Heidelberg University

Machine Learning and Physics

Winter term 2022/2023

Prof. Dr. Fred Hamprecht

Acknowledgements

This script is based on the "Machine Learning and Physics" lecture delivered in the Philosophenweg auditorium of Heidelberg University in the winter semester of 2022.

Please report any errors by opening an issue, or suggest improvements by making a pull request, to https://github.com/FredHamprecht/ml-and-physics

I would like to thank Christof Gehrig, Marcia Kroker and Simon Wagner for their writing the entire first version of the script! They have done a magnificent job of turning my illegible black board scribbles into a coherent text.

I would also like to thank the participants for many excellent questions, for spotting and correcting my mistakes, and for their curiosity that kept them going through a tough course and semester.

Finally, I would like to thank all assistants from my lab – mainly Ocima Kamboj and Roman Remme for the present course – who have helped create and run tutorials over the years, and who are teaching me so much.

Fred Hamprecht
Heidelberg, July 5th, 2023

License

Contents

1 Principal component analysis (PCA)

1.1 Supervised and unsupervised learning

In machine learning, we differentiate between two main approaches, supervised and unsupervised learning. For supervised learning, we have data that consists of some *features/explanatory variables* and corresponding *labels* of a quantity of interest. We usually want to learn to predict the labels given the features. For unsupervised learning, we have *unlabeled data* and our goal is usually to visualize the data, find inherent structures or learn representations.

1.2 Dimension reduction techniques

When we have high-dimensional data, we may want to use dimension reduction techniques to be able to visualize the data more easily and to compress the data (with losses) to speed up training and inference. An example of high dimensional data and its visualization is shown in Figure 1.1 and Figure 1.2. This dimension reduction works because usually the intrinsic dimensionality of the data is smaller than the nominal dimensionality of the initial data set. Dimension reduction techniques belong to unsupervised machine learning, because we can use unlabeled data.

	H_10_OWNPV_X	H_10_OWNPV_Y	H_10_OWNPV_Z	H_10_IP_OWNPV	H_10_FD_OWNPV	H_10_DIRA_OWNPV	H_10_P	H_10_PT	H_10_PE	H_1
0	0.316851	-2.253858	-0.569767	0.115474	-0.569767	-0.147303	0.944242	1.324925	0.940307	-1.7
1	0.093897	-2.159661	0.565860	0.118032	0.565860	-0.146285	0.128807	0.555123	0.124709	1.3
2	1.314110	-2.107392	0.482418	0.315618	0.482418	-0.033911	-1.035826	-0.443795	-1.036710	0.7
3	1.449947	-2.693863	-1.158423	-1.121059	-1.158423	-0.486022	2.295295	-1.105588	2.296491	0.0
4	1.555626	-1.956518	2.504659	0.731588	2.504659	0.253691	-0.078069	1.124767	-0.083084	2.1
...	...	...	...	...	...	...	...	...	...	
11158	0.413309	-2.151830	0.353911	-0.066392	0.353911	-0.236654	-0.717479	-0.393165	-0.716087	0.3
11159	0.389883	-1.912995	-1.523739	0.426988	-1.523739	0.037377	-0.061270	0.766609	-0.067016	0.0
11160	0.462537	-2.247286	-0.962591	-0.635019	-0.962591	-0.427649	-0.832103	-0.882463	-0.834196	-0.1
11161	1.281788	-2.076162	-1.626494	-0.899724	-1.626494	-0.471444	1.087712	-0.682265	1.094477	0.3
11162	0.959539	-2.002083	0.038553	-0.562597	0.038553	-0.410723	-0.897992	-0.861851	-0.898595	-0.1

11163 rows × 126 columns

Figure 1.1: Exemplary plot of initial data set of simulated jets of quarks in the LHCb, data set from [col20]. We have a lot of features that initially don't tell us much.

Figure 1.2: First two principal components of the same data as shown in Figure 1.1, data set from [col20]. We can now identify some structure in the data.

1.3 Conventions

In the following chapters, we will consider our training data to be given as a $\mathfrak{p} \times \mathfrak{n}$-matrix $\boldsymbol{X} \in \mathbb{R}^{\mathfrak{p} \times \mathfrak{n}}$, with the number of features (/attributes/independent variables) $\mathfrak{p}$ and the number of samples (/cases/observations/examples) $\mathfrak{n}$. Typically, we will have less features (~ 10 to 10^4) than samples (~ 100 to 10^7).

1.4 Principal component analysis

In *principal component analysis* (PCA), we want to project the data into the linear subspace of the original space that "best approximates" the data or "contains" most of the data. The data is assumed to be on interval scale (continuous). We take the following steps:

1. *Normalize data:* The best subspace must contain the mean, so we first center the data such that
   $\frac{1}{\mathfrak{n}} \sum_{j=1}^{\mathfrak{n}} x_{ij} \stackrel{!}{=} 0 \quad \forall i \in 1, \cdots, \mathfrak{p} \quad \Leftrightarrow \quad \frac{1}{\mathfrak{n}} \boldsymbol{X} \mathbf{1}_{\mathfrak{n}} = \mathbf{0}_{\mathfrak{p}}$. We get this by

$$\begin{aligned} \boldsymbol{X}_{centered} &= \boldsymbol{X} - \frac{1}{\mathfrak{n}} \boldsymbol{X} \mathbf{1}_{\mathfrak{n}} \mathbf{1}_{\mathfrak{n}}^{\top} && (1.1) \\ &= \boldsymbol{X} (\boldsymbol{I}_{\mathfrak{n}} - \frac{1}{\mathfrak{n}} \mathbf{1}_{\mathfrak{n}} \mathbf{1}_{\mathfrak{n}}^{\top}) && (1.2) \\ &= \boldsymbol{X} \boldsymbol{C}_{\mathfrak{n}} && (1.3) \end{aligned}$$

   Here, $\mathbf{1}_{\mathfrak{n}}$ is a $\mathfrak{n} \times 1$ vector with ones as all entries. $\boldsymbol{I}_{\mathfrak{n}}$ is the $\mathfrak{n} \times \mathfrak{n}$ identity matrix. Furthermore, we have to scale the data to adjust the variance. Ideally, we have some idea of the natural scaling of the different features and can scale accordingly. Otherwise, we just normalize each of the features to have unit variance. Bringing the mean to 0 and variance to 1 is also called standardizing.

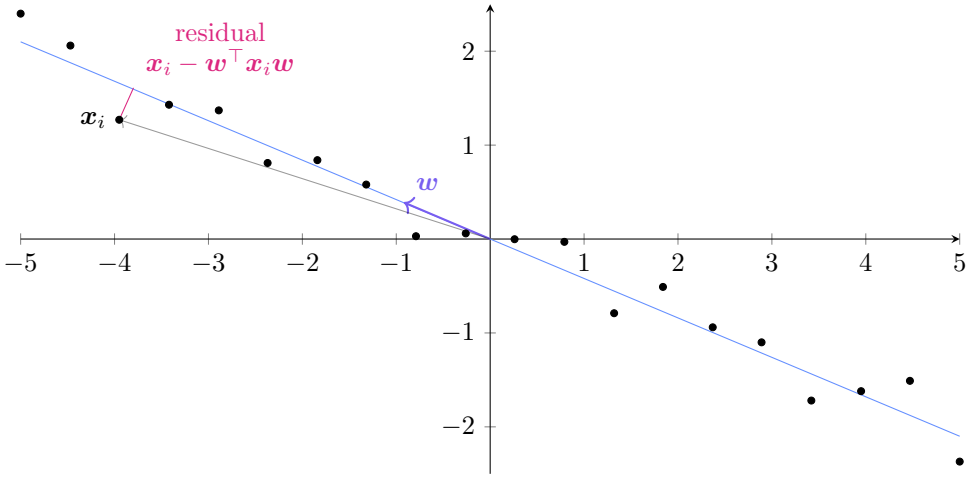


Figure 1.3: The first principal component (blue line) is optimized to have minimum squared residuals $\Leftrightarrow$ maximum variance along the two-dimensional, centered data points. We can view the residuals as springs pulling the blue line to its ideal position.

2. *Calculate the principal components:*

   We want to calculate the first principal component. To do so, we want to minimize the sum of squared residuals between the projections onto the subspace and the original data points. For a 2D example, this problem is shown in Figure 1.3.

$$\hat{\boldsymbol{w}} = \underset{\boldsymbol{w}:\, \boldsymbol{w}^\top \boldsymbol{w}=1}{\arg\min} \sum_i \| \underset{\mathfrak{p}\times 1}{\boldsymbol{x}_i} - \overbrace{\underset{1\times \mathfrak{p}}{\boldsymbol{w}^\top} \underset{\mathfrak{p}\times 1}{\boldsymbol{x}_i} \underset{\mathfrak{p}\times 1}{\boldsymbol{w}}}^{\boldsymbol{w}\boldsymbol{w}^\top \boldsymbol{x}_i} \|_2^2 \tag{1.4}$$

$$= \underset{\boldsymbol{w}:\, \boldsymbol{w}^\top \boldsymbol{w}=1}{\arg\min} \| \underset{\mathfrak{p}\times \mathfrak{n}}{\boldsymbol{X}} - \underset{\mathfrak{p}\times 1}{\boldsymbol{w}} \underset{1\times \mathfrak{p}}{\boldsymbol{w}^\top} \underset{\mathfrak{p}\times \mathfrak{n}}{\boldsymbol{X}} \|_F^2 \tag{1.5}$$

$$= \underset{\boldsymbol{w}:\, \boldsymbol{w}^\top \boldsymbol{w}=1}{\arg\min} \operatorname{tr}\left(\boldsymbol{X}^\top - \boldsymbol{X}^\top \boldsymbol{w}\boldsymbol{w}^\top\right)\left(\boldsymbol{X} - \boldsymbol{w}\boldsymbol{w}^\top \boldsymbol{X}\right) \tag{1.6}$$

$$= \underset{\boldsymbol{w}:\, \boldsymbol{w}^\top \boldsymbol{w}=1}{\arg\min} \operatorname{tr}(\underbrace{\boldsymbol{X}^\top \boldsymbol{X}}_{\text{independent of } \boldsymbol{w}} - 2\boldsymbol{X}^\top \boldsymbol{w}\boldsymbol{w}^\top \boldsymbol{X} + \boldsymbol{X}^\top \boldsymbol{w} \underbrace{\boldsymbol{w}^\top \boldsymbol{w}}_{=1} \boldsymbol{w}^\top \boldsymbol{X}) \tag{1.7}$$

$$= \underset{\boldsymbol{w}:\, \boldsymbol{w}^\top \boldsymbol{w}=1}{\arg\min} \operatorname{tr}\left(-\boldsymbol{X}^\top \boldsymbol{w}\boldsymbol{w}^\top \boldsymbol{X}\right) \tag{1.8}$$

$$= \underset{\boldsymbol{w}:\, \boldsymbol{w}^\top \boldsymbol{w}=1}{\arg\max} \operatorname{tr}(\underbrace{\boldsymbol{w}^\top \boldsymbol{X}\boldsymbol{X}^\top \boldsymbol{w}}_{\text{scalar}}) \tag{1.9}$$

$$= \underset{\boldsymbol{w}:\, \boldsymbol{w}^\top \boldsymbol{w}=1}{\arg\max} \boldsymbol{w}^\top \boldsymbol{X}\boldsymbol{X}^\top \boldsymbol{w} \tag{1.10}$$

In Equation 1.4 we used the Frobenius norm

$$\|\boldsymbol{A}\|_F^2 \coloneqq \sum_i \sum_j A_{ij}^2 = \operatorname{tr}\left(\boldsymbol{A}^\top \boldsymbol{A}\right) = \operatorname{tr}\left(\boldsymbol{A}\boldsymbol{A}^\top\right) \tag{1.11}$$

This is equivalent to maximizing the objective

$$\hat{\boldsymbol{w}} = \underset{\boldsymbol{w}}{\arg\max} \frac{\boldsymbol{w}^\top \boldsymbol{X}\boldsymbol{X}^\top \boldsymbol{w}}{\boldsymbol{w}^\top \boldsymbol{w}} \tag{1.12}$$

From linear algebra, you might recall that $\frac{\boldsymbol{w}^\top \boldsymbol{X}\boldsymbol{X}^\top \boldsymbol{w}}{\boldsymbol{w}^\top \boldsymbol{w}}$ is the Rayleigh quotient of $\boldsymbol{X}\boldsymbol{X}^\top$. It is maximized for $\boldsymbol{w}$ the eigenvector of largest eigenvalue of $\boldsymbol{X}\boldsymbol{X}^\top$, hence we can simply solve the eigenvalue problem to find the first principal component.

We can also show this by using Lagrange multipliers

$$\mathcal{L}\left(\boldsymbol{w}, \lambda\right) = \boldsymbol{w}^\top \boldsymbol{X}\boldsymbol{X}^\top \boldsymbol{w} - \lambda(\boldsymbol{w}^\top \boldsymbol{w} - 1) \tag{1.13}$$

$$\frac{\partial \mathcal{L}\left(\boldsymbol{w}, \lambda\right)}{\partial \boldsymbol{w}} = 2\boldsymbol{w}^\top \boldsymbol{X}\boldsymbol{X}^\top - 2\lambda \boldsymbol{w}^\top \stackrel{!}{=} 0 \tag{1.14}$$

$$\Rightarrow \boldsymbol{X}\boldsymbol{X}^\top \boldsymbol{w} = \lambda \boldsymbol{w} \tag{1.15}$$

Here we used the numerator-layout notation for matrix calculus and have

$$\frac{\partial \boldsymbol{A}\boldsymbol{x}}{\partial \boldsymbol{x}} = \boldsymbol{A} \tag{1.16}$$

$$\frac{\partial \boldsymbol{x}^\top \boldsymbol{A}}{\partial \boldsymbol{x}} = \boldsymbol{A}^\top \tag{1.17}$$

Be careful when computing the eigenvectors, eigensolvers don't always sort by the magnitude of the eigenvalues by default.

The next highest principal component can be calculated by projecting the data in the subspace orthogonal to the first principal component. Iteratively we can then calculate

$$\boldsymbol{X}^{k+1} = \boldsymbol{X} - \sum_{i=1}^{k} \boldsymbol{w}_i \boldsymbol{w}_i^\top \boldsymbol{X} \tag{1.18}$$

The approach of minimizing the sum of squared residuals is equivalent to finding the subspace in which the data has maximum spread

$$\hat{\boldsymbol{w}} = \underset{\boldsymbol{w}:\, \boldsymbol{w}^\top \boldsymbol{w}=1}{\arg\max} \|\boldsymbol{w}^\top \boldsymbol{X}\|_2^2 \tag{1.19}$$

$$= \underset{\boldsymbol{w}:\, \boldsymbol{w}^\top \boldsymbol{w}=1}{\arg\max} \boldsymbol{w}^\top \boldsymbol{X}\boldsymbol{X}^\top \boldsymbol{w} \tag{1.20}$$

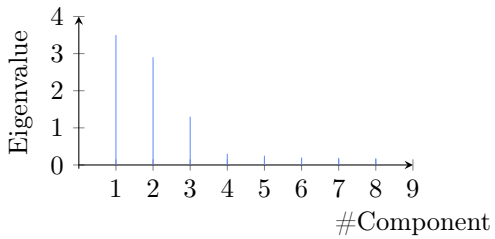


Figure 1.4: Scree plot: Eigenvalues sorted by their magnitude

Comments

- A *scree plot* like in Figure 1.4 can be used to visualize the (relative) magnitude of the eigenvalues. It can be used to determine how many principal components are relevant and should be included.
- The scaling in step 1 is very important! In the case where all features have a natural relationship we can scale accordingly, otherwise we normalize to unit variance and zero mean. PCA maximizes variance (because it projects data onto the direction which maximizes variance), therefore we have to scale the relative variance of the features so that it has meaning. For example, think of a feature of temperature values given in kelvin or in millikelvin. The same feature would have a lot higher variance in absolute numbers if given in millikelvin and could dominate the PCA a lot more compared to if it given in kelvin.
- Could we do non-linear PCA? Possible approaches might be:
  - Transform data so that it is linear (kernel PCA).
  - Add more dimensions and hope that it is linear there.
  - Fit linear curves to the data in short intervals where the data is linear (principal curves).
- PCA is sensitive to outliers, which becomes clear when looking at Figure 1.3 again and thinking of the sum of squared residuals as the potential of springs pulling the line into its position. Outliers also pull on the line and may have a strong influence.

Literature

- Trevor Hastie, Robert Tibshirani, and Jerome H Friedman, "The elements of statistical learning: data mining, inference, and prediction". 2009, [HTF09]
- Kevin P Murphy, *Probabilistic machine learning: an introduction*. 2022, Chap. 20 [Mur22]

1.5 Robust estimators

Real world data usually has some outliers and doesn't perfectly follow an assumed statistical model. Unfortunately, the result of some estimators like the mean, can significantly change with even one bad outlier. Robust estimators are estimators that are stable against this behavior. Here, we take an exemplary look at estimators of location, which estimate the true value ("location") of a quantity given a set of measurements.

The *maximum likelihood estimator* (MLE) estimates parameters of an assumed probability density function f given data $\boldsymbol{x}$. This is done by maximizing the likelihood of the observed data given the estimated parameters

$$\hat{\boldsymbol{\mu}}_{MLE} \coloneqq \arg\max_{\boldsymbol{\mu}} \prod_{i=1}^{n} f(\boldsymbol{x}_i; \boldsymbol{\mu}). \tag{1.21}$$

We can equivalently minimize the logarithm of the likelihood, giving us the (negative) log-likelihood estimator

$$\hat{\boldsymbol{\mu}} \coloneqq \arg\min_{\boldsymbol{\mu}} \sum_{i=1}^{n} \underbrace{-\log f(\boldsymbol{x}_i; \boldsymbol{\mu})}_{\coloneqq \rho(\boldsymbol{x}_i, \boldsymbol{\mu})}. \tag{1.22}$$

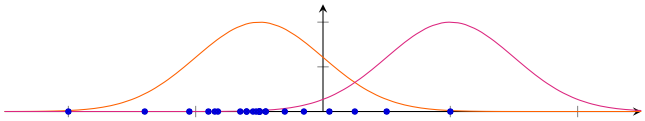


Figure 1.5: Distribution of measured data (blue) and two possible underlying Gaussian distributions (orange and pink).

For example, we try to estimate the location of the distribution shown in Figure 1.5. We assume the underlying distribution to be Gaussian with unit variance, $f \sim \exp -\frac{1}{2}(x-\mu)^2$. The resulting estimator is simply the mean, the objective of maximizing the likelihood/minimizing the log-likelihood is equivalent to minimizing L_2-loss:

$$\hat{\mu} = \arg\min_{\mu} \sum_{i=1}^{n} -\log \exp -\frac{1}{2}(x_i - \mu)^2 = \arg\min_{\mu} \frac{1}{2} \sum_{i=1}^{n} (x_i - \mu)^2 \tag{1.23}$$

To find the minimum we set the derivative of the RHS to zero and find that $\hat{\mu} = \frac{1}{n} \sum_{i=1}^{n} x_i$.

The stability of an estimator can be quantified by using the empirical influence function and the influence function. The empirical influence function (EIF) is defined as

$$EIF(\boldsymbol{x}, \{\boldsymbol{x}_i : i = 1, \ldots, \mathfrak{n}\}, T)_i \coloneqq \mathfrak{n}\Big(\underbrace{\underbrace{T\big(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_{i-1}, \boldsymbol{x}, \boldsymbol{x}_{i+1}, \ldots, \boldsymbol{x}_{\mathfrak{n}}\big)}_{\text{perturbed sample, i-th observation at x}}}_{\text{perturbed estimator}} - \underbrace{\underbrace{T\big(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_{\mathfrak{n}}\big)}_{\text{unperturbed sample}}}_{\text{unperturbed estimator}} \Big) \tag{1.24}$$

$$\tag{1.25}$$

Here, T is our estimator and $\mathfrak{n}$ is our number of samples. The empirical influence function measures how displacing sample x_i affects the estimator.

For the influence function (IF), we use the probability density distribution F instead of a set of samples.

$$IF(\boldsymbol{x}, T, F) \coloneqq \lim_{t \to 0^+} \frac{\overbrace{T\big((1-t)\,F + t\delta\boldsymbol{x}\big)}^{\text{estimator on perturbed distribution}} - \overbrace{T\big(F\big)}^{\text{estimator on original distribution}}}{\underbrace{t}_{\text{strength of perturbation}}} \tag{1.26}$$

For the perturbed distribution, some "mass" t was subtracted and then added at δx. From the ρ defined in equation 1.22, we can estimate the IF using

$$\psi(\boldsymbol{x}; \boldsymbol{\mu}) = \frac{\partial \rho(\boldsymbol{x}, \boldsymbol{\mu})}{\partial \boldsymbol{x}} \sim IF(\boldsymbol{x}). \tag{1.27}$$

We can consider ρ as the analogue of a potential and ψ/the influence function as the analogue of a force to get a more intuitive understanding of the effect of outliers and robustness. We compare different estimators (MLE with different underlying probability density functions) in Table 1.1.

Robust estimators are underused!

Summary

- PCA finds lossy compression in terms of "best" linear subspace. To calculate the principal component we have to solve the eigenvalue problem. The relative scaling of the features matters a lot.
- Robust estimators, like the median, are estimator that are stable against outliers.

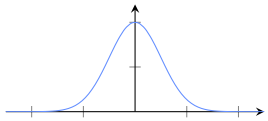
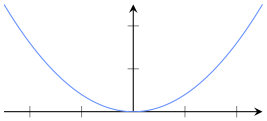
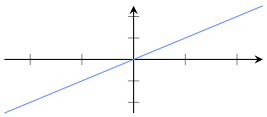
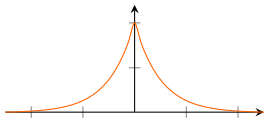
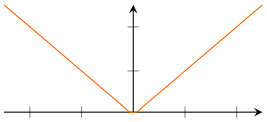
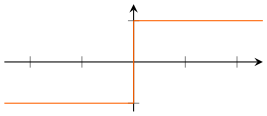
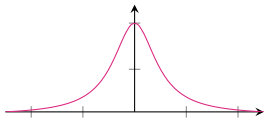
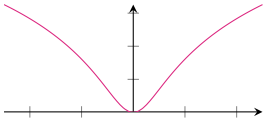
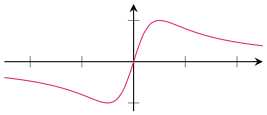
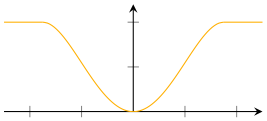
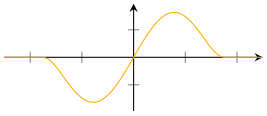
estimator, minimized loss	$f(\boldsymbol{x};\boldsymbol{\mu})$ assumed underlying probability density function	$\rho(\boldsymbol{x};\boldsymbol{\mu}) = -\log(f(\boldsymbol{x};\boldsymbol{\mu}))$ "potential"	$\psi(\boldsymbol{x};\boldsymbol{\mu}) = \frac{\mathrm{d}\rho(\boldsymbol{x};\boldsymbol{\mu})}{\mathrm{d}\boldsymbol{x}} \sim$ IF "force"
mean, L_2-loss, least squares	 $\sim \exp(-\frac{1}{2}(x-\mu)^2)$ Gaussian distribution	 $\sim +\frac{1}{2}(x-\mu)^2$	 "perfect spring": the further away the outlier, the bigger the influence ⇒ NOT ROBUST
median, L_1-loss	 $\sim \exp(-\lvert x-\mu\rvert)$ Laplace distribution	 $\sim \lvert x-\mu\rvert$	 it doesn't matter how far away the outlier is ⇒ ROBUST
	 $\sim \frac{1}{1+x^2}$ Cauchy distribution	 $\sim \log(1+x^2)$	 far away outliers contribute less ⇒ VERY ROBUST
Tukey	does not exist		 far away outliers don't contribute at all ⇒ VERY ROBUST

Table 1.1: Maximum likelihood estimators with different underlying probability density functions. The Influence Function gives us the "force" that an outlier has on our estimate. We can see that the estimators respond very differently to outliers.

2 Nonlinear dimension reduction

2.1 Motivation

Dimension reduction in general is used to visualize data (reduce to $\mathfrak{p}$=2 or $\mathfrak{p}$=3) and/or lossily compress data, as already described in the PCA chapter. Nonlinear dimension reduction techniques can be used in cases where PCAs performance doesn't suffice, as shown in Figure 2.1. Here, the UMAP algorithm manages to cluster the images of handwritten digits in the MNIST data set without needing their labels, while applying PCA does not work as good. Nonlinear dimension reduction techniques like UMAP can also be useful in physics, as shown in Example 2.1

Formally, given high-dimensional samples $\boldsymbol{x}_i \in \mathbb{R}^{\mathfrak{p}^*}$ with distances $d^*_{ij} = ||\boldsymbol{x}_i - \boldsymbol{x}_j||$ we want to find a low-dimensional embedding $\boldsymbol{z}_i \in \mathbb{R}^{\mathfrak{p}}$, $\mathfrak{p} << \mathfrak{p}^*$, with distances $d_{ij} = ||\boldsymbol{z}_i - \boldsymbol{z}_j||$.

(a) PCA (b) UMAP

Figure 2.1: From [MHM20]: PCA and UMAP applied to the MNIST data set. The different digits are represented by different colors. UMAP separates the data into clusters a lot more clearly compared to PCA. It is important to note that during the embedding the class labels (here corresponding to the colors) were not used!

Example 2.1: Search for Dark Matter

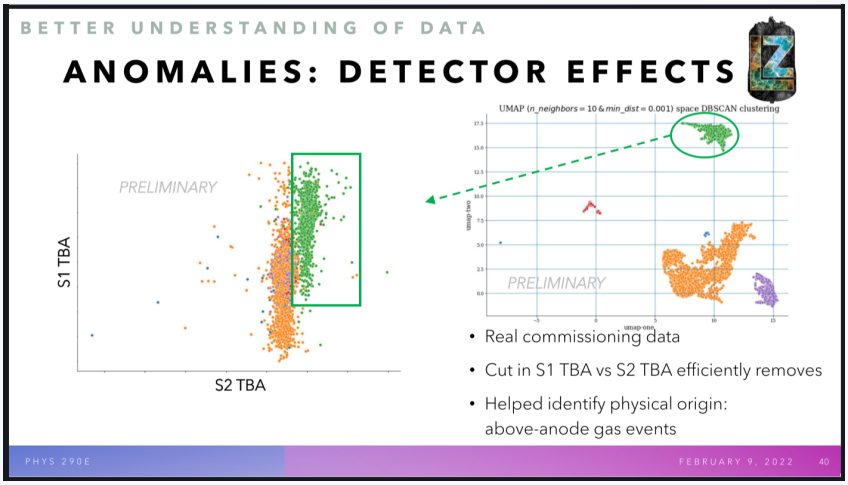


Figure 2.2: From [Kra22]: Right plot: UMAP is applied to data from a dark matter detector to reduce the data set to two dimensions, umap-one and umap-two. Several clusters emerge. Left plot: Two of the original, detected features with data points colored to show their cluster membership according to UMAP.

Nonlinear dimension reduction can be used to analyze anomalies in a data set of dark matter detector events. UMAP can be used to identify clusters of data points exhibiting the same anomaly. Comparing these different clusterings in some of the original features can help to identify causes for the anomalous behavior.

2.2 First approach: preserve distances

The first generation of nonlinear dimension reduction algorithms attempted to preserve all distances between samples in the new low dimensional embedding, $d^*_{ij} \stackrel{!}{=} d_{ij}$. A number of different objectives that minimize the mismatch between d_{ij} and d^*_{ij} are listed in Table 2.1. An example of CCA and Sammon's nonlinear mapping of a sphere can be seen in Figure 2.3. It shows the difference that we get from the objectives weighting the close distances d_{ij} (CCA) or d^*_{ij} (Sammon) more.

Name	Objective
Metric multidimensional scaling (MDS) "raw stress", [Kru64]	$\arg\min_{\{\boldsymbol{z}_i\}} \sum_{ij}(d^*_{ij} - d_{ij})^2$
"s-stress", [TYD77]	$\arg\min_{\{\boldsymbol{z}_i\}} \sum_{ij}({d^*_{ij}}^2 - {d_{ij}}^2)^2$
Sammon's nonlinear mapping [Sam69]	$\arg\min_{\{\boldsymbol{z}_i\}} \sum_{ij}(d^*_{ij} - d_{ij})^2 \frac{1}{d^*_{ij}}$
Curvilinear component analysis (CCA) [DH97]	$\arg\min_{\{\boldsymbol{z}_i\}} \sum_{ij}(d^*_{ij} - d_{ij})^2 - f(d_{ij})$

Table 2.1: Different objectives for finding the optimal new embedding $\{\boldsymbol{z}_i\}$. Here, as above, $d_{ij} = ||\boldsymbol{z}_i - \boldsymbol{z}_j||$ and $d^*_{ij} = ||\boldsymbol{x}_i - \boldsymbol{x}_j||$. f is some function to weight close distances d_{ij} more, as also done in Sammon's nonlinear mapping with d^*_{ij}. In practice often $I(d_{ij} \leq \Theta)$ is used, with Θ some maximum distance.

