mathbf{x}^{(i)}) - y^{(i)} \right) \mathbf{x}^{(i)}$$
$$b \leftarrow b - \alpha \left(h_\theta(\mathbf{x}^{(i)}) - y^{(i)} \right)$$

3. Loop step 2. until convergence

6.2 Characteristics of Mini-Batch Gradient Descent (MBGD)

Mini-Batch Gradient Descent is a good compromise between Batch Gradient Descent and Stochastic Gradient Descent.

General Characteristics

- Tends to move in direction of the global minimum, but not always
- Ends up wandering close around the global minimum. Regularising effect

- Allows for escaping local minima. Regularising effect
- As with batch gradient descent, the learning principle is generalisable to many other "hypothesis families"

Advantages

- Faster than Batch Gradient Descent, since parameters are updated for each mini-batch
- Level of noise in the learning curve is reduced compared to stochastic gradient descent
- Allows for vectorised implementations, potentially more efficient through pipelining
- Straightforward distribution of computational load by computing mini-batches on different machines or cores
- Can handle very large sets of data (**out-of-core learning**)
- Allows for incremental learning (**online learning**)
- Parallelisable on GPU and in HPC (high-performance computing)

Disadvantages

- Batch size needs to be optimised

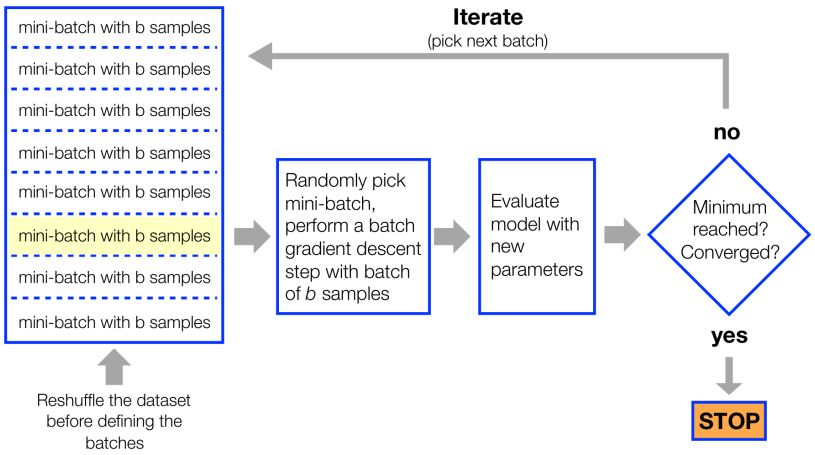


Figure 6.1: Flow diagram of Mini-Batch Gradient Descent

6.2.1 Mini-Batch Gradient Descent for Simple Binary Classification with Cross Entropy Loss

1. Start with some inital $\theta = (b, \mathbf{w})$
2. Select mini-batch of b training samples with indices $i_1, ..., i_b$ at random

$$(\mathbf{x}^{(i_1)}, y^{(i_1)}), ..., (\mathbf{x}^{(i_b)}, y^{(i_b)})$$

and perform the update of the θ with this sample

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \sum_{j=1}^{b} \left(h_\theta(\mathbf{x}^{(i_j)}) - y^{(i_j)} \right) \mathbf{x}^{(i_j)}$$

$$b \leftarrow b - \alpha \sum_{j=1}^{b} \left(h_\theta(\mathbf{x}^{(i_j)}) - y^{(i_j)} \right)$$

3. Loop step 2. until convergence

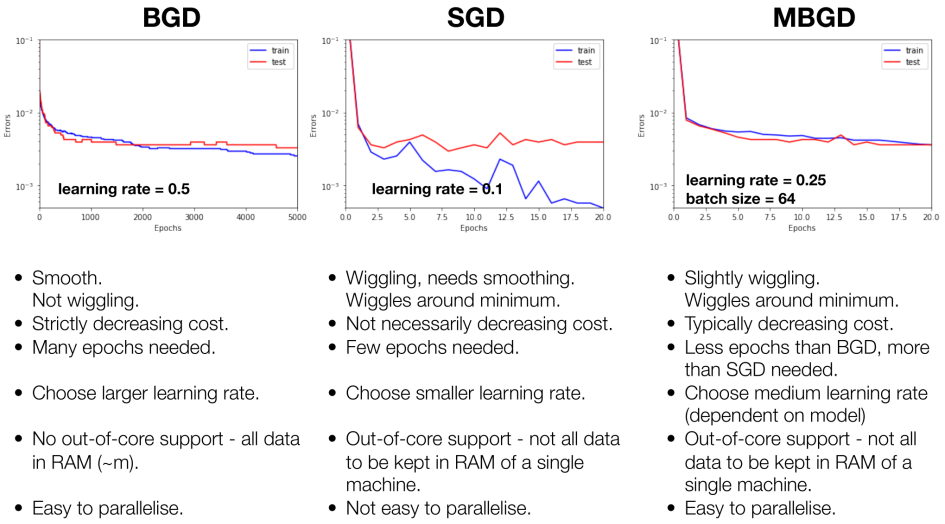


Figure 6.2: Comparison between the different Gradient Descent methods

7 Multi-Class Classification and Softmax

7.1 Multiple Binary Classifiers for Multi-Class Classification

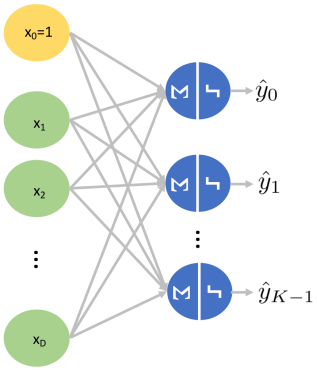
Class labels (K classes): $0 \leq l < K$

Structured as K independent binary classification problems:

$$h_{\theta_l}(\mathbf{x}) = \sigma(\mathbf{w}_l \cdot \mathbf{x} + b_l)$$

$\theta_l = (\mathbf{w}_l, b_l)$ Model parameters for class l, where each class is trained individually

Predicted class $\widehat{y}(\mathbf{x}) = \underset{l}{\text{argmax}} \{h_{\theta_l}(\mathbf{x})\} \in \{0, 1, ..., K-1\}$



However, as each class is trained separately and independent from the rest, the model is not trained to provide normed probabilities over all the classes. This means that the system is not forced to decide between one of the classes during training.

7.2 Multi-Classification with Softmax

softmax function

$$p_l = \text{softmax}(\mathbf{z})_l = \frac{\exp(\mathbf{z}_l)}{\sum_{j=0}^{K} \exp(\mathbf{z}_j)}$$

where $z_l = \mathbf{w}_l \cdot \mathbf{x} + b_l, 0 \leq l < K$ or as a vector $\mathbf{z} = \mathbf{W} \cdot \mathbf{x} + \mathbf{b}$

Properties

- Normed: $\sum_{l=0}^{K-1} p_l = 1$, so the result can be interpreted as normed probabilities
- Peaks at the largest z_l. The components of softmax smoothly approximate the index of the largest element

$$p_l \approx \delta_{l,\text{argmax}\{z_k\}} \text{ if } z_l \gg z_k (\forall k \neq l)$$

- Parameters: $\theta = ((\mathbf{w}_1, b_1), ..., (\mathbf{w}_{K-1}, b_{K-1})) = (\mathbf{W}, \mathbf{b})$

7.2.1 Cost Function with Softmax

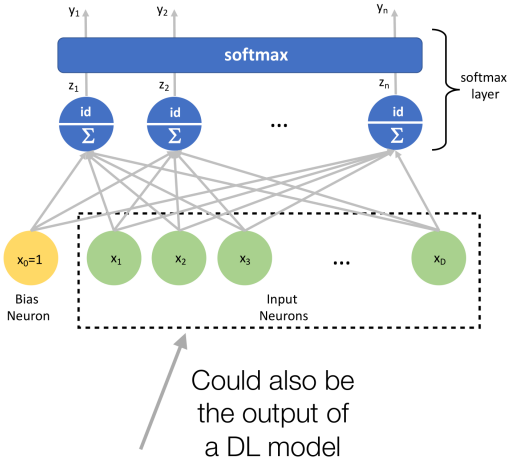
The predicted class stays the same $\widehat{y}(\mathbf{x}) = \underset{l}{\text{argmax}} \{h_{\theta_l}(\mathbf{x})\} \in \{0, 1, ..., K-1\}$, but $h_{\theta,l}(\mathbf{x})$ is now a probability distribution for the different classes.

$$p(y = l | \mathbf{x}, \theta) = h_{\theta,y}(\mathbf{x})$$

The cost function is the cross entropy in accordance with the maximum likelihood principle

$$\begin{aligned} J_{CE}(\theta) &= -\frac{1}{m} \sum_{i=1}^{m} \log \left(p(y^{(i)} | \mathbf{x}^{(i)}, \theta) \right) \\ &= -\frac{1}{m} \sum_{i=1}^{m} \log \left(h_{\theta, y^{(i)}}(\mathbf{x}^{(i)}) \right) \end{aligned}$$

The mapping provided by the softmax function is a specific form of neuron with m inputs and m outputs. The inputs $\mathbf{z} = (z_1, ..., z_m)$ are computed by an affine transformation from the D dimension input $\mathbf{x}$ (with weights matrix $\mathbf{W}$ and bias vector $\mathbf{b}$).
The outputs $p_1, ..., p_K$ can be interpreted as normed probabilities associated with the different classes. Theses ingredients form a *softmax layer* which is typically used as a final layer in classification problems.



7.2.2 Calculation of the Gradient

With the output of the softmax layer $h_{\theta,l}(\mathbf{x}$ differentiated in respect to the true label z_k

$$\frac{\partial h_{\theta,l}(\mathbf{x})}{\partial z_k} = \delta_{l,k} h_{\delta,l}(\mathbf{x}) - h_{\delta,l}(\mathbf{x}) h_{\delta,k}(\mathbf{x})$$

one obtains the gradient in respect to the weights

$$\begin{aligned}
\frac{\partial}{\partial \mathbf{w}_j} J_{\mathrm{CE}}(\theta) &= -\frac{1}{m}\sum_{i=1}^{m} \frac{\partial}{\partial \mathbf{w}_j} \log\left(h_{\theta,y^{(i)}}(\mathbf{x}^{(i)})\right) \\
&= -\frac{1}{m}\sum_{i=1}^{m} \frac{1}{h_{\theta,y^{(i)}}(\mathbf{x}^{(i)})} \frac{\partial h_{\theta,y^{(i)}}(\mathbf{x}^{(i)})}{\partial \mathbf{w}_j} \\
&= -\frac{1}{m}\sum_{i=1}^{m} \frac{1}{h_{\theta,y^{(i)}}(\mathbf{x}^{(i)})} \frac{\partial h_{\theta,y^{(i)}}(\mathbf{x}^{(i)})}{\partial z_j} \frac{\partial z_j}{\partial \mathbf{w}_j} \\
&= -\frac{1}{m}\sum_{i=1}^{m} \frac{1}{h_{\theta,y^{(i)}}} \left(\delta_{j,y^{(i)}} h_{\theta,j}(\mathbf{x}^{(i)}) - h_{\theta,j}(\mathbf{x}^{(i)}) h_{\theta,y^{(i)}}(\mathbf{x}^{(i)})\right) \frac{\partial z_j}{\partial \mathbf{w}_j} \\
&= -\frac{1}{m}\sum_{i=1}^{m} \left(\delta_{j,y^{(i)}} - h_{\theta,y^{(i)}}(\mathbf{x}^{(i)})\right) \mathbf{x}^{(i)}
\end{aligned}$$

with $\delta_{k,j} = \begin{cases} 1 & (k = j) \\ 0 & (k \neq j) \end{cases}$ being the **Kronecker symbol**.

Gradient with respect to the weights and biases of the softmax layer

$$\begin{aligned}
\frac{\partial}{\partial \mathbf{w}_j} J_{\mathrm{CE}}(\theta) &= -\frac{1}{m}\sum_{i=1}^{m} \left(\delta_{j,y^{(i)}} - h_{\theta,y^{(i)}}(\mathbf{x}^{(i)})\right) \mathbf{x}^{(i)} \\
\frac{\partial}{\partial b_j} J_{\mathrm{CE}}(\theta) &= -\frac{1}{m}\sum_{i=1}^{m} \left(\delta_{j,y^{(i)}} - h_{\theta,y^{(i)}}(\mathbf{x}^{(i)})\right)
\end{aligned}$$

Update rules for the weights and biases of the softmax layer

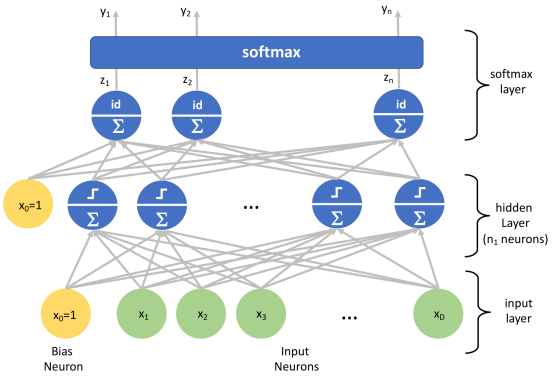
$$\begin{aligned}
\mathbf{w}_j &\leftarrow \mathbf{w}_j - \alpha \frac{\partial}{\partial \mathbf{w}_j} J_{\mathrm{CE}}(\theta) \\
b_j &\leftarrow b_j - \alpha \frac{\partial}{\partial b_j} J_{\mathrm{CE}}(\theta)
\end{aligned}$$

These rules apply also to the parameters of softmax layers when used as last layer of DL models.

8 Increasing the Capacity of Models

8.1 Adding One Hidden Layer

One additional layer with n_1 neurons using the sigmoid as the activation function. They are fully connected to the input and output neurons and posses a bias neuron (yellow in the graphic).



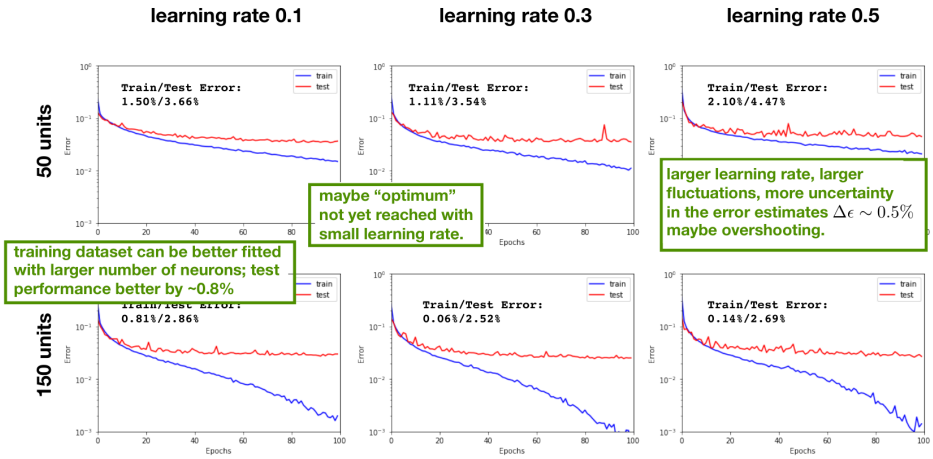


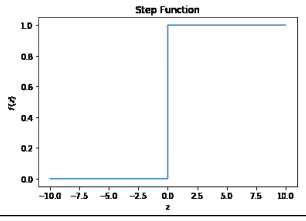
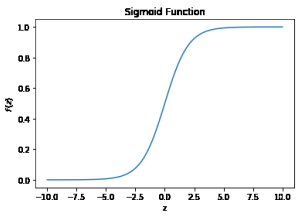
Figure 8.1: Training and Test results with a hidden layer on the MNIST set

8.1.1 Improvements by Adding One Hidden Layer

The error rate is reduced from 8.0% to 2.4% by only adding one hidden layer with sufficiently many neurons. By increasing the number of neurons in the hidden layer the performance improves but seems to plateau at 200 neurons and a 2.4% error rate. Further improvements could be made through tuning the hyper-parameters learning rate and number of epochs, using improved optimisation schemes and Convolutional Neural Nets. Adding more hidden layers does not seem to improve the performance, one explanation could be that the correlation between pixels is already captured through one single hidden layer and that there is not much more structural information to be captured for the MNIST dataset.

8.2 Role of the Activation Function

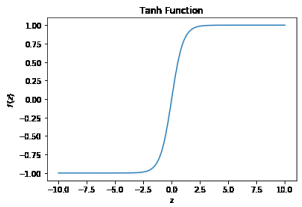
Non-linearities in the mapping between input and output of a neural network are **crucial for gaining sufficient representational capacity** for learning a task with sufficient accuracy. If the activation function is linear for all neurons in a neural network, the mapping function for the neural network is linear and the representational capacity very limited. The choice of activation functions also has an impact on the robustness and performance of the learning algorithm.

Function	Notes	Plot
Heaviside $f(z) = \begin{cases} 1 & (z \geq 0) \\ 0 & (z < 0) \end{cases}$	As in Rosenblatt's perceptron. Not differentiable. No practical use.	
Sigmoid $f(z) = \frac{1}{1+e^{-z}} = \frac{e^z}{e^z+1}$	Most commonly used in textbooks and in illustrative examples. In practice, typically used in output layers in binary classification. Smooth and differentiable. But saturation regions leading to vanishing gradients.	

Tanh

$f(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$

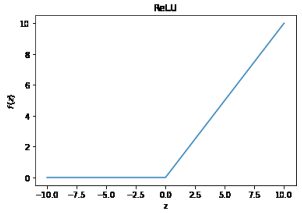
Often preferred over sigmoid since the output is centred around 0. Used in practice in LSTMs. Smooth and differentiable. Saturation regions leading to vanishing gradients.



Recti-Linear Unit

$f(z) = \max(0, z)$

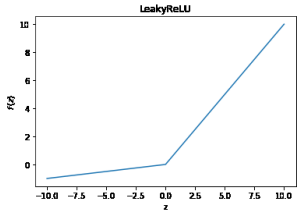
Used as de facto standard. Introduced to alleviate the vanishing gradient problem. Suffers from dying units problem for $z < 0$ where the activation and resulting gradient is 0.



Leaky Recti-Linear Unit

$f(z) = \max(\alpha \cdot z, z)$

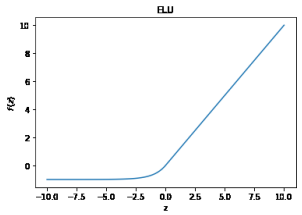
Alleviates both, the vanishing gradient problem and the dying units problem. Uses a small hyperparameter $0 < \alpha < 1$ which ensures that the unit never dies (typical default $\alpha = 0.01$).

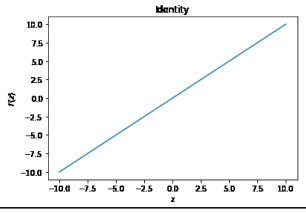
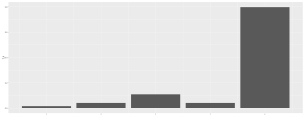


Exponential Linear Unit

$f(z) = \begin{cases} \alpha(e^z - 1) & (z < 0) \\ z & (z \geq 0) \end{cases}$

Similar to Leaky ReLU. Negative activation stabilises at $-\alpha$. Gradient vanishes at small negative values. More expensive to compute.



Identity $f(z) = z$	Used only in specific cases such as in the last layer of a network for performing a regression task.	
Softmax $f(\mathbf{z})_l = \frac{e^{z_l}}{\sum_{j=1}^{l} e^{z_j}}$	Used as last layer in a network for a classification task with l classes. Output vector can be interpreted as probability distribution.	

9 Universal Function Approximation

Neural networks provide models for predicting an outcome y from an input $\mathbf{x}$. In general, we expect to find only an approximate mapping $\mathbf{x} \to y$. The mapping is approximated by searching in a suitable parametric family of functions $h_\theta(\mathbf{x})$ with parameters $\boldsymbol{\theta}$

$$\mathbf{x} \to y = h(\mathbf{x}) \approx h_\theta(\mathbf{x}) = \widehat{y}$$

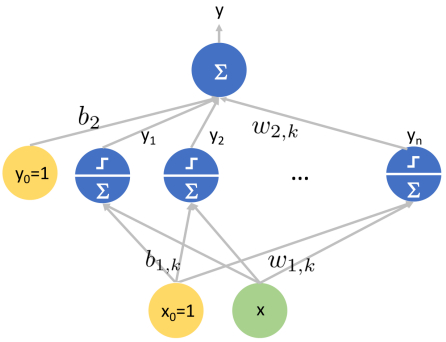
The richness of this family of functions drives the ability to represent more or less complex mappings. This ability is also referred to as *representational capacity*. For neural networks, the representational capacity is determined by the number of layers, type of layers, the number of neurons per layer, type of activation function, type of neuron.

9.1 Universal Approximation Theorem

Theorem 2 *A feedforward network with a linear output layer and at least one hidden layer with a non-linear ("squashing") activation function (e.g. sigmoid) can approximate a large class of functions $\mathbb{R}^{n_x} \to \mathbb{R}^{n_y}$ with arbitrary accuracy - provided that the network is given a sufficient number of hidden units and the parameters are suitably chosen.*

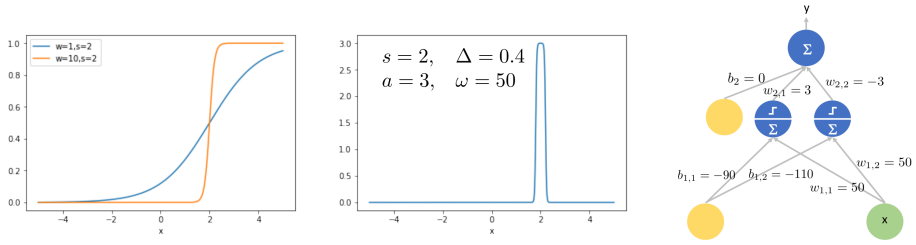
One-dimensional real-valued input x. One hidden layer with n neurons and sigmoid activation function, weights $w_{1,k}$ and bias $b_{1,k}$, one linear output layer with weights $w_{2,k}$ and biases $b_{2,k}$. The linear output layer leads to a linear combination of sigmoids

$$h_\theta(x) = \sum_{k=1}^{n} w_{2,k} \cdot \sigma(w_{1,k} \cdot x + b_{1,k}) + b_2$$



The theorem states that any quite general function ($\mathbb{R}^1 \to \mathbb{R}^1$) can be approximated if suitable parameters and a suitable number of neuron n are chosen.

For convenience the sigmoids with the position of the ste $s = -\frac{b}{w}$ are re-parametrised so that $\sigma(w \cdot x + b) = \sigma(w \cdot (x - s))$. With the parameter w the slope of the step is controlled, that is the larger w the steeper the step, as indicated in the figure on the left.



By combining two such step functions a peak at location s with width Δ is easily created

$$\phi(x; s, a, \Delta, w) = a \left(\sigma(w \cdot (x - s + \frac{\Delta}{2})) - \sigma(w \cdot (x - s - \frac{\Delta}{2})) \right)$$

9.1.1 Generalisation to Arbitrary Functions

The scheme for $1d$ input and $1d$ output can easily be generalised to nd input and md output.

Problems For improving the accuracy when modelling with step-functions, more and more neurons are needed and many parameters need to be determined. In problems with high-dimensional input, an exponentially growing number of neurons is needed.

Accordingly, more data sampled on a sufficiently fine grid is needed. With growing dimensionality in the input this becomes infeasible. This is known as curse of dimensionality.

If just the available data is used and the data between points is interpolated with step functions, the training data is significantly overfitted.

The theorem does not provide a scheme for efficiently learning the parameters from available limited data.

10 Overfitting and Generalisation

10.1 Overfitting

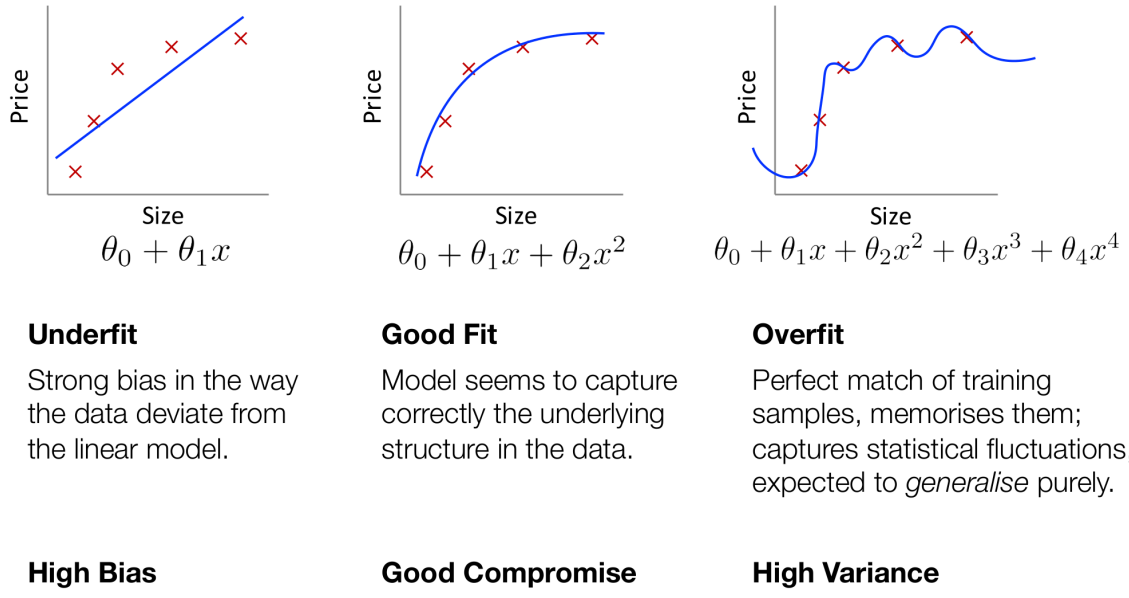


Figure 10.1: An example of data being progressively more overfitted

Guiding Principle Occam's Razor: Among competing hypothesis that explain known observations equally well the simplest is the best.

10.1.1 Characterisation of Overfitting

Overfitting occurs when the learned hypothesis (trained model) fits the training data set very well, but fails to generalise to new examples. Overfitting can occur when

- the number of parameters of the model is too large, that is the model being too flexible
- the training set is too small in comparison with the dimensionality of the input data, which introduces sampling noise
- the training set is too noisy

The problem with overfitting is, that the model is likely to detect patterns in the noise itself. These patterns are unlikely to generalise to previously unobserved new inputs. A machine learning algorithm to perform well should thus be able to make the

1. training error, also called bias error, small
2. gap between training and test error, also called variance error, small
3. bias and variance error comparable in magnitude

How to Examine Overfitting Evaluate the performance on the training and the test set for different models with different model complexity, by using different training set sizes and after different numbers of training epochs.

