Chapter 6

Neural Networks

Computers are better than humans at playing Chess, Go, Poker, Dota, or Starcraft. They compose pop songs, write fiction stories, draw paintings, replace actors in movies, and drive vehicles. Whenever a computer does something mind-boggling, you can bet that a neural network is involved. Neural networks have become fascinating function approximators. How so? At their core, neural networks are based on simple linear mappings, combined with non-linear activation functions and gradient descent. So conceptually neural networks are not so different from our discussions in Chapter 5. But size matters! The biggest neural networks have up to 175 billion weights. So training needs data, hardware and patience.

6.1 Nodes and Networks

Definition 6.1 (Node)**.** *A node (or neuron) is a computing unit v that produces an activation value y. The node v first calculates an affine transformation on $\boldsymbol{x} \in \mathbb{R}^d$, then applies an activation function $\sigma : \mathbb{R} \to \mathbb{R}$:*

$$y = \sigma(\boldsymbol{w}^T \boldsymbol{x}),$$

where $\boldsymbol{x}$ is an input vector and $\boldsymbol{w} \in \mathbb{R}^d$ are learned weights. We call $z = \boldsymbol{w}^T \boldsymbol{x}$ the pre-activation value. Like in Chapter 5 (Definitions 5.5 and 5.30), we assume that w_0 is integrated into $\boldsymbol{w}$, i.e., $\boldsymbol{w} = (w_0, w_1, \ldots, w_{d-1})^T$, and $\boldsymbol{x}$ includes an additional constant 1, i.e., $\boldsymbol{x} = (1, x_1, \ldots, x_{d-1})^T$.

Remarks:

- In the literature, the intercept w_0 is sometimes referred to as "bias" b, and kept separate from $\boldsymbol{w}$, i.e., $y = \sigma(\boldsymbol{w}^T \boldsymbol{x} + b)$. This naming complicates the vector notation; it may also be confusing since we used the term bias for a model property in Section 5.4.
- The activation function σ can take many different forms. Some nodes may simply use the identity as activation function, i.e., $\sigma(z) = z$. Most nodes apply non-linear activation functions in order to allow the model to approximate non-linear functions, e.g. the sigmoid function $\psi(\boldsymbol{w}^T \boldsymbol{x}) = \frac{1}{1+\exp(-\boldsymbol{w}^T \boldsymbol{x})}$ of Definition 5.30.

- In order to allow for gradient-based training, the activation function must be differentiable.
- We combine many neural nodes into a network:

Definition 6.2 (Neural Network)**.** *A neural network is a directed acyclic graph (DAG) formed by a set of nodes V that are connected by a set of directed edges E. The input $\boldsymbol{x}$ of the network is stored by the n input nodes V_i (with no incoming edges). The output $\boldsymbol{y}$ of the network will be computed by the m output nodes V_o (with no outgoing edges). All other nodes (with incoming and outgoing edges) are called hidden nodes V_h. We have $V_i + V_h + V_o = V$. Note that we use the letters x and y to refer to both, the input and output of the whole network as well as the input and output of a single node. The we will use subscripts if the usage is not clear from the context.*

In neural networks the function is computed in the forward direction: The input $\boldsymbol{x}_v$ of each node v is the vector of computed outputs $\boldsymbol{y}$ of its DAG predecessor nodes. Then, v computes its own output as $y_v = \sigma(\boldsymbol{w}_v^T \boldsymbol{x}_v)$. Hence then nodes must be processed in DAG order.

Given an input $\boldsymbol{x} \in \mathbb{R}^n$, a neural network as a whole then approximates a function $f(\boldsymbol{x}) : \mathbb{R}^n \to \mathbb{R}^m$ by calculating $\hat{f}(\boldsymbol{x}) \approx f(\boldsymbol{x})$.

```
1  # V = V_i ∪ V_h ∪ V_o  = network
2  # v.x = v's input, the output of v's DAG input-nodes
3  def forward(V):
4      for v in V_h ∪ V_o (in DAG order):
5          v.y = v.σ(v.w,v.x)   # Definition 6.1
6      return V_o
```

Algorithm 6.3: Feed-forward computation in DAG.

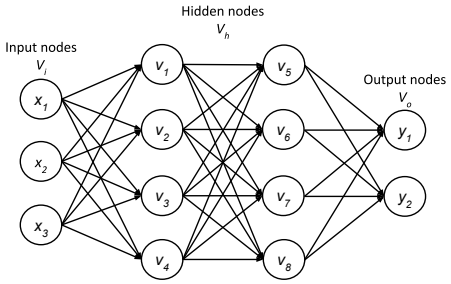

Figure 6.4: Example of a neural network for an input $\boldsymbol{x}$ and an output $\boldsymbol{y}$ with three input nodes, eight hidden nodes and two output nodes.

Remarks:

- There exist cyclic neural networks as well, see Definition 6.25.
- In neural networks, nodes generally follow a structure that can be represented with layers.

Definition 6.5 (Multi-Layer Perceptron or MLP)**.** *The nodes are often organized in layers. The first layer are the input nodes V_i, the last layer the output nodes V_o. Each hidden node is in a layer l, and all nodes of layer l have the same input, namely the outputs y_v of all nodes v of layer $l-1$. Layered networks are known as Multi-Layer Perceptrons, where perceptron is an older name for node.*

Remarks:

- Layering will help us to speed up computation, as the pre-activation of a whole layer can be computed with a single matrix-vector multiplication $\boldsymbol{z} = \mathbf{W} \cdot \boldsymbol{x}$, where the matrix $\mathbf{W}$ is composed of the weight rows $\boldsymbol{w}$, $\boldsymbol{z}$ is the vector of pre-activation values in the nodes and $\boldsymbol{x}$ are the output values of the previous layer (with the additional 1).
- The number of layers determines the depth of the network. A "deep" network is a neural network with multiple layers.
- Can a neural network compute/approximate any function?

6.2 Power and Limitations

Theorem 6.6 (Universal Approximation Theorem)**.** *Given a continuous function $f : \mathbb{R}_+ \to \mathbb{R}$ (for simplicity, input $x \geq 0$), there exists a neural network $\hat{f}$ with one hidden layer that approximates f arbitrarily well. That is*

$$|f(x) - \hat{f}(x)| < \varepsilon \text{ for all } x \geq 0 \text{ and } \varepsilon > 0.$$

Proof. We construct a neural network with sigmoid non-linearities in the hidden nodes v_i ($i > 0$) and a single linear output node v_o.

The single output node v_o computes function $\hat{f}(x) = \boldsymbol{w}^T \boldsymbol{y}$, where $\boldsymbol{w}$ are v_o's weights, and y_i are the outputs produced by the hidden nodes. As usual, $\boldsymbol{w}$ includes an additional value w_0, and $\boldsymbol{y}$ starts with a additional constant 1. We choose $w_0 = f(0)$, such that $\hat{f}(0) = f(0)$ even without any hidden nodes. Every hidden node v_i computes $y_i = g_i(x) = \psi(\kappa \cdot x + b_i)$, where $\kappa \to \infty$ is a large constant. While the sigmoid function $\psi(z) = \frac{1}{1+\exp(-z)}$ from Definition 5.30 is a smooth step function, $\kappa \to \infty$ will make that step sharp.

