Deep Learning: Architectures and Methods

Summary

Fabian Damken

October 18, 2022

TECHNISCHE UNIVERSITÄT DARMSTADT

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

This summary of the course “Deep Learning: Architectures and Methods” held at the TU Darmstadt covers a lot of topics in the wast field of deep learning and neural network architectures. The first part focuses on fully connected and convolutional networks and in the end some more advanced architectures are touched, namely recurrent, long short-term memory and transformer networks. Knowledge of basic machine learning taxonomy like “features”, “training data” and “supervised learning” is assumed to be known as well as basic mathematical knowledge.

The goal of deep architectures is learning a feature hierarchy where higher-level features are built on top of lower-level features. This is done by stacking multiple linear layers (perceptrons) on top of each other with nonlinearities between them to get a larger support. While it is, due to the universal function approximation theorem, not necessary to have multiple layers (one hidden layer is enough to approximate any sufficiently well-behaving function), it has been shown empirically that multiple layers improve the performance. In the last century the extreme computational power available leveraged the success of deep learning. This success was so extraordinary that deep learning became a hype that everyone jumped onto.

This first chapter covers some milestones and history as well as the reason for the extreme success of deep learning.

1.1 Milestones

The first successful application of deep learning was in 1989 with LeNet to recognize zip codes. Image detection and classification was develop a lot further since that and in 2012, AlexNet broke the human prediction error on ImageNet, a collection of (then) $200\,\mathrm{GB}$ labeled images. But deep learning is not only useful for image classification: In 2013, DeepMind beat the best human players on basic Arcade games and in 2016 AlphaGo defeated the world champion in the game Go. All of this was possible due to the massive power of deep neural networks and reinforcement learning.

1.2 The (Surprising?) Success of Deep Neural Networks

It is interesting that deep neural networks are so successful as they are built from extremely simple blocks. Also, the algorithms used for training these networks are extremely basic gradient descent descendants that only find local minima. In other words: they are greedy algorithms which are even more limited in what they can do than the network itself (if a problem has a greedy solution, it is usually regarded as an “easy” problem). But still they are extremely powerful!

This is partially due to that hierarchical representations as induced by neural networks, are ubiquitous in artificial intelligence and information representation in general (e.g., images are hierarchical as well as natural language). Also, it seems like most learning problem are actually (relatively) easy and they have a gradient descent path towards a good model.

2 Optimization

This chapter covers basic (numerical) optimization techniques that are used in machine and deep learning. Hence, it mainly focuses in gradient descent variants and less on, for example, sequential quadratic programming or trust region methods. A first method when thinking of optimization is a random search: instead of using sophisticated update rules for the parameters, they are sampled randomly. Obviously, this does not yield robust results and most often does not even yield any useful results. One advantage nevertheless is that random search does not get stuck in local minima and is, with infinite time, capable of finding the global optimum.

2.1 Gradient Descent

The basic idea of gradient descent (GD) is to follow the slope of the curve, i.e. the gradient[1]. The gradient always points in the direction of the steepest *ascent*, so its negative direction points towards the steepest descent. Intuitively, GD works like standing on a hill (the function that is being optimized) and always walking into the direction where the slope is the steepest. One might find the valley that way, but it is also possible to get stuck in a small hole. Formally the update equation for GD is given as

$$\boldsymbol{\theta}^{(k+1)} = \boldsymbol{\theta}^{(k)} - \alpha \cdot \boldsymbol{\nabla}_{\boldsymbol{\theta}} L\big(\boldsymbol{\theta}^{(k)}\big),$$

where $\boldsymbol{\theta}$ are the parameters and L is the function to be minimized.

2.1.1 Evaluating the Gradient

As GD needs the values of the gradient, it has to be evaluated. There are three ways of approaching this: numerical evaluation, analytical derivatives or automatic differentiation.

Numerical evaluation is extraordinarily simple, but is only approximate and slow to evaluate (when using forward difference quotients, the function has to be evaluated two times for each parameter).

Analytical derivatives is hard and time-consuming, but exact and fast. But it is also error-prone as both the original function and the derivative have to be both implemented and derived. To verify that an analytical gradient is correct one method used in practice is to also evaluate the gradient numerically and checking if the results are roughly equal.

As neural networks usually have lots of parameters and are fairly complex computational structures, neither of these methods are practical. Hence, automatic differentiation is used. In automatic differentiation, a computational graph is built and then the gradients are propagated backwards through it. This is called backpropagation and is covered in more detail in chapter 3.

[1] In this case the gradient refers to the vector of partial derivatives of a function w.r.t. all parameters.

2.1.2 Mini-Batch and Stochastic Gradient Descent

In mini-batch GD, only a small portion (common sizes are 32, 64, or 128 samples) of the training set is used for computing the gradient. This technique is used as it might not be possible to compute the whole gradient due to memory limitations. This makes the gradient noisy, but the progress is still good on average. It is also called *stochastic* GD as the gradient gets replaced by a Monte-Carlo estimate of the expectation of the gradient.

2.2 Newton's Method and L-BFGS

Instead of walking along the gradient, *Newton's method* leverages further knowledge from calculus and uses not only the gradient but the Hessian, too. The update equation is

$$\boldsymbol{\theta}^{(k+1)} = \boldsymbol{\theta}^{(k)} - \alpha \cdot \left(\boldsymbol{H}_L\big(\boldsymbol{\theta}^{(k)}\big)\right)^{-1} \boldsymbol{\nabla}_{\boldsymbol{\theta}} L\big(\boldsymbol{\theta}^{(k)}\big),$$

where $\boldsymbol{H}_L$ is the Hessian. Newton's method is capable of finding the minima of quadratic functions in one step and also converges fast for other problems. But due to its dependence on the Hessian, which is costly to compute, and due to the inverse of the said, it is rarely used in deep learning. The inversion has a time complexity of $\mathcal{O}(d^3)$, where d are the number of parameters. As neural networks usually have millions of parameters, inverting the matrix is not practical. Also, matrix inversion is numerically unstable and the Hessian might even be singular.

Arising from these problems, the L-BFGS algorithm was developed which directly approximates the inverse of the Hessian. Hence, neither calculating nor inverting it is necessary.

2.3 Convergence

A method is said to converge *quadratically* if the error $\epsilon_k = x_* - x$ depends quadratically on the previous error:

$$\epsilon_{k+1} = \mu \epsilon_k^2, \qquad \epsilon_k \in \mathcal{O}\big(\mu^{2^k}\big)$$

Analogously, a method converges *linearly* if the error depends linearly on the previous error:

$$\epsilon_{k+1} \leq \mu \epsilon_k, \qquad \epsilon_k \in \mathcal{O}(\mu^n)$$

For SGD, when the learning rate is changed to $1/n$, the convergence is $\epsilon_k \in \mathcal{O}(1/n)$. So SGD is terrible compared to other methods, but still it is used in practice as one million iterations or so are still enough for reaching single point precision in the error and one million iterations are okay to compute.

2.4 Momentum

When using SGD, the trajectory along the algorithm converges is very jittery on steep locations as the gradient is large and huge steps are made. On the other hand, only very slow progress is made at locations where the gradient is small. One approach for tackling this problem is *momentum*. The formal definition is

$$\begin{aligned}
\boldsymbol{v}^{(k+1)} &= \mu \boldsymbol{v}^{(k)} - \alpha \cdot \boldsymbol{\nabla}_{\boldsymbol{\theta}} L\big(\boldsymbol{\theta}^{(k)}\big) \\
\boldsymbol{\theta}^{(k+1)} &= \boldsymbol{\theta}^{(k)} + \boldsymbol{v}^{(k+1)}
\end{aligned}$$

with a "friction" coefficient μ which is usually 0.5, 0.9, or 0.99 or annealed over time (e.g., from 0.5 to 0.99). The intuition is that the gradient can build up velocity along shallow regions of the loss which gets damped in steep regions. This reduces the jittering at steep regions by reducing the size of the gradients and ramps up the speed on shallow regions.

The momentum is usually initialized with zeros.

2.4.1 Nesterov Momentum

A variant of the vanilla momentum is *Nesterov momentum*, where the update is compute "one step ahead" by following the old momentum:

$$\begin{aligned}\boldsymbol{v}^{(k+1)} &= \mu\boldsymbol{v}^{(k)} - \alpha \cdot \boldsymbol{\nabla}_{\boldsymbol{\theta}} L\big(\boldsymbol{\theta}^{(k)} + \mu\boldsymbol{v}^{(k)}\big)\\ \boldsymbol{\theta}^{(k+1)} &= \boldsymbol{\theta}^{(k)} + \boldsymbol{v}^{(k+1)}\end{aligned}$$

Notice the change in the top equation when calculating the gradient. But this calculation is slightly inconvenient as usually the forward pass is computed for the current parameter settings anyway, e.g., for computing the predication accuracy. This inconvenience can be circumvented by a slight change of variables. Setting $\boldsymbol{\phi}^{(k)} = \boldsymbol{\theta}^{(k)} + \mu\boldsymbol{v}^{(k)}$ and rearranging the formulas a bit, the update equation becomes:

$$\begin{aligned}\boldsymbol{v}^{(k+1)} &= \mu\boldsymbol{v}^{(k)} - \alpha \cdot \boldsymbol{\nabla}_{\boldsymbol{\theta}} L\big(\boldsymbol{\phi}^{(k)}\big)\\ \boldsymbol{\phi}^{(k+1)} &= \boldsymbol{\phi}^{(k)} - \mu\boldsymbol{v}^{(k)} + (1+\mu)\boldsymbol{v}^{(k+1)}\end{aligned}$$

The disadvantage of this change of variables is that now the previous value of the momentum has to be kept in memory, but this is mainly an implementation pitfall one has to care for.

2.4.2 AdaGrad

Another variant of SGD is *AdaGrad* which changes the update in a ways such that the gradients are normalized by the square-root of gradient norms. This reduces the learning rate when gradients are high and increases it when gradients are small. Hence, the algorithms moves faster in shallow and slower in steep regions:

$$\begin{aligned}s^{(k+1)} &= s^{(k)} + \Big\|\boldsymbol{\nabla}_{\boldsymbol{\theta}} L\big(\boldsymbol{\theta}^{(k)}\big)\Big\|_2^2\\ \boldsymbol{\theta}^{(k+1)} &= \boldsymbol{\theta}^{(k)} - \frac{\alpha}{\sqrt{s^{(k+1)}} + \epsilon}\boldsymbol{\nabla}_{\boldsymbol{\theta}} L\big(\boldsymbol{\theta}^{(k)}\big)\end{aligned}$$

The "gradient cache" s is initialized with zero and ϵ is a small number added to the denominator to prevent divisions by zero. Problems of AdaGrad are that the first step can be far off and that the learning rate vanishes over time due to s building up large values (as norms can never be negative), causing learning to stagnate.

2.4.3 RMSProp

RMSProp tackles the problem of AdaGrad that the learning rate vanishes by adding a decay rate $\eta \in [0, 1)$ to it:

$$\begin{aligned}s^{(k+1)} &= s^{(k)}\beta s^{(k)} + (1-\beta)\Big\|\boldsymbol{\nabla}_{\boldsymbol{\theta}} L\big(\boldsymbol{\theta}^{(k)}\big)\Big\|_2^2\\ \boldsymbol{\theta}^{(k+1)} &= \boldsymbol{\theta}^{(k)} - \frac{\alpha}{\sqrt{s^{(k+1)}} + \epsilon} \cdot \boldsymbol{\nabla}_{\boldsymbol{\theta}} L\big(\boldsymbol{\theta}^{(k)}\big)\end{aligned}$$

With a decay of $\beta = 0$, RMSProp is equivalent to AdaGrad. The decay rate should be set to something high like $\beta = 0.9$ as small learning rates cause exploding updates in the first steps. Like for AdaGrad, the first few steps can be quite far off.

2.4.4 Adam

Adam is the state-of-the-art optimizer used nearly every time for training neural networks. It combines the idea of momentum with RMSProp

$$\begin{aligned}
\boldsymbol{v}^{(k+1)} &= \beta_1 \boldsymbol{v}^{(k)} + (1-\beta_1) \cdot \nabla_{\boldsymbol{\theta}} L\big(\boldsymbol{\theta}^{(k)}\big) \\
s^{(k+1)} &= \beta_2 s^{(k)} + (1-\beta_2) \big\| \nabla_{\boldsymbol{\theta}} L\big(\boldsymbol{\theta}^{(k)}\big) \big\|_2^2 \\
\hat{\boldsymbol{v}}^{(k+1)} &= \frac{\boldsymbol{v}^{(k+1)}}{1-\beta_1^{k+1}} \\
\hat{s}^{(k+1)} &= \frac{s^{(k+1)}}{1-\beta_2^{k+1}} \\
\boldsymbol{\theta}^{(k+1)} &= \boldsymbol{\theta}^{(k)} - \frac{\alpha}{\sqrt{\hat{s}^{(k+1)}} + \epsilon} \boldsymbol{v}^{(k+1)}
\end{aligned}$$

where the hatted variables are a bias correction that compensates for the zero-initialization of $\boldsymbol{v}$ and s and only really affects the process in the first few iterations where k i small.

2.5 Learning Rate

All of the above optimizers share at least one hyperparameter: the learning rate α. This hyperparameter is usually the one with the most influence on the learning process. If the learning rate is too small, no learning happens. But if it is too high, the parameters blow up and the loss rises indefinitely. A good learning rate is somewhere im between. Tuning of the learning rate and hyperparameter search will also be covered in section 4.4.

As usually most learning happens in the beginning and later on fine-tuning happens, it can be useful to decay the learning rate over time, i.e., make α k-dependent: α_k. Two options are exponential decay

$$\alpha_k = \alpha_0 e^{-\eta k}$$

with an initial learning rate α_0 and a decay rate η. Another option is $1/k$-decay

$$\alpha_k = \frac{\alpha_0}{1+\eta k},$$

again with an initial learning rate α_0 and a decay rate η. Other options are for example step-based decay (e.g., every n iterations, go done by a factor) or even manual decay.

3 Backpropagation

As already discussed in chapter 2, the gradient of the loss function has to be computed in order to update the parameters of a neural network. The best method for doing this is automatic differentiation which is both fast (compared to numerical evaluation) and not as error-prone as analytical derivatives which have to be both implemented and derived by hand. The heart of automatic differentiation is building a computational graph and applying the chain rule

$$\frac{\mathrm{d}}{\mathrm{d}x} f\big(g(x)\big) = \frac{\partial f\big(g(x)\big)}{\partial g(x)} \frac{\mathrm{d}g(x)}{\mathrm{d}x}$$

multiple times. As an example Figure 3.1 shows the computational graph for $\big((x+y)z\big)^2$, including a forward pass (for computing the output) as well as a backward pass (for computing the derivatives). Computing the derivative goes as follows: start with one on the output (this is the derivative of the output w.r.t. the output). Then subsequently compute the local derivatives for each node with respect to the node the edge is coming from. This local derivative is then multiplied with the *upstream gradient*, i.e., the value of the derivative coming in from the right. This yields the derivative of the output w.r.t. the node the edge the current number is computed for is coming from.

This exhibits some interesting properties of the gradient flow and the patterns several gates induce. First of all, an add gate distributes the gradient between the incoming paths:

$$\frac{\partial}{\partial x}(x+y) = 1 \qquad\qquad \frac{\partial}{\partial y}(y+y) = 1$$

A product gate "switches" the gradient as the first downstream gradient gets multiplied by the value of the second flow:

$$\frac{\partial}{\partial x}(xy) = y \qquad\qquad \frac{\partial}{\partial y}(xy) = y$$

Another interesting gate is the max gate which "routes" the gradient depending on the input values, i.e., the gradient of which the flow has the smaller variable vanishes:

$$\frac{\partial}{\partial x}\max(x,y) = \begin{cases} 1 & \text{if } x > y \\ 0 & \text{if } x < y \end{cases} \qquad\qquad \frac{\partial}{\partial y}\max(x,y) = \begin{cases} 0 & \text{if } x > y \\ 0 & \text{if } x < y \end{cases}$$

An important property here is that the derivative for $x = y$ is undefined as the maximum is not differentiable for $x = y$. In practice, however, exact equality is rarely the case so this is not a real problem. Similarly a ReLU $\max(0, x)$ works like a gradient switch where it can only pass through if the value was positive during the forward pass.

In is also possible go forward through the network. This is called *forward differentiation* opposed to *backward differentiation*. In forward differentiation, the derivative $\partial x/\partial y$ is computed, in backward differentiation the derivative $\partial y/\partial x$ is computed. As neural networks usually have a scalar loss as the output, backward differentiation is trivial and easy to calculate.

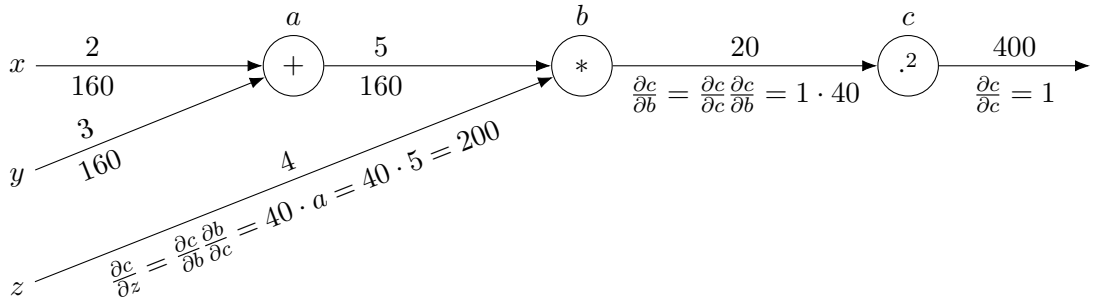


Figure 3.1: Computational graph for $\big((x+y)z\big)^2$. The numbers above the edges represent the forward pass, the numbers below the edges the backward pass (the derivatives). For some edges the application of the chain rule is written out explicitly. All other edges get computed analogous.

In real-world application, the backward pass would be computed in a batched/vectorized fashion for multiple input/output data and weights at once. This can be implemented efficiently using Einstein summation. Listing 3.1 shows a forward- and backward-pass through a multi-layer perceptron with two hidden layers and sigmoid nonlinearities after each hidden layer and no output nonlinearity.

Listing 3.1: Forward- and backward-pass for a multi-layer perceptron.

```
# W1, W2, W3, b1, b2, b3 are the weights/biases of the layers.
hid_1 = sigmoid(W1 @ X + b1[:, np.newaxis])
hid_2 = sigmoid(W2 @ hid_1 + b2[:, np.newaxis])
outputs = W3 @ hid_2 + b3[:, np.newaxis]

dL  = deriv_squared_loss(outputs, targets)
dW3 = np.einsum('ib,jb->ij', dL, hid_2)
db3 = np.einsum('ib->i', dL)

dS2 = deriv_sigmoid(W2 @ hid_1 + b2[:, np.newaxis])
dL  = np.einsum('kb,ki,ib->ib', dL, W3, dS2)
dW2 = np.einsum('ib,jb->ij', dL, hid_1)
db2 = np.einsum('ib->i', dL)

dS1 = deriv_sigmoid(W1 @ X + b1[:, np.newaxis])
dL  = np.einsum('kb,ki,ib->ib', dL, W2, dS1)
dW1 = np.einsum('ib,jb->ij', dL, X)
db1 = np.einsum('ib->i', dL)
```

3.1 Activation Functions

The *activation function* is the nonlinearity behind a linear layer of a neural network, enriching the prediction power of the network. The big problem with activation functions is that—as neural networks lack interpretability—it is not really known what nonlinearities are good for which task. This section discusses some of the most popular activation functions and their respective pros and cons.

Sigmoid

One of the first activation functions was *sigmoid* which squashes all input numbers into a range from zero to one:

$$\sigma : \mathbb{R} \to (0,1) : x \mapsto \frac{1}{1+e^{-x}} \qquad \sigma'(x) = \sigma(x)\big(1-\sigma(x)\big)$$

Sigmoid is best for learning logical inputs, i.e., functions with binary inputs and is used for control signals in LSTM networks. But they can kill gradients for values $|x| \lg 0$ as the derivative is near zero for values far away from the center. Sigmoid is also not good for image networks (better use ReLU) and is not zero-centered. The latter is a problem because the sigmoid itself requires zero-centered input data for producing non-trivial (constant) results.

Additionally, an always-positive input to a neuron causes the gradients to always be all-positive or all-negative. this leads to a zigzag path through the parameter space (much like axial iteration) which is not near an optimal optimization trajectory. This is also why the input data shall always be zero-centered beforehand (i.e., the mean of the input data should be zero).

Hyperbolic Tangent (Tanh)

The shape of the *hyperbolic tangent* (tanh) activation function is similar, but the numbers are squashed into a range from plus to minus one:

$$\tanh : \mathbb{R} \to (-1,1) : x \mapsto \frac{e^x - e^{-x}}{e^x + e^{-x}} \qquad \tanh'(x) = 1 - \tanh^2(x)$$

Much like the sigmoid, gradients are killed for x values away from zero. But as the output is zero-centered, the tanh activation function is better suited for most problems as opposed to the sigmoid. It is also used for bounded, but signed, values in LSTM networks. But they are not as good as sigmoid for binary functions.

Rectified Linear Unit (ReLU)

The *Rectified Linear Unit* (ReLU) is (at the moment) the go-to activation function for most problems:

$$\mathrm{ReLU} : \mathbb{R} \to [0,\infty) : x \mapsto \max(0,x) \qquad \mathrm{ReLU}'(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x < 0 \end{cases}$$

It saturates on half of the real axis and kills the gradient there, but it does never kill the gradient on the other half and is thus better suited for most problems than sigmoid or tanh. This is also visible empirically: networks using ReLU activations converge around six times faster than networks using sigmoid/tanh activations. But it is also not suitable for logical functions or for control in recurrent networks. Also, the output is not zero-centered. Another (theoretical) problem is that the gradient at $x = 0$ is not defined. In practice this is not a big problem as values are almost never exactly equal to zero. If they are, one usually chooses zero or one as a gradient value, it does not really matter.

To prevent the gradients from dying (a saturated ReLU is often called a *dead ReLU*), the network is usually initialized with a slightly positive bias (e.g., 0.01).

Leaky and Parametric ReLU

An alternative to the vanilla ReLU are the *leaky* and *parametric* ReLU. The parametric ReLU is given as

$$\mathrm{PReLU}_\alpha : \mathbb{R} \to (-\infty, \infty) : x \mapsto \max(\alpha x, x) \qquad \mathrm{PReLU}'_\alpha(x) = \begin{cases} 1 & \text{if } x > 0 \\ \alpha & \text{if } x < 0 \end{cases}$$

where for the leaky ReLU the parameter is fixed to $\alpha = 0.01$. This parameter is also learnable. Compared to all activation functions discussed before, the leaky and parametric ReLU do not saturate and have most advantages of the ReLU (e.g., faster convergence than sigmoid/tanh). Also the outputs are closer to zero-mean compared to a vanilla ReLU. But it is still not differentiable at $x = 0$.

Exponential Linear Unit (ELU)

The *exponential linear unit* (ELU) combines the pros of a ReLU with differentiability at $x = 0$:

$$\mathrm{ELU} : \mathbb{R} \to (-\alpha, \infty) : x \mapsto \begin{cases} x & \text{if } x > 0 \\ \alpha\big(e^x - 1\big) & \text{if } x \leq 0 \end{cases} \qquad \mathrm{ELU}'(x) = \begin{cases} \alpha e^x & \text{if } x > 0 \\ 1 & \text{if } x < 0 \\ 1 & \text{if } x = 0 \text{ and } \alpha = 1 \end{cases}$$

As can be seen from the derivative, if $\alpha = 1$ is chosen, the function is continuously differentiable everywhere. Otherwise it is not differentiable at $x = 0$. The value of α defines the value of the ELU as $x \to -\infty$.