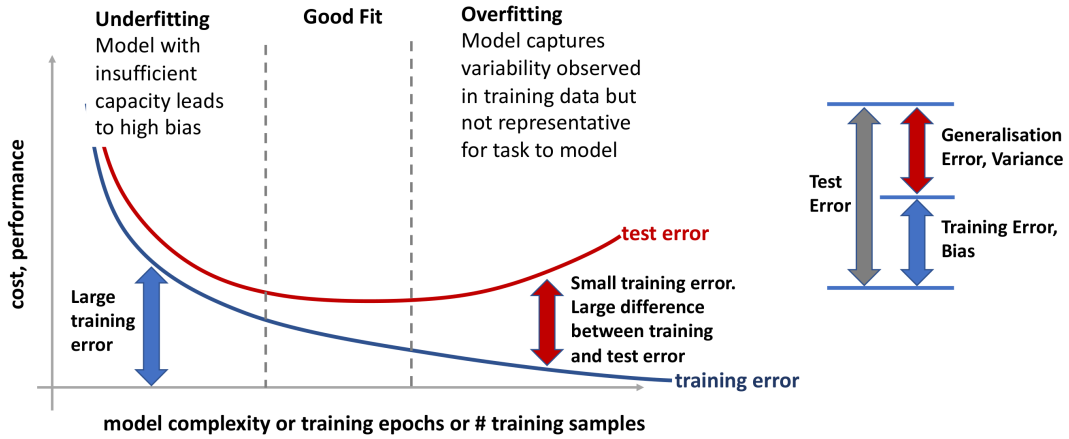


Figure 10.2: Strategies to reduce overfitting

10.2 Model Selection

The goal of model selection is to select the model with the best performance from a family of models. To achieve this the performance for different model with differing complexity needs to be evaluated, but to avoid overfitting on the test set, we must not use information from the test set to determine the hyper-parameters of the model.

The solution is to further split the original training set into two subsets, a training set used for training the models, and a validation set used for evaluating the performance and selection of the model and hyper-parameters.

10.2.1 Hyper-Parameters

Hyper-Parameters specify higher level properties of the mapping function (model) and the learning process. These parameters are not trainable (not determined by optimising the cost on the training set), but optimised on the validation set.

Examples of hyper-parameters are

- Learning Rate
- Batch Size
- Number of hidden layers
- Number of neurons in a layer
- Number of epochs

10.3 Training, Validation, Test Split

		typical split ratio	
		small datasets	large datasets
Training Set	Subset of *trustable* data used to train the model	60%	98%
Validation Set	Subset of *trustable* data used to select models, for example by selecting the best hyper-parameters of the model	20%	1%
Test Set	Subset of *trustable* data used to measure the performance of the finally selected model	20%	1%

Trustable means in this context

- composed of data acquired under the same conditions
- data with the same characteristics (range of value, distribution, sparseness)
- dataset sufficiently large to have confidence in the parameter estimates or evaluation metrics

10.3.1 Selecting a Split Ratio

Considering different split ratios for a given fixed dataset, it is expected that

- the **training cost or training error** increases with a higher split ratio
  it gets more difficult to perfectly fit a model with more data
- the **validation cost or validation error** decreases with a higher split ratio, the validation becomes very wiggly for large split ratios
  the model trained with more data is expected to capture more details about the underlying problem

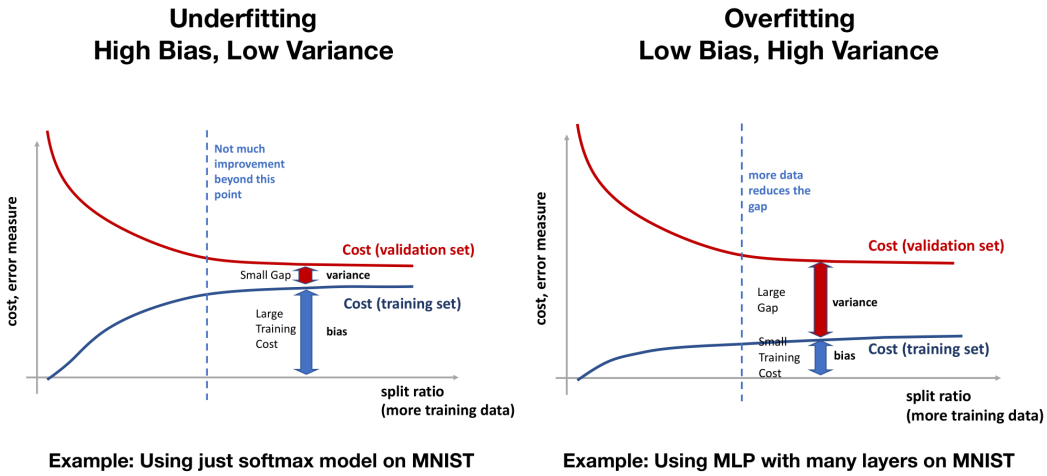


Figure 10.3: Analysis of different learning curves

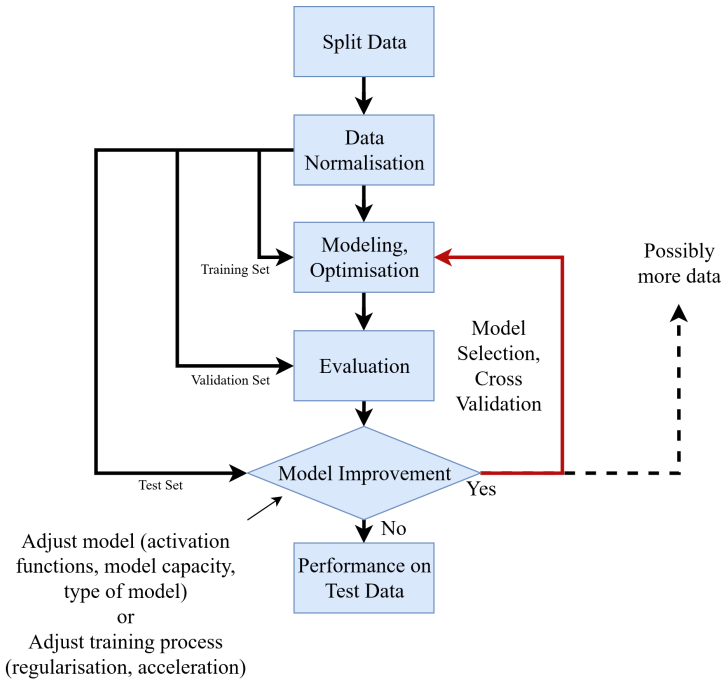


Figure 10.4: The general process used in Machine Learning

10.4 k-fold Cross Validation

Procedure for making efficient use of the data for more robust estimation of validation set performance, particularly useful if data is scarce.

Algorithm

- Randomly shuffle the dataset.
- Split dataset into test set (D_{test}) and the rest (D_0)
- Split D_0 into k equally sized folds. *Typical choice* $k = \{5, 10\}$

- For each fold:
  - Use the selected fold as validation set and the remaining folds as training set.
  - Fit the model on the remaining folds.
  - Evaluate the fitted model with the selected validation fold.
  - Retain the evaluation score (performance)
- Report the performance as average of retained evaluation scores.

Integrated into Model Selection Process

- Select the model, the hyper-parameters leading to the best performance
- Train with the combined training and validation set
- Report its performance on the basis of the test set

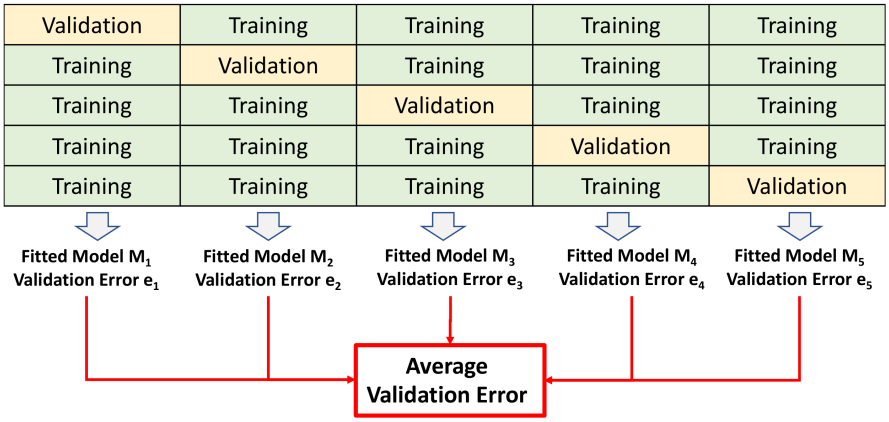
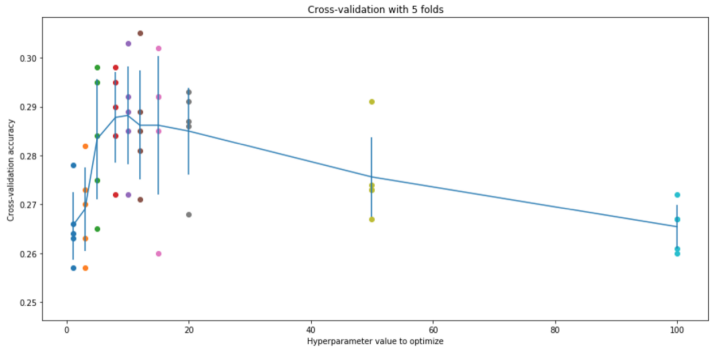


Figure 10.5: Visualisation of k-fold cross validation

10.4.1 Benefits of Cross Validation



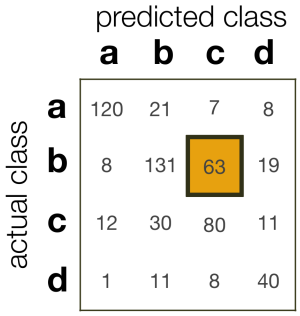
For selected hyper-parameter values, the performance for each of the five folds is computed. The spread on a given hyper-parameter value shows the sensitivity of the performance metric on the split of the data and it gives a feeling about the confidence of the performance estimates.

11 Performance Measures

11.1 Confusion Matrix

A confusion matrix measures the test performance of a classification system on a per-class basis by indicating the number of samples of actual class a predicted as class *b*. The rows relate to the actual class labels a and the columns to the predicted class labels *b*.

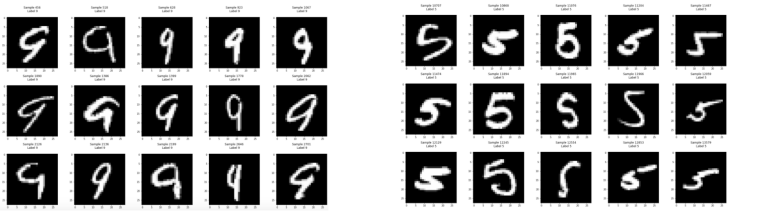
In the example, the orange box indicates that 63 samples of total 221 of actual class *b* (8+131+63+19) are predicted as class *c*.




```
array([[1336,    0,    2,    3,    5,    7,   13,    3,    9,    2],
       [   1, 1575,   11,    7,    1,    7,    2,    3,   23,    2],
       [   8,    9, 1290,   23,   15,    5,   16,   19,   43,    5],
       [   8,    4,   31, 1258,    2,   41,    3,   23,   47,   14],
       [   2,    8,   10,    1, 1244,    1,    8,    7,   21,   26],
       [  12,    9,    4,   40,   22, 1091,   33,   11,   65,   10],
       [   9,    1,    6,    0,   18,   13, 1275,    0,    9,    0],
       [   5,    4,   10,    7,   16,    1,    1, 1365,    5,   30],
       [   9,   19,   11,   22,   11,   22,   10,    5, 1227,   15],
       [   6,   10,    2,   19,   60,    8,    1,   53,   20, 1194]])
```

Actual class '9', predicted as '4'

Actual class '5', predicted as '8'



11.2 Overall Accuracy and Error Rate

Accuracy percentage of correct classifications

$$\text{accuracy} = \frac{\sum \text{diagonal elements}}{\text{number of samples}}$$

Error Rate percentage of wrong classifications

$$\text{error rate} = 1 - \text{accuracy}$$

11.3 Confusion Table

A **confusion table** is used to measure the classification performance of a two-class system.

		Predicted		
		Positive	Negative	*Total*
Actual	Positive	**TP**	FN	(TP+FN)
	Negative	FP	**TN**	(FP+TN)
	Total	(TP+FP)	(FN+TN)	N

11.4 Class Accuracy

Class Accuracy percentage of correct classifications considering a given class against the others

$$\frac{TP + TN}{Total}$$

Example Digit '5'

Actual \ Predicted	P	N	Total
P	1091	206	1297
N	105	12598	12703
Total	1196	12804	14000

Class accuracy digit '5':
(1'091 + 12'598) / 14'000
= 97.80%

11.5 Per Class Sensitivity or Recall

Class Sensitivity or Recall percentage of correct classification for a given class

$$\frac{TP}{TP + FN}$$

Example Digit '5'

Actual \ Predicted	P	N	Total
P	1091	206	1297
N	105	12598	12703
Total	1196	12804	14000

Class sensitivity digit '5':
1091 / (1091 + 206)
84.1%

11.6 Class Precision

Class Precision percentage of correct classification in the predicted outputs for a given class

$$\frac{TP}{TP+FP}$$

Example Digit '5'

Actual \ Predicted	P	N	Total
P	1091	206	1297
N	105	12598	12703
Total	1196	12804	14000

Class precision for digit '5':
1091 / (1091 + 105)
= 91.8%

11.7 F-Score

F-Score or F1-Score harmonic mean of recall and precision

$$\frac{1}{\frac{1}{2}\left(\frac{1}{\text{recall}}+\frac{1}{\text{precision}}\right)} = \frac{TP}{TP+\frac{FP+FN}{2}}$$

Example Digit '5'

Actual \ Predicted	P	N	Total
P	1091	206	1297
N	105	12598	12703
Total	1196	12804	14000

F1-score for digit '5':
1091 / (1091+ (206+105)/2)
precision = 86.34%, F1 = 87.08%

11.8 System Performance Metrics

System Accuracy	Average Class Accuracy
System Recall	Average Class Recall
System Precision	Average Class Precision
System F1-Score	Harmonic Mean of System Precision and System Recall

12 Curse of Dimensionality

In classical machine learning there is a local smoothness assumption of the function the model tries to learn. This means that this function does not vary much locally and stays approximately constant in small regions. This assumption allows to generalise to variations in small regions from closely located training examples. Training examples are needed in all the regions of interest. If no or only few examples are available in a given region, no confident predictions can be made. This assumption works fine if the dimensionality of the data is not too high, but fails in typical Deep Learning applications when dealing with image, speech or language data.

"...when the dimensionality increases, the volume of the space increases so fast that the available data become sparse. This sparsity is problematic for any method that requires statistical significance. In order to obtain a statistically sound and reliable result, the amount of

data needed to support the result often grows exponentially with the dimensionality."

12.1 Representation of Variational Characteristics

Goal is to find a good representation for the variational characteristics observed in the data as needed by the underlying task. Typically, the patterns found in the data and its variational degrees of freedom are expected to be well represented on a much lower dimensional manifold.

Theorem 3 *The **core idea in Deep Learning** is*

- *the assumption that the data was generated by the composition of factors or features, potentially at multiple levels in a hierarchy*
- *learning involves discovering a set of underlying factors of variation that can be described in terms of other, simpler underlying factors arranged in a hierarchy*

13 Computational Graphs

A computational graph is a directed graph where

- nodes correspond to operations or input variables
- edges correspond to inputs of an operation which can originate from input variables or outputs of other operations.

Two types of input variables: Input data and model parameters.

$$g(x; w, b) = \sigma(w \cdot x + b)$$

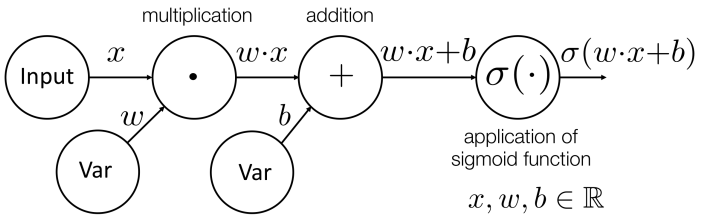


Figure 13.1: Example of a computational graph

Neural networks can be considered as a special form of computational graphs. Inputs into and outputs out of the nodes are multi-dimensional arrays called *tensors*: scalars, 1d vectors, 2d matrices and higher rank objects.

$$g(\mathbf{x}; \mathbf{W}, \mathbf{b}) = \sigma(\mathbf{W} \cdot \mathbf{x} + \mathbf{b})$$

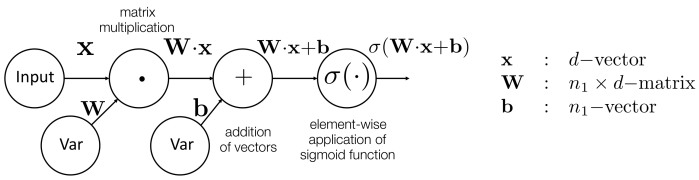


Figure 13.2: Computational graph for a network with just an output layer

The operations of a single layer (affine transformation and application of activation function) can be collapsed into a single node. Layers are the abstraction level most Deep Learning frameworks use to define the architecture of a Deep Learning model.

13.1 Multi-Layer Perceptron

Represents function that computes from a sample $\mathbf{x}^{(i)}$ of $n_x = n_0$ input features (represented by an n_x-vector) an output vector $\hat{\mathbf{y}}^{(i)}$.

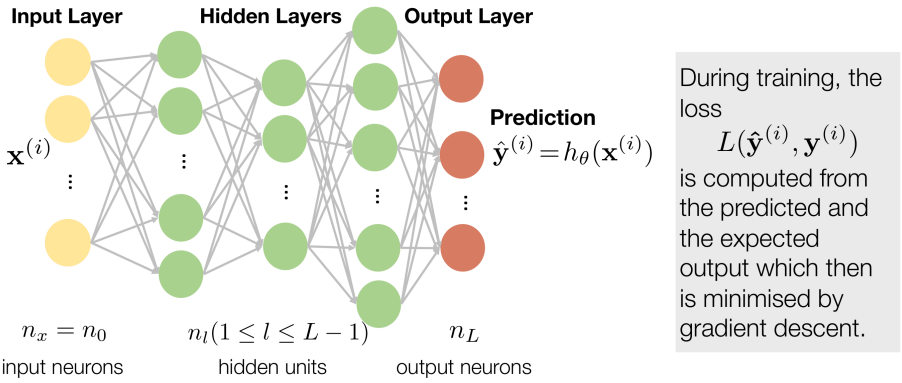
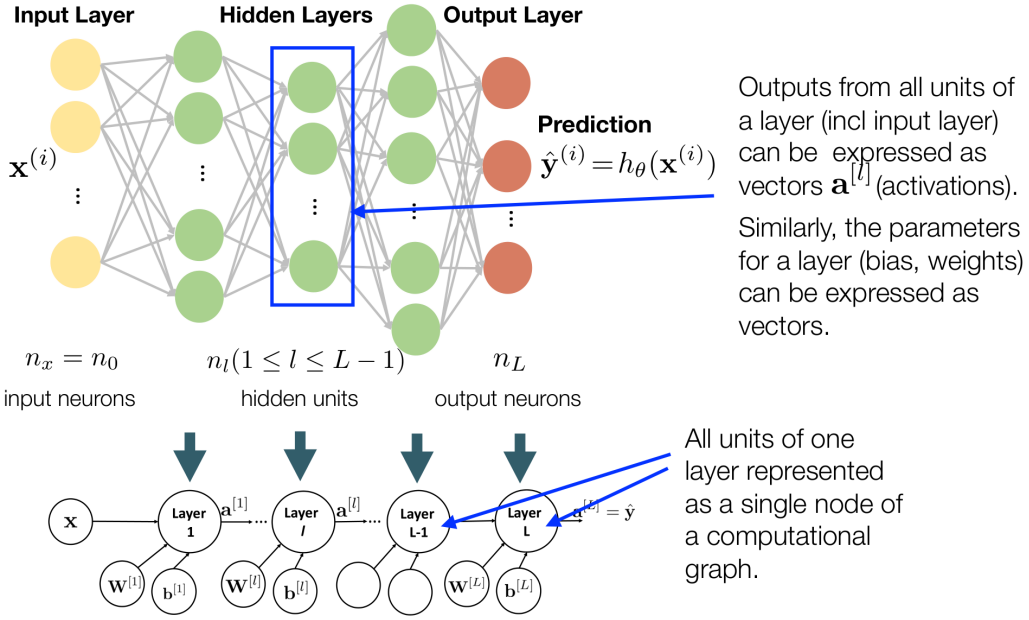
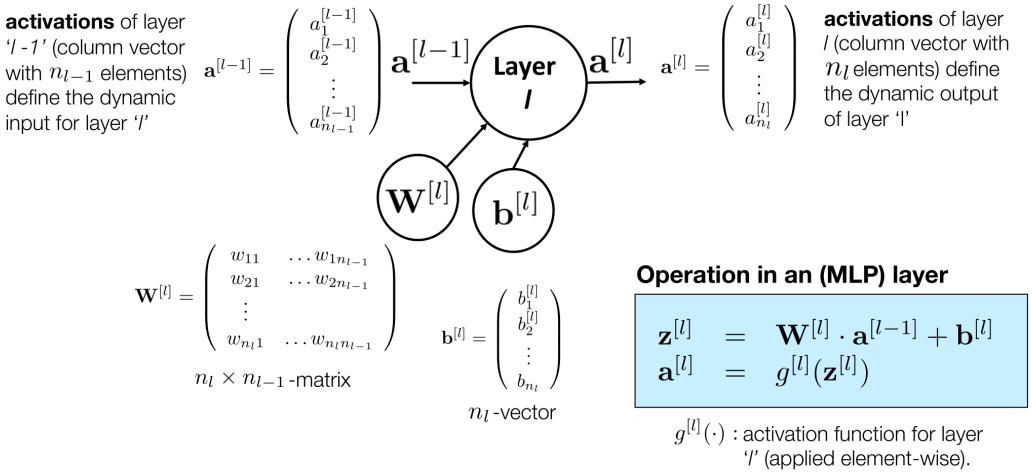


Figure 13.3: The nodes represent neural units which operate on output originating from many other units and produce scalar output

13.2 Unit-/Layer-Wise Representation



13.3 Notations for Layer in an MLP



Forward Propagation chains the calculations of activations in a MLP.

$$
\begin{aligned}
z^{[1]} &= \mathbf{W}^{[1]} \cdot \mathbf{a}^{[0]} + \mathbf{b}^{[1]} \\
\mathbf{a}^{[1]} &= g^{[1]}(\mathbf{z}^{[1]}) \\
z^{[2]} &= \mathbf{W}^{[2]} \cdot \mathbf{a}^{[1]} + \mathbf{b}^{[2]} \\
\mathbf{a}^{[2]} &= g^{[2]}(\mathbf{z}^{[2]}) \\
&\vdots \\
z^{[L]} &= \mathbf{W}^{[L]} \cdot \mathbf{a}^{[L-1]} + \mathbf{b}^{[L]} \\
\mathbf{a}^{[L]} &= g^{[L]}(\mathbf{z}^{[L]}) \\
L(\hat{\mathbf{y}}, \mathbf{y}) &= \mathbf{a}^{[L]}
\end{aligned}
$$

13.4 Computing the Output of the MLP with Vectors

Vectorise over the samples in a (mini-)batch. This will allow for efficient parallelisation on GPU. Put the input samples $\mathbf{x}^{(i)}$ as columns in a $n_x \times m$ - matrix:

$$
\mathbf{X} = \mathbf{A}^{[0]} = \begin{pmatrix} \vdots & \vdots & & \vdots \\ \mathbf{x}^{(1)} & \mathbf{x}^{(2)} & \cdots & \mathbf{x}^{(m)} \\ \vdots & \vdots & & \vdots \end{pmatrix}
$$

The same for the activations and logits in the layer $l = 1, ..., L$ ($n_l \times m$ - matrices)

$$
\mathbf{A}^{[l]} = \begin{pmatrix} \vdots & \vdots & & \vdots \\ \mathbf{a}^{[l](1)} & \mathbf{a}^{[l](2)} & \cdots & \mathbf{a}^{[l](m)} \\ \vdots & \vdots & & \vdots \end{pmatrix}
$$

$$
\mathbf{Z}^{[l]} = \begin{pmatrix} \vdots & \vdots & & \vdots \\ \mathbf{z}^{[l](1)} & \mathbf{z}^{[l](2)} & \cdots & \mathbf{z}^{[l](m)} \\ \vdots & \vdots & & \vdots \end{pmatrix}
$$

Then the equations for layer $l = 1, ..., L$ can then be written in a very compact manner

$$\mathbf{Z}^{[l]} = \mathbf{W}^{[l]} \cdot \mathbf{A}^{[l-1]} + \mathbf{b}^{[l]}$$
$$\mathbf{A}^{[l]} = g^{[l]}(\mathbf{Z}^{[l]})$$

13.5 Chain Rule of Calculus

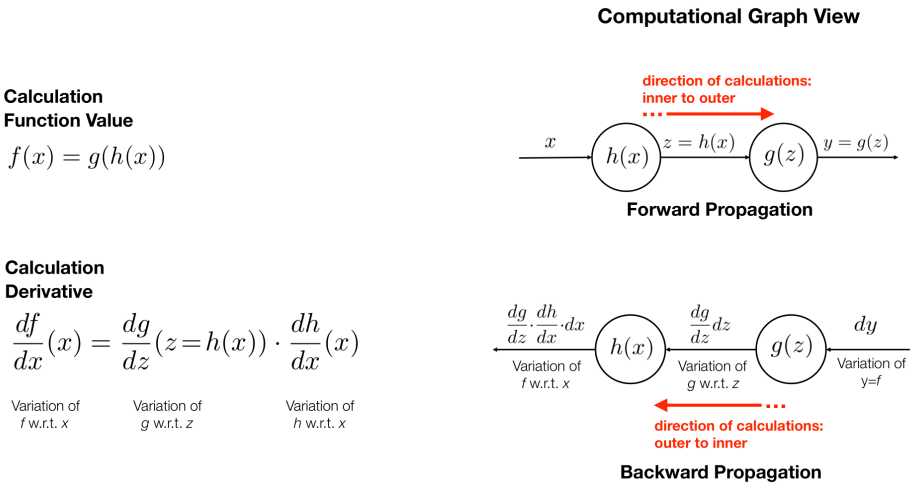
Rule to compute the derivative of a function with a nested structure, represented as a chain of function evaluations. According to the chain rule, the change is given by the product of the changes in the nested components. Thus a progressive calculation of inner derivatives can be done

$$\begin{aligned} L(x) &= f(g(h(k(x)))) \\ \underbrace{\frac{dL}{}}_{\text{change in target}} &= \frac{\partial f}{\partial g} \cdot \frac{\partial g}{\partial h} \cdot \frac{\partial h}{\partial k} \cdot \frac{\partial k}{\partial x} \cdot \underbrace{dx}_{\text{change in input variable}} \\ &= f'(g(h(k(x)))) \times g'(h(k(x))) \times h'(k(x)) \times k'(x) \end{aligned}$$

13.5.1 Example of the Chain Rule

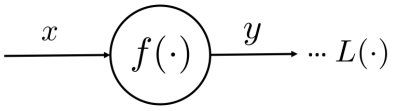
$$\begin{aligned} f(x) &= \sqrt{1+x^2}, \quad g(z) = \sqrt{z}, \quad h(x) = 1 + x^2 \\ f(x) &= g(h(x)) \\ \frac{df}{dx}(x) &= \underbrace{\frac{dg}{dz}(z = h(x))}_{\frac{dg}{dz}(z)=\frac{1}{2\sqrt{z}}} \cdot \underbrace{\frac{dh}{dx}(x)}_{\frac{dh}{dx}(x)=2x} \\ \frac{df}{dx}(x) &= \frac{dg}{dz}(z) \cdot \frac{dh}{dx} = \frac{1}{2\sqrt{z}} \cdot 2x \overset{z=h(x)}{=} \frac{1}{2\sqrt{1+x^2}} \cdot 2x = \frac{x}{\sqrt{1+x^2}} \end{aligned}$$

13.6 Chain Rule in regards to the Computational Graph

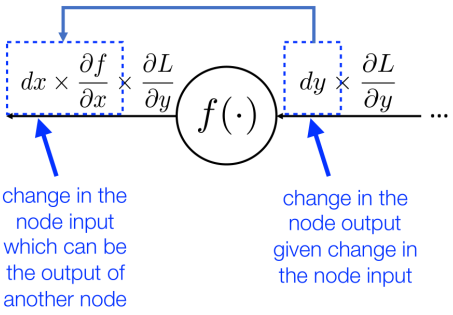


13.7 Backpropagation Contribution per Node

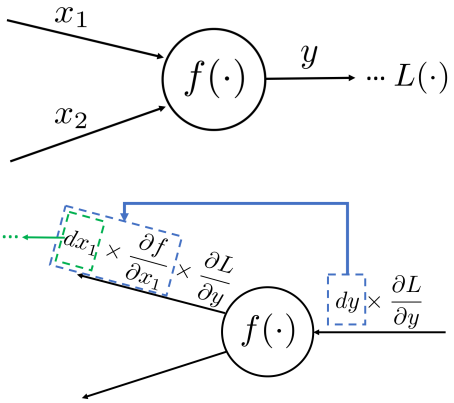
Steps to compute the change of a function represented by a computational graph with regard to the input variable (derivative):



Start at the end of the computational graph and progressively walk back through the graph and initialise the result with 1.
At each node, multiply the result obtained so far with the derivative of the node function with regard to the node input.