The construction is inductive. We start out with $x = 0$, hence $\hat{f}(x) = f(x)$. As long as the difference between $\hat{f}(x)$ and $f(x)$ is less than ε we keep growing x. As soon as $|f(x) - \hat{f}(x)| \geq \varepsilon$, we introduce a new hidden node v_i. The value b_i of the hidden node v_i is representing the current position x, as $b_i = -\kappa \cdot x$. This makes sure that v_i will introduce a new step in $\hat{f}$ right at the current x. The weight w_i of v_o for the new input y_i is set as $w_i = f(x) - \hat{f}(x)$. This corrects the output of $\hat{f}$ for the newly accumulated error. If $f(x)$ was increasing, then a correcting $+\varepsilon$ step is added, if $f(x)$ was decreasing, a correcting $-\varepsilon$ step is added. In both cases, we again get $\hat{f}(x) \approx f(x)$. Figure 6.7 visualizes the effects of the parameters b_i and w_i. □

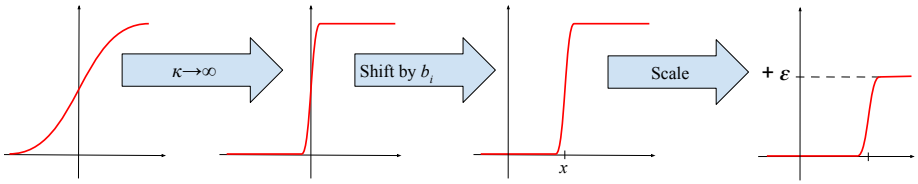

Figure 6.7: Effect of weights on the sigmoid function.

Remarks:

- Our proof is a simplified version of the original. The full theorem is more general, also applicable to continuous functions with inputs from a multi-dimensional compact set. There exist also versions using other activation functions than sigmoid.
- The theoretical construction given in the proof is not used in practice, as it would lead to numerical instabilities ($\kappa \to \infty$).
- What is the minimum size of a neural network in order to approximate a function f?

Definition 6.8 (VC-dimension)**.** *The Vapnik–Chervonenkis dimension is a measure of the complexity/capacity of a model $\mathcal{F}$. Given an arbitrary input set X of size n. Assume that each input $x_i \in X$ maps to a binary classification output $y_i \in \{0, 1\}$. If for all 2^n possible outputs there exist a function $f \in \mathcal{F}$ that correctly classifies all inputs, i.e., $f(x_i) = y_i$ for all $x_i \in X$, then the model $\mathcal{F}$ has (at least) VC-dimension n. The VC dimension is therefore defined as the size of the largest set for which there exists a classifier $f \in \mathcal{F}$ for any labeling of the set.*

Example 6.9. *Let $\mathcal{F}$ be the family of interval classifiers given by*

$$f_{\alpha,\theta}(x) = \begin{cases} 1 & \text{for } x \in [\theta, \theta + \alpha] \\ 0 & \text{otherwise} \end{cases}$$

The VC dimension of this classifier is at least 2, as for 2 data points x_1 and x_2 we can always choose θ and α to either include none, one or both of the points. Therefore any labeling of 2 points can be represented by an interval classifier $f \in \mathcal{F}$. However, for 3 points x_1, x_2 and x_3 with $x_1 < x_2 < x_3$ we cannot define an interval that includes x_1 and x_3 but not x_2. The labeling $y_1 = 1, y_2 = 0, y_3 = 1$ can therefore not be represented and we conclude that the VC dimension of $\mathcal{F}$ is 2.

Example 6.10. *Let $\mathcal{F}$ be the family of trees with n leaves which classify a scalar input $x \in \mathbb{R}$, i.e. the n leave nodes are labeled with either 0 or 1. Traversing this tree from the root to the leaves we can see that each internal node divides the remaining interval into two sub-intervals. This means that the whole tree divides the real number line into n distinct intervals whose boundaries we can choose by the threshold weights of the nodes in the tree. We can therefore construct a classifier from $\mathcal{F}$ that surrounds each data point in a set of n points with a*

correctly labeled interval to reproduce all possible labelings (as we are also free to choose the label on the leave nodes). The VC dimension of $\mathcal{F}$ is therefore at least n. On the other hand, the VC dimension cannot be larger than n. Consider $n+1$ ordered data points with alternating labels. It follows that no two points may be in the same interval. Thus we would need $n+1$ many intervals, which is impossible with n leaves.

Remarks:

- As can be seen from the examples, the VC dimension is not a measure specific to neural networks, but can be applied to any model $\mathcal{F}$.
- With a sigmoid activation function, a neural network with w weights has at least VC-dimension w^2. A network with a simple sign activation function on the other hand has at most VC-dimension $w \log w$.
- The VC dimension can be generalized in various ways, e.g. beyond binary classification, or by restricting the possible outputs.
- However, the promise of a neural network is not only to approximate any function, but rather to *learn* how to approximate any function.

6.3 Training Neural Networks

During training, neural networks learn to automatically extract features from the raw input: In the forward computation the representation of the data held by the network becomes progressively closer to the value of the approximated function. Therefore, neural networks are effectively feature extractors.

Definition 6.11 (Feature Extractor ϕ)**.** *A feature extractor ϕ is a function that transforms raw input data $\boldsymbol{x}$ into features. These features represent the initial data in a way that simplifies approximating a function f.*

Remarks:

- In Section 5.2 we manually engineered ϕ. In Sections 5.3 and 5.4 we then learned how to tell whether we did a good job.
- Neural networks on the other hand automatically learn ϕ and thus, no manual feature extraction is needed. The idea is that every additional hidden layer of the network represents the data more abstractly than the previous one. With every layer, the representation of the data is less like input $\boldsymbol{x}$ and more like output $\boldsymbol{y}$.
- Why and how exactly this works is not well understood – this is the *mystique* of deep neural networks.
- Training a neural network is similar to training a linear regression node with gradient descent (Chapter 5): We calculate the gradient of the loss with respect to the network parameters and adjust the parameters accordingly. This is called backpropagation.