Maxout Neuron

The *maxout neuron* is a special kind of neuron that combines the weighting and bias with the activation function:

$$\max\big(\boldsymbol{w}_1^T \boldsymbol{x} + b_1,\, \boldsymbol{w}_2^T \boldsymbol{x} + b_2\big)$$

It generalizes the parametric ReLU and works in the linear regime, and hence does not saturate. But it has the problem that it doubles the number of parameters as two weight matrices and biases have to be used.

In Practice...

...everything should be tried, usually ReLU. But also try leaky ReLU, Maxout and ELU. For binary (logical) functions, use sigmoid and also try out tanh (but do not hope for much).

3.2 Regularization

The number of layers and neurons per layer has an extreme impact on the performance and capacity of a network. In general more layers work better over more neurons per layer and more layers/neurons in general work better than less. But large networks can lead to overfitting when there are a lot more parameters then there is data! But instead of regularizing the network using the network size, it is better to use a strong regularization on the weights instead, e.g., L1- or L2-regularization. In these regularization methods, the sum of the absolute (L1) or squared (L2) weights is added to the loss with a penalty factor λ (usually 0.1, 0.01, or 0.001). This keeps the parameters small and thus reduces model complexity.

4 Training Neural Networks

This chapter covers the complete learning process of neural networks. Some quirks and tricks for data pre-processing and normalization are covered as well as hyperparameter optimization and regularization.

4.1 Data Pre-Processing: Normalization

As already seen during the discussion of activation functions in section 3.1, it is desirable to have zero-centered input data, i.e., data with a zero mean. Because of that it is common (and recommended) to first *normalize* the input data before training by subtracting the mean. Additionally the data is also often *standardized*, meaning the normalized data is divided by the standard deviation of all samples to get a variance of one in every direction. This way the data does not fluctuate much along all axis which eases the training. It is also common to *whiten* the data by applying a PCA (Principal Component Analysis) transformation to it, producing a unit Gaussian and decorrelating the features by rotating the data (such that the covariance matrix is the identity matrix).

For images, however, it is common to not standardize or whiten the data but to just subtract the mean (e.g. the mean image in AlexNet of the per-channel mean in VGGNet). This is due to the numbers of an image being bounded by nature, e.g. to the interval from zero to one. Hence different variances along the different axis might even encode information that shall not be removed.

During test time, the test input data has to be shifted and scaled the same way the training data was, based on the mean/standard deviation/PCA matrix/... that was computed from the training data. So the mean is not re-computed during test time!

4.2 Weight Initialization

An important factor of training neural networks is the initialization of the weights and biases in the neurons. If, for example, all weights are initialized with zeros, all downstream gradients vanish as they get multiplied with the weights. Hence, no learning happens. Besides the methods proposed below, initialization remains an open topic in neural network research and is not well understood. What is well understood, however, is that initialization plays a really important role and more often than expected is the key factor for the success of a neural network[1].

Small or Big Random Numbers Another idea is to initialize all weights by sampling from a Gaussian distribution with a small standard deviation, e.g. $\sigma = 0.01$. This works fine for small networks, but for deep networks this initialization causes the mean and standard deviation of the activations in each layer to collapse quickly to zero. This is due to multiple small numbers getting multiplied together a lot of times. But activations with (near) zero values cause the gradient of the weights to vanish! Like for an initialization with all zeros, this causes no learning to happen.

[1] Jonathan Frankle and Michael Carbin (2018): "The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks"

So small numbers are not the key to success. On the other hand, using a variance of $\sigma = 1$ for generating the initial weights does not work out either because the activation functions saturate quickly, yielding vanishing gradients. Hence again, no learning happens. For ReLU, this can also lead to exploding values as ReLU is not bounded.

Figure 4.1 shows the activation statistics of random data propagating through a ten-layer neural network with 300 neurons each. It can be seen that the mean and standard deviation saturate quickly, characterized by the histograms getting sharper. Figure 4.2 shows the same kind of plot, but for the big standard deviation. In this case the ReLU activation function is not shown in the mean/standard deviation plots because it blows up exponentially. But for sigmoid and tanh it can be seen that they saturate, too, causing the gradients to vanish.

Xavier Initialization *Xavier initialization* also uses normally distributed weights, but the standard deviation is reciprocally proportional to the square-root of the number of neurons in the previous layer (called *fan in*): $\sigma = 1/\sqrt{\textit{fan-in}}$. This is a reasonable activation and works well when using sigmoid or tanh activations. But when using ReLU activations, the gradients vanish again as nearly all activations get saturated in the left half of the ReLU (where it is flat). Instead, He et al. (2015) proposed to divide the fan in by two to get a better initialization, i.e., $\sigma = 1/\sqrt{\textit{fan-in}/2}$. This works reasonably well also for ReLU activations.

Figure 4.3 shows the activation statistics like before. For tanh the mean approaches zero, but this is a good thing as zero-centered data reduces saturation in the next layer as long as the histogram is sufficiently wide. But it does not work good for ReLU which saturates quickly. The alternative Xavier initialization proposed by He et al. is shown in Figure 4.4, yielding similar results for sigmoid and tanh. But even for the ReLU, the histogram is not as sharp as for vanilla Xavier initialization.

4.2.1 Batch Normalization

As already seen, unit Gaussian data is best as an input for most activation functions. The idea of batch normalization to just make the data unit Gaussian by subtracting the mean and dividing by the (empirical) standard deviation at each layer. This is done by averaging the incoming data along each dimension and computing the standard deviation for each dimension. Then element-wise subtraction and division is used for normalize and standardize the data. As the network then learns weights and biases on this data, it is theoretically possible that the networks learns to recover the original data by learning the values of the mean and standard deviation. Hence, no prediction power is taken away from the network by adding a batch normalization layer[2].

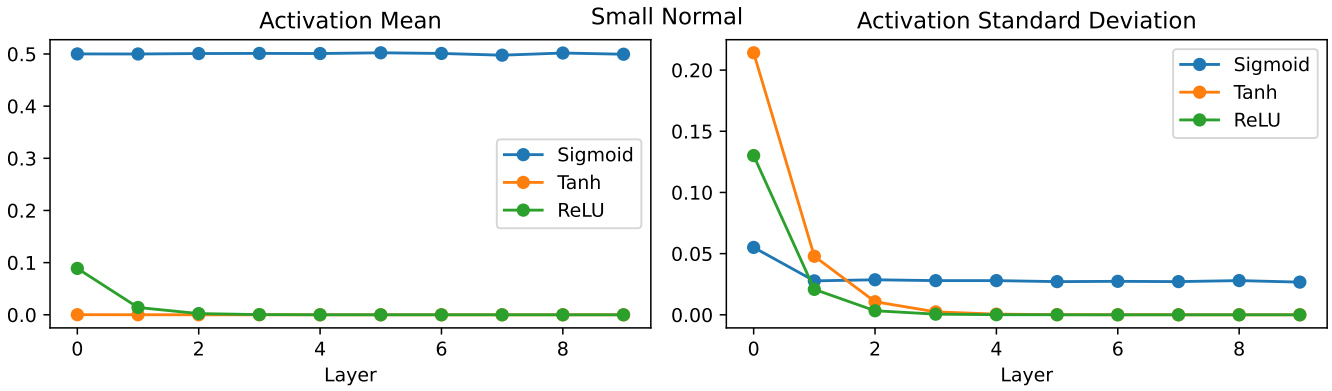
Advantages of batch normalization is that it improves the gradient flow through the network, allows higher learning rates by high-quality gradients, and reduces the strong dependence on initialization. It also kind of works as a form of regularization and maybe reduces the need for dropout.

During test time, the mean and standard deviation that were computed during training are used to normalizing/standardizing the data—the test data is not used to compute the mean/standard deviation again.

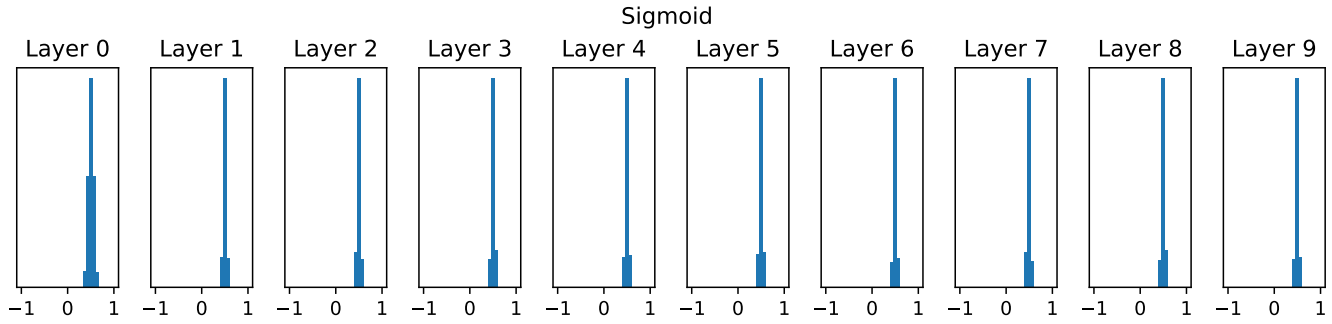
4.3 Designing a Network and Hyperparameter Optimization

This section covers the basic steps of designing a neural network, i.e., choosing the architecture, the loss, etc. The first step is to pre-process the data (normalization) as this is usually useful. The next step is to choose the architecture. This heavily depends on the task and prior intuition and knowledge are extremely helpful for settings up a good initial network. Furthermore, double check that the loss is reasonable: do not use

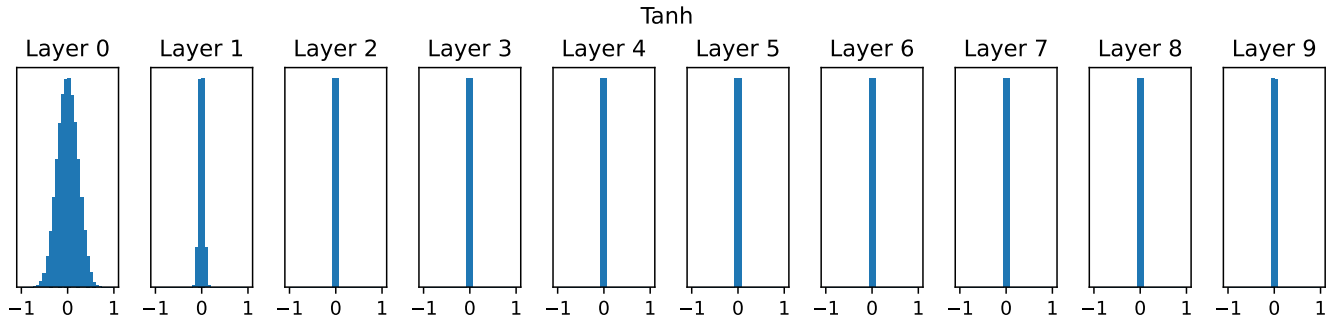
[2]Note that this is just a semi-theoretical argument to support batch normalization. In practice, the neural network nearly never learns the mean and standard deviation



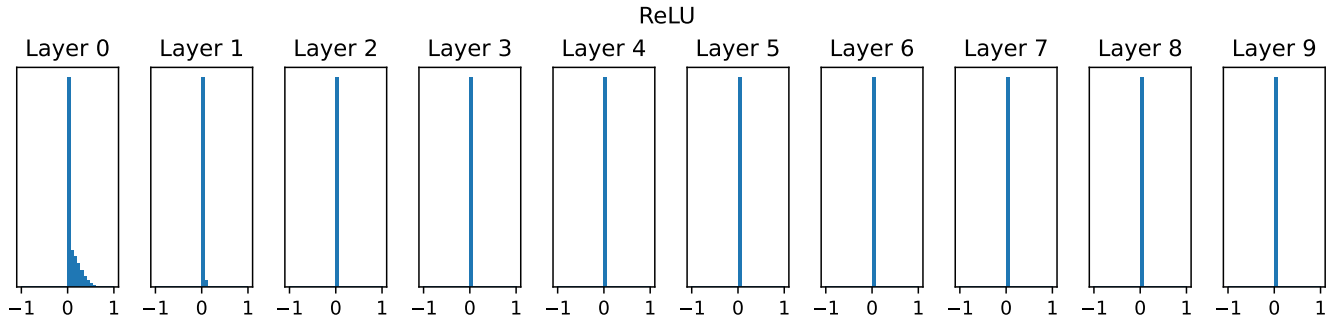
(a) Mean and standard deviation in each layer.



(b) Histogram of the values in each layer for the sigmoid activation function.

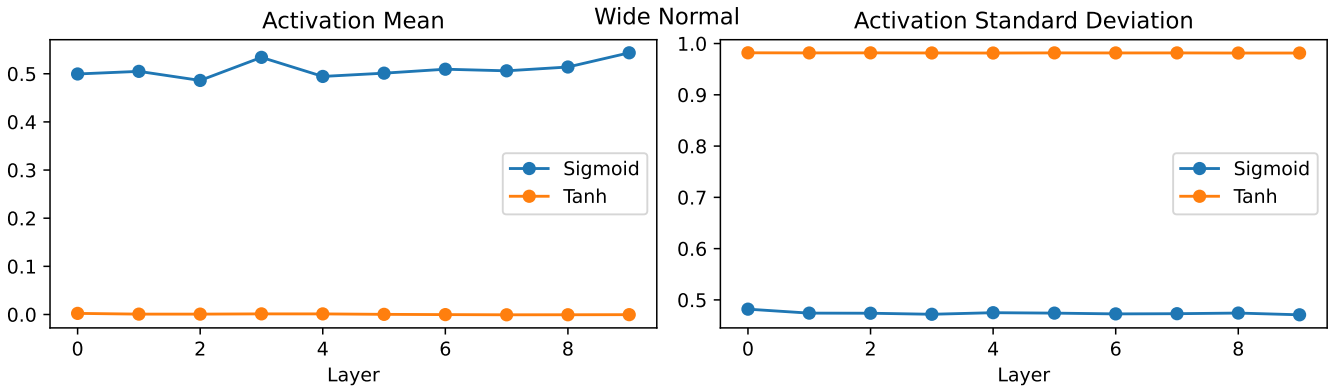


(c) Histogram of the values in each layer for the tanh activation function.

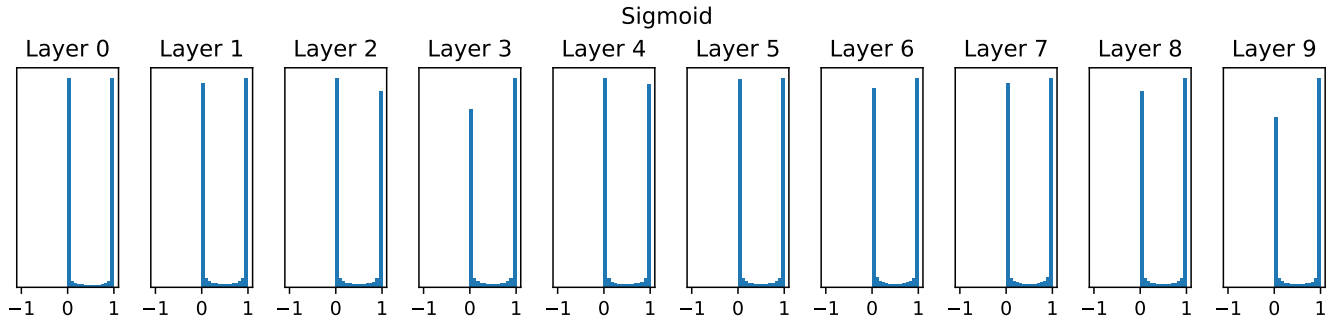


(d) Histogram of the values in each layer for the ReLU activation function.

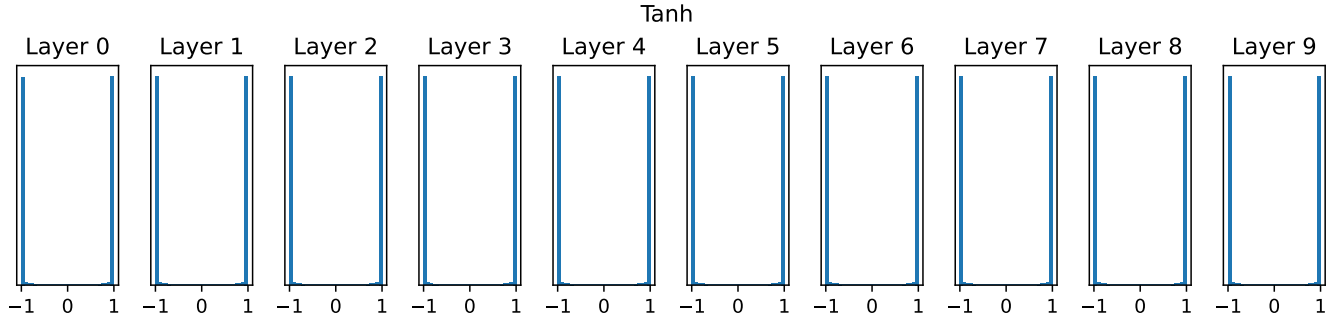
Figure 4.1: Activation statistics for weights initialized by a Gaussian with small variance. Layer zero is the input layer. For all activation functions it can be seen that the mean saturates quickly and the standard deviation vanishes.



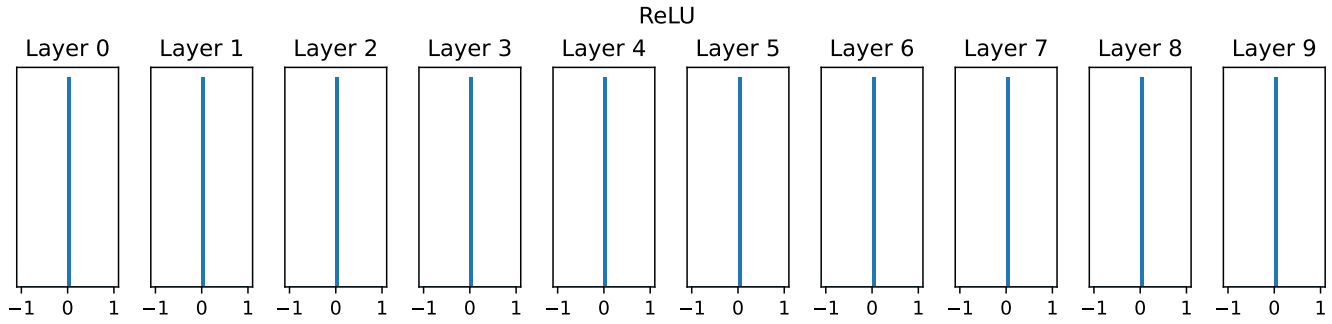
(a) Mean and standard deviation in each layer.



(b) Histogram of the values in each layer for the sigmoid activation function.

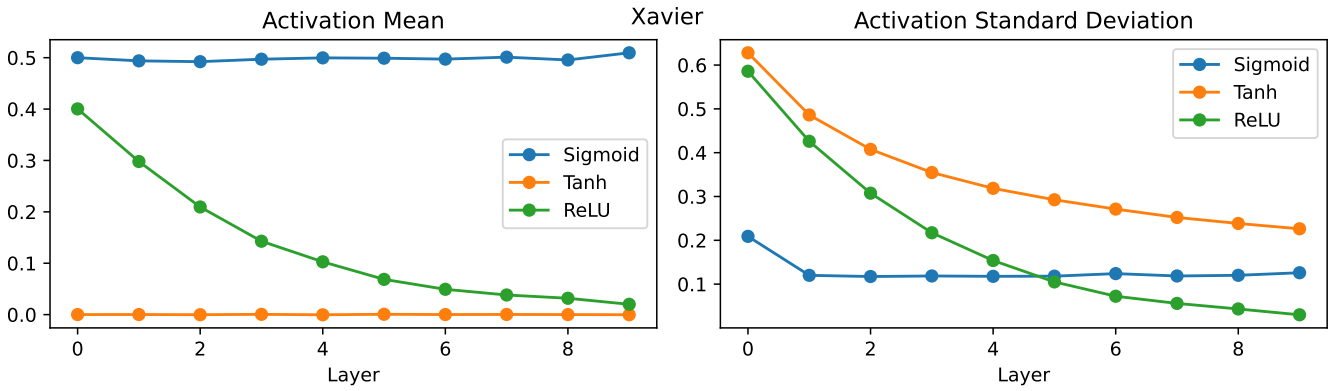


(c) Histogram of the values in each layer for the tanh activation function.

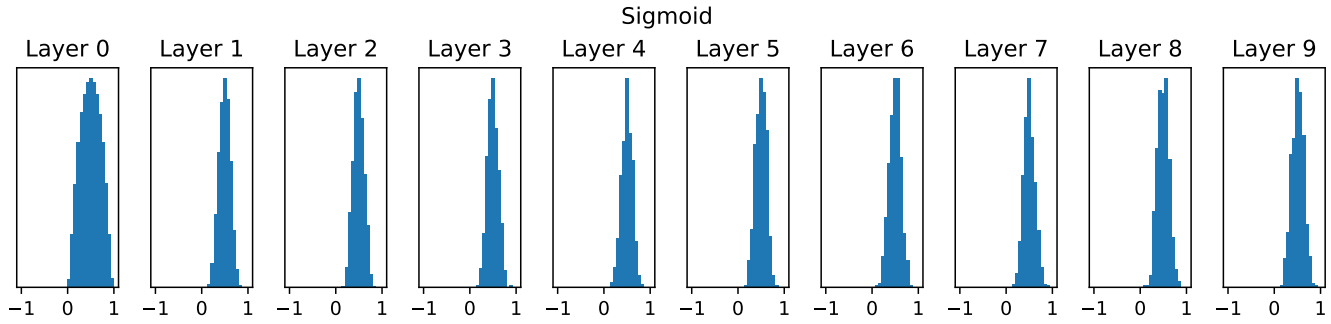


(d) Histogram of the values in each layer for the ReLU activation function.

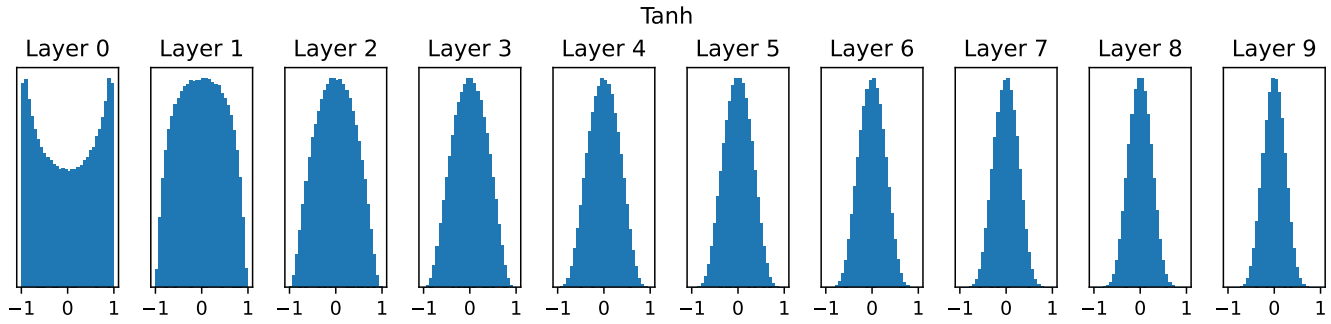
Figure 4.2: Activation statistics for weights initialized by a Gaussian with big variance. Layer zero is the input layer. For all activation functions it can be seen that the neurons saturates quickly. The standard deviation is not zero, but this is caused by the sigmoid and tanh activation to have two saturation values.



