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 530 billion weights. No 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 \( x \in \mathbb{R}^d \), then applies an activation function \( \sigma: \mathbb{R} \rightarrow \mathbb{R} \):

\[
y = \sigma(w^T x)
\]

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

Remarks:
- In the literature, the intercept \( w_0 \) is sometimes referred to as “bias” \( b \) and kept separate from \( w \), i.e., \( y = \sigma(w^T x + b) \). 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 \( \sigma \) can take many different forms. Some nodes may simply use the identity as activation function, i.e., \( \sigma(x) = x \). Most nodes apply non-linear activation functions in order to allow the model to approximate non-linear functions, e.g. the sigmoid function \( \psi(w^T x) = \frac{1}{1+e^{-w^T x}} \) 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 \( x \) of the network is stored by the \( n \) input nodes \( V_i \) (with no incoming edges). The output \( y \) 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 \( V_i \cup V_h \cup V_o = V \). 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. 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 \( x \) of each node \( v \) is the vector of computed outputs \( y \) of its DAG predecessor nodes. Then, \( v \) computes its own output as \( y_v = \sigma(w_v^T x_v) \). Hence the nodes must be processed in DAG order.

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

Algorithm 6.3: Feed-forward computation in DAG.

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

Remarks:
- There exist cyclic neural networks as well, see Definition 6.22.
- 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 \( V_1 \), the last layer the output nodes \( V_n \). Each hidden node is in a layer \( l \), and all nodes of layer \( l \) have the same input, namely the outputs \( y_j \) of all nodes \( v_i \) 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 \( x = W \cdot z \), where the matrix \( W \) is composed of the weight rows \( \mathbf{w} \), \( z \) is the vector of pre-activation values in the nodes and \( x \) 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 Universal Approximation

#### Theorem 6.6 (Universal Approximation Theorem).

Given a continuous function \( f : \mathbb{R} \rightarrow \mathbb{R} \) (for simplicity, input \( x \geq 0 \)), there exists a neural network \( f \) with one hidden layer that approximates \( f \) arbitrarily well. That is \( |f(x) - f(x)| < \epsilon \) for all \( x \geq 0 \) and \( \epsilon > 0 \).

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

The single output node \( v_n \) computes function \( f(x) = \mathbf{w}^T \mathbf{y} \), where \( \mathbf{w} \) are \( v_n \)'s weights, and \( \mathbf{y} \) are the outputs produced by the hidden nodes. As usual, \( \mathbf{w} \) includes an additional value \( v_0 \), and \( \mathbf{y} \) starts with a additional constant \( 1 \). We choose \( v_0 = f(0) \), such that \( f(0) = f(0) \) even without any hidden nodes. Every hidden node \( v_i \) computes \( y_i = g_i(x) = \psi(x + b_i) \), where \( \kappa \rightarrow \infty \) is a large constant. While the sigmoid function \( \psi(x) = \frac{1}{1 + e^{-x}} \) from Definition 5.30 is a smooth step function, \( \kappa \rightarrow \infty \) will make that step sharp.

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

### 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 6.8 (Feature Extractor function).** A feature extractor \( \phi \) is a function that transforms raw input data \( x \) into features. These features represent the initial data in a way that simplifies approximating any function, but rather to learn how to approximate any function. It can be shown via various reductions (Definition 2.8) that learning is NP-hard (Definition 2.20).

#### 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, and multidimensional version, etc.
- The theoretical construction given in the proof is not used in practice, as it would lead to numerical instabilities (\( x \rightarrow \infty \)).
- However, the premise of a neural network is not only to approximate any function, but rather to learn how to approximate any function.

![Figure 6.7: Effect of weights on the sigmoid function.](image-url)
Definition 6.9 (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 $x$. By the chain rule, we can calculate the error of the output nodes with respect to $z$ as

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

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

$$\frac{\partial L}{\partial \mathbf{w}} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial z} \cdot \mathbf{z}, \quad \text{and} \quad \frac{\partial L}{\partial \mathbf{x}} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial z} \cdot \mathbf{z}.$$ 

The gradients with respect to the weights $\mathbf{w}$ can then be used for updating the weights, while the gradient with respect to the inputs can be aggregated to pass the gradient $\frac{\partial L}{\partial \mathbf{x}}$ to the preceding nodes in the network. Concretely, each node in the network adjusts its own weights $\mathbf{w}$ based on the error signal $\frac{\partial L}{\partial \mathbf{w}}$ and tells its input nodes $\mathbf{x}$, to adjust by backpropagating $\frac{\partial L}{\partial \mathbf{x}}$. The backpropagation algorithm is given in Algorithm 6.10.