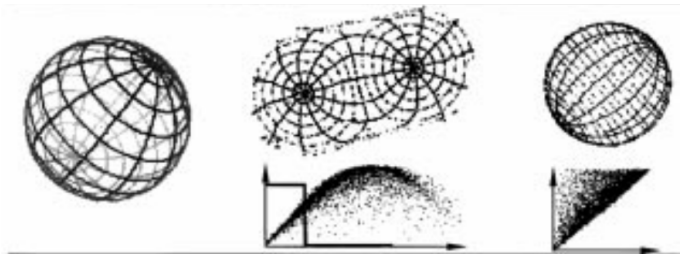


Figure 2.3: From [DH97]: A sphere in $\mathbb{R}^3$ (left) is mapped to $\mathbb{R}^2$ with CCA (middle) and Sammon nonlinear mapping (right).

2.3 Graphs

The more recent generation of dimension reduction techniques rely on k-nearest neighbor graphs. A graph is pair $G = (\mathcal{V}, \mathcal{E})$, of a set of *vertices* $\mathcal{V}$ and a set of *edges* $\mathcal{E}$. We use it to describe objects (represented by the vertices) and their relations to each other (represented by the edges). Edges can be directed or undirected. We can for example represent nearest neighborhood in a graph, as shown in Figure 2.4. We connect two vertices if one of them is within the k-nearest neighbors (by distance) of the other.

k-nearest neighbor graphs can help find clusterings. An example from [MHV07] is shown in Figure 2.5.

(a) Directed nearest neighbor graph. The arrows/directed edges each point from a sample to its $k = 1$ nearest neighbors. We see that nearest neighborhood is an asymmetric quantity.

(b) Undirected/symmetric nearest neighbor graph. The lines/edges connect samples if one is within the $k = 1$ nearest neighbors of the other.

Figure 2.4: Nearest neighbor graphs of three samples.

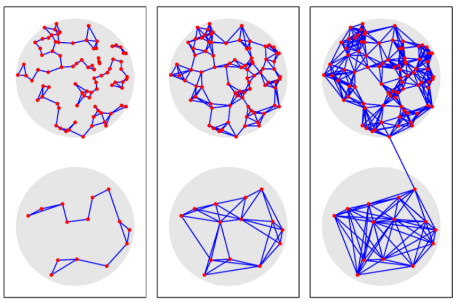


Figure 2.5: From [MHV07]: Data points from two clusters (gray dots). With the undirected kNN graph, for $k = 2$ (left) and $k = 4$ (middle), we find the two separate clusters. The clusters are not separated for $k = 8$ (right).

2.4 Second approach: preserve k-nearest neighbor graph

The more recent generation of dimension reduction techniques try to only preserve the k-nearest neighbor structure of the old, high-dimensional embedding $\boldsymbol{x}_i$ in the new, low-dimensional embedding $\boldsymbol{z}_i$. This can be done by following these steps:

1. Determine the symmetric k-nearest neighbor graph of the high-dimensional embedding
2. Define a force field over the new randomly initialized low-dimensional embeddings such that
   - there is attractive potential between $\boldsymbol{z}_i$, $\boldsymbol{z}_j$ if $\boldsymbol{x}_i$, $\boldsymbol{x}_j$ are symmetric nearest neighbors $\sim (k\mathfrak{n})$ terms
   - there is a (weak) repulsive potential between all/selected pairs $\boldsymbol{z}_i$, $\boldsymbol{z}_j$ $\sim (\mathfrak{n}^2)$ terms, to prevent overlapping of clusters

   $\implies$ the resulting system is frustrated, meaning the force terms are in conflict
3. Find local energy minimum

By using different potentials in step 2, we get different techniques. For some of them, their performance on the MNIST data set is shown in Table 2.2.

The techniques in Table 2.2 are defined with different potentials, but it is possible to tune their parameters in a way that two techniques behave similarly, as shown in Figure 2.6 for t-SNE and in Figure 2.7 for UMAP. Therefore, we have a continuum of techniques that can be obtained by scaling attraction and repulsion.

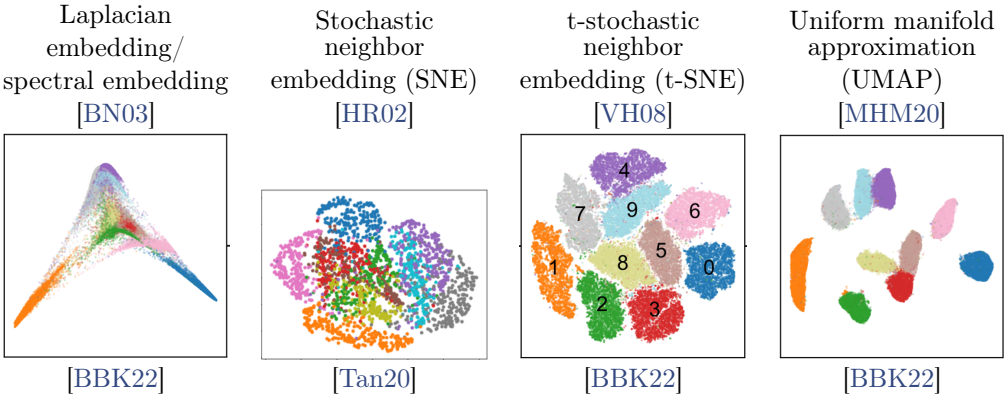


Table 2.2: Performance of different embeddings on the MNIST data set. The different colors represent the different digits from 0 to 9.

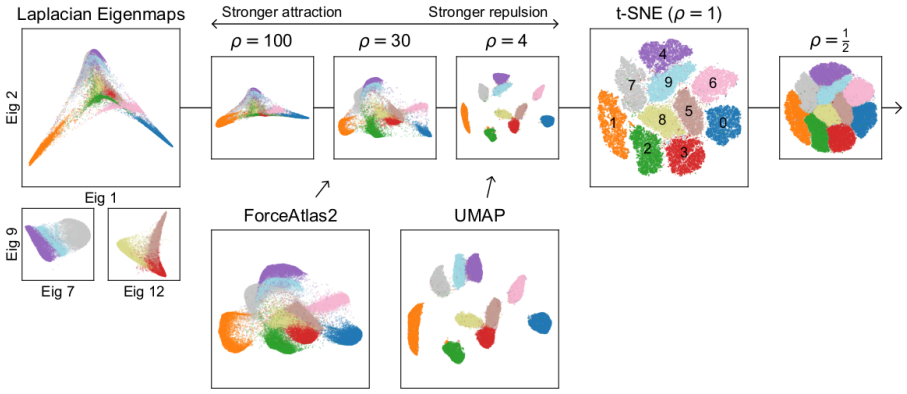


Figure 2.6: From [BBK22]: The t-SNE embedding is applied to the MNIST data set and the attractive forces are multiplied by a factor ρ. For different ρ, this corresponds/behaves similarly to different other embeddings, like UMAP for $\rho \approx 4$. The different colors represent the different digits from 0 to 9.

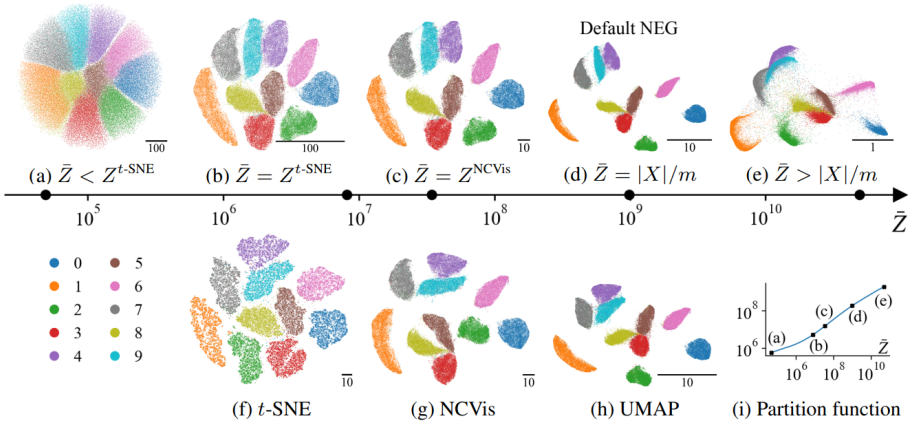


Figure 2.7: From [Dam+22]: Neg-t-SNE embedding with different partition functions $\overline{Z}$. For different $\overline{Z}$, this corresponds/behaves similarly to different other embeddings, like UMAP or t-SNE. The different colors represent the different digits from 0 to 9.

2.5 Comparison of the techniques

We can compare the potentials that are used in the different techniques in subsection 2.4. We can even define potentials for the techniques of the first generation in subsection 2.2 to be able to compare all the discussed approaches. The potentials are shown in Table 2.3. We observe that even small changes in the potentials can give very different embeddings.

The choice of potential has drastic consequences on the quality of the embedding. It is still an open question which techniques are the best for a particular case. All techniques are unfortunately prone to create artefactual structures, for example as shown in Figure 2.8, especially if not done well.

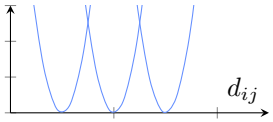
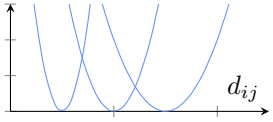
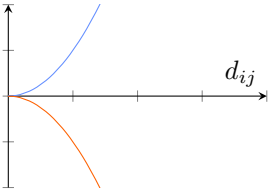
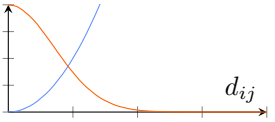
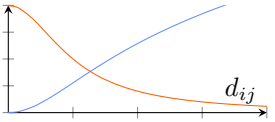
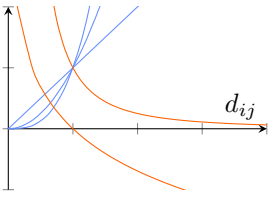
Name	Potential
MDS with raw stress	
Sammon's nonlinear mapping	
Laplacian embedding	
SNE	
Neg-t-SNE	
force-directed layout/ energy-based graph drawing	

Table 2.3: Qualitative potentials associated with the different nonlinear dimension reduction techniques, with attractive potentials in blue and repulsive ones in orange.

2.6 Speedup of computation

The calculation of the forces for the second approach has a high computational cost of $\mathcal{O}\left(N^2\right)$ for direct force summation. By borrowing from physics and using the Barnes-Hut algorithm as

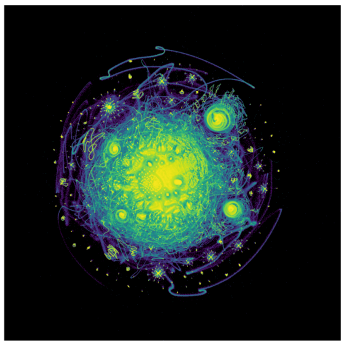


Figure 2.8: From [MHM20]: Structure created by UMAP in a data set of integers that are represented by high dimensional sets of binary vectors, each representing the divisibility of the integer by a prime number. The swirls are artefacts of the algorithm. A later application, [Kob20], of t-SNE on the dataset only generated clusters of integers, based on the number of prime factors.

described in Example 2.2, the computational cost can be reduced to $\mathcal{O}(N \log(N))$. For t-SNE, the speedup using Barnes-Hut instead of direct summation is usually three or more orders of magnitude (depends on the data).

Example 2.2: Barnes-Hut Algorithm for the N-Body Problem

The Barnes-Hut algorithm was originally developed for solving the N-body problem in astrophysics.

1. Construct a quadtree (tree with 4 leaves on a node) for 2D problem or octtree (tree with 8 leaves on a node) for 3D problem:
   (a) Divide the spacial dimensions successively into halves (so for a two (tree) dimensional space in "quadrants" ("octands")) until all partitions of space hold only one single sample. Simultaneously construct a tree whose leaves correspond to the partitions with only one sample and nodes to the larger partitions.
   (b) Traverse tree in postorder (as shown in Figure 2.9, start at leftmost child; always move up the tree if all children of the current node are accounted for; else go to leftmost unaccounted child), and compute for each node

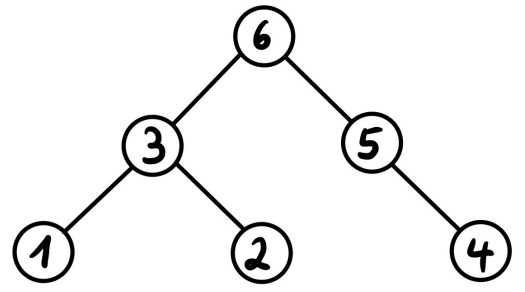


Figure 2.9: Scheme of postorder tree traversal.

   i. the number of samples it, or all of its children, contain
   ii. and the center of mass of the node and all its children.

2. Loop over all samples: For each sample start by considering the root node. If node diameter (width of quadrant/octant)/ node distance (between sample and center of mass of current node) ratio is smaller than a given threshold, compute the force on the sample originating from an effective particle at the center of mass of the node with the total mass of all contained samples. Else repeat the process for all children.

The resulting tree structure for a 2D example is visualized in Figure 2.10. The speedup of computation is due to the fact that the forces on one sample from far away groups of

samples can be simplified by using the mass and center of mass of the far away group, instead of calculating all interactions explicitly.

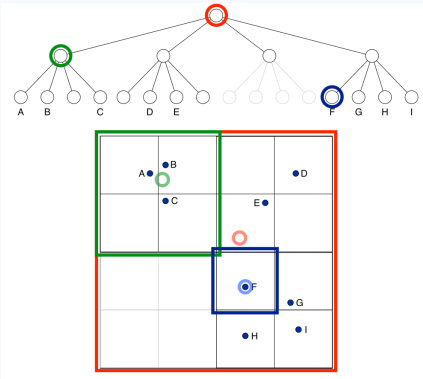


Figure 2.10: From [Van14]: Division of 2D space and corresponding quadtree for some samples.

Summary

Nonlinear dimension reduction techniques are powerful, but ill-understood and prone to create artefactual structure, especially if not done well. Small changes in the potentials of the techniques give very different embeddings. We can use ideas from N-body simulations to speed up the computation.

Literature

- Jan Niklas Böhm, Philipp Berens, and Dmitry Kobak, "Attraction-repulsion spectrum in neighbor embeddings". 2022, [BBK22]

3 Density estimation

The task of *density estimation* is the following: Given a list of (empirical) samples, we want to reconstruct the unknown distribution p^* our data were sampled from. In this chapter, we present the two methods of histograms and kernel density estimation. We start with the well-known histograms, which, although they may seem trivial at first, involve some intricacy in higher dimensions.

3.1 Histograms

A histogram splits the feature space into discrete regions and simply counts the relative amount of samples in each region to estimate the density. This discretization introduces a lot of rather arbitrary choices. In the one-dimensional case, the number line is usually split into intervals (bins) of the same size. This comes along with the two hyperparameters of the bin size and the bin offset.

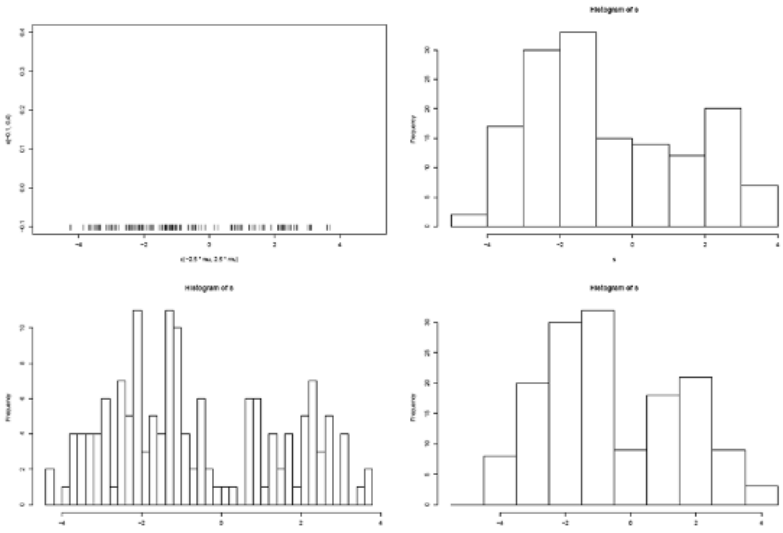


Figure 3.1: Three different one-dimensional histograms describing the same data using different bin sizes and offsets.

Depending on their value, we can obtain many distinct histograms, see Figure 3.1. As a result, we might miss important structures if the bin size is too big or conversely overfit on the data if the bin size is too small for the given distribution.

The problem of arbitrary choices gets even worse in higher dimensions. In contrast to the one-dimensional case, there isn't a single natural way to divide the feature space into discrete regions. In 2D, we could use rectangles or hexagons, for instance. If the feature space is tree-dimensional, we must choose between many layouts such as cubes, a body-centered cubic lattice (bcc) or a face-centered cubic lattice (fcc) known from crystallography. We are also forced to decide on a rotation of our regions, as there is no natural choice for the orientation of, e.g., hexagons.

In dimensions bigger than three, we can't draw the regions anymore. We can only try to represent shapes in 3D. Figure 3.2 shows a visualization of an eight-dimensional hypercube. It has the same radial distribution as the real hypercube $[-1, 1]^8$. This means that the integrated volume up to a certain radius shows the same behavior for both objects. We see that the $2^8 = 256$ vertices are sharp spikes in the representation while most of the volume is in the middle of the object. The interior of the hypercube appears almost empty. Finding a tessellation of higher-dimensional spaces that has a radial distribution similar to the sphere is a hard problem on its own.

3.2 Kernel density estimation

As a motivation, we will first consider the seemingly unrelated problem of the *heat equation*

$$\begin{cases} \frac{\partial}{\partial t} u(\boldsymbol{x}, t) - \Delta u(\boldsymbol{x}, t) = 0, & \boldsymbol{x} \in \mathbb{R}^{\mathfrak{p}} \times (0, \infty) \\ \frac{\partial}{\partial t} u(\boldsymbol{x}, 0) = u_0(\boldsymbol{x}), & \boldsymbol{x} \in \mathbb{R}^{\mathfrak{p}}. \end{cases} \tag{3.1}$$

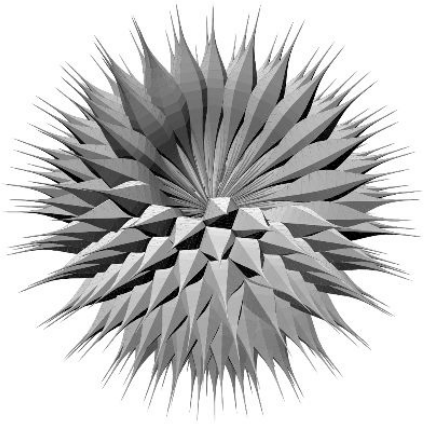


Figure 3.2: Three-dimensional representation of an eight-dimensional hypercube. It has the same radial volume distribution and the same number of vertices as the hypercube. A small fraction of the mass lies near a vertex while most of the interior is void. Figure from [HA03].

The solution u is the heat distribution at time t given the initial temperature distribution u_0. The heat equation is a special case of the Fokker-Planck equation which we will meet again in the chapter on diffusion models, see Equation 19.3. A Green's function of the linear partial differential equation can be obtained by application of the Fourier transform. With the Green's function, we can write the solution of the heat equation as the convolution

$$u(\boldsymbol{x}, t) = \frac{1}{(4\pi t)^{\mathtt{p}/2}} \int_{\mathbb{R}^d} u_0(\boldsymbol{y}) \exp\left(-\frac{\|\boldsymbol{x} - \boldsymbol{y}\|^2}{4t}\right) \mathrm{d}y. \tag{3.2}$$

We will use this idea to define the *kernel density estimation* (KDE). Again, given a set of samples $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_{\mathtt{n}} \in \mathbb{R}^{\mathtt{p}}$, we want to estimate the density p^* that generated the samples. A primitive estimate would be to use the so-called "nail-board" $\frac{1}{\mathtt{n}} \sum_{i=1}^{\mathtt{n}} \delta(\boldsymbol{x} - \boldsymbol{x}_i)$, where δ is the Dirac delta distribution. Obviously, this estimate is not useful in practice as it just represents our

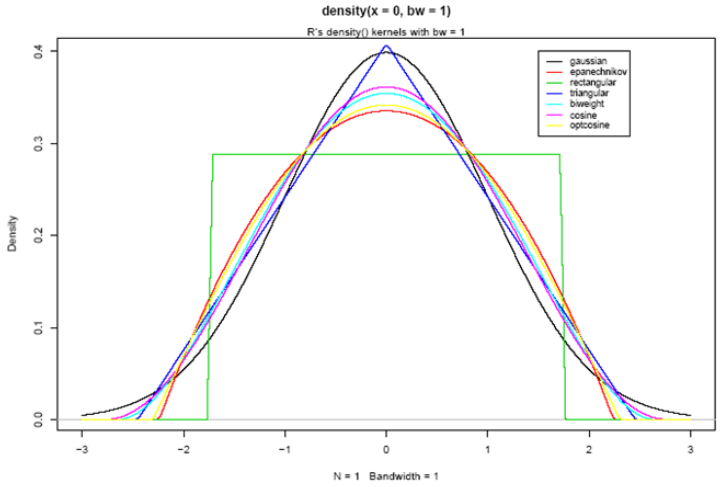


Figure 3.3: Plot of different kernel functions for bandwidth $w = 1$.

empirical realization of the distribution. We therefore interpret the obtained samples as sources of "heat" at the location $\boldsymbol{x}_i$ and let the system evolve for some time. Instead of the above Green's

function, we convolve the nail-board with a general *kernel* function k to define a density estimator

$$\hat{p}(\boldsymbol{x}) = \frac{1}{\mathfrak{n}} \int \sum_{i=1}^{\mathfrak{n}} \delta(\boldsymbol{y} - \boldsymbol{x}_i)\, k(\boldsymbol{x} - \boldsymbol{y}; w) \mathrm{d}y \tag{3.3}$$

$$= \frac{1}{\mathfrak{n}} \sum_{i=1}^{\mathfrak{n}} k(\boldsymbol{x} - \boldsymbol{x}_i; w), \tag{3.4}$$

where we introduced a circumflex to distinguish the estimator $\hat{p}$ from the true density p^*. The kernel function k has an additional hyperparameter w that we call the *bandwidth*. It scales the width of the kernel function and corresponds to the time parameter t in the heat equation. For $\hat{p}$ to be a proper density, the kernel function has to be normalized and non-negative, i.e.,

$$\int k(\boldsymbol{x}; w) \mathrm{d}x = 1 \quad \text{and} \quad k(\boldsymbol{x}; w) \geq 0. \tag{3.5}$$

It is also desirable that the kernel function is differentiable (used in Equation 4.4 for mean shift) and has a finite support. A few examples of kernel functions are plotted in Figure 3.3. Figure 3.4 shows the result of kernel density estimation in one dimension for different bandwidths. We see

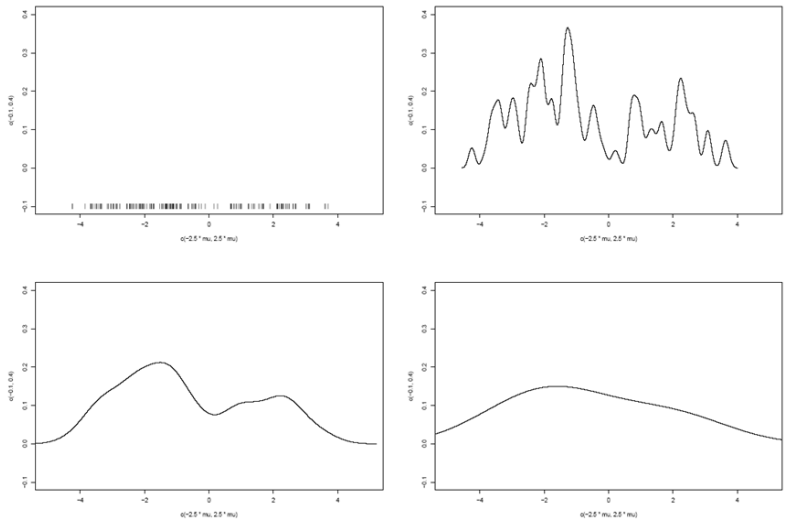


Figure 3.4: Kernel density estimation in one dimension for different bandwidths. Too narrow a bandwidth leads to overfitting, too wide one to oversmoothing.

that a small bandwidth leads to a very spiky estimate while a large bandwidth leads to a very smooth estimate that might miss important structures. Possible sharp features like edges or cusps in the true density will be smoothed out if we are using a large bandwidth.

The optimal bandwidth can also be distinct for different regions of the feature space. In a low density region, we want to use a large bandwidth to avoid overfitting while we rather want a small bandwidth in high density regions to avoid oversmoothing. If the feature space has more than one dimension, we should either use a different bandwidth for each dimension or scale the feature space such that all dimensions have the same variance. Otherwise, we might overfit the samples in one dimension while oversmoothing in another dimension.

Advantages

- KDE eliminates some arbitrary choices required for histograms.
- The density estimate is differentiable if the kernel is.

Disadvantages

- The proper choice of bandwidth can be difficult.

- Tricky in high dimensions.
- Slow unless N-body techniques or approximate nearest-neighbor search are used.

Before we can prove a statement about the expected value of KDE, we want to elaborate a bit on the concept of random variables.

3.3 Random variables, expectation value

Statistics deals with *random events*. The results are different under equal circumstances. A natural language to describe these results is set theory, see Table 3.1. A *random variable* X is a function that maps every trial from the sample space Ω onto a real number,

$$\begin{aligned} X &: \Omega \to \mathbb{R} \\ &\omega \mapsto X(\omega). \end{aligned} \tag{3.6}$$

The name "random variable" is infelicitous, since we are dealing with a purely deterministic function. Only the argument ω is random. The *probability distribution* of X is the function

$$\begin{aligned} p &: \mathbb{R} \to \mathbb{R} \\ &x \mapsto p(x) = p(X = x). \end{aligned} \tag{3.7}$$

Probability theory term	**Symbol**	**Interpretation**	**Set theory term**
trial, outcome	ω	possible result of a single event	element
event	A, B, C	set of outcomes	set
sample space	Ω	all possible outcomes	basic set
impossible event	$\emptyset$	event cannot occur	empty set
complementary event	A^c	A does not occur	complementary set
combination	$A \cup B$	A or B occur (inclusive)	union
A and B	$A \cap B$	A as well as B occur	intersection

Table 3.1: Symbols used in probability theory and their analogues in set theory.

We note that we can easily extend random variables and distributions to higher dimensions by replacing $\mathbb{R}$ with $\mathbb{R}^\mathfrak{p}$ in the above definitions. The probability distribution of a discrete random variable is called *probability mass function* and the probability distribution of a continuous random variable is called *probability density function*.

The *expectation value* of a random variable X characterizes the location of a distribution. In the discrete case, it is defined as

$$\mathbb{E}[X] = \sum_{i=1}^{\mathfrak{n}} x_i \, p(X = x_i) = \sum_{i=1}^{\mathfrak{n}} x_i \, p(x_i) \tag{3.8}$$

and in the continuous case, it is defined as

$$\mathbb{E}[X] = \int_{-\infty}^{\infty} x \, p(x) \, \mathrm{d}x. \tag{3.9}$$

For an arbitrary function $r : \mathbb{R} \to \mathbb{R}$, the expectation value is given by

$$\begin{aligned} \mathbb{E}[r(X)] &= \sum_{i=1}^{\mathfrak{n}} r(x_i) \, p(x_i) \qquad \text{or} \\ \mathbb{E}[r(X)] &= \int_{-\infty}^{\infty} r(x) \, p(x) \, \mathrm{d}x. \end{aligned} \tag{3.10}$$

The expectation value is a linear operator, i.e., for any two (possibly dependent) random variables X, Y and any two real numbers a, b we have

$$\mathbb{E}[aX + bY] = a \, \mathbb{E}[X] + b \, \mathbb{E}[Y]. \tag{3.11}$$

We can use these properties to calculate the expected value of KDE. The expectation value is taken over the random variables $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_\mathfrak{n}$ while $\boldsymbol{x}$ is fixed. We can make this explicit by writing the random variables in the index of the expectation value,

$$\mathbb{E}[\hat{p}(\boldsymbol{x})] = \mathbb{E}_{\boldsymbol{x}_1,\ldots,\boldsymbol{x}_\mathfrak{n}}[\hat{p}(\boldsymbol{x})] = \mathbb{E}_{\boldsymbol{x}_1,\ldots,\boldsymbol{x}_\mathfrak{n}}\left[\frac{1}{\mathfrak{n}}\sum_{i=1}^{\mathfrak{n}} k(\boldsymbol{x}-\boldsymbol{x}_i; w)\right]. \tag{3.12}$$

Using the linearity of the expectation, we obtain

$$\mathbb{E}[\hat{p}(\boldsymbol{x})] = \frac{1}{\mathfrak{n}}\sum_{i=1}^{\mathfrak{n}} \mathbb{E}_{\boldsymbol{x}_i}\left[k(\boldsymbol{x}-\boldsymbol{x}_i; w)\right]. \tag{3.13}$$

We assume that all observations follow the same distribution, and thus, the expectation value is the same for all $\boldsymbol{x}_i$. We can therefore write

$$\mathbb{E}[\hat{p}(\boldsymbol{x})] = \frac{1}{\mathfrak{n}}\mathfrak{n}\,\mathbb{E}_{\boldsymbol{x}_1}[k(\boldsymbol{x}-\boldsymbol{x}_1; w)] = \int k(\boldsymbol{x}-\boldsymbol{x}_1; w)\, p^*(\boldsymbol{x}_1)\,\mathrm{d}\boldsymbol{x}_1. \tag{3.14}$$

It turns out that the expectation value of KDE is the convolution of the true, unknown density p^* with the kernel k. Note that this result does *not* depend on the number of samples $\mathfrak{n}$. It does depend on the kernel function and the bandwidth w, though.