Definition 6.12 (Backpropagation)**.** *Backpropagation is an algorithm that computes the gradient of the loss L with respect to the parameters W of the neural network. In the DAG representation of a neural network, each node computes its pre-activation value z as a weighted sum over its input values $\boldsymbol{x}$. By the chain rule, we can calculate the error of the output nodes with respect to z as*

$$\frac{\partial L(\hat{f}, D)}{\partial z} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial z}.$$

Given this gradient, and using $\boldsymbol{z} = \boldsymbol{w}^T \boldsymbol{x}$, we can calculate the error with respect to the weights w_i and the error with respect to the node's inputs x_i as

$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial z} \cdot x_i, \text{ and } \frac{\partial L}{\partial x_i} = \frac{\partial L}{\partial z} \cdot w_i.$$

The gradients with respect to the weights $\boldsymbol{w}$ can then be used for updating the weights, while the gradient with respect to the inputs can be aggregated to pass the gradient $\frac{\partial L}{\partial y}$ to the preceding nodes in the network. Concretely, each node in the network adjusts its own weights $\boldsymbol{w}$ based on the error signal $\frac{\partial L}{\partial w}$ and tells its input nodes x_i to adjust by backpropagating $\frac{\partial L}{\partial x_i}$. The backpropagation algorithm is given in Algorithm 6.13.

```
# v.x = v's input, stored during forward computation
# v.z = pre-activation value, stored during forward computation
# v.err = initial error of the node, set to ∂L/∂y for output nodes
# v.err = 0 initially for hidden and input nodes
# v.w_grad = gradient vector of node v
# v.prev = list of indices of v's input nodes
def backward(V):
    for v in V (in reversed DAG order):
        v.z_err = v.err * ∂v.y/∂v.z  # ∂y/∂z is the gradient of σ(z)
        v.w_grad = v.z_err * v.x  # v.w_grad and v.x are vectors
        for v_i in v.prev:
            v_i.err += v.z_err * v.w[v_i]
    return [v.w_grad for v in V]
```

Algorithm 6.13: Backpropagation Algorithm

Remarks:

- Many libraries backpropagate gradients with a simple function call. → notebook
- Memory may be problem, since we need to memorize $\boldsymbol{x}$ and z at every node.
- Backpropagation is only the method for computing the gradient, while another algorithm, such as stochastic gradient descent (Definition 5.26), is used to perform learning using this gradient.

- Backpropagating gradients, i.e., applying the chain rule, means performing a number of multiplications. For deep neural networks this can lead to numerical issues.

Definition 6.14 (Vanishing Gradients)**.** *Gradient descent updates can stagnate due to vanishing gradients, i.e., gradients that are close to 0. These can occur during backpropagation for different reasons:*

- *The activation function $\sigma(\cdot)$ saturates, i.e. the gradient $\frac{\partial y}{\partial z} \approx 0$. In this case, the gradient $\frac{\partial L}{\partial w_i}$ of all weights w_i and the backpropagated gradients $\frac{\partial L}{\partial x_i}$ of the node will also be close to zero. The node stops learning.*
- *The summation in Line 12 of Algorithm 6.13 can accidentally become 0 (when terms cancel each other).*

Definition 6.15 (Exploding Gradients)**.** *The gradient calculations with backpropagation can also lead to devastatingly large gradients, called exploding gradients. Note that gradient descent only converges for sufficiently small gradient steps and too large gradients can lead to divergence. Reasons are:*

- *Some large weight $|w_i| \gg 1$ boosts the backpropagated error.*
- *The summation in Line 12 of Algorithm 6.13 can accidentally become large because of many positive (or negative) terms.*

Remarks:

- Vanishing or exploding gradients can propagate in a network, i.e. nodes with vanishing or exploding gradients can backpropagate the problem to their predecessor nodes.
- Simple yet generally effective solutions include reducing the number of layers or clipping the gradients (for the exploding case). A related solution is to normalize the activation values, as done in techniques such as *batch normalization* or *layer normalization*.
- Another effective solution for the vanishing gradient problem is the introduction of skip connections (connect nodes which are not in neighboring layers) which provide an additional path for the flow of information. During backpropagation this helps the gradients to continuously flow backwards, even if they vanish in certain points of the network.
- An additional option for mitigating the vanishing gradients problem is to use activation functions that do not saturate in both directions (positive and negative). The best-known of such activation functions is the Rectified Linear Unit activation (ReLU).

Definition 6.16 (ReLU)**.** *The Rectified Linear Unit activation is defined as:*

$$\sigma(x) = \max(0, x)$$

Remarks:

- The ReLU activation function introduces a nonlinear transformation that remains very close to linear (piecewise linear with only two pieces) and does not saturate in the positive direction.
- Remarkably, ReLU is non-differentiable, which violates Definition 6.1. In fact, for gradient-based learning it is enough if the subderivatives of the function exist (and for ReLU they do).
- The gradient $\frac{\partial y}{\partial z}$ of the ReLU activation is 0 when $x \leq 0$, 1 otherwise. This simplicity speeds up the computation of backpropagation.
- ReLU activation is the current default choice for neural networks. However, a large number of related activation functions exist, which modify some aspects of the function, e.g., leakyReLU has a non-zero slope for values smaller than 0. Some other functions are: PReLU, GeLU, SeLU, Maxout, etc.
- Another design choice that impacts the performance of the model is the initialization scheme.

Definition 6.17 (Initialization scheme)**.** *Rule that determines the initial parameter values W of a neural network, i.e., the values before training starts.*

Remarks:

- As seen in Figure 5.27, when the loss function is non-convex (has multiple local minima), starting the learning process at different points can lead to different solutions.
- Stochastic initialization is a good default for initializing the parameters of a neural network. These schemes give random initial values (with some constraints) to the parameters of the network in order to "break the symmetry", i.e., to prevent that nodes with the same input and same activation converge to the same values during optimization.
- The loss landscape of neural networks is complex, with a large number of local minima. Surprisingly, converging to a local minimum during training is good enough for a neural network to perform well usually.

6.4 Practical Considerations

The complex loss landscape of neural networks makes the learning process significantly more complicated than in classical machine learning models. Therefore, sophisticated learning algorithms (also called optimizers) that build on top of Stochastic Gradient Descent (Section 5.6) are used in practice. There is no consensus on which of the existing algorithms is best.

Remarks:

- Adam optimizer is currently considered a good default. It belongs to the family of adaptive learning rate algorithms, which adapt the learning rate for each parameter individually throughout the course of learning.
- Other popular optimizers include SGD with Momentum, RMSProp, and linear learning rate decay.
- The learning scheme/rate is probably the most important hyperparameter in neural networks. Finding an appropriate learning rate can produce a dramatic improvement in the performance of the network.
- Neural networks often have a remarkably large amount of hyperparameters, which do have a strong impact on the performance of the model. Although tuning hyperparameters is more and art than a science there are automatic hyperparameter optimization algorithms that can help in this process.

Definition 6.18 (Hyperparameter Optimization Algorithm)**.** *A hyperparameter optimization algorithm is an algorithm that wraps the learning algorithm of a model and chooses its hyperparameters, hiding this choice from the user.*

Remarks:

- When there are few hyperparameters to set, a common approach is Grid Search as discussed in Definition 5.24. The main problem of Grid Search is that the computational cost grows exponentially with the number of hyperparameters, which makes it expensive for large neural networks.
- An alternative is Random Search: the hyperparameter values are samples from a uniform distribution in a certain interval. Random search converges faster to an optimum.
- A large number of hyperparameter optimization algorithms exist using techniques such as evolutionary algorithms, Bayesian optimization or population-based-training.
- Hyperparameter optimization algorithms often have their own hyperparameters, such as the range of values that have to be explored. Fortunately, these secondary hyperparameters are easier to set in the sense that similar secondary hyperparameters can lead to acceptable performance in a wide range of tasks.