Algorithm 6.10: Backpropagation Algorithm

Remarks:

- Many libraries backpropagate gradients with a simple function call.
- Memory may be a problem, since we need to memorize $x$ 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 the parameter update 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 6.11 (Vanishing Gradients). Gradient descent updates can stagnate due to vanishing gradients, i.e., gradients that are close to 0. This can occur during backpropagation for different reasons:

- The activation function $\sigma$ saturates, i.e. the gradient $\frac{\partial \sigma}{\partial x} \approx 0$. In this case, the gradient $\frac{\partial L}{\partial \mathbf{w}}$ of all weights $\mathbf{w}$, and the backpropagated gradients $\frac{\partial L}{\partial \mathbf{x}}$ of the node will also be close to zero. The node stops learning.
- The summation in Line 12 of Algorithm 6.10 can accidentally become 0 (when terms cancel each other).

Definition 6.12 (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 gradients.

Remarks:

- Some large weight $|w| > 1$ boosts the backpropagated error.
- The summation in Line 12 of Algorithm 6.10 can accidentally become large because of many positive (or negative) terms.

6.3. TRAINING NEURAL NETWORKS

Definition 6.13 (ReLU). The Rectified Linear Unit activation is defined as:

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

The activation function $\sigma$ saturates, i.e. the gradient $\frac{\partial \sigma}{\partial x} \approx 0$. In this case, the gradient $\frac{\partial L}{\partial \mathbf{w}}$ of all weights $\mathbf{w}$, and the backpropagated gradients $\frac{\partial L}{\partial \mathbf{x}}$ of the node will also be close to zero. The node stops learning.

The summation in Line 12 of Algorithm 6.10 can accidentally become 0 (when terms cancel each other).

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 gradient 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).
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, SiLU, Maceut, etc.
- Another design choice that impacts the performance of the model is the initialization scheme.

Definition 6.14 (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.

### 6.5 Regularization

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 an art than a science there are automatic hyperparameter optimization algorithms that can help in this process.

Definition 6.15 (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.
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 6.16 (Dropout)**. Dropout is a regularization technique: for each sample at each training iteration, we set the output $y_i$ 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 easily be combined with other regularization strategies, for example, with early stopping.

**Definition 6.17 (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. 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 6.18 (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.

![Convolution operation](Image)

Figure 6.19: Convolution operation for a $4 \times 4$ input image $X$, a $3 \times 3$ filter $W$, and a $2 \times 2$ output $Y$. The filter $W$ slides over the image and for each position of the filter, a value $y_{ij}$ is calculated as the dot-product of the filter and the sub-matrix of $X$ that it covers, e.g., $y_{ij} = w_1x_{i1} + w_2x_{i2} + \cdots + w_9x_{i9}$.

**Example 6.20**. We want to detect vertical edges in images. Vertical edges can be found calculating the convolution (with symbol ⊛) of the image and a vertical...
A Sobel filter, given by the matrix $W$:
$$
W = \begin{bmatrix}
1 & 0 & -1 \\
2 & 0 & -2 \\
1 & 0 & -1
\end{bmatrix}
$$

As an example, consider the $3 \times 5$ grayscale image $X$ where each pixel takes a value from 0 to 255 represented by the following matrix:
$$
X = \begin{bmatrix}
80 & 92 & 163 & 234 & 230 \\
85 & 98 & 237 & 253 & 232 \\
83 & 96 & 236 & 235 & 231
\end{bmatrix}
$$