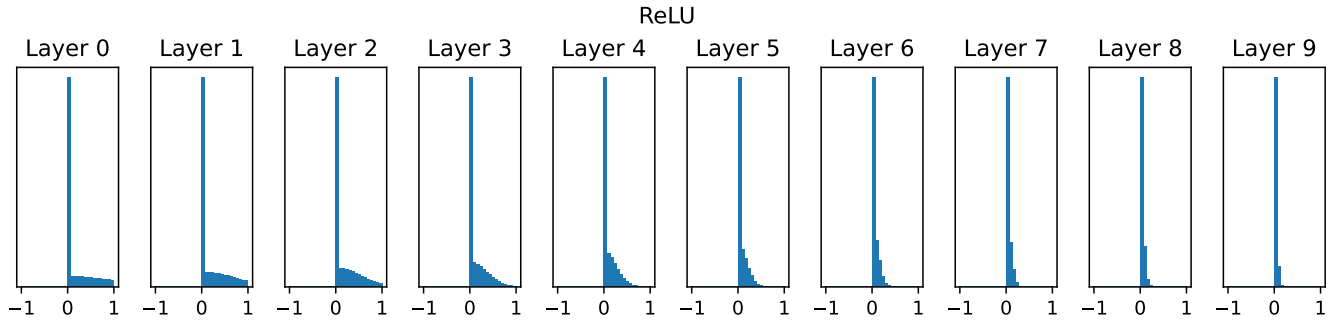
(a) Mean and standard deviation in each layer.



(b) Histogram of the values in each layer for the sigmoid activation function.

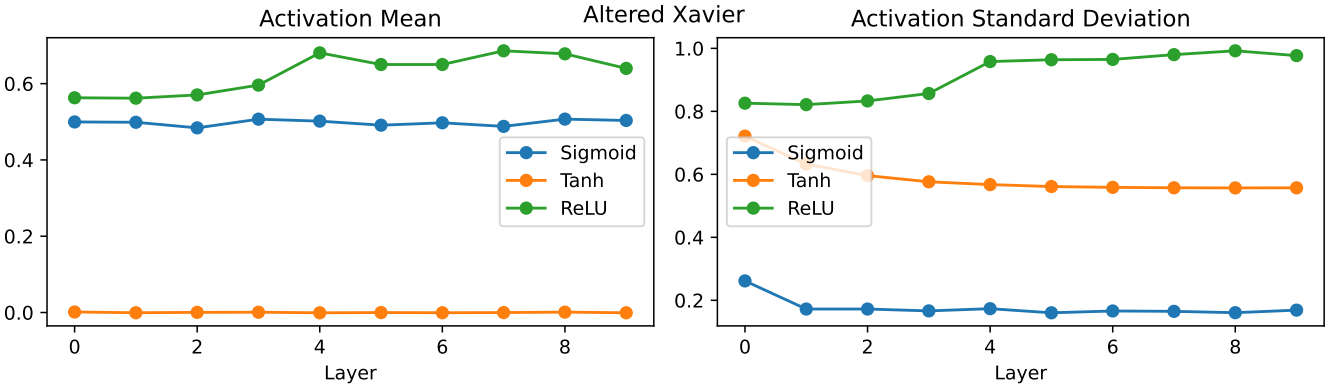


(c) Histogram of the values in each layer for the tanh activation function.

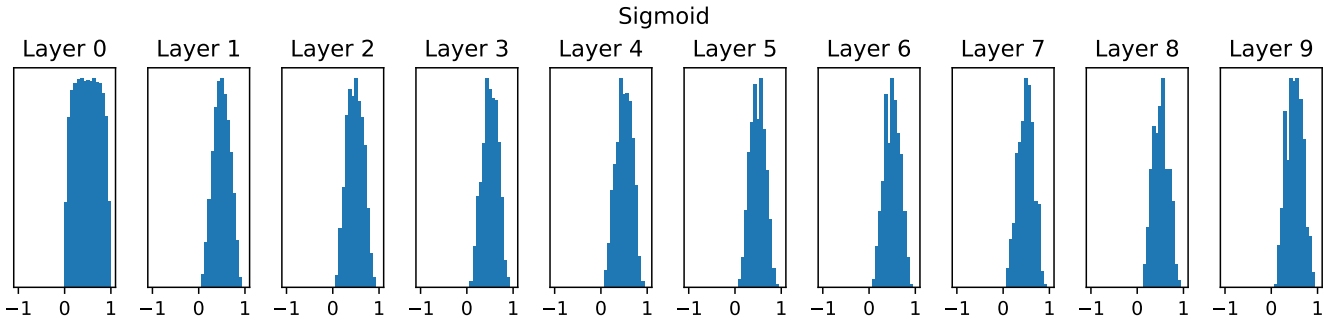


(d) Histogram of the values in each layer for the ReLU activation function.

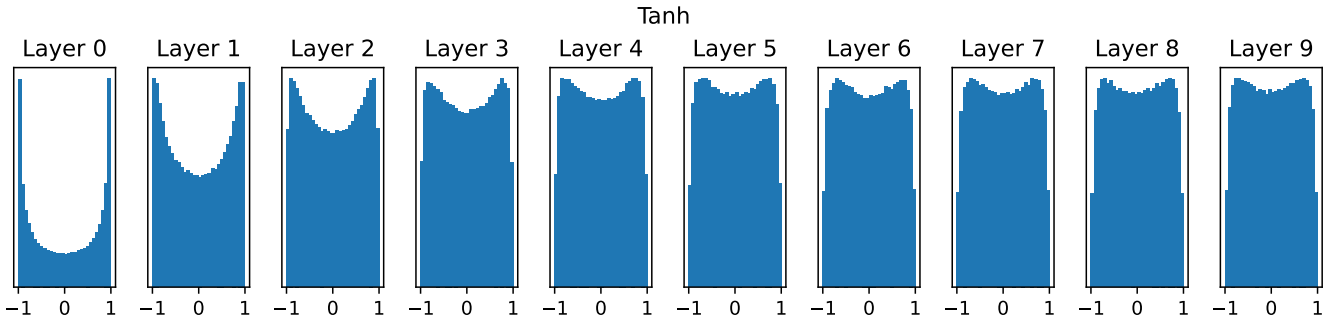
Figure 4.3: Activation statistics for weights initialized using Xavier initialization. Layer zero is the input layer. Except for the ReLU activation, Xavier initialization works well and the neurons do not saturate.



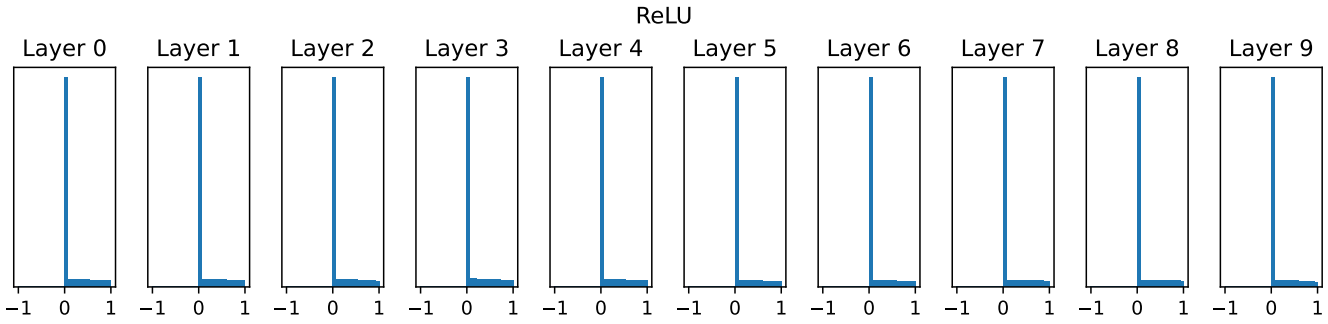
(a) Mean and standard deviation in each layer.



(b) Histogram of the values in each layer for the sigmoid activation function.



(c) Histogram of the values in each layer for the tanh activation function.



(d) Histogram of the values in each layer for the ReLU activation function.

Figure 4.4: Activation statistics for weights initialized using the modified Xavier initialization by He et al. Layer zero is the input layer. It can be seen that the histograms in the layers are sufficiently wide, enabling learning. Also the ReLU does not saturate.

cross-entropy for regression or quadratic loss for classification and remember to disable any regularization tricks. When enabling regularization, the loss should go up—that can be checked, too.

The next step is to train the network on a small portion of the training data to see if the model can overfit. If the model is not able to overfit on a very small dataset, its capacity is not high enough and the neural network has to be made larger. If that works, switch to training on the complete dataset and tuning of the learning rate. If the loss is not going down, the learning rate is usually too low. If the loss is exploding, the learning rate is usually too high. A principled way of finding a good learning rate is to use cross-validation on several learning rates that are magnitudes apart (a learning rate of 0.0002 usually performs similar to 0.0001, so try out 0.1, 0.01, 0.001, $\dots$). This boils down to hyperparameter optimization as the learning rate is a hyperparameter.

4.4 Hyperparameter Optimization

Hyperparameter optimization is a big topic in designing neural networks and deep learning in general. It usually boils down to trying out different hyperparameter settings in a principled way. As a first step, a coarse search for only a few epochs should be done to get a rough idea what parameter settings work. Afterwards, a finer search for more iterations is executed to get more precise parameters. To speed up this process it is often advisable to stop the learning if the loss gets more than three times as high as the original loss. Also, the update ratio, i.e., the ratio between the norm of the old weights and the norm of the new weights $\|\boldsymbol{\theta}^{(k)}\|_2 / \|\boldsymbol{\theta}^{(k+1)}\|_2$, should be around 0.001 or so.

For the coarse search it is best to search in log-space. This is especially true for the learning rate which is the most important parameter. Other hyperparameters to try are for example the network architecture itself, the decay schedule of the learning rate and its update type, regularization (L2/dropout strength), etc. When looking at the accuracy, there should also be a small gap between the training and test accuracy. If the gap is too big, the model overfitted and the regularization strength should be increased. If these is no gap, the model capacity is most likely too low and the network should be enlarged.

4.4.1 Grid vs. Random Search

When evaluating multiple parameters at once, it is advisable to not evaluate them on a grid, but to sample each from a uniform distribution. It is often the case that one hyperparameter (e.g. the learning rate) has a higher influence on the result than another (e.g. the initialization of the bias). If one uses a grid, a lot of redundant computations will be performed for a single parameter. When using random values, this happens less frequent. This is illustrated in Figure 4.5.

4.5 Ensembles

Ensembles leverage multiple simpler models for constructing a single big model with strong prediction power. There are two basic types for ensembles: *Bagging* (Bootstrap Aggregation) and *Boosting*. In bagging, multiple models are trained on samples of the data and a majority vote (for classification) or average (for regression) is used for prediction. In *boosting*, the learners are ordered and training takes place one after another. Then each learner tries to reduce the residual error of the previous trainers on the examples that where misclassified. Bagging usually reduces the variance whereas boosting reduces the bias of the model (and also probably the variance, but this is not true for every model). In both cases, the resulting prediction is a sum or vote of the base learner predictions.

Usually bagging yields around $2\,\%$ of extra accuracy, so models can be made better but one can not fix a broken model using bagging. Hence, neural networks are fairly often used with bagging. But they are only

(a) Grid Search (b) Random Search

Figure 4.5: Comparison of grid search (left) and random search (right). It can be seen that for random search, more (nine) values of the important parameter are tried while when using grid search, only three values are tested.

rarely used with boosting. The reason for this is that deep neural networks usually model global effects and wide networks model local effects, whereas boosting models from global to local. This means that the first models of a boosting sequence models the global effects and later models are used for fine-tuning locally. So boosting with the same network architecture does not yield good results because the shift from global to local is not representable. An open questions is if it is possible to move from deep to wide networks the further the process of boosting gets. The problem that arises in this case is that it is hard to detect whether the next model in a boosting process models global or local properties.

A common disadvantage of bagging and boosting is of course that it takes longer to train compared to a single model. One trick to circumvent this in bagging is to average multiple model checkpoints, i.e., the weights of a network at different iterations, rather than completely different models.

4.6 Dropout Regularization

As neural networks have lots of parameters, sometimes more than there is training data, overfitting is a big problem in deep learning. One regularization technique for tackling this is *dropout*. The core concept of dropout is to randomly set some neurons to zero with probability p for each neuron during the forward pass, disabling them for the prediction. The idea is that this forces the network to learn redundant representations, making it more robust towards changes in the inputs. Another interpretation is that dropout trains a large ensemble of models that share parameters.

During test time, it would be ideal to integrate out all the noise, but in practice this is obviously intractable. Instead, the result will be approximated using Monte Carlo integration techniques: calculate multiple forward passes using different (random) dropout masks and average all predictions. But in fact this can be done (approximately) with a single forward pass with all neurons being active! To compensate for the inflation caused by roughly $1/p$ more neurons being active during the fully activated forward pass, all activations have to be scaled by p.

As scaling the activations at test time reveals details about the training process[3], it is also possible to scale the activations by $1/p$ in the training forward pass. This is called *inverted dropout*.

[3]Take, for example, a model that is deployed on lots on end-user devices (like a face detection model). If the developer finds a better way of training that improves the model, but uses dropout, the end-user model would have to be changed just because the training process changed.

4.7 Gradient Clipping

In gradient descent, big gradients cause divergence while small gradients cause slow convergence. As divergence is much worse and gradients can explode pretty fast, a technique called *gradient clipping* can be used. Gradient clipping limits the magnitude of each gradient

$$\hat{g}_i = \min\left(g_{\max}, \max(-g_{\max}, g_i)\right)$$

such that $|\hat{g}_i| \leq g_{\max}$. Then a decreasing learning rate is used to converge to an optimum. Standard gradient clipping limits the larges gradient dimensions while others max be very small.

An alternative, *extreme* gradient clipping which is used in AdaGrad and RMSProp scale all dimensions by the inverse standard deviation such that all dimensions have unit standard deviation. An even more extreme approach are one-bit gradients where all gradient dimensions are clipped such that only their sign is left. This actually works on some problems! But usually, gradient clipping is not used this aggressive.

4.8 Gradient Noise

In stochastic gradient descent, the gradient resulting from the Monte Carlo approximations is noisy, but SGD still makes good progress on average. This gives the idea of whether it might be good to add a little noise to the gradients? Experiments have shown that this is actually the case, reducing overfitting in complex models. Usually the noise is additive

$$\hat{g}_i^{(k)} = g_i^{(k)} + \epsilon_t^{(k)}, \quad \epsilon_t^{(k)} \sim \mathcal{N}(0, \sigma_k^2)$$

with variance

$$\sigma_k^2 = \frac{\eta}{(1+k)\gamma}$$

where $\eta \in \{0.01, 0.3, 1\}$ and $\gamma = 0.55$. This way the noise added to the gradients is reduced over time, producing more stable gradients towards the end.

Gradient noise raises some interesting theoretical questions as it turns the model parameters into a Bayesian inference task where the noise magnitude controls the temperature of the distribution (higher noise $\rightarrow$ higher temperature). Cf. momentum which reduces the level of detail.

4.9 Vanishing Gradients and Residual Networks

The *vanishing gradient problem* is like the opposite of exploding gradients: if an activation function like tanh is used which squashes all numbers between zero and one and only produces gradients between zero and one, multiplying many of these numbers together quickly approaches zero. As many activation values and activation function gradients are stacked on top of each other in a deep network, this causes the gradients to *vanish* or *die*. One way to circumvent this is to use *residual networks* with add *skip connections* from one layer to later layers (see Figure 4.6).

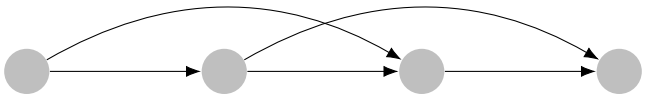


Figure 4.6: Illustration of the skip connections in a residual network. To compute the gradient of a node with two outgoing connections, the two incoming gradients are added together. It is also possible to add connections that skip more than one layer.

5 Convolutional Neural Networks

While fully connected neural networks (multilayer perceptrons) are extremely powerful, they are not really well suited for processing large images. This is due to the shear amount of parameters that would be needed for the first layers as all pixels are connected to every neuron. But images do not really have a global correlation, the spatial correlation is fairly local! Hence, fully connected layers are just a waste of resources. Instead, *convolutional neural networks* (CNNs) use convolutions from computer vision as learnable filters to extract locally correlated information first and later plugging the downscaled result into fully connected layers for further processing.

A convolution is described by an $F_y \times F_x$ matrix that gets shifted along the input image using element-wise products and sums for computing a pixel at that point. Figure 5.1 illustrates this for a 3×3 filter on a 5×5 image. Obviously, the size of the image shrinks because the size of the filter. This can be circumvented by adding *zero-padding*. This means that the edges of the image get extended by P_x/P_y zeros on all sides such that the image is bigger. It is also possible to specify a *stride* S_x/S_y along the width/height which prescribes the number of pixels the filter is moved in every step. The default is a stride of one which is also used in Figure 5.1 (also referred to as "no stride"). Higher values for the stride result in a smaller output image.

In real-world applications, input images may have multiple *channels*. A channel is one layer of the image, e.g. the "red" layer in an RGB image. Usually images have three (RGB) or four (RGBA, with transparency) channels. Then a filter is created for each layer and the results of all these filters are added together after applying them. It is common to add a bias (which, for a single filter, is a scalar value) that gets added to each resulting pixel. Usually in CNNs multiple filters are used per layer in the network resulting in as many output channels as filters where used. So the filters do not interact within a layer, but the next layer can combine them as needed. A really good visualization of a filter processing an image is given here:

`https://cs231n.github.io/assets/conv-demo/index.html`

Normally all parameters in a filter layer (weights and biases of the filters) are learnable, resulting in

$$\#\text{Parameters} = D\big(dF_xF_y + 1\big)$$

parameters where d is the number of input channels, D is the number of output channels, and F_x/F_y are the size of each filter (width/height). The *activation volume* has the spatial size

$$O_{x/y} = \frac{I_{x/y} + 2P_{x/y} - F_{x/y}}{S_{x/y}} + 1$$

where $O_{x/y}$ is the width/height of the output data, $I_{x/y}$ is the width/height of the input data, $P_{x/y}$ is the horizontal/vertical padding, and $S_{x/y}$ is the horizontal/vertical stride. If square input data and filters are used, the equations simplify to

$$\#\text{Parameters} = D\big(dF^2 + 1\big) \qquad\qquad O = \frac{I + 2P - F}{S} + 1.$$

In convolutional networks, these kind of layers are then stacked on top of each other, forming a feature hierarchy (like in multilayer perceptrons). As usual, it is not good to shrink down the spatial dimensions

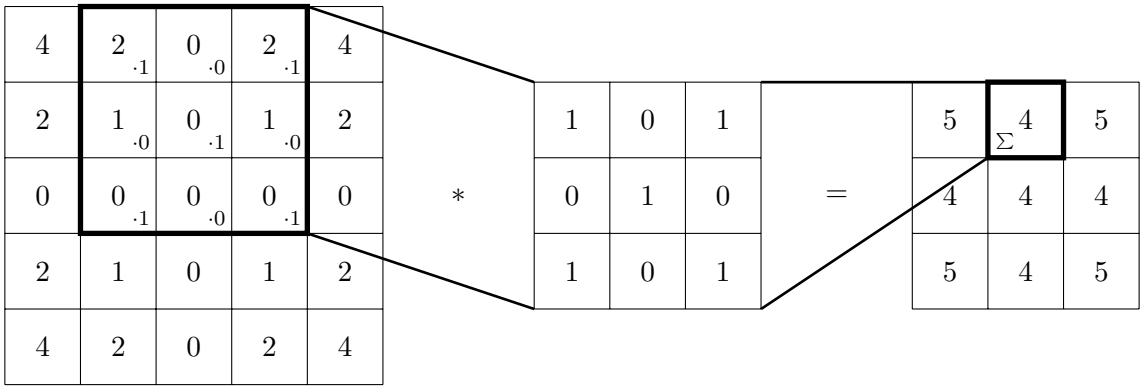


Figure 5.1: Illustration of a 3×3 convolution along an 5×5 image.

too slow nor too fast. Common settings are power of two for the number of filters and filter sizes, padding and stride of $(F, S, P) = (3, 1, 1)$, $(F, S, P) = (5, 1, 2)$, $(F, S, P) = (5, 2, ?)$[1], or $(F, S, P) = (1, 1, 0)$. Usually the convolutional part of a network is followed by some fully connected layers for performing the actual classification/regression task. The interpretation is that the convolutional layers produce good features that can then be used by the following multiplayer perceptron.

It is also perfectly fine to use a scalar filter which basically results in a dot product between the layer and the filters.

5.1 Pooling

A *pooling* layer scales the input down by combining multiple pixels into a single pixel (like a convolution), but usually destructive operations like a maximum (known as *max pooling*) are used. In contrast to convolutional layers, pooling layers operate independently on the input channels, producing as many output channels as there are input channels (i.e., the results to not get combined).

5.2 Case Studies

This section covers some case studies (concrete models) used for certain tasks, usually classification.

Like previously for multi-layer perceptrons, the trend goes toward using smaller filters and instead using deeper architectures. Another trend is to get rid of pooling and fully connected layers and instead just use convolutional layers.

LeNet-5 LeNet-5 was proposed by LeCun et al. in 1998 for classification on the MNIST dataset:

1. Convolution, 6 filters, size 5×5, stride 1, pad 2.
2. Max pooling, size 2×2, stride 2.
3. Convolution, 16 filters, size 5×5, stride 1, pad 0.
4. Max pooling, size 2×2, stride 2.

[1] Whatever fits for the padding.

5. Fully connected, 120 neurons.
6. Fully connected, 84 neurons.
7. Fully connected (output), 10 neurons.

AlexNet AlexNet was proposed by Krizhevsky et al. in 2012 and is more advanced and evolved than LeNet-5:

1. Convolution, 96 filters, size 11×11, stride 4, pad 0.
2. Max pooling, size 3×3, stride 2.
3. Normalization.
4. Convolution, 256 filters, site 5×5, stride 1, pad 2.
5. Max pooling, size 3×3, stride 2.
6. Normalization.
7. Convolution, 384 filters, size 3×3, stride 1, pad 1.
8. Convolution, 384 filters, size 3×3, stride 1, pad 1.
9. Convolution, 256 filters, size 3×3, stride 1, pad 1.
10. Max pooling, size 3×3, stride 2.
11. Fully connected, 4096 neurons.
12. Fully connected, 4096 neurons.
13. Fully connected, 1000 neurons.

Further details: AlexNet was the first model to use ReLU activations and used normalization layers which is not common anymore. Furthermore heavy data augmentation was done as well as dropout with $p = 0.5$.