We close this chapter by noting that a ransom variable can have many forms. Besides the aforementioned random scalars and vectors, we can also have

- random sequences (discrete in space/time)
- random fields (discrete in space/time)
- stochastic processes (continuous in space/time)
- random elements (trees, graphs, tessellations, you name it…)

Literature

- J.H. Conway, N.J.A. Sloane, and Bannai, *Sphere Packings, Lattices and Groups.* 2013, [CSB13] (More on E8)
- Russell Lyons and Yuval Peres, *Probability on Trees and Networks.* 2016, [LP16]

4 Cluster analysis

Cluster analysis deals with the problem of finding natural groupings among data samples. For example a possible task would be to look at data acquired from a particle accelerator experiment and group samples together that were produced by the same type of particle. Importantly, cluster analysis is again an unsupervised method which in this context means that we do not have access to labels, like the particle type in the example above, but purely the explanatory variables. Apart from that, clustering also has many other applications, including summarizing complex data distributions by a few representative samples (e.g., cluster centers) or lossy compression used, for example, in the transmission of phone signals.

Clustering algorithms can be classified in the following ways

- *crisp* vs. *fuzzy*: In crisp clustering each data sample is assigned to one unique cluster whereas in fuzzy clustering samples may belong to several clusters. In the latter case, each sample is assigned a score between 0 and 1 per cluster which measures how similar this sample is compared to other samples in the same cluster.
- *flat* vs. *hierarchical*: "Flat" means that we have a single clustering of the dataset whereas "hierarchical" refers to having several different clusterings which are typically nested. That means that the data in each cluster may be divided into further clusters so that hierarchical clustering can be summarized in a tree where each node represents one cluster.

Clustering also has a few limitations: It is often used in situations where the ground truth of the data labels is not known. Although this makes it certainly more flexible, it also makes it harder to benchmark. We will explore methods to quantify the quality of a clustering in section 5. In addition, most clustering algorithms strongly depend on one or several hyperparameters which determine, e.g., the spacial scale or the number of clusters. If we consider, for example, the task of clustering objects in the universe, depending on which scale parameter we choose for our clustering, we could end up with either individual atoms, {planets, stars, etc.} or whole galaxies.

Example 4.1: Unveiling phase transitions with Machine Learning

Clustering can for example be used to analyze phase transitions in solid state models as shown in [Can+19]. In the paper the authors investigate the so called "axial next-nearest-neighbor Ising" (ANNNI) model which is given by the following Hamiltonian:

$$H = -J \sum_{j=1}^{N} (\sigma_j^z \sigma_{j+1}^z - \kappa \sigma_j^z \sigma_{j+2}^z + g \sigma_j^x) \tag{4.1}$$

where σ_j is the spin at lattice site j and J, κ, g are constants. From theoretical analysis it is known that this model shows three main phases, namely ferromagnetic, paramagnetic and antiphase/floating phases. To investigate these phases numerically they compute the set of correlation functions

$$\{\langle \sigma_i^x \sigma_j^x \rangle, \langle \sigma_i^y \sigma_j^y \rangle, \langle \sigma_i^z \sigma_j^z \rangle\}, \; j > i, \; i \in \{1, ..., N-1\} \tag{4.2}$$

for many different combinations of the parameters κ and g. They then use k-means clustering (described below) with $k = 3$ to assign each observation set to one of three clusters. In this manner, these clusters can be interpreted as containing observations which stem from states of the system in the same phase. With this analysis the authors find the phase boundaries (in $\kappa - g$ space) shown in Figure 4.1 as the black lines. Comparing these with the theoretical predictions shown as the blue and orange lines we see that the clustering approach indeed yields very good results, considering that the algorithm does not use any prior physical knowledge.

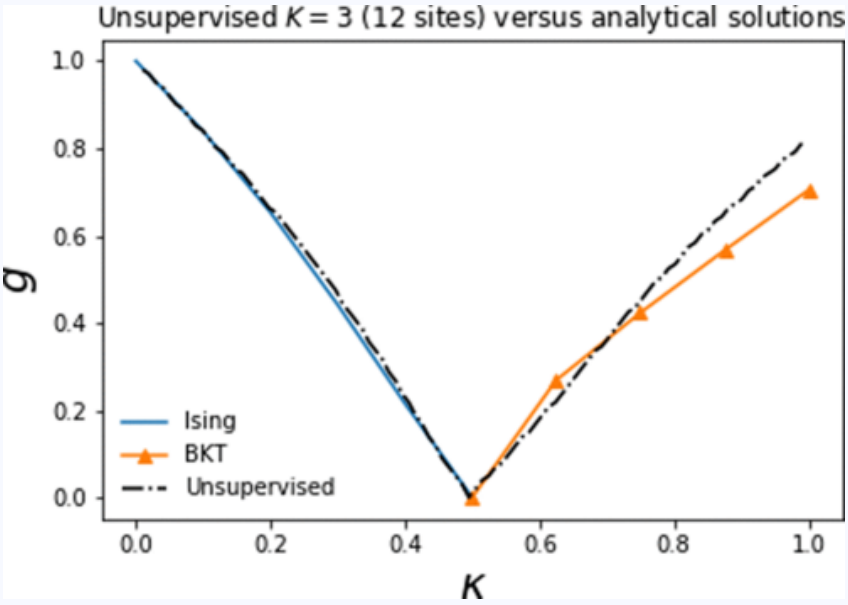


Figure 4.1: Phase boundaries of the ANNNI model as obtained by the clustering approach marked in black compared to the theoretical predictions marked in blue and orange. Figure from [Can+19].

4.1 Mean shift clustering

Most clustering algorithms rely on some sense of nearness in feature space. Samples which are separated only by a small distance are intuitively more likely to belong to the same cluster. In mean shift clustering we make use of the results of the last chapter. The idea is to approximate the initial data distribution using kernel density estimation (KDE) and then iteratively moving each sample along the direction of steepest ascent according to the estimated density. This method is in general also known as *gradient ascent*. After some time, the samples will converge to a number of different points namely the local maxima of the density. One then assigns each sample to a cluster depending on which maximum they converged to. The resulting trajectories of samples are exemplarily shown in Figure 4.2. It is clear that this procedure crucially depends on the choice of bandwidth parameter of the KDE. For a small bandwidth the density will have a lot of local maxima whereas for a large bandwidth there will only be a few.

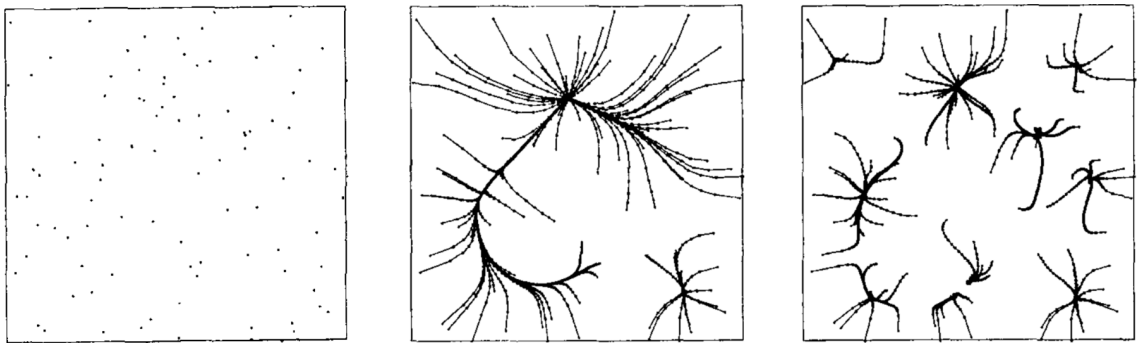


Figure 4.2: Trajectories of samples in mean shift clustering depending on the bandwidth parameter of the KDE. The left panel shows the initial data distribution. The middle panel shows the trajectories for a large bandwidth whereas the right panel depicts the results for a low bandwidth. Figure adapted from [Che95]

Let us formalize this idea: We can describe a small update of a sample in the gradient ascent

procedure as

$$\boldsymbol{x}_j^{t+1} = \boldsymbol{x}_j^t + \alpha \left(\frac{\partial \rho(\boldsymbol{x})}{\partial \boldsymbol{x}} \bigg|_{\boldsymbol{x}=\boldsymbol{x}_j^t} \right)^T, \tag{4.3}$$

where $\alpha > 0$ is the step size and $\rho(\boldsymbol{x})$ is the estimated density of the initial distribution. Using the results of the previous chapter we can write:

$$\frac{\partial \rho(\boldsymbol{x})}{\partial \boldsymbol{x}} = \frac{1}{\mathfrak{n}} \sum_{i=1}^{\mathfrak{n}} \frac{\partial k(\|\boldsymbol{x} - \boldsymbol{x}_i\|)}{\partial \boldsymbol{x}}, \tag{4.4}$$

with kernel k. As an example, we look at the Epanechnikov kernel given by

$$k(\|\boldsymbol{x} - \boldsymbol{x}_i\|) \sim \begin{cases} 1 - (\|\boldsymbol{x} - \boldsymbol{x}_i\|)^2, & \|\boldsymbol{x} - \boldsymbol{x}_i\| < 1 \\ 0, & \text{else} \end{cases} \tag{4.5}$$

and illustrated in Figure 4.3.

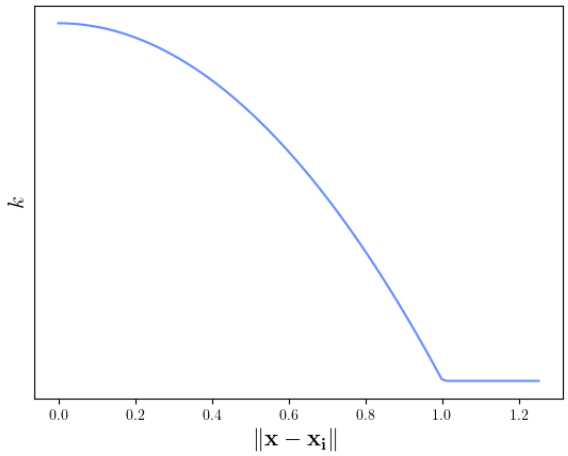


Figure 4.3: Epanechnikov kernel.

The gradient of the Epanechnikov kernel is given by

$$\frac{\partial k(\|\boldsymbol{x} - \boldsymbol{x}_i\|)}{\partial \boldsymbol{x}} \sim \begin{cases} -2(\boldsymbol{x}^T - \boldsymbol{x}_i^T), & \|\boldsymbol{x} - \boldsymbol{x}_i\| < 1 \\ 0, & \text{else} \end{cases}. \tag{4.6}$$

We can then plug this into equations 4.4 and 4.3 to obtain

$$\boldsymbol{x}_j^{t+1} = \boldsymbol{x}_j^t + \frac{2\alpha}{n} \sum_{i:\|\boldsymbol{x}_i - \boldsymbol{x}_j^t\| < 1} (\boldsymbol{x}_i - \boldsymbol{x}_j^t) \tag{4.7}$$

The last term is a vector which points into the direction of the local mean, where "local" is referring to the fact that only samples within a radius of 1 are considered. Thus, rather than taking small steps in the direction of the local mean one can directly move the point of interest to the local mean. This is not equivalent to the above procedure since the local neighborhood of the sample might change in some steps, however the two approaches should in general yield very similar results. We can also look at the convergence properties of the sequence in Equation 4.7, while keeping the local

neighborhood fixed. The fixed point of this sequence x_j^* is given by

$$0 = \frac{2\alpha}{n} \sum_{i:\|\boldsymbol{x}_i - \boldsymbol{x}_j^0\|<1} (x_i - x_j^*) \tag{4.8}$$

$$\iff \sum_{i:\|\boldsymbol{x}_i - \boldsymbol{x}_j^0\|<1} x_i = \sum_{i:\|\boldsymbol{x}_i - \boldsymbol{x}_j^0\|<1} x_j^* = x_j^* \sum_{i:\|\boldsymbol{x}_i - \boldsymbol{x}_j^0\|<1} 1 \tag{4.9}$$

$$\iff x_j^* = \frac{\sum\limits_{i:\|\boldsymbol{x}_i - \boldsymbol{x}_j^0\|<1} x_i}{\sum\limits_{i:\|\boldsymbol{x}_i - \boldsymbol{x}_j^0\|<1} 1}, \tag{4.10}$$

which is again the local mean. The procedure of repeatedly shifting every sample to its local mean according to the original data distribution until all samples have converged is called *mean shift*. As we have seen, this algorithm arises if we choose the Epanechnikov kernel for our KDE. For a general kernel one update takes the following form:

$$x_j^{t+1} = \frac{\sum\limits_i x_i f(\|\boldsymbol{x}_i - \boldsymbol{x}_j^t\|)}{\sum\limits_i f(\|\boldsymbol{x}_i - \boldsymbol{x}_j^t\|)} \tag{4.11}$$

with some weighting function f. For the Epanechnikov kernel we have

$$f(\|\boldsymbol{x}_i - \boldsymbol{x}_j^t\|) = I_{\{x_i:\|\boldsymbol{x}_i - \boldsymbol{x}_j^t\|<1\}} \tag{4.12}$$

We can summarize the advantages and disadvantages of the algorithm as follows:

Advantages

- We do not need to specify the absolute number of clusters.
- It is very simple to implement.

Disadvantages

- The KDE ascent procedure can move samples very far which leads to diffusive clusters.
- Mean shift has problems with elongated clusters as they might contain several local maxima and thus break down into multiple smaller clusters.
- The algorithm is slow unless N-body techniques are used.
- Unless a variable bandwidth is used, we implicitly assume a similar density across the clusters which might not be the case.

4.2 k-means clustering

The basic idea of k-means, also known as the Linde-Buzo-Gray algorithm, is to approximate the data with k cluster centers and assign each data point to the cluster of its nearest cluster center. In more detail, given the location of the cluster centers $\{\boldsymbol{\mu}_l\}$, $l \in \{1, ..., k\}$ the cluster assignment of observation i denoted as $c(i) \in \{1, ..., k\}$ is given by

$$c(i) = \arg\min_l \|\boldsymbol{\mu}_l - \boldsymbol{x}_i\|. \tag{4.13}$$

Of course the optimal position of cluster centers is not known a priori. Assuming that we already have cluster assignments $c(i)$ for all observations we can determine the position of cluster centers by setting them equal to the center of mass of their associated observations,

$$\mu_l = \arg\min_{\mu_l} \sum_{i:\ c(i)=l} \|\boldsymbol{x}_i - \mu_l\|^2. \tag{4.14}$$

In k-means, one starts with a random initialization of the cluster centers and then applies the above steps in an alternating fashion (see Algorithm 4.1). An example for the application of the algorithm is shown in Figure 4.4.

Algorithm 4.1: k-means

repeat
  assign observation to the closest cluster center;
  move each cluster to the center of mass of its associated observations;
until *convergence*;

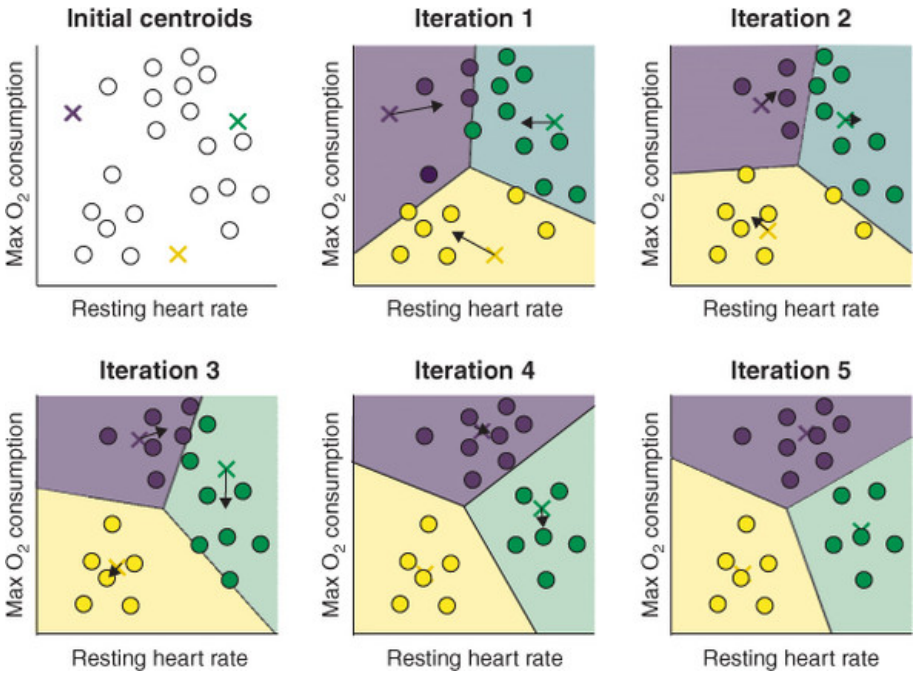


Figure 4.4: Example for application of k-means. The crosses show the position of the cluster centers, whereas the circles are the data points including their cluster assignment marked by their color. Figure from [Rhy20].

The advantages and disadvantages can be summarized as follows:

Advantages

- It is very simple.
- The algorithm is guaranteed to converge (to a local optimum).

Disadvantages

- In contrast to mean-shift, the number of clusters is not determined by the algorithm but has to be chosen in advance.
- It is sensitive to outliers.
- The local optimum depends on the initialization.

A property which can be good as well as bad depending on the situation is that k-means has a strong bias in favor of compact clusters.

4.3 k-means++

k-means++ is an adaptation of k-means which specifically targets its disadvantage of initialization dependence. Instead of choosing the initialization randomly the following scheme is used:

1. Choose the first cluster center randomly.

2. For all observations, compute the squared distance to the closest cluster center.
3. Weigh all observations with this squared distance and sample the next cluster center randomly according to those weights.
4. Go to step 2 if there are not yet k cluster centers.
5. Apply conventional k-means.

Literature

- Yizong Cheng, "Mean shift, mode seeking, and clustering". 1995, [Che95]
- Kevin P Murphy, *Probabilistic machine learning: an introduction.* 2022, Chap. 21.3 [Mur22]

5 Comparing partitions/clustering results

Given a set $\mathcal{D}$, a *partition* or *clustering* is a set of sets $\mathcal{C} = \{C_1, C_2, \ldots, C_k\}$ with $\bigcup_i C_i = \mathcal{D}$ and $C_i \cap C_j = \emptyset$ for $i \neq j$. An example taken from particle physics is the tagging of jets, where the jets are clustered according to the type of particle which initiated the jet, i.e., bottom, charm and lighter quarks (up, down, strange). We present two ways to visualize the comparison of clusterings.

Confusion matrix and contingency table

- In classification problems (supervised learning), the predictions are known labels. The corresponding visualization in Figure 5.1 is called *confusion matrix*. It has a meaningful ordering of columns.

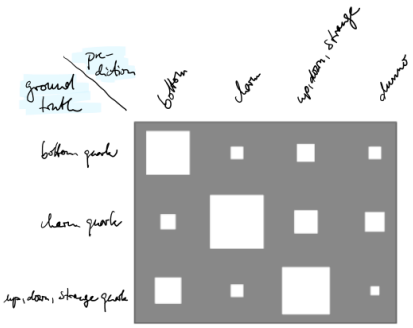


Figure 5.1: Example of a confusion matrix to compare the results of jet flavor tagging to the ground truth. The size of the squares in this *Hinton diagram* corresponds to the number of samples which match both the row and column class.

- In cluster analysis (unsupervised learning), clusters have arbitrary names. The visualization in Figure 5.2 is known as *incidence table* or *contingency table*. It has no preferred ordering of rows or columns.

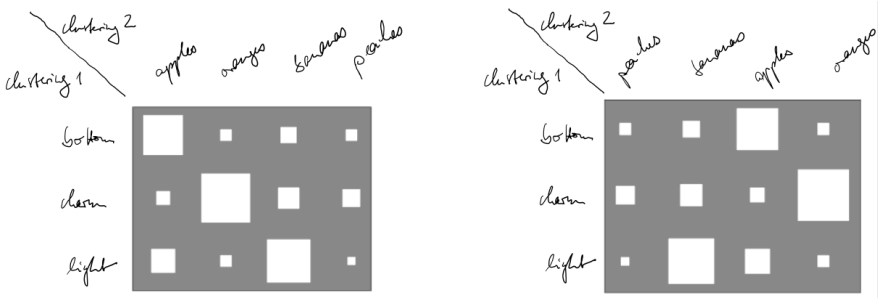


Figure 5.2: Example of two incidence/contingency tables using the same partitions. The order of the columns has no meaning.

In a confusion matrix, the diagonal entries show the accordance of the two partitions. This property is lost in the incidence table, as the order of the columns and rows does not have any meaning. Comparing Figure 5.2 with Figure 5.3, we can see that an incidence table nevertheless allows us to make statements on the agreement of different partitions. The latter contains squares that are all about the same size and thus show a lower correspondence of the different classes. Some quantitative methods to do so will be discussed below.

5.1 RAND index

The RAND index (named after William Rand) is a way to compare clusterings without ground truth. From a first perspective, we focus on pairs of observations. This is equivalent to considering the edges of the complete graph, where the vertices are the data points. We define the RAND

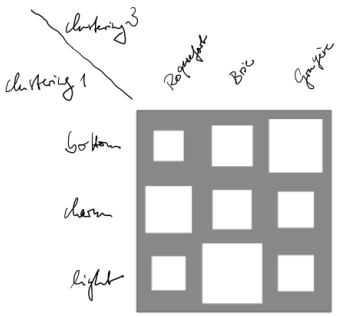
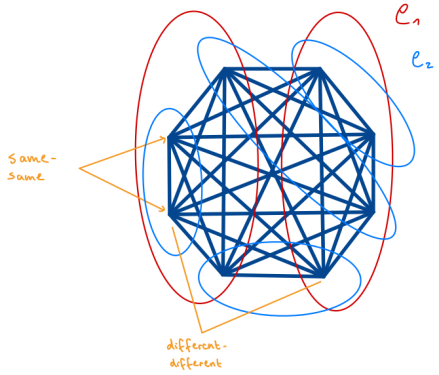


Figure 5.3: Incidence table of clusterings 1 and 3.

index as

$$\begin{aligned} \text{RAND} &= \frac{\#\left(\begin{array}{c}\text{pairs of observations that are in the}\\ \text{same cluster in both partitions}\end{array}\right) + \#\left(\begin{array}{c}\text{pairs of observations that are in}\\ \text{different clusters in both partitions}\end{array}\right)}{\#\left(\text{all pairs}\right)} \\ &= \frac{\#\text{agreements}}{\#\text{agreements} + \#\text{disagreements}}. \end{aligned} \tag{5.1}$$

The calculation of the RAND index is visualized in Figure 5.4. Eight data points are displayed as a complete graph. The number of agreements is the sum of lines connecting two points that are



	b_1	b_2	b_j	$\cdots$	b_C	
a_1	n_{11}	n_{12}	n_{1j}	$\cdots$	n_{1C}	$n_{1\cdot}$
a_2	n_{21}	n_{22}	n_{2j}	$\cdots$	n_{2C}	$n_{2\cdot}$
a_i	n_{i1}	n_{i2}	n_{ij}	$\cdots$	n_{iC}	$n_{i\cdot}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$
a_R	n_{R1}	n_{R2}	n_{Rj}	$\cdots$	n_{RC}	$n_{R\cdot}$
	$n_{\cdot 1}$	$n_{\cdot 2}$	$n_{\cdot j}$	$\cdots$	$n_{\cdot C}$	N

Figure 5.4: Left: Two partitions $\mathcal{C}_1$ and $\mathcal{C}_2$ on a set of eight points displayed as a complete graph. Right: General bivariate contingency table.

in the same cluster in both partitions ('same-same') and the lines connecting two points that are in different clusters in both partitions ('different-different'). The RAND index is the number of agreements divided by the number of all pairs.

We can also look at the RAND index from a different perspective. In this approach, we focus on single observations and analyze the bivariate contingency table defined by two partitions (see Figure 5.4 on the right). From the table, we can read of the number of pairs that signify disagreement. Samples in cluster a_1 can appear in clusters b_1 or b_2, for example. The corresponding number of disagreeing pairs is $n_{11} \cdot n_{12}$. We obtain the number of agreements if we subtract these cross-terms from the total number of pairs,

$$\begin{aligned} \#\text{agreements} &= \binom{N}{2} - \frac{1}{2}\left(\sum_{i=1}^{R} n_{i\cdot}^2 + \sum_{j=1}^{C} n_{\cdot j}^2 - 2\sum_{i=1}^{R}\sum_{j=1}^{C} n_{ij}^2\right) \\ &= \binom{N}{2} + 2\sum_{i=1}^{R}\sum_{j=1}^{C}\binom{n_{ij}}{2} - \left(\sum_{i=1}^{R}\binom{n_{i\cdot}}{2} + \sum_{j=1}^{C}\binom{n_{\cdot j}}{2}\right) \\ &= \underbrace{\sum_i \sum_j \binom{n_{ij}}{2}}_{\#\text{same-same}} + \underbrace{\frac{1}{2}\sum_i \sum_j n_{ij} \cdot \left(\sum_{k\neq i}\sum_{l\neq j} n_{kl}\right)}_{\#\text{different-different}}, \end{aligned} \tag{5.2}$$

where we used the abbreviations $n_{i\cdot} = \sum_{j=1}^{C} n_{ij}$, $n_{\cdot j} = \sum_{i=1}^{R} n_{ij}$ and $N = \sum_{i,j} n_{ij}$. We note that the RAND index is invariant under a permutation of rows or columns. We see that it is not true that cells on the main diagonal represent agreement and cells off the main diagonal represent disagreement.

Multivariate distributions

In the following, we define discrete distributions describing two random variables A and B. The extension to the continuous case is straightforward.

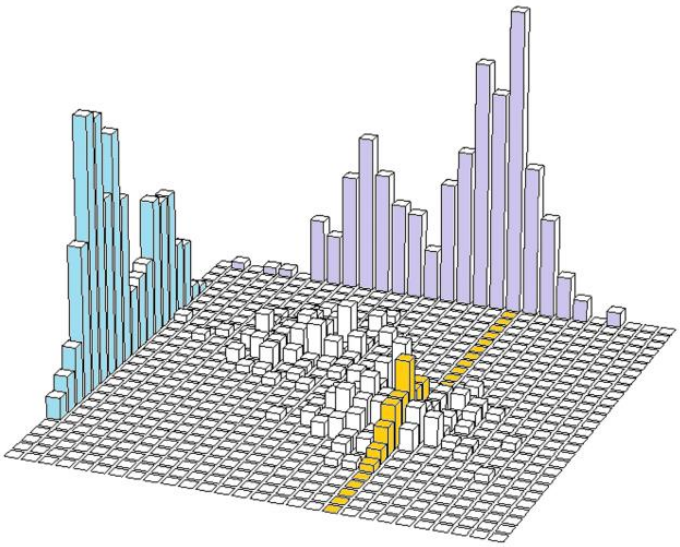


Figure 5.5: Plot of a bivariate discrete distribution. The marginals are shown in blue and violet, a (scaled) conditional is indicated in yellow.

- *Joint distribution* $P(A, B)$ has normalization $\sum_a \sum_b P(A = a, B = b) = 1$
- *Marginal distribution* $P(A) = \sum_b P(A, B = b)$ has normalization $\sum_a P(A = a) = 1$
  $P(B) = \sum_a P(A = a, B)$ has normalization $\sum_b P(B = b) = 1$
- *Conditional distribution* $P(A|B = b) = \frac{P(A,B=b)}{P(B=b)}$ has normalization $\sum_a P(A = a|B = b) = 1$
- Two distributions are *independent* if $P(A, B) = P(A) \cdot P(B)$. Note that independent is not the same as uncorrelated.

Note that in the above example, both random variables seem to map to ordinal scale. But all mathematical statements above also hold for any permutation of realization/also hold for random variables with categorical outcomes.

5.2 Shannon/Gibbs entropy

The Shannon information content is defined as

$$h(x) = \log_2 \frac{1}{P(x)} = -\mathrm{lb}\, P(x) = -\,\mathrm{ld}\, P(x), \tag{5.3}$$

where we used different notations for the binary logarithm. The idea behind this definition is that the occurrence of a rare event carries much surprise and thereby has large information content. Inversely, the occurrence of a certain event is no surprise and has zero information content. The information content is not arbitrary, but is the sole function that satisfies a number of reasonable axioms.

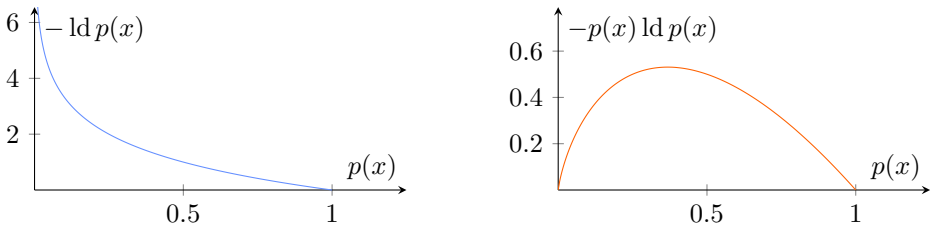


Figure 5.6: Left: Shannon information content. Right: Single contribution to the Entropy.