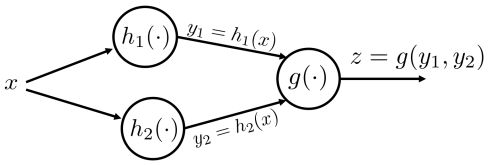
If the node has multivariate input, then compute the change in the output with regard to the change in each input variable (partial derivarive) separately. Then compute the change in the input variable by further traversing the graph structure in backwards direction.



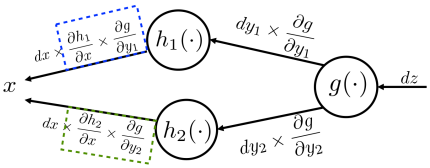
If several paths within the computational graph trace back to the same node (input or intermediary), the contributions from the different paths are summed.

Example

$z = g(h_1(x), h_2(x))$

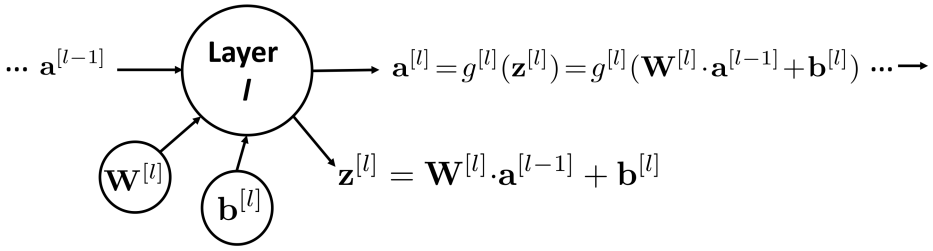


$$dz = dx \times \left(\frac{\partial h_1}{\partial x} \times \frac{\partial g}{\partial y_1} + \frac{\partial h_2}{\partial x} \times \frac{\partial g}{\partial y_2} \right)$$

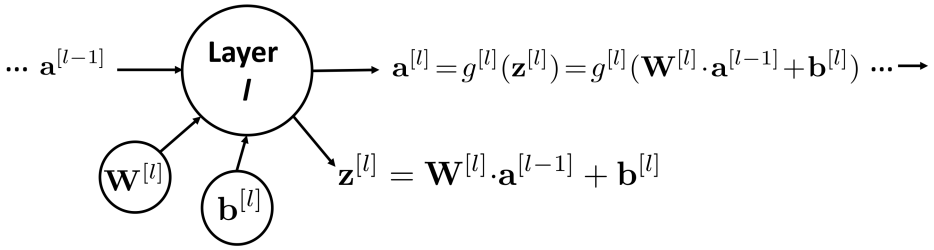


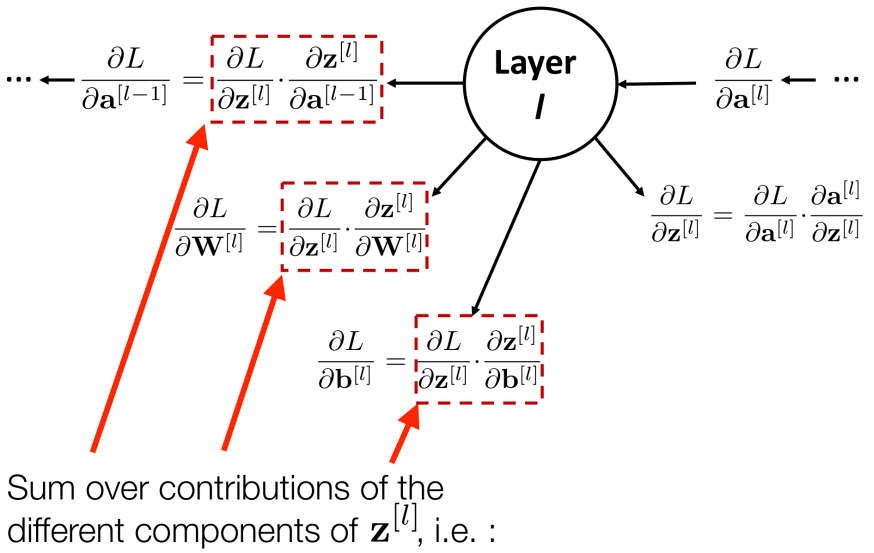
13.8 Backpropagation Through a Single Layer

Forward Propagation Layer represented by a single node



Backward Propagation Proper handling of tensor indices ignored





$$\frac{\partial L}{\partial a_k^{[l-1]}} = \sum_j \frac{\partial L}{\partial z_j^{[l]}} \frac{\partial z_j^{[l]}}{\partial a_k^{[l]}} \qquad \frac{\partial L}{\partial W_{ki}^{[l]}} = \sum_j \frac{\partial L}{\partial z_j^{[l]}} \frac{\partial z_j^{[l]}}{\partial W_{ki}^{[l]}} \qquad \frac{\partial L}{\partial b_k^{[l]}} = \sum_j \frac{\partial L}{\partial z_j^{[l]}} \frac{\partial z_j^{[l]}}{\partial b_k^{[l]}}$$

13.9 Derivatives for Chosen Layers

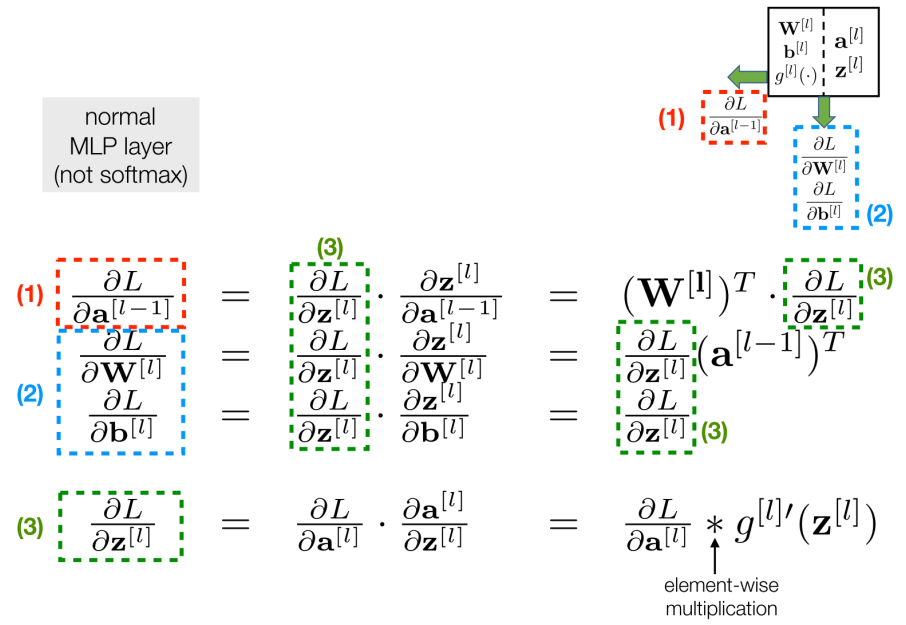


Figure 13.4: Derivatives for a normal layer (not softmax or cross-entropy)

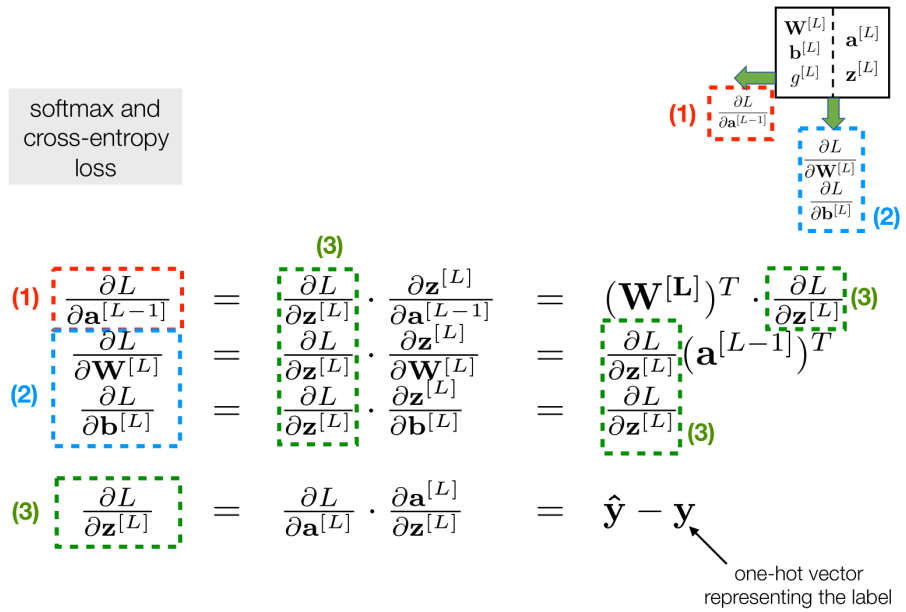


Figure 13.5: Derivatives for softmax and cross-entropy layers

13.10 Vectorised Formulation of Forward and Backward Propagation

Forward Propagation

$$\mathbf{Z}^{[L]} = \mathbf{W}^{[l]} \cdot \mathbf{A}^{[l-1]} + \mathbf{b}^{[l]}$$
$$\mathbf{A}^{[l]} = g^{[l]}\mathbf{Z}^{[l]}$$

Backward Propagation

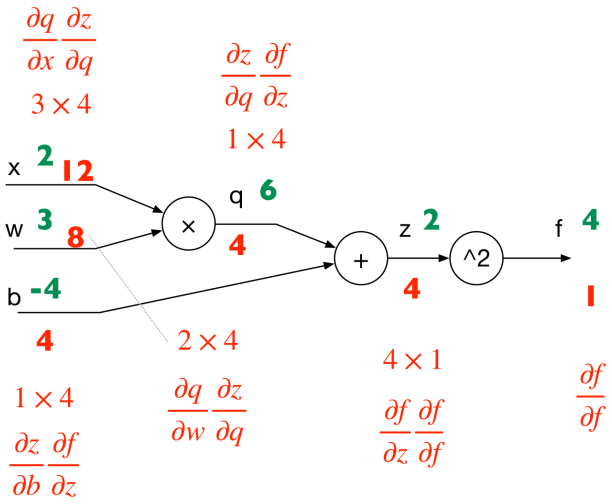
$$\frac{\partial L}{\partial \mathbf{A}^{[l-1]}} = \frac{\partial L}{\partial \mathbf{Z}^{[l]}} \frac{\mathbf{Z}^{[l]}}{\partial \mathbf{A}^{[l-1]}} = (\mathbf{W}^{[l]})^T \cdot \frac{\partial L}{\partial \mathbf{Z}^{[l]}}$$
$$\frac{\partial L}{\partial \mathbf{W}^{[l]}} = \frac{\partial L}{\partial \mathbf{Z}^{[l]}} \cdot \frac{\partial \mathbf{Z}^{[l]}}{\partial \mathbf{W}^{[l]}} = \frac{\partial L}{\partial \mathbf{Z}^{[l]}} (\mathbf{A}^{[l-1]})^T$$
$$\frac{\partial L}{\partial \mathbf{b}^{[l]}} = \frac{\partial L}{\partial \mathbf{Z}^{[l]}} \cdot \frac{\partial \mathbf{Z}^{[l]}}{\partial \mathbf{b}^{[l]}} = \text{sum}\left(\frac{\partial L}{\partial \mathbf{Z}^{[l]}}, \text{axis} = 1\right)$$
$$\frac{\partial L}{\partial \mathbf{Z}^{[l]}} = \frac{\partial L}{\partial \mathbf{A}^{[l]}} \cdot \frac{\partial \mathbf{A}^{[l]}}{\partial \mathbf{Z}^{[l]}} = \frac{\partial L}{\partial \mathbf{A}^{[l]}} * g^{[l]\prime}(\mathbf{Z}^{[l]})$$

$*$: element-wise multiplication

13.10.1 A Simple Numerical Example

$$f = (wx + b)^2$$

$$f = z^2 \quad \frac{\partial f}{\partial z} = 2z$$
$$z = q + b \quad \frac{\partial z}{\partial q} = 1, \frac{\partial z}{\partial b} = 1$$
$$q = wx \quad \frac{\partial q}{\partial x} = w, \frac{\partial q}{\partial w} = x$$



13.11 Advantages of Computational Graphs

1. Intuitive interpretation of gradient backpropagation
2. We can easily define new nodes using the forward/backward pattern
3. Node composition or factorisation: any complex learning architecture can be composed from atomic nodes. No need to compute complex global gradient
4. The loss functions can actually be seen as extra nodes in the graph

Meta-nodes like the Sigmoid can be composed of multiple atomic nodes, or equivalently, a sub-graph composed of nodes can be re-implemented in a single node if an analytic form of the gradients can be computed.

14 Keras

Keras is a high-level open-source neural networks API, written in Python and capable of running on top of **TensorFlow**, **CNTK**, or **Theano**. It was developed with a focus on enabling fast experimentation.

A **model** is the way Keras organises the layers of neurons, it is differentiated between the sequential and the functional model. The sequential model corresponds to a regular stack of feed-forward layers. The functional API allows to define **graphs** of layers and is used to construct non-sequential architectures. It is typically used to allow for independent networks to diverge or merge.

Dense layers represent fully connected layers of neurons in Keras, they are added as follows:

14.0.1 Complex Example

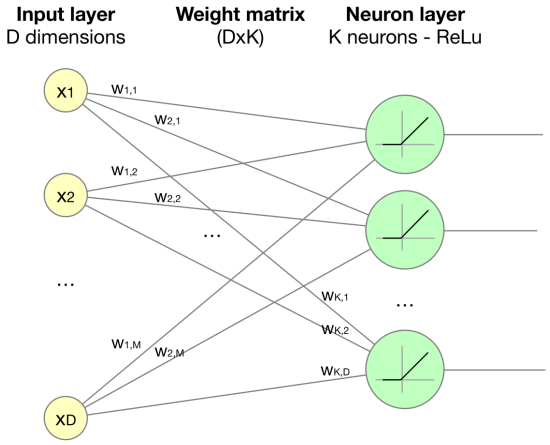


Figure 14.1: The model in the example

14.1 Functional API of Keras

The **sequential** model of Keras corresponds to a regular stack of layers while the **functional** API is used for non sequential architectures. The functional API allows for multiple paths in the computational graph through layer sharing, multiple inputs, and multiple outputs.

Unlike the Sequential model, one must create and define a stand-alone Input layer that specifies the shape of input data. The input layer takes a shape argument that is a tuple that indicates the dimensionality of the input data. When input data is one-dimensional, such as for a MLP, the shape must explicitly leave room for the shape of the mini-batch size. Therefore, the shape tuple is always defined with a hanging last dimension when the input is one-dimensional.

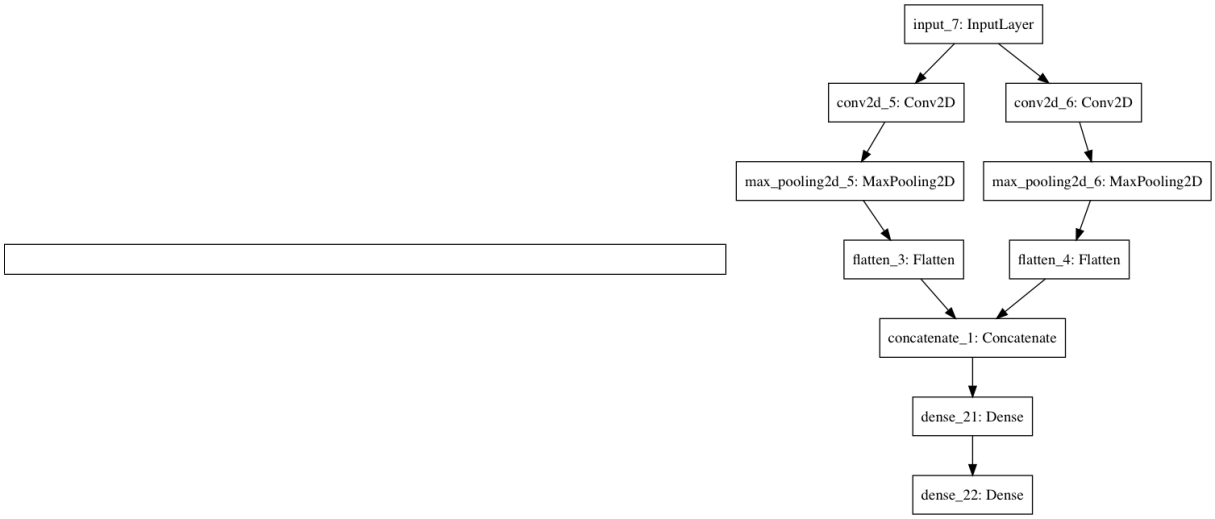
The layers in the model are connected pairwise. This is done by specifying where the input comes from when defining each new layer. A callable syntax is used, such that once the layer is created, the layer from which the input is taken is specified through a call.

A Model is declared with its inputs and its outputs that are both tensors defining the inputs and outputs of the computational graph.

14.1.1 Python syntax

Objects are callable when the class definition includes a method. It allows to call the object as we would call a method with the syntax .

14.1.2 CNN Multiple Paths Example



14.1.3 Multiple Inputs

Such graphs are useful to merge different modalities, such as multiple images of the same object, stereo speech recordings, or different parameters of an acquisition device.

14.1.4 Callbacks

Callback are functions to be applied at given stages of the training procedure. Callbacks can be used to get a view on internal states and statistics of the model during training. Some are automatically applied in the fit() method such as BaseLogger and History.

14.1.5 Best Practices

- **Use Consistent Variable Names** Use same variable name for input (visible) and output layers (output), hidden layers (hidden1, hidden2)
- **Review Layer Summary** Always print the model summary and review the layer outputs to ensure that the model was connected together as you expected
- **Review Graph Plots** Create a plot of the model graph and review it to ensure that everything was put together as you intended
- **Name the layers** Assign names to layers that are used when reviewing summaries and plots of the model graph
- **Use Callbacks** Use callbacks to save the best models from epochs to epochs as a safety against interrupt and overfitting

15 Regularisation and Faster Optimisers

15.1 Vanishing and Exploding Gradients

There were some difficulties in training deep neural nets in 2007 with the given techniques that were available. These difficulties were mitigated through various improvements, important contributions were made from Glorot, Bengio, Ioffe, Szegedy and others.

These aim at avoiding the saturation regions of the activation functions at early stages of the training by stabilising the variance when propagating forward and backward, and choosing more suitable activation functions that differ from the sigmoid.

In the so-called saturation regions the gradients are close to zero and become even smaller when they are back-propagated. This means that learning comes to a standstill. The suggested solution was to use suitably initialised parameters with the "Glorot" or "Xavier" initialisation and a non-sigmoidal activation function.

15.1.1 Reason for Vanishing Gradients

When backpropagating through the neural network the chain rule of calculus is used, this means we have a product term with derivatives of the neuron's activation function.

$$\frac{\partial L}{\partial w^{[l]}} = a^{[l-1]} \cdot g'\left(z^{[l]}\right) \cdot \left(\prod_{k=l+1}^{L} w^{[k]} g'(z^{[k]})\right) \cdot \frac{\partial L}{\partial a^{[l]}}$$

$$\frac{\partial L}{\partial b^{[l]}} = g'\left(z^{[l]}\right) \cdot \left(\prod_{k=l+1}^{L} w^{[k]} g'(z^{[k]})\right) \cdot \frac{\partial L}{\partial a^{[l]}}$$

This product term is responsible for gradients to shrink to zero or becoming too large.

$$\sigma'(z) = \sigma(z)(1 - \sigma(z))$$

This product term gets small or very small for large values of z (label correct or completely wrong). The problem can be alleviated through

- **Parameter Initialisation** Properly initialise the weights so that the logits z do not grow too large in magnitude
- **Batch Normalisation, Gradient Clipping** Make sure that the weights do not grow too large in magnitude during training
- **Suitable Activation Functions** Use activation functions that cannot saturate

15.2 Xavier/He Initialisation

- **Randomly initialise weights** to break symmetries at start of the learning
- **Initialise weights at proper scale** to keep the variance stable across layers, ideally in forward and backward propagation. Properly scaling the width of the distribution depends on the random distribution used to generate the weights and activation function (refer to figure **??**).

Activation function	Uniform distribution [–r, r]	Normal distribution
Logistic	$r = \sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = \sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$
Hyperbolic tangent	$r = 4\sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = 4\sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$
ReLU (and its variants)	$r = \sqrt{2}\sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = \sqrt{2}\sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$

Figure 15.1: Different distribution for activation functions

15.2.1 Parameter Initialisation in Keras

Figure 15.2: Example initialisation of parameters in Keras

15.3 Batch Normalisation

Idea introduced by Ioffe and Szegedy to normalise the input to each layer per mini-batch just before applying the activation function, estimate mean and standard deviation used for the normalisation from the current mini-batch. Afterwards, let the model learn the optimal scale and mean of the inputs for each layer.

It proves very effective for improving the stability of the training, and significantly reduces the problem of coordinating updates across many layers. It also limits the amount to which parameter updates in earlier layers can affect the distribution of values the later layers see, and can be applied to any input or hidden layer in the network.

Ioffe and Szegedy recommend to apply batch normalisation to the inputs to the activation function, that is the logits $z_k^{[l]}$.

Normalisation per Mini-Batch

$$\left(Z_{\text{norm}}^{\{r\},[l]}\right)_{k,i} = \frac{Z_{k,i}^{\{r\},[l]} - \mu_k^{\{r\},[l]}}{\sigma_k^{\{r\},[l]} + \epsilon}$$

Scaling and Shifting

$$\left(\widehat{Z}_{\text{norm}}^{\{r\},[l]}\right)_{k,i} = \gamma_k^{[l]} \left(Z_{\text{norm}}^{\{r\},[l]}\right)_{k,i} + \beta_k^{[l]}$$

$\gamma_k^{[l]}$ and $\beta_k^{[l]}$ are parameters to be learned and optimised.

15.4 Non-Saturating Activation Functions

Avoid S-shaped activation functions that flatten out at larger z-magnitudes, such as ReLU, LeakyReLU or ELU

15.5 Gradient Clipping

Counter *exploding gradients* by simply clipping gradients in length during back-propagation. Intuition provided by I. Goodfellow: Deep networks often have extremely steep regions resembling

cliffs - as a result of the composite structure of the operations where weights get multiplied. At the cliffs, the gradient can be become very large.