5.3 Transfer Learning

A common saying about CNNs and computer vision tasks in general is that they take a lot of data for training. While is is true for training untrained models, it is possible to use pre-trained models and fine-tune them on the data at hand. This is called *transfer learning*. The idea is to take a trained deep CNN that was, for example, trained on ImageNet and plug an untrained *classification head*, fix all weights of the "body" of the model and only train the weights and biases of the head. This way deep features provided by the pre-trained model can be used as these are utterly effective, but training time and the amount of data needed is still relatively low. If more data is available, it might also be possible to fine-tune some of the convolutional layers for even better accuracy, but this is often not even needed. Tip: use a smaller learning rate for fine-tuning the final layers (around $1/10$ of the original learning rate) and use even smaller rates for intermediate layers (around $1/100$ of the original learning rate).

6 Computer Vision Tasks

This chapter covers tasks that are common in computer vision like classification, localization, and object detection. In classification, the task is to classify a given image based on what is in it. In combined classification and localization, the image is classified and a bounding box is detected around the object that was classified. Object detection describes the same but for multiple objects, so multiple different objects can be classified and localized within an image. In *image segmentation*, the bounding box is extended so a polygon that gets built around the objects. This is the hardest task of the four. Figure 6.1 shows a comparison of the four tasks.

6.1 Combined Classification and Localization

In classification, the input to the model is an image and the output is a class label. In localization, the input to the model is an image and the output is a box in the image, given by its coordinates and its width/height.

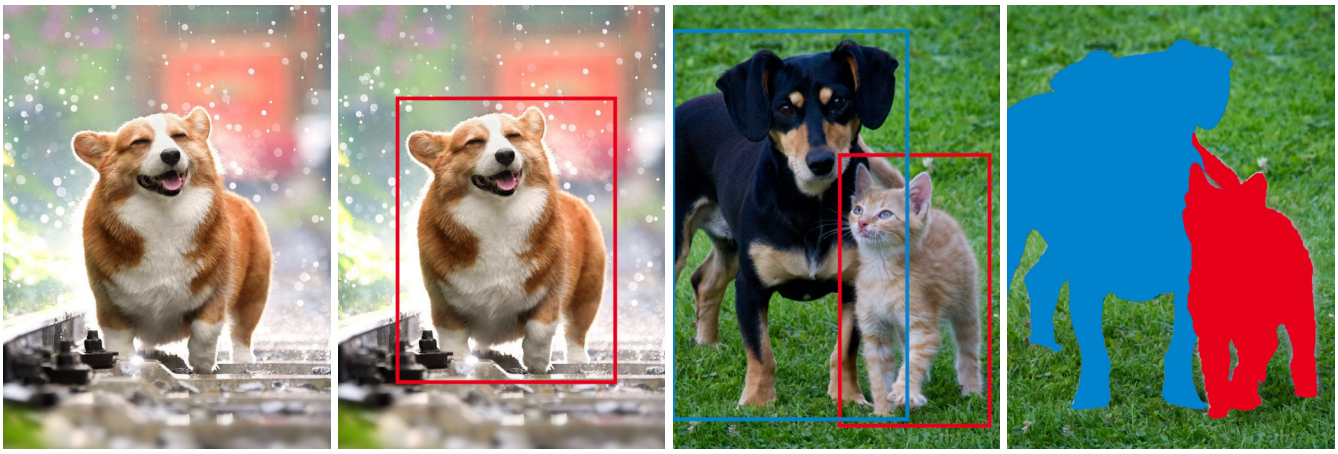
Interpreting localization as regression is a pretty natural thing to do as the output is (kind of) continuous. For training such a model, a cross-entropy loss is used for classification and an L2 loss is used for regression separately. The process is as follows: firstly, train or download a classification model. Secondly, attach a new regression head to the network that will be used for localization. The network is now two-headed. Then fix the downloaded CNN and train only the regression head using the L2 loss and after that potentially fine-tune the classification head. At test time, both heads are used simultaneously to retrieve both class scores and box coordinates.

For localizing multiple objects in an image (e.g. whole face, left ear, right ear, etc.), the regression head can also output $4K$ numbers where K is the number of objects to localize. Compared to object detection, the number of objects that are detected is fixed and there is only one class label for the image. One application is for example human pose estimation where the K objects represent the K joints of a human.

To further optimize the accuracy of the predictions a *sliding window* can be used that slides over the input image and executes the classification and regression network multiple times. The predictions are then combined into a single final prediction. One example model that uses this technique is Overfeat which uses many sliding window locations and multiple scales. Also, the fully connected layers are turned into convolutions for more efficiency.

6.2 Object Detection

Compared to combined classification and localization, in object detection a variable sized output is necessary. One approach is again using a sliding window which has the disadvantage that the classifier has to be executed lots of times. But if the model is sufficiently fast, this is a good approach. Additionally it is possible to only look at a tiny subset of possible positions using *region proposals*. Region proposals first find “blobby” image regions which probably contain objects and then a classification model is applied to these proposals for detecting objects. This massively reduces the amount of regions that need to be scanned such that powerful, yet resource hungry, models like CNNs can be used. This then forms the *R-CNN* model.



(a) Classification
Dog

(b) Class. & Localization
Dog

(c) Object Detection
Dog, Cat

(d) Object Segmentation
Dog, Cat

Figure 6.1: Comparison of computer vision task, from the easiest (left) to the hardest (right).

But in the end, all approaches discussed here are really the human way. One thing is that given an image and a classification task, humans often do not detect objects which are far larger than usual, but deep CNNs do[1].

6.2.1 R-CNN

R-CNN combines regions proposals with CNNs to form an efficient detector. For training R-CNNs, the process is as follows:

1. Train or download a classification model for ImageNet (e.g. AlexNet).
2. Reinitialize and fine-tune the final fully connected layer on a small portion of the classes that should be detected.
3. Extract training features by extracting the region proposals and cropping/scaling them such that they fit into the CNN. Then save all outputs of the final pooling layer.
4. Train a binary support vector machine per class to classify the features extracted in the previous step.
5. Train a linear regression model per class to map from the images to offsets of the bounding box. This makes up for region proposals that where slightly wrong (too wide, small, etc.).

To evaluate the performance of R-CNNs, the *mean average precision* (mAP) is used: compute the *average precision* (AP) separately for each class and then average over the classes. It is computed as follows. A detection is a true positive if the intersection over union with the true box is greater than some threshold (usually 0.5). Then to compute the AP all detections are used to draw a precision/recall-curve. The AP then is the area under the curve.

Problems of R-CNN is that it is slow at test time as a full forward pass through the CNN has to be computed for each region proposals. Also the SVMs and regressors are post-hoc, i.e., the CNN features are not updated in response to the SVMs and regressors. And the training process is really complicated.

[1] Miguel P. Eckstein, Kathryn Koehler, Lauren E. Welbourne, and Emre Akbas (2017): "Humans, but Not Deep Neural Networks, Often Miss Giant Targets in Scenes"

Fast R-CNN

Fast R-CNN is a variant of R-CNN that speeds up predictions at test time by sharing computation of convolutional layers between proposals for an image. Also, to tackle the second and third problem of R-CNNs, the whole model is just trained end-to-end with no complex training pipeline. Compared to vanilla R-CNN, Fast R-CNN is 25 times faster at test time.

Faster R-CNN

An even faster version of R-CNN is *Faster R-CNN* where the extraction of regional proposals is done after the convolutional layers using a *region proposal network* (RPN). This works by sliding a small window along the feature map that uses a small network that is used to classify between "object" and "not object" and outputs bounding boxes (which are used for finer localization in reference to the current window).

To train Faster R-CNN, the original paper proposed an ugly pipeline consisting of alternating optimization of the RPN and all other components. But since then, the community shifted to just training everything jointly using four losses: RPN classification, RPN regression, Fast R-CNN classification, and Fast R-CNN regression. Compared to vanilla R-CNN, Faster R-CNN is 250 times faster at test time.

6.2.2 YOLO: You Only Look Once

The *You Only Look Once* (YOLO) model uses a different approach than R-CNN and interprets object detection as regression. The image is divided into an $S \times S$ grid and within each cell bounding boxes with four coordinated and their respective confidence as well as C class scores are calculated. The model than directly predicts these values using a CNN onto a $7 \times 7 \times (5 \cdot B + C)$ tensor. The YOLO model is even faster than Faster R-CNN, but the prediction power is not as good.

7 Recurrent Neural Networks

Until now, all discussed neural networks all have a clear topological ordering and a sensor of forward and backward and they can only process one input at a time, producing a single output. *Recurrent neural networks* (RNNs) are a generalization of so-called *feedforward networks* to networks that have a notion of "time" by introducing cycles. They are designed for processing a sequence $\boldsymbol{x}_1, \cdots, \boldsymbol{x}_n$ of inputs and producing a sequence $\boldsymbol{y}_1, \cdots, \boldsymbol{y}_m$ of outputs. Figure 7.1 shows a single RNN cell with its hidden state $\boldsymbol{h}_t$ and how it processes a single input $\boldsymbol{x}_t$. The computed hidden state is then stored and used for the next incoming vector.

To compute a forward- and backward-pass (i.e., computing the gradients), RNNs are usually unrolled in time, forming a computational graph that can be differentiated. Figure 7.2 illustrates the unrolling for three time steps and a single RNN cell. It is also often done to stack multiple RNN layers on top of each other, i.e., multiple RNN cells, each with an own hidden state, computed after each other. This is shown in Figure 7.3 for two cells and again three time steps. The nodes on the same level share all parameters while the nodes behind each other (producing the two separate hidden states) do not share weights.

In general, the RNN formalism is extremely powerful and offers lots of flexibility. As it is a complete generalization of feedforward neural networks, they would theoretically also allow a one-to-one mapping, but this is usually not called an RNN. But they allow also for lots of other mapping types:

One-to-One This is equivalent to a vanilla neural network. A single input is mapped onto a single output. An example application is image classification.

One-to-Many A single input is mapped onto multiple outputs. An example application is image captioning where the RNN gets features extracted by a CNN and generated a sequence of words that describe the image.

Many-to-One Multiple inputs are mapped onto a single output. An example application is sentiment classification where a sequence of words is read and the RNN predicts the sentiment of the sentence (e.g. positive/negative).

Many-to-Many Multiple inputs are mapped onto multiple outputs. This mode can come in two variants: either an output for time t is generated directly after reading the input at time t or all outputs are generated after the last item has been read. Of course various mixtures and different delays may also be used. Example applications are model learning where the RNN learns to predict the next position/velocity of e.g. a pendulum (first variant) and machine translation (second variant).

These four/five variants are illustrated in Figure 7.4.

7.1 Vanilla RNN

The dynamics of the hidden state of a vanilla RNN as illustrated in Figure 7.1 is described by a parametric function

$$\boldsymbol{h}_t = \boldsymbol{f}_{\boldsymbol{\theta}}(\boldsymbol{h}_{t-1}, \boldsymbol{x}_t)$$

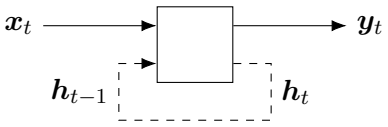
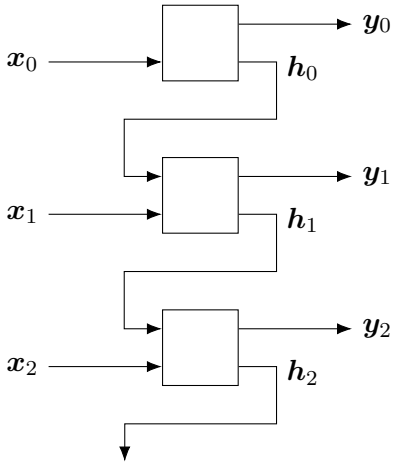
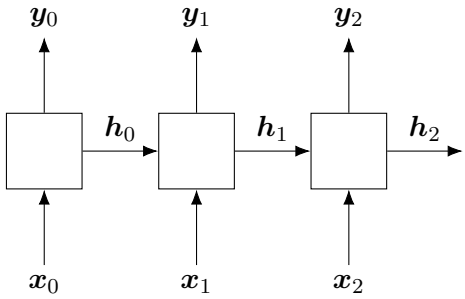


Figure 7.1: Illustration of a single RNN cell processing an input x_t, producing an output y_t and a hidden state h_t that is delayed one time step.



(a) Unrolling with analogous notation to Figure 7.1.



(b) Unrolling how it is usually drawn.

Figure 7.2: First three node of an unrolled recurrent neural network with the hidden state h_t. The two pictures show the same model using two different notations.

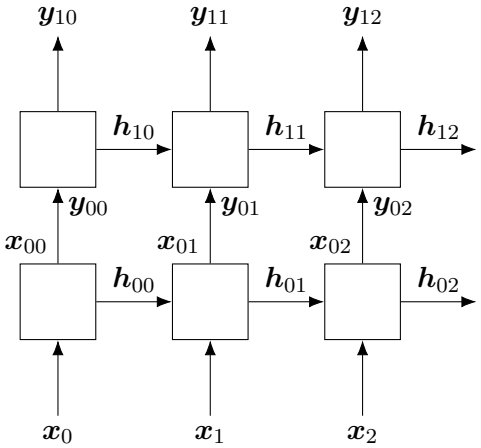


Figure 7.3: RNN with two RNN cells after each other. This picture shows the unrolled RNN for three time steps.

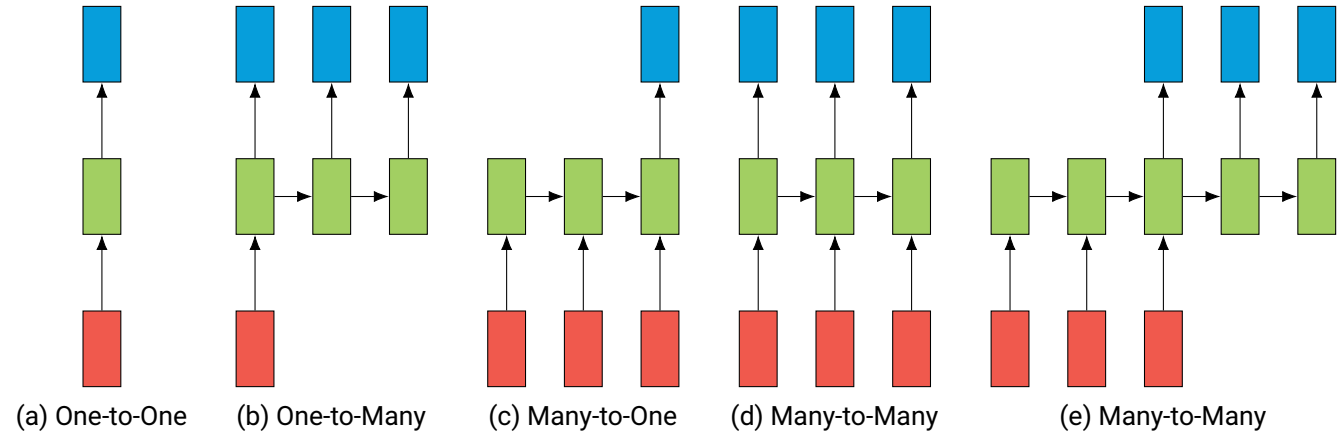


Figure 7.4: Illustration of the four/five variants of RNN applications.

where $\boldsymbol{h}_t$ and $\boldsymbol{h}_{t-1}$ are the next and current time step, respectively, and $\boldsymbol{x}_t$ is the current input. This formula is then iterated for every step of the input sequence (with the same function and parameters per step). Usually the function $\boldsymbol{f}_{\boldsymbol{\theta}}$ is a tanh with linear combinations of the hidden state and input as parameters

$$\boldsymbol{f}_{\boldsymbol{\theta}} = \tanh\big(\boldsymbol{W}_{hh}\boldsymbol{h}_{t-1} + \boldsymbol{W}_{xh}\boldsymbol{x}_t\big) \tag{7.1}$$

with $\boldsymbol{\theta} = \big\{\boldsymbol{W}_{hh}, \boldsymbol{W}_{xh}\big\}$. The hidden state is then re-weighted, forming the output of the RNN cell:

$$\boldsymbol{y}_t = \boldsymbol{W}_{hy}\boldsymbol{h}_t$$

This can either be taken as the output or be fed into another neural network, often a fully connected one. For one-to-many and delayed many-to-many RNNs, the inputs of the cells that do not have inputs are just set to zero such that they are ignored in Equation 7.1.

7.1.1 Image Captioning

For using RNNs in image captioning, the hidden state is usually high-dimensional and $\boldsymbol{h}_0$ is initialized with the output of a CNN. Then a "start" token is fed into the RNN as the first input and the second input is just the first output. To indicate that the captioning is complete the RNN spits out an "end" token. This is illustrated in Figure 7.5. Instead of initializing $\boldsymbol{h}_0$ it may also be helpful to learn an additional matrix that transforms the first input

$$\boldsymbol{f}_{\boldsymbol{\theta}} = \tanh\big(\boldsymbol{W}_{hh}\boldsymbol{h}_{t-1} + \boldsymbol{W}_{xh}\boldsymbol{x}_t + \boldsymbol{W}_{ih}\boldsymbol{v}\big),$$

where $\boldsymbol{v}$ is the output vector of the CNN. Fancier models for image captioning use *attention* to look at different parts of the image for different parts of the caption. This is covered in more detail in chapter 11.

7.2 Long Short-Term Memory (LSTM)

One problem with vanilla RNNs is that if the sequences are very long, gradients vanish quickly as the tangent always reduces the gradient values. Also, if the largest Eigenvalue of weight matrix for the hidden state is larger than one, gradient will explode. If it is lower than one, they will vanish. So only for a single Eigenvalue the gradients are stable. *Long short-term memory networks* tackle the problem of vanishing gradients by adding

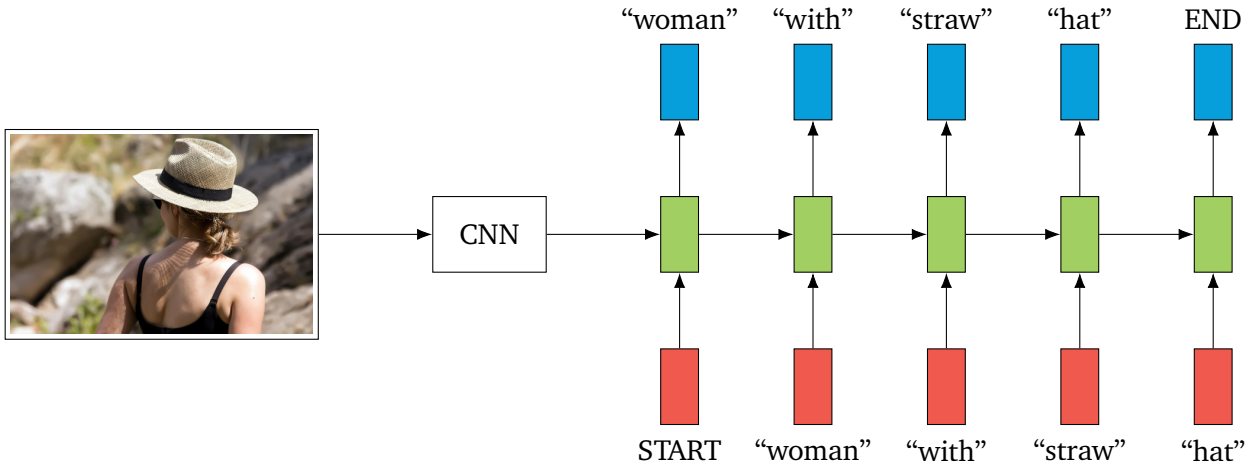


Figure 7.5: Illustration of image captioning using a CNN to initialize the hidden state.

“highway”[1] connections for the gradients. Exploding gradients can be solved by gradient clipping, so the problem is not that severe.

Assuming stacked RNNs, the equation for updating the hidden state in a vanilla RNN is[2]

$$\boldsymbol{h}_t^l = \tanh \boldsymbol{W}^l \begin{bmatrix} \boldsymbol{h}_t^{l-1} \\ \boldsymbol{h}_{t-1}^l \end{bmatrix}$$

where $\boldsymbol{h}_{t-1}^l$ is the hidden state of layer l of the previous time step, $\boldsymbol{h}_t^{l-1}$ is the hidden state of the previous layer $l-1$ of the current time step and $\boldsymbol{h}_t^l$ is the next hidden state. For the first layer, the hidden state equals the input: $\boldsymbol{h}_t^0 = \boldsymbol{x}_t$. The weight matrix $\boldsymbol{W}^l$ then summarizes the two matrices of (7.1). The vanilla RNN was previously introduced like every cell has an extra output with an extra weight matrix, but this formalism is not necessary and the hidden state can just be passed to the next layer. It will be multiplied by a learnable weight matrix anyway.

LSTMs add a *cell state* $\boldsymbol{c}_t^l \in \mathbb{R}^n$ captures the state of the cell and is not directly pushed through a sigmoidal nonlinearity. Hence, gradient vanishing is not that big of a problem. The update equations for the hidden state and the cell state are

$$\begin{bmatrix} \boldsymbol{i} \\ \boldsymbol{f} \\ \boldsymbol{o} \\ \boldsymbol{g} \end{bmatrix} = \begin{bmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{bmatrix} \boldsymbol{W}^l \begin{bmatrix} \boldsymbol{h}_t^{l-1} \\ \boldsymbol{h}_{t-1}^l \end{bmatrix} \qquad \boldsymbol{c}_t^l = \boldsymbol{f} \odot \boldsymbol{c}_{t-1}^l + \boldsymbol{i} \odot \boldsymbol{g} \qquad \boldsymbol{h}_t^l = \boldsymbol{o} \odot \tanh\left(\boldsymbol{c}_t^l\right)$$

where now $\boldsymbol{W}^l \in \mathbb{R}^{4n \times 2n}$ and $\boldsymbol{i}, \boldsymbol{f}, \boldsymbol{o}, \boldsymbol{g} \in \mathbb{R}^n$ and $\odot$ denotes the Hadamard product. These vectors have the following interpretations:

- $\boldsymbol{i}$ is the activation vector for the input gate,
- $\boldsymbol{f}$ is the activation vector for the forget gate,
- $\boldsymbol{o}$ is the activation vector for the output gate, and

[1] Not to be confused with highway networks!

[2] Let n be the dimensionality of the hidden state, then the weight matrix has size $n \times 2n$.

- $\boldsymbol{g}$ is the cell input activation.

The forget vector controls how much of the cell state will be lost in this cell, i.e., how much the network forgets. This way the network can learn to both capture short- and long-term behavior. These equations are also illustrated in Figure 7.6.

One can say that LSTMs are to RNNs what ResNets are to vanilla nets.

There are also other variants of LSTMs, for example the *gated recurrent unit* (GRU) which uses slightly different and simpler update equations for the hidden state and does not feature an additional cell state. GRUs have shown to work similarly well as LSTMs on large datasets but better on small datasets because they have less parameters.