The *Entropy* is the expected information content or uncertainty of X:

$$H(X) = \sum_x P(X = x) \operatorname{ld} \frac{1}{P(X = x)} = -\sum_x P(x) \operatorname{ld} P(x). \tag{5.4}$$

We define $H(0) = 0$ and note that the slope of the entropy at the origin is infinite (see also Figure 5.6), which is important for entropy regularization. The Entropy is non-negative, $H(X) \geq 0$, and is maximized for a uniform distribution. We define the entropy for joint and conditional distributions and provide the chain rule for entropy.

- *Joint entropy:*

$$H(X, Y) = -\sum_x \sum_y p(x, y) \operatorname{ld} p(x, y) \tag{5.5}$$

  If X and Y are independent, the joint entropy simplifies to $H(X, Y) = H(X) + H(Y)$.

- *Conditional entropy:*

$$\begin{aligned} H(X|Y = y) &= -\sum_x p(x|y) \operatorname{ld} p(x|y), \\ H(X|Y) &= \sum_y p(y) H(X|y) \overset{p(x|y)p(y)=p(x,y)}{=} -\sum_x \sum_y p(x, y) \operatorname{ld} p(x|y) \end{aligned} \tag{5.6}$$

- *Chain rule* for entropy:

$$H(X, Y) = H(X|Y) + H(Y) = H(Y|X) + H(X) \tag{5.7}$$

  The chain rule follows from $p(x, y) = p(x|y)p(y)$ and $\sum_x p(x, y) = p(y)$, such that

$$\begin{aligned} H(X, Y) &= -\sum_{x,y} p(x, y) \left(\operatorname{ld} p(x|y) + \operatorname{ld} p(y)\right) && (5.8) \\ &= -\sum_{x,y} p(x, y) \operatorname{ld} p(x|y) - \sum_y p(y) \operatorname{ld} p(y) && (5.9) \\ &= H(X|Y) + H(Y). && (5.10) \end{aligned}$$

  An analogous proof for the second expression is obtained if X and Y are interchanged.

5.3 Mutual information

The Mutual information, defined as

$$I(X; Y) \coloneqq H(X) - H(X|Y), \tag{5.11}$$

can be described as the average reduction in uncertainty about X that results from learning the values of Y. Another way to think about it is as the average amount of information that X conveys about Y [Mac03]. Using the above chain rule for entropy (Equation 5.7), we can write

$$I(X; Y) = H(X) + H(Y) - H(X, Y). \tag{5.12}$$

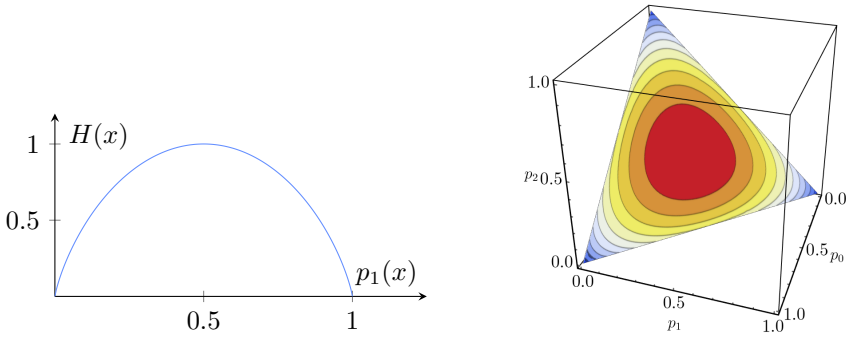


Figure 5.7: Left: Entropy of a distribution with two possible outcomes, $p_1(x) + p_2(x) = 1$. Right: Entropy for a distribution with three possible outcomes $p_1(x) + p_2(x) + p_3(x) = 1$. Figure from [glS20].

This way we see that the mutual information is symmetric, $I(X;Y) = I(Y;X)$. As will be elaborated below, the mutual information is non-negative, $I(X;Y) \geq 0$. We use $P(x) = \sum_y P(x,y)$ to further rewrite

$$\begin{aligned} I(X;Y) &= -\sum_x P(x) \operatorname{ld} P(x) - \sum_y P(y) \operatorname{ld} P(y) + \sum_x \sum_y P(x,y) \operatorname{ld} P(x,y) \\ &= \sum_x \sum_y P(x,y) \operatorname{ld} \frac{P(x,y)}{P(x) \cdot P(y)} \\ &= D_{\mathrm{KL}} \left(P(x,y) \,\middle\|\, P(x) \cdot P(y) \right), \end{aligned} \tag{5.13}$$

which means that the mutual information is the KL divergence between joint and product of marginals. Here, the *Kullback-Leibler divergence* or *relative entropy* is defined as

$$D_{\mathrm{KL}}(P \,\|\, Q) = \sum_x P(x) \operatorname{ld} \frac{P(x)}{Q(x)} = -H(X) - \sum_x P(x) \operatorname{ld} Q(x). \tag{5.14}$$

It is a statistical distance that measures how different the distributions P and Q are. The last term $-\sum_x P(x) \operatorname{ld} Q(x)$ is known as *cross-entropy*, *logistic loss* or just *log loss*. One can show that the Kullback-Leibler divergence is non-negative,

$$D_{\mathrm{KL}}(P \,\|\, Q) = \sum_x P(x) \left(\operatorname{ld} P(x) - \operatorname{ld} Q(x) \right) \geq 0. \tag{5.15}$$

This is also known as the Gibbs-inequality and implies the non-negativity of the mutual information.

Information Diagrams

In the last paragraph, we defined many statistical quantities. To obtain a visual representation, these quantities can be displayed in information diagrams. It is important to mention that the components are not sets, and it is wrong to call these diagrams Venn diagrams. The diagrams merely summarize additive relations. From Figure 5.8, we can read of the relations

$$H(X,Y) = H(X) + H(Y) - I(X;Y) = H(X|Y) + H(Y|X) + I(X;Y). \tag{5.16}$$

We would like to warn that information diagrams are misleading for more than two random variables. Let us underline this warning by a quote from [Mac03]:

> No other 'three-term entropies' will be defined. For example, expressions such as $I(X;Y;Z)$ and $I(X|Y;Z)$ are illegal.

We now rephrase the above in the setting of clusterings. We are given a data set $\mathcal{D}$ and two partitions $\mathcal{C} = \{C_1, C_2, \ldots, C_k\}$ and $\mathcal{C}' = \{C'_1, C'_2, \ldots, C'_{k'}\}$. If we pick a random point form $\mathcal{D}$, what is the uncertainty about its cluster membership? We can think of $\mathcal{C}$ and $\mathcal{C}'$ as random

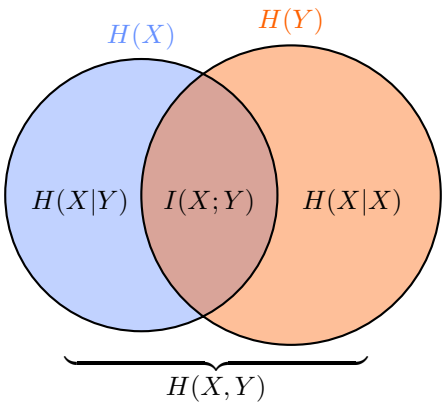


Figure 5.8: Example of an information diagram.

variables assigning a cluster member to a random point from $\mathcal{D}$, so that we can calculate their entropy in the following. We define

$$n_i = |C_i|,\ P_i = \frac{n_i}{|\mathcal{D}|} \quad \text{and} \quad n_i' = |C_i'|,\ P_i' = \frac{n_i'}{|\mathcal{D}|} \tag{5.17}$$

to obtain

$$H(\mathcal{C}) = -\sum_{1=1}^{k} P_i \operatorname{ld} P_i, \quad H(\mathcal{C}') = -\sum_{1=1}^{k} P_i' \operatorname{ld} P_i'. \tag{5.18}$$

Using the definitions

$$n_{ij} = |C_i \cap C_j'| \text{ and } P_{ij} = \frac{n_{ij}}{|\mathcal{D}|}, \tag{5.19}$$

the mutual information between both partitions can be written as

$$I(\mathcal{C};\mathcal{C}') = \sum_{i=1}^{k}\sum_{j=1}^{k'} P_{ij} \operatorname{ld} \frac{P_{ij}}{P_i \cdot P_j}. \tag{5.20}$$

To give a better understanding of the mutual information between two partitions, we quote from [Mei03]:

> Intuitively, we can think of $I(\mathcal{C};\mathcal{C}')$ in the following way: we are given a random point in $\mathcal{D}$. The uncertainty about its cluster in $\mathcal{C}'$ is measured by $H(\mathcal{C}')$. Suppose now that we are told which cluster the point belongs to in $\mathcal{C}$. How much does this knowledge reduce the uncertainty about $\mathcal{C}'$? This reduction in uncertainty, averaged over all points, is equal to $I(\mathcal{C},\mathcal{C}')$.

Therefor a high value of the mutual information corresponds to a good agreement between two clusterings and vice versa.

5.4 Variation of information

The last quantitative comparison of partitions we will look at is the variation of information (VI/VoI). It was proposed by Marina Meilă in [Mei03] and is given by

$$\begin{aligned} \operatorname{VoI}(\mathcal{C},\mathcal{C}') &= H(\mathcal{C}|\mathcal{C}') + H(\mathcal{C}|\mathcal{C}') \\ &= H(\mathcal{C}) + H(\mathcal{C}') - 2I(\mathcal{C};\mathcal{C}'). \end{aligned} \tag{5.21}$$

The corresponding information diagram is given in Figure 5.9. An interesting fact about the VoI is that it is a proper distance, meaning that it is positive-definite, symmetric and satisfies the triangle inequality. In particular, the VoI is only zero if the partitions $\mathcal{C}$ and $\mathcal{C}'$ are equal. This property justifies its usability in practical applications.

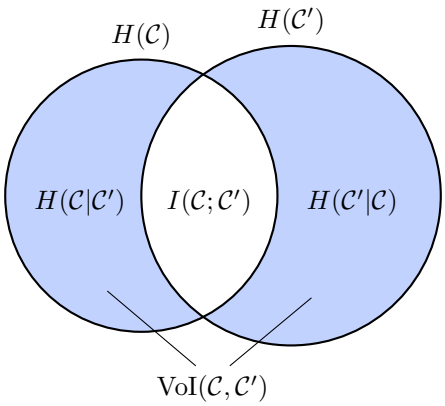


Figure 5.9: Information diagram for the variation of information.

Summary

- RAND and VoI allow comparing partitions without computing explicit assignments between clusters.
- RAND is typically and VoI sometimes adjusted for random matches (not discussed here).
- VoI is a proper distance.

Questions

- What partitioning will have low/high entropy?
- Assume $\mathcal{C}$ is a ground truth partitioning. Do we want $\mathcal{C}'$ to have large or small VoI?
- What does the information diagram look like if each C'_i is a strict subset of some C_j?
- Why use VoI rather than mutual information when comparing to some ground truth partitioning?

Literature

- David J.C. MacKay, *Information Theory, Inference and Learning Algorithms*. 2003, [Mac03] (Chapters 2.4 and 8 give a detailed view on entropy.)

6 k-nearest neighbor classifier

In the standard scenario of supervised learning we have a *training set* $\{\boldsymbol{x}_i, y_i\}$ with the *features/attributes/explanatory variables/inputs* $\boldsymbol{x}_i$ and *labels/targets/outputs* y_i and want to make predictions of y_i given $\boldsymbol{x}_i$. The prediction of discrete class labels y_i is called classification, while the prediction of a continuous quantity y_i is called Regression. In this chapter we look at one method of classification called k-nearest neighbor classifier.

6.1 Input and output scales and parameter number

The input and output of our methods can be on different scales:

- *nominal:* membership in category, e.g. particle type

  Example of problem with nominal scale output: standard classification problem.

- *ordinal:* ordered, e.g. 1st detection, 2nd detection, ...

- *interval:* continuous interval, e.g. centigrade scale

  Allows measure of difference. Example of problem with interval scale output: standard regression problem.

- *ratio:* interval scale with meaningful zero, e.g. Kelvin scale

We further differentiate between models based on their number of parameters:

- *Parametric method:* the model has a fixed number of parameters, e.g. like a standard deep feed-forward neural network, as we introduce in section 13.

- *"Non-parametric method"/ infinite dimensional method/ exemplar based method/ memory-based method:* the number of parameters of the model grows with the training set, e.g. like in kernel density estimation and in nearest neighbor classifiers. The name "non-parametric method" is a bit of a misnomer because the models do have parameters, but it is often used regardless.

6.2 (One-) nearest neighbor classifier

The nearest neighbor (NN) classifier is a non-parametric classifier with inputs on an interval scale. We try to predict the output of a query by finding the nearest neighbor (with respect to the features/input) in the training set, and then take its label as the prediction, as shown in Figure 6.1a. The *Voronoi region* of a sample in the training set is the region of all points that are nearest neighbors of the sample, meaning points that are closer to the it than to all other samples in the training set. Then the prediction for a point is the label of the sample of its Voronoi region. The boundary where we would change from predicting one label to predicting another is called *decision boundary*. For the example from before, Voronoi regions and decision boundaries are visualized in Figure 6.1b.

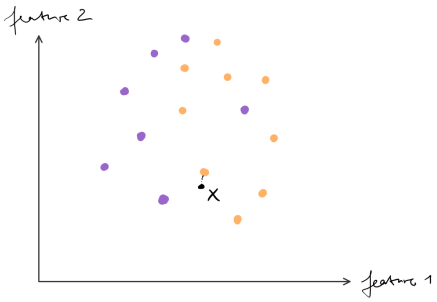
Imagine we have multiple training sets from the same source. The mean decision boundary would be close to the optimum, meaning there would be low bias (which we want). However, there would be much spread around this mean decision boundary, meaning there would be high variance (which we do not want). While it might seem that we have no parameters here, the effective number of parameters grows with $\mathfrak{n}$. We need to properly scale the features, so that the distance to neighbors is representative in all dimensions. The "algorithm" for classification is quite easy:

1. Given a sample we want to predict, find the closest sample from training set.

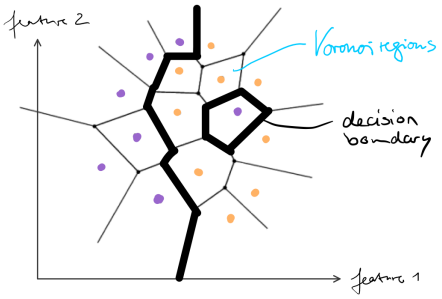
2. Use its label as prediction.

The Voronoi regions are not calculated explicitly.

This approach gives an error of 0 on the training set, but this does not imply perfect prediction for samples that are not in the training set, it is merely an extreme case of overfitting. It is a somewhat lazy strategy. We have zero effort at train time, but that is counteracted by the effort of $\mathcal{O}(\mathfrak{n})$ for one prediction in a naive implementation.



(a) Training set of orange and violet points, and point x for which we want to make a prediction. We predict x to be orange based on its nearest neighbor.



(b) The "Voronoi diagram" or "Voronoi tesselation" shows the Voronoi regions and decision boundaries of the training set.

Figure 6.1

To reduce the effort of a prediction, we can try to prune (reduce) the training set, while maintaining the same decision boundaries. To keep exactly the same decision boundaries we can look at the Delaunay triangulation, as shown in Figure 6.2. We get the same decision boundaries if we prune samples that only share boundaries with Voronoi regions of samples of the same class, or equivalently samples which are only connected to samples of the same class by the edges of the Delaunay triangulation. However, it is computationally expensive to calculate the Voronoi tesselation or Delaunay triangulation. Additionally, in higher dimensions the Delaunay tessalation converges to a complete graph, so that we can't prune without changing the decision boundaries (equivalently, in higher dimesions points share boundaries of Voronoi regions with a growing number of other points). It is much simpler to only approximately keep the same boundaries. An algorithm to do so would be:

1. Define the active set $\mathcal{A} = \{\}$.
2. Repeat: randomly select sample from training set.
   (a) If the sample is correctly classified by the NN-classifier over $\mathcal{A}$ discard it.
   (b) Else add sample to $\mathcal{A}$.

A disadvantage of this technique is that the resulting active set is dependent on the order of selection in step 2, and therefore we get different classifiers depending on the selection order. On the other hand, its predictions are faster.

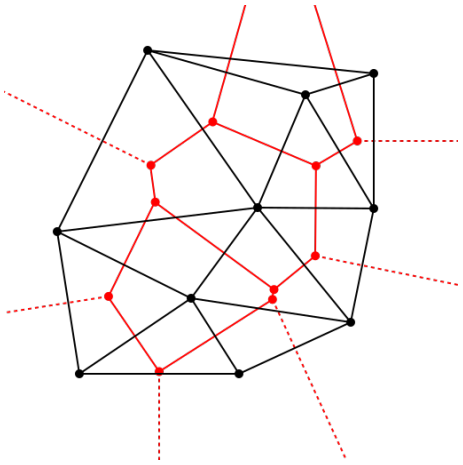


Figure 6.2: From [Hfe11]. The black dots represent the collection of samples in the training set. The red Voronoi diagram visualizes the boundaries of the Voronoi regions. The black lines connect points that share a boundary in the Voronoi diagram, this is called the "Delaunay triangulation".

6.3 k-nearest neighbor classifier

The k-NN-classifier includes not only one, but k of the nearest neighbors of a point and takes a vote among them for the prediction. This is done to get a less noisy estimate, as shown in Figure 6.3. For the kNN-classifier, we have to choose the "hyperparameter" k. There is a trade-off between loosing details if k is too large, and overfitting and getting a noisy classifier if k is too small.

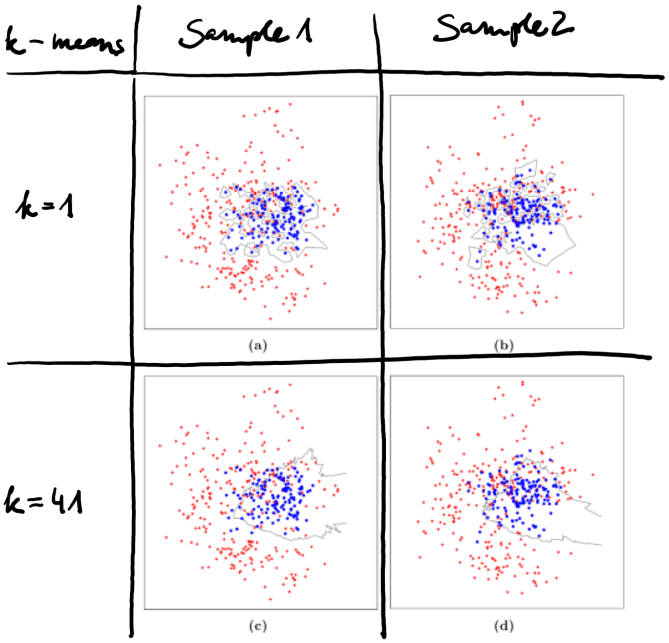


Figure 6.3: Given training sets 1 and 2 from the same source, we compare the NN-classifier with k=1 to the kNN-classifier with k=41. The decision boundaries, drawn in black are very different for k=1, the classifier is very noisy. For k=41, the decision boundaries have "broadly" the same shape.

Not all predictions of the kNN classifier are born equal. We should doubt predictions when only a smaller number of the neighbors agrees on the prediction. For example, the prediction of category A using a k=11-NN classifier is a lot more trustworthy if 11 of 11 neighbors agree on category A compared to if only 6 of 11 neighbors have label A and 5 of 11 have label B. We should also doubt predictions when the density of points in the training set is low in the region of the sample we want to predict. We can avoid this problem of low densities by using an ϵ-NN classifier, where only points within an ϵ-ball are included in the prediction. We can also use a kernel to weigh far away points less.

Order-k Voronoi diagrams can give a geometric interpretation of kNN classifiers. Here we only take a look at second-order Voronoi diagrams, as shown in Figure 6.4. The diagram visualizes the regions where all the points inside have the same two nearest neighbors in the training set.

6.4 Hyperparameters and cross-validation

Hyperparameters of a model are parameters that the model does not learn while training but instead are chosen beforehand. Examples for hyperparameters would be the number of layers in a neural network as well as the k in the k-nearest neighbor classifier. We can use different approaches to determine the best hyperparameters.

If we have a lot of training data available, we randomly partition it into the *training set* ($\sim 80\%$), the *validation set* ($\sim 10\%$) and the *test set* ($\sim 10\%$). To find the best hyperparameters for the model, grid search over possible hyperparameters:

1. Train multiple models with different hyperparameters on the training set.
2. Evaluate all models on the validation set and pick the hyperparameters that give the smallest error.

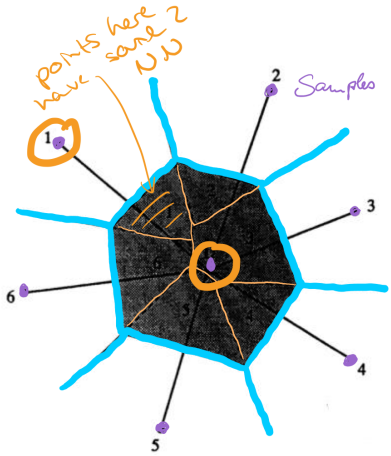


Figure 6.4: Edited, from [MS+91]. The boundaries of the first-order Voronoi region are drawn in blue, the ones of the second-order Voronoi regions are drawn in orange. The orange shaded region for example is the region where the two points marked in orange are the two nearest neighbors.

3. Evaluate the chosen model on the test set to get an estimate for its accuracy. (This is done on a separate set, so that the estimate is not too optimistic due to overfitting on the validation set. It should only happen once. Don't touch the test set beforehand!)

4. With the chosen best hyperparameters, train on all data (training set + validation set + test set) and deploy the model.

Frequent errors that can occur in this process are that the training set is contaminated by samples from validation or test set (e.g. when there is replicates in the dataset), that the partitioning into the different sets is not random, and that the test set is used multiple times. Be careful, in some literature the names of validation set and test set are exchanged!

If we do not have a lot of training data, we can use *cross-validation*.

1. Partition data randomly into k_f "folds", often $k_f = 10$.

2. Grid search over hyperparameters:

   (a) Train k_f models for the same hyperparameters by training on all but one of the folds (each time leave out a different fold).

   (b) Evaluate each model on the fold it was not trained on.

   (c) Estimate the mean and spread of the k_f models (for the same hyperparameters).

3. Choose the best hyperparameters, train on all data and deploy the model.

Advantages of this method are that it makes better use of small data sets, and also that the same hyperparameters are evaluated on an ensemble of models, which gives an errorbar on the accuracy. A disadvantage of this method is that the estimate for the accuracy might be too optimistic, because we don't have a separate test set.

Summary

- We can classify models based on the type of input and output types they work with and how many parameters they have.
- The kNN classifier is a decent, basic method that especially works well in low dimensions.
- Hyperparameters are parameters that are not learned by training but instead chosen beforehand. Cross-validation is useful for tuning hyperparameters.

Literature

- Trevor Hastie, Robert Tibshirani, and Jerome H Friedman, "The elements of statistical learning: data mining, inference, and prediction". 2009, Chap. 13.3 [HTF09] (Overview)
- Luc Devroye, László Györfi, and Gábor Lugosi, *A probabilistic theory of pattern recognition*. 2013, [DGL13] (Proofs on asymptotic consistency, etc. of k-NN)
- Brian D Ripley, *Pattern recognition and neural networks*. 2007, [Rip07] (Summary of such properties)

7 Statistical decision theory

7.1 Probability theory

We first repeat and expand some important concepts from probability theory, that were introduced in subsection 3.3.

- Ω is the *sample space*, the set of all possible "atomic" events/outcomes resulting from a random experiment.
- Subsets of Ω are called *events* and are sets of atomic events/outcomes themselves. An event α is realized if an atomic event $\omega \in \alpha$ occurs. Figure 7.1 shows a common way to visualize the sample space Ω and two events α and β.

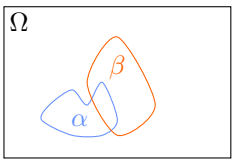


Figure 7.1: Sample space Ω and events α and β.

- $\mathcal{F}$ is the *event space*, the set of all distinguishable events.
- The *probability* P is a mapping $P : \mathcal{F} \to [0, 1]$, such that:

  (1) $0 \leq P(\alpha) \leq 1 \quad \forall \alpha \in \mathcal{F}$

  (2) $P(\Omega) = 1$

  (3) $P(\cup_i \alpha_i) = \sum_i P(\alpha_i)$ for exclusive events $\alpha_i \cap \alpha_j = \emptyset$

  For non-exclusive events like in Figure 7.1 we have $P(\alpha \cup \beta) = P(\alpha) + P(\beta) - P(\alpha \cap \beta)$
- The triple $(\Omega, \mathcal{F}, P)$ is called *probability space*.
- The *Joint Probability* of two events α and β is the probability that both events occur.

$$P(\alpha \cap \beta) = P(\alpha, \beta) \tag{7.1}$$

- A collection of events $\{\alpha_i\}_{i=1}^n$ is *independent*, if for every subset $\{\alpha_{i_1}, ..., \alpha_{i_j}\}$:

$$P(\alpha_{i_1} \cap \cdots \cap \alpha_{i_j}) = P(\alpha_{i_1}, \ldots, \alpha_{i_j}) = P(\alpha_{i_1}) \ldots P(\alpha_{i_j}) \tag{7.2}$$

  It is not enough for all pairs of events α_i and α_j to be independent to guarantee joint independence.
- The *Conditional Probability* is the probability for event α, given that event β has occurred and can be written as

$$P(\alpha|\beta) = \frac{P(\alpha, \beta)}{P(\beta)} \tag{7.3}$$

  We call $P(\alpha|\beta)$ "the probability of α given β".
- From Equation 7.3 we get the *chain rule of probability*.

$$P(\alpha, \beta) = P(\beta|\alpha) P(\alpha) \tag{7.4}$$
$$= P(\alpha|\beta) P(\beta) \tag{7.5}$$
$$P(\alpha, \beta, \gamma) = P(\alpha|\beta, \gamma) P(\beta|\gamma) P(\gamma) \tag{7.6}$$

- For the expectation of some function $f(X, Y)$ of two random variables X, Y as defined in subsection 3.3, we have the following relation

$$\mathbb{E}_{X,Y}\left[f(X,Y)\right] = \mathbb{E}_X\left[\mathbb{E}_{Y|X}\left[f(X,Y)\right]\right]. \tag{7.7}$$

Here, $\mathbb{E}_{X,Y}$ or $\mathbb{E}_{XY}$ is the expectation over the joint distribution of X and Y. $\mathbb{E}_{Y|X}$ is the expectation over the conditional distribution of $Y|X$, which is the distribution of Y given a known value for X. In the case of continuous distributions, this relation can be written as

$$\int f(x,y)p(x,y)dxdy = \int \left(\int f(x,y)p(y|x)dy\right)p(x)dx, \tag{7.8}$$

where p denotes the respective probability density functions (see also subsection 3.3).

7.2 Bayes theorem

By reordering Equation 7.4 and Equation 7.5, we obtain Bayes theorem.

$$\underbrace{P(\alpha|\beta)}_{\text{posterior}} = \frac{\overbrace{P(\beta|\alpha)}^{\text{likelihood}}\overbrace{P(\alpha)}^{\text{prior}}}{\underbrace{P(\beta)}_{\text{evidence}}} \tag{7.9}$$

In the case of a classification problem with (continuous) features $\boldsymbol{x}$ and labels y, we want to predict the posterior probability $p(y|\boldsymbol{x})$ of class y. We then have

$$p(y|\boldsymbol{x}) = \frac{p(\boldsymbol{x}|y)\,p(y)}{p(\boldsymbol{x})}, \tag{7.10}$$

where the likelihood $p(\boldsymbol{x}|y)$ is the probability density of class y, the prior $p(y)$ is the prior probability of y and the evidence $p(\boldsymbol{x})$ is the marginal density of $\boldsymbol{x}$.

For some 1D toy data, the class densities and posteriors are shown in Figure 7.2 and Figure 7.3.

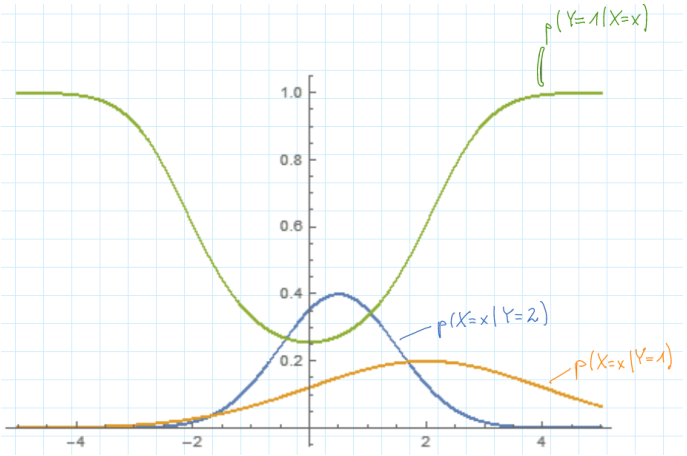


Figure 7.2: Example of 1D toy data. Class density of class 1 in orange and class 2 in blue. The priors are assumed to be $\frac{1}{2}$ each. The evidence is calculated using $p(x) = p(x|1)p(1) + p(x|2)p(2)$. The posterior in green can then be calculated from Bayes theorem.