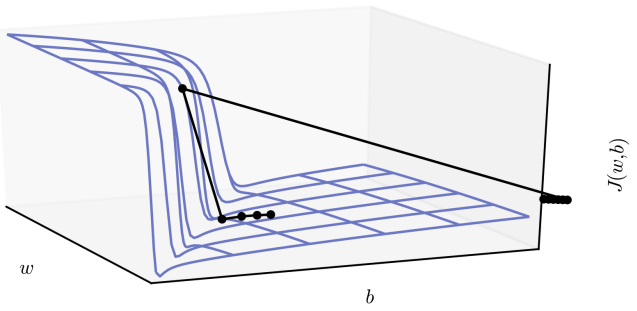


Figure 15.3: The first update steps lead towards the minimum, but overshoots and gets into the "cliff region" where the gradients explode away from the minimum (I. Goodfellow "Deep Learning")

16 Advanced Optimisation Methods

16.1 Overview on Optimisers

Stochastic Gradient Descent Vanilla GD (BGD, MBGD, SGD)

Momentum, Nesterov Allows to surpass flat regions or saddle points — like a ball that keeps on rolling down if it has an initial speed when entering flat regions

RMS Prop Adaptively adjusts the learning rate to incorporate differences in the steepness along different directions in parameter space

Adam Combination of Momentum / Nesterov and RMSprop, state of the art right now, but active field of research

16.1.1 Momentum, Nesterov

It's a modified schme for computing the gradient, with a notion of a momentum encoded within. Algorithm: Compute an exponentially decaying moving average of past gradients and move in the direction of the moving average.

$$\mathbf{m} \leftarrow \beta_1 \mathbf{m} + \alpha \nabla J(\theta)$$
$$\theta \leftarrow \theta - \mathbf{m}$$

β_1 is the 'momentum' hyper-parameter which controls the decay and the friction (large friction and fast decay with small values of β_1 and vice versa, a good default is $\beta_1 = 0.9$).

With the Nesterov scheme the algorithm gets modified, which is considered to be efficient

$$\mathbf{m} \leftarrow \beta_1 \mathbf{m} + \alpha \nabla J(\theta + \beta_1 \mathbf{m})$$
$$\theta \leftarrow \theta - \mathbf{m}$$

16.1.2 Adam Optimisation

Combines Momentum and RMS Prop and is currently considered as the de facto standard.

$$\begin{aligned}
\mathbf{m} &\leftarrow \beta_1 \mathbf{m} + (1-\beta_1)\nabla J(\theta) \\
\mathbf{s} &\leftarrow \beta_2 \mathbf{s} + (1-\beta_2)\nabla J(\theta) \odot \nabla J(\theta) \\
\widehat{\mathbf{m}} &= \frac{\mathbf{m}}{1-\beta_1} \\
\widehat{\mathbf{s}} &= \frac{\mathbf{s}}{1-\beta_2} \\
\theta &\leftarrow \theta - \alpha \frac{\widehat{\mathbf{m}}}{\sqrt{\widehat{\mathbf{s}}+\epsilon}} \odot \nabla J(\theta)
\end{aligned}$$

$\odot$: Symmetric Tensor Product, Element-wise operations are adopted for the squared gradient, the division and square-root in the update step. Initialise $\mathbf{m}$ and $\mathbf{s}$ to 0. In order to avoid that at the beginning these terms are systematically smaller in magnitude, the steps 3&4 are applied.

Hyper-Parameter	Symbol	Default Value
Learning Rate	α	0.9
Momentum	β_1	0.9
RMS Decay	β_2	0.999
Numerical Stabilisation	ϵ	10^{-8}

16.1.3 Learning Schedule

Instead of adapting the learning rate in accordance with information collected during the optimisation process along the trajectory in parameter space, the learning rate may be manually adjusted in the different phases of learning (epochs or update steps). Typically, a higher learning rate is used at the start and then reduced

Piecewise Constant Learning

Epochs	Rate
1-100	0.1
101-300	0.01
301-500	

Performance scheduling
Measure the validation error every *N* steps, reduce the learning rate by a factor of λ when the error stops dropping.

Exponential scheduling
Set the learning rate to a function of the iteration number *t*:

$$\alpha(t) = \alpha_0 \cdot e^{-t/T}$$

Power Scheduling

$$\alpha(t) = \alpha_0 (1 + t/T)^{-c}$$

The hyperparameter c is typically set to 1. Similar to exponential scheduling, but learning rate drops much more slowly.

16.2 Regularisation

Regularisation is a mean to avoid overfitting. I. Goodfellow:

"Regularisation is any modification to a learning algorithm that is intended to reduce its generalisation error but not its training error."

Regularisation Methods

- **Weight Penalty** Constraints on parameters (length of parameter vector, number of parameters) to give preference to simple models

- **Dropout** Randomly drop (neutralise) neurons during training steps to make the solution less dependent on individual neurons
- **Early Stopping** Stop training at the minimum of the cost function on the validation set
- **Data Augmentation** Generate more training data with additional characteristics (symmetries) the solution should have

16.2.1 Weight Penalty

The loss function is modified to give preference to smaller or fewer weights.

$$J = J_{\text{data}} + J_{\text{reg}}$$

J_{data}: **Performance term** how far differs the prediction on the data from the ground truth,
J_{reg}: **Regularisation term** limit the weights from becoming too large

$$\mathbf{J} = \mathbf{J}_0 + \lambda \cdot \Omega(\mathbf{W}) \qquad (\lambda \geq 0)$$

$\mathbf{J}_0$: Original Loss Function, λ: Regularisation Parameter, $\Omega(\mathbf{W})$: Penalty term that favours models with smaller and fewer weights.

Two forms of penalties are popular

$$\begin{aligned} L_1\text{-Regularisation} \quad & \Omega(\mathbf{W}) = \|\mathbf{W}\|_1 = \textstyle\sum_{l,k,j} \|W_{kj}^{[l]}\| \\ L_2\text{-Regularisation} \quad & \Omega(\mathbf{W}) = \|\mathbf{W}\|_2^2 = \textstyle\sum_{l,k,j} \|W_{kj}^{[l]}\|^2 \end{aligned}$$

For optimisation with gradient descent, the derivatives of the loss are modified by the penalty term in a direction to make the loss and the penalty term smaller.

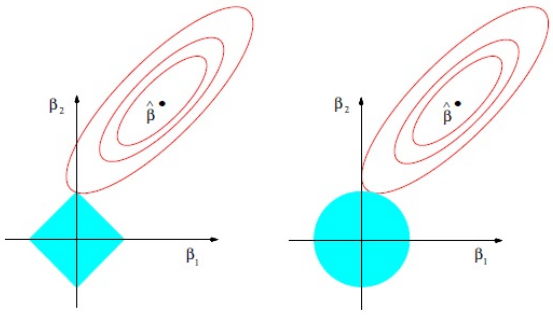


FIGURE 3.11. *Estimation picture for the lasso (left) and ridge regression (right). Shown are contours of the error and constraint functions. The solid blue areas are the constraint regions* $|\beta_1| + |\beta_2| \leq t$ *and* $\beta_1^2 + \beta_2^2 \leq t^2$*, respectively, while the red ellipses are the contours of the least squares error function.*

Figure 16.1: Illustration of L1 and L2 regularisation

16.2.2 Dropout

Most popular regularisation technique for deep neural networks, and highly succesfull: Even state-of-the-art neural networks get a 1-2% accuracy increase simply by adding dropout.

- At each training step, each neuron (including input neurons, excluding output neurons) has a probability p ("dropout rate") of being ignored during this step and having its activations masked
- Dropout rate typically set to 50% for hidden, and 20% for input units

- For testing or in production, neurons don't get dropped, but weights or outputs are corrected (since weights are trained including the dropout rate)

The implications of using Dropout are that simpler models with less units are considered during training, models learn to be less dependent on single units and units cannot co-adapt with their neighbouring units, they have to be as useful as possible on their own. As a result a more robust network is obtained that generalises better.

- **Computationally very cheap**
  Drawing a random binary number for each unit, O(n) additional memory to store these numbers, no additional cost at test time or in production
- **Very versatile**
  Is not limited to certain architectures or models, and can be combined with other regularisation techniques
- **Reduced representational capacity**
  As a regularisation technique, dropout reduces the effective capacity of a model. For very large datasets, regularisation implies little reduction in generalisation error, so the computational cost of using dropout and larger models may outweigh the benefit of regularisation

16.2.3 Early Stopping

Stop training when validation error begins to increase while training error still decreases.

- Run optimisation algorithm to train the model and simultaneously compute validation set error
- Store a copy of the model parameters as long as the validation set error improves
- Iterate until validation set error stops improving
- Return the parameters where the smallest validation set error is observed

Training time (number of training steps/epochs) is considered as hyper-parameter. Early Stopping controls the *effective capacity* of the model by determining the number of steps it can take to fit the training set, thus restricting the volume in parameter space that can be searched. It's furthermore an efficient, non-intrusive search for this hyper-parameter and is easy to combine with other regularisation techniques.

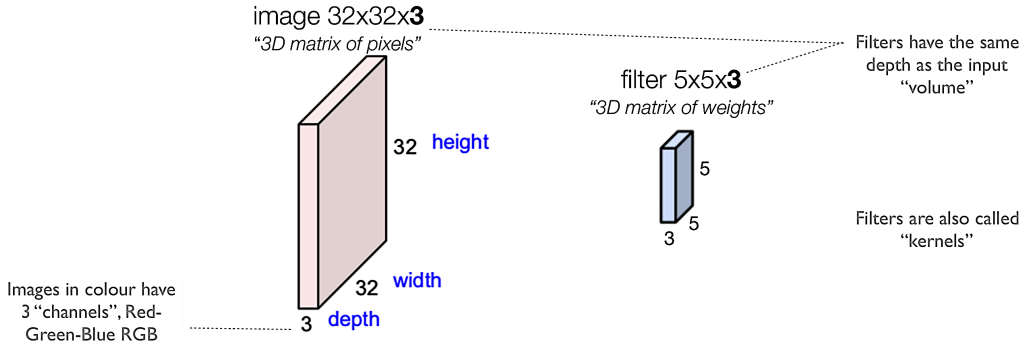
17 Convolutional Neural Networks

The general idea behind CNNs was to define new type of layers and connections that will bring preservation of the spatial structure, hierarchical feature detection, to utilise the fact that objects in images are composed of features that are themselves composed of other features, and to have a robustness to object variabilities such as viewpoint, occlusion and more.

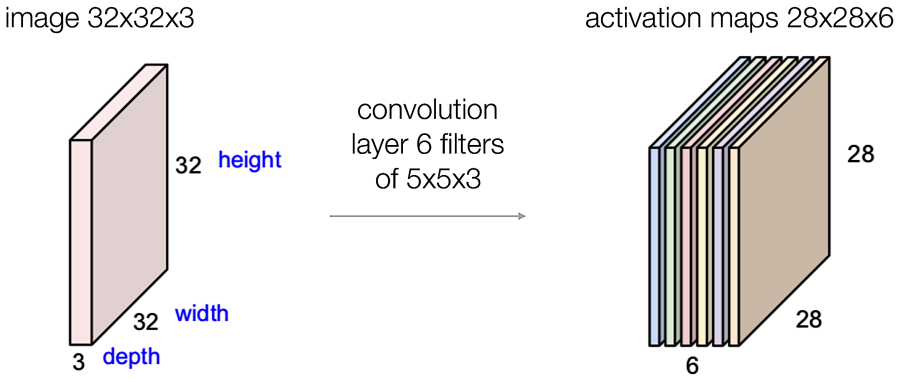
The challenges for image recognition is, amongst others, the intra-class variabilities the model is expected to handle.

17.1 2D Signal Convolutional Layer

The idea is to preserve the spatial structure in the original data with specialised layers coming from image processing, called filters or kernels.



The second idea is to **convolve the filter with the image**, that is to "slide" over the image spatially, computing dot products of the weights of the filter with the window of pixels where the filter is located. The sub-area of an input that influences a component of the output is sometimes called the **receptive field** of the latter. This produces a **activation map** that is the result of the convolution of the filter. This sliding of the filter gives us **translation invariance** regarding to what the filter is sensitive for. Several individual filters can produce several activation maps.



17.2 1D Signal Convolutional Layer

Given $x = (x_1, ..., x_W)$ and a convolution kernel of width $u = (u_1, ..., u_w)$, the convolution $x \circledast u$ is a vector of size $W - w + 1$, with

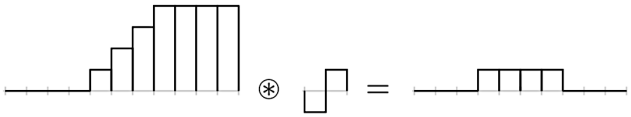
$$(x \circledast u)_i = \sum_{j=1}^{w} x_{i-1+j} \cdot u_j = (x_i, ..., x_{i+w-1}) \cdot u$$

Example:

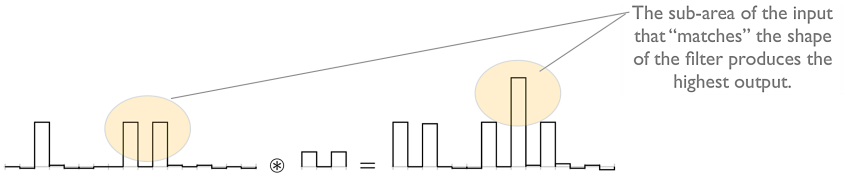
$$(1, 2, 3, 4) \circledast (3, 2) = (3 + 4, 6 + 6, 9 + 8) = (7, 12, 17)$$

Such filter can be differential operators

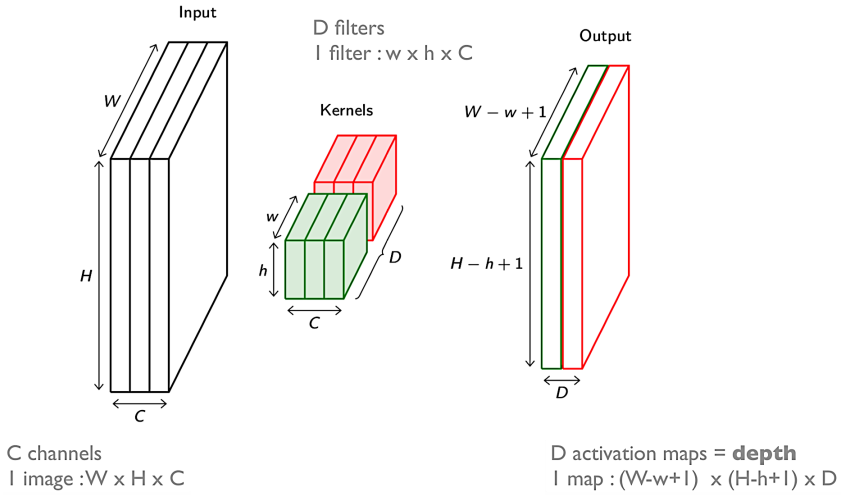
$$(0,0,0,0,1,2,3,4,4,4,4) \circledast (-1,1) = (0,0,0,1,1,1,1,0,0,0).$$



or match simple shapes

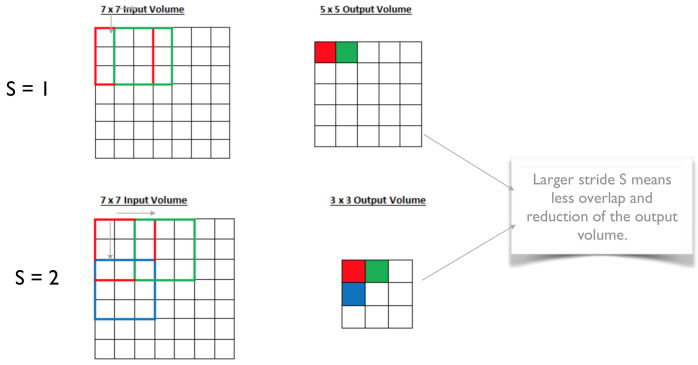


Input images usually have three channels for colour, thus the filters should have the same depth. The activation map have a depth equal to the number of fiters.



17.3 Stride

The stride S specifies a step size when moving the filter across the signal. The rationale behind this is to reduce computation, reduce output size and to omit the need for a strong overlap. Some stride values are incompatible with the input size. Frameworks may trigger errors in case of incompatible stride values.

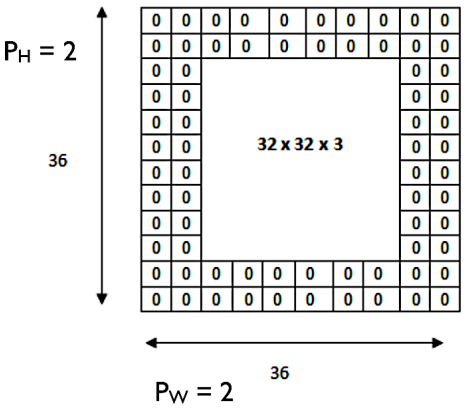


17.4 Padding

The padding P specifies the size of a zeroed frame added around the input, so that the filter can be applied at the border of the input as well and that a reduction in the dimension between convolutional layers can be avoided. Condition to obtain an output size equal to the input size:

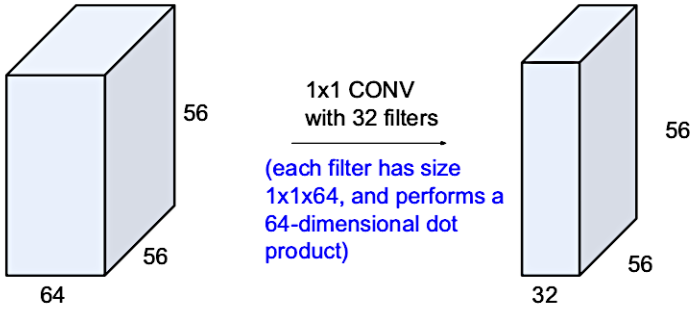
$$P_W = \frac{w-1}{2}$$
$$P_H = \frac{h-1}{2}$$

w: Width of filter, h: Height of filter



17.5 1x1 Convolution

1x1 convolutions are sometimes used along the depth of the input and compress this depth to the number of filters



17.6 Output Size and Number of Parameters

The output size can be computed with

$$O_W = \frac{W - w + 2P_W}{S_w} + 1$$
$$O_H = \frac{H - h + 2P_H}{S_h} + 1$$

(W, H): width and height of input, (w, h): width and height of filter, P: padding, S: stride
The number of parameters in the convolutional layer can be computed with

$$N_{\text{parameters}} = \underbrace{w \cdot h \cdot C \cdot D}_{\text{weights}} + \underbrace{D}_{\text{bias}}$$

C: number of channels, D: number of filters

17.7 Max Pooling Layer

Reduces the spatial size of the representation and applies independently to every depth, defined by a stride and size. The rationale behind this layer is that the most significant activations are kept which brings a hierarchical approach, to reduce the amount of computation and control overfitting.

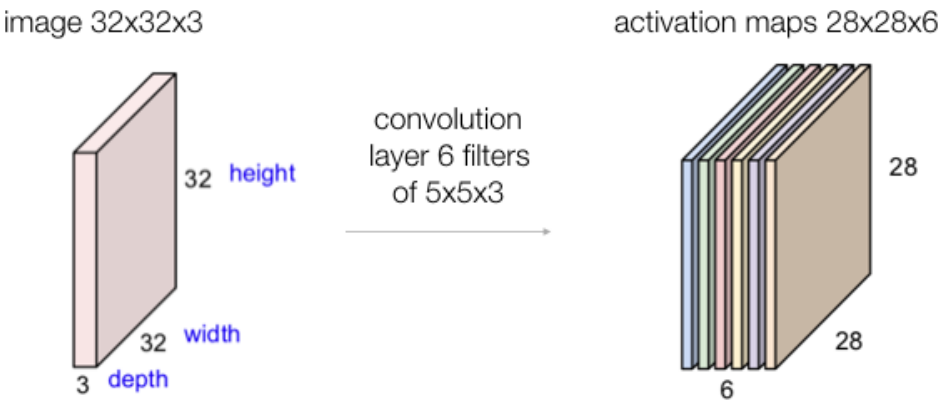
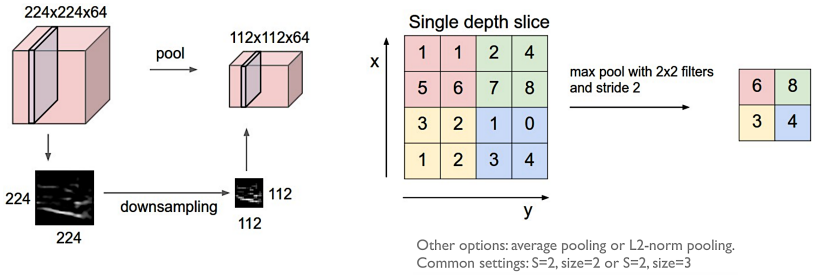


Figure 17.1: An example with $5 \cdot 5 \cdot 3 \cdot 6 + 6 = 456$ parameters



17.8 Other Layers in this Context

The Dropout layer stays the same, as does the rationale behind it, to prevent overfitting and decrease the generalisation error. The Dense layer, however, can be viewed as a convolutional layer with K filters spanning the full input space and is usually used at the output of the CNN architecture to utilise all the features provided by the previous layer in order to take a decision about a class label.

Typical configuration for an image recognition task is to stack sequences CONV-ReLU-POOL and end with a fully connected Dense layer:

CONV	ReLU	CONV	ReLU	POOL	CONV	ReLU	CONV	ReLU	POOL	CONV	ReLU	CONV	ReLU	POOL	DENSE

18 CNNs and Hierarchical Features

The filter parameters are learnt through the training process. The whole system is learning not only to classify but also to extract features.

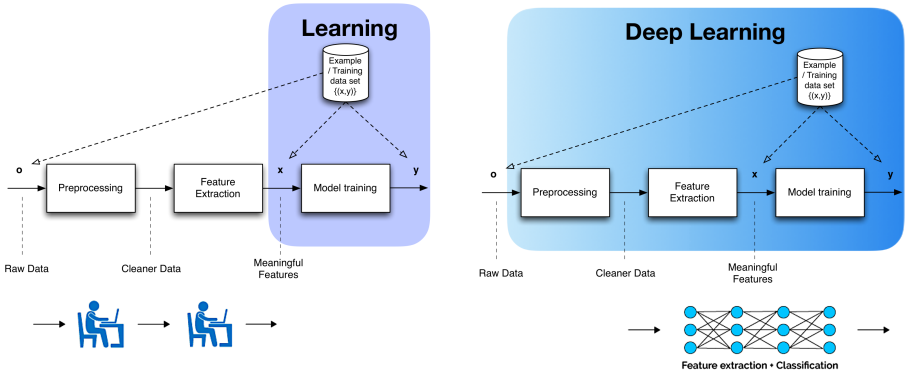


Figure 18.1: Difference in feature extraction in Machine Learning and Deep Learning

The deeper a convolutional neural network the bigger the number of filters in the convolutional layers and the smaller the activation maps due to max pooling. The intuitive interpretation of this is that early layers will extract lower-level features while late layers extract higher-level features.

To visualise what happens in these layer, the **activation maps can be visualised** as images. For a given convolutional layer, consider each filter independent and visualise the corresponding activation map.

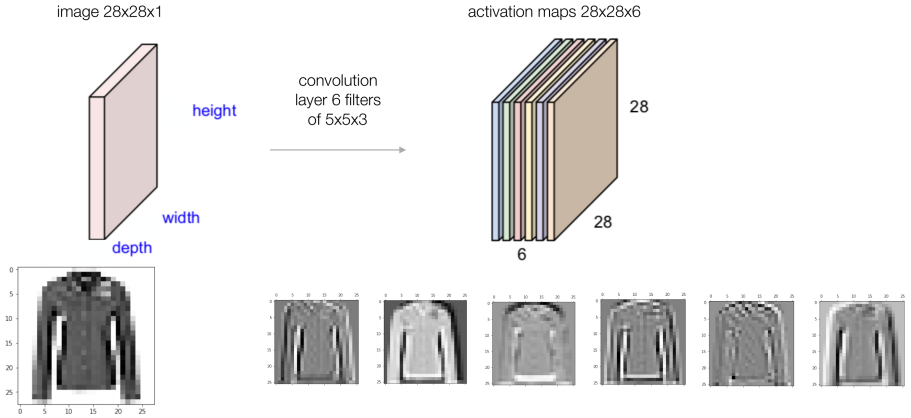
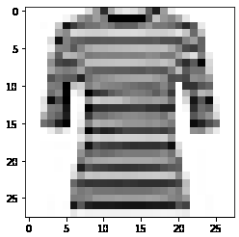
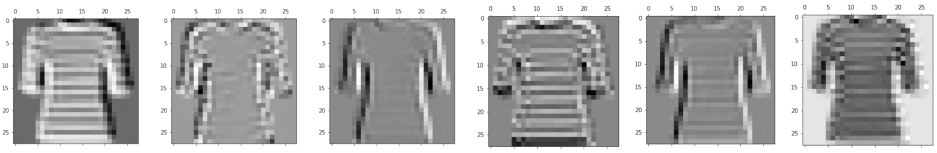


Figure 18.2: Visualisation of activation maps

18.1 Visualisation of Activation Maps - Code Example





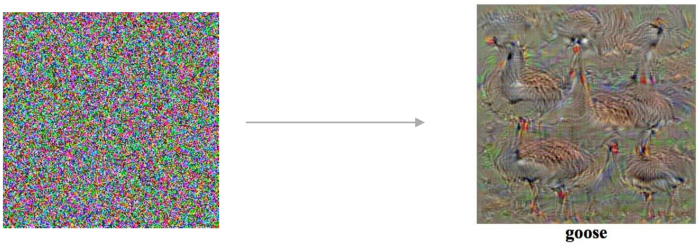
18.2 Visualisation Through Maximising Activations with Custom Inputs

Find **input images that maximise the activation** of a given neuron

1. Start with random image $\mathbf{x}$
2. Forward: compute the activation of a given neuron $a_{i,j}(\mathbf{x})$
3. Backward: perform the backpropagation of the gradient of $a_{i,j}(\mathbf{x})$ up to the pixel values $\frac{\partial a_{i,j}(\mathbf{x})}{\partial \mathbf{x}}$
4. This gradient gives the information on how to change pixel values to increase the activation of the neuron. Apply the update rule in the form of gradient ascent:

$$x \leftarrow x + \alpha \frac{\partial a_{i,j}(\mathbf{x})}{\partial \mathbf{x}} + \text{Regularisation Term}$$

5. Iterate from 2. until convergence



19 Data Augmentation

Overfitting can be caused by having networks with too many parameters that are trained on too few samples. Data augmentation takes the approach of generating artificially more training data from existing training samples. It is similar to bootstrapping in Statistics. For 2D data, this takes the form of affine transformations. The quantity of the distortion is chosen at random.