6.5 Regularization

Everything discussed so far portraits neural networks as powerful function approximators. Neural networks can approximate any continuous functions and even functions of high complexity, e.g., functions from a function class with high VC-dimension. The issue with this is that neural networks tend to overfit.

To give an intuitive explanation why this is the case, recall the bias-variance trade-off from the previous chapter. There, we saw that polynomials of too high degree yield a high variance which leads to a bad generalization performance. Now, the universal approximation theorem states that a sufficiently large neural network can approximate any continuous function. Hence, a sufficiently large neural network can also approximate any polynomial. Without any restrictions, the variance of a neural network can be very high and the generalization performance very poor.

- To prevent it, classical parameter norm penalty can be applied, like the L2 (ridge) and L1 (lasso) penalties seen in Definition 5.23. These penalties are applied by including the penalty term in the loss function of the model, exactly the same as in Section 5.5.

- Furthermore, there are some other regularization techniques specific to neural networks.

Definition 6.19 (Dropout)**.** *Dropout is a regularization technique: for each sample at each training iteration, we set the output y of each node to zero with probability p. After training has completed we do not drop nodes anymore as we want to use the full capacity of the network. However, we multiply each activation where dropout was applied with $1 - p$. This is done to keep the activation on the same level as it was in expectation during training.*

Remarks:

- Effectively, dropout trains a different model at each iteration, where all models share the non-zeroed parameters. For large networks there is no risk that dropout breaks the information flow between the input and the output of the network.

- Dropout reduces the inter-dependencies between nodes in the network, which helps the model to learn more robust features, and also reduces overfitting.

- Dropout is computationally cheap and can be applied to any model that uses distributed representations and that is trained with gradient descent, i.e., any neural network.

- Dropout reduces overfitting but does not completely eliminate the problem. Luckily, dropout can be easily combined with other regularization strategies, for example, with early stopping.

Definition 6.20 (Early Stopping)**.** *Early stopping is a regularization strategy that returns to the parameter setting that produces the lowest validation error. In early stopping, training terminates when the best recorded validation error does not improve for a predefined number of epochs; this number is called patience.*

Remarks:

- Early stopping can be understood as an efficient algorithm for selecting the number of training steps, which is a hyperparameter.
- The cost of early stopping in terms of computation is that the validation needs to be run periodically after each epoch and that at least one copy of the parameters needs to be stored in memory. These costs are however small and generally do not cause any limitation.
- Early stopping does not affect the learning dynamics, can be used in conjunction with other regularization strategies, and is easy to implement.

6.6 Advanced Layers

While neural networks can theoretically learn any function, large networks (with too many weights) often struggle to converge to good solutions. Often we can use knowledge about the underlying problem to reduce the number of weights substantially. Some early successes of neural networks were achieved in image processing, where methods from classical computer vision were adapted to neural networks in the form of convolutions.

Definition 6.21 (Convolutional Neural Network or CNN)**.** *A convolutional neural network is a neural network layer that works on structured input data such as the pixels of an image. A CNN applies the same function (the same weights) to all neighborhoods of the input layer.*

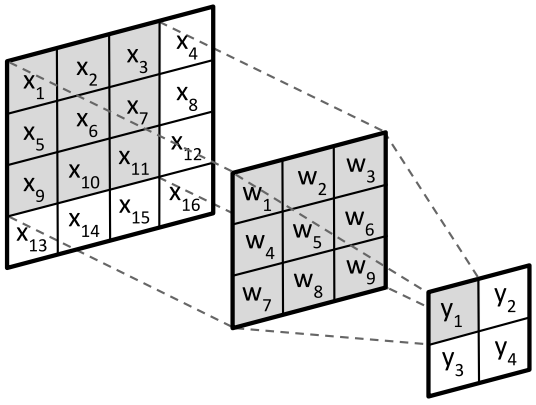

Figure 6.22: Convolution operation for a 4×4 input image $\boldsymbol{X}$, a 3×3 filter $\boldsymbol{W}$, and a 2×2 output $\boldsymbol{Y}$. The filter $\boldsymbol{W}$ slides over the image and for each position of the filter, a value y_i is calculated as the dot-product of the filter and the sub-matrix of $\boldsymbol{X}$ that it covers. E.g., $y_1 = w_1 x_1 + w_2 x_2 + \cdots + w_9 x_{11}$.

Example 6.23. *We want to detect vertical edges in images. Vertical edges can be found calculating the convolution (with symbol $\circledast$) of the image and a vertical*

Sobel filter, given by the matrix $\boldsymbol{W}$:

$$\boldsymbol{W} = \begin{pmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{pmatrix}$$

As an example, consider the 3×5 *greyscale image* $\boldsymbol{X}$ *where each pixel takes a value from* 0 *to* 255 *represented by the following matrix:*

$$\boldsymbol{X} = \begin{pmatrix} 80 & 92 & 163 & 234 & 230 \\ 85 & 98 & 237 & 233 & 232 \\ 83 & 96 & 236 & 235 & 231 \end{pmatrix}$$

To compute the convolution operation, the filter $\boldsymbol{W}$ *slides over the image* $\boldsymbol{X}$*. Generally, we require that the output* $\boldsymbol{Y} = \boldsymbol{X} \circledast \boldsymbol{W}$ *is of the same size as the input* $\boldsymbol{X}$*, for this, we need to pad the matrix* $\boldsymbol{X}$*. To prevent the padding from creating artificial edges, we extend the image by copying the border pixels* $\boldsymbol{X'}$*:*

$$\boldsymbol{X'} = \begin{pmatrix} 80 & 80 & 92 & 163 & 234 & 230 & 230 \\ 80 & 80 & 92 & 163 & 234 & 230 & 230 \\ 85 & 85 & 98 & 237 & 233 & 232 & 232 \\ 83 & 83 & 96 & 236 & 235 & 231 & 231 \\ 83 & 83 & 96 & 236 & 235 & 231 & 231 \end{pmatrix}$$

In a convolution, the filter $\boldsymbol{W}$ *slides over the image* $\boldsymbol{X}$*. For each position of the filter on the padded image (see Figure 6.22), one element of* $\boldsymbol{Y}$ *is calculated as the sum of the element-wise multiplication of the overlap. As the filter slides horizontally and vertically, this operation is repeated to obtain the values of each element of* $\boldsymbol{Y}$*. The resulting matrix* $\boldsymbol{Y}$ *is:*

$$\boldsymbol{Y} = \begin{pmatrix} -49 & -401 & -561 & -196 & 13 \\ -51 & -540 & -551 & -52 & 10 \\ -52 & -611 & -552 & 20 & 13 \end{pmatrix}$$

The large values in the second and third column of matrix $\boldsymbol{Y}$ *indicate that there is a strong gradient (variation) between those columns in the image, i.e., the image has a vertical dark/light edge. This edge is less pronounced in the first row.*

Remarks:

- In this example the filter $\boldsymbol{W}$ was given. A CNN does not know these weights, but only the information that $\boldsymbol{W}$ is a 3×3 convolution filter between two layers. With appropriate training data, the CNN will learn the weights of $\boldsymbol{W}$ by using backpropagation.