7.3 Discriminative vs. generative classifiers

We can differentiate between discriminative and generative classifiers. Discriminative classifiers directly estimate the left hand side of Bayes theorem, the posterior. They can only be used to classify samples. Generative classifiers estimate the right hand side of Bayes theorem. They learn the underlying probability distribution and can thus be used to generate new data.

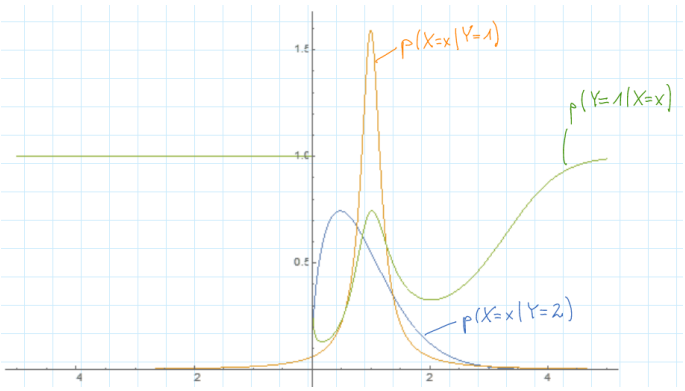


Figure 7.3: Example of 1D toy data like in Figure 7.2, but with different initial class densities. Class density of class 1 in orange and class 2 in blue. The priors are assumed to be $\frac{1}{2}$ each. The evidence is calculated using $p(x) = p(x|1)p(1) + p(x|2)p(2)$. The posterior in green can then be calculated from Bayes theorem.

$$\underbrace{P(\alpha|\beta)}_{\substack{\text{estimate LHS:}\\ \text{discriminative classifier}}} = \underbrace{\frac{P(\beta|\alpha)\,P(\alpha)}{P(\beta)}}_{\substack{\text{estimate RHS:}\\ \text{generative classifier}}} \tag{7.11}$$

7.4 Statistical/Bayesian decision theory

We might ask what the best conceivable classifier looks like and how good is it? How good is it? To answer these questions, we need:

1. Statistical model of the data: the joint distribution of features $\boldsymbol{x}$ and labels y is given by

$$p(\boldsymbol{x}, y) = \underbrace{p(\boldsymbol{x}|y)}_{\substack{\text{class-conditional}\\ \text{density}}} \quad \underbrace{p(y)}_{\substack{\text{prior probability}\\ \text{of a class}}} \tag{7.12}$$

   The class-conditional density here is the probability density function of one class y.

2. Quantitative description of "bad" and "good" classifier: We need to define some *loss function* $L(y, \hat{y})$ which specifies a penalty for all possible combinations of the predicted class $\hat{y}$ and the true class y.

   Example: For the loss function shown in Table 7.1, we consider a case with 3 possible classes where additionally samples can be rejected during prediction. The cases where the prediction matches the true class do not get a penalty. For the other, false predictions we can set different penalties. This can be useful when some false predictions are worse than others. For example, you might want to penalize your self-driving car a lot more for mistaking a grandma for a shadow and not stopping, compared to the other way around.

$L(y, \hat{y})$	$\hat{y} = 1$	$\hat{y} = 2$	$\hat{y} = 3$	$\hat{y} = rejected$
$y = 1$	0	100	1	10
$y = 2$	1	0	2	10
$y = 3$	0.1	0.1	0	10

Table 7.1: Example of the values of a possible loss function.

3. Formula for expected loss: We generally define the *risk* $R(f)$ of a classifier f as its expected loss.

$$R(f) \coloneqq \mathbb{E}_{XY}\left(L\left(Y = y, f(X = \boldsymbol{x})\right)\right) \tag{7.13}$$

We can then define the "Bayes risk" and "Bayes classifier" for a given data distribution as

$$\text{Bayes risk} := \min_f R(f) \tag{7.14}$$

$$\text{Bayes classifier} := \arg\min_f R(f) \tag{7.15}$$

Using Equation 7.1, the risk can be calculated by:

$$R(f) = \mathbb{E}_X \mathbb{E}_{Y|X} L\left(Y = y, f(X = \boldsymbol{x})\right) \tag{7.16}$$

$$= \int \mathbb{E}_{Y|X} L\left(Y = y, f(X = \boldsymbol{x})\right) p^*(\boldsymbol{x}) dx \tag{7.17}$$

The minimization of this risk can be done point wise for each $\boldsymbol{x}$.

$$\mathbb{E}_{Y|X=\boldsymbol{x}} L\left(Y = y, f(X = \boldsymbol{x})\right) = \sum_{y \in Y} L\left(y, f(\boldsymbol{x})\right) p^*(y|\boldsymbol{x}) \tag{7.18}$$

Here, $p^*(y|\boldsymbol{x})$ is the true posterior, not an estimate! Usually, the true posterior is not known, therefore the Bayes classifier is a theoretical concept.

One important loss function is the 0-1-loss, as shown in Table 7.2. Using 0-1-loss we get

$$\mathbb{E}_{Y|X=\boldsymbol{x}} L\left(Y = y, f(X = \boldsymbol{x})\right) = 0 \cdot p^*(f(\boldsymbol{x})|\boldsymbol{x}) + \sum_{y \in Y \setminus f(\boldsymbol{x})} 1 \cdot p^*(y|\boldsymbol{x}) \tag{7.19}$$

$$= 1 - p^*(f(\boldsymbol{x})|\boldsymbol{x}) \tag{7.20}$$

Here we used that $\sum_y p^*(y|\boldsymbol{x}) = 1$. We see that minimizing the point wise risk is equivalent to maximizing $p^*(f(\boldsymbol{x})|\boldsymbol{x})$. Therefore for the 0-1-loss, the Bayes classifier always predicts the class with the highest posterior probability.

$$f_{\text{Bayes}}(\boldsymbol{x}) = \arg\max_y p^*(y|\boldsymbol{x}) \tag{7.21}$$

$L(y, \hat{y})$	$\hat{y} = 1$	$\hat{y} = 2$	$\hat{y} = 3$
$y = 1$	0	1	1
$y = 2$	1	0	1
$y = 3$	1	1	0

Table 7.2: Important loss: 0-1-loss.

The Bayes classifier is the best classifier, but it is not perfect, so usually $\min_f R(f) \neq 0$. This can be seen in the example of the Bayes classifier applied to the toy data set, shown in Figure 7.4. The risk depends on how close to integrality the posterior class probabilities are, or equivalently, how strongly the classes overlap. This makes acquiring discriminative features very important, as shown in Figure 7.5.

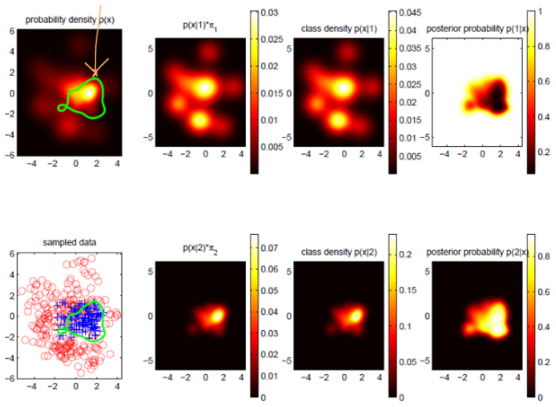
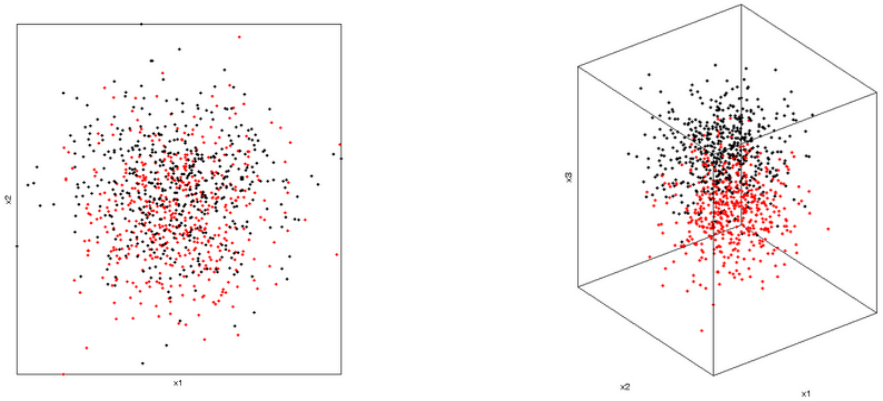


Figure 7.4: Toy data example: decision boundary of the Bayes classifier shown in green.



(a) Some samples belonging to two classes, red and black. The features x_1 and x_2 are not suitable to differentiate between the two classes.

(b) Some samples belonging to two classes, red and black. The feature x_3 can differentiate nicely between the two classes.

Figure 7.5: Bad selection (left) vs. good selection of features (right) for a toy dataset.

Summary

- Forever remember Bayes theorem!

$$P(\alpha|\beta) = \frac{P(\beta|\alpha)\,P(\alpha)}{P(\beta)} \tag{7.22}$$

- Discriminative classifiers estimate the LHS of Bayes theorem and can only classify data. Generative classifiers estimate the RHS of Bayes theorem, they learn the underlying distribution and can generate new data.
- The Bayes classifier is a very important theoretical concept. It relies on the knowledge of the true posterior class probabilities $p^*(y|\boldsymbol{x})$. These are usually not known, therefore the Bayes classifier remains elusive. The Bayes classifier is the best classifier, but it is not perfect. It is very important to have discriminative features.

Literature

- Christopher M Bishop and Nasser M Nasrabadi, *Pattern recognition and machine learning*. 2006, Chap. 1.5 [BN06]

8 Bayesian inference

8.1 Bayesian inference

Consider an ε-NN classifier. The *frequentist estimation* of the probability of a class k would simply be

$$\hat{P}(Y = k|X = x) = \frac{\sum_{i=1}^{n} I(\|x_i - x\| < \varepsilon \wedge y_i = k)}{\sum_{i=1}^{n} I(\|x_i - x\| < \varepsilon)}. \tag{8.1}$$

Let's look at a two class case where there are one sample belonging to class A and two belonging to class B in the ε-ball, compared to the case where there are 10 samples belonging to class A and 20 belonging to class B. The frequentist approach gives the same probability for both cases. But the case of 10:20 samples should be a lot more certain than the case of 1:2 samples. We search for a technique that reflects this.

8.1.1 Important distributions

Bernoulli distribution

The Bernoulli distribution is a distribution with

$$\text{Bern}(x|\mu) = \mu^x(1-\mu)^{1-x}, \tag{8.2}$$
$$\mathbb{E}[X] = \mu, \tag{8.3}$$
$$\text{Var}[X] = \mu(1-\mu) \tag{8.4}$$

with $P(X = 1) = \mu$ and $P(X = 0) = 1 - \mu$. It describes an experiment with one yes-no question asked, like one coin throw. It is a special case of the Binomial distribution with $N = 1$.

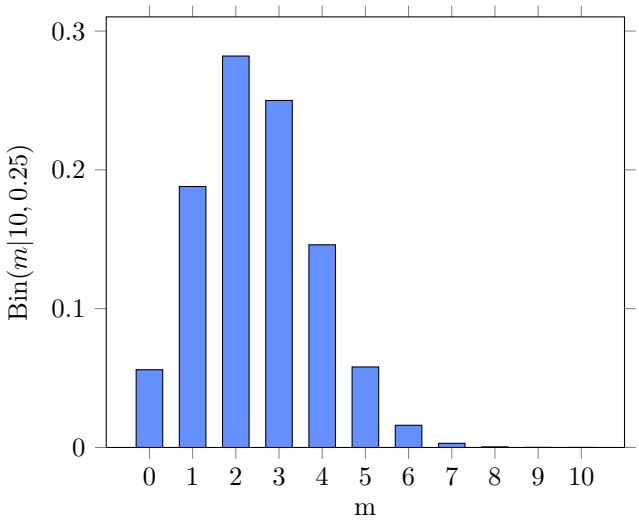


Figure 8.1: Example of Binomial distribution for $\mu = 0.25$ and $N = 10$ trials.

Binomial distribution

The binomial distribution is a distribution with

$$\text{Bin}(m|N,\mu) = \binom{N}{m}\mu^m(1-\mu)^{N-m}, \tag{8.5}$$

$$\mathbb{E}[m] = \sum_{m=0}^{N} m\text{Bin}(m|N,\mu) = N\mu, \tag{8.6}$$

$$\text{Var}[m] = \sum_{m=0}^{N} (m - \mathbb{E}[m])^2\text{Bin}(m|N,\mu) = N\mu(1-\mu). \tag{8.7}$$

It is the probability distribution for the number of successes m of N independent experiments of a yes-no question/coin throw, with each having a probability of μ for a success and $(1-\mu)$ for a failure, as before. An example for a Binomial distribution is shown in Figure 8.1.

Beta distribution

The beta distribution is a distribution in the interval $\mu \in [0,1]$ with

$$\text{Beta}(\mu|a,b) = \frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)}\mu^{a-1}(1-\mu)^{b-1} \tag{8.8}$$

$$\mathbb{E}[\mu] = \frac{a}{a+b} \tag{8.9}$$

$$\text{Var}[\mu] = \frac{ab}{(a+b)^2(a+b+1)}. \tag{8.10}$$

For different values of a and b, it is plotted in Figure 8.2.

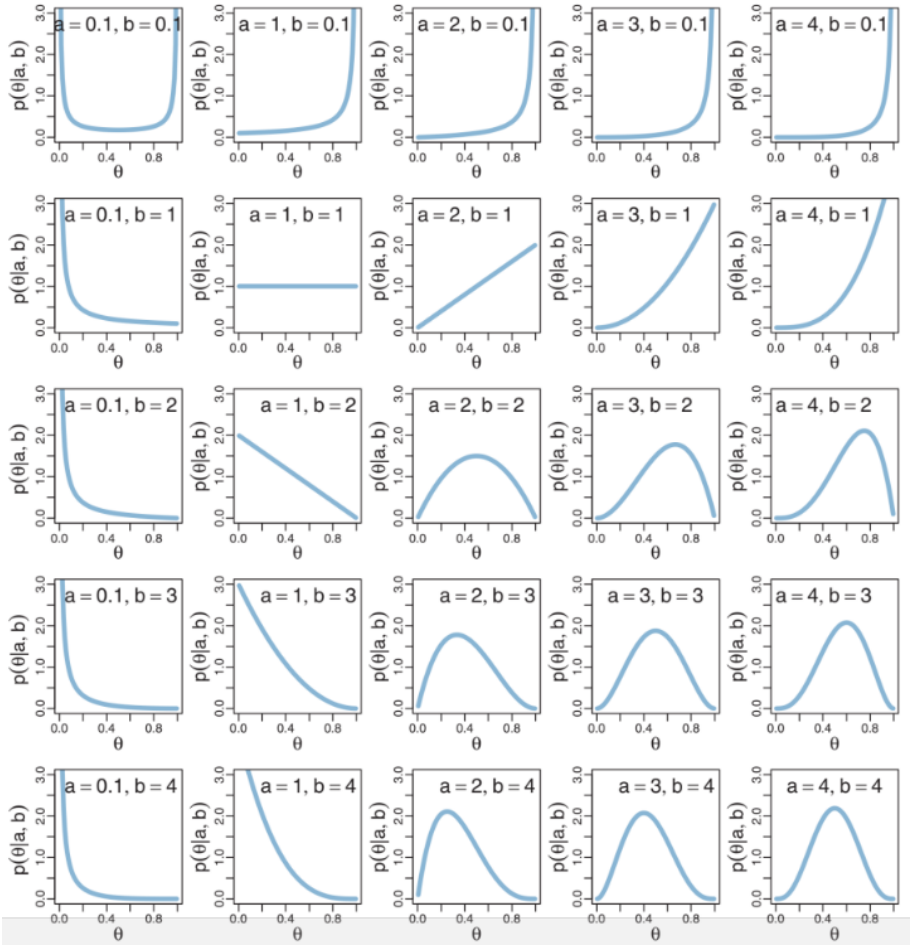


Figure 8.2: Beta distribution for different values of a and b. Figure from [Kru14].

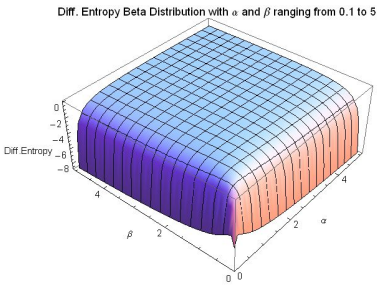
The differential entropy is the extension of the entropy to continuous distributions.

$$H[X] = \mathbb{E}[-\log(p(X))] = \int -p(x)\log(p(x))\mathrm{d}x \tag{8.11}$$

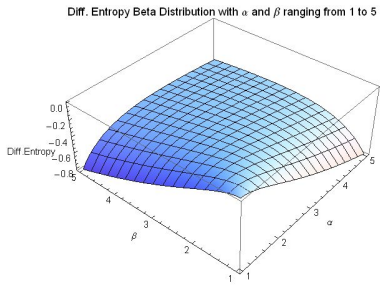
The differential entropy of the beta distribution is given by

$$H(\text{Beta}(\mu|a,b)) = -(a-1)\psi(a) - (b-1)\psi(b) + (a+b-2)\psi(a+b) + \text{const.} \tag{8.12}$$

with the "digamma function" $\psi(x) \sim \ln x - \frac{1}{2x}$. For different values of a and b it is plotted in Figure 8.3. It is largest for $a = b = 1$, which is equivalent to a uniform distribution. This makes sense, as the uncertainty is largest if all events have the same probability.



(a) From Figure from [J R12a].. Differential Entropy for $a, b \in [0.1, 5]$



(b) From Figure from [J R12b].. Differential Entropy for $a, b \in [1, 5]$

Figure 8.3: Differential Entropy for the Beta Distribution. The entropy is largest for $a = b = 1$.

8.1.2 Conjugacy and inference

We look at an example of N coin flips. The results of the flips are given in a data set $\mathcal{D} = x_1, \ldots, x_N\}$, with m heads ($x_i = 1$) and $N - m$ tails ($x_i = 0$). We now want to predict the probability of success, μ.

Let's remember Bayes theorem for a hypothesis μ given the data set $\mathcal{D}$.

$$\underbrace{P\left(\mu|\mathcal{D}\right)}_{\text{posterior}} = \frac{\overbrace{P\left(\mathcal{D}|\mu\right)}^{\text{likelihood}} \cdot \overbrace{P\left(\mu\right)}^{\text{prior}}}{\underbrace{P\left(\mathcal{D}\right)}_{\text{evidence}}}. \tag{8.13}$$

Frequentist approach

We can use a frequentist approach, where we try to find the value of μ that will maximize the probability of data $\mathcal{D}$ with the maximum likelihood estimator (MLE),

$$\hat{\mu}_{\text{MLE}} = \arg\max_{\mu} P(\mathcal{D}|\mu). \tag{8.14}$$

This means maximizing the likelihood in Bayes theorem Equation 8.1.2. In our example of independent coin flips that are each described by the Bernoulli distribution, we get

$$P(\mathcal{D}|\mu) = \prod_{n=1}^{N} p(x_n|\mu) = \prod_{n=1}^{N} \mu^{x_n}(1-\mu)^{1-x_n}, \tag{8.15}$$

$$\hat{\mu}_{\text{MLE}} = \frac{1}{N}\sum_{n=1}^{N} x_n = \frac{m}{N}. \tag{8.16}$$

For a measurement of $\mathcal{D} = \{1, 1, 1\}$ this would give us an estimated probability of success $\hat{\mu}_{\text{MLE}} = \frac{3}{3} = 1$, so we would predict heads (1) every time. Obviously, we have overfitted to $\mathcal{D}$ with this approach.

Bayesian inference

As an alternative, we can use Bayesian inference. Instead of the likelihood, we try to maximize the posterior, the probability of hypothesis μ given the data set $\mathcal{D}$.

$$\hat{\mu}_{\text{MP}} = \arg\max_{\mu} P\left(\mu|\mathcal{D}\right) \tag{8.17}$$

$$= \arg\max_{\mu} \frac{P\left(\mathcal{D}|\mu\right) P\left(\mu\right)}{P\left(\mathcal{D}\right)} \tag{8.18}$$

$$= \arg\max_{\mu} P\left(\mathcal{D}|\mu\right) P\left(\mu\right) \tag{8.19}$$

In the last step, we can drop the evidence $P(\mathcal{D})$, because it is independent of the μ we try to maximize. In probability theory, if posterior and prior are in the same family of distributions, they are called *conjugate distributions*. The prior that gives a conjugate posterior for a given likelihood is called *conjugate prior*.

Now, let's look at our experiment of coin throws again.

$$p(\mu|a_0, b_0, \mathcal{D}) \propto p(\mathcal{D}|\mu)p(\mu|a_0, b_0) \tag{8.20}$$

$$= \left(\prod_{n=1}^{N} \mu^{x_n}(1-\mu)^{1-x_n}\right) \text{Beta}(\mu|a_0, b_0) \tag{8.21}$$

$$\propto \mu^{m+a_0-1}(1-\mu)^{N-m+b_0-1} \tag{8.22}$$

$$\propto \text{Beta}(\mu|a_N, b_N) \tag{8.23}$$

with $a_N = a_0 + m$ and $b_N = b_0 + N - m$. The Beta distribution is the conjugate prior of the Bernoulli distribution. We can use this to make predictions now. Given the data set $\mathcal{D}$, what is the probability that the next coin flip will land heads up, $X = 1$?

$$p(X = 1|a_0, b_0, \mathcal{D}) = \int_0^1 p(X = 1|\mu)p(\mu|a_0, b_0, \mathcal{D})\mathrm{d}\mu \tag{8.24}$$

$$= \int_0^1 \mu p(\mu|a_0, b_0, \mathcal{D})\mathrm{d}\mu \tag{8.25}$$

$$= \mathbb{E}[\mu|a_0, b_0, \mathcal{D}] \tag{8.26}$$

$$= \frac{a_0 + m}{a_0 + b_0 + N} \tag{8.27}$$

With increasing size of the data set $\mathcal{D}$, it becomes more representative and trustworthy. The priors conversely become less important for larger N, as shown in Figure 8.4. For large N the expectation of Bayesian inference goes to the expectation of the frequentist approach,

$$a_N \to m \tag{8.28}$$

$$b_N \to N - m \tag{8.29}$$

$$\mathbb{E}[\mu] = \frac{a_N}{a_N + b_N} \to \frac{m}{N} = \hat{\mu}_{\text{MLE}} \tag{8.30}$$

$$\text{Var}(X) = \frac{a_N b_N}{(a_N + b_N)^2(a_N + b_N + 1)} \to 0. \tag{8.31}$$

It is important to note that different distributions have different conjugate priors, we do not always need the Beta distribution. For example for the multinomial distribution, the conjugate prior is the Dirichlet distribution.

Let's go back to the ε-NN classifier example we considered in the beginning. We have reached our goal of differentiating the cases of the ratios 1:2 and 10:20 for class 1 to class 2. Using a Beta distribution with $a_0 = b_0 = \frac{1}{2}$[1], we get very different posteriors, as shown in Figure 8.5.

[1]This is the so called"Jeffrey's prior" for the Bernoulli distribution

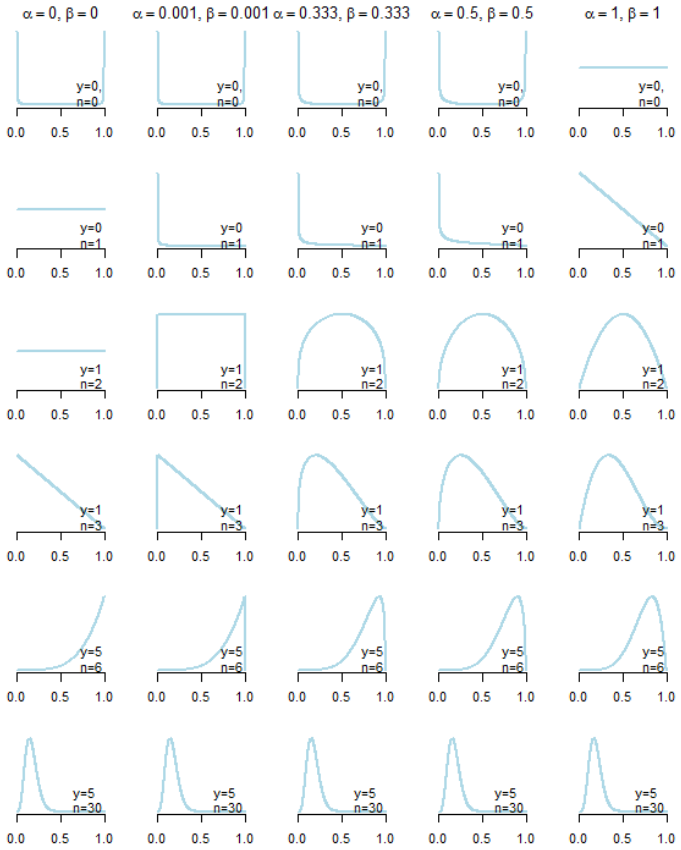


Figure 8.4: The upper line of plots are different priors, the column below each prior shows the resulting posteriors given m=y successes in N=n trials. Observe how the effect of the prior vanishes for a large number of trials, in the lowest row, with m=5, N=30, there is almost no difference for the different priors. Figure from [Tim17].

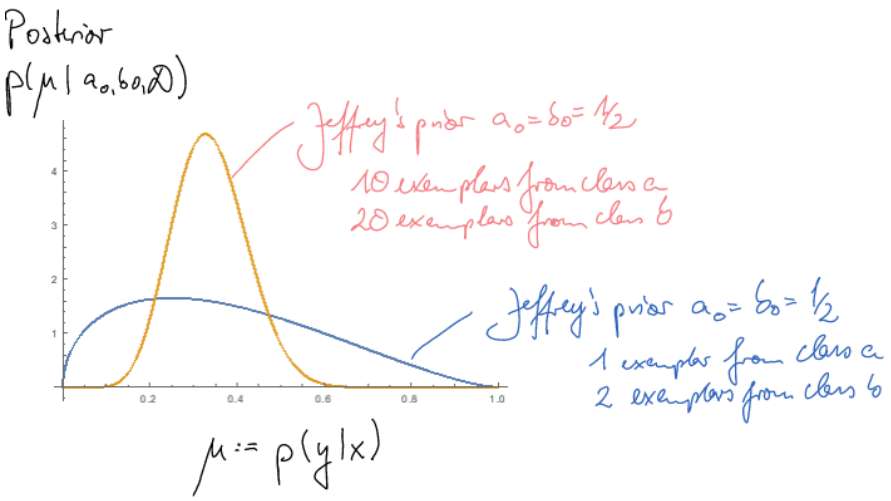


Figure 8.5: Posteriors for 1:2 and 10:20 samples for the ratio of class a to class b.

8.2 Active learning / Bayesian optimization

In real world scenarios, it is often the case that measurements of some features $\boldsymbol{x}$ are cheap, but getting the labels y of the measurements is expensive. So we want to avoid labeling features that are uninformative for our problem. Active learning describes techniques to determine which measurements are most informative and should be labeled. In conclusion, we use it in cases where:

- A large set of unlabeled instances $\mathcal{U}$ are available (or can be synthesized).
- There is only a small or empty set of labeled instances $\mathcal{L}$ available, but more labels are available upon request.

Active learning follows the scheme shown in Figure 8.6. A *guide* chooses the samples that should be labeled. There are different techniques for the guide:

1. Uncertainty sampling (easy): Train the model on all labeled samples. Label the sample whose prediction by the current model is most uncertain, according to one of the following criteria:

   - Least confidence: $\arg\min_{\boldsymbol{x}\in\mathcal{U}}(\max_y \hat{P}(y|\boldsymbol{x}))$
   - Maximum entropy: $\arg\max_{\boldsymbol{x}\in\mathcal{U}}(-\sum_y \hat{P}(y|\boldsymbol{x})\log\hat{P}(y|\boldsymbol{x}))$

   Then train the model again with the new additional sample. Repeat the process until a desired number of samples is labeled, or a desired accuracy is reached. The effect of uncertainty sampling compared to random sampling is demonstrated in Figure 8.7. Limitations of this approach are that we need $\hat{P}(y|x)$, which is not a one-hot vector and should be not confident when far away from all labeled data ($\Leftrightarrow$ the guide becomes obsessed with overlap regions). Also, outliers are unfortunately likely to get labeled, there is no density weighting preventing this.

2. Query-by-committee (easy): Train several different models on all labeled data. Let the models make a prediction for all unlabeled samples. Label the sample where there was the most disagreement between the models.