- **Rotation** range of degrees
- **Translation** percentage of shift in a range for both directions
- **Flip** may not be applicable for all applications (for example data with text)
- **Shear**
- **Zoom**

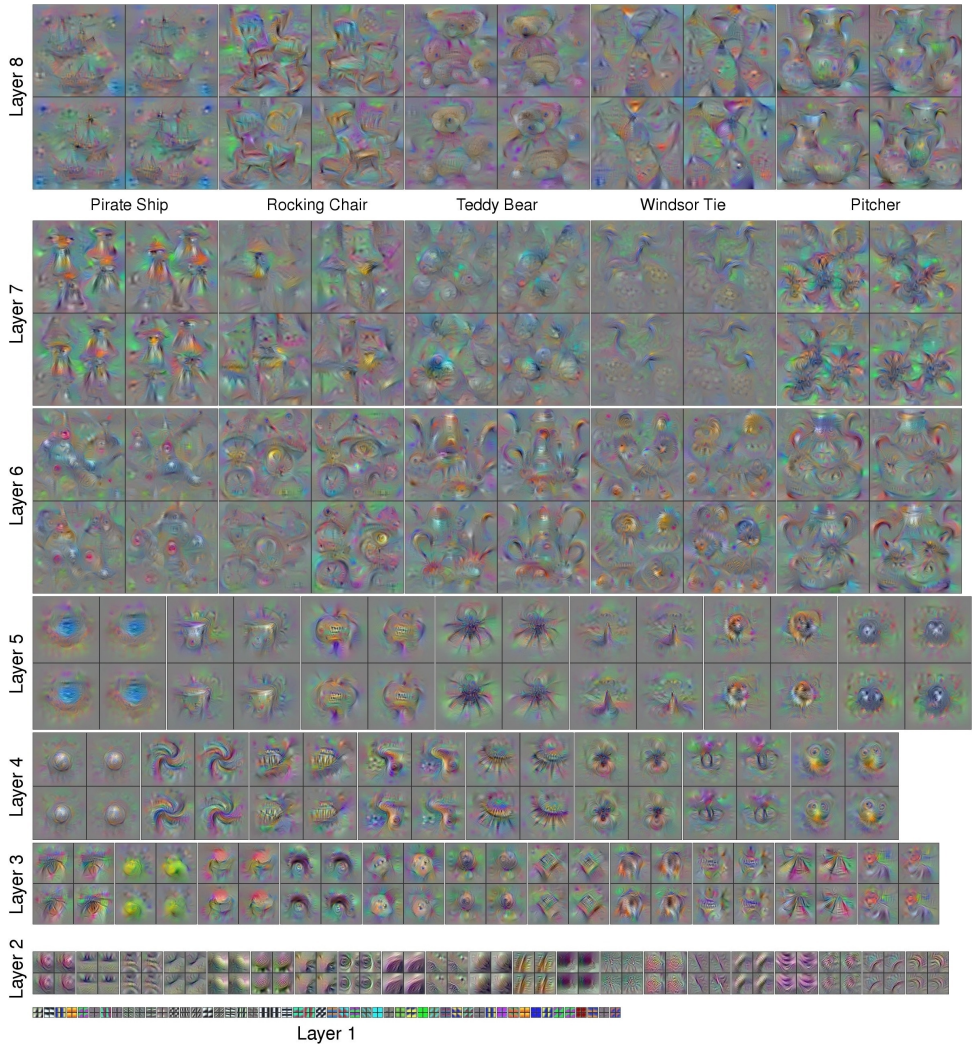
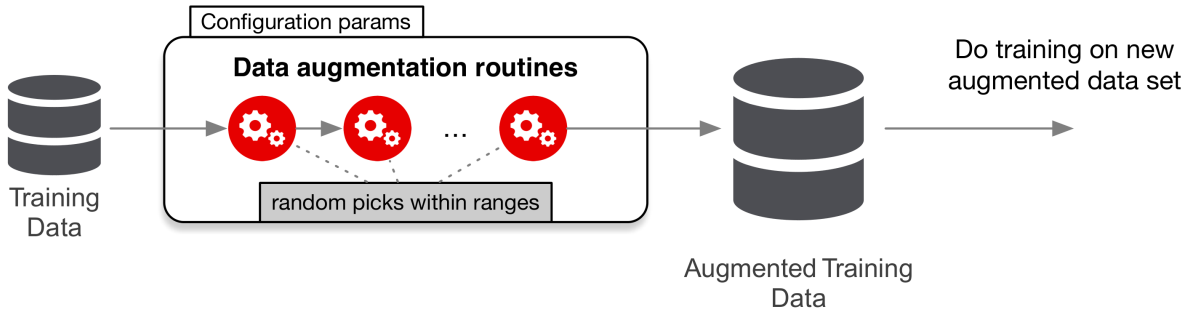


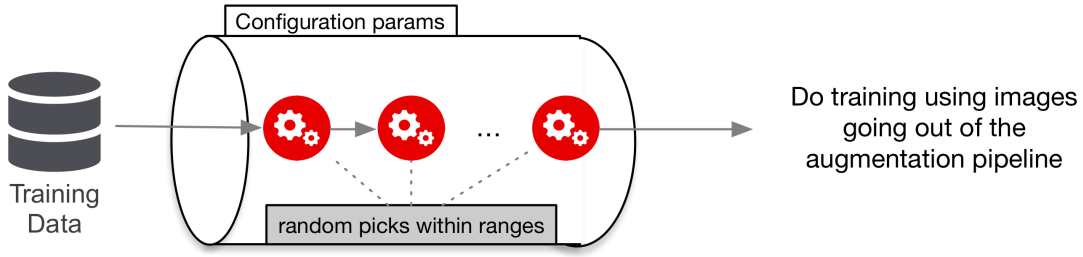
Figure 18.3: Input images that maximise the activations for the respective layer for a given class

- **Colour**

Data augmentation can be separated into pre-augmentation and online augmentation



(a) Pre-augmentation



(b) Online augmentation

Figure 19.1: Data augmentation strategies

The online data augmentation strategy is implemented in Keras. The needed steps are a) configure the image data genertor, b) instantiate the pipeline on the input data set and c) adjust the training method to use said pipeline.

20 Deep CNN Architectures

20.1 GoogLeNet

GoogLeNet is a deep neural network with only 22 layers and nine chained "inception" modules, which can be thought of as networks inside the network. The idea why this inception block works is the notion of mixing filter size to allow looking for more or less details from the previous layer depending on the receptive field size. The object of interest could be close or far, with this both possibilities can be covered. The 1x1 convolutional filters are so called **bottleneck layers** and reduce the depth while preserving the spatial dimensions.

20.2 ResNet

He et al. observed that increasing the number of layers did not succeed. Their hypothesis was, that the problem lies in the optimisation and that deeper model are difficult to optimise. Their ansatz was to provide a fallback to deeper layers in providing them the option to perform an identity mapping if there are difficulties in converging. The solution they came up with was to fit a residual mapping instead of directly try to fit a desired underlying mapping.

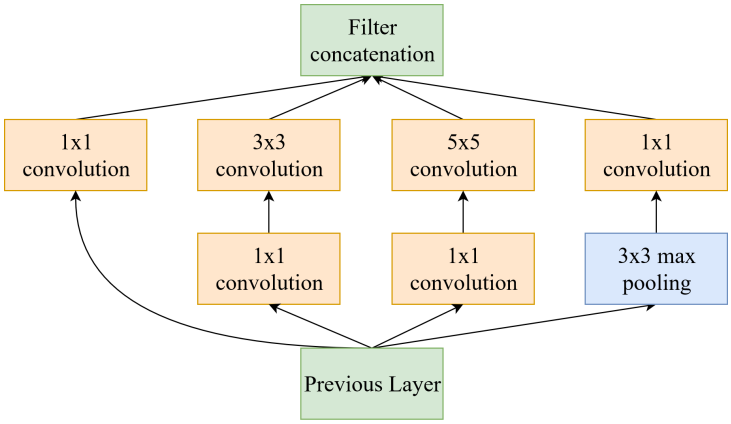


Figure 20.1: An inception block of the GoogLeNet architecture

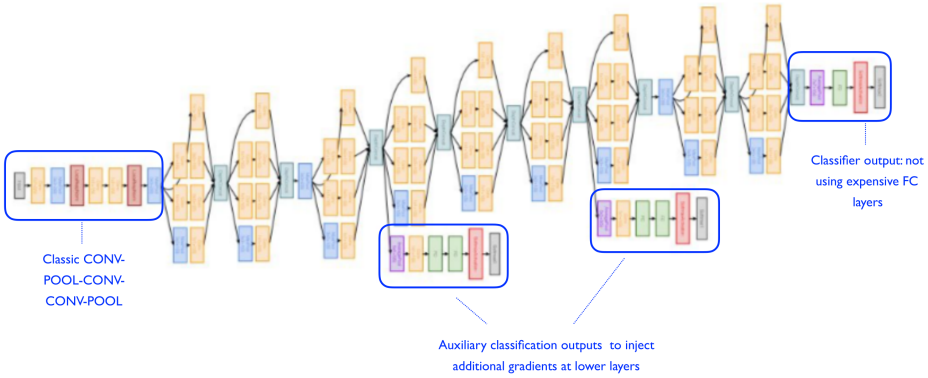


Figure 20.2: Full architecture of GoogLeNet

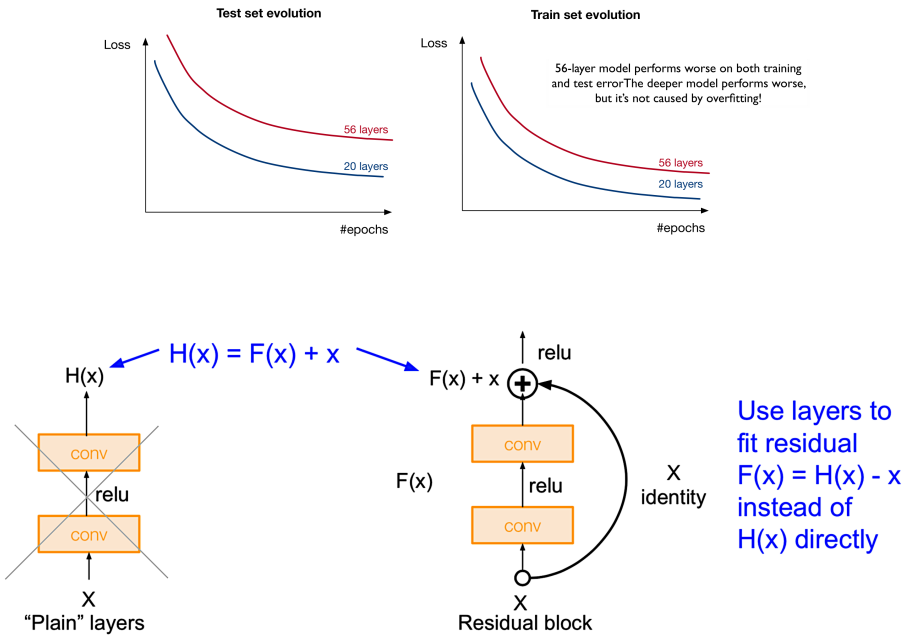


Figure 20.3: Solution to the deep layer optimisation problem in fitting the residual mapping

Other optimisations used to go deep

- Batch Normalization after every CONV layer

- Xavier 2/ initialization from He et al.
- SGD + Momentum (0.9)
- Learning rate: 0.1, divided by 10 when validation error plateaus
- Mini-batch size 256
- Weight decay of 1e-5
- No dropout used

21 Transfer Learning

The filter parameters are learnt through the training process. The whole system is learning not only to classify but also to extract features. In Transfer Learning the attempt is to use the part of a neural network that does the preprocessing and feature extraction. **Transfer learning** is about using knowledge learned from tasks for which a lot of labelled data is available in settings where only little labelled data is available. Kickstart the generalisation by starting from patterns learned for a different task which is in the same problem domain.

There are many pretrained models available for image recognition in Keras.

Model	Size	Top-1 Accuracy	Top-5 Accuracy	Parameters	Depth
Xception	88 MB	0.790	0.945	22,910,480	126
VGG16	528 MB	0.713	0.901	138,357,544	23
VGG19	549 MB	0.713	0.900	143,667,240	26
ResNet50	99 MB	0.749	0.921	25,636,712	168
InceptionV3	92 MB	0.779	0.937	23,851,784	159
InceptionResNetV2	215 MB	0.803	0.953	55,873,736	572
MobileNet	16 MB	0.704	0.895	4,253,864	88
MobileNetV2	14 MB	0.713	0.901	3,538,984	88
DenseNet121	33 MB	0.750	0.923	8,062,504	121
DenseNet169	57 MB	0.762	0.932	14,307,880	169
DenseNet201	80 MB	0.773	0.936	20,242,984	201
NASNetMobile	23 MB	0.744	0.919	5,326,716	-
NASNetLarge	343 MB	0.825	0.960	88,949,818	-

21.1 Using MobileNetV2

Import the MobileNetV2 pre-trained model from Keras. Then load the network architecture specifying the types of weights (here 'imagenet' weights). Finally specify the input shape which should be equivalent to the one used for training this network. The system needs to re-train with new images. For that the ImageDataGenerator utility is used (see section **??**). The input images need to be resized to be compatible with the input of the pre- trained MobileNet network. Also, a preprocessing similar to the one used for training MobileNet has to be plugged in with function .

21.2 Bayes Law

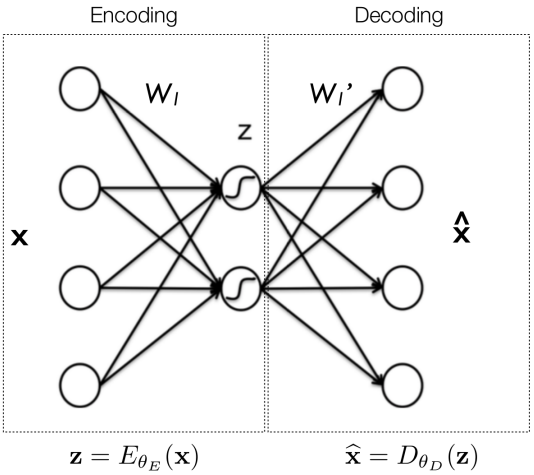
Bayes rule: Elect as winning category the one having the largest *a posteriori* probability. Doing so we are guaranteed to maximise the accuracy. Usually, pre-trained networks are trained on "balanced" data sets, so all $P(C_k)$ are equal to $\frac{1}{K}$. If the deployment environment has different priors than the train/test data, the a posteriori probabilities may need to be rescaled using the new priors.

$$P(C_k|\mathbf{x}) = \frac{p(\mathbf{x}|C_k)P(C_k)}{p(\mathbf{x})}$$

22 Auto-Encoders

An autoencoder is a neural network trained to reproduce its input and able to discover structures and efficient encodings of the input space. The simplest form of an autoencoder is a feedforward, non-recurrent neural network similar to the MLP. In such a network, the output layer has the same number of nodes as the input layer. The mapping function $h_\theta(\mathbf{x})$ is trained to reconstruct its own inputs.

$$\widehat{\mathbf{x}} = h_\theta(\mathbf{x})$$



In order to learn something else than the identity function, the network is often composed in such a way that the number of nodes in the hidden layer is smaller than the number of nodes in the input and output layers, forming a so-called diabolo network. The principle is to separate the network into two pieces after training.

The loss function simply the mean square error between input and reconstructed input

$$\begin{aligned} J(\theta) = J(\theta_E, \theta_D) &= \frac{1}{2N}\sum_{n=1}^{N}(h_\theta(\mathbf{x}_n) - \mathbf{x}_n)^2 \\ &= \frac{1}{2N}\sum_{n=1}^{N}(\widehat{\mathbf{x}}_n - \mathbf{x}_n)^2 \end{aligned}$$

Auto-encoders can be used for:

- Data compression or dimensionality reduction train the auto-encoder, send the decoder architecture and learned weights, afterwards data can be encoded, sent in compressed form and decoded by the decoder network. The error or loss can be pre-computed.
- Feature extractors use the encoder part of the network to identify features
- De-noising Train the auto-encoder on noisy input and clean output
- Image in-painting, information recovery Train on input data with missing parts and complete outputs, so that the auto-encoder learns robust features
- Initialise Deep Network for supervised learning

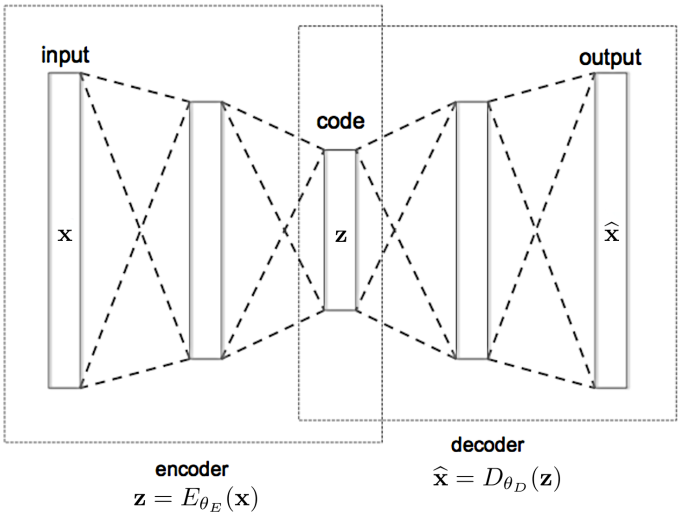


Figure 22.1: Stacked or convolutional auto-encoder

23 Word Embeddings

Inputs of networks need to be numerical continuous values, but using a one-hote atomic representation of words to feed is generally not working as it is too sparse and does not carry information. A **word embedding** is a projection of a word into vectors of real number, that is a mapping of a space where each word has its own dimension to a space with a lower dimensionality. This has the advantages of exposing good features, allowing the back propagation to train this projection for the given classification, and there is a notion of proximity of related words.

Example: projection into a 3 dimensional space

anarchism	0.5	0.1	−0.1
originated	−0.5	0.3	0.9
as	0.3	−0.5	−0.3
a	0.7	0.2	−0.3
term	0.8	0.1	−0.1
of	0.4	−0.6	−0.1
abuse	0.7	0.1	−0.4

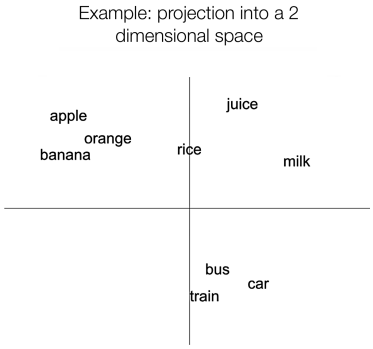


Figure 23.1: Projection of word embeddings into lower dimension spaces

23.1 Embedding Layer

An embedding layer is a simple single layer neural network with the one-hot encoding of the words in a vocabulary of dimension V and an output with a vector of dimension N.

The embedding layer is the first layer of a natural language processing network and performs a first feature extraction. There are three strategies to be considered before using a word embedding layer. Either an embedding layer can be trained on the given dataset from scratch, a pre-trained generic embedding such as *word2vec* or *GloVe* can be used as is (that is not adjust the weights while training), or use a pre-trained embedding layer as initialisation and fine-tune on the dataset.

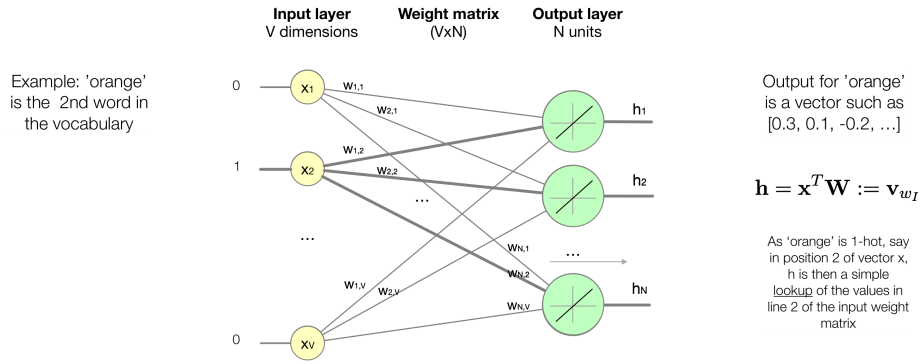
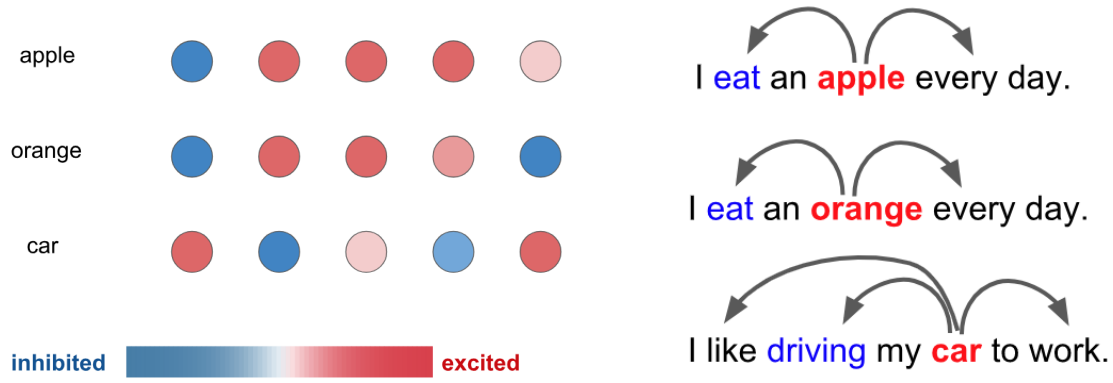


Figure 23.2: V to N embedding layer

23.2 Generic representation *word2vec*

word2vec relates to models producing word embedding into space with **contextual similarity** or words, that is words that share common context are located in close proximity. The idea behind *word2vec* is to represent words with continuous levels of activations where words in similar context share similar activations.



(a) Example of continuous activations in *word2vec*

(b) Word context assumption

The hypothesis behind *word2vec* is that a word is also represented by its context, though the order of the context words is disregarded. This approach is called the bag-of-words assumption. The word2vec mapping is not only about clustering related words, it is also able to preserve some path meaning. In vector terms, $v_{\text{woman}} + (v_{\text{king}} - v_{\text{man}})$ is close to v_{queen}.

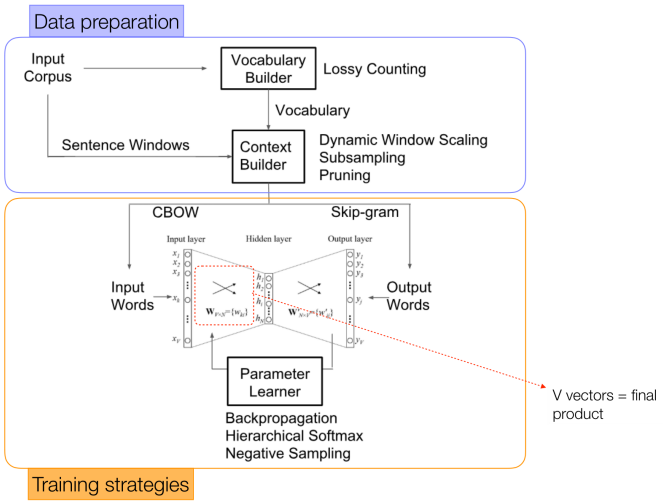
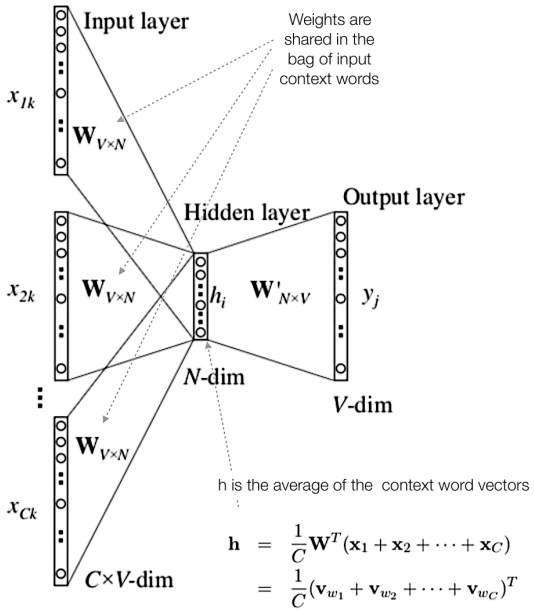


Figure 23.4: Training strategy with *word2vec*

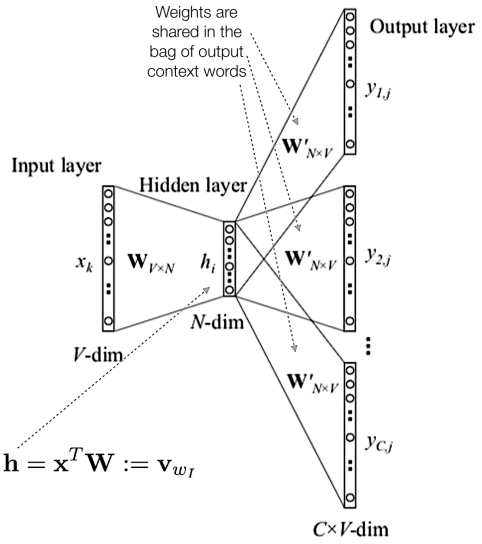
23.3 CBOW - Continuous Bag of Words

CBOW can be seen as an extension of a one word context to a multi-word context at the **input** layer, where a word is predicted from a bag of context words.



23.4 Skip Gram Model

Skip Gram can be seen as an extension of a one word context to a multi-word context at the **outut** layer, where a bag of context words are predicted from a word.



23.5 Practical Considerations on Softmax

The softmax and the derived update rules need to iterate through the whole vocabulary, which is impractical if V and the training corpus are large, which is the case most of the time. Proposed solutions for this problem are **Hierarchical Softmax** and **Negative Sampling**, where only a sample of the outputs are updated instead of all of them.

23.6 Implementation of *word2vec*

23.7 GloVe

The main difference is that GloVe is not based on artificial neural networks but on statistical models, where the dimensionality reduction is achieved through co-occurrence of word X context. Still, *word2vec* and approaches based on co-occurrence matrices are related.

23.8 FastText

An extension to *word2vec* which breaks up the words into several n-grams, or sub-words, before feeding them into the neural network. For example the tri-gram for the word "apple" is "app", "ppl" and "ple". The word embedding vector for a word will be the sum of all of its n-grams, and after training there are word embeddings for all the n-grams of the training set. The main advantage of this approach is that **out-of-vocabulary words can be represented**, that is words not in the training set, since their n-grams probably appear in other words that are in the training set.

24 Recurrent Neural Networks

Recurrent neural networks are used to capture context information over a sequence of steps from which conclusions can be drawn. They are specialised for modelling sequences the same way as CNNs are specialised for modelling grids. The idea is that successive elements are not independent, there are local and possibly long-term dependencies that can be exploited.