- Sometimes these learned patterns correspond to those that we as humans consider meaningful, e.g., edges. However, usually the patterns extracted by CNNs are not understandable. The power of CNNs reside in learning complex patterns that humans would not be able to design.

- Discrete convolutions can be performed beyond two dimensions. For example, an audio signal can be represented by the signal intensity at discrete time steps. In that case, a 1-dimensional convolution can be applied over the time dimension to filter the signal.
- In general, to apply convolutions to a given input, the input has to be structured as a tensor.

Definition 6.24 (Tensor)**.** *A tensor of order d is a d-dimensional array. A tensor generalizes vectors (1-dimensional) and matrices (2-dimensional). The shape of a tensor is a list of d integers defining the size of each dimension of the tensor.*

Remarks:

- An image, represented by its RGB (red, green, blue) pixel values can be naturally represented as a tensor of order 3 and shape [3, height, width]. An index $(0, x, y)$ into this tensor yields the intensity of red in the pixel at location (x, y). → notebook
- Note that the memory requirements of higher order tensors can be high. E.g., an order 5 tensor with 100 values in each dimension stores 100^5 values, which, given a floating point precision of 32 bits, requires 40 GB of memory.
- We have seen that CNNs exploit translation invariance in the structure of the data. Are there other such structural biases we can exploit? For example, what if we want a neural network to remember important features over several time steps of a sequential input?

Definition 6.25 (Recurrent Neural Network or RNN)**.** *In contrast to feed-forward neural networks, a recurrent neural network operates with time steps t. Each time step t gets an input $\boldsymbol{x}_t$ and a state $\boldsymbol{s}_t$. It outputs an updated state $\boldsymbol{s}_{t+1}$ and an output $\boldsymbol{y}_t$. More formally, we define the mappings*

$$\boldsymbol{y}_t = \hat{g}(\boldsymbol{x}_t, \boldsymbol{s}_t)$$

$$\boldsymbol{s}_{t+1} = \hat{h}(\boldsymbol{x}_t, \boldsymbol{s}_t)$$

where $\{\boldsymbol{x}_t\}_{t=0}^{\tau}$ and $\{\boldsymbol{y}_t\}_{t=0}^{\tau}$ are the input and corresponding output sequence of length τ and $\hat{g}(\cdot)$ and $\hat{h}(\cdot)$ are differentiable functions with learnable parameters. The initial state $\boldsymbol{s}_0$ can be a vector of learnable parameters, or simply initialized to $\mathbf{0}$.

Remarks:

- There are several ways of how to define $\hat{g}(\cdot)$ and $\hat{h}(\cdot)$, from simple linear projections to complex combinations of operations to combine the given inputs.

Example 6.26. *Consider that we want to solve the multi-path problem of wireless transmission, i.e., given a signal we wish to filter delayed copies from the signal. To do this online, i.e., while the signal is received, we have to remember*

the current input signal to filter a similar pattern later. We therefore seek to train an RNN to remember a given input for a few time steps and then reproduce it for filtering purposes. E.g., given the input signal $[5, 10, 0, 1.5, 3.5, 1]$ *we want the RNN to output* $[5, 10, 0, 0, 0, 0]$*. This can be achieved if we initialize the state* $\boldsymbol{s}_0$ *to* $\mathbf{0}$ *and parametrize* $\boldsymbol{s}_{t+1} = \hat{h}(x_t, \boldsymbol{s}_t) = \boldsymbol{W}\boldsymbol{s}_t + \boldsymbol{w}_h \cdot x_t$ *with* $\boldsymbol{W}$ *and* $\boldsymbol{w}_h$ *as*

$$\boldsymbol{W} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ -0.1 & -0.3 & 0 & 0 \end{pmatrix} \qquad \boldsymbol{w}_h = [0, 0, 0, 1]^T$$

where $\boldsymbol{W}$ *shifts the state and filters new inputs and* $\boldsymbol{w}_h$ *reads in the new symbol. The readout is given by* $y_t = \hat{g}(x_t, \boldsymbol{s}_t) = \boldsymbol{w}_g^T \boldsymbol{s}_t + x_t$ *with*

$$\boldsymbol{w}_g = [-0.1, -0.3, 0, 0]^T$$

Note that this parametrization simply reads the input symbol into the state $\boldsymbol{s}_t$*, propagates the symbol for some time steps and then subtracts it from a later input.*

Remarks:

- Instead of these given weights, neural networks will learn $\hat{g}$ and $\hat{h}$ when trained on real world signals. This is particularly useful if the input is a vector of multiple correlated noisy signals and a simple remember-and-reproduce solution is sub-optimal.

- Earlier we discussed that neural networks are directed acyclic graphs (DAGs). But RNNs are cyclic, as the state from step t gets fed back to the network in step $t + 1$. How can we train such a network?

- The solution is to copy the network τ times, i.e., unroll the cycle. This yields one long DAG where the state $\boldsymbol{s}_t$, calculated as intermediate output of one copy, is fed into the next copy. The calculated gradients for each copy are then summed to update the parameters. This is called *backpropagation through time (BPTT)*. Note that this can lead to vanishing/exploding gradients as we are essentially trying to train a network of depth τ.

- RNNs that are commonly used today are *Gated Recurrent Units (GRUs)* and *Long Short Term Memories (LSTMs)*. These address the issue of vanishing/exploding gradients in their definition of $\hat{g}$ and $\hat{h}$. The resulting architectures implement ideas similar to that of skip connections in feed-forward neural networks, albeit historically GRUs and LSTMs came long before people started talking about skip connections in MLPs and CNNs.

- As apparent from the equations in the definition above, RNNs are inherently sequential. They can process one input only after the previous input has been processed. This is slower than approaches that can process the whole sequence in parallel (such as CNNs).

- Both, CNNs and RNNs take advantage of *weight sharing*. In CNNs, the same weights (filters) are applied to all locations of the image. In RNNs, the same functions $\hat{g}(\cdot)$ and $\hat{h}(\cdot)$ (with the same learnable weights) are applied to all time steps t.

- What if we do not want to apply the same function everywhere? More specifically, what if only a selection of the input is of interest? Can we design an architecture that favors solutions which select features from the input instead of using the whole input? Can we index the input in a differentiable way?