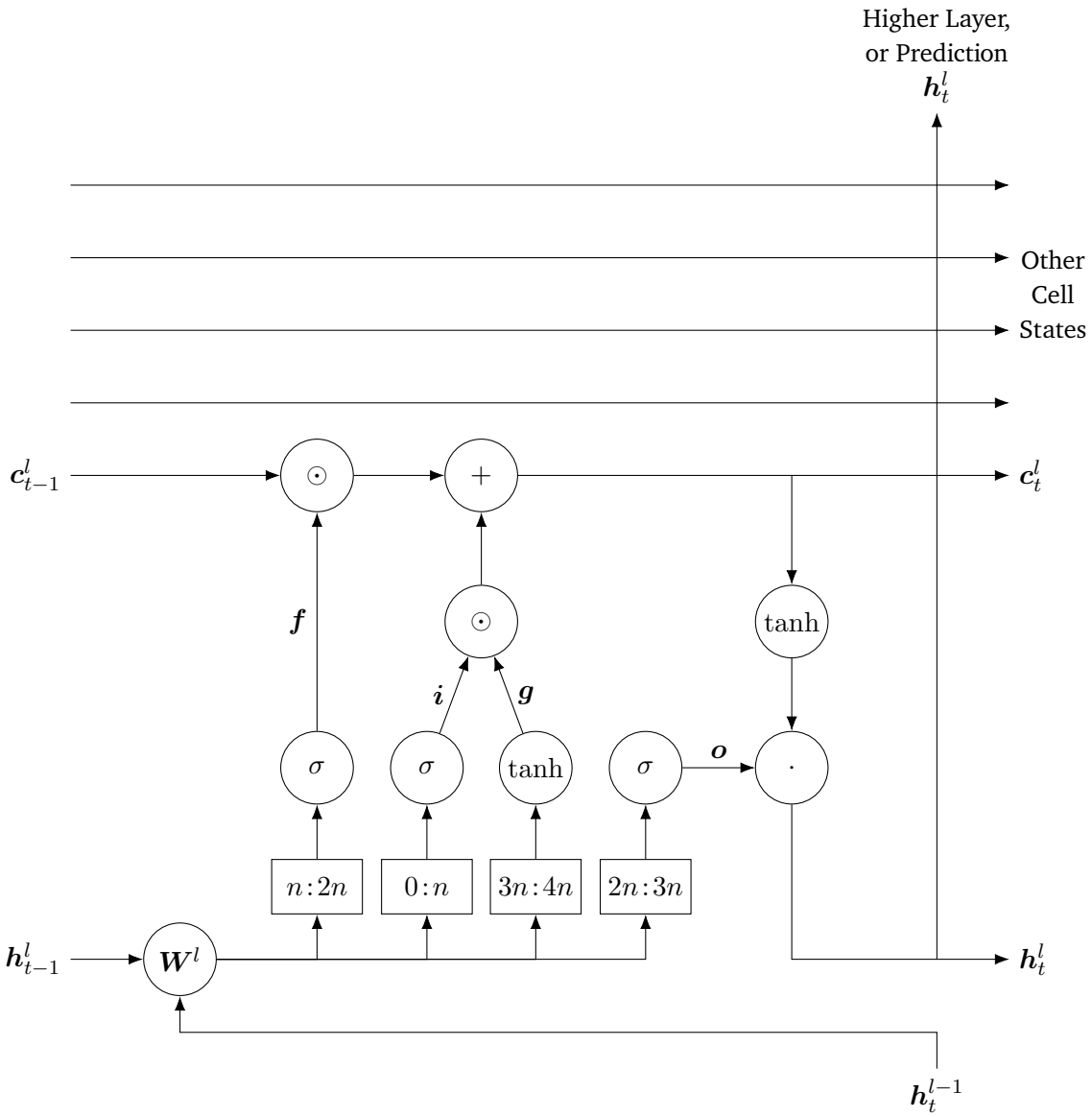


Figure 7.6: Illustration of a single LSTM cell.

8 Generative Models

In supervised learning (everything discussed) before, the training dataset contains both input data as well as labels for that data and the goal is to learn a function that maps from the former to the latter. In unsupervised ML, the training data only contains "input" data and no labels. The goal here is to learn the underlying structure in the data rather than an explicit mapping. Examples for this are clustering, dimensionality reduction, and density estimation. Hence, unsupervised learning is the "holy grail" of machine learning: being able to learn and detect structure in data without explicitly providing information on this structure would lead to understanding the world.

Generative models are a class of models that try to learn the underlying distribution of the training data rather than directly learning, e.g. discrimination rules. In other words: the goal is to learn a distribution $p_{\text{model}}(\boldsymbol{x})$ that is similar to $p_{\text{real}}(\boldsymbol{x})$. So generative models boil down to density estimation. Density estimation comes in two flavors:

Explicit The distribution $p_{\text{model}}(\boldsymbol{x})$ is explicitly defined and solved for.

Implicit A model is learned that can sample from $p_{\text{model}}(\boldsymbol{x})$ without explicitly defining it.

As the distribution is known after successful training, generative models can be used for generating data, e.g. for super-sampling or colorizing images. If time series data is learned, they can be used for simulation and planning which is especially useful in control (e.g. receding horizon control) and reinforcement learning. Also, training generative models can enable inference of latent representations which can be used as general features (e.g. in variational autoencoders, see section 8.2). Figure 8.1 shows the most known models and groups of models in a hierarchy.

8.1 A Glimpse at Other Models

This section only glimpses at some other models than the two models that are discussed primarily in this chapter.

8.1.1 Fully Visible Belief Network, PixelRNN, and PixelCNN

A *fully visible belief network* is an explicit (tractable) density model that uses the probability chain rule for composing the likelihood $p(\boldsymbol{x})$ of a data point using the scalar probability density for each element of the vector:

$$p_{\boldsymbol{\theta}}(\boldsymbol{x}) = p_{\boldsymbol{\theta}}(x_1) \prod_{i=2}^{n} p_{\boldsymbol{\theta}}(x_i \,|\, x_1, \cdots, x_{i-1}) \tag{8.1}$$

For training, the likelihood is maximized.

If $\boldsymbol{x}$ is an image, then the individual values x_i are the pixels of that image. To express the likelihood like in (8.1), an order of pixels in an image has to be defined. In *PixelRNN*, images are generated starting from a corner and the order of pixels is defined from that corner to the opposite corner (the dependencies are

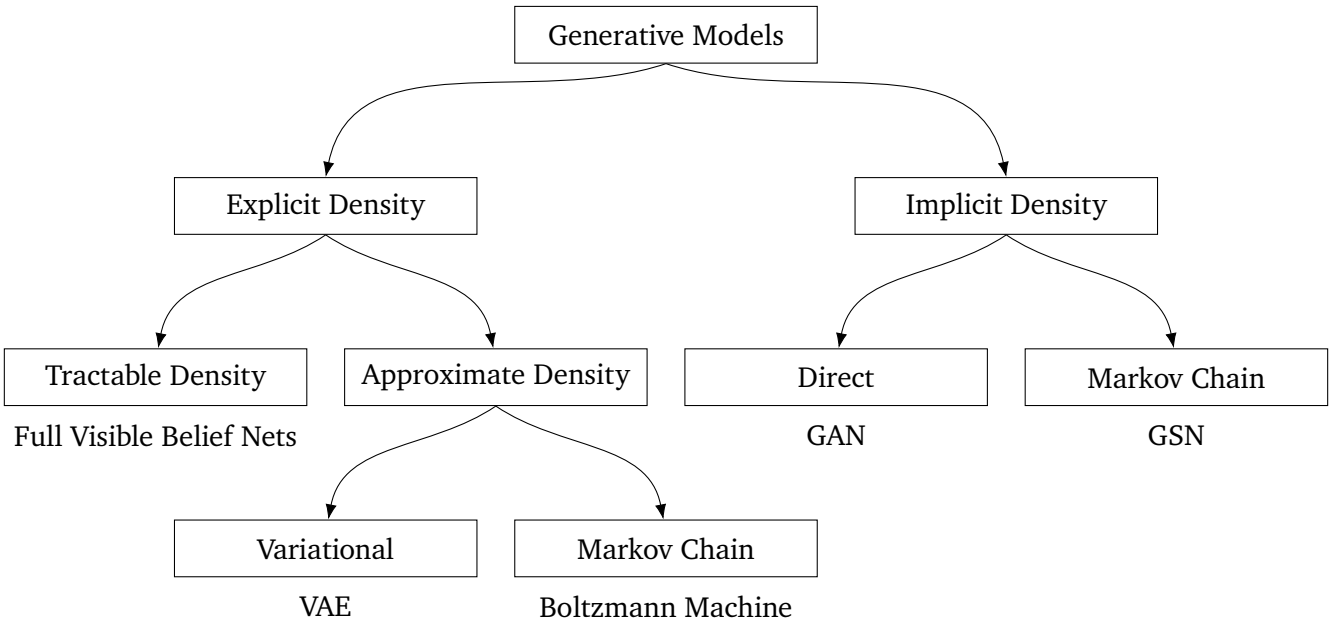


Figure 8.1: Taxonomy of generative models. The text below the leaves name some examples models in the respective category (where VAE stands for Variational Autoencoder, GAN for Generative Adversarial Network, and GSN for Generative Stochastic Network).

illustrated in Figure 8.2). Then, but iteratively sampling from the distributions, a complete image can be generated. The big disadvantage of this approach is that it extremely slow. *PixelCNN* is similar to PixelRNN in the sense that image generation still starts from the corner, but the dependency on previous pixels is now modeled using a CNN. Thus, training is faster then for PixelRNN as convolutions can be computed in paralleled, but generation is still sequential and slow.

Advantages of fully visible belief networks are that the likelihood can be computed explicitly which also gives a good evaluation metric. Also, the generated samples (e.g. of images) are good. But the big disadvantage is that they are extremely slow for high-dimensional data like images as sampling has to be done sequentially. The performance of PixelCNN can for example be improved by using gated convolutional layers, shortcut connections, discretized logistic loss, multi-scale, training tricks, and a lot more. One advancement of PixelCNN is PixelCNN++ (Salimans et al., 2017).

8.1.2 Change of Variables

Instead of sampling from the actual distribution $p(\boldsymbol{x})$, it is possible to learn a mapping $\boldsymbol{y} = \boldsymbol{g}(\boldsymbol{x})$ such that

$$p_{\boldsymbol{x}}(\boldsymbol{x}) = p_{\boldsymbol{y}}\big(\boldsymbol{g}(\boldsymbol{x})\big)\left|\det\left(\frac{\partial g(\boldsymbol{x})}{\partial \boldsymbol{x}}\right)\right|.$$

Then sampling can be done from the distribution $p_{\boldsymbol{y}}(\boldsymbol{y})$ instead and the results can be mapped back to the visible space.

Disadvantages are that the transformation $\boldsymbol{g}(\cdot)$ must be invertible and that the latent dimension must match the real dimension.

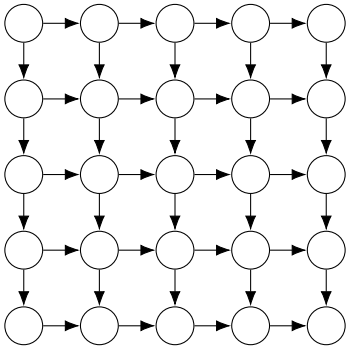


Figure 8.2: Dependencies of pixels as assumed by a PixelRNN. Generation starts from the top left and propagates through the whole image.

8.1.3 Boltzmann Machine

A *Boltzmann machine* defines the likelihood as

$$p(\boldsymbol{x}) = \frac{1}{Z} \exp\{-E(\boldsymbol{x}, \boldsymbol{z})\} \qquad Z = \sum_{\boldsymbol{x}} \sum_{\boldsymbol{z}} \exp\{-E(\boldsymbol{x}, \boldsymbol{z})\}$$

As this is intractable, sampling requires Markov chain Monte Carlo methods.

8.2 Variational Auto-Encoder (VAE)

While fully visible belief networks define a tractable density function $p_{\boldsymbol{\theta}}(\boldsymbol{x}) = p_{\boldsymbol{\theta}}(x_1) \prod_{i=2}^{n} p_{\boldsymbol{\theta}}(x_i \mid x_1, \cdots, x_{i-1})$ that can be used to optimize the likelihood of the data, *variational auto-encoders* (VAEs) define an intractable density function

$$p_{\boldsymbol{\theta}}(\boldsymbol{x}) = \int p_{\boldsymbol{\theta}}(\boldsymbol{z})\, p_{\boldsymbol{\theta}}(\boldsymbol{x} \mid \boldsymbol{z})\, \mathrm{d}\boldsymbol{z}$$

using a latent representation $\boldsymbol{z}$. This can not be used to directly optimize the likelihood. Instead, a lower bound is derived and optimized instead.

This section will first introduce non-variational (classical) auto-encoders and subsequently lay down the definition of the VAE model, the derivation of the lower bound, and how to train and sample from the model

8.2.1 Auto-Encoder

An *auto-encoder* is composed of two parts (usually neural networks): the *encoder* and the *decoder*. The encoder is used to compute a low-dimensional[1] feature representation $\boldsymbol{z}$ from the original data. This feature representation is then passed to the decoder which reconstructs the original data. The latent dimension can then be used in various different ways: either to pass it to another model and exploit that the latent dimension is low-dimensional and has to contain meaningful information about the original data or to actually compress and transmit data. To achieve this, the encoder and decoder are trained jointly using an L2 loss between the input data and the reconstructed data. The boundary between the encoder and decoder is usually called the *bottleneck* of the auto-encoder as it forces all data to be compressed into this dimensionality. Figure 8.3 is an illustration of this concept.

[1] Usually at least lower-dimensional than the original data.

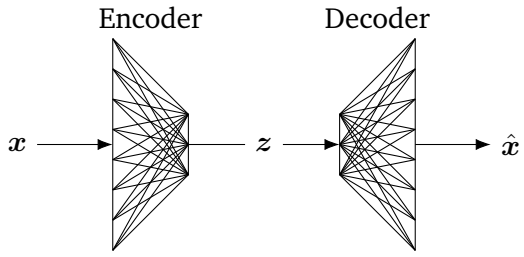


Figure 8.3: Illustration of a classical auto-encoder.

To generate new data from a classical auto-encoder, it is possible to sample latent vectors—but this does not work well. This is due to the vectors in the latent dimension being unrestricted at how the scatter across the whole space. Also it is unclear on how to actually sample the vectors because no distribution is defined within the latent space.

8.2.2 Model and Evidence Lower Bound

As already seen in the previous section, auto-encoders work well in finding a low-dimensional embedding that explains the data and provides nonlinear mappings into and out of this embedding. A variational autoencoder is now an extension of classical auto-encoders that have a probabilistic interpretation and place a distribution on the latent space such that it can be sampled. Let $p_{\boldsymbol{\theta}^*}(\boldsymbol{z})$ be the true prior of the latent dimension and let $p_{\boldsymbol{\theta}^*}(\boldsymbol{x} \mid \boldsymbol{z})$ be the true conditional. Usually the prior is chosen to be simple, e.g. a Gaussian, and the conditional distribution is represented by a neural network. This is called the decoder network and serves a similar purpose as in classical auto-encoders. To train the model, the data likelihood

$$p_{\boldsymbol{\theta}}(\boldsymbol{x}) = \int p_{\boldsymbol{\theta}}(\boldsymbol{z})\, p_{\boldsymbol{\theta}}(\boldsymbol{x} \mid \boldsymbol{z})\, \mathrm{d}\boldsymbol{z} \tag{8.2}$$

is maximized. But this integral is intractable! And thus the posterior density

$$p_{\boldsymbol{\theta}}(\boldsymbol{z} \mid \boldsymbol{x}) = \frac{p_{\boldsymbol{\theta}}(\boldsymbol{x} \mid \boldsymbol{z})\, p_{\boldsymbol{\theta}}(\boldsymbol{z})}{p_{\boldsymbol{\theta}}(\boldsymbol{x})}$$

is also intractable because the normalization factor is intractable. The core idea of VAEs is to define an additional encoder network $q_{\boldsymbol{\phi}}(\boldsymbol{z} \mid \boldsymbol{x})$ that approximates the true posterior $p_{\boldsymbol{\theta}}(\boldsymbol{z} \mid \boldsymbol{x})$. This allows to derive a lower bound on the data likelihood that can be optimized end-to-end. When choosing a Gaussian distributions for the latents, the encoder and decoder networks are now two-headed: they both produce the mean and covariance[2] of the Gaussian, explicitly modeling the distributions.

Deriving the Lower Bound

To derive the ELBO, there are two approaches: one uses Jensen's inequality and the other does not. This section contains both of the derivations.

[2]When implementing, it is often helpful to not let the network directly produce the covariance matrix or standard deviation, but rather the logarithm. This does not make a difference mathematically, but taking the logarithm afterwards (which will in the ELBO) would result in numerical instability as the network might produce negative numbers. Taking the logarithm implicitly circumvents this problem.

With Jensen's Inequality To derive the ELBO using Jensen's inequality, the data likelihood is first written according to (8.2), and then massaged using statistical properties:

$$\begin{aligned}
\log p_{\boldsymbol{\theta}}(\boldsymbol{x}) &= \log \int p_{\boldsymbol{\theta}}(\boldsymbol{z})\, p_{\boldsymbol{\theta}}(\boldsymbol{x} \mid \boldsymbol{z})\, \mathrm{d}\boldsymbol{z} \\
&= \log \int q_{\boldsymbol{\phi}}(\boldsymbol{z} \mid \boldsymbol{x})\, \frac{p_{\boldsymbol{\theta}}(\boldsymbol{x} \mid \boldsymbol{z})\, p_{\boldsymbol{\theta}}(\boldsymbol{z})}{q_{\boldsymbol{\phi}}(\boldsymbol{z} \mid \boldsymbol{x})}\, \mathrm{d}\boldsymbol{z} \\
&\geq \int q_{\boldsymbol{\phi}}(\boldsymbol{z} \mid \boldsymbol{x})\, \log \frac{p_{\boldsymbol{\theta}}(\boldsymbol{x} \mid \boldsymbol{z})\, p_{\boldsymbol{\theta}}(\boldsymbol{z})}{q_{\boldsymbol{\phi}}(\boldsymbol{z} \mid \boldsymbol{x})}\, \mathrm{d}\boldsymbol{z} \\
&= \int q_{\boldsymbol{\phi}}(\boldsymbol{z} \mid \boldsymbol{x})\, \log p_{\boldsymbol{\theta}}(\boldsymbol{x} \mid \boldsymbol{z})\, \mathrm{d}\boldsymbol{z} - \int q_{\boldsymbol{\phi}}(\boldsymbol{z} \mid \boldsymbol{x})\, \log \frac{q_{\boldsymbol{\phi}}(\boldsymbol{z} \mid \boldsymbol{x})}{p_{\boldsymbol{\theta}}(\boldsymbol{z})}\, \mathrm{d}\boldsymbol{z} \\
&= \mathbb{E}_{\boldsymbol{z} \sim q_{\boldsymbol{\phi}}(\cdot \mid \boldsymbol{x})}\big[\log p_{\boldsymbol{\theta}}(\boldsymbol{x} \mid \boldsymbol{z})\big] - D_{\mathrm{KL}}\big(q_{\boldsymbol{\phi}}(\boldsymbol{z} \mid \boldsymbol{x}) \,\|\, p_{\boldsymbol{\theta}}(\boldsymbol{z})\big) \\
&=: \mathcal{L}_{\mathrm{ELBO}}(\boldsymbol{x}; \boldsymbol{\theta}, \boldsymbol{\phi})
\end{aligned}$$

In the second step the integrand was multiplied by one, the third step uses Jensen's inequality, and the rest is just moving around and exploiting of properties of the logarithm.

Without Jensen's Inequality To derive the ELBO without Jensen's inequality, the data likelihood is surrounded by a noop integral which can then be transformed into the ELBO:

$$\begin{aligned}
\log p_{\boldsymbol{\theta}}(\boldsymbol{x}) &= \int q_{\boldsymbol{\phi}}(\boldsymbol{z} \mid \boldsymbol{x})\, \log p_{\boldsymbol{\theta}}(\boldsymbol{x})\, \mathrm{d}\boldsymbol{z} \\
&= \int q_{\boldsymbol{\phi}}(\boldsymbol{z} \mid \boldsymbol{x})\, \log \frac{p_{\boldsymbol{\theta}}(\boldsymbol{x} \mid \boldsymbol{z})\, p_{\boldsymbol{\theta}}(\boldsymbol{z})}{p_{\boldsymbol{\theta}}(\boldsymbol{z} \mid \boldsymbol{x})}\, \mathrm{d}\boldsymbol{z} \\
&= \int q_{\boldsymbol{\phi}}(\boldsymbol{z} \mid \boldsymbol{x})\, \log \frac{p_{\boldsymbol{\theta}}(\boldsymbol{x} \mid \boldsymbol{z})\, p_{\boldsymbol{\theta}}(\boldsymbol{z})}{p_{\boldsymbol{\theta}}(\boldsymbol{z} \mid \boldsymbol{x})} \frac{q_{\boldsymbol{\phi}}(\boldsymbol{z} \mid \boldsymbol{x})}{q_{\boldsymbol{\phi}}(\boldsymbol{z} \mid \boldsymbol{x})}\, \mathrm{d}\boldsymbol{z} \\
&= \int q_{\boldsymbol{\phi}}(\boldsymbol{z} \mid \boldsymbol{x})\, \log p_{\boldsymbol{\theta}}(\boldsymbol{x} \mid \boldsymbol{z})\, \mathrm{d}\boldsymbol{z} - \int q_{\boldsymbol{\phi}}(\boldsymbol{z} \mid \boldsymbol{x})\, \log \frac{q_{\boldsymbol{\phi}}(\boldsymbol{z} \mid \boldsymbol{x})}{p_{\boldsymbol{\theta}}(\boldsymbol{z})}\, \mathrm{d}\boldsymbol{z} + \int q_{\boldsymbol{\phi}}(\boldsymbol{z} \mid \boldsymbol{x})\, \log \frac{q_{\boldsymbol{\phi}}(\boldsymbol{z} \mid \boldsymbol{x})}{p_{\boldsymbol{\theta}}(\boldsymbol{z} \mid \boldsymbol{x})}\, \mathrm{d}\boldsymbol{z} \\
&= \mathbb{E}_{\boldsymbol{z} \sim q_{\boldsymbol{\phi}}(\cdot \mid \boldsymbol{x})}\big[\log p_{\boldsymbol{\theta}}(\boldsymbol{x} \mid \boldsymbol{z})\big] - D_{\mathrm{KL}}\big(q_{\boldsymbol{\phi}}(\boldsymbol{z} \mid \boldsymbol{x}) \,\|\, p_{\boldsymbol{\theta}}(\boldsymbol{z})\big) + \underbrace{D_{\mathrm{KL}}\big(q_{\boldsymbol{\phi}}(\boldsymbol{z} \mid \boldsymbol{x}) \,\|\, p_{\boldsymbol{\theta}}(\boldsymbol{z} \mid \boldsymbol{x})\big)}_{\geq 0} \\
&\geq \mathbb{E}_{\boldsymbol{z} \sim q_{\boldsymbol{\phi}}(\cdot \mid \boldsymbol{x})}\big[\log p_{\boldsymbol{\theta}}(\boldsymbol{x} \mid \boldsymbol{z})\big] - D_{\mathrm{KL}}\big(q_{\boldsymbol{\phi}}(\boldsymbol{z} \mid \boldsymbol{x}) \,\|\, p_{\boldsymbol{\theta}}(\boldsymbol{z})\big) \\
&=: \mathcal{L}_{\mathrm{ELBO}}(\boldsymbol{x}; \boldsymbol{\theta}, \boldsymbol{\phi})
\end{aligned}$$

The first step adds an integral over $\boldsymbol{z}$ which is valid because $p_{\boldsymbol{\theta}}(\boldsymbol{x})$ does not depend on $\boldsymbol{z}$ and is thus just a multiplicative constant w.r.t. the integral. In the second step the data likelihood is written out using Bayes' rule. The third step multiplies the argument of the logarithm with a one, not changing the overall value. Then, in the fourth step, the integral can be split into three parts by exploiting properties of the logarithm. This yields, in the fifth step, three components: a tractable expectation, a tractable KL-divergence and an intractable KL-divergence. As the latter it always non-negative it can be discarded in the sixth step, yielding the ELBO (seventh step).