3. Expected uncertainty reduction / expected error reduction (hard): Fantasize label y for a point $\boldsymbol{x}$ with probability $\hat{P}(y|\boldsymbol{x};\mathcal{L})$. $\hat{P}$ is the current test guess of probability of label y at the unlabeled point $\boldsymbol{x}$. We then try to minimize the uncertainty the retrained classifier has on all unlabeled samples.

$$\arg\min_{\boldsymbol{x}\in\mathcal{U}} \sum_y \hat{P}(y|\boldsymbol{x};\mathcal{L}) \sum_{\boldsymbol{x}'\in\mathcal{U}} \underbrace{\sum_{y'} -\hat{P}(y'|\boldsymbol{x}';\mathcal{L}\cup\{(\boldsymbol{x},y)\})\log\hat{P}(y'|\boldsymbol{x}';\mathcal{L}\cup\{(\boldsymbol{x},y)\})}_{\text{entropy when retraining classifier with additional } (\boldsymbol{x},y)} \tag{8.32}$$

   This approach is slow, especially for more than two classes. However, at the same time it does implicit density weighting, which means it is more robust to outliers compared to uncertainty sampling.

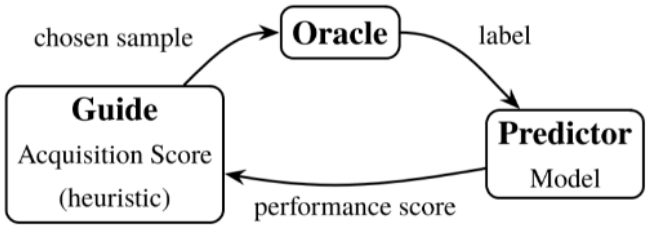


Figure 8.6: Schematic representation of active learning process. Figure from [Hau21].

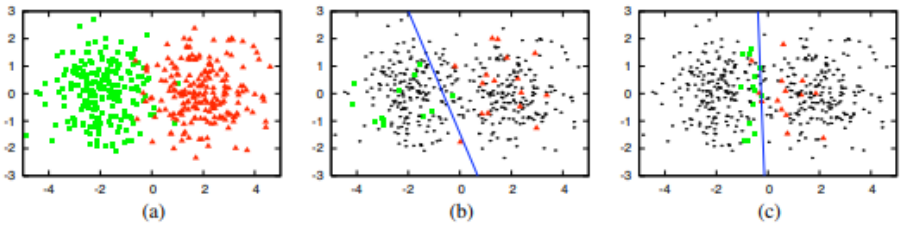


Figure 8.7: Data set of samples from two Gaussian distributions. (a) shows all instances labeled. Comparing the graph with random sampling (b) to the one with uncertainty sampling (c), we see that uncertainty sampling helps to determine the decision boundary a lot more accurately. Figure from [Set09].

8.3 Covariance and multivariate normal

The multivariate normal distribution was already introduced in section 5. The probability density of a $\mathfrak{p}$-variate normal distribution of the $\mathfrak{p}$-dimensional random vector $\boldsymbol{x}$ is given by

$$p(\boldsymbol{x}) = \frac{1}{(2\pi)^{\mathfrak{p}/2}} \frac{1}{\sqrt{\det(\boldsymbol{\Sigma})}} \exp\left[-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu})\right] \tag{8.33}$$

In general, we can define the *covariance matrix*, which is a $\mathfrak{p} \times \mathfrak{p}$ matrix

$$\overbrace{\boldsymbol{\Sigma}}^{\mathfrak{p}\times\mathfrak{p}} = \mathrm{Cov}(\boldsymbol{X}) = \mathbb{E}\left[\overbrace{(\boldsymbol{X}-\mathbb{E}[\boldsymbol{X}])}^{\mathfrak{p}\times 1}\overbrace{(\boldsymbol{X}-\mathbb{E}[\boldsymbol{X}])^\top}^{1\times\mathfrak{p}}\right] \tag{8.34}$$

$$= \mathbb{E}\left[\boldsymbol{X}\boldsymbol{X}^\top\right] - \mathbb{E}[\boldsymbol{X}]\,\mathbb{E}\left[\boldsymbol{X}^\top\right]. \tag{8.35}$$

From the last expression we can deduce the following relation for an arbitrary matrix A

$$\mathrm{Cov}(\boldsymbol{A}\boldsymbol{X}) = \boldsymbol{A}\,\mathrm{Cov}(\boldsymbol{X})\,\boldsymbol{A}^\top. \tag{8.36}$$

The inverse of the covariance matrix $\boldsymbol{\Sigma}^{-1}$ is called *precision matrix*. An empirical estimate of $\boldsymbol{\Sigma}$ can be obtained by

$$\boldsymbol{Q} = \frac{1}{\mathfrak{n}-1}\sum_{i=1}^{\mathfrak{n}}(\boldsymbol{x}_i-\bar{\boldsymbol{x}})(\boldsymbol{x}_i-\bar{\boldsymbol{x}})^\top. \tag{8.37}$$

The factor of $\mathfrak{n}-1$ in the denominator accounts for the fact we effectively lost a degree of freedom by estimating the mean of the distribution $\bar{\boldsymbol{x}}$ from the same sample. This makes Q an *unbiased* estimator. The *correlation matrix* is given by

$$[\mathrm{Corr}(\boldsymbol{X})]_{ij} = \frac{[\mathrm{Cov}(\boldsymbol{X})]_{ij}}{[\mathrm{Cov}(\boldsymbol{X})]_{ii}^{1/2}\,[\mathrm{Cov}(\boldsymbol{X})]_{jj}^{1/2}} \tag{8.38}$$

In the case of a bivariate distribution, the covariance would be

$$\boldsymbol{\Sigma} = \begin{bmatrix}\sigma_1^2 & \rho\sigma_1\sigma_2 \\ \rho\sigma_1\sigma_2 & \sigma_2^2\end{bmatrix}, \tag{8.39}$$

where σ_1, σ_2 are the standard deviations of the two random variables and $\rho \in [-1, 1]$ is the correlation between them. The independence of two random variables implies that their correlation is zero, however crucially the converse is not true, i.e non-independent random variables can have a correlation of zero. Some bivariate normal distributions are shown in Figure 8.8. In general, the $\mathfrak{p}$-variate normal distribution is closed under marginalization and conditioning.

→ The marginal of a normal distribution is a normal distribution.

→ The conditional of a normal distribution is a normal distribution.

→ Independent random variables with normal marginals have joint normal distribution.

→ However, normal marginals do NOT guarantee joint normal distribution, as shown in Figure 8.9.

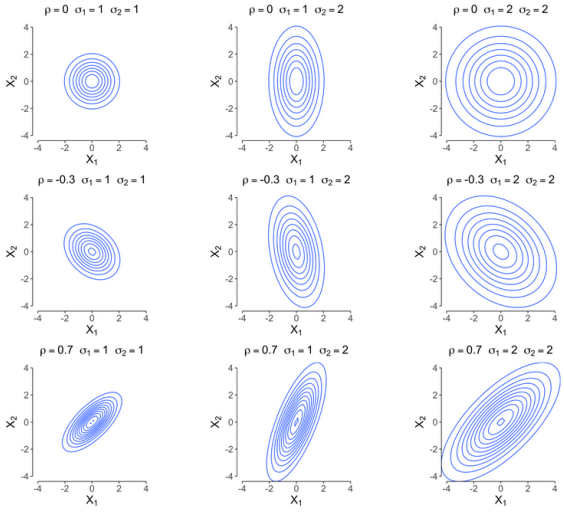


Figure 8.8: Different bivariate normal distributions. For $\rho = 0$ the random variables are independent (upper left and upper right plot), for $\rho \neq 0$ they are dependent (other plots). We can see that the marginal and the conditional distributions are normal as well. Figure from [Dab19].

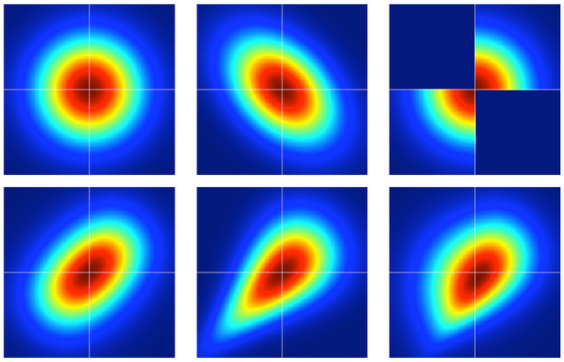


Figure 8.9: Joint distribution of two random variables with normal marginals. The bivariate normal distribution in the upper left corner is a joint normal distribution of two independent random variables, $\rho = 0$. The bivariate normal distribution in the upper middle has dependent random variables, $\rho = -0.4$. The other distributions have normal marginals, but are not bivariate normal distributions. Figure from [car12].

Summary

- Bayesian inference can give a meaningful indication of uncertainty. The choice of prior is often somewhat arbitrary. The Beta distribution is the conjugate prior of the Binomial distribution. For a prior $\text{Beta}(a_0, b_0)$ and an experiment with m successes in N trials, the posterior is $\text{Beta}(a_0 + m, N - m + b_0)$.
- Active learning describes techniques used in scenarios where the labeling of samples is expensive, to determine which measurements are most informative and would be most useful to be labeled.
- The $\mathfrak{p}$-variate normal distribution is closed under marginalization and conditioning. However, normal marginals don't imply a joint normal distribution.

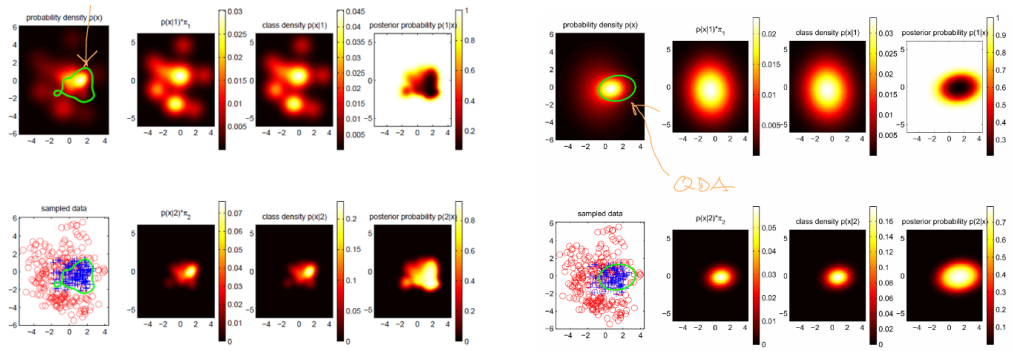
Literature

- Richard O. Duda, Peter E. Hart, and David G. Stork, *Pattern Classification*. 2000, [DHS00]

9 Quadratic discriminant analysis, classification trees

9.1 Quadratic discriminant analysis

QDA is a parametric, generative model for classification. It is restricted to data on the interval scale and works well in high dimensions. It has no hyperparameters.



(a) "Unknown Truth": True probability densities, class densities, posterior probabilities for a two class example. These true distributions are unknown to us, we only get the sampled data (lower left corner). The best possible decision boundary (green line) would be given by Bayes theorem.

(b) Our Model: We estimate the class densities to be Gaussian. Using Bayes theorem we get the corresponding decision boundary (green line), that approximates the true decision boundary of (a).

Figure 9.1

In quadratic discriminant analysis (QDA) we approximate the class densities of the single classes with Gaussian distributions with means $\boldsymbol{\mu}_i$ and covariances $\boldsymbol{\Sigma}_i$, based on the training data, as shown in Figure 9.1. Then we plug in these class densities into the Bayes theorem. This allows us to calculate the posteriors and the decision boundaries.

Let's look at a two class scenario. For class 1 the Bayes theorem gives us

$$p(1|x) = \frac{p(x|1)P(1)}{p(x)} = \frac{p(x|1)P(1)}{\sum_{l\in\{1,2\}} p(x|l)p(l)}. \tag{9.1}$$

The class density $p(x|1)$ is now estimated as a Gaussian distribution with mean $\boldsymbol{\mu}_1$ and covariance $\boldsymbol{\Sigma}_1$ (these parameters can be acquired using the Maximum Likelihood method).

The decision boundary should be where the posteriors of class 1 and class 2 are equal. So we set

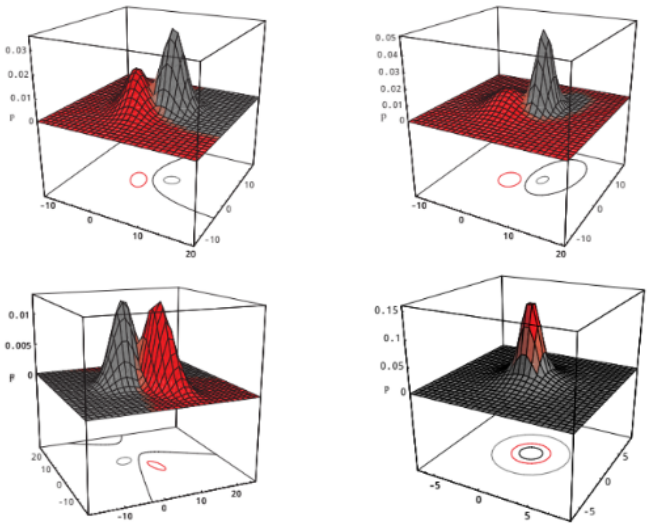
$$p(1|x) = p(2|x) \tag{9.2}$$
$$\rightarrow p(x|1)P(1) = p(x|1)P(1) \tag{9.3}$$

We insert the class densities and get:

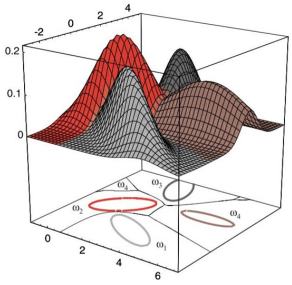
$$c_1 \exp\left[-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu}_1)^\top \boldsymbol{\Sigma}_1^{-1}(\boldsymbol{x}-\boldsymbol{\mu}_1)\right]\cdot p(1) = c_2 \exp\left[-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu}_2)^\top \boldsymbol{\Sigma}_2^{-1}(\boldsymbol{x}-\boldsymbol{\mu}_2)\right]\cdot p(2) \tag{9.4}$$

$$\underbrace{\ln\frac{c_1}{c_2}\frac{p_1}{p_2}}_{\text{scalar}} \underbrace{-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu}_1)^\top \boldsymbol{\Sigma}_1^{-1}(\boldsymbol{x}-\boldsymbol{\mu}_1) + \frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu}_2)^\top \boldsymbol{\Sigma}_2^{-1}(\boldsymbol{x}-\boldsymbol{\mu}_2)}_{\text{quadratic in } \boldsymbol{x}} = 0 \tag{9.5}$$

The decision boundaries that fulfill this equation are quadrics (ellipses, hyperboloids, parabolas, parallel lines), as shown in Figure 9.2a. For a higher number of classes, the decision boundaries can get more complex, as shown in Figure 9.2b.



(a) Different combinations of two Gaussians (red and gray) give different quadrics as decision boundaries (black).



(b) A combination of four Gaussians (red, dark gray, light gray, brown) can already result in a very complex decision boundary (black).

Figure 9.2: Figure from [DHS00].

Limitations:

- The Bayes classifier, which we derived in subsection 7.4 to be the best conceivable classifier is far from quadric. QDA is therefore an approximate technique.
- In the case of too little training data (small $\mathfrak{n}$, large $\mathfrak{p}$) we may have to reduce the number of parameters. There are different ways to do this:

  1. Linear Discriminant Analysis (LDA): Assume all Gaussians to have the same covariance $\mathbf{\Sigma}_1 = \mathbf{\Sigma}_2 = \mathbf{\Sigma}$. The quadratic terms in Equation 9.5 cancel and we are left with a linear decision boundary. A comparison of LDA and QDA is given in Figure 9.3

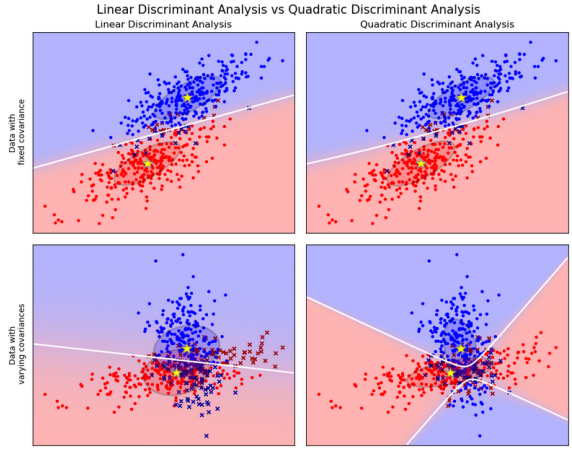
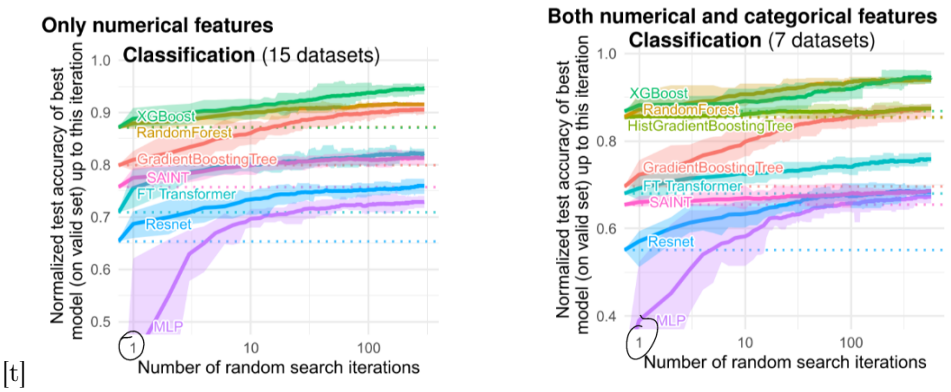


Figure 9.3: Application of LDA and QDA on two different data distributions. For the data with fixed covariance, QDA and LDA perform the same, both result in a linear decision boundary. Meanwhile, for the data with varying covariance, QDA produces a hyperbolic decision boundary and fits the data a lot better than LDA, which is confined to always fitting a linear decision boundary. Figure from [dev].

  2. Approximate the covariance matrix to be sparse. A sparse matrix is a matrix with only a few non-zero elements.
  3. Assume $\mathbf{\Sigma}_1$, $\mathbf{\Sigma}_2$ are diagonal matrices.

4. Assume isotropic Gaussians, i.e. $\boldsymbol{\Sigma}_1 = \sigma_1^2 \boldsymbol{I}$, $\boldsymbol{\Sigma}_2 = \sigma_2^2 \boldsymbol{I}$.
5. Assume $\boldsymbol{\Sigma}_1 = \boldsymbol{\Sigma}_2 = \sigma^2 \boldsymbol{I}$.
6. Do PCA (section 1) before doing QDA to reduce the number of dimensions.

9.2 Classification trees



[t]

(a) Tabular data with only numerical features.

(b) Tabular data with numerical features and categorical features.

Figure 9.4: Different models are tested and iteratively optimized by randomly trying out sets of hyperparameters. The test accuracy of these models is plotted over the number of iterations. The tree-based methods outperform the deep learning methods. Their performance with default parameters (1 iteration) especially are a lot better. Figure from [GOV22].

Tree-based models are still very successful on tabular data compared to deep learning models like MLPs (section 13, as shown in Figure 9.4). Tabular data is data that has no meaningful order of features, so that the features could be permuted without changing the information of the data. It can have numerical or categorical features. The tree-based models perform well on this data with default hyperparameters.

We focus here on classification, though there also exist regression trees. Classification trees use the following scheme:

1. We iteratively, greedily partition the feature space, which we can do based on different optimal criterions. Here, a greedy algorithm describes an algorithm that in each iteration makes the choice that momentarily appears best, and then moves on to the resulting subproblems, never reconsidering its choices. Usually, we resume splitting until each split only contains training samples of one class.

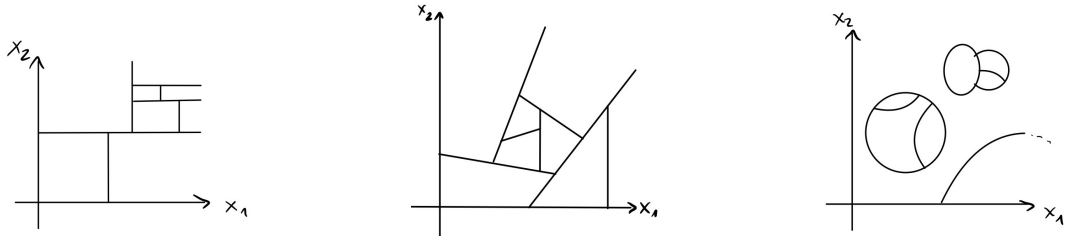
2. We estimate the class (posterior probability) for each partition based on the classes of the training samples inside it.

For the partitioning of feature space, there are different possibilities. Typical splits, as shown in Figure 9.5 are:

- Axis-orthogonal/"monothetic" splits: These work best for tabular data.
- Oblique splits: These work best for correlated observations.
- Splits using balls or spheres.

An example of a two-dimensional, two class problem with its decision tree and its monothetic splits in feature space can be seen in Figure 9.6.

The complexity of finding the optimal splits in the tree construction belongs to the class of "NP-hard" problems, meaning we have to use an approximate approach to solve it. We can use a "greedy sequential approach". For each node we find the split that reduces the uncertainty of a node's prediction the most. We then split the node in this way, to get the greatest uncertainty reduction. Typical uncertainty measures H are:

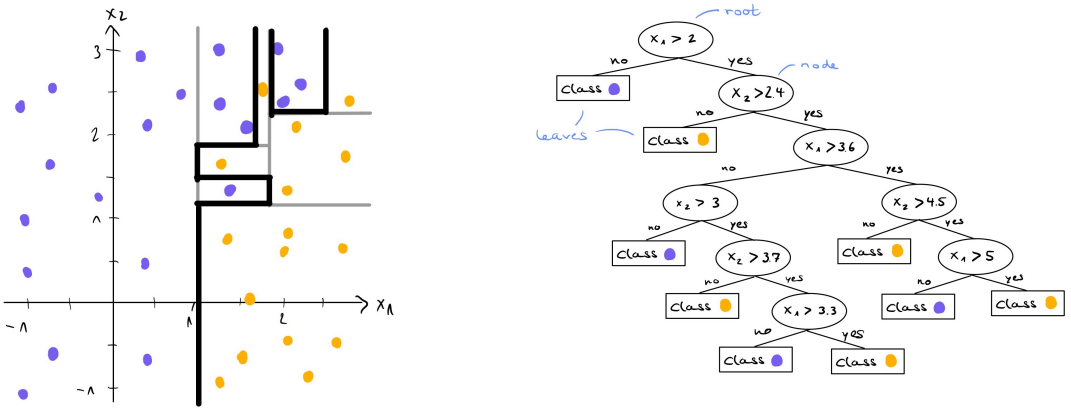


(a) axis-orthogonal / "monothetic" splits

(b) oblique splits

(c) splits using balls

Figure 9.5: Different ways of partitioning the feature space.



(a) Data points belonging to the two classes orange and purple are separated by the gray splits. This results in the black decision boundary.

(b) The tree structure corresponding to the splits.

Figure 9.6: An example of a classification tree to separate two-dimensional data into two classes.

- The entropy

$$H(\text{node}) = -\sum_y p(y|\text{node}) \log_2 p(y|\text{node}) \tag{9.6}$$

  where $p(y|\text{node})$ is the frequency of class y in the node.

- The misclassification error, given by

$$H(\text{node}) = 1 - \max_y p(y|\text{node}) \tag{9.7}$$

- The *Gini index* is the expected error when picking a random observation from the node and using its label as the prediction. The probability of picking the correct label y is $p(y|\text{node})$ and the probability of misclassification $\tilde{y}$ is $\sum_{\tilde{y}\neq y} p(\tilde{y}|\text{node}) = 1 - p(y|\text{node})$. The Gini index then is the sum of the products of these probabilities for all possible labels:

$$H(\text{node}) = \sum_y \left(p(y|\text{node}) \sum_{\tilde{y}\neq y} p(\tilde{y}|\text{node}) \right) \tag{9.8}$$

$$= \sum_y \left(p(y|\text{node}) - p(y|\text{node})^2 \right) = 1 - \sum_y p(y|\text{node})^2 \tag{9.9}$$

  It also captures nonlinear interactions.

A comparison of these uncertainty measures is shown in Figure 9.7.

The uncertainty reduction that is introduced by a split can be calculated in the following way

$$\Delta H = H(\text{node}) - \frac{L}{L+R} H(\text{left child}) - \frac{R}{L+R} H(\text{right child}). \tag{9.10}$$

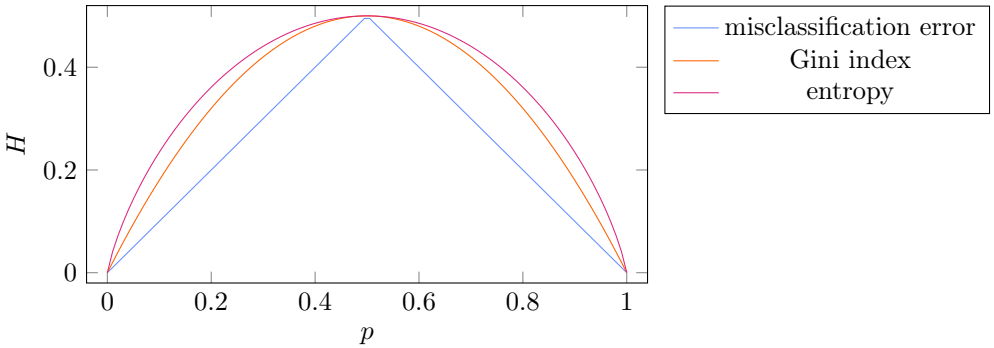


Figure 9.7: Comparison of the different uncertainty measures H as a function of the probability p in the case of a two class example. The entropy was scaled by a factor of 0.5.

Here, $L(R)$ is the number of observations assigned to the left (right) child of the node,

Classification trees are fast both in training and testing. They can fit complex decision boundaries, but at the same time tend to overfit.

9.3 Random forest

A random forest is an ensemble of decision trees. Every tree in the ensemble is created by using subsets of the data randomly. There are two sources of randomness that are needed:

1. *Bootstrap aggregation, "bagging":* Each tree sees a different, randomly sampled subset of the training data.
2. Each split (of every node on any tree) only has access to a small, random subset of the *features*. Therefore, when finding a split, not all options are tried out, only a few.

The random forest makes a prediction by taking a vote among the trees. It has the following useful features:

- The *out-of-bag error* gives an estimate of generalized performance. For each sample, the prediction is done with only the trees that did not use the sample during tree construction. The out-of-bag error is then the mean error of these predictions.
- The *Gini importance* measures the importance of features for monothetic trees by counting how often each feature was used to split the data. The Gini importance captures nonlinear effects, which can be seen in Figure 9.8. To find a subset of important features, nonlinear measures like the Gini importance work a lot better than linear ones.
- The ensemble of trees makes *query-by-committee* active learning possible, which was described in subsection 8.2.

The random forest has two hyperparameters we need to set before training, the number of trees in the forest (typically 256) and the size of the random feature subsets (typically $\log(\mathfrak{p})$). It is not well calibrated, meaning the votes of the ensemble do not give a good approximation of the posterior probability (the so called "Platt scaling" can help here). Training and testing is still quite fast. There are also other ways of combining classification trees than random forest, e.g. "extreme gradient boosting".

Summary

- QDA is a parametric, generative model for classification restricted to data on interval scale and can work well in higher dimensions.
- Classification Trees partition the feature space and make predictions based on the training samples found in each partition. They are fast, but tend to overfit.
- Random Forests are ensembles of decision trees that take a vote for predictions. They are still quite fast, can deal with ordinal and nominal scale features and especially have a high accuracy for tabular data.

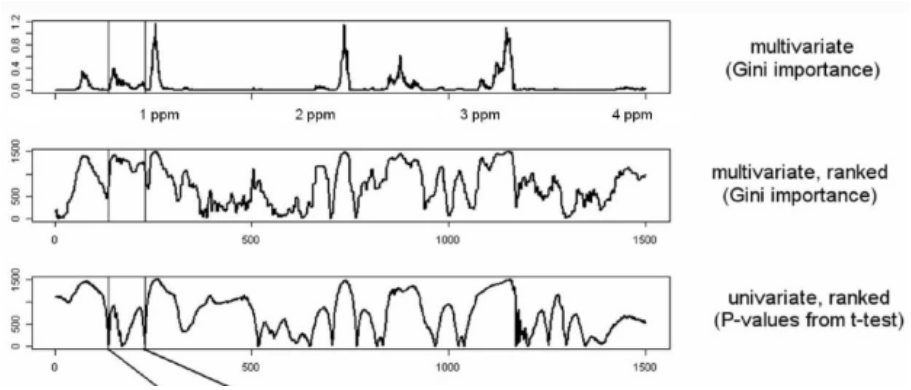


Figure 9.8: Gini importance of absolute values and of ranked values, and "p-values from t-test", a univariate measure, are plotted for a dataset of spectral data. The Gini importance differs from the univariate measure, because it captures non-linear effects. Figure from [Men+09].

Literature

- Trevor Hastie, Robert Tibshirani, and Jerome H Friedman, "The elements of statistical learning: data mining, inference, and prediction". 2009, Chap. 9.2 [HTF09]

10 Linear regression

The supervised learning problems we considered so far were all classification problems, i.e., given features x, we had to predict class labels y which could only take a finite number of values. Now we want to turn to so-called *regression problems* where we predict responses y given features x. In contrast to classification the responses y are continuous i.e. are on the interval scale (see subsection 6.1).