To compute the convolution operation, the filter $W$ slides over the image $X$. Generally, we require that the output $Y = X \circledast W$ is of the same size as the input $X$; for this, we need to pad the matrix $X$. To prevent the padding from creating artificial edges, we extend the image by copying the border pixels $X'$:
$$
X' = \begin{bmatrix}
80 & 80 & 92 & 163 & 234 & 230 & 230 \\
80 & 80 & 80 & 92 & 163 & 234 & 230 \\
83 & 83 & 83 & 96 & 236 & 235 & 231 \\
83 & 83 & 83 & 96 & 236 & 235 & 231
\end{bmatrix}
$$

In a convolution, the filter $W$ slides over the image $X$. For each position of the filter on the padded image (see Figure 6.18), one element of $Y$ 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 $Y$. The resulting matrix $Y$ is:
$$
Y = \begin{bmatrix}
-49 & -401 & -561 & -196 & 13 \\
-51 & -540 & -551 & -52 & 10
\end{bmatrix}
$$

The large values in the second and third columns of matrix $Y$ 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 $W$ was given. A CNN does not know these weights, but only the information that $W$ is a $3 \times 3$ convolution filter between two layers. With appropriate training data, the CNN will learn the weights of $W$ by using backpropagation.
- Bias ($u_t$) and non-linearity (e.g., ReLU activation) are omitted in our examples for improved clarity.
- Compared to a fully connected network with the same input size, a CNN has a significantly lower number of weights.

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 an image, represented by its RGB (red, green, blue) pixel values can be extracted by CNNs are not understandable. The power of CNNs resides 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 6.21 (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 $(x, y)$.
- 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 roughly 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 6.22 (Recurrent Neural Network or RNN). In contrast to feed-forward neural networks, a recurrent neural network operates with time steps. Each time step $t$ gets an input $x_t$ and a state $s_t$. It outputs an updated state $s_{t+1}$ and an output $y_t$. More formally, we define the mappings:
$$
y_t = \hat{y}(x_t, s_t)
$$
$$
s_{t+1} = \hat{s}(x_t, s_t)
$$
where $(x_t)_{t=0}^T$ and $(y_t)_{t=0}^T$ are the input and corresponding output sequence of length $T$ and $\hat{y}(\cdot)$ and $\hat{s}(\cdot)$ are differentiable functions with learnable parameters.

The initial state $s_0$ can be a vector of learnable parameters, or simply initialized to 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 6.23. 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 wish 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 \([5, 10, 0, 1, 3, 5, 1]\) we want the RNN to output \([5, 10, 0, 0, 0, 0, 0]\). This can be achieved if we initialize the state \(s_0\) to \(0\) and parameterize \(s_{t+1} = \hat{h}(s_t, x_t) = Ws_t + u_s, x_t\) with \(W\) and \(u_s\) as

\[
W = \begin{pmatrix}
  0 & 1 & 0 & 0 \\
  0 & 0 & 1 & 0 \\
  0 & 0 & 0 & 1 \\
-0.1 & -0.3 & 0 & 0
\end{pmatrix},
\]

where \(W\) shifts the state and filters new inputs and \(u_s\) reads in the new symbol. The readout is given by

\[
y_t = \hat{g}(s_t, x_t) = Ws_t + u_y x_t + x_t
\]

Note that this parameterization simply reads the input symbol into the state \(s_t\), 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 \( \hat{g}(W, u_y) \) and \( \hat{h}(u_s) \) 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 0 gets fed back to the network in step 1. How can we train such a network?
- The solution is to copy the network \( h \) times, i.e., unroll the cycle (see Figure 6.24). This yields one long DAG where the state \(s_t\) 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 \( h \).
- 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 definitions 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.

Definition 6.25 (Attention). **Attention** is a method to aggregate inputs in a **differential way**. 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?

\[
\text{Figure 6.24: Unrolling the RNN through time. At each time step, } \hat{g} \text{ and } \hat{h} \text{ compute output } y_t \text{ and state } x_{t+1} \text{ respectively, given input } x_t \text{ and the previous state } s_t. \text{ BPTT propagates the gradients through the unrolled network.}
\]

6.6. ADVANCED LAYERS

- 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.
- 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?

\[
\text{Figure 6.24: Unrolling the RNN through time. At each time step, } \hat{g} \text{ and } \hat{h} \text{ compute output } y_t \text{ and state } x_{t+1} \text{ respectively, given input } x_t \text{ and the previous state } s_t. \text{ BPTT propagates the gradients through the unrolled network.}
\]