Typical applications

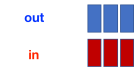
- Speech Recognition
- Sentiment Classification
- Machine Translation
- Captioning/Subtitling
- Chatbots
- Named Entity Recognition
- Word or Music Generation
- Time Sequence Modelling

24.1 Sequence Models

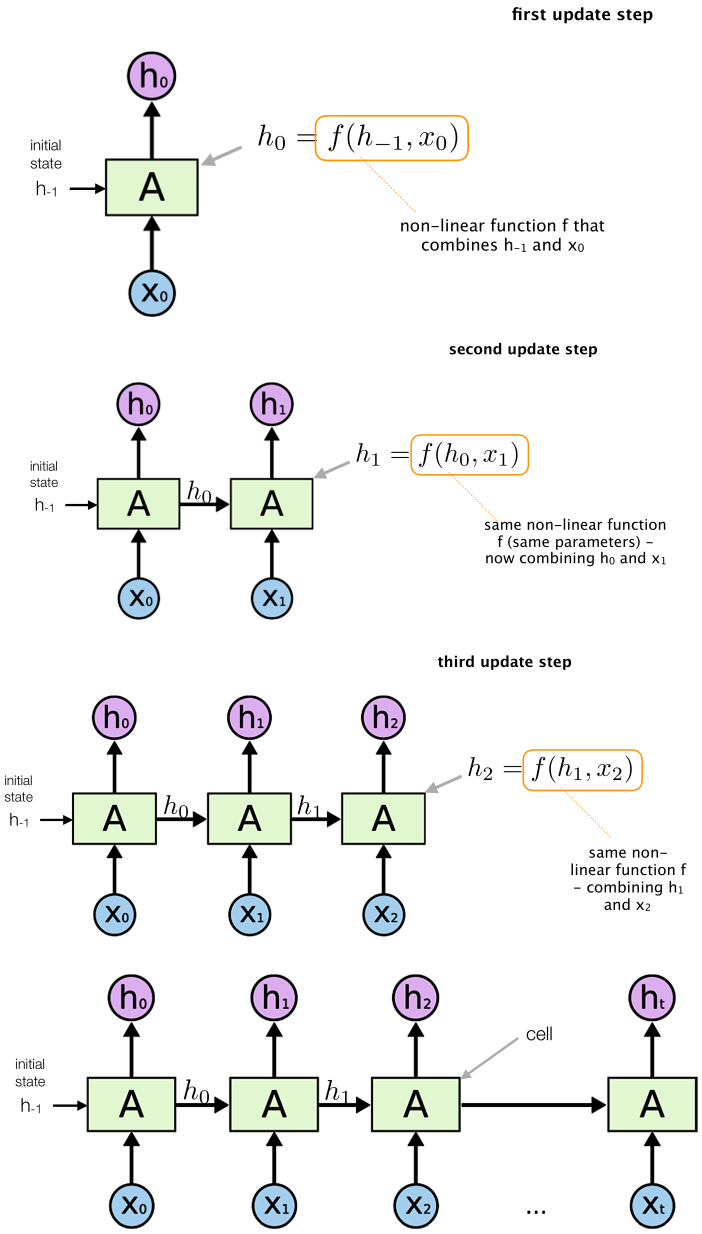
$$\begin{matrix}\text{Input sequence } x \text{ with length } T_x \\ x = (x_1, x_2, ..., x_{T_x})\end{matrix} \Rightarrow \begin{matrix}\text{Output sequence } y \text{ with length } T_y \\ y = (y_1, y_2, ..., y_{T_y})\end{matrix}$$

Typically, the elements of x and y are **multi-dimensional vectors** and are **parametrised** by an **index** or **time variable** to describe the position in the sequence.

24.1.1 Sequence Model Categories

Category	Input/Output Length	Examples	Illustration
one-to-many	$T_x = 1, T_y \geq 1$	Image captioning: Image → Sequence of Words	
many-to-one	$T_x \geq 1, T_y = 1$	Sentiment classification: Sequence of Words → Sentiment	
many-to-many	$T_x \geq 1, T_y \geq 1, T_x = T_y$	Named entity recognition: Sequence of Words → Sequence of entity classes	
many-to-many	$T_x \geq 1, T_y \geq 1, T_x \neq T_y$	Language translation, speech recognition, chatbots: Sequence of Words → Sequence of Words	

24.2 Recurrent Cells

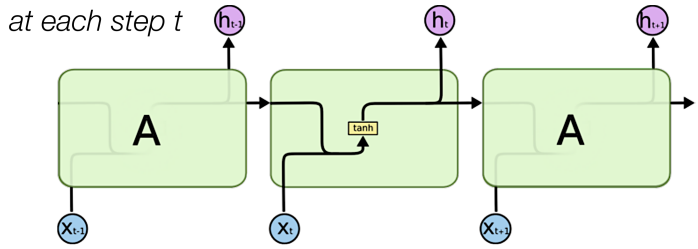


(a) Computational graph unrolled

24.3 Single Layer RNN

Updated through a succession of steps. At update t, it consumes the next element of the input sequence (if available) and the previous state and computes the new state by applying a suitable function $h_t = f(h_{t-1}, x_t)$. Parameters are shared by applying the same function for updating. By consuming the state of the previous cell, it can memorise information. State is passed on to the next cell or possibly to another layer and suitably initialised before the first step.

24.3.1 Computation for *Simple* RNN-Cells



The computation for forward propagation is an affine transformation of the input with a non-linear activation function tanh().

$$h_t = g(\mathbf{W_x} \cdot x_t + \mathbf{W_h} \cdot h_{t-1} + b_h) \tag{8}$$

$\mathbf{W_x}$: $n_h \times n_x$, $\mathbf{W_h}$: $n_h \times n_h$, and b_h are the same for all time steps

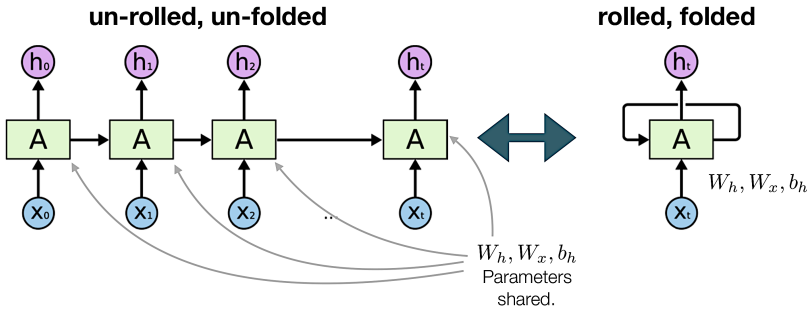


Figure 24.2: Rolled representation for a Simple RNN

24.3.2 Example in Keras

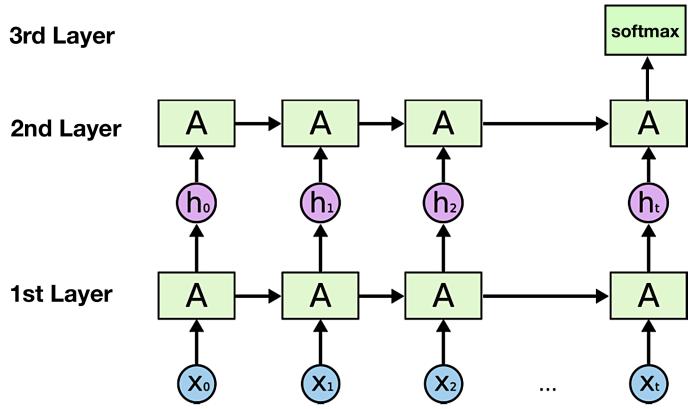
: Positive integer, dimensionality of the output space, : Boolean. Whether to return the last output in the output sequence, or the full sequence, : length of input sequence () and dimension of an element of the input sequence, softmax layer for binary classification

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
simple_rnn_3 (SimpleRNN)     (None, 64)                7744
_________________________________________________________________
dense_3 (Dense)              (None, 2)                 130
=================================================================
Total params: 7,874
Trainable params: 7,874
Non-trainable params: 0
_________________________________________________________________
```

$$\begin{aligned} \mathbf{W}_h &: 64 \cdot 64 = 4096 \\ \mathbf{W}_x &: 64 \cdot 56 = 3584 \\ \mathbf{b}_h &: 64 \cdot 1 = 64 \\ &= 7744 \end{aligned}$$

$$2 \cdot 64 + 2 = 130$$

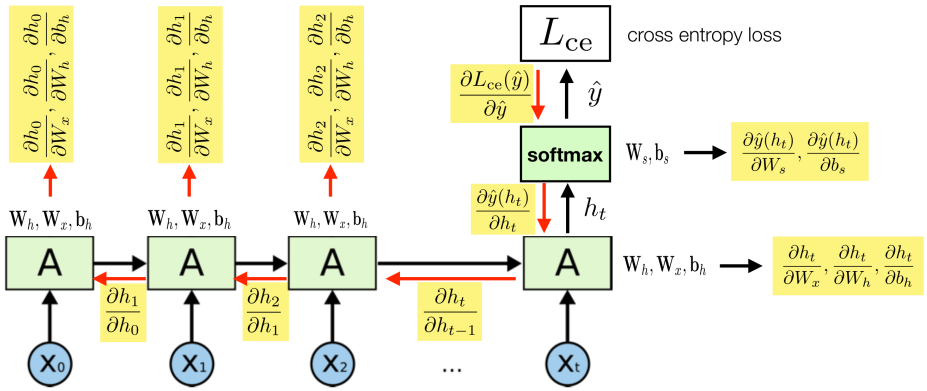
24.4 Stacked RNNs



Layer (type)	Output Shape	Param #
simple_rnn_19 (SimpleRNN)	(None, 15, 64)	7744
dropout_8 (Dropout)	(None, 15, 64)	0
simple_rnn_20 (SimpleRNN)	(None, 64)	8256
dropout_9 (Dropout)	(None, 64)	0
dense_16 (Dense)	(None, 2)	130

Total params: 16,130
Trainable params: 16,130
Non-trainable params: 0

24.5 Back-propagation with a Single-Layer Simple RNN



A $h_t = g(z_t), z_t = \mathbf{W_x} \cdot x_t + \mathbf{W_h} \cdot h_{t-1} + b_h$

The partial derivatives are vectors of matrices, and therefore straightforward to compute $\frac{\partial h_t}{\partial h_{t-1}} = W_h^T \cdot g'(z_t)$

24.5.1 Gradient for Weights Matrix of a Layer

As every layer has the same weights matrix, their contribution is summed.

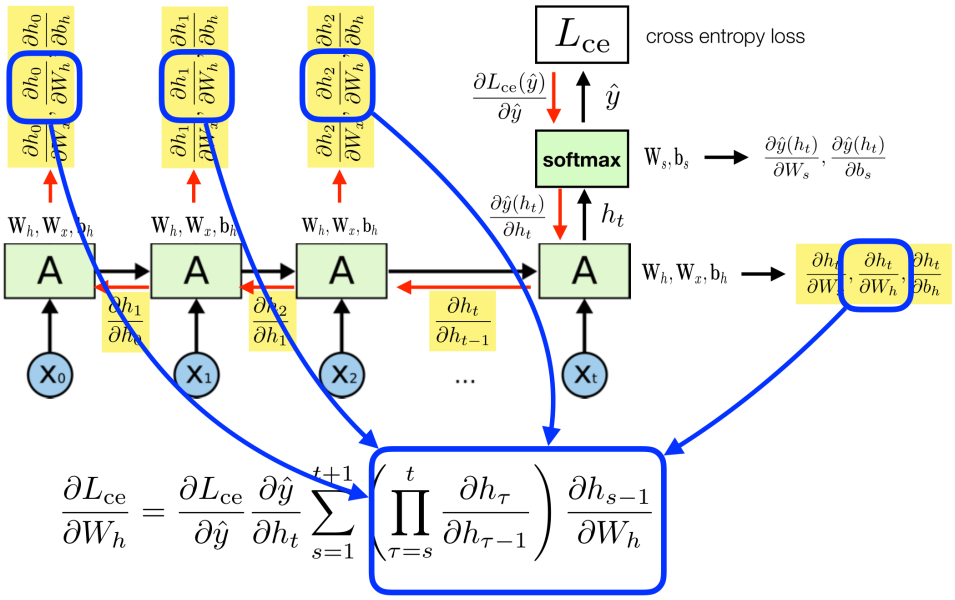


Figure 24.3: Contribution of weights matrix in each ceall

$$\frac{\partial L_{\text{CE}}}{\partial W_h} = \frac{\partial L_{\text{CE}}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial h_t} \sum_{s=1}^{t+1} \left(\prod_{\tau=s}^{t} \frac{\partial h_\tau}{\partial h_{\tau-1}} \right) \frac{h_{s-1}}{\partial W_h}$$

$\prod_{\tau=s}^{t} \frac{\partial h_\tau}{\partial h_{\tau-1}}$ is a product with many factors, each of the form $\mathbf{W}_h^T \cdot g'(\mathbf{z}_\tau)$. Since the same parameters are shared across the different cells, many terms contribute. If $t - s$ is large, that is there are long sequences, the product term can become very small and the gradient vanishes, or very large and the gradient explodes. This makes learning long-range dependencies hard to learn.

24.6 Strategies to Overcome the Problem in RNNs

Properly initialise the weights, particularly the recurrent weights W_x, W_h. Use a non-saturating activation functions (ReLU, Leaky ReLU) to alleviate the vanishing gradients problem. Batch Normalisation in vertical direction, not horizontally across time. Clip the gradients to alleviate the problem of exploding gradients. Or use models with built-in long-term memory such as GRU or LSTM.

24.6.1 Recurrent Weight Initialisation

IRNN Identity matrix for the recurrent weights, so that the state variable remains unchanged in case there are no inputs $\mathbf{x}$

$$W_h = \begin{pmatrix} 1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 1 \end{pmatrix}$$

np-RNN Recurrent weight matrix initialised to a normalized-positive definite matrix with the highest eigenvalue = 1 and all the others < 1. No support in Keras, custom initialiser has to be used.

$$W_h = \frac{1}{e}(A + I)$$
$$A = \frac{1}{N} R^T \cdot R$$
$$e = \max(\lambda(A + I))$$

With

R : Standard Gaussian

$\lambda(x)$: Set of eigenvalues of X

both strategies ReLU is used to alleviate the vanishing gradients problem, and random values are used to initialise W_x.

RNN Type	Test Accuracy	Number of Parameters	Sensitivity to Parameters
IR-NN	67.0%	1 x	high
np-RNN	75.2%	1 x	low
LSTM	78.5%	4 x	low

Figure 24.4: Comparison of different initialisation and architecture

24.7 Long Short Term Memory Cell (LSTM)

LSTM networks have the same general structure of RNNs with cyclicly updated cells. But the cell itself has a much richer structure, it is designed to keep a **long-term memory** that is kept as an additional state variable $\mathbf{c}_t$ to be updated.

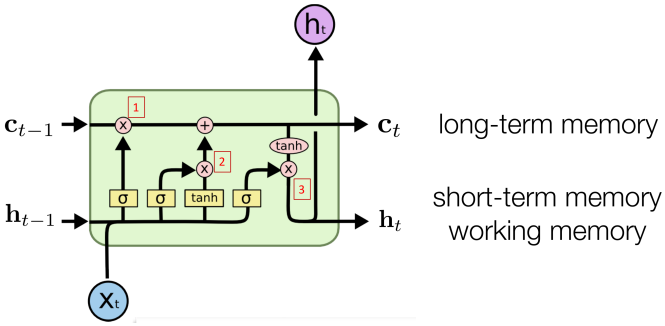
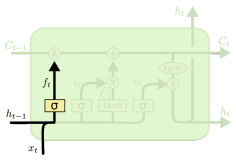


Figure 24.5: A LSTM unit with (1) forget gate, (2) input gate and (3) output gate

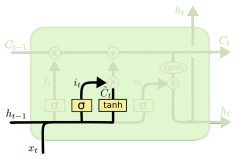
The long-term is updated through **gates** (refer to figure **??**), which are implemented by affine transformation of inputs, an activation function and element-wise multiplication ($\times$ in the figure). The element-wise multiplication is only available in LSTM cells and adds representational capacity. **Note**: $\odot$ denotes the Hadamard product, and stands for element-wise multiplication.

Forget Mechanism If a scene ends, the model can forget some scene-specific information, but should remember other information



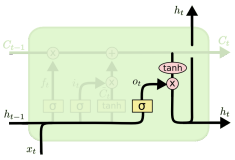
$$f_t = \sigma(\mathbf{W}_{f,x} \cdot x_t + \mathbf{W}_{f,h} \cdot h_{t-1} + \mathbf{b}_f)$$

Save Mechanism When the model sees a new image, it needs to learn whether any information about the image is worth using and saving



$$i_t = \sigma(\mathbf{W}_{i,x} \cdot x_t + \mathbf{W}_{i,h} \cdot h_{t-1} + \mathbf{b}_i)$$
$$\tilde{c}_t = \tanh(\mathbf{W}_{c,x} \cdot x_t + \mathbf{W}_{c,h} \cdot h_{t-1} + \mathbf{b}_c)$$
$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t$$

Output Mechanism The model needs to learn which part of the long-term memory is useful in the given situation.



$$o_t = \sigma(\mathbf{W}_{o,x} \cdot x_t + \mathbf{W}_{o,h} \cdot h_{t-1} + \mathbf{b}_o)$$
$$h_t = o_t \odot \tanh(c_t)$$

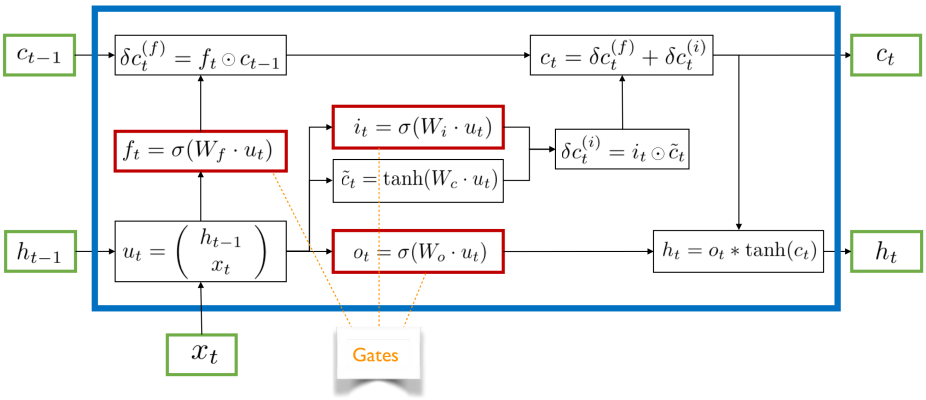
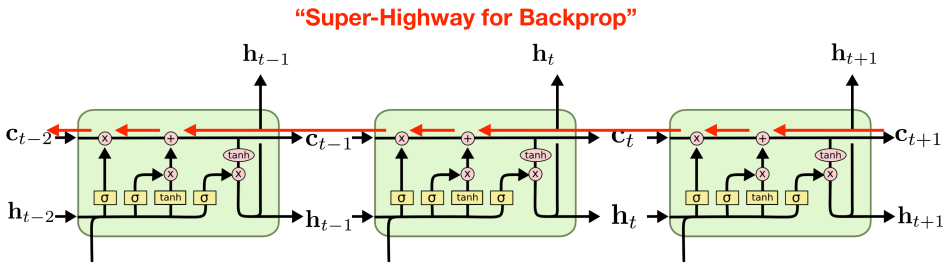


Figure 24.6: The computational graph for a single LSTM step

The matrices in figure **??** $W_f = (W_{f_h} W_{f,i})$, $W_i = (W_{i,h} W_{i,i})$, $W_c = (W_{c,h} W_{c,i})$, $W_o = (W_{o,h} W_{o,i})$ have the shape $n_h \times (n_h + n_x)$ with n_h the dimension of the hidden state variables h_t, c_t and n_x the dimension of the input x_t.

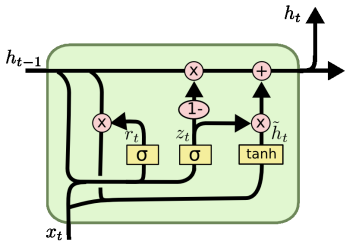
24.7.1 Back-propagation in LSTM



$$\mathbf{c}_t = f_t \odot \mathbf{c}_{t-1} + i_t \odot \tilde{\mathbf{c}}_t \Rightarrow \frac{\partial \mathbf{c}_t}{\partial \mathbf{c}_{t-1}} = f_t$$
$$\Rightarrow \frac{\partial \mathbf{c}_t}{\partial \mathbf{c}_s} = \prod_{\tau=s+1}^{t} \frac{\partial \mathbf{c}_\tau}{\partial \mathbf{c}_{\tau-1}} = \prod_{\tau=s+1}^{t} \underbrace{f_\tau}_{\text{time dependent in }]0,1[}$$

24.8 Gated Recurrent Unit (GRU)

Variation of LSTM without separate long-term memory, where the forget and input gate are merged and a relevance or reset gate has been added.



Less parameters, since there are only three weight matrices of dimension $n_h \times (n_h + n_x)$: W_r, W_c, W_u

Relevance Gate

$$r_t = \sigma(W_{rh} \cdot h_{t-1} + W_{rx} \cdot x_t + b_r)$$

Candidate State

$$\tilde{h}_t = \tanh(W_{ch} \cdot (r_t \odot h_{t-1}) + W_{cx} \cdot x_t + b_c)$$

Update Gate

$$u_t = \sigma(W_{uh} \cdot h_{t-1} + W_{ux} \cdot x_t + b_u)$$

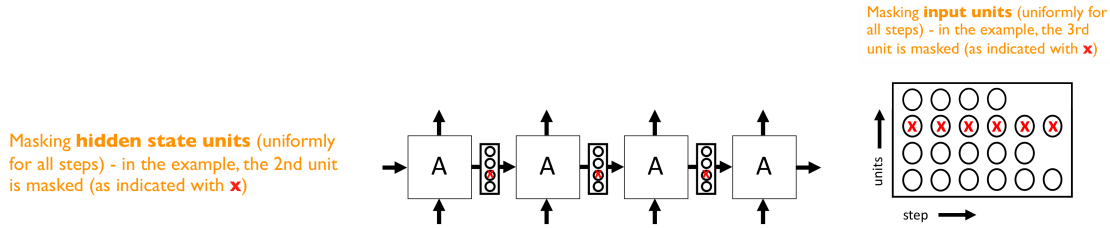
Update

$$h_t = (1 - u_t) \odot h_{t-1} + u_t \odot \tilde{h}_t$$

24.9 A Note on Regularisation Schemes

Batch normalisation for LSTMs is problematic, as they are deepest along temporal dimension and repeated scaling could lead to exploding gradients. It's more advisable to batch-normalise the hidden state variable h and input x independently (at each time step). Use separate but fixed scaling parameters. Do not do a separate batch-normalisation of the long-term memory state c.

Dropout is implemented as *variational dropout* (in Keras) in LSTM and masks units in all time steps. Either the mask is applied to **input/output units** with or a Dropout layer, or applied to the **hidden state units** with the keyword .

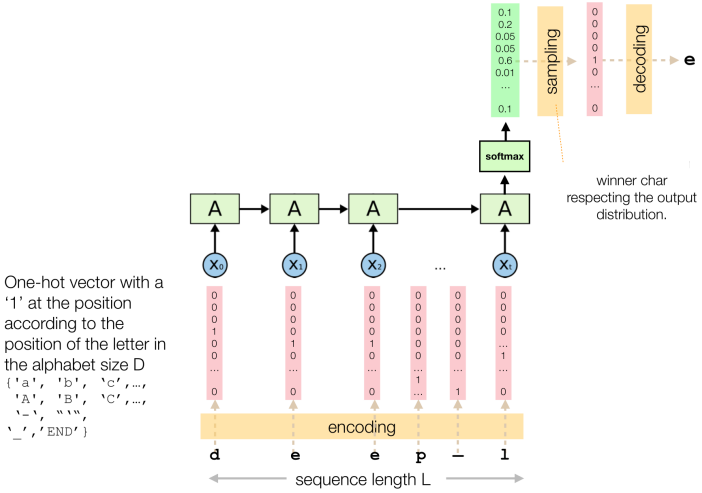


25 Generative RNNs

A generative system is a system able to generate new consistent data from a seed. By consistent, we mean respecting temporal or spatial structures that have been learned from the input space. The two approaches to this would be the many-to-one (from a preceding sequence of words predict the next) and many-to-many (from a preceding sequence predict the next few words).

25.1 Many-to-One Approach

The principle is to train the RNN to predict the next token of the sequence in a many-to-one setting. At testing time, an initial sequence of tokens is used as seed and an output is sampled respecting the posterior probabilities computed by the softmax layer.



$$\mathbf{w}_h : 256 \times 256 = 65536$$
$$\mathbf{w}_x : 256 \times 65 = 16640$$
$$\mathbf{b}_h : 256 \times 1 = 256$$
$$= 82432$$

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
simple_rnn (SimpleRNN)       (None, 256)               82432
_________________________________________________________________
dense (Dense)                (None, 65)                16705
_________________________________________________________________
activation (Activation)      (None, 65)                0
=================================================================
Total params: 99,137
Trainable params: 99,137
Non-trainable params: 0
_________________________________________________________________
```

25.1.1 Using the Trained Network

1. Define a seed
2. Predict the output probabilities and sample the next character from this distribution respecting the a posterior probability
3. Shift the input by removing the first character and appending the predicted character
4. Repeat

25.2 Many-to-Many Approach

The difference to the many-to-one approach is mainly at training time where the RNN cells are emitting an output at each step and the loss is computed from all the outputs. Thus the output is time distributed across the whole sequence.

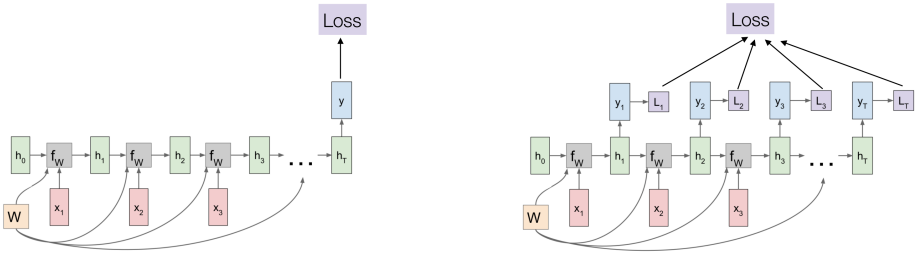


Figure 25.1: Difference in loss calculation for many-to-one (left) and many-to-many (right)

In theory you go forward through the entire sequence to compute loss, then backwards through the entire sequence to compute the gradients. But this is infeasible if the sequence is long. In practice the sequence gets truncated, and the forward and backward propagation is done on this truncated sequence.