Applications of regression include:

- Interpolation and extrapolation
- Parameter estimation incl. their uncertainty
- Variable selection, i.e., determining most influential explanatory variables

The first ingredient in a regression model is the choice of a family of functions which models the relationship between explanatory variables and responses. The goal of regression then is to find the function within this family which best describes this relationship for a given dataset.

Example 10.1: Differential optical absorption spectroscopy (DOAS)

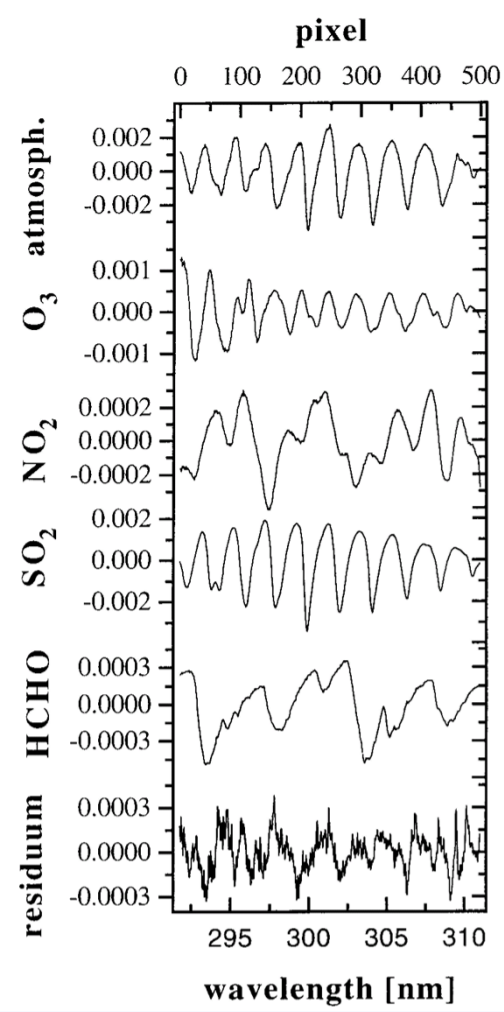


Figure 10.1: Absorption spectrum of the atmosphere and fitted reference spectra for individual trace gases as well as residual spectrum. Figure from [SP96].

One can choose even more complex basis functions for a linear regression model as described in [SP96]. Differential optical absorption spectroscopy is used to measure the concentration of trace gases in the atmosphere. To this end an absorption spectrum of the atmosphere $a(\nu)$ is measured. This spectrum is a superposition of all trace gases in the atmosphere (see Figure 10.1)

$$a(\nu) = \beta_1 a_{O_3}(\nu) + \beta_2 a_{NO_2}(\nu) + \dots \quad (10.1)$$

$a_{O_3}(\nu), a_{NO_2}(\nu), \dots$ are normalized spectra of the respective trace gases and the $\{\beta_i\}$ are factors which dependent on the concentration of the gas in the atmosphere. Thus, we are again in a linear regression setting where the explanatory "variables" are now functions. The solution $\hat{\boldsymbol{\beta}}$ of this problem yields a prediction for the desired gas concentrations. Figure 10.2 shows the distribution of NO_x which was measured using DOAS.

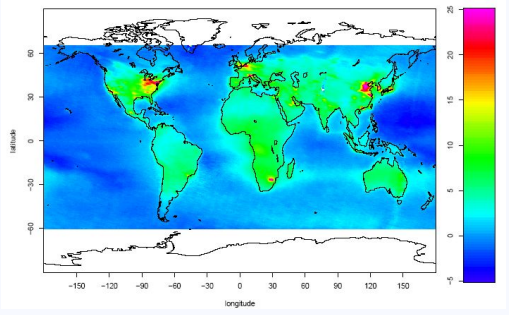


Figure 10.2: NO_x distribution as measured using DOAS. Figure from [SP96].

In the following we look at some examples for different regression models:

1. One of the simplest choices for the functional form of the responses y is to assume an affine dependence on the features x. For a two-dimensional feature space this can then be written as follows:
$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 \tag{10.2}$$
In particular, this model is also linear in the model parameters $\boldsymbol{\beta}$. Models with this property are an instance of *linear regression* which is the main topic of this chapter. Note that linear regression only requires linearity in the model parameters and not necessarily in the original features as the following examples show.

2. *Polynomial regression*
$$y = \beta_0 + \beta_1 x_+ \beta_2 x^2 \tag{10.3}$$
This model is linear in the parameters $\boldsymbol{\beta}$ which makes it an instance of linear regression. It is however quadratic in the original explanatory variable x which allows the model to fit nonlinear data distributions as shown in Figure 10.3.

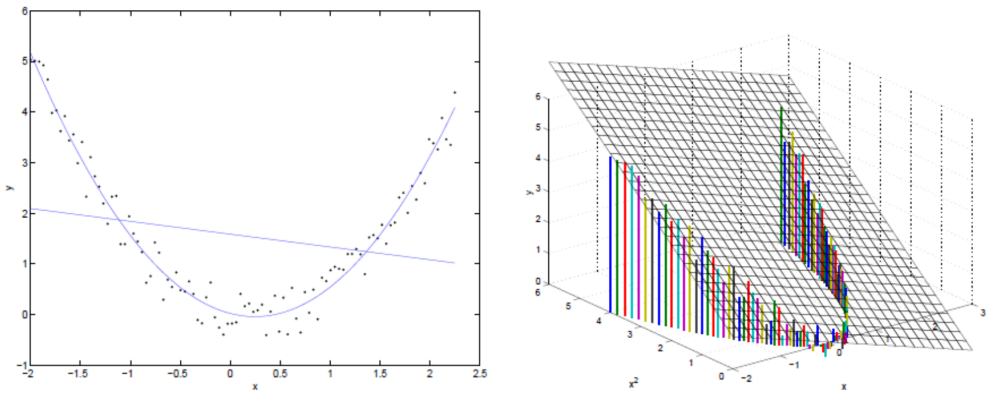


Figure 10.3: Through the choice of basis functions ($\{1, x, x^2\}$), the originally one dimensional data shown in the left panel is non-linearly projected to two dimensions as shown in the right panel. In this higher dimensional space, the data approximately lies within a plane (an affine subspace of 3D Euclidean space), which is why it can be fitted using linear regression.

3. *Radial basis function (RBF) regression*
$$y = \beta_0 + \beta_1 \exp\left(-\frac{1}{2\sigma_1^2}(x-\mu_1)^2\right) + \beta_2 \exp\left(-\frac{1}{2\sigma_2^2}(x-\mu_2)^2\right) + \ldots \tag{10.4}$$
This model is linear in the parameters $\{\beta_i\}$, but not in $\{\mu_i\}$ and $\{\sigma_i\}$. Thus, it is only an example for linear regression if the parameters $\{\mu_i\}$ and $\{\sigma_i\}$ are kept fixed. An example for RBF regression is shown in Figure 10.4.

4. Non-linear regression
$$y = \beta_0 + \beta_1 + \exp(\beta_2 x) \tag{10.5}$$
This is an example of a non-linear regression method since the parameter β_2 appears inside the exponential function.

Model assumptions for linear regression

Apart from the deterministic functional form of the model, we also have to choose an appropriate noise model to describe the given data, as real world data is always contaminated by noise. The underlying model of linear regression takes the following form

$$y(\boldsymbol{x}) = \boldsymbol{\beta}^{*T}\boldsymbol{x} + \epsilon(\boldsymbol{x}), \quad \boldsymbol{x} \in \mathbb{R}^{\mathfrak{p}+1}, \tag{10.6}$$

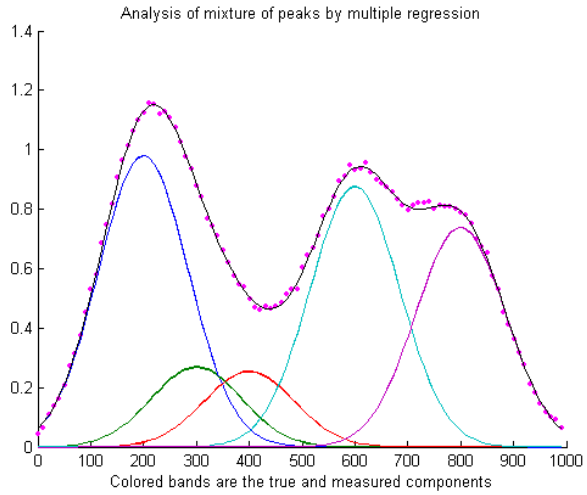


Figure 10.4: Regression fit using radial basis function regression. The points show the measured data and the black line is the fit which is the sum of the scaled radial basis functions shown as the colored curves. Figure from [OHa].

where ϵ is a random variable which models the noise of the data. The feature vector has $\mathfrak{p}+1$ elements since we are working with homogeneous coordinates, meaning that we add a new entry (w.l.o.g. as the first component x_0) to our feature vector which contains a constant 1. Thus, the corresponding entry in our parameter vector β_0 is an offset parameter. This choice of coordinates is used to simplify the notation.

We make the following further assumptions:

1. No error in explanatory variables $\boldsymbol{x}$
2. $\mathbb{E}\left[\epsilon(\boldsymbol{x})\right] = 0$
3. $\mathrm{Var}\left[\epsilon(\boldsymbol{x})\right] =$ constant independent of $\boldsymbol{x}$ (homoscedasticity)
4. $\epsilon(\boldsymbol{x})$ are independent and identically distributed (iid)

In order to find the optimal solution for $\boldsymbol{\beta}$, we have to choose an objective function which we want to minimize. The most popular choice in linear regression is the sum of squared residuals (SSQ)

$$SSQ = \sum_i (y_i - \sum_j [\boldsymbol{X}]_{ji}\,\beta_j)^2 = (\boldsymbol{y} - \boldsymbol{X}^T\boldsymbol{\beta})^T(\boldsymbol{y} - \boldsymbol{X}^T\boldsymbol{\beta}). \tag{10.7}$$

The optimization problem can then be written as

$$\hat{\boldsymbol{\beta}} = \arg\min_{\boldsymbol{\beta}} (\boldsymbol{y} - \boldsymbol{X}^T\boldsymbol{\beta})^T(\boldsymbol{y} - \boldsymbol{X}^T\boldsymbol{\beta}). \tag{10.8}$$

The optimal solution for $\boldsymbol{\beta}$ can be found by taking the derivative of the objective w.r.t. $\boldsymbol{\beta}$ and setting it equal to 0:

$$\frac{\partial SSQ}{\partial \boldsymbol{\beta}} = \frac{\partial}{\partial \boldsymbol{\beta}} (\boldsymbol{y}^T\boldsymbol{y} - 2\boldsymbol{\beta}^T\boldsymbol{X}\boldsymbol{y} + \boldsymbol{\beta}^T\boldsymbol{X}\boldsymbol{X}^T\boldsymbol{\beta})\Big|_{\boldsymbol{\beta}=\hat{\boldsymbol{\beta}}} \tag{10.9}$$

$$= -2\boldsymbol{y}^T\boldsymbol{X}^T + 2\hat{\boldsymbol{\beta}}^T\boldsymbol{X}\boldsymbol{X}^T \stackrel{!}{=} 0 \tag{10.10}$$

Solving for $\hat{\boldsymbol{\beta}}$ yields:

$$\hat{\boldsymbol{\beta}}^T\boldsymbol{X}\boldsymbol{X}^T = \boldsymbol{y}^T X^T \tag{10.11}$$

$$\iff \boldsymbol{X}\boldsymbol{X}^T\hat{\boldsymbol{\beta}} = \boldsymbol{X}\boldsymbol{y} \tag{10.12}$$

$$\iff \hat{\boldsymbol{\beta}} = (\boldsymbol{X}\boldsymbol{X}^T)^{-1}\boldsymbol{X}\boldsymbol{y} \tag{10.13}$$

where Equation 10.12 is known as the *normal equation*. We can now calculate the predictions for the responses of the original features as

$$\hat{\boldsymbol{y}} = \hat{\boldsymbol{\beta}}\boldsymbol{X} = \boldsymbol{y}\underbrace{\boldsymbol{X}^T(\boldsymbol{X}\boldsymbol{X}^T)^{-1}\boldsymbol{X}}_{:=\boldsymbol{H}} = \boldsymbol{y}\boldsymbol{H}, \tag{10.14}$$

where $\boldsymbol{H}$ is known as the *hat matrix*, since loosely speaking it "puts the hat on top of the $\boldsymbol{y}$". Using the hat matrix we can also express the residuals as

$$\boldsymbol{y} - \hat{\boldsymbol{y}} = \boldsymbol{y}(\boldsymbol{I} - \boldsymbol{H}). \tag{10.15}$$

The quality of the linear regression fit crucially depends on the distribution of the features. In particular, we do not want the covariance of the $\boldsymbol{\beta}$-estimate to become too large. In order to quantify this further, we calculate the covariance of $\hat{\boldsymbol{\beta}}$ as follows:

$$\begin{aligned}
\mathrm{Cov}_\epsilon(\hat{\boldsymbol{\beta}}) &= \mathbb{E}\left[(\hat{\boldsymbol{\beta}} - \mathbb{E}[\hat{\boldsymbol{\beta}}])(\hat{\boldsymbol{\beta}} - \mathbb{E}[\hat{\boldsymbol{\beta}}])^T\right] = \mathrm{Cov}_\epsilon((\boldsymbol{X}\boldsymbol{X}^T)^{-1}\boldsymbol{X}\boldsymbol{y}) && (10.16)\\
&= \mathrm{Cov}_\epsilon\left((\boldsymbol{X}\boldsymbol{X}^T)^{-1}\boldsymbol{X}\left(\boldsymbol{X}^T\boldsymbol{\beta}^* + \epsilon(\boldsymbol{x})\right)\right) && (10.17)\\
&= \mathrm{Cov}_\epsilon\left((\boldsymbol{X}\boldsymbol{X}^T)^{-1}\boldsymbol{X}\epsilon(\boldsymbol{x})\right) && (10.18)\\
&= (\boldsymbol{X}\boldsymbol{X}^T)^{-1}\boldsymbol{X}\sigma^2\boldsymbol{I}\boldsymbol{X}^T(\boldsymbol{X}\boldsymbol{X}^T)^{-1} = \sigma^2(\boldsymbol{X}\boldsymbol{X}^T)^{-1} && (10.19)
\end{aligned}$$

From Equation 10.17 to 10.18 we have used that $(\boldsymbol{X}\boldsymbol{X}^T)^{-1}\boldsymbol{X}\boldsymbol{X}^T\boldsymbol{\beta}^*$ is deterministic which means that its covariance vanishes. From Equation 10.18 to 10.19 we have used the identity

$$\mathrm{Cov}(\boldsymbol{A}\boldsymbol{\epsilon}) = \boldsymbol{A}\,\mathrm{Cov}(\boldsymbol{\epsilon})\boldsymbol{A}^T = \boldsymbol{A}\sigma^2\boldsymbol{I}\boldsymbol{A}^T \tag{10.20}$$

for a vector of iid. Gaussian random variables $\boldsymbol{\epsilon} \in \mathbb{R}^m$ with variance σ^2.

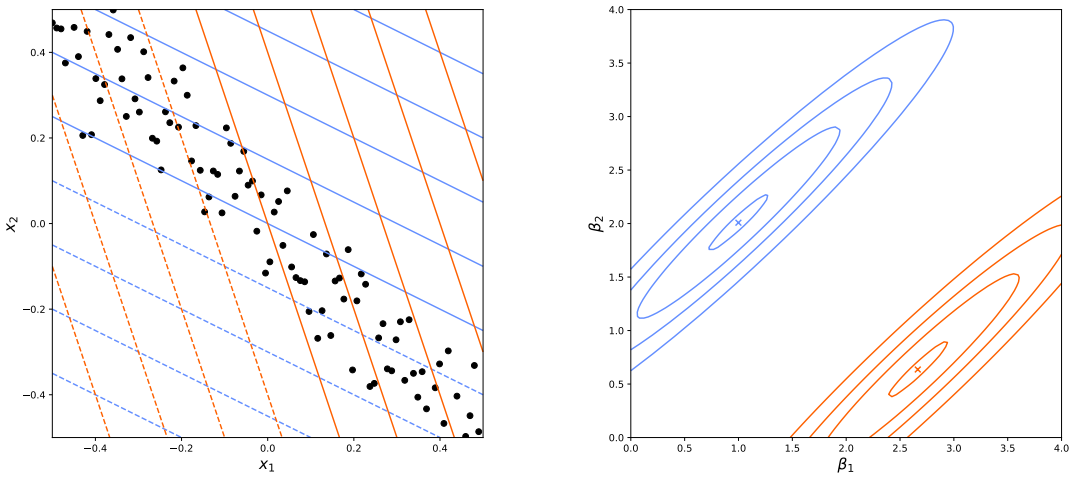


Figure 10.5: The left panel shows a distribution of two-dimensional input features marked with black dots. The blue and red lines show equicontours of two different linear functions which we want to fit using linear regression and the given input features. The right panel shows the distribution of the fitted $\hat{\boldsymbol{\beta}}$. We can clearly see that the uncertainty for $\hat{\boldsymbol{\beta}}$ is minimal in the direction where the original data distribution has its maximal spread. Also, the mean depends on the fitted function, whereas the shape of the distributions is independent of it.

We see that the covariance of $\hat{\boldsymbol{\beta}}$ scales with the inverse of the matrix $\boldsymbol{X}\boldsymbol{X}^T$. Note that in contrast to section 1 we are not necessarily working with centered data, meaning that in general this matrix does not correspond to the scatter matrix. It rather contains the so called "raw moments" which correspond to the covariance of the data around 0. The fact that we are working with homogeneous coordinates, i.e. the first row of $\boldsymbol{X}$ contains only 1s, further complicates the

interpretation of $\boldsymbol{X}\boldsymbol{X}^T$. Let us thus for a moment assume that our dataset is centered. In particular this means that the offset β_0 will be 0, so that we can work in non-homogeneous coordinates. For centered data $\boldsymbol{X}\boldsymbol{X}^T$ corresponds to the scatter matrix, i.e. is an approximation of the true covariance matrix of the distribution. Since the scatter matrix is symmetric we can diagonalize it. In its diagonal form the inverse is simply given by inverting the diagonal entries. That means that along (eigen-)directions where the spread of features is small compared to the variance of the noise $\boldsymbol{\sigma}$, the prediction for $\boldsymbol{\beta}$ will be uncertain. We also observe that the covariance does not depend on the targets $\boldsymbol{y}$ but only on the features $\boldsymbol{x}_i$. The relation between the covariance of fit parameters and the distribution of the training data is illustrated in Figure 10.5.

In the general case of uncentered data the meaning of $\boldsymbol{X}\boldsymbol{X}^T$ is harder to analyze but the main insight from above still holds: The parameters will be most uncertain in directions where the spread of data is small.

In the next chapter we will see how to deal with regression problems where the distribution of input features leads to unstable solutions for $\hat{\boldsymbol{\beta}}$.

Literature

- Trevor Hastie, Robert Tibshirani, and Jerome H Friedman, "The elements of statistical learning: data mining, inference, and prediction". 2009, Chap. 3.2 [HTF09]

11 Regularized regression

In linear regression we found

$$\beta = (\boldsymbol{X}\boldsymbol{X}^T)^{-1}\boldsymbol{X}\boldsymbol{y}. \tag{11.1}$$

We already saw in the last chapter that the matrix $\boldsymbol{X}\boldsymbol{X}^T$ is often nearly irregular, i.e., some of its eigenvalues are close to 0. Possible reasons for this include

- The number of features being larger than the number of samples $\mathfrak{p} > \mathfrak{n}$.
- There is collinearity among some explanatory variables.

Since in Equation 11.1 we invert $\boldsymbol{X}\boldsymbol{X}^T$, which in the diagonal basis involves dividing by almost vanishing eigenvalues, the solution for $\boldsymbol{\beta}$ becomes highly unstable. We thus need to regularize the linear regression problem.

Example 11.1: Computed tomography

One example where one needs to apply regularized regression is computed tomography (CT). It is used to obtain a 3D reconstruction of objects, perhaps best known for its many applications in medicine. The basic setup is illustrated in Figure 11.1.

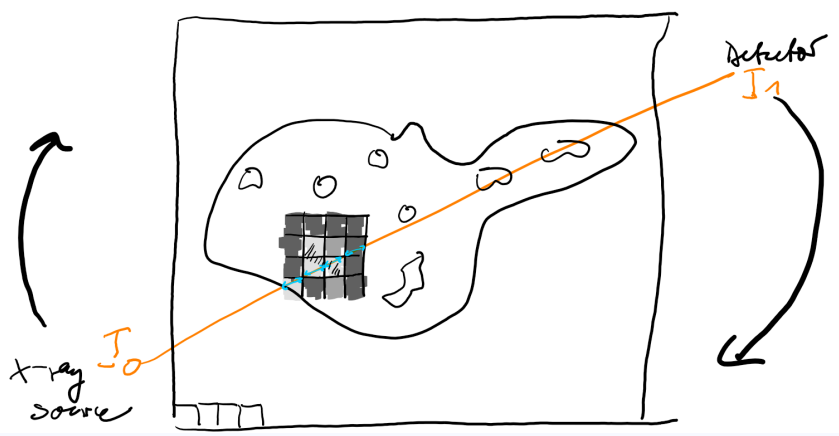


Figure 11.1: Setup of a CT scan.

The object to be examined is placed between an array of X-ray sources and an array of X-ray detectors. X-rays passing through the object loose energy proportional to the local absorption coefficient of the material μ. The radiation intensity reaching the detector I_1 can then be expressed as

$$I_1 = I_0 \exp\left(-\int \mu(l)\,\mathrm{d}l\right) \tag{11.2}$$

where I_0 is the radiation intensity of the source. The aim of CT is to find $\mu(x)$ for all coordinates x within the object. Equation 11.2 only yields a path integrated value for μ. Therefore, the arrays of X-ray sources and detectors can rotate around the object and many measurements from different angles are taken. For the analysis of the data, the object is represented as a fine grid of $\mathfrak{p}$ pixels where one assumes a constant absorption coefficient for each pixel. Through this discretization of space, Equation 11.2 may be rewritten as

$$y_i := \ln\left(\frac{I_{i,1}}{I_{i,0}}\right) = -\int \mu(l_i)\,\mathrm{d}l_i = \left[\boldsymbol{X}^T\boldsymbol{\beta}\right]_i. \tag{11.3}$$

The vector $\boldsymbol{\beta} \in \mathbb{R}^{\mathfrak{p}}$ has a number of components equal to the number of pixels and each entry of $\boldsymbol{\beta}$ contains the absorption coefficient of the corresponding pixel. $\boldsymbol{\beta}$ is thus the quantity one wants to know. $\boldsymbol{X}$ on the other hand is the so-called "design matrix" and is of the form $\mathfrak{p} \times \mathfrak{n}$ where $\mathfrak{n}$ is the number of measurements. It encodes how much each ray intersects with each pixel. As this information depends purely on the geometry of the setup, $\boldsymbol{X}$ is known.

Equation 11.3 is in the form of a standard regression problem. However, one typically has much more pixels than measurements, i.e., $\mathfrak{p} \approx 10^6 \gg 10^3 \approx \mathfrak{n}$ which is exactly one of the situations described at the beginning of this chapter where regularization is needed.

In the following we will look at three possible approaches to regularized regression.

11.1 Principal component regression (PCR)

The idea is to project the data onto an r dimensional subspace where $r < \mathfrak{p}$ using principal component analysis. In the cases where we need to regularize, we know that the lowest eigenvalues are close to 0 so that we do not lose a lot of information by applying PCA. At the same time, the eigenvalues in the projected subspace are non-vanishing as desired. After the projection, ordinary least squares can be applied in the subspace.

11.2 Ridge regression

In ridge regression, also known as Tikhonov regularization or weight decay, one adds a regularization term[2] to the objective function. This term penalizes large values of $\boldsymbol{\beta}$ which would result out of diverging terms in $(\boldsymbol{X}\boldsymbol{X}^T)^{-1}$,

$$\hat{\boldsymbol{\beta}}_{\text{ridge}} = \arg\min_{\beta} \|\boldsymbol{y} - \boldsymbol{X}^T\boldsymbol{\beta}\|_2^2 + \lambda\|\boldsymbol{\beta}\|_2^2. \tag{11.4}$$

The factor λ is called *regularization strength*. This objective can then be minimized analogously to the case of ordinary least squares.

$$\begin{aligned} 0 &\stackrel{!}{=} \frac{\partial}{\partial\boldsymbol{\beta}}\left(\boldsymbol{y}^T\boldsymbol{y} - 2\boldsymbol{y}^T\boldsymbol{X}^T\boldsymbol{\beta} + \boldsymbol{\beta}^T\boldsymbol{X}\boldsymbol{X}^T\boldsymbol{\beta} + \lambda\boldsymbol{\beta}^T\boldsymbol{\beta}\right) && (11.5)\\ &= -2\boldsymbol{y}^T\boldsymbol{X}^T + 2\boldsymbol{\beta}^T\boldsymbol{X}\boldsymbol{X}^T + 2\lambda\boldsymbol{\beta}^T && (11.6)\\ &= 2\boldsymbol{\beta}^T(\boldsymbol{X}\boldsymbol{X}^T + \lambda\boldsymbol{I}) - 2\boldsymbol{y}^T\boldsymbol{X}^T && (11.7)\end{aligned}$$

$$\implies \hat{\boldsymbol{\beta}}_{\text{ridge}} = (\boldsymbol{X}\boldsymbol{X}^T + \lambda\boldsymbol{I})^{-1}\boldsymbol{X}\boldsymbol{y}. \tag{11.8}$$

This is the same result we already had for ordinary least squares, but now we add the regularization strength λ to all diagonal elements of $\boldsymbol{X}\boldsymbol{X}^T$. As discussed in the previous chapter, the diagonal elements of $\boldsymbol{X}\boldsymbol{X}^T$ are the variances of the data (around 0) along the respective dimensions. Thus, by adding $\lambda\boldsymbol{I}$ the observations are "spread out" along all dimensions. This is illustrated in Figure 11.2 for a one dimensional example. Note that by adding the regularization we are reducing the variance of the fit parameters but also introduce a bias of $\boldsymbol{\beta}$ towards 0. This is also known as bias-variance tradeoff.

As we penalize large values of $\boldsymbol{\beta}$, its components converge to 0 as the regularization strength goes to infinity. This is shown in Figure 11.5a.

11.3 Lasso regression

Sometimes it is desirable to get a sparse solution for $\boldsymbol{\beta}$, e.g., for increased computational performance or because it logically makes more sense for the problem at hand. Ridge regression does not produce such sparse solutions as can be seen in Figure 11.5a.

To get closer to sparse solutions, we first mention that it is possible to reformulate Equation 11.4 as a constraint problem in the following way

$$\hat{\boldsymbol{\beta}}_{\text{ridge}} = \arg\min_{\boldsymbol{\beta}} \|\boldsymbol{y} - \boldsymbol{X}^T\boldsymbol{\beta}\|_2^2 \quad \text{s.t. } \|\boldsymbol{\beta}\|_2^2 \leq \gamma \tag{11.9}$$

for some appropriate $\gamma \geq 0$. The constraint imposed by the L_2 norm is isotropic and hence does not favor sparse solutions where one or more components would be identical to zero. We thus look

[2] Remember that we work in homogeneous coordinates (one component of the feature vector equal to 1) so that β_0 corresponds to an offset parameter. We usually do not want to regularize this parameter. In order to keep the notation simple we will however write $\lambda\|\boldsymbol{\beta}\|_2^2$ as regularization term as if we would regularize all components of $\boldsymbol{\beta}$.

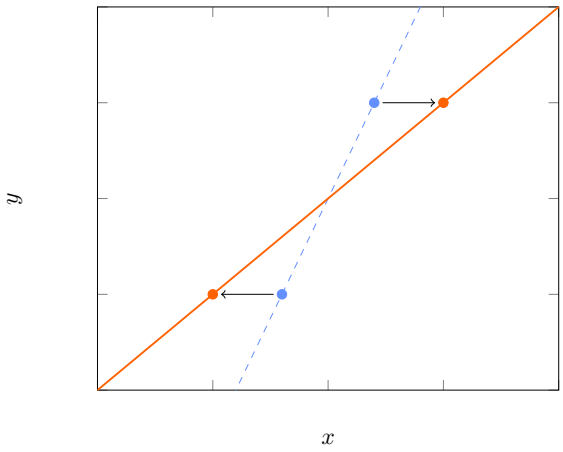


Figure 11.2: One-dimensional example for the increased scatter in the explanatory variable x. The blue dots represent the original data and the dashed line the OLS fit. The solid line represents the ridge regression estimate. It can be interpreted as OLS fit on modified data with increased variance.

for other norms we can use instead of L_2 which have the desired property. In general, the class of L_q norms is defined as

$$\|\boldsymbol{\beta}\|_q = \left(\sum_{i=1}^{\mathfrak{p}} |\beta_i|^q \right)^{1/q} . \tag{11.10}$$

The contour lines for $\|\boldsymbol{\beta}\|_q = 1$ for some values of q are shown in Figure 11.3.