While this derivation is a tad longer than the derivation using Jensen's inequality, it fives some more insights on what the ELBO is actually neglects: the KL-divergence, or distance, between the approximate posterior and the actual posterior. Thus it is possible that optimizing the ELBO does not even closely approximate the true posterior as it is not enforced. But as the first and second part of the ELBO ensure reconstruction and staying close to the prior, respectively, optimizing it usually yields good embeddings.

8.2.3 Training Procedure and Generating Data

To optimize a VAE, the ELBO has to be maximized. This is done by computing a forward pass through the network and both sampling layers and backpropagating through it. As it is not possible to differentiate a sampling procedure per se, a trick called the *reparametrization trick* is used: instead of directly sampling from the encoder distribution, samples are taken from a fixed distribution and are transformed via the sufficient statistics produced by the encoder network. For a Gaussian $q_{\boldsymbol{\phi}}(\boldsymbol{z} \mid \boldsymbol{x}) = \mathcal{N}(\boldsymbol{z} \mid \boldsymbol{\mu}_{\boldsymbol{x}}, \boldsymbol{\Sigma}_{\boldsymbol{x}})$, where the mean and covariance are produced by the encoder network, the latent variables are not sampled directly from $q_{\boldsymbol{\phi}}$, but an auxiliary variable $\boldsymbol{\epsilon}$ is sampled from a unit Gaussian. This sample is then transformed using the mean and covariance like it has been sampled from the posterior directly:

$$\boldsymbol{z} \sim \mathcal{N}(\boldsymbol{z} \mid \boldsymbol{\mu}_{\boldsymbol{x}}, \boldsymbol{\Sigma}_{\boldsymbol{x}}) \qquad \Longrightarrow \qquad \boldsymbol{z} = \boldsymbol{\mu}_{\boldsymbol{x}} + \boldsymbol{\Sigma}_{\boldsymbol{x}}^{1/2}\boldsymbol{\epsilon}, \quad \boldsymbol{\epsilon} \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{I})$$

Here $\boldsymbol{\Sigma}_{\boldsymbol{x}}^{1/2}$ is the Cholesky decomposition of the covariance matrix.

To generate samples from the VAE, it is now possible to just sample from the prior and push the result through the decoder network. When a diagonal prior was used (like it is usually done), the latent variables are independent and different dimensions of $\boldsymbol{z}$ might encode interpretable information about the data. This is also a good feature representation that can be used in subsequent models.

8.2.4 Pros and Cons

The advantages of VAEs are that they are a principled approach to generative models and allow inference of $q_{\boldsymbol{\phi}}(\boldsymbol{z} \mid \boldsymbol{x})$ which exhibits useful feature representations for other tasks. Disadvantages are that the maximization of the ELBO work reasonable well, but is not an evaluation metric as good as the density in PixelRNN/PixelCNN. Also the samples are blurrier and of lower quality compared to state-of-the-art generative models (Generative Adversarial Models, GANs).

But they are extremely flexibly and allow rich approximations of the posterior, also more than Gaussian distributions. They also incorporate structure in the latent variables that allow for interpretability.

8.3 Generative Adversarial Networks

Generative adversarial networks (GANs) take a step back and give up explicitly modeling the density $p(\boldsymbol{x})$ in the favor of specializing on generating data. This is done by training two networks jointly: the generator network $G_{\boldsymbol{\theta}_g(\boldsymbol{z})}$ which has the only requirement of being differentiable—especially not necessarily invertible—and the discriminator network $D_{\boldsymbol{\theta}_d}(\boldsymbol{x})$ which must also be differentiable. The generator network is used for transforming the *noise vector* $\boldsymbol{z}$, which is usually sampled from a simple distribution, into output data $\boldsymbol{x}$ (often these are images). Hence, it "generates" images. For some theoretical guarantees the noise vector must be of higher dimension than the image dimensionality, but this is not a necessity. The discriminator network then tries to distinguish real and fake images by assigning each input data a probability of it being real.

The next sections will cover the training process and the model in more detail and glimpse at some related network architectures.

8.3.1 Training Procedure

Training a GAN is done using a gradient descent variant like Adam on two batches of data simultaneously: the first batch are real images, the training examples, and the second batch are fake images generated by the

generator network. Both networks are then trained simultaneously in a minimax game

$$\min_{\boldsymbol{\theta}_g} \max_{\boldsymbol{\theta}_d} \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}(\cdot)}\big[\log D_{\boldsymbol{\theta}_d}(\boldsymbol{x})\big] + \mathbb{E}_{\boldsymbol{z}\sim p(\boldsymbol{z})}\Big[\log\Big(1 - D_{\boldsymbol{\theta}_d}\big(G_{\boldsymbol{\theta}_g}(\boldsymbol{z})\big)\Big)\Big]$$

where the two expectations are (as usual) approximated using minibatches of data. The first expectation describes the log-likelihood that the discriminator classifies the real images correctly and the second expectation described the log-likelihood that the discriminator classifies the fake images correctly. As the discriminator maximizes and the generator minimizes the objective, the goal of the former is to correctly classify all data while the goal of the latter is to fool the discriminator. To optimize this objective, the training process alternates between gradient ascent on the discriminator and gradient descent on the generator parameters.

The problem of this approach is that training the generator does not work well. This is because the loss in the regions where a sample is likely fake, i.e., not good enough, is relatively flat compared to regions where samples are already good. Hence, the gradient vanishes and no learning happens. This is due to the minimax game having an equilibrium at the saddle point of the loss. Do circumvent this, the training process must not alternate between minimizing and maximizing the same loss w.r.t. different parameters, but instead perform alternating gradient ascent on two different objectives:

$$\begin{aligned}
&\max_{\boldsymbol{\theta}_d} \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}(\cdot)}\big[\log D_{\boldsymbol{\theta}_d}(\boldsymbol{x})\big] + \mathbb{E}_{\boldsymbol{z}\sim p(\boldsymbol{z})}\Big[\log\Big(1 - D_{\boldsymbol{\theta}_d}\big(G_{\boldsymbol{\theta}_g}(\boldsymbol{z})\big)\Big)\Big] \\
&\max_{\boldsymbol{\theta}_g} \mathbb{E}_{\boldsymbol{z}\sim p(\boldsymbol{z})}\big[\log D_{\boldsymbol{\theta}_d}\big(G_{\boldsymbol{\theta}_g}(\boldsymbol{z})\big)\big]
\end{aligned}$$

This is essentially the same, but now the equilibrium is no longer describable with a single loss and the minimax game is non-saturating. But this is also heuristically motivated! But it has the advantage that the generator can also learn when the discriminator successfully rejects all generator samples, making it kind of robust towards a too good discriminator.

It is also possible to view GANs as cooperative rather than adversarial games by interpreting the output of the discriminator as an estimate of the data and model distributions. The discriminator then "informs" the generator over its estimate such that the generate can improve itself. This is based on the fact the optimal discriminator distributions is

$$D(\boldsymbol{x}) = \frac{p_{\text{data}}(\boldsymbol{x})}{p_{\text{data}}(\boldsymbol{x}) + p_{\text{model}}(\boldsymbol{x})}$$

and the generator tries to estimate the real distribution: $p_{\text{model}}(\boldsymbol{x}) \approx p_{\text{data}}(\boldsymbol{x})$. If the generator would be optimal, $p_{\text{model}}(\boldsymbol{x}) = p_{\text{data}}(\boldsymbol{x})$, even the optimal discriminator would output $D(\boldsymbol{x}) = 1/2$, which is equivalent to random guess.

Two remaining problems of GANs are that optimizing the objective is still unstable which lies in the nature oj jointly training two networks. This is an open area of research! Another problem is the *mode collapse*: fully training the discriminator while the generator is held constant is safe and probably yields a good discriminator. But fully training the generator while the discriminator is fixed results in a generator that maps all inputs to the argmax of the discriminator, overfitting on the current discriminator and exploiting its weaknesses rather than learning to generate good samples. This can partially be fixed by adding nearest-neighbor features to the discriminator that are constructed from the current minibatch (*minibatch GAN*).

8.3.2 Convolutional Architectures

Convolutional GANs use super-sampling in the generator with convolutions using a fractional stride, i.e., the stride is a fraction and thus pixels are used multiple times, resulting in a activation output volume, and a vanilla CNN architecture for the discriminator. Some more tips for designing CNN GANs are:

- Replace pooling layers with convolutions with stride in the discriminator and with convolutions with fractional stride in the generator.
- Use batch normalization in both networks.
- Remove fully connected layers for deeper architectures.
- Use ReLU activations for every layer in the generator except for the output which uses tanh.
- Use leaky ReLU activations for every layer in the discriminator.

Using CNN architectures as GANs results in high-definition images that look extremely realistic.

8.3.3 Vector Space Arithmetic

Basic arithmetic on the noise vectors $\boldsymbol{z}$ can yield interesting interpretable results. If, for instance, a noise vector $\boldsymbol{z}_{\text{woman}}$ generates a woman, $\boldsymbol{z}_{\text{man}}$ generates a man without glasses, and $\boldsymbol{z}_{\text{glasses}}$ generates a man with glasses, then $\boldsymbol{z}_{\text{glasses}} - \boldsymbol{z}_{\text{man}} + \boldsymbol{z}_{\text{woman}}$ results in a woman with glasses. To use these properties it is of course necessary to get the corresponding noise vectors at first. But when using sufficiently low-dimensional noise vectors, finding the correct ones should not be too hard.

8.3.4 Pros and Cons

A big advantage of GANs is that they generate beautiful, state-of-the-art samples and images. The downsides on the other hand are that they are a lot trickier to train compared to, for example, VAEs and that they can not solve inference queries such as $p(\boldsymbol{x})$ and $p(\boldsymbol{z} \mid \boldsymbol{x})$ as GANs do not model the distributions explicitly.

Active areas of research regarding GANs are better loss functions that induce a more stable training (Wasserstein GAN, LSGAN, etc.), Conditional GANs for training multi-purpose GANs, and many more.

8.4 Optimization and Game Theory

Optimization and game theory are two topics that should not be viewed independently and have lots of links between them. In optimization, the goal is to find a set of parameters $\boldsymbol{\theta}^*$ that are optimal w.r.t. some loss $J(\boldsymbol{\theta})$:

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \; J(\boldsymbol{\theta})$$

In game theory (or, more specifically, a two-player game), there ate two sets of parameters: $\boldsymbol{\theta}_1$ and $\boldsymbol{\theta}_2$. The first player controls $\boldsymbol{\theta}_1$ and tries to minimize its objective $J_1(\boldsymbol{\theta}_1, \boldsymbol{\theta}_2)$ and the second players controls $\boldsymbol{\theta}_2$ and tries to minimize its objective $J_2(\boldsymbol{\theta}_1, \boldsymbol{\theta}_2)$. Depending on these losses, the game is either cooperative or non-cooperative. Optimization can therefore be viewed as a subset of the more general two-players games by setting $\boldsymbol{\theta}_1 = \boldsymbol{\theta}$, $\boldsymbol{\theta}_2 = \emptyset$, $J_1(\boldsymbol{\theta}_1, \boldsymbol{\theta}_2) = J(\boldsymbol{\theta}_1)$, and $J_2(\boldsymbol{\theta}_1, \boldsymbol{\theta}_2) = 0$.

A solution of a non-cooperative is characterized by its *Nash equilibrium*. In a Nash equilibrium $\boldsymbol{\theta}_1^*$, $\boldsymbol{\theta}_2^*$, neither player can reduce its loss by changing their own strategy/parameters:

$$\forall \boldsymbol{\theta}_1 : J_1(\boldsymbol{\theta}_1, \boldsymbol{\theta}_2^*) \geq J_1(\boldsymbol{\theta}_1^*, \boldsymbol{\theta}_2^*)$$
$$\forall \boldsymbol{\theta}_2 : J_2(\boldsymbol{\theta}_1^*, \boldsymbol{\theta}_2) \geq J_2(\boldsymbol{\theta}_1^*, \boldsymbol{\theta}_2^*)$$

In other words: each player's loss is minimal with respect to the other player's strategy. Finding Nash equilibria can thus be seen again as a subset of optimization as a Nash equilibrium always characterized the global optimum.

Some well-studies cases for games are finite minimax games (zero-sum games), finite mixed-strategy games, continuous and convex games, and differential games. In continuous games, the Nash equilibrium is a saddle point of the losses. More specifically, it has to be a maximum for one and a minimum for the other player. A necessary condition for an optima are—as usual—that the gradient of the loss vanishes and that the Hessian is positive semi-definite. A sufficient condition is that the Hessian is positive definite.

This yields sufficient conditions for gradient descent to converge for both players. Given the joint gradient

$$\boldsymbol{\omega} = \begin{bmatrix} \boldsymbol{\nabla}_{\boldsymbol{\theta}_1} J_1(\boldsymbol{\theta}_1, \boldsymbol{\theta}_2) \\ \boldsymbol{\nabla}_{\boldsymbol{\theta}_2} J_2(\boldsymbol{\theta}_1, \boldsymbol{\theta}_2) \end{bmatrix}$$

the Eigenvalues of the *generalized Hessian*

$$\boldsymbol{\nabla}_{\boldsymbol{\theta}_1, \boldsymbol{\theta}_2} \boldsymbol{\omega} = \begin{bmatrix} \boldsymbol{\nabla}_{\boldsymbol{\theta}_1} \boldsymbol{\nabla}_{\boldsymbol{\theta}_1} J_1(\boldsymbol{\theta}_1, \boldsymbol{\theta}_2) & \boldsymbol{\nabla}_{\boldsymbol{\theta}_1} \boldsymbol{\nabla}_{\boldsymbol{\theta}_2} J_2(\boldsymbol{\theta}_1, \boldsymbol{\theta}_2) \\ \boldsymbol{\nabla}_{\boldsymbol{\theta}_2} \boldsymbol{\nabla}_{\boldsymbol{\theta}_1} J_1(\boldsymbol{\theta}_1, \boldsymbol{\theta}_2) & \boldsymbol{\nabla}_{\boldsymbol{\theta}_2} \boldsymbol{\nabla}_{\boldsymbol{\theta}_2} J_2(\boldsymbol{\theta}_1, \boldsymbol{\theta}_2) \end{bmatrix}$$

must have a positive real part. This condition can be interpreted as each player's Hessian should have large, positive, Eigenvalues, expressing a strong urge to keep their strategy. The Jacobian of the gradient of one player w.r.t. the other player's strategy should have a small effect on the Eigenvalues, meaning that the change of strategy of the player has only a limited ability to change the other player's behavior. But sadly this does not apply to GANs, so their convergence still is an open question.

Some heuristics for finding equilibria are to keep the parameter values close to their running average and periodically assign the running average to the parameters. Also add a loss for deviation from the running average.

Other games in artificial intelligence are for example robust optimization and robust control, domain-adversarial learning for domain adaption, adversarial privacy, guided cost learning, and predictability minimization.

9 Probabilistic Graphical Models

This chapter introduces *probabilistic graphical models* and *tractable inference*. The core if inference is to answer given queries q, for example "What is the probability that today is a Monday and there is a traffic jam on Herzl Str.?" Instead of fitting a predictive model, inference takes the step of formulating these kinds of queries statistically over a given model m of the world. The said query could be expressed as the probability $q(m) = p_m(\text{Day} = \text{Monday}, \text{Jam}_{\text{Herzl}} = 1)$. Other types of queries are also possible and different types of these will be discussed in subsection 9.1.1.

For a given family $\mathcal{M}$ of models, a class of queries $\mathcal{Q}$ is called *tractable on* $\mathcal{M}$ iff for any query $q \in \mathcal{Q}$ and model $m \in \mathcal{M}$, computing $q(m)$ has time complexity $\mathcal{O}\big(\text{poly}(|m|\,|q|)\big)$ where $\text{poly}(\cdot)$ is any polynomial but is in practice often linear. Also, if $\mathcal{M}$ and $\mathcal{Q}$ are compact in the number of random variables $\boldsymbol{X}$, i.e. $|m|, |q| \in \mathcal{O}\big(\text{poly}(|\boldsymbol{X}|)\big)$, then the query time is in $\mathcal{O}\big(\text{poly}(|\boldsymbol{X}|)\big)$.

It is also possible to do *approximate* inference instead of exact inference, but this brings some caveats: firstly, if it is possible to to exact inference, why do approximate inference? This can be answered in the context of tractability vs. expressiveness which is discussed in **??**. Also, approximate inference can be intractable as well—but sometimes it comes with certain guarantees. But even those guarantees might not be enough and can mislead learners when the approximation is incorrect. It is also possible to do approximate inference by using exact inference in an approximate model. This is something one often has to resort to as the model is usually not exact.

The next content of this chapter will be the different query types that are possible as well as certain concrete models and their pros and cons in terms of tractability and expressiveness. Afterwards probabilistic circuits and how to build them is covered and some applications are presented.

9.1 Tractability vs. Expressiveness

The core problem in inference is that more tractable queries and models are not very expressive whereas expressive models are usually not very tractable. This is known as the tractability vs. expressiveness tradeoff. To understand more what this means, the next section introduces the different variants of queries that can occur and the section after that highlights some models and their respective complexity on the queries.

9.1.1 Inference and Queries

As already said, this section will cover different query types and some examples of what such a query might encode.

Complete Evidence Queries (EVI)

Complete evidence (EVI) queries ask for the probability of all random variables having specific values. This comes down to evaluating the complete joint probability. An example natural language query is

> "What is the probability that today is a Monday at 12pm and there is a traffic jam only on Herzl Str.?"

where all random variables are the day, the time, and whether there is a traffic jam for every street. The formalized version is

$$q(m) = p_m(\text{Day} = \text{Monday}, \text{Time} = 12\text{pm}, \text{Jam}_{\text{Herzl}} = 1, \text{Jam}_{\text{Str2}} = 0, \cdots, \text{Jam}_{\text{Str}N} = 0)$$

and the general version is

$$p_m(\boldsymbol{Q} = \boldsymbol{q}).$$

This kind of query is essential in maximum likelihood learning where the joint distribution is computed for every data point.

Marginal Queries (MAR)

Marginal (MAR) queries ask for the probability of some random variables having specific values. This comes down to evaluating the marginal probability of the said random variables. An example natural language query is

> "What is the probability that today is a Monday ~~at 12pm~~ and there is a traffic jam ~~only~~ on Herzl Str.?"

where the time and other streets are marginalized out. The formalized version is

$$q(m) = p_m(\text{Day} = \text{Monday}, \text{Jam}_{\text{Herzl}} = 1)$$

and the general version is

$$p_m(\boldsymbol{Q} = \boldsymbol{q}) = \int p_m(\boldsymbol{Q} = \boldsymbol{q}, \boldsymbol{H} = \boldsymbol{h}) \, \mathrm{d}\boldsymbol{h}$$

with $\boldsymbol{Q} \subseteq \boldsymbol{X}$ and $\boldsymbol{H} = \boldsymbol{X} \setminus \boldsymbol{Q}$.

Conditional Queries (CON)

Conditional (CON) queries ask for the probability of some random variables having specific values given that other random variables are known to have specific values. This comes down to evaluating the conditional probability of the said setting. An example natural language query is

> "What is the probability that these is a traffic jam on Herzl Str. *given that* today is a Monday?"

where the day is the given random variable. The formalized version is

$$q(m) = p_m(\text{Jam}_{\text{Herzl}} = 1 \,|\, \text{Day} = \text{Monday})$$

and the general version is

$$p_m(\boldsymbol{Q} = \boldsymbol{q} \,|\, \boldsymbol{E} = \boldsymbol{e}) = \frac{p_m(\boldsymbol{Q} = \boldsymbol{q}, \boldsymbol{E} = \boldsymbol{e})}{p_m(\boldsymbol{E} = \boldsymbol{e})}$$

with $\boldsymbol{Q} \subseteq \boldsymbol{X}$, $\boldsymbol{E} \subseteq \boldsymbol{X}$, and $\boldsymbol{Q} \cap \boldsymbol{E} = \emptyset$. As conditional probabilities can be purely expressed using marginals, the ability to answer CON queries follows from the ability of answering MAR queries.

Maximum A-Posteriori (MAP)

Maximum a-posteriori (MAP) or *most probable explanation* (MPE) queries ask for the values of some random variables that are most likely given other random variables are known to have specific values. The combination of variables in question and given variables has to be exhaustive. This comes down to evaluating the configuration of the random variables in question that maximizes the posterior probability. An example natural language query is

"Which combination of streets is most likely to be jammed on Monday at 12pm?"

where the configuration of streets are the random variables that is asked for and day and time are the given random variables. The formalized version is

$$q(m) = \arg \max_{j_1, \cdots, j_N} \; p_m(\text{Jam}_{\text{Str}1} = j_1, \cdots, \text{Jam}_{\text{Str}N} = j_N \,|\, \text{Day} = \text{Monday}, \text{Time} = 12\text{pm})$$

and the generalized version is

$$\arg\max_{\boldsymbol{q}} \; p_m(\boldsymbol{Q} = \boldsymbol{q} \,|\, \boldsymbol{E} = \boldsymbol{e})$$

with $\boldsymbol{Q} \subseteq \boldsymbol{X}$, $\boldsymbol{E} \subseteq \boldsymbol{e}$, $\boldsymbol{Q} \cap \boldsymbol{E} = \emptyset$, and $\boldsymbol{Q} \cup \boldsymbol{E} = \boldsymbol{X}$.

These kinds of queries are intractable for latent variable models!

Marginal MAP (MMAP)

Marginal maximum a-posteriori (MMAP) queries ask for the values of some random variables that are most likely given other random variables are known to have specific values. The combination of variables in question and given variables dows not have to be exhaustive. This comes down to evaluating the configuration of the random variables in question that maximizes the marginal posterior probability. An example natural language query is

"Which combination of streets is most likely to be jammed ~~on Monday~~ at 12pm?"

where the configuration of streets are the random variables that is asked for and the day is marginalized out. The formalized version is

$$q(m) = \arg \max_{j_1, \cdots, j_N} \; p_m(\text{Jam}_{\text{Str}1} = j_1, \cdots, \text{Jam}_{\text{Str}N} = j_N \,|\, \text{Time} = 12\text{pm})$$

and the generalized version is

$$\arg\max_{\boldsymbol{q}} \; p_m(\boldsymbol{Q} = \boldsymbol{q} \,|\, \boldsymbol{E} = \boldsymbol{e}) = \arg\max_{\boldsymbol{q}} \; \sum_{\boldsymbol{h}} p_m(\boldsymbol{Q} = \boldsymbol{q}, \boldsymbol{H} = \boldsymbol{h} \,|\, \boldsymbol{E} = \boldsymbol{e})$$

with $\boldsymbol{Q} \cup \boldsymbol{H} \cup \boldsymbol{E} = \boldsymbol{X}$.

These kinds of queries are NP^{PP}-complete, NP-hard for trees, and even NP-hard for naive Bayes!

Advanced Queries

More advanced queries might combine several of the ones introduced above, for example

"Which day is most likely to have a traffic jam on my route to work?"

and

"What is the probability of seeing more traffic jams in Jaffa than Marina?"