Definition 6.27 (Attention)**.** *Attention is a method to aggregate inputs in a selective manner. Given n input vectors $\{\boldsymbol{x}_i\}_{i=0}^{n-1} \in \mathbb{R}^d$, each input vector is projected into a key vector $\boldsymbol{k}_i \in \mathbb{R}^{d_k}$ and a value vector $\boldsymbol{v}_i \in \mathbb{R}^d$. The projection is done by two learned matrices $\boldsymbol{W}_k \in \mathbb{R}^{d \times d_k}$ and $\boldsymbol{W}_v \in \mathbb{R}^{d \times d}$. Additionally, attention learns a query vector $\boldsymbol{q} \in \mathbb{R}^{d_k}$. For each input vector an attention score s_i is calculated as the dot-product between the query $\boldsymbol{q}$ and the key vector $\boldsymbol{k}_i$:*

$$s_i = \boldsymbol{q} \cdot \boldsymbol{W}_k \boldsymbol{x}_i = \boldsymbol{q} \cdot \boldsymbol{k}_i$$

The attention scores are normalized by a softmax (Definition 5.34) operation and each normalized score is multiplied by its corresponding value vector $\boldsymbol{v}_i = \boldsymbol{W}_v \boldsymbol{x}_i$. The results are added to produce the attention output $\boldsymbol{y} \in \mathbb{R}^d$:

$$\boldsymbol{y} = \frac{\sum_{i=0}^{n-1} \exp(s_i) \cdot \boldsymbol{v}_i}{\sum_{j=0}^{n-1} \exp(s_j)}$$

Remarks:

- The attention mechanism described here is called dot-product attention. This is the most common type of attention, but other variants exist as well.

Example 6.28. *Consider that we want to find in what position a sequence of $n = 5$ integers contains the value "3". Our input is $\boldsymbol{x} = [7, -5, -2, 3, 4]^T$. We use a ReLU for the keys: $\boldsymbol{k}_i = ReLU\,[x_i - 3, 3 - x_i]^T$. And we use a one-hot encoding for the values $\boldsymbol{v}_i$. For example, $x_1 = 7$ gets key and value*

$$\boldsymbol{k}_1 = [4, 0]^T, \boldsymbol{v}_1 = [1, 0, 0, 0, 0]^T.$$

Using $\boldsymbol{q} = [-1, -1]^T$ we get the scores $\boldsymbol{s} = [-4, -8, -5, 0, -1]^T$. Plugging the scores and values into the softmax gives

$$\boldsymbol{y} = [0.013, 0.000, 0.004, 0.718, 0.264]^T.$$

Because of the softmax, the output is not quite as clean as one might hope, since "$4 \approx 3$".

Remarks:

- Thanks to weight sharing, attention can processes input vectors of any length.
- A neural network can consist of multiple attention aggregations to select multiple (potentially different) inputs.
- If the scores are calculated based on the inputs, i.e., $s_i = f_i(\{\boldsymbol{x}_i\}_{i=0}^{k-1})$ for some functions f_i, attention is also referred to as *self-attention*, as the input "attends" to itself.
- Attention architectures yield the state-of-the-art performance in natural language processing tasks, as most of the time some words are more important than others to understand a sentence.
- All architectures presented in this section, i.e., CNNs/RNNs and Attention, take some domain knowledge to tailor the neural network to a specific purpose. This is also referred to as an *inductive bias*.

6.7 Architectures

In the previous section we introduced several building blocks that give different inductive biases. Let us now see how different losses and architectures can be combined to solve advanced computational challenges.

Definition 6.29 (Autoencoder)**.** *An autoencoder is a neural architecture formed by two neural networks: an encoder $f_{enc}(\cdot)$, which encodes the input $\boldsymbol{x} \in \mathbb{R}^n$ into a representation $\boldsymbol{z} \in \mathbb{R}^m$ called latent code; and a decoder $f_{dec}(\cdot)$, which decodes the latent code back into an approximation of the original input, i.e., $f_{dec}(f_{enc}(\boldsymbol{x})) \approx \boldsymbol{x}$. The representation is often designed to compress information by setting $m \ll n$. Autoencoders minimize a loss term named "reconstruction loss", that represents the difference between the output and the input.*

Remarks:

- The encoder and the decoder can be any type of neural network, e.g., MLPs, CNNs, RNNs or a combination of them.
- An example of reconstruction loss is the L^2 error:

$$L = \frac{1}{|D|} \sum_{\boldsymbol{x} \in D} ||\boldsymbol{x} - f_{dec}(f_{enc}(\boldsymbol{x}))||_2^2$$

- A schematic depiction of an autoencoder is shown in Figure 6.30.

Remarks:

- The requirement $m \ll n$ is not a necessity. We can also design autoencoders to have a specific structure in the latent code, or a latent code tailored to a given purpose through an additional loss.

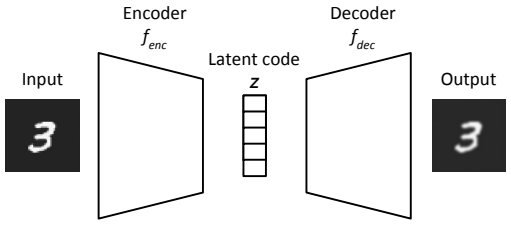

Figure 6.30: Schematic view of an autoencoder.

- The main advantage of autoencoders is that they learn the latent representation (or code) in an unsupervised manner, i.e., without labels. This makes them a versatile architecture that can be used in a wide range of problems, such as dimensionality reduction, compression, data denoising or unsupervised feature extraction.
- There exist many different types of autoencoders, with different losses, architectural elements or with additional inductive biases.
- Besides compression and encoding, neural networks can also be used for data generation.

Definition 6.31 (Generative Adversarial Network or GAN)**.** *GANs are a class of deep generative models in which two neural networks are trained simultaneously, while competing in a two-player minimax game. The generator's* $f_{gen}(\boldsymbol{r})$ *tries to produce realistic synthetic samples from a random input* $\boldsymbol{r}$*. The binary classifier called discriminator* $f_{dis}(\boldsymbol{x})$ *estimates whether a sample* $\boldsymbol{x}$ *is real or synthetic. The goal of the generator is to maximize the probability that the discriminator makes a mistake on* $f_{dis}(f_{gen}(\boldsymbol{r}))$*.*

Remarks:

- As in the case of autoencoders, the discriminator and generator can be any type of neural network.
- During training, the discriminator improves its ability to recognize synthetic samples while the generator learns to produce increasingly realistic samples to deceive the discriminator. In this adversarial setting, the equilibrium is reached when the generator produces realistic samples such that the discriminator cannot distinguish whether they are real or synthetic.
- The architecture of a vanilla GAN is shown in Figure 6.34.

Remarks:

- GANs achieved remarkable results in image generation. In particular, they can generate realistic-looking pictures and videos, which has raised concerns about malicious uses of these models to generate deepfakes.

- A GAN is a fully automated Turing Test with generator = testee, and discriminator = tester.