Definition 6.25 (Attention). **Attention** is a method to aggregate inputs in a **selective manner**. Given \( n \) input vectors \( \{x_i\}_{i=0}^{n-1} \in \mathbb{R}^d \), each input vector is projected into a key vector \( k_i \in \mathbb{R}^d \), and a value vector \( v_i \in \mathbb{R}^e \). The projection is done by two learned matrices \( W_k \in \mathbb{R}^{d \times d} \) and \( W_v \in \mathbb{R}^{e \times d} \). Additionally, attention is given a query vector \( q \in \mathbb{R}^d \). For each input vector an attention score \( s_i \) is calculated as the dot-product between the query \( q \) and the key vector \( k_i \):

\[
s_i = q \cdot W_k x_i = q \cdot k_i
\]

The attention scores are normalized by a softmax (Definition 5.34) operation and each normalized score is multiplied by its corresponding value vector \( v_i = W_v x_i \). The results are added to produce the attention output \( y \in \mathbb{R}^e \):

\[
y = \sum_{i=0}^{n-1} \frac{\exp(s_i)}{\sum_{j=0}^{n-1} \exp(s_j)} \cdot v_i
\]

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 6.26. Consider that we want to find in what position a sequence of n = 5 integers contains the value “3”. One input is \( \mathbf{x} = [7, -5, -2, 3, 4]^T \). We use a ReLU for the keys. \( \mathbf{k}_i = \text{ReLU}(\mathbf{x} - 3, \mathbf{a}) \). And we use a one-hot encoding for the values \( \mathbf{v}_i \). For example, \( \mathbf{x}_1 = 7 \) gets key and value \( \mathbf{k}_1 = [4, 0]^T, \mathbf{v}_1 = [1, 0, 0, 0, 0]^T \).

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

\[
\mathbf{y} = [0.013, 0.000, 0.004, 0.718, 0.268]^T.
\]

Because of the softmax, the output is not quite as clean as one might hope, since “4 ≈ 3”.

Remarks:

- Thanks to weight sharing, attention can process 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., \( s_i = f_i(x_1, \ldots, x_n) \) 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 6.27 (Autoencoder). An autoencoder is a neural architecture formed by two neural networks: an encoder \( f_{\text{enc}}(\cdot) \), which encodes the input \( x \in \mathbb{R}^n \) into a representation \( \mathbf{z} \in \mathbb{R}^m \) called latent code; and a decoder \( f_{\text{dec}}(\cdot) \), which decodes the latent code back into an approximation of the original input, i.e., \( f_{\text{dec}}(f_{\text{enc}}(x)) \approx x \). 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.

Definition 6.29 (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 \( f_{\text{gen}}(\cdot) \) tries to produce realistic synthetic samples from a random input \( r \). The binary classifier called discriminator \( f_{\text{dis}}(\cdot) \) estimates whether a sample \( x \) is real or synthetic. The goal of the generator is to maximize the probability that the discriminator makes a mistake on \( f_{\text{dis}}(f_{\text{gen}}(r)) \).
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 6.32.
- 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 6.30: 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 6.31. Faceswap-GAN is a popular implementation of a model trained 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 6.32: 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 6.33.
6.8. REINFORCEMENT LEARNING

Definition 6.34 (Markov Decision Process or MDP). A Markov decision process formally defines an environment as an MDP $<S, A, T, R, s_0>$, where $S$ is a set of states, $A$ is a set of possible actions, $T: S \times A \rightarrow S$ is a state transition function, which describes the next state based on current state and action, $R: S \times A \rightarrow \mathbb{R}$ is the reward function, $s_0 \in S$ (or $s_0: S \rightarrow [0,1]$) is an initial state (distribution). Finally, $R$ is a reward function. Rewards can be given when reaching certain states ($R: S \rightarrow \mathbb{R}$), or when taking the right action in a state, $R: S \times A \rightarrow \mathbb{R}$.