The former needs marginals and maximum a-posteriori and logical events and can be formalized as

$$q(m) = \arg\max_{\text{day}} \; p_m\Big(\text{Day} = \text{day}, \bigvee\nolimits_{i \in \text{Route}} \text{Jam}_{\text{Str}i} = 1\Big)$$

and the latter needs counts and group comparisons. These kinds of queries can get very complex very fast.

9.1.2 Models

This section will introduce some models in different levels of detail and relate them to the different query types introduces beforehand in terms of tractability and expressiveness.

GANs and VAEs

GANs are powerful generative models that can generate good-looking samples, but they do not serve well for inference. That is due to the lack of an explicit likelihood and hence EVI is not tractable. Also the training is unstable due to the mode collapse and lots of samples are needed for effective Monte Carlo inference.

Similar goes for VAEs. They serve an explicit likelihood model, but computing $\log p_{\boldsymbol{\theta}}(\boldsymbol{x})$ is intractable as it is an infinite and uncountable mixture of distributions. Hence, EVI is not tractable. Also, the ELBO has to be optimized instead of the real data likelihood which is "broken"[1][2].

Mixture Models

Mixture models are built by combining multiple simpler models

$$p(\boldsymbol{X}) = w_1 p_1(\boldsymbol{X}) + w_2 p_2(\boldsymbol{X}),$$

e.g. Gaussians. They can be seen as a marginalization over a categorical latent variable Z that describes the amount one of the distributions participates to the mixture model. This increases the expressiveness of the simpler distributions.

In fact, a mixture of Gaussians can represent any distribution, but this would need lots of Gaussians. This is known as *expressive efficiency*, settings the expressiveness of a model in relation to its complexity/size.

Probabilistic Graphical Models: Markov and Bayes Networks

Probabilistic graphical models are—in a nutshell—a graphical notation for conditional independence assumptions and a compact representation for probability distributions. They utilize graphs where the nodes represent random variables and the edges represent dependencies between them. They divide into three main types: *Markov networks*, *Bayesian networks*, and *factor graphs*. Markov networks and Bayesian networks are closely related with the difference that the edges in Bayesian networks are directed while they are not in Markov networks. Factor graphs are a more general form of PGMs than can represent all kinds of relations, including everything that can be represented using a Markov or Bayesian network.

In general, exact inference for MAR and CON queries in PGMs is #P-complete. Also approximate inference of MAR and CON queries with a relative error of $2^{n^{1-\epsilon}}$ for any fixed ϵ is NP-hard. The complexity can be

[1] Alexander Alemi, Ben Poole, Ian Fischer, Joshua Dillon, Rif A. Saurous, and Kevin Murphy (2018): "Fixing a Broken ELBO"
[2] Bin Dai and David Wipf (2019): "Diagnosing and Enhancing VAE Models"

further refined using the *treewidth* of a PGM. The treewidth w described how tree-like a graphical model m is. Formally, it is the minimum width of any tree-decomposition of m. With that number, the time complexity of MAR and CON queries on a graphical model with treewidth w is $\mathcal{O}(2^w\,|\boldsymbol{X}|)$, which is linear for a fixed w. One idea is to bound the tree-width by design as inference is possible in practice for $w \approx 20$. The extreme form are trees which are covered after Bayesian networks as they are a special form of them.

Fully Factorized Models *Fully factorized models* are PGMs with no connections between the random variables. That is, each random variable is independent from all others and EVI, MAR, MAP, and MMAP inference is linear in time. But these models are definitely not expressive.

Bayesian Networks and Variable Elimination Bayesian networks are digraphs in which every node is a random variable and all parents of that not are dependencies. Figure 9.1 shows an example of a Bayesian network that describes the joint distribution

$$P(A, B, C, D, E) = P(A)\,P(B)\,P(C \mid A, B)\,P(D \mid B)\,P(E \mid C).$$

In general, the joint distributions for a Bayesian network with the random variables $X_1, \cdots, X_N$ is described by

$$P(X_1, \cdots, X_N) = \prod_{i=1}^{N} P\big(X_i \,\big|\, \mathrm{Pa}(X_i)\big)$$

where $\mathrm{Pa}(X_i)$ is the set of parents of the random variable X_i. In the example shown in Figure 9.1, this would, for example, be $\mathrm{Pa}(C) = \{A, B\}$. The local Markov assumption in Bayesian networks is that each random variable is independent from its non-descendants given its parents. As the joint distribution can be computed efficiently, EVI queries are tractable!

To evaluate MAR queries, an algorithm called *variable elimination* can be used. It is based on the observation that in the general form of a marginalization,

$$P(\boldsymbol{Q}) = \sum_{X_j \notin \boldsymbol{Q}}^{\text{marg}} \prod_{i=1}^{N} P\big(X_i \,\big|\, \mathrm{Pa}(X_i)\big),$$

where the first sum described a marginalization on its summation index, i.e., a summation of all possible realizations of X_j, the product and the sum can be exchanged under certain circumstances. That is, if all of the factors are independent of a variable X_k, these factors can be pulled before the sum:

$$\begin{aligned} P(\boldsymbol{Q}) &= \sum_{X_j \notin \boldsymbol{Q}}^{\text{marg}} \prod_{i=1}^{N} P\big(X_i \,\big|\, \mathrm{Pa}(X_i)\big) \\ &= \prod_{\substack{i=1 \\ X_k \notin \{X_i\} \cup \mathrm{Pa}(X_i)}}^{N} P\big(X_i \,\big|\, \mathrm{Pa}(X_i)\big) \sum_{X_j \notin \boldsymbol{Q}}^{\text{marg}} \prod_{\substack{i=1 \\ X_k \in \{X_i\} \cup \mathrm{Pa}(X_i)}}^{N} P\big(X_i \,\big|\, \mathrm{Pa}(X_i)\big) \end{aligned}$$

This way not the complete joint has to be computed for each combination of the random variables that are marginalized out. algorithm 1 shows the pseudocode for this algorithm.

But even with this rather efficient algorithm, inference in Bayesian networks is still NP-hard!

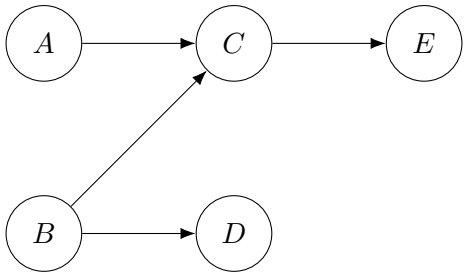


Figure 9.1: Example of a Bayesian network with the random variables A, B, C, D, and E.

Algorithm 1: Variable Elimination

Input: Query $P(\boldsymbol{Q})$.
Output: Probability $P(\boldsymbol{Q})$.

Initialize factors $\mathcal{F} = \{f_1, \cdots, f_N\}$ with $f_i = P\big(X_i \,\big|\, \mathrm{Pa}(X_i)\big)$
for $i = 1, \cdots, N$ **do**
    if $X_i \in \boldsymbol{Q}$ **then**
        skip
    Collect factors $f_1, \cdots, f_k$ that contain X_i.
    // Generate a new factor by eliminating X_i from these factors:
    $g = \sum_{X_i}^{\text{marg}} \prod_{j=1}^{k} f_j$
    // Add g to the set of factors:
    $\mathcal{F} \leftarrow \mathcal{F} \cup \{g\}$
Factors $\mathcal{F}$ now include $P(\boldsymbol{Q})$.

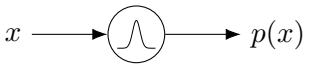
Low-Treewidth PGMs: Trees If in a Bayesian network every random variable has at most one parent, the Bayesian network is a *tree*. Then EVI, MAR, and CON queries can be answered exactly in linear time $\mathcal{O}(|\boldsymbol{X}|)$. Also, exact learning from d samples takes $\mathcal{O}(d\,|\boldsymbol{X}|^2)$ with the Chow-Liu algorithm.

This comes at the cost of not being able to represent rich and complex classes of distributions. In general, bounded-treewidth PGMs lose the ability to represent all possible distributions.

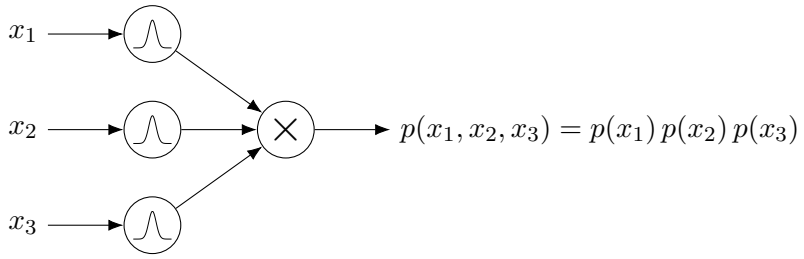
9.2 Probabilistic Circuits

Probabilistic circuits are computational graphs for probabilistic models. They are built from simple distributions as building blocks and summation and multiplication between them.

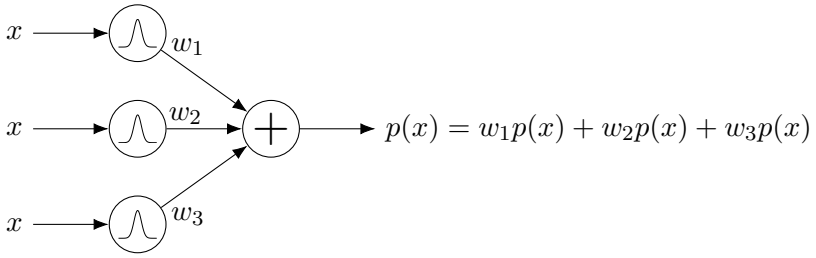
The starting point (base case) are univariate distributions, i.e. distributions over scalars. Generally they are tractable for EVI (by answering the value of the density/mass at the query point), MAR (bay answering 1 if normalized or Z), and MAP queries (by answering the location of the mode). The univariate building block is written as



as they are usually Gaussians. Multiple univariate probabilities can also be multiplied which is denoted as



for three random variables. More random variables are denoted analogous. One example application for this is to model a multivariate Gaussian with diagonal covariance matrix. To represent mixture models over a single random variable, weighted sums are used. These are denoted as



where the weights are written on the edges from the univariate distributions to the summing node. By stacking all these different nodes, expressive efficiency can be increased. It is, however, not possible to arbitrarily stack sums, products and distributions, certain structural constraints have to be fulfilled to ensure tractability. These will be covered in subsection 9.2.1.

But probabilistic circuits are not PGMs! They are probabilistic and graphical, but the nodes represent computations, not random variables and the edges determine the order of execution and do not encode dependencies. In PGMs, inference is done using conditioning, elimination, and message passing while in probabilistic circuits inference is done using forward- and backward-passes. So they are more like neural networks than PGMs. Due to them being computational graphs, a lot of things are possible: repeated computations can be cached, the operational semantics are clear and tractable in terms of circuit size, and they are differentiable, so gradient-based optimization is possible.

9.2.1 Ensuring Tractability

In order to ensure tractability in a probabilistic circuit, certain constraints have to be fulfilled. The first two obvious ones are *decomposability* and *smoothness*:

Decomposability All product nodes have to be decomposable. A product node is decomposable if its children depend on a disjoint set of variables, i.e., the same variable does not appear twice on the incoming nodes.

Smoothness All sum nodes have to be smooth. A sum node is smooth if its children depend on the same variable sets, i.e., the same variable has to be present on all incoming nodes.

These two properties enable tractable MAR and CON queries! This is due to large integrals that can be decomposed into easier ones

$$p(\boldsymbol{x}, \boldsymbol{y}) = p(\boldsymbol{x})\, p(\boldsymbol{y}) \quad \Longrightarrow \quad \iint p(\boldsymbol{x}, \boldsymbol{y})\, \mathrm{d}\boldsymbol{x}\, \mathrm{d}\boldsymbol{y} = \iint p(\boldsymbol{x})\, p(\boldsymbol{y})\, \mathrm{d}\boldsymbol{x}\, \mathrm{d}\boldsymbol{y} = \left(\int p(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x}\right)\left(\int p(\boldsymbol{y})\, \mathrm{d}\boldsymbol{y}\right)$$

and the marginalization sums can be pulled out of the integrals

$$p(\boldsymbol{x}) = \sum_i w_i p_i(\boldsymbol{x}) \quad \Longrightarrow \quad \int p(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} = \int \sum_i w_i p_i(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} = \sum_i w_i \int p_i(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x}$$

pushing the integrals down to the children. On both these cases, pre premise follows from the decomposability and smoothness property, respectively. To compute for example $p(X_2, X_3)$ when there are four random variables, the inputs for X_1 and X_4 are set to 1 (as they get integrated out) and the inputs for X_2 and X_3 are set to their respective probability which is provided from the unimodal distribution.

The next property that has to be ensured is *determinism* (also known as *selectivity*):

Determinism All sum nodes have to be deterministic. A sum node is deterministic if the output of at most one children is non-zero for any input, e.g. if their distributions have disjoint support.

This property enables tractable MAP queries! This is due to the argmax on a mixture model that gets drastically simplified because the sum over the weights can be discarded. Let $\boldsymbol{q} = \{\boldsymbol{q}_x, \boldsymbol{q}_y\}$ and $\boldsymbol{e} = \{\boldsymbol{e}_x, \boldsymbol{e}_y\}$ be separated into two arbitrary sets of random variables, then from decomposability is follows that $p(\boldsymbol{q}, \boldsymbol{e}) = p(\boldsymbol{q}_x, \boldsymbol{e}_x, \boldsymbol{q}_y, \boldsymbol{e}_y) = p(\boldsymbol{q}_x, \boldsymbol{e}_x)\, p(\boldsymbol{q}_y, \boldsymbol{e}_y)$. Thus, the argmax can be split at the product notes:

$$\arg\max_{\boldsymbol{q}}\ p(\boldsymbol{q} \mid \boldsymbol{e}) = \arg\max_{\boldsymbol{q}}\ p(\boldsymbol{q}, \boldsymbol{e}) = \arg\max_{\boldsymbol{q}_x, \boldsymbol{q}_y}\ p(\boldsymbol{q}_x, \boldsymbol{e}_x, \boldsymbol{q}_y, \boldsymbol{e}_y) = \arg\max_{\boldsymbol{q}_x}\ p(\boldsymbol{q}_x, \boldsymbol{e}_x),\ \arg\max_{\boldsymbol{q}_y}\ p(\boldsymbol{q}_y, \boldsymbol{e}_y)$$

and the optimizations can be solved separately. Due to determinism, i.e.

$$p(\boldsymbol{q}, \boldsymbol{e}) = \sum_i w_i p_i(\boldsymbol{q}, \boldsymbol{e}) = w_c p_c(\boldsymbol{q}, \boldsymbol{e})$$

where p_i is the only non-zero distribution, the argmax of a sum node is extremely simple:

$$\arg\max_{\boldsymbol{q}}\ p(\boldsymbol{q}, \boldsymbol{e}) = \arg\max_{\boldsymbol{q}}\ \sum_i w_i p_i(\boldsymbol{q}, \boldsymbol{e}) = \arg\max_{\boldsymbol{q}}\ w_c p_c(\boldsymbol{q}, \boldsymbol{e})$$

This pushes the argmax to the children. By repeatedly applying these two rules, the argmax can be pushed to the leaves (the random variables) where a MAP query is simple to answer as the mode of the unimodal distribution. In practice, the evaluation of a MAP query goes as follows:

1. Turn all sum nodes into (weighted) max nodes.
2. Evaluate $p(\boldsymbol{e})$, the marginal of the given evidence, in a forward pass.
3. Retrieve the max activations in a backward pass.
4. Compute MAP queries at the leaves.

Compares to MAR and CON queries, the computation has to be run two times through the network, but it is still linear in circuit size. If the circuit is non-deterministic, MAP is intractable, but the algorithm can still be used as an approximation. The circuit is then called a *sum-product network* (SPN) which are equivalent the exception of determinism.

The final requirement is *structured decomposability*:

Structured Decomposability A product node is structured decomposable if it decomposes according to a node in a binary tree.

This is a stronger requirement than decomposability and is not necessary for MAR, CON, and MAP queries. But it enables tractability of the *entropy* of the circuit, *symmetric* and *group queries* (exactly k, odd number, more, etc.), and, for the "right" binary tree, also the probability of logical circuit event in a probabilistic circuit, *multiplication* of two probabilistic circuits, *KL-divergence* between two circuits, *same-decision probability*, *expected same-decision probability*, *expected classifier agreement*, and *expected predictions*. But as structure decomposability is a pretty harsh requirement, it is usually not enforced.

9.2.2 Other Circuits and Circuit Compilation

This section addresses the similarities and differences between probabilistic, logical, and other circuits. The tractable probabilistic inference view exploits efficient summation for decomposable functions in the commutative semi-ring of probabilities, $(\mathbb{R}, +, \times, 0, 1)$. This is also possible in other semi-rings with generalized operations, $(\mathbb{S}, \oplus, \otimes, 0_\oplus, 1_\otimes)$. Historically a very well studied semi-ring is the one for boolean functions: $(\mathbb{B} = \{0, 1\}, \vee, \wedge, 0, 1)$. These lead to logical circuits which represent logical functions in a compact way. These look the same way as probabilistic circuits, but the nodes are logical ands and ors instead of sums and products. Also similar to probabilistic circuits, it is possible to define structural properties like *decomposability*, *smoothness*, and *determinism* to allow for tractable computations and—in some sense—inference of boolean values.

Circuit Compilation: From Trees to Circuits

A big step into tractability is to compile probabilistic graphical models—or probabilistic trees only, in this section—into probabilistic circuits. This can be done in a bottom-up approach: at first a leaf node of the probabilistic tree is compiled into a sum node structure in the probabilistic circuit where for each value of the leaf there exists one leaf in the circuit and for each value of the dependent variable there exists a sum node. The weights are the probabilities. This is done for every leaf node of the tree. Then these structures are taken as leaves with multiple outputs, corresponding to each value of the random variables. The "leaf structures" are then used to recursively build up the parent nodes of the probabilistic tree which builds up the circuit in the end.

Take, for example, the probabilistic tree shown in Figure 9.2. To compile the first node, the probabilities

$$P(A = 0 \,|\, C = 0) = 0.3 \qquad P(A = 1 \,|\, C = 0) = 0.7$$
$$P(A = 0 \,|\, C = 1) = 0.6 \qquad P(A = 1 \,|\, C = 1) = 0.4$$

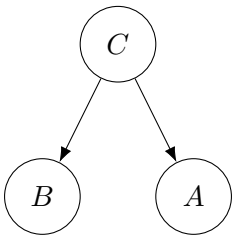


Figure 9.2: Example probabilistic tree.

are used. First the circuit for $P(A \mid C = 0)$ is built and subsequently the circuit for $P(A \mid C = 1)$ is added. Both of these steps are shown in Figure 9.3a and 9.3a. Analogously the circuit for $P(B \mid C = 0/1)$ is built in Figure 9.3c and 9.3d. To now build the probabilities $P(A, B, C = 0/1)$ and $P(A, B, C)$, the already build probabilities have to be multiplied and then C has to be summed up w.r.t. the probabilities $P(C = 0) = 0.2$ and $P(C = 1) = 0.8$. All of this is shown in Figure 9.3e.

Equivalent and Related Formulations

As trees, polytrees and thin junction trees can be turned into decomposable, smooth, and deterministic probabilistic circuits, they all support tractable EVI, MAR, CON, and MAP queries. The same goes for arithmetic circuits (ACs) which are also decomposable, smooth, and deterministic.

Other equivalent formulations are:

- *Cutset networks* (CNets): CNets are weighted decision trees whose leaves are Bayesian networks. It is possible to replace every decision node by a deterministic and smooth sum node, and thus it is possible to compile CNets into probabilistic circuits. Hence, they are decomposable, smooth, and deterministic and thus EVI, MAR, CON, and MAP queries are tractable.
- *Probabilistic sentential decision diagrams* (PSDDs): These are also *structured* decomposable, smooth, and deterministic. Hence, EVI, MAR, CON, MAP, and also complex queries are tractable!

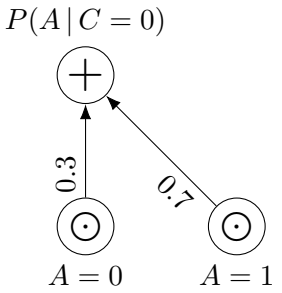
As probabilistic circuits are so diverse and expressive, they belong to the most expressive while still tractable models!

9.3 Building Circuits

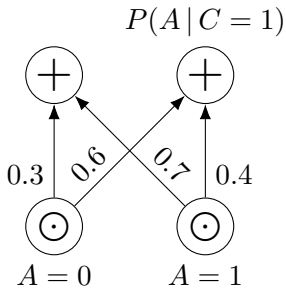
After discussing probabilistic circuits and seeing how expressive and expression efficient they are, the next big question is how to build these circuits tractably because usually the circuits are not given but the shall be learned from data. The first step is to define a *tractable learner*: a learner L is *tractable* for a class of queries $\mathcal{Q}$ if and only if

1. for any dataset $\mathcal{D}$, the learner $L(\mathcal{D})$ runs in $\mathcal{O}\big(\text{poly}(|\mathcal{D}|)\big)$. This guarantees that the size of the learned model is in $\mathcal{O}\big(\text{poly}(|\mathcal{D}|)\big) \cap \mathcal{O}\big(\text{poly}(|\boldsymbol{X}|)\big)$.
2. it outputs a probabilistic model that is tractable for all queries in the class $\mathcal{Q}$. This guarantees efficient querying for $\mathcal{Q}$ in time $\mathcal{O}\big(\text{poly}(|\boldsymbol{X}|)\big)$.

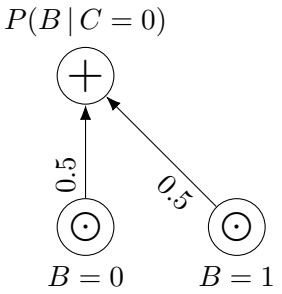
This section will now introduce different methods for learning circuit parameters and also structure. Additionally, compilation of other models into circuits is covered.



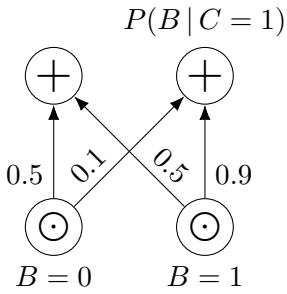
(a) Compilation of $P(A \mid C = 0)$.



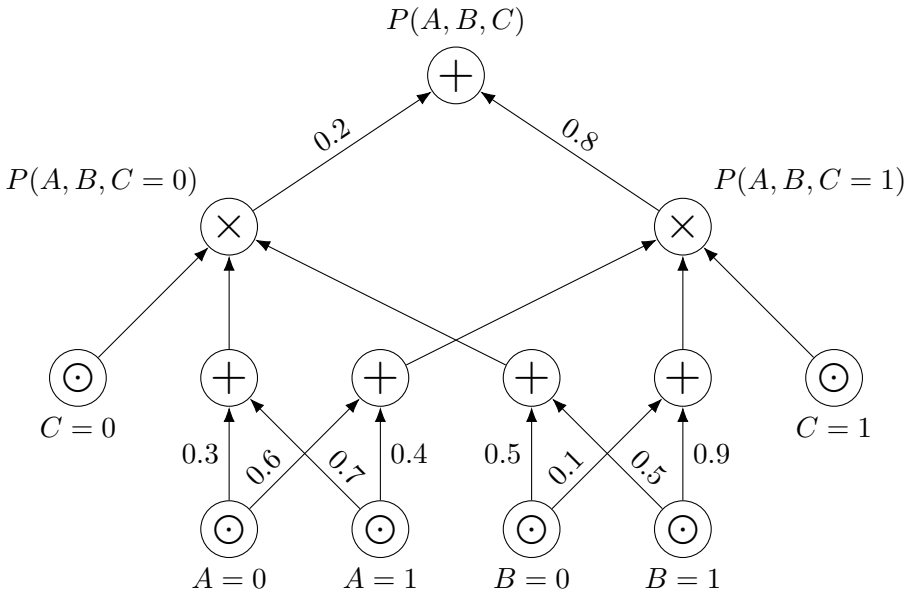
(b) Compilation of $P(A \mid C = 1)$.