25.3 Seq2Seq Models

A **sequence to sequence** model is learning to map sequences of different length in situations where the entire input sequence is required in order to start predicting the target sequence. Typical applications include machine translation, question-answering bots, grapheme to phoneme mappers.

25.4 Encoder - Decoder Principle

An **encoder** processes the input sequence and compresses the information into a context vector h_t of a fixed length. This representation is expected to be a good summary of the meaning of the whole source sequence. The **decoder** is initialized with the context vector to emit the transformed output. In its original form, Seq2Seq only used the last state of the encoder network as the decoder initial state. This works reasonably well for short sequences but is more difficult for long sequences, as the vector needs to encode a lot of information. A solution is to plug-in **attention** mechanisms.

25.5 Seq2Seq with Attention

Attention is an interface between the encoder and decoder that provides the decoder with information from all (or parts of) the encoder hidden states. There are two types of attention. **Global** uses all of the encoder hidden states, while **Local** uses only a subset. The idea is to provide the decoder with a notion of **alignment** on which segments of original sequence match with their corresponding segments of the translation, and where the decoder should focus to map the current output better.

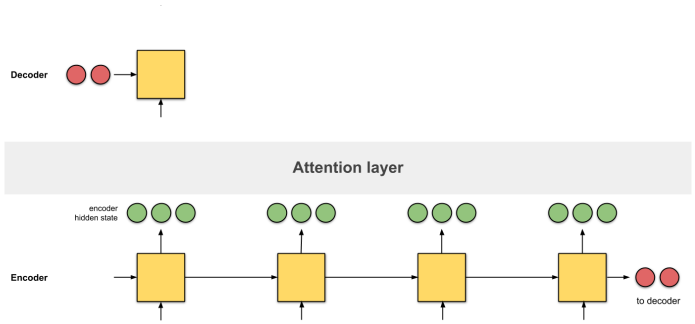


Figure 25.2: Seq2Seq with attention layer

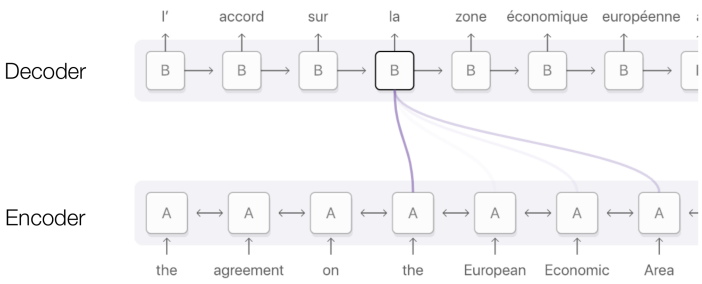


Figure 25.3: Notion of focus

25.6 Steps for Seq2Seq with Attention

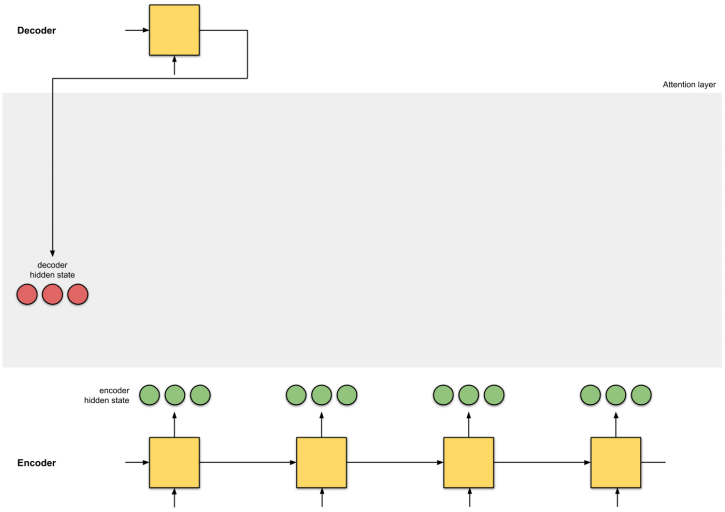


Figure 25.4: Prepare all the available encoder hidden states (green) and the first decoder hidden state (red)

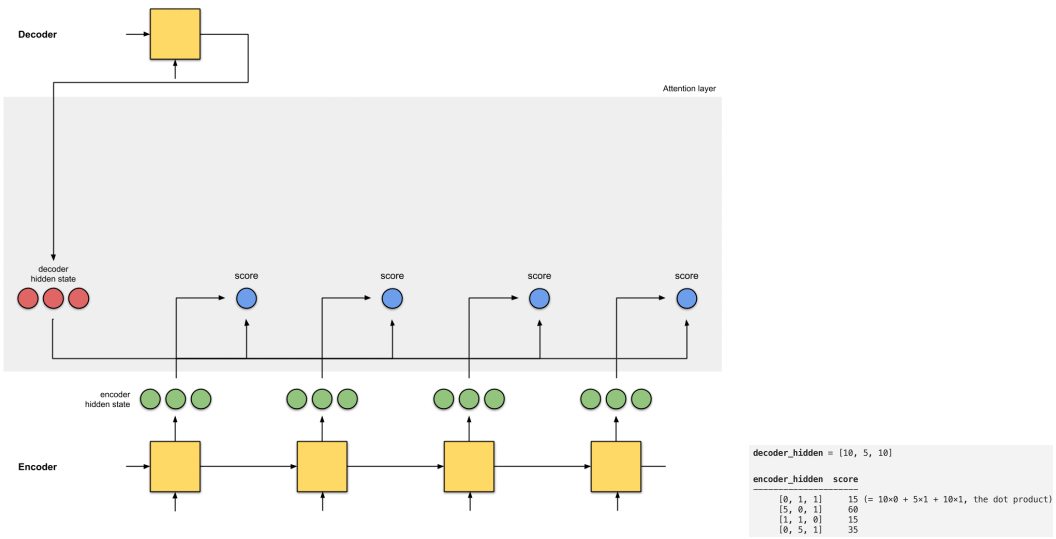
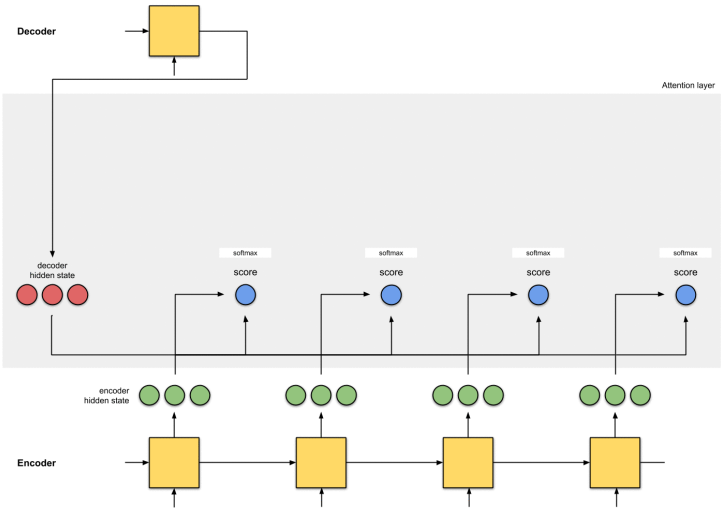
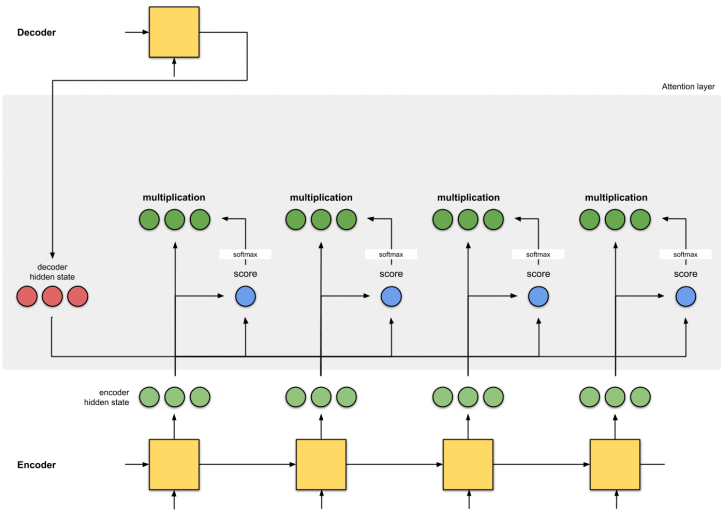


Figure 25.5: A **score** (scalar value) is obtained by a **score function**



encoder_hidden	score	score^
[0, 1, 1]	15	0
[5, 0, 1]	60	1
[1, 1, 0]	15	0
[0, 5, 1]	35	0

Figure 25.6: Put the scores to a softmax layer so that the softmaxed scores (scalar) add up to 1



encoder	score	score^	alignment
[0, 1, 1]	15	0	[0, 0, 0]
[5, 0, 1]	60	1	[5, 0, 1]
[1, 1, 0]	15	0	[0, 0, 0]
[0, 5, 1]	35	0	[0, 0, 0]

Figure 25.7: By multiplying each encoder hidden state with its softmaxed score (scalar) the alignment vector is obtained

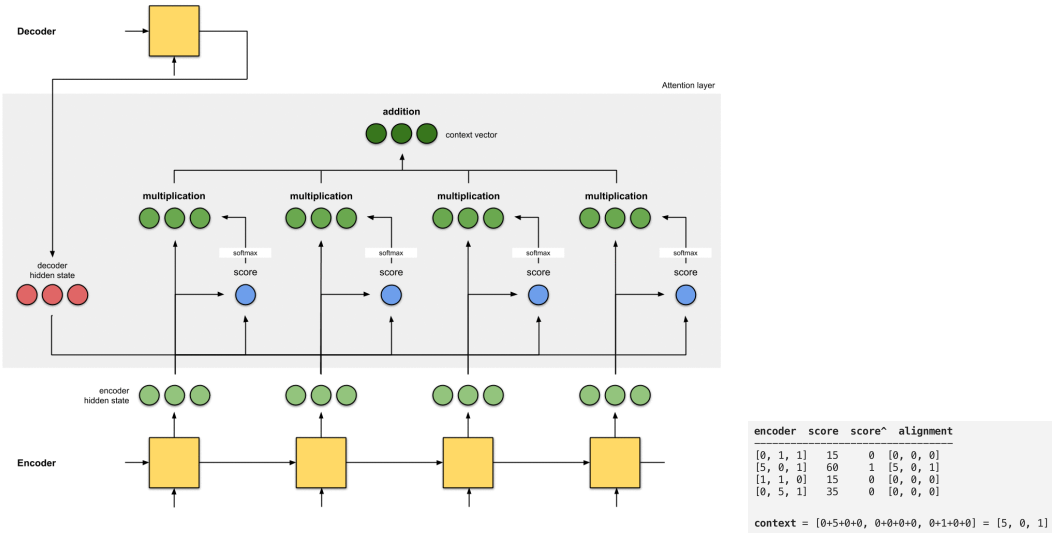


Figure 25.8: The alignment vectors are summed up to produce the context vector. A context vector is an aggregated information of the alignment vectors from the previous step

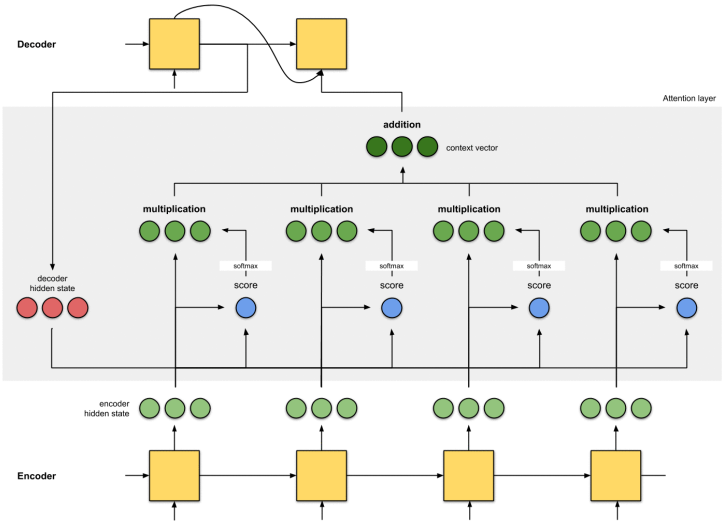


Figure 25.9: Feed the context vector into the decoder

Backpropagation will do whatever it takes to ensure that the outputs will be close to the ground truth. This is done by altering the weights in the RNNs and in the score function, if any. These weights will affect the encoder hidden states and decoder hidden states, which in turn affect the attention scores.

A Keras Cheat Sheet

Python For Data Science *Cheat Sheet*

Keras

Learn Python for data science Interactively at www.DataCamp.com

Keras

Keras is a powerful and easy-to-use deep learning library for Theano and TensorFlow that provides a high-level neural networks API to develop and evaluate deep learning models.

A Basic Example

```
>>> import numpy as np
>>> from keras.models import Sequential
>>> from keras.layers import Dense
>>> data = np.random.random((1000,100))
>>> labels = np.random.randint(2,size=(1000,1))
>>> model = Sequential()
>>> model.add(Dense(32,
                    activation='relu',
                    input_dim=100))
>>> model.add(Dense(1, activation='sigmoid'))
>>> model.compile(optimizer='rmsprop',
                  loss='binary_crossentropy',
                  metrics=['accuracy'])
>>> model.fit(data,labels,epochs=10,batch_size=32)
>>> predictions = model.predict(data)
```

Data

Also see NumPy, Pandas & Scikit-Learn

Your data needs to be stored as NumPy arrays or as a list of NumPy arrays. Ideally, you split the data in training and test sets, for which you can also resort to the `train_test_split` module of `sklearn.cross_validation`.

Keras Data Sets

```
>>> from keras.datasets import boston_housing,
                               mnist,
                               cifar10,
                               imdb
>>> (x_train,y_train),(x_test,y_test) = mnist.load_data()
>>> (x_train2,y_train2),(x_test2,y_test2) = boston_housing.load_data()
>>> (x_train3,y_train3),(x_test3,y_test3) = cifar10.load_data()
>>> (x_train4,y_train4),(x_test4,y_test4) = imdb.load_data(num_words=20000)
>>> num_classes = 10
```

Other

```
>>> from urllib.request import urlopen
>>> data = np.loadtxt(urlopen("http://archive.ics.uci.edu/
ml/machine-learning-databases/pima-indians-diabetes/
pima-indians-diabetes.data"),delimiter=",")
>>> X = data[:,0:8]
>>> y = data [:,8]
```

Model Architecture

Sequential Model

```
>>> from keras.models import Sequential
>>> model = Sequential()
>>> model2 = Sequential()
>>> model3 = Sequential()
```

Multilayer Perceptron (MLP)

Binary Classification

```
>>> from keras.layers import Dense
>>> model.add(Dense(12,
                    input_dim=8,
                    kernel_initializer='uniform',
                    activation='relu'))
>>> model.add(Dense(8,kernel_initializer='uniform',activation='relu'))
>>> model.add(Dense(1,kernel_initializer='uniform',activation='sigmoid'))
```

Multi-Class Classification

```
>>> from keras.layers import Dropout
>>> model.add(Dense(512,activation='relu',input_shape=(784,)))
>>> model.add(Dropout(0.2))
>>> model.add(Dense(512,activation='relu'))
>>> model.add(Dropout(0.2))
>>> model.add(Dense(10,activation='softmax'))
```

Regression

```
>>> model.add(Dense(64,activation='relu',input_dim=train_data.shape[1]))
>>> model.add(Dense(1))
```

Convolutional Neural Network (CNN)

```
>>> from keras.layers import Activation,Conv2D,MaxPooling2D,Flatten
>>> model2.add(Conv2D(32,(3,3),padding='same',input_shape=x_train.shape[1:]))
>>> model2.add(Activation('relu'))
>>> model2.add(Conv2D(32,(3,3)))
>>> model2.add(Activation('relu'))
>>> model2.add(MaxPooling2D(pool_size=(2,2)))
>>> model2.add(Dropout(0.25))

>>> model2.add(Conv2D(64,(3,3), padding='same'))
>>> model2.add(Activation('relu'))
>>> model2.add(Conv2D(64,(3, 3)))
>>> model2.add(Activation('relu'))
>>> model2.add(MaxPooling2D(pool_size=(2,2)))
>>> model2.add(Dropout(0.25))

>>> model2.add(Flatten())
>>> model2.add(Dense(512))
>>> model2.add(Activation('relu'))
>>> model2.add(Dropout(0.5))
>>> model2.add(Dense(num_classes))
>>> model2.add(Activation('softmax'))
```

Recurrent Neural Network (RNN)

```
>>> from keras.klayers import Embedding,LSTM
>>> model3.add(Embedding(20000,128))
>>> model3.add(LSTM(128,dropout=0.2,recurrent_dropout=0.2))
>>> model3.add(Dense(1,activation='sigmoid'))
```

Preprocessing

Also see NumPy & Scikit-Learn

Sequence Padding

```
>>> from keras.preprocessing import sequence
>>> x_train4 = sequence.pad_sequences(x_train4,maxlen=80)
>>> x_test4 = sequence.pad_sequences(x_test4,maxlen=80)
```

One-Hot Encoding

```
>>> from keras.utils import to_categorical
>>> Y_train = to_categorical(y_train, num_classes)
>>> Y_test = to_categorical(y_test, num_classes)
>>> Y_train3 = to_categorical(y_train3, num_classes)
>>> Y_test3 = to_categorical(y_test3, num_classes)
```

Train and Test Sets

```
>>> from sklearn.model_selection import train_test_split
>>> X_train5,X_test5,y_train5,y_test5 = train_test_split(X,
                                                         y,
                                                         test_size=0.33,
                                                         random_state=42)
```

Standardization/Normalization

```
>>> from sklearn.preprocessing import StandardScaler
>>> scaler = StandardScaler().fit(x_train2)
>>> standardized_X = scaler.transform(x_train2)
>>> standardized_X_test = scaler.transform(x_test2)
```

Inspect Model

`>>> model.output_shape`	Model output shape
`>>> model.summary()`	Model summary representation
`>>> model.get_config()`	Model configuration
`>>> model.get_weights()`	List all weight tensors in the model

Compile Model

MLP: Binary Classification

```
>>> model.compile(optimizer='adam',
                  loss='binary_crossentropy',
                  metrics=['accuracy'])
```

MLP: Multi-Class Classification

```
>>> model.compile(optimizer='rmsprop',
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])
```

MLP: Regression

```
>>> model.compile(optimizer='rmsprop',
                  loss='mse',
                  metrics=['mae'])
```

Recurrent Neural Network

```
>>> model3.compile(loss='binary_crossentropy',
                   optimizer='adam',
                   metrics=['accuracy'])
```

Model Training

```
>>> model3.fit(x_train4,
               y_train4,
               batch_size=32,
               epochs=15,
               verbose=1,
               validation_data=(x_test4,y_test4))
```

Evaluate Your Model's Performance

```
>>> score = model3.evaluate(x_test,
                            y_test,
                            batch_size=32)
```

Prediction

```
>>> model3.predict(x_test4, batch_size=32)
>>> model3.predict_classes(x_test4,batch_size=32)
```

Save/ Reload Models

```
>>> from keras.models import load_model
>>> model3.save('model_file.h5')
>>> my_model = load_model('my_model.h5')
```

Model Fine-tuning

Optimization Parameters

```
>>> from keras.optimizers import RMSprop
>>> opt = RMSprop(lr=0.0001, decay=1e-6)
>>> model2.compile(loss='categorical_crossentropy',
                   optimizer=opt,
                   metrics=['accuracy'])
```

Early Stopping

```
>>> from keras.callbacks import EarlyStopping
>>> early_stopping_monitor = EarlyStopping(patience=2)
>>> model3.fit(x_train4,
               y_train4,
               batch_size=32,
               epochs=15,
               validation_data=(x_test4,y_test4),
               callbacks=[early_stopping_monitor])
```

B Exercises

B.1 SW01 - Perceptron

Perceptron Learning Algorithm

B.2 SW02 - Numpy, Sigmoid & Gradient Descent

Numpy Broadcasting

Sigmoid Function

Students: Pascal Baumann, Claudio Paonessa

a)

Compute the derivative of the sigmoid function $\sigma(z) = \frac{1}{1+e^{-z}}$

$$
\begin{aligned}
\frac{d}{dz}\sigma(z) &= \frac{d}{dz}\left(\frac{1}{1+e^{-z}}\right) \\
&= \frac{d}{dz}(1+e^{-z})^{-1} \\
&= -1 \cdot (1+e^{-z})^{-2} \cdot \frac{d}{dz}(e^{-z}) \\
&= -1 \cdot (1+e^{-z})^{-2} \cdot -1 \cdot (e^{-z}) \\
&= (1+e^{-z})^{-2} \cdot (e^{-z}) \\
&= \frac{e^{-z}}{(1+e^{-z})^2}
\end{aligned}
$$

b)

Show that the derivative fullfills the equation $\sigma'(z) = \sigma(z) \cdot (1 - \sigma(z))$

$$
\begin{aligned}
1 - \sigma(z) &= 1 - \frac{1}{1+e^{-z}} \\
&= \frac{1+e^{-z}}{1+e^{-z}} - \frac{1}{1+e^{-z}} \\
&= \frac{1+e^{-z}-1}{1+e^{-z}} \\
&= \frac{e^{-z}}{1+e^{-z}} \\
\sigma'(z) &= \frac{e^{-z}}{(1+e^{-z})^2} \\
&= \frac{1}{1+e^{-z}} \cdot \frac{e^{-z}}{1+e^{-z}} \\
&= \sigma(z) \cdot (1 - \sigma(z))
\end{aligned}
$$

c)

Compute the first and second derivative of $\zeta(z) = -log(\sigma(-z))$

Compute the asymptotes for $z \to \pm\infty$. Create a plot of ζ.

$$\begin{aligned}\frac{d}{dz} - ln(\sigma(-z)) &= \frac{d}{dz} - ln\left(\frac{1}{1+e^z}\right) \\ &= -\left(\frac{1}{1+e^z}\right)^{-1} \cdot \left(\frac{1}{(1+e^z)^2}\right) \\ &= -(1+e^z) \cdot \frac{1}{(1+e^z)^2} \\ &= -\frac{1}{1+e^z}\end{aligned}$$

$$\begin{aligned}\frac{d^2}{dz^2} - ln(\sigma(-z)) &= \frac{d}{dz} - \frac{1}{1+e^z} \\ &= \frac{d}{dz}(-1) \cdot (1+e^z)^{-1} \\ &= (-1) \cdot (-1) \cdot (1+e^z)^{-2} \cdot e^z \\ &= \frac{e^z}{(1+e^z)^2}\end{aligned}$$

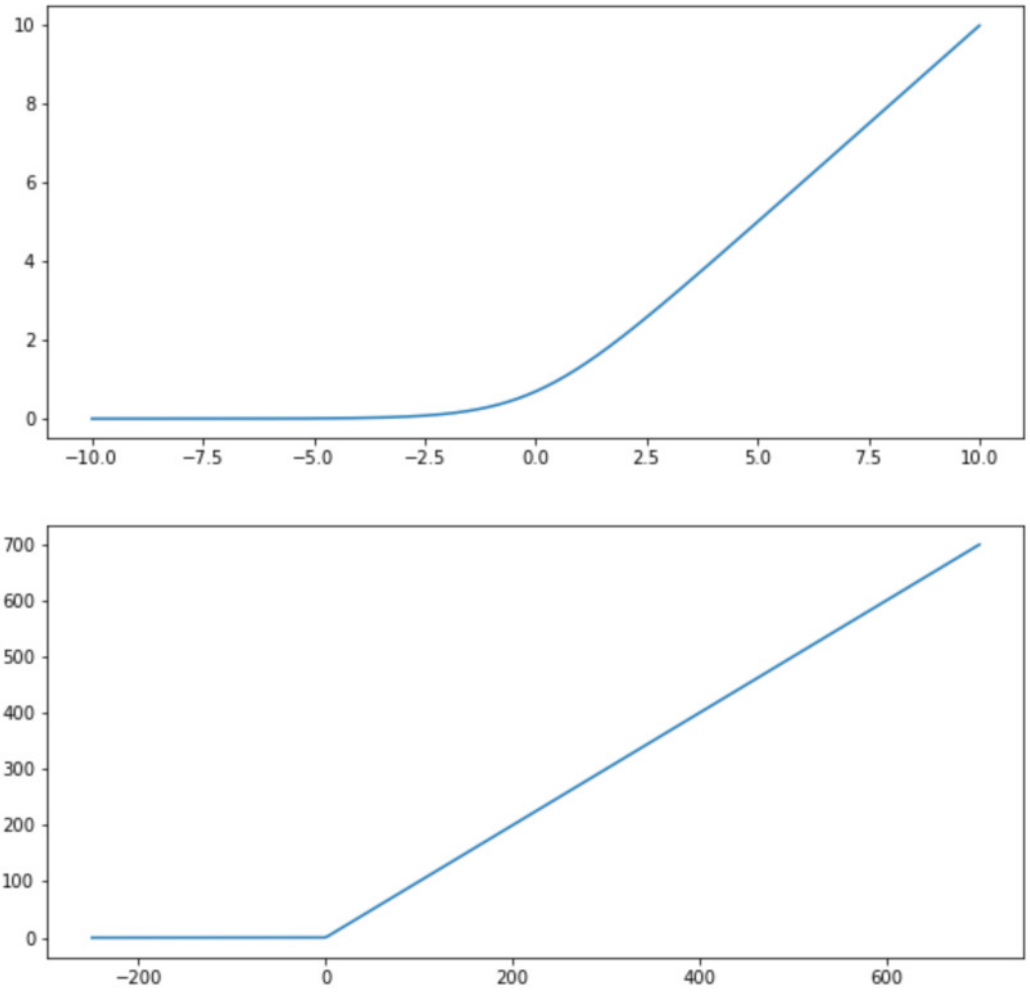
$$\begin{aligned}\lim_{z \to -\infty} - ln\left(\frac{1}{1+e^z}\right) &= -ln\left(\frac{1}{1+e^{-\infty}}\right) \\ &= -ln\left(\frac{1}{1+0}\right) \\ &= -ln(1) \\ &= 0 \\ \lim_{z \to -\infty} \frac{e^z}{1+e^z} &= \frac{e^{-\infty}}{1+e^{-\infty}} \\ &= 0\end{aligned}$$

Asymptote for minus infinity is thus the x-Axis.

$$\begin{aligned}\lim_{z \to \infty} - ln\left(\frac{1}{1+e^z}\right) &= -ln\left(\frac{1}{1+e^{\infty}}\right) \\ &= -ln\left(\frac{1}{1+\infty}\right) \\ &= -ln(0) \\ &= \infty \\ \lim_{z \to -\infty} \frac{e^z}{1+e^z} &= \frac{e^{\infty}}{1+e^{\infty}} \\ &= 1\end{aligned}$$

Asymptote for plus infinity is thus a line with an inclination of 45°

In [4]:
```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(-10,10,10000)
y = - np.log(1/(1+np.e ** x))
plt.figure(figsize=(10,10))
plt.subplot(2,1,1)
plt.plot(x,y)
plt.subplot(2,1,2)
x = np.linspace(-250,700,10000)
y = - np.log(1/(1+np.e ** x))
plt.plot(x,y)
plt.show()
```



d)

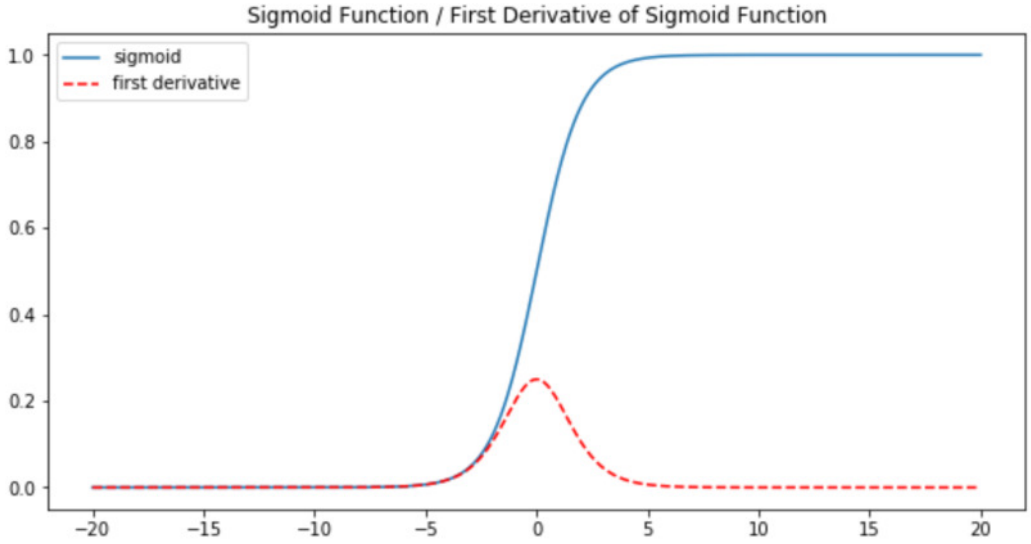
Implement the sigmoid function in a Jupyter Notebook. Make it work such that you can pass numpy arrays of arbitrary shape and the function is applied element-wise. Plot the sigmoid function and its derivative by using matplotlib.