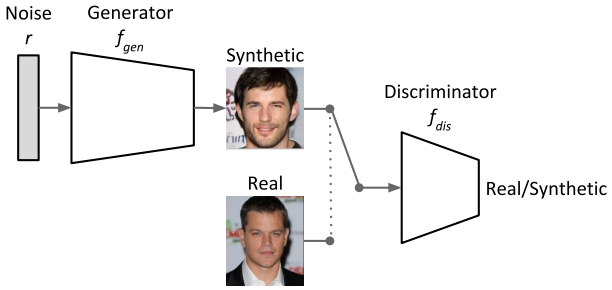

Figure 6.32: GAN architecture for generation of synthetic images of celebrities. For each sample, the discriminator needs to decide whether its input corresponds to a real or to a synthetic celebrity.

Example 6.33. *Faceswap-GAN is a popular implementation of a trained model for deepfake generation. At high level, this model uses an autoencoder as generator. Given an image of a human face, it produces a segmentation mask as well as the reconstructed input image. A segmentation mask is a representation of an image that delineates the most important objects in the image; in the case of human faces, these are the eyes, nose, ears, etc. Roughly speaking, an arbitrary image of a face A can be combined with the segmentation mask of another face B in order to generate an image that replaces the features of image B with those of A, i.e., a deepfake. This is what the model does at inference time.*

During training, Faceswap-GAN used a discriminator that determines whether an input image is a real face or a deepfake, as well as some other advanced methods such as a perceptual loss that improve image quality.

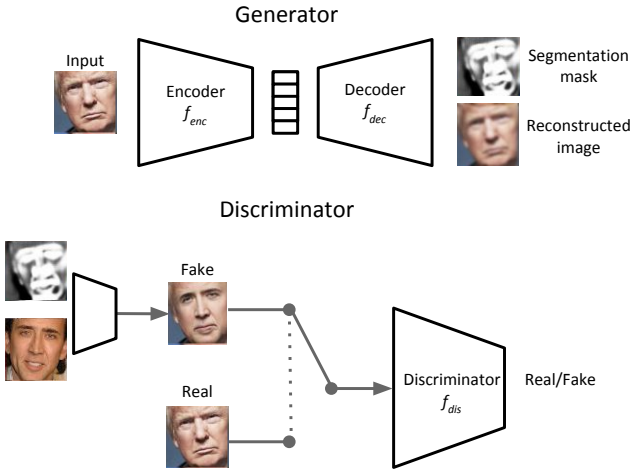

Figure 6.34: Architecture of Faceswap-GAN.

Remarks:

- Although originally conceived for generative tasks such as denoising, reconstruction or data generation, GANs have proved useful in other domains such as supervised learning, semi-supervised learning or reinforcement learning.
- Deep learning is applied in many different areas and consequently, there is a wide range of architectures. We list some promising architectures in Table 6.35:

Name	Description	Purpose
Transformer	A family of sequence-to-sequence models based on the self-attention operation.	State of the art in natural language processing (NLP) and gaining relevance in other areas. Recently received a lot of media attention with models like BERT and GPT-3.
Variational Auto-Encoder (VAE)	A generative model that tries to match the data distribution by enforcing a simplified posterior distribution in the latent space of an auto-encoder.	Data generation and posterior approximation in inherently stochastic models.
Siamese Network/ Contrastive Learning	The same network applied to different inputs, trained to recover a notion of similarity in the output representation.	Unsupervised representation learning (e.g., recovering an approximately metric space), authentication, hashing and matching
Graph Neural Network	Replicating a network on all nodes of a graph and incorporating operations for message exchanges with neighbors on the graph.	Graph/node/edge classification, community detection in social network and predicting protein/molecular interactions
Implicit Network	Training a neural network to recover a value from an index.	Data compression, super-resolution, image in-painting
Hyper Network	A neural network that outputs the weights of another neural network.	Mode abstraction and meta learning

Table 6.35: A glossary of promising architectures.

6.8 Reinforcement Learning

A neural network and its corresponding loss have to be end-to-end differentiable in order to apply gradient descent. So what if a problem is not differentiable? What if we want to find an optimum in a sequential setting, like an optimal sequence of decisions to reach a desired goal in an environment?

Definition 6.36 (Markov Decision Process or MDP)**.** *A Markov decision process formally defines an environment. An MDP is a 5-tuple* (S, A, T, R, s_0)*,*

where S is a set of states, A is a set of possible actions, $T : S \times A \to S$ is a state transition function, which describes the next state based on current state and action. However, T could also be probabilistic, i.e., $T : S \times A \to S \times [0, 1]$. $s_0 \in S$ (or $s_0 : S \to [0, 1]$) is an initial state (distribution). Finally, R is a reward function. Rewards can be given when reaching certain states ($R : S \to \mathbb{R}$), or when taking the right action in a state, $R : S \times A \to \mathbb{R}$.

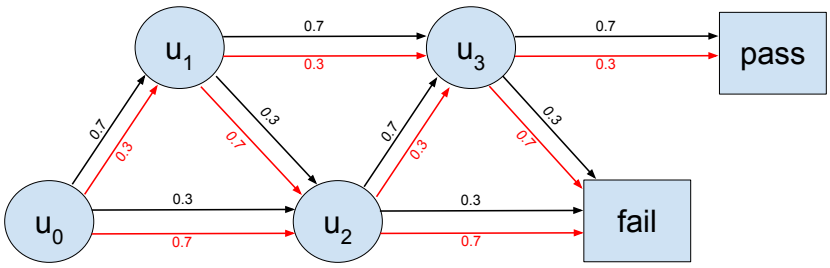

Figure 6.37: An example Markov decision process to figure out whether and when one should study or party in front of an exam.

Problem 6.38. *Figure 6.37 shows a simple example of an MDP with 6 states $S = \{u_0, u_1, u_2, u_3, fail, pass\}$ and two possible actions $A = \{study, party\}$. The states fail and pass are terminal states, and represent whether the agent fails or passes the exam, respectively. The agent starts in state u_0, i.e., $s_0 = u_0$. The transition probabilities for choosing the 'party' action (red) or 'study' action (black) are shown in the figure. Every time the agent chooses to party, it receives a reward of +1. If the agent chooses to study, it receives a reward of -1 (studying is painful). At the terminal states, the agent gets a reward of +10 for passing the exam and -10 in the failing it. Intuitively, the states u_1 and u_3 represent states where the agent has learned something. These states are more likely to be reached when studying rather than partying, and from these states the agent is more likely to pass the exam. How should the agent act? We can calculate the solution backward from the terminal states by filling in a 2×4 matrix Q giving the quality of each action in each of the non-terminal states. In u_3 taking the action 'study' will yield an expected reward of $Q[u_3, study] = -1 + 0.7 \cdot 10 + 0.3 \cdot (-10) = 3$. Similarly, choosing 'party' in this state yields an expected reward of $Q[u_3, party] = +1 + 0.3 \cdot 10 + 0.7 \cdot (-10) = -3$. In all earlier states we can assume that we take the action with higher expected reward in later states and thereby calculate the remaining values recursively as given in Table 6.39.*

Definition 6.40 (Policy)**.** *A policy $\pi : S \times A \to [0, 1]$ describes how the agent acts in the environment, i.e., how likely it will take an action $a \in A$ in a given state $s \in S$.*

State	study	party
u_3	3	-3
u_2	-1.9	-5.1
u_1	0.53	0.57
u_0	-1.171	-0.159

Table 6.39: Expected reward Q for each action in each non-terminal state. It's best to party in states u_0 and u_1, and best to study in states u_2 and u_3. Note that filling in this table is dynamic programming (Definition 1.11).