Problem 6.36. Figure 6.35 shows a simple example of an MDP with 6 states $S = \{u_0, u_1, u_2, u_3, \text{fail}, \text{pass}\}$ and two possible actions $A = \{\text{study}, \text{party}\}$. The

<table>
<thead>
<tr>
<th>State</th>
<th>study</th>
<th>party</th>
</tr>
</thead>
<tbody>
<tr>
<td>$u_0$</td>
<td>3</td>
<td>-3</td>
</tr>
<tr>
<td>$u_2$</td>
<td>-1.9</td>
<td>-5.1</td>
</tr>
<tr>
<td>$u_3$</td>
<td>0.53</td>
<td>0.57</td>
</tr>
<tr>
<td>$u_4$</td>
<td>-1.172</td>
<td>0.159</td>
</tr>
</tbody>
</table>

Table 6.37: Expected reward $Q$ for each action in each non-terminal state. It is best to party in states $u_0$ and $u_2$, and best to study in states $u_2$ and $u_3$. Note that filling in this table is dynamic programming (Definition 1.11).

Definition 6.38 (Policy). A policy $\pi: S \times A \rightarrow [0,1]$ describes how the agent acts in the environment, i.e., how likely it will take an action $a \in A$ in a given state $s \in S$. 

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., $u_0 = u_0$. 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 $-10$ for failing. 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 $Q(u_0, \text{study}) = -1 + 0.7 \cdot 10 + 0.3 \cdot (-10) = 0.3$. Similarly, choosing ‘party’ in this state yields an expected reward of $Q(u_0, \text{party}) = -1 + 0.3 \cdot 10 + 0.7 \cdot (-10) = -1.3$. 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 6.37.
Remarks:
• Given an MDP and a policy, the state distribution of the agent after \(\tau\) steps, i.e., how likely it is that the agent is in a given state after \(\tau\) actions can be calculated.
• The goal in reinforcement learning is to find a good policy \(\pi\), that is, a policy that accumulates positive rewards.
• More formally, we want to find the optimal policy \(\pi^*\) that maximizes the expected cumulative \(\gamma\)-discounted reward:

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

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 \(P(s_{t+1}, a_t|s_t)\) given the state \(s_t\) and action \(a_t\).
• To find such a policy, we need to know how valuable each state is to a given policy.

Definition 6.39 (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 \(s\):

\[
V_\pi(s) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t) \mid s_0 = s, \pi\right]
\]

Definition 6.40 (Action-Value Function or Q-Function). An action-value function or Q-function \(Q_\pi : S \times A \to \mathbb{R}\) is a policy specific function that given a state and action returns the expected cumulative discounted reward of taking action \(a\) in state \(s\) and following policy \(\pi\) thereafter. That is

\[
Q_\pi(s, a) = R(s) + \mathbb{E}[\sum_{t=1}^{\infty} \gamma^t R(s_t) \mid s_0 = s, a_0 = a, \pi]
\]

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

Remarks:
• The value function can also be defined in terms of the Q function as

\[
V_\pi(s) = \mathbb{E}_{\pi}[Q_\pi(s, a)]
\]

• 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 \(\pi^*\).
• Note that \(V_\pi\) gives us for each state \(s \in S\) the maximal expected cumulative reward that can be achieved when starting in state \(s\).

6.8. REINFORCEMENT LEARNING

• Further, if we are given \(Q^\pi\), it is easy to derive the optimal policy by simply taking the action that maximizes \(Q^\pi\), i.e.,

\[
\pi^*(a|s) = \begin{cases} 1 & \text{if } a = \arg\max_a Q^\pi(s, a) \\ 0 & \text{else} \end{cases}
\]

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:

Algorithm 6.41: Value iteration

Remarks:
• If the MDP has cycles and \(\gamma \to 1\), then this algorithm may not converge.

Lemma 6.42. 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 6.41. Therefore, after one iteration, all states have the optimal value \(V^{\pi^*}\) assigned.
Remarks:

- However, in many real world applications, the state space is just hilariously large. The game Go for instance has $3^{19	imes 19} \approx 10^{172}$ 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 [5], deep learning only became widely adopted 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 nonlinearities [1] and later generalized to other nonlinearities [3]. Even before it was shown that learning various functions is NP-complete [4, 6]. This is still an active research area, e.g. [2]. The VC-Dimension discussion is even older [7]. We summarize further milestones achieved by neural networks in the table below.

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

Table 6.43: Neural Network Milestones

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

Bibliography