In [22]:
```
def sigmoid(x):
    y = 1/(1 + np.e ** -x)
    return y

def first_derivative_sigmoid(x):
    y = sigmoid(x) * (1 - sigmoid(x))
    return y

def plot_sigmoid_and_derivative(x):
    y1 = sigmoid(x)
    y2 = first_derivative_sigmoid(x)
    plt.figure(figsize=(10,5))
    plt.title("Sigmoid Function / First Derivative of Sigmoid Function")
    plt.plot(x, y1, label='sigmoid')
    plt.plot(x, y2, c='r',linestyle='dashed', label='first derivative')
    plt.legend()
    plt.show()
```

In [23]:
```
x = np.array([1,2,3,4])
y = sigmoid(x)
print(y)
```

```
[0.73105858 0.88079708 0.95257413 0.98201379]
```

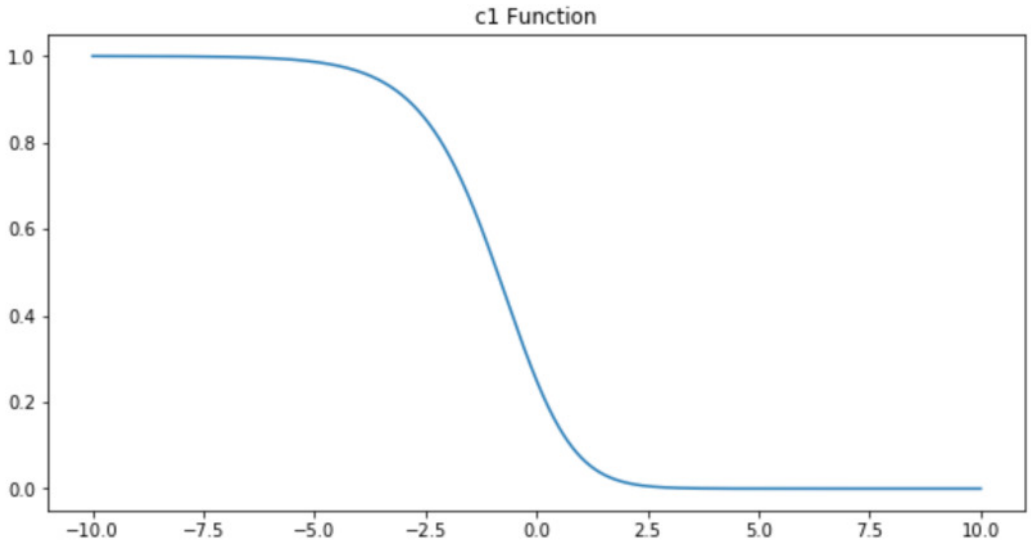
In [24]:
```
x = np.linspace(-20,20,1000)
plot_sigmoid_and_derivative(x)
```



f)

Show that the function $c_1(x) = (\sigma(x) - 1)^2$ is non-convex.

In [25]:
```
def c1function(x):
    return (sigmoid(x)-1)**2

x = np.linspace(-10,10,1000)
plt.figure(figsize=(10,5))
plt.title("c1 Function")
plt.plot(x,c1function(x))
plt.show()
```



Add $x = -1$ the gradient becomes zero, thus we could not optimise $c_1(x)$ more by descending the gradient.

g)

Compute the first and second derivative of the function

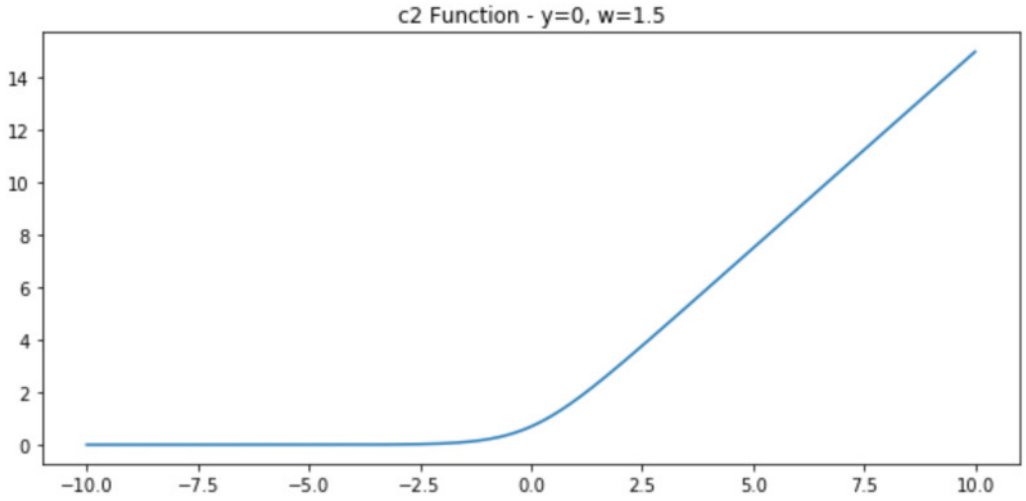
$$c_2(x) = -(y \cdot \log(\sigma(w \cdot x)) + (1 - y) \cdot \log(1 - \sigma(w \cdot x)))$$

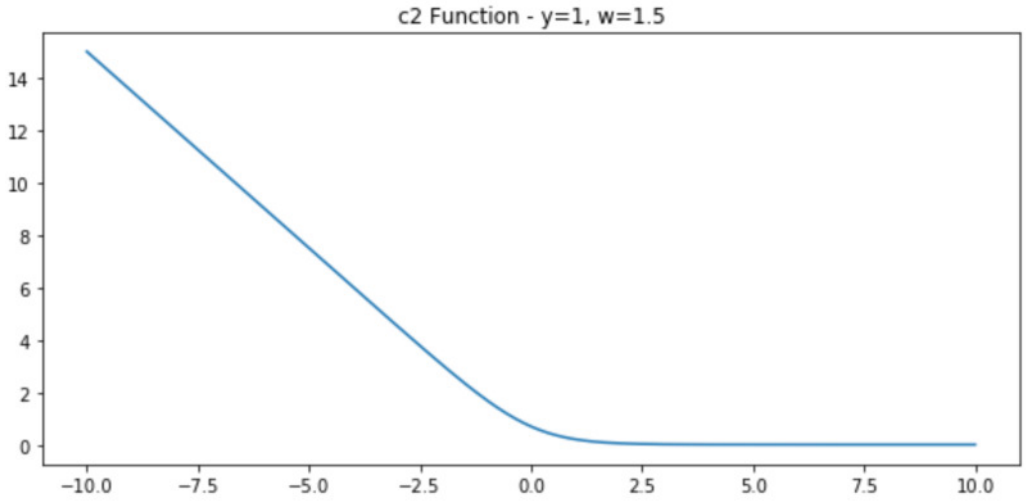
with respect to $w \in \mathbb{R}$ and for a given $y \in \{0, 1\}$. Show that c_2 is convex

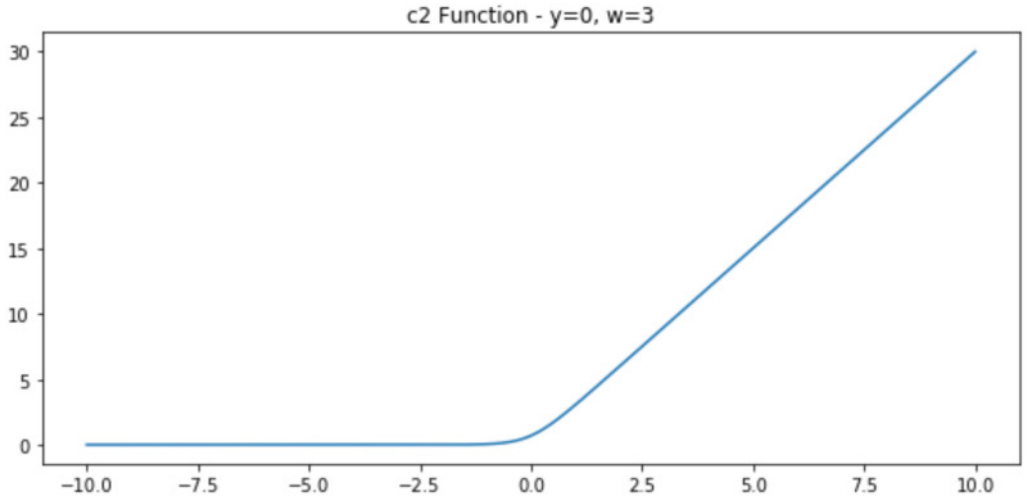
$$\frac{d}{dw} c_2 = -\frac{x((y-1)e^{wx} + y}{1 + e^{wx}}$$

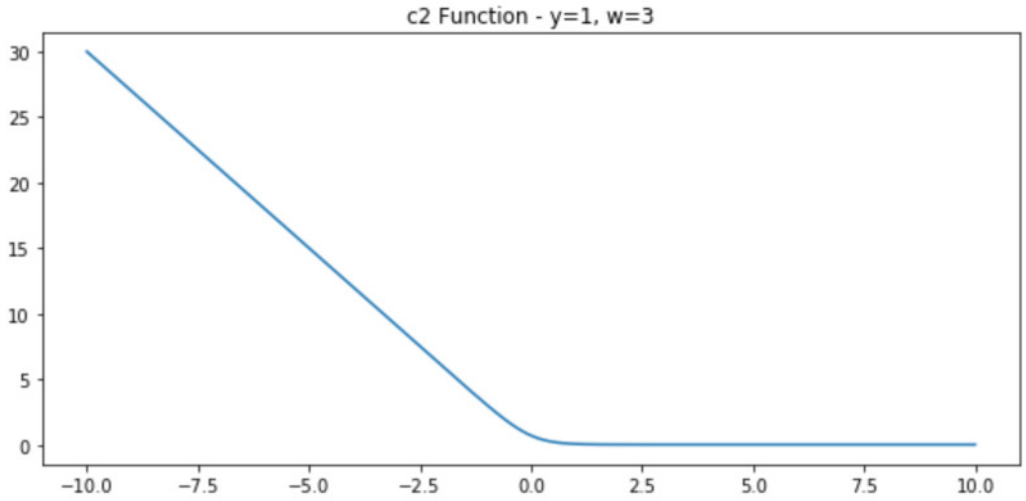
$$\frac{d^2}{dw^2} c_2 = \frac{x^2 e^{wx}}{(1 + e^{wx})^2}$$

In [40]:
```
def c2function(x,y,w):
    return -(y*np.log(sigmoid(w*x)) + (1-y)* np.log(1-sigmoid(w*x)))

x = np.linspace(-10,10,1000)
plt.figure(figsize=(10,10))
plt.subplot(2,1,1)
plt.title("c2 Function - y=0, w=1.5")
plt.plot(x,c2function(x,0,1.5))
plt.subplot(2,1,2)
plt.title("c2 Function - y=1, w=1.5")
plt.plot(x,c2function(x,1,1.5))
plt.show()
plt.figure(figsize=(10,10))
plt.subplot(2,1,1)
plt.title("c2 Function - y=0, w=3")
plt.plot(x,c2function(x,0,3))
plt.subplot(2,1,2)
plt.title("c2 Function - y=1, w=3")
plt.plot(x,c2function(x,1,3))
plt.show()
```


c2 Function - y=0, w=1.5
14
12
10
8
6
4
2
0
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0


c2 Function - y=1, w=1.5
14
12
10
8
6
4
2
0
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0

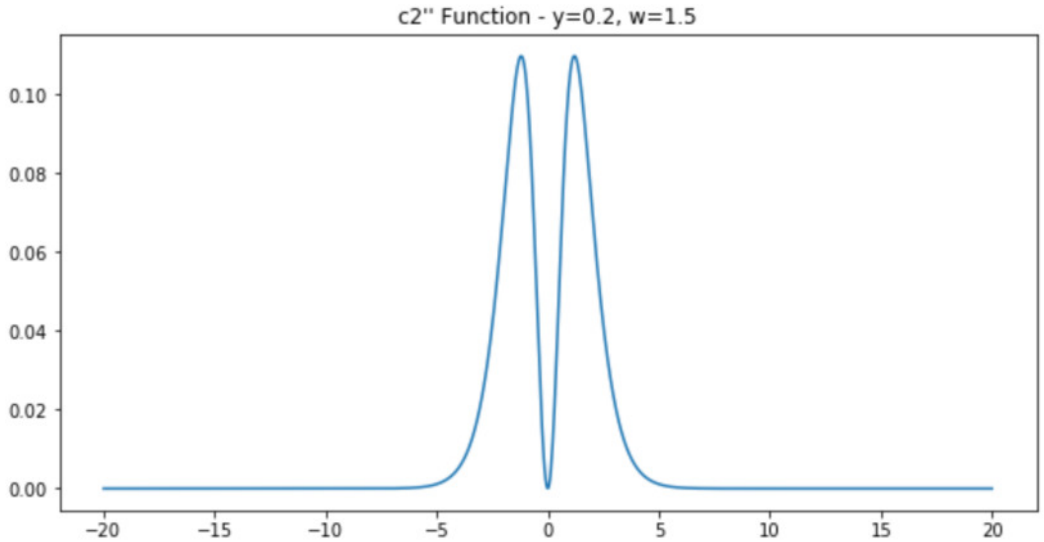




In [42]:
```
def c2function_2nd_derivative(x,y,w):
    return ((x**2) * (np.e ** (w*x)))/((np.e ** (w*x) + 1)**2)
```

In [44]:
```
x = np.linspace(-20,20,1000)
plt.figure(figsize=(10,5))
plt.title("c2'' Function - y=0.2, w=1.5")
plt.plot(x,c2function_2nd_derivative(x,0,2))
```

Out[44]:
```
[<matplotlib.lines.Line2D at 0x16f7f6c8888>]
```



The second derivative with respect to w of the function c2 can not be negative. This means the function c2 is convex.

In [ ]:

Cost Function Implementation

B.3 SW04 - Overfitting, Model Selection & Bayes Law

Handling Null Values

B.4 SW06 - Deep Learning Frameworks

Computational Graph Backpropagation

$$\frac{\delta h}{\delta h} = 1.0$$

$$\frac{\delta}{\delta x} x^{-1} = -x^{-2} = -1.0183^{-2} = -0.9644$$

$$\frac{\delta}{\delta x}(1 + x) = 1$$

$$\frac{\delta}{\delta x} e^x = e^x = e^{-4} = 0.0183$$

$$0.0183 \cdot -.9644 = -0.0176$$

$$\frac{\delta}{\delta x} - x = -1$$

$$\frac{\delta}{\delta x} x + b = 1$$

$$\frac{\delta}{\delta b} x + b = 1$$

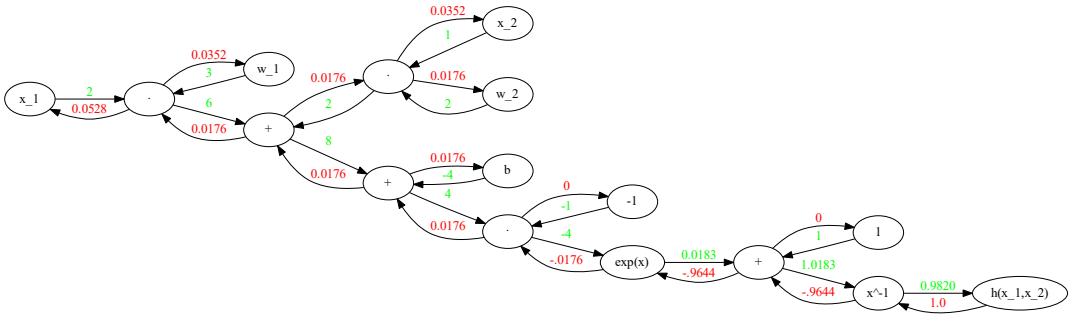
$$\frac{\delta}{\delta x} x + y = 1$$

$$\frac{\delta}{\delta y} x + y = 1$$

$$\frac{\delta}{\delta x} x \cdot w = w$$

$$\frac{\delta}{\delta w} x \cdot w = x$$

$$0.0176 \cdot w_1 = 0.0528, 0.0176 \cdot x_1 = 0.0352, 0.0176 \cdot w_2 = 0.0352, 0.0176 \cdot x_1 = 0.0176$$



MLP with Keras

B.5 SW07 - Regularisation & Optimisation

Glorot Initialisation

Optimisers

B.6 SW08 - CNN

Computation of Convolutions

Computation of Convolutions

Students: Pascal Baumann, Claudio Paonessa

a) 1D Example

$$(x \circ u)_i = \sum_{j=1}^{w} x_{(i-1+j)} \cdot u_j$$

i_1	i_2	i_3	i_4	i_5	i_6	i_7	i_8	i_9
1	3	-2	0	2	-1	3	1	2

w_1	w_2	w_3	b
2	1	-1	2

S=1, P=0

i_1	i_2	i_3	i_4	i_5	i_6	i_7	i_8	i_9
1	3	-2	0	2	-1	3	1	2

$$(1 \cdot 2 + 3 \cdot 1 + -2 \cdot -1) + 2 = 9$$
$$(6 + -2 + 0) + 2 = 6$$
$$(-4 + 0 + -2) + 2 = -4$$
$$\vdots$$

o_1	o_2	o_3	o_4	o_5	o_6	o_7
9	6	-4	5	2	2	7

S=2, P=0

i_1	i_2	i_3	i_4	i_5	i_6	i_7	i_8	i_9
1	3	-2	0	2	-1	3	1	2

$$(1 \cdot 2 + 3 \cdot 1 + -2 \cdot -1) + 2 = 9$$
$$(-4 + 0 + -2) + 2 = -4$$
$$\vdots$$

o_1	o_2	o_3	o_4
9	-4	2	7

S=4, P=0

i_1	i_2	i_3	i_4	i_5	i_6	i_7	i_8	i_9
1	3	-2	0	2	-1	3	1	2

$$(1 \cdot 2 + 3 \cdot 1 + -2 \cdot -1) + 2 = 9$$
$$(4 + 1 + -3) + 2 = 2$$
$$(4 + ? + ?) + 2 = ?$$

o_1	o_2	o_3
9	2	-

! Incompatible stride value !

S=1, P=1

p_1	i_1	i_2	i_3	i_4	i_5	i_6	i_7	i_8	i_9	p_2
0	1	3	-2	0	2	-1	3	1	2	0

$$(0 \cdot 2 + 1 \cdot 1 + 3 \cdot -1) + 2 = 0$$
$$(2 + 3 + 2) + 2 = 9$$
$$(6 + -2 + 0) + 2 = 6$$
$$(-4 + 0 + -2) + 2 = -4$$
$$\vdots$$

o_1	o_2	o_3	o_4	o_5	o_6	o_7	o_8	o_9
0	9	6	-4	5	2	2	7	6

S=4, P=1

p_1	i_1	i_2	i_3	i_4	i_5	i_6	i_7	i_8	i_9	p_2
0	1	3	-2	0	2	-1	3	1	2	0

$$(0 \cdot 2 + 1 \cdot 1 + 3 \cdot -1) + 2 = 0$$
$$(0 + 2 + 1) + 2 = 5$$
$$(2 + 2 + 0) + 2 = 6$$

o_1	o_2	o_3
0	5	6

As we can see, we only get the same Dimensions for a stride value of 1 and padding 1

b) 2D Example

- We will get 2 activation maps (RGBFilter1, RGBFilter2)
- 3x3x2
- 2x2x2
- 2x2x3, S=2, P=2

CNN in Keras

```
Model: "sequential_12"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_20 (Conv2D)           (None, 32, 32, 32)        896
_________________________________________________________________
activation_22 (Activation)   (None, 32, 32, 32)        0
_________________________________________________________________
max_pooling2d_8 (MaxPooling2 (None, 16, 16, 32)        0
_________________________________________________________________
flatten_3 (Flatten)          (None, 8192)              0
_________________________________________________________________
dense_5 (Dense)              (None, 512)               4194816
_________________________________________________________________
activation_23 (Activation)   (None, 512)               0
_________________________________________________________________
dropout_9 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_6 (Dense)              (None, 10)                5130
_________________________________________________________________
activation_24 (Activation)   (None, 10)                0
=================================================================
Total params: 4,200,842
Trainable params: 4,200,842
Non-trainable params: 0
```

B.7 SW09 - CNN with Keras

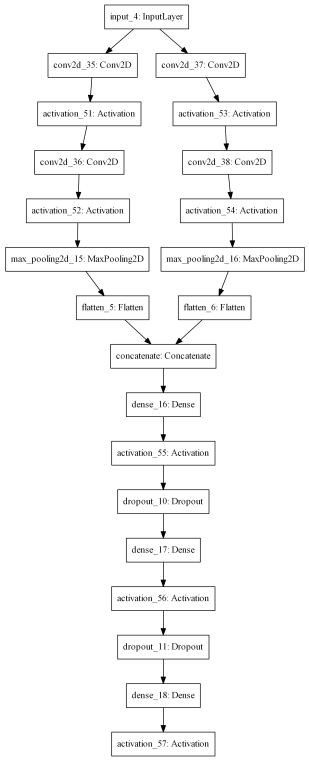
New Adam Optimiser

- The loss is relevant for multiclass, single-label classification problem. Categorical is used because there are 10 classes to predict from. If there were 2 classes, we would have used .
- The optimizer is an improvement over SGD(Stochastic Gradient Descent). The optimizer is defining the update rule for the weights of the neurons during backpropagation gradients.

Data Augmentation Pipeline / Generator

B.8 SW10 - CNN with Keras II

Functional API nonlinear Graph



B.9 SW12 - Recurrent Neural Networks

SimpleRNN with Keras

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
simple_rnn (SimpleRNN)       (None, 64)                8960
_________________________________________________________________
dense (Dense)                (None, 5)                 325
=================================================================
Total params: 9,285
Trainable params: 9,285
Non-trainable params: 0
```

Stacked LSTM with Keras

```
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 128, 9)]          0
_________________________________________________________________
lstm (LSTM)                  (None, 128, 32)           5376
_________________________________________________________________
lstm_1 (LSTM)                (None, 128, 32)           8320
_________________________________________________________________
lstm_2 (LSTM)                (None, 32)                8320
```

```
_________________________________________________________________
dense (Dense)                (None, 6)                 198
=================================================================
Total params: 22,214
Trainable params: 22,214
Non-trainable params: 0
```

Findings for Human Activity Recognition As one can see from the myriad of models, 93% seems to be the best accuracy we can achieve on the test set. The most complicated cell (LSTM) seems to deliver the best (consistent) performance. Stacking more than three layers or increasing the number of hidden units beyond 32 seems not to increase the performance however.

Transfer Learning with Keras

```
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         [(None, 100)]             0
_________________________________________________________________
embedding (Embedding)        (None, 100, 100)          500000
_________________________________________________________________
lstm (LSTM)                  (None, 100, 128)          117248
_________________________________________________________________
lstm_1 (LSTM)                (None, 100, 64)           49408
_________________________________________________________________
lstm_2 (LSTM)                (None, 32)                12416
_________________________________________________________________
dense (Dense)                (None, 1)                 33
=================================================================
Total params: 679,105
Trainable params: 179,105
Non-trainable params: 500,000
```

B.10 SW13 - Recurrent Neural Networks II

Sequence Generator - Many To Many Approach

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
gru_1 (GRU)                  (None, 10, 64)            39744
_________________________________________________________________
time_distributed_1 (TimeDist (None, 10, 141)           9165
=================================================================
Total params: 48,909
Trainable params: 48,909
Non-trainable params: 0
```

Findings Sequence Generator The Many-to-One architectures work fine, the stacked GRUs give the most plausible results (given that most of them sound like reasonable company names). After fixing the prediction code for the Many-to-Many approach (by taking the last prediction). We think for this short of a sentence the Many-to-Many approach does not have a big advantage over the Many-to-One, which we can see in our results.