Remarks:

- Given an MDP and a policy, the state distribution of the agent after τ steps, i.e., how likely it is that the agent is in a given state after τ actions can be calculated.
- The goal in reinforcement learning is to find a good policy π, that is, a policy that accumulates positive rewards.
- More formally, we want to find the optimal policy π^* that maximizes the expected cumulative γ-discounted reward:

$$\pi^* = \max_\pi \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t)\right]$$

  where $\gamma \in [0, 1]$ is a discount factor that weighs immediate returns relative to future returns, where s_t is the state at time step t, respectively. The expectation is taken over actions sampled from the policy and states sampled from the transition distribution $P(\cdot|s, a)$ given the state s and action a.
- To find such a policy, we need to know how valuable each state is to a given policy.

Definition 6.41 (Value Function)**.** *A value function $V_\pi : S \to \mathbb{R}$ is a policy specific function that given a state returns the expected cumulative discounted reward of the policy starting in state s_t.*

$$V_\pi(s_t) = \mathbb{E}\left[\sum_{\tau=t}^{\infty} \gamma^{\tau-t} R(s_\tau)\right]$$

Definition 6.42 (Action-Value Function or Q-Function)**.** *An action-value function or Q(uality)-function $Q_\pi : S \times A \to \mathbb{R}$ is a policy specific function that given a state s and an action a returns the expected cumulative discounted reward of taking action a in state s and following policy π thereafter. That is*

$$Q_\pi(s, a) = R(s) + \mathbb{E}\left[\gamma V_\pi(s')\right]$$

where the expectation is over states s' sampled according to $T(s'|a, s)$.

Remarks:

- The value function can also be defined in terms of the Q function as

$$V_\pi(s) = \mathbb{E}_{a\sim\pi}\left[Q_\pi(s,a)\right].$$

- Given the general definition of value and action-value function, we can get a better understanding of what we want to find: the optimal policy π^*
- Note that V_{π^*} gives us for each state $s \in S$ the maximal expected cumulative reward that can be achieved when starting in state s.
- Further, if we are given Q_{π^*} it is easy to derive the optimal policy by simply taking the action that maximizes Q_{π^*}, i.e.

$$\pi^*(a|s) = \begin{cases} 1 \text{ if } a = \operatorname{argmax}_{a'} Q^{\pi^*}(a', s) \\ 0 \text{ else} \end{cases}$$

  In other words, the optimal policy in an MDP is deterministic!
- Knowing that we can derive the optimal policy from a quantity which requires the optimal policy might be a bit "recursive", but we can nevertheless try:

```
# S = states
# A = actions
# T = transitions
# R = rewards
def value_iteration(S, A, R, T):
    V = zero vector of size len(S)
    Q = zero matrix of size len(S) × len(A)
    while Q not converged:
        for s in S:
            for a in A:
                Q[s][a] = R(s) + γ∑_{s'∈S} T(s'|a,s) * V[s']
            V[s] = max(Q[s])  # max over a in A
    return Q
```

Algorithm 6.43: Value iteration

Remarks:

- If the MDP has cycles and $\gamma \to 1$, then this algorithm may not converge.

Lemma 6.44. *If we are given an MDP that can be represented as a DAG, value iteration converges in one iteration if we process the states $s \in S$ in reversed DAG order.*

Proof. Reversed DAG order means we will start at the terminal states and propagate the cumulative rewards back to the initial states. By induction, we always propagate the maximal achievable value by the max operation in Line 12 of Algorithm 6.43. Therefore, after one iteration, all states have the optimal value V_{π^*} assigned. □

Remarks:

- However, in many real world applications, the state space is just hilariously large. The game Go for instance has $3^{19\times 19} \approx 10^{172}$ possible states, so it is infeasible to compute or even store the whole Q-table. We can however train a neural network to approximate how likely a given position is to lead to a win. In combination with a bit of look-ahead planning (recursion) a neural network was able to beat the world champion.

Chapter Notes

While the beginnings of artificial neural networks go back to the 1940s [3], deep learning only became widely adapted and efficient in recent years with the use of GPUs to run computations in parallel. However, many of the theoretical investigations and architectures presented here have been known for quite some time by now. The universal approximation capability of neural networks was first shown for sigmoid non-linearities [1] and later generalized to other non-linearities [2]. Also the VC Dimension goes back to around 1970 [4]. We summarize further milestones achieved by neural networks in the table below.

Year	Name	Milestone
1989	MNIST	Handwritten digit classification
2005	DARPA	Self-driving car challenge: 212km in 7h
2012	AlexNet	Image classification breakthrough
2014	Deepface	Human level performance in face recognition
2014	DQN	Superhuman performance in many Atari games
2016	AlphaGo	Beats Champion in Go
2017	Waymo	Fully autonomous self-driving on public roads
2018	Obvious	Sells art generated by a GAN for $432,500
2020	OpenAI	GPT-3 model can create poetry, code, etc.
2020	AlphaFold	Achieves 90% in CASP protein folding

Table 6.45: Neural Network Milestones

This chapter was written in collaboration with Damian Pascual and Oliver Richter.

Bibliography

[1] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2(4):303–314, Dec 1989.

[2] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4(2):251 – 257, 1991.

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

[4] V. N. Vapnik and A. Ya. Chervonenkis. On the uniform convergence of relative frequencies of events to their probabilities. *Theory of Probability & Its Applications*, 16(2):264–280, 1971.

Chapter 7

Computability

Computability was pioneered by Alan Turing and Kurt Gödel. Turing probably committed suicide by eating an apple he poisoned with cyanide. Gödel on the other hand had an obsessive fear of being poisoned with food; when his wife was hospitalized, he refused to eat, and eventually starved to death. Now it's your turn to study this intoxicating subject.

7.1 Undecidability

In the previous chapters, we have analyzed various computational tasks. While some functions were easy to compute efficiently, others were difficult; in these cases, we have focused on approximations or heuristics. However, given enough time and resources, could we always find the solution to a problem?

Problem 7.1 (Halting problem)**.** *Given a program P and an input x to P, does $P(x)$ halt (stop running) after a finite amount of time?*

Remarks:

- Can we write a Python program that solves Problem 7.1? Somewhat surprisingly, the input of our program is also a program (plus an input parameter of this program).
- Naturally, we must somehow encode the input program. There are various ways to do this. We can, for example, consider the whole code of the program as a long string of text, encoding each character in this string with a byte.
- The halting problem is sometimes easy to solve, for example, in case of the simple programs in Algorithm 7.2. But is it always?

```
1 def P_1(x):
2     print("Hello, world!")
3     return
4
```