(c) Compilation of $P(B \mid C = 0)$.



(d) Compilation of $P(B \mid C = 1)$.



(e)

Figure 9.3: Steps of compiling the probabilistic tree in Figure 9.2 into a probabilistic circuit. The distribution node with a $\odot$ described the binary distribution that outputs 0 if the condition below the node holds and 1 otherwise. Hence, the nodes $A = 0$ and $A = 1$ are mutually exclusive and determinism holds (similar for the others).

9.3.1 Learning Circuit Parameters

The easiest part of circuit learning is learning the parameters, i.e. the probabilities, of the circuit. These are leaf parameters $\boldsymbol{\theta}$ and the weights $\boldsymbol{w}$ on summing nodes. For deterministic circuits, this can be done in closed form with convex optimization. For non-deterministic circuits, this is harder. Some algorithms for learning the parameters in non-deterministic circuits are SGD, soft/hard EM, Bayesian moment matching, collapsed variational Bayes, CCCP, and extended Baum-Welch. This section will covert the former two algorithms.

Deterministic Circuits

For deterministic circuits, approximating the parameters can simply be done by maximizing the data likelihood

$$\boldsymbol{\theta}^{\mathrm{MLE}}, \boldsymbol{w}^{\mathrm{MLE}} = \arg\max_{\boldsymbol{\theta},\boldsymbol{w}} \; L(\boldsymbol{\theta}, \boldsymbol{w}; \mathcal{D}) = \arg\max_{\boldsymbol{\theta},\boldsymbol{w}} \; \prod_{d \in \mathcal{D}} p_{\boldsymbol{\theta},\boldsymbol{w}}(d).$$

With determinism, the likelihood decomposes over the parameters and the optimization problem has a closed-form solution.

Hard/Soft Parameter Updating: Gradient Descent and EM

Optimizing the parameters via gradient descent is simply done by computing the gradient of the likelihood and applying standard optimization techniques. For using EM, each summing node is considered as the marginalization over a hidden variable that is estimated in the E-step.

Bayesian Parameter Learning

In *Bayesian* parameter learning, a prior $p(\boldsymbol{w})$ is put on the weights of the circuit. Then the posterior

$$p(\boldsymbol{w} \mid \mathcal{D}) \propto p(\mathcal{D} \mid \boldsymbol{w})\, p(\boldsymbol{w})$$

is maximized. But this posterior is intractable! Instead of a general prior, a factorized prior

$$p(\boldsymbol{w}) = \prod_{i \in \text{Sum Nodes}} Dir(\boldsymbol{w}_i \mid \alpha_i)$$

is assumed in a circuit with normalized weights $w_{ij} \geq 0$ and $\sum_j w_{ij} = 1$ where $i \in$ Sum Nodes. The posterior then becomes a mixture of Dirichlet distributions where the number of mixture components is exponential in the number of sum nodes.

Moment Matching (oBMM) The posterior is approximated with a tractable distribution after each update that matches some moments of the exact, but intractable, posterior. The joint $p(\boldsymbol{w})$ is approximated by a product of Dirichlet distributions and the first and second moment of each marginal $p(\boldsymbol{w}_i)$ are used to set the parameters α_i of each Dirichlet.

oBMM Extended Uses continuous models with Gaussian leavers.

CVB-SPN Uses a collapsed variational inference algorithm and produces better results than oBMM.

Sequential Monomial Approximation and Concave-Convex Procedure Any complete and decomposable circuit can be transformed into a mixture of trees where each tree corresponds to a product of univariate distributions. *Sequential monomial approximation* learns the parameters based on a maximum likelihood estimator which can be formulated as a sigmonial program. This program can equivalently be transformed into a different of convex functions, yielding the *concave-convex procedure*.

9.3.2 Structure Learning

Instead of "just" learning the parameters of a probabilistic circuit, *structure learning* algorithms address the problem of learning the structure of the circuit, i.e., the connections between the nodes.

A general approach for learning the structure is to start with a dense circuit and then find the structure by learning the weights and removing connections with near-zero weights. This is, in principle, possible to every arithmetic circuit by implementing it in any autograd library. It also works well but there are more advanced algorithms that produce better results.

LearnSPN and Variants *LearnSPN* learns both structure and parameters of a circuit simultaneously. For this it starts from a data matrix that lists all values for each random variable. Then clusters are searched in the data for building sum nodes. Once clusters have been found and sum nodes have been created, an independence test is used within the leaves to find independent random variables. Once these have been found, the circuit is factorized into these random variables. These two steps iterate until either no clusters can be found anymore.

A variant of LearnSPN is ID-SPN which starts with a single arithmetic circuit (AC) that represents a tractable Markov network. The process of clustering instances and variables for sum and product nodes is then stopped before the process reaches univariate distributions. These are replaced by a learned tractable MN represented by an AC factorizing multiple distributions. This creates an SPN with tractable multivariate distributions as leaves, so-called Markov network arithmetic circuits (MN AC).

Another variant is *bottom-up learning* which start from a simple model over a small variable scope and increase this model successively over a larger variable scope guided by dependence tests and a maximum mutual information principle. Other variants are *greedy for deterministic circuits* and *graph SPNs* which transform the network as a hill climbing algorithm using split and merge operations and form tree SPNs my merging similar sub-structures, respectively.

9.3.3 Ensembles of Probabilistic Circuits

To solve issues like the tendency of overfitting of probabilistic circuits and to improve accuracy of a single model, probabilistic circuits an be used in an ensemble in the form of a mixture model

$$p(\boldsymbol{x}) = \sum_{i=1}^{K} \lambda_i C_i(\boldsymbol{x}), \quad \lambda_i \geq 0, \sum_{i=1}^{K} \lambda_1 = 1.$$

To learn both the parameters of each circuit and to learn the mixture components weights $\boldsymbol{\lambda}$, an EM-algorithm can be employed. But this raises some issues in convergence and stability, making this impractical.

A better alternative for using circuit ensembles is bagging which is more efficient than EM. For training the circuits the mixture components are all set equally probable and are then learned independently on different bootstraps. To make these models even more efficient, random subspace projects can be added (like for CNets).

It is also possible to boost probabilistic circuits. One method the *boosting density estimation* (BDE) which works like AdaBoost and combined models in the form $f_m = (1 - \eta_m) f_{m-1} + \eta_m c_m$ where f_m is the m-th

model, c_m is the current weak learner and η_m is the weight of that learner. Another method is GBDE which is kernel-based. It is also possible to use sequential EM which jointly optimized η_m and c_m while keeping f_{m-1} fixed.

9.4 Applications

This sections shows some of the applications of tractable inference and probabilistic graphical models in general.

9.5 Takeaways and Open Challenges

The three primary challenges in probabilistic graphical models are:

1. *Better benchmarks:*
   To move beyond toy problems and dataset reflecting to complex and heterogeneous nature of real data. Move towards fully automated reasoning!

2. *Hybridizing tractable and intractable models:*
   Use tractable models inside intractable loops and glue together small intractable boxed by tractable inference.

3. *Scaling tractable learning:*
   Learn tractable models with millions of datapoints and thousands of feature in tractable time.

The takeaways are that tractability is a spectrum and there are more and less tractable models, it is possible to be both tractable and expressive at the same time (probabilistic circuits), probabilistic circuits are the foundation for tractable inference and learning, and that there is still a lot more to do.

10 Natural Language Processing

Natural language processing (NLP) is concerned with processing, interpreting, and translating naturally spoken or written text. This chapter will cover some parts of natural language understanding in form of the underlying semantics (section 10.1) as well as translation and structure extraction (section 10.2).

10.1 Text Semantics

In NLP, *text semantics* addresses the problem of extracting the meaning of texts. The models in this category can be split into two supercategories:

Propositional Semantics Text gets translated into a logical language, e.g. predicate calculus. For example the text "dog bites man" can be translated into the formula $\mathrm{bites}(\mathrm{dog}, \mathrm{man})$, where $\mathrm{bites}(\cdot, \cdot)$ is a binary relation and man and dog are objects. It is also possible to attach probabilities to these formulas.

Vector Representation Text is embedded into a high-dimensional vector space. For example the text "dog bites man" can be represented as a vector $(0.2, -0.3, 1.5, \cdots) \in \mathbb{R}^n$. Sentences with a similar meaning should then be close to this embedding.

Text semantics can again be decomposed into two kinds of analysis: *lexical* and *compositional* semantics. In the former the meaning of individual words is analyzed while in the latter the meaning depends on the surrounding words and how they are combined. The next two sections will focus on propositional semantics and vector representations, respectively which can be used for both lexical and compositional semantics.

10.1.1 Propositional Semantics

Propositional semantics allow for logical inferences and allow the use of powerful result from logic to interpret text. For example: it is possible to infer "Socrates is mortal" from "Socrates is a man" and "all men are mortal".

10.1.2 Vector Representation

The second approach to semantic analysis are vector embedding of words. These are based on a notion of *similarity* and are based on the definition of paradigmatic similarity: similar words occur in similar contexts and are exchangeable. Word embeddings are vectors in a high dimensional vector space usually use a

bag-of-words, for example:

$$\text{vec("dog")} = \begin{bmatrix} 0.2 \\ -0.3 \\ 1.5 \\ \vdots \end{bmatrix} \qquad \text{vec("bites")} = \begin{bmatrix} 0.5 \\ 1.0 \\ -0.4 \\ \vdots \end{bmatrix} \qquad \text{vec("man")} = \begin{bmatrix} -0.1 \\ 2.3 \\ -1.5 \\ \vdots \end{bmatrix}$$

$$\implies \qquad \text{vec("dog bites man")} = \text{vec("dog")} + \text{vec("bites")} + \text{vec("man")} = \begin{bmatrix} 0.6 \\ 2.0 \\ -0.4 \\ \vdots \end{bmatrix}$$

The following sections will cover four models that all fall into the category of vector representations: latent semantic analysis, word2vec, skip-thought vectors and Siamese networks.

Latent Semantic Analysis

Latent semantic analysis (LSA) uses a bag-of-words format for studying documents. All documents are encoded in a matrix $\boldsymbol{T} \in \mathbb{R}^{N \times M}$ where the entry T_{ij} represents how often word j appears in document i[1]. Often not the count is used as a metric but tf-idf (term frequency-inverse document frequency) or other "squashing" functions. Obviously, most entries of the matrix are zero. Then an approximate factorization $\boldsymbol{T} \approx \boldsymbol{U}^T \boldsymbol{S} \boldsymbol{V}$ with $\boldsymbol{U}^T \in \mathbb{R}^{N \times K}$, $\boldsymbol{S} \in \mathbb{R}^{K \times K}$, $\boldsymbol{V} \in \mathbb{R}^{K \times M}$ is computed where K is the latent dimensionality. If $K = M$, then this is the singular value decomposition of $\boldsymbol{T}$. The latent dimensionality $\boldsymbol{K}$ is chosen based on the Eigenvalues of $\boldsymbol{S}$ by leaving out the smallest Eigenvalues.

Given a document $\boldsymbol{t}$, i.e., a row of $\boldsymbol{T}$, $\boldsymbol{v} = \boldsymbol{V}\boldsymbol{t}$ is an embedding of the document in the latent space and $\boldsymbol{t}' = \boldsymbol{U}^T \boldsymbol{v} = \boldsymbol{U}^T \boldsymbol{V} \boldsymbol{t}$ is the decoding from its embedding. The latent space can then be used to compute similarities between documents.

Note how singular value decomposition of the term frequency matrix is used in order to compute the most relevant Eigenvectors for the embedding. This is similar to principal component analysis which essentially does the same thing, but on the covariance matrix. Also, as PCA first normalizes the data, it looses the sparseness of the factorized matrix, making it infeasible for large bag-of-words.

Word2Vec

As opposed to latent semantic analysis, *word2vec* does not use entire documents but only a few positions before and after a word to determine its local context. The pairs of these words are called *skip-grams*. For example, if three context words are used[2], the red word is the center word and the blue words are the context words:

"This is a sentence that does not contain any content and can thus be ignored."

In a complete word2vec model, all words are considered as center words. The model is then trained for trying to predict the context words given a center word. It is also possible to train the inverse, i.e., predict the center word given context words, but skip-grams in general perform better. Training is done by optimizing a softmax loss

$$p(j \mid i) = \frac{\exp\left\{\boldsymbol{u}_j^T \boldsymbol{v}_i\right\}}{\sum_k \exp\left\{\boldsymbol{u}_k^T \boldsymbol{v}_i\right\}}$$

[1] Where N and M are the total number of documents and words/word features, respectively.
[2] Three to five words are usual.

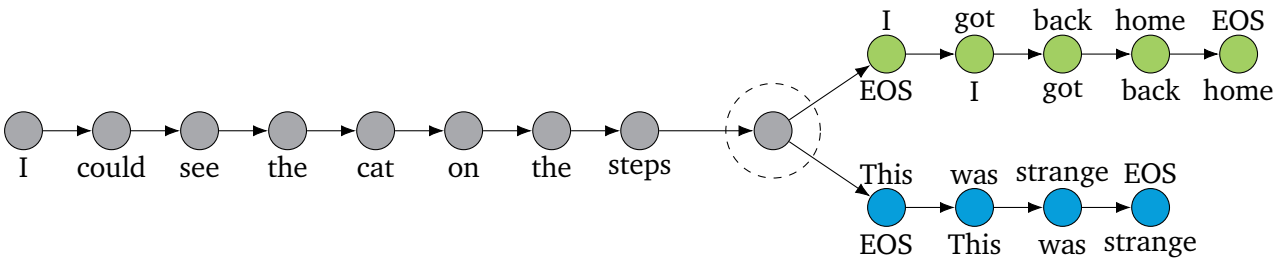


Figure 10.1: Illustration of the RNN that backs skip-thought embeddings. The text below the nodes represent inputs to the RNN cell, the text above represents the output. Gray nodes are the input cells, green and blue are the prediction for the previous and next sentence, respectively. The hidden state of the surrounded cell is used as the embedding. "EOS" stands for "end of sentence".

where j is the output word, i is the input word, and j ranges over the context (e.g. from -3 to 3 if three context words are used). The vectors $\boldsymbol{u}_j$ and $\boldsymbol{v}_i$ are output and input embedding vectors, respectively. Word2Vec can thus be implemented with standard deep learning tools by differentiating the loss to optimize the output and input embeddings.

Compared to LSA, local contexts reveal much more information about relations and properties between words than LSA. Relations between words are characterized by their displacement, e.g. $\text{vec}(\text{“woman”}) - \text{vec}(\text{“man”})$ describes the distance between the word "woman" and the word "man". Word2Vec manages to generalize these structures, such that the said distance is for example roughly equal to the distance between "queen" and "king":

$$\text{vec}(\text{“woman”}) - \text{vec}(\text{“man”}) \approx \text{vec}(\text{“queen”}) - \text{vec}(\text{“king”})$$

These kinds of relations can be found a lot when analyzing the results from word2vec models.

Skip-Thought Embeddings

All vector representation models discussed before only capture bag-of-words, i.e., the order of how words appear is neglected. But order matters! Take for example the two sentences "dog bites man" and "man bites dog": they have opposite meanings. *Skip-though embeddings* use many-to-many RNNs for predicting the next and previous sentence. The hidden state after the complete sentence was entered into the RNN (but before the prediction started) is then used as the vector embedding. Once the RNN is trained, the two prediction heads can be discarded. page 65 illustrates this.

Siamese Networks

Siamese networks utilize two recurrent neural networks (usually LSTMs) that share theirs parameters. It is trained by inputting two different sentences into each RNN and combining them using the Manhattan distance. This produces a similarity measure y. The training data contains pairs of sentences that are labeled with a similarity measure such that well-known backpropagation and gradient descent can be used for training the Siamese network.

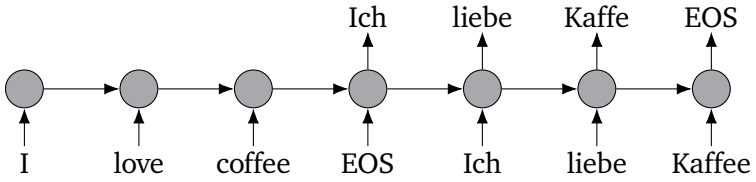


Figure 10.2: Illustration of machine translation using a many-to-many RNN. Again, "EOS" stands for "end of sentence".

10.2 Translation Models

This chapter covers *machine translation* and *translation models* that do not use attention and transformer networks. These will be covered in chapter 11.

In theory it is possible to develop machine translation models by using many-to-many RNNs that take a sentence as an input and produce the sentence as an output in another language (see Figure 10.2). It is also possible to generate multiple possible output sequences, compare them to each other and choose the best. One possible score for comparing machine translations is a *unigram precision*. It compares machine translations against human translations by dividing the number of words of the machine translation that also appear in the reference by the total number of words in the candidate sentence (a *unigram* is a single word). This has the problem that for example the candidate "the the the the the the the" compared to the references "the cat is on the mat" and "there is a cat on the mat" has the maximum precision of $7/7 = 1$:

$$\text{Unigram Precision} == \frac{\text{Correct words occuring in reference sentence.}}{\text{Words occuring in candidate sentence.}} = \frac{7 \cdot \text{“the”}}{7 \cdot \text{“the”}} = 7/7 = 1$$

An alternative is to use the maximum number of occurrences throughout the candidates as the nominator. This way, in the above example, the unigram precision would be $2/7$ because the word "the" appears two times in "the cat is on the mat". A generalization are n*-gram precisions* that are defined similarly:

$$n\text{-Gram Precision} = \frac{\text{Correct } n\text{-grams occuring in reference sentence.}}{n\text{-Grams occuring in candidate sentence.}}$$

Like the unigram precision, the modified n-gram precision clips the nominator by the maximum occurrences throughout the candidates. Unigram precisions tend to capture adequacy while n-gram precisions tend to capture fluency.

To combine multiple n-gram precisions, the *BLEU* score can be used. It defines a weighted geometric mean over N different n-scores:

$$\text{BLEU} = \text{BP} \cdot \exp\left\{\sum_{n=1}^{N} w_n \log p_n\right\} \qquad \text{BP} = \begin{cases} 1 & \text{if } c > r \\ e^{1-r/c} & \text{if } c \leq r \end{cases}$$

Here p_n is the n-gram precision, c is the length of the candidate and r is the length of the reference translation. It has been shown that the BLEU score highly correlates with human judgment.

11 Attention and Transformers

Attention models us *attention masks* to let the model know which parts of the input it should attend to. They can be separated into two classes: hard and soft attention. In hard attention, the model is forced to attend to a single input location. Thus, it is not possible to use gradient descent and techniques from reinforcement learning have to be leveraged to train these models. Soft attention models on the other hand compute a weighted combination over some inputs using an attention network. The weights are the *attention mask*. As this is differentiable, backpropagation and gradient descent can be used for end-to-end training.

A really good introduction to attention and transformer networks by *Justin Johnson* can be found here:

`https://youtu.be/YAgjfMR9R_M`

11.1 Soft Attention for Translation

The starting point for soft attention are recurrent neural networks and RNN translation which have the problem that the complete information about the sentence, no matter how long, has to be pushed into a single hidden state. Attention models use an attention network to generate weights that are then used for adding another input to each generating RNN cell that is directly computed from the input sentence (or, more specifically, the hidden states of the RNN input cells). The input to the decoder/generator network is

$$\boldsymbol{c}_i = \sum_{j=1}^{T} \alpha_{ij} \boldsymbol{h}_j$$

where α_{ij} are the weights for the annotations $\boldsymbol{h}_j$. These are the attention mask

$$\alpha_{ij} = \frac{\exp\{e_{ij}\}}{\sum_k \exp\{e_{ik}\}}$$

where $e_{ij} = a(\boldsymbol{s}_{i-1}, \boldsymbol{h}_j)$ is the *alignment score* which is a learnable function. The vector $\boldsymbol{s}_{i-1}$ the hidden state of the RNN cell right before emitting the output[1]. The soft attention models are extremely powerful and it only took one year for them to reach the state-of-the-art after release. Currently they are surpassing any preexisting translation models.

11.2 Attention for Captioning

It is also possible to use attention for image captioning. Here the output of a CNN is used an interpreted as stacked features which are then weighted according to the attention. This enables the image captioning system (a RNN) to look at different parts of the image for different words. A similar procedure can be used for video data.

[1] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio (2015): "Neural Machine Translation by Jointly Learning to Align and Translate"

When using soft attention, the features are summed up and weighted according the (normalized) attention. In hard attention, a features is sampled from according to the (normalized) attention which form a probability distribution. But as sampling is not differentiable, one might resort to using the feature with the maximum attention instead. Nevertheless the derivative of this will be zero, killing all subsequent gradients, hence no learning via gradient descent is possible. That is why reinforcement learning is needed in this case and why soft attention is preferred.

11.3 Attending to Arbitrary Regions

Attending according to the original paper "Show, Attend, and Tell" (Xu et al., 2015) only allows attending for regions on a fixed grid—but would not attending to arbitrary regions be better? It was shown by Graves et al. in 2013 that it is possible to attend to arbitrary regions by predicting the parameters of a mixture model. *DRAW* (Gregor et al., 2015) can classify images by attending to arbitrary images of the input and can also generated images by attending to arbitrary regions of the output.

Another attention mechanism are *spatial transformer networks* (Jaderberg, 2015). They try to find a mapping between the coordinated of in- and output pixels. Repeating this for all pixels in the output yields a sampling grid. Then a bilinear transformation is used to compute the output.

11.4 Transformer Networks

Transformer networks where originally introduced by Vaswani et al. in 2017 in a paper called "Attention is all you need". They propose an encoder-decoder architecture that is purely based on attention and does not utilize any recurrence. The core idea is that of *multihead attention*, meaning that multiple attentions are computed per query with different weights. Additionally layer normalization is added which normalizes the values of each layer to have zero mean and unit variance[2]. This reduces the covariate shift and therefore reduces training time. Also codes for the position of an input are added as otherwise self-attention layers would have no notion of order.

Compared to RNNs, transformer networks are much more efficient as everything can be computed in parallel as compared to sequential models. Also the maximum path length is reduced, reducing the problem of vanishing gradients. The transformer network achieves better results in machine translation and natural language generation than every previous state-of-the-art model. Example models that use transformers are GPT, GPT-2, and BERT.

11.5 Takeaways

Performance Attention models can improve accuracy and reduce computation time at the same time.

Complexity There are many different design choices for attention networks that have a big impact on performance. Attention models can be quite complex, but they get simplified whenever possible.

Explainability Attention can yield more interpretability by seeing where the model looks.

[2]Note that this is different to batch normalization which normalizes all batches to have zero mean and unit variance.

Hard vs. Soft Soft models are generally easier to train; hard models require reinforcement learning. But they can also be combined (Luong et al., 2015).