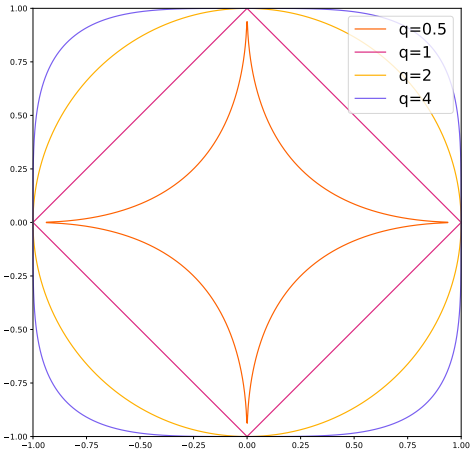
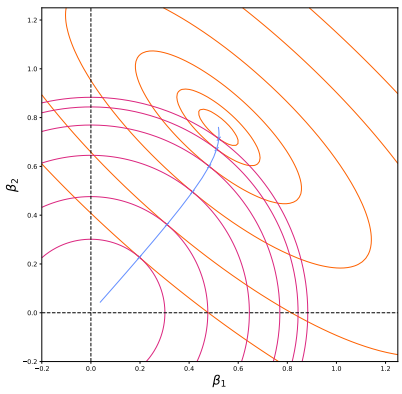


Figure 11.3: Contour lines for $(|x|^q + |y|^q)^{1/q} = 1$ shown for different values of q.

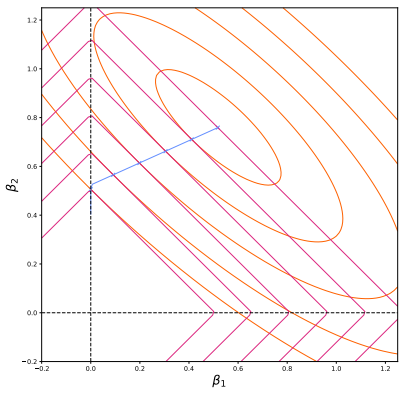
We are in particular interested in the norms with lower values of q where the contour lines in the directions of the coordinate axis are narrower. Thinking in terms of the constraint formulation makes clear that these norms are more likely to produce sparse solutions. The extreme case for $q \to 0$ would be the norm which just counts the number of non-zero components. We further would like our optimization problem to be convex (just like in ridge regression) as this makes the computation much faster. This is only the case if we choose a norm with $q \geq 1$. The compromise between these considerations is the choice of $\|\cdot\|_1$. The resulting regression method is called lasso ("least absolute shrinkage and selection operator") regression and can be written as

$$\hat{\boldsymbol{\beta}}_{\text{lasso}} = \arg\min_{\boldsymbol{\beta}} \|\boldsymbol{y} - \boldsymbol{X}^T\boldsymbol{\beta}\|_2^2 + \lambda\|\boldsymbol{\beta}\|_1 \tag{11.11}$$

The tendency to produce sparse solutions is illustrated in Figure 11.4.



(a) Solutions as obtained by ridge regression. Only for $\lambda \to \infty$ the components of $\boldsymbol{\beta}$ become 0



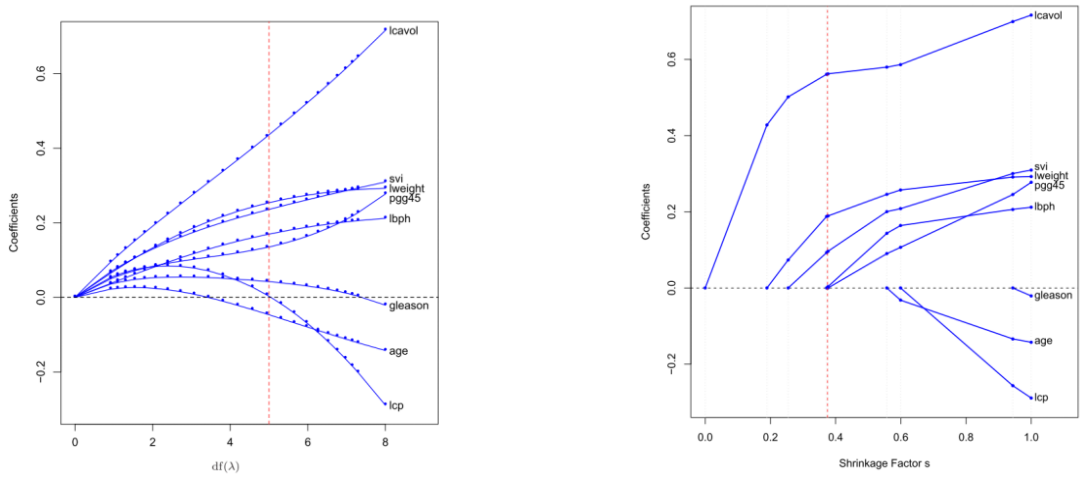
(b) Solutions as obtained by lasso regression. Already for a finite value of λ one of the components of $\boldsymbol{\beta}$ becomes 0

Figure 11.4: Optimal solutions for $\boldsymbol{\beta}$ in a two-dimensional toy problem as obtained by ridge regression and lasso regression respectively. The orange contour lines belong to the sum of squares term, whereas the red lines are the contour lines of the respective regularization term. The blue curve consists of optimal solutions to the respective regression problem for different values of the regularization strength λ.

Figure 11.5 shows another example with more parameters where lasso produces sparse solutions.

Literature

- Trevor Hastie, Robert Tibshirani, and Jerome H Friedman, "The elements of statistical learning: data mining, inference, and prediction". 2009, Chap. 3.4 [HTF09]



(a) Components of $\boldsymbol{\beta}_{\text{ridge}}$. $\text{df}(\lambda)$ denotes the effective degrees of freedom. The regularization strength grows from right to left. As $\text{df}(\lambda) \to 0$, i.e., $\lambda \to \infty$ all components of $\boldsymbol{\beta}$ converge to 0.

(b) Components of $\boldsymbol{\beta}_{\text{lasso}}$. The regularization strength grows from right to left. The individual components vanish for different finite values of the regularization strength λ.

Figure 11.5: The figure shows regression fits to data from a prostate cancer study published in [Sta+89]. Figure from [HTF09].

12 Logistic Regression

In this chapter we will discuss Logistic regression. Although it has "regression" in its name, it is actually a discriminative classification method. It is a useful method on its own but also provides the essential building blocks to construct simple neural networks, which we will discuss in detail in the next chapters.

12.1 Two-class case

Let us take a step back and look at the assumptions we had to make for linear regression. We first assumed that the data has Gaussian errors i.e. $y|\boldsymbol{x} \sim \mathcal{N}(\mu(\boldsymbol{x}), \sigma^2)$. Furthermore, we assumed the mean of the error distribution to be linear $\mu(\boldsymbol{x}) = \boldsymbol{\beta}^T\boldsymbol{x}$. Given these assumptions, we could apply the maximum likelihood principle to get an analytic solution for $\boldsymbol{\beta}$. This is illustrated in Figure 12.1.

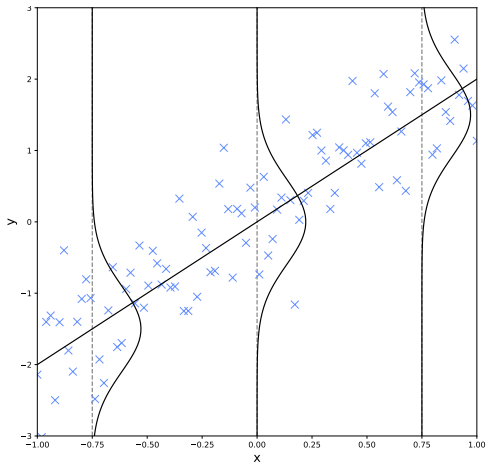


Figure 12.1: Linear regression fit to toy data.

For logistic regression, we follow the same conceptual steps. We will at first focus on a situation where the number of classes in our classification problem is two. The appropriate noise model for this scenario is given by a Bernoulli distribution, i.e.,

$$y|\boldsymbol{x} \sim \text{Ber}(\mu(\boldsymbol{x})). \tag{12.1}$$

The next question is which model we should assume for the mean $\mu(\boldsymbol{x})$. As the mean of a Bernoulli distribution, $\mu(\boldsymbol{x})$ is constrained to take values between 0 and 1. It is also reasonable to require that there is a smooth transition between these two extreme values. Popular functions which fulfill these requirements are

- tanh $\in (-1, 1)$ (renormalized to the interval $(0, 1)$)
- sigmoid[3] $\in (0, 1)$
- erf $\in (-1, 1)$ (renormalized to the interval (0,1))

For logistic regression, one chooses the sigmoid function commonly denoted as a σ:

$$\mu(\boldsymbol{x}) = \sigma(\boldsymbol{\beta}^T\boldsymbol{x}) = \frac{1}{1 + \exp(-\boldsymbol{\beta}^T\boldsymbol{x})} = \frac{\exp(\boldsymbol{\beta}^T\boldsymbol{x})}{\exp(\boldsymbol{\beta}^T\boldsymbol{x}) + 1} \tag{12.2}$$

This model is linear in the so called log-odds

$$\boldsymbol{x}^T\boldsymbol{\beta} = \ln\left(\frac{\mu}{1-\mu}\right) \tag{12.3}$$

where $\frac{\mu}{1-\mu}$ are the odds also called "logit".

[3]Note that we mean the logistic sigmoid function when we do not give further specifications.

Given these assumptions, we can again use the maximum likelihood method to derive a solution for $\boldsymbol{\beta}$. The likelihood is given by

$$\mathcal{L}(\boldsymbol{\beta}) = \prod_{i=1}^{n} \mu(\boldsymbol{x}_i)^{y_i} (1 - \mu(\boldsymbol{x}_i))^{1-y_i}. \tag{12.4}$$

Thus we can write the negative log-likelihood as

$$-\log \mathcal{L}(\boldsymbol{\beta}) = \sum_{i=1}^{n} -\log \left(\mu(\boldsymbol{x}_i)^{y_i}\right) - \log \left((1 - \mu(\boldsymbol{x}_i))^{1-y_i}\right) \tag{12.5}$$

$$= \sum_{i=1}^{n} -y_i \log \left(\mu(\boldsymbol{x}_i)\right) - (1 - y_i) \log \left(1 - \mu(\boldsymbol{x}_i)\right). \tag{12.6}$$

y_i encodes the class of the i-th sample and can take the values 0 and 1. The expression in Equation 12.6 is called "logistic loss", "log loss" or "binary cross-entropy". The functional form is illustrated in Figure 12.2 for $n = 1$.

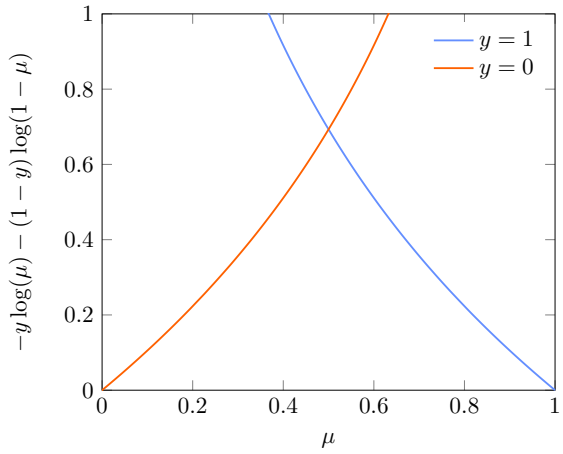


Figure 12.2: Logistic loss for one sample depending on the true class label of the sample y.

We see that there is a drastic penalty if the true class label y is very different from the prediction μ. In order to fit the model, the negative log likelihood is minimized according to

$$\hat{\boldsymbol{\beta}} = \arg\min_{\boldsymbol{\beta}} \sum_{i=1}^{n} -y_i \log \sigma(\boldsymbol{\beta}^T \boldsymbol{x}_i) - (1 - y_i) \log(1 - \sigma(\boldsymbol{\beta}^T \boldsymbol{x}_i)). \tag{12.7}$$

In contrast to other classification methods we discussed before, this problem cannot be solved analytically. Thus, numerical optimization techniques like gradient descent methods need to be applied. These will be explained in detail in the following chapters.

As a further comment, we note that if the data samples are linearly separable, the optimal solution for $\boldsymbol{\beta}$ will go to infinity, i.e., the sigmoid will approach a step function. This behavior is also illustrated in Figure 12.3. Furthermore, there will be an ambiguity of the location of the step function. To avoid these problems, one should apply some kind of regularization in this case.

12.2 Multi-class logistic regression

The next step is to extend the concept of logistic regression to more than two classes. To this end, it is convenient to represent the class labels as one-hot encoding that is a vector of dimensionality equal to the number of classes with a 1 in the position of the true class and 0 everywhere else,

$$[\boldsymbol{y}_i]_k = \begin{cases} 1 & \text{if } \boldsymbol{x}_i \text{ belongs to class } i \\ 0 & \text{else} \end{cases}. \tag{12.8}$$

The rest of the steps follows the same logic as for the two class case:

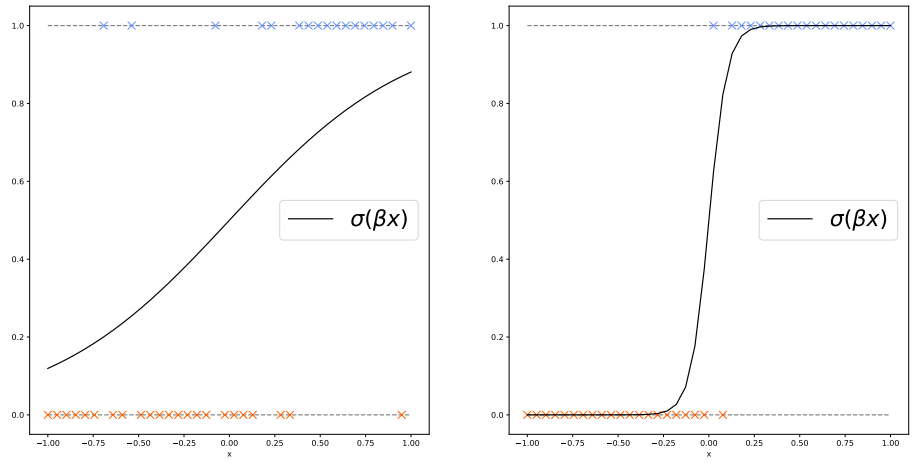


Figure 12.3: Data samples from a Bernoulli distribution with $y|x \sim \text{Ber}(\sigma(\beta x))$. The plot on the left shows samples for $\beta = 2$ the plot on the right for $\beta = 20$

- We first choose an appropriate noise model. In this case this is the categorical or multinoulli distribution, which models an experiment with $\mathfrak{K}$ possible outcomes of probabilities $p(y = k) = \mu_k \leq 1$ and $\sum_k \mu_k = 1$. We write

$$y|\boldsymbol{x} \sim \text{Cat}(\boldsymbol{\mu}(\boldsymbol{x})). \tag{12.9}$$

- Next, we choose a model for $\boldsymbol{\mu}(\boldsymbol{x})$. The multidimensional extension of the sigmoid is given by the softmax function[4]:

$$\mu_k(\boldsymbol{x}) = \frac{\exp(\boldsymbol{\beta}_k^T \boldsymbol{x})}{\sum_{j=1}^{\mathfrak{K}} \exp(\boldsymbol{\beta}_j^T \boldsymbol{x})} \tag{12.10}$$

  We recognize this expression as a Boltzmann distribution which we know from statistical mechanics. Consequently the term in the denominator is the corresponding partition function. This choice naturally fulfills the normalization conditions for $\boldsymbol{\mu}(\boldsymbol{x})$ required by the Categorical distribution. Due to summation constraint, $\boldsymbol{\beta}$ has only $\mathfrak{K} - 1$ independent components. An example of the softargmax is illustrated in Figure 12.4.

  The softargmax is also the minimum free energy solution to the following optimization problem:

$$\underbrace{\underset{\substack{\boldsymbol{z} \in \mathbb{S}^{\mathfrak{K}-1} \\ \uparrow \\ \text{simplex}}}{\arg\max}\ \boldsymbol{z}^T \begin{bmatrix} \boldsymbol{\beta}_1^T \boldsymbol{x} \\ \vdots \\ \boldsymbol{\beta}_{\mathfrak{K}}^T \boldsymbol{x} \end{bmatrix}}_{\text{vecargmax}} - \underbrace{\frac{1}{\lambda} \sum_{j=1}^{\mathfrak{K}} z_j \log z_j}_{\text{entropy regularization}}. \tag{12.11}$$

  A simplex is the generalization of a triangle in arbitrary dimensions. Here it ensures that the components of z sum up to 1. Without the entropy regularization term, the solution would just be the argmax function.

- For training, we again minimize the negative log-likelihood

$$\mathcal{L}(\boldsymbol{w} = \{\boldsymbol{\beta}_i\}_{i=1}^{\mathfrak{K}}) = -\frac{1}{\mathfrak{n}} \sum_{i=1}^{\mathfrak{n}} \log \left[\boldsymbol{\mu}(\boldsymbol{x}_i)\right]_{y_i} = -\frac{1}{\mathfrak{n}} \sum_{i=1}^{\mathfrak{n}} \log \frac{\exp(\boldsymbol{\beta}_{y_i}^T \boldsymbol{x}_i)}{\sum_{j=1}^{\mathfrak{K}} \exp(\boldsymbol{\beta}_j^T \boldsymbol{x}_i)}, \tag{12.12}$$

  where we included a conventional $\frac{1}{\mathfrak{n}}$ term which does not influence the solution. The resulting optimization objective is also called "cross entropy loss". This name is consistent with our

[4]This is another misnomer. It should be called "softargmax" as it converges to the argmax for $\beta \to \infty$.

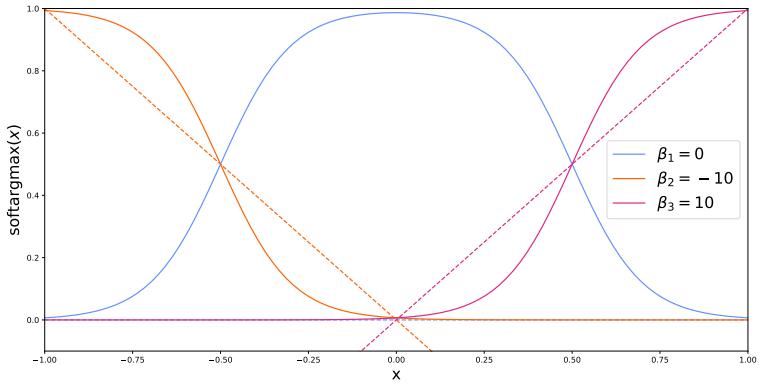


Figure 12.4: A three component softargmax function for a one dimensional input x. The solid lines depict the components of the softargmax, whereas the dashed lines show the functions $\beta_i^T x/10$. The input vector is appended with a constant 1 and the betas are 2-dimensional which allows for the shifts in x direction seen in the figure.

definition of cross entropy in section 5, as we can write

$$\mathcal{L}(\boldsymbol{w}) = -\frac{1}{\mathfrak{n}} \sum_{i=1}^{\mathfrak{n}} \log p(y_i|\boldsymbol{x}_i; \boldsymbol{w}) \approx \mathbb{E}_{y|\boldsymbol{x}\sim p^*(y|\boldsymbol{x})} \log p(y|\boldsymbol{x}; \boldsymbol{w}), \tag{12.13}$$

where we used the fact that the sum over the dataset is an approximation for the expectation value. This is the cross entropy between the distributions $p^*(y|\boldsymbol{x})$ and $p(y|\boldsymbol{x}; \boldsymbol{w})$.

We can also readily check that the multi-class logistic regression is a generalization of the two class case. To this end, we set $\mathfrak{K} = 2$, then we find for μ_1:

$$\mu_1(\boldsymbol{x}) = \frac{\exp(\boldsymbol{\beta}_1^T \boldsymbol{x})}{\exp(\boldsymbol{\beta}_1^T \boldsymbol{x}) + \exp(\boldsymbol{\beta}_2^T \boldsymbol{x})} = \frac{1}{1 + \exp\left((\boldsymbol{\beta}_2 - \boldsymbol{\beta}_1)^T \boldsymbol{x}\right)} \tag{12.14}$$

$$= \frac{1}{1 + \exp(\boldsymbol{\beta}\boldsymbol{x})} = \sigma(\boldsymbol{\beta}\boldsymbol{x}) = \mu(\boldsymbol{x}), \qquad \boldsymbol{\beta} \coloneqq \boldsymbol{\beta}_2 - \boldsymbol{\beta}_1 \tag{12.15}$$

which is the same expression we found earlier for the two class case.

Literature

- Bolin Gao and Lacra Pavel, "On the properties of the softmax function with application in game theory and reinforcement learning". 2017, [GP17]

13 Multilayer perceptrons

We start this section by recapitulating and comparing linear and logistic regression (see also Table 13.1). While the former is truly a regression model, logistic "regression" is a classifier. Linear regression assumes that the dependent variable is Gaussian distributed, whereas logistic regression assumes that all labels are true. The parameter is a vector $\boldsymbol{w} \in \mathbb{R}^{\mathfrak{p}}$ in the case of linear regres-

	Linear regression	Logistic regression
type of model	regression	classification
observation model	$y\|\boldsymbol{x}_i \sim N\left(\mu(\boldsymbol{x}_i;\boldsymbol{w}), \sigma^2\right)$	$\boldsymbol{y}\|\boldsymbol{x}_i \sim \mathrm{Cat}\left(\mu(\boldsymbol{x}_i;\boldsymbol{W})\right)$
linear model	$\mu(\boldsymbol{x}_i, \boldsymbol{w}) = \boldsymbol{w}^\top \boldsymbol{x}_i$	$\boldsymbol{\mu}(\boldsymbol{x}_i;\boldsymbol{W}) = \mathrm{soft\,max}(\boldsymbol{W}\boldsymbol{x}_i)$
loss	squared ℓ_2-loss: $\varepsilon(\boldsymbol{w}) = -\sum_{i=1}^{\mathfrak{n}} (y_i - \boldsymbol{w}^\top \boldsymbol{x}_i)^2$	cross-entropy loss: $\varepsilon(\boldsymbol{W}) = \sum_{i=1}^{\mathfrak{n}} \boldsymbol{y}_i^\top \cdot \ln \boldsymbol{\mu}(\boldsymbol{x}_i;\boldsymbol{W})$

Table 13.1: Compilation of linear and logistic regression.

sion, so the prediction of the model is $\boldsymbol{w}^\top$. Logistic regression contains a matrix $\boldsymbol{W} \in \mathbb{R}^{\mathfrak{K}\times\mathfrak{p}}$ as parameters, where $\mathfrak{K}$ is the number of possible classes. The prediction is $\mathrm{soft\,max}(\boldsymbol{W}\boldsymbol{x})$, so that its components are given by

$$[\boldsymbol{\mu}(\boldsymbol{x},\boldsymbol{W})]_k = \frac{\exp\left(\boldsymbol{w}_k^\top \boldsymbol{x}\right)}{\sum_{j=1}^{\mathfrak{K}} \exp\left(\boldsymbol{w}_j^\top \boldsymbol{x}\right)} \quad \text{with} \quad \boldsymbol{W} = \begin{bmatrix} — \, \boldsymbol{w}_1^\top \, — \\ \vdots \\ — \, \boldsymbol{w}_{\mathfrak{K}}^\top \, — \end{bmatrix}. \tag{13.1}$$

Finally, the linear regression uses the squared ℓ_2-loss, whereas logistic regression uses the cross-entropy

$$\varepsilon(\boldsymbol{W}) = -\sum_{i=1}^{\mathfrak{n}} \boldsymbol{y}_i^\top \cdot \ln \boldsymbol{\mu}(\boldsymbol{x}_i;\boldsymbol{W}), \tag{13.2}$$

whereby the logarithm is applied element-wise, such that each addend is a scalar product.

Diagrams

To visualize architectures of various machine learning models, electrical engineering-style diagrams are widely used. The data is entered on the left, processed in various ways and output in the right. We will introduce them by giving two examples.

i) Logistic regression (Figure 13.1): The components of the input data $\boldsymbol{x}$ are shown on the left. As done before, we appended a constant $x_0 = 1$ to allow for an offset or bias. Each of the components is multiplied by $\mathfrak{K}$ weights, which are represented by the numbers above the lines. The lines connect to a node that sums over all inputs (Σ). The softmax function takes these sums as input and produces $\mathfrak{K}$ outputs. This final output is identical to Equation 13.1.

ii) Binary logistic regression (Figure 13.2): We limit the logistic regression to two classes, $\mathfrak{K} = 2$. This means that we don't need to encode every class as its own component, but only need one output between 0 and 1. Note that instead of appending a constant x_0 to the input variables, we added a 1 underneath the sum node. It is scaled by the bias b. The output $\boldsymbol{w}^\top \boldsymbol{x} + b$ is fed into the sigmoid function σ.

13.1 Perceptrons

The diagram in Figure 13.2 is called perceptron. It constitutes the basic building block of neural networks. The parameters of a perceptron are the weights $\boldsymbol{w}$ and the bias b. It transforms the

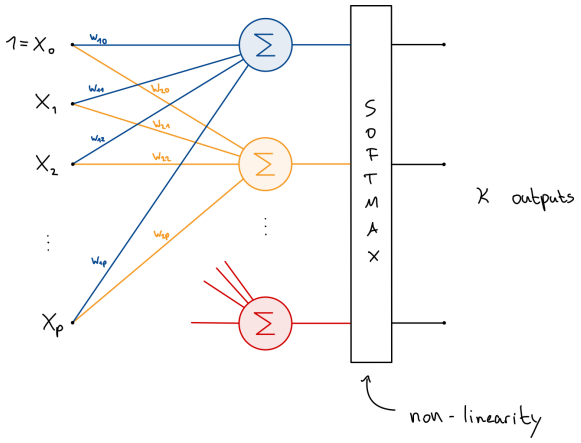


Figure 13.1: Diagram of logistic regression.

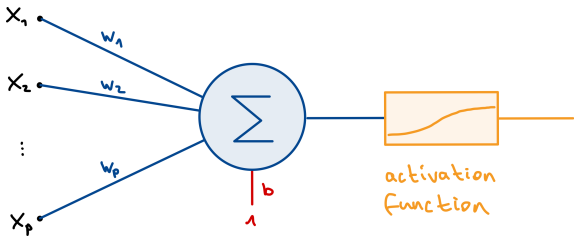


Figure 13.2: Diagram of a perceptron.

inputs linearly[5] and applies a non-linear *activation function* σ that is not limited to the sigmoid function. Together, the output is given by

$$y = \sigma\left(\boldsymbol{w}^\top \boldsymbol{x} - b\right). \tag{13.3}$$

A downside of two-class logistic regression is that it can only separate regions along one straight. Figure 13.3 shows one-dimensional sample data that can't be classifies by a single perceptron. The

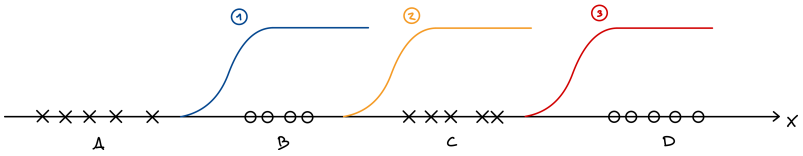


Figure 13.3: Example of data that can not be separated by one perceptron. One class is denoted by crosses, one by circles.

solution to this problem is to use more perceptrons that are arranged in multiple layers. This architecture is called *Multilayer Perceptron* (MLP). The input of an MLP is often referred to as *input layer* (even though not made of perceptrons), the last layer is the *output layer* and every intermediate layer of perceptrons is called *hidden layer*[6].

To classify the data from Figure 13.3, an MLP with three perceptrons in the first layer and one in the second layer is sufficient (see Figure 13.4). It consists of the input and output layer as well as one hidden layer. The weights and biases of the three perceptrons in the first layer are chosen such that the output looks like the graphs in Figure 13.3. The first perceptron is approximately 0 in region A and 1 in regions B, C and D. The second perceptron splits regions $\mathrm{A} \cup \mathrm{B}$ and $\mathrm{C} \cup \mathrm{D}$ while the last perceptron is 1 only in region D.

By using three perceptrons, we projected the one-dimensional input to a three-dimensional space spanned by the outputs (o_1, o_2, o_3). Together with the regions A, B, C and D, this space is

[5] Strictly speaking, the transformation is affine, but we follow the literature and call it linear.

[6] There are different naming conventions of input, output and hidden layers. Some libraries like PyTorch count the output layer as hidden layer.

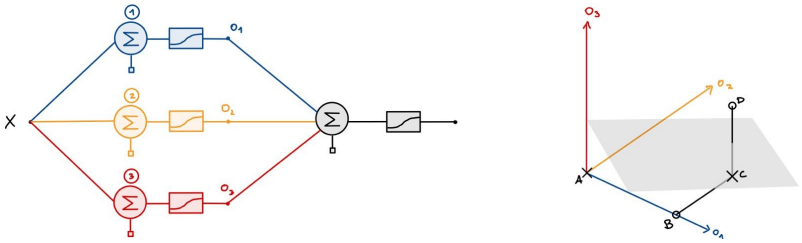


Figure 13.4: Left: MLP with three perceptrons in the first layer and one in the second layer. Right: Outputs of the first layer and plane that separates the two classes.

pictured in Figure 13.4 on the right. We see that the two classes are now separable by a plane, so that one perceptron is enough to classify the crosses and circles. Note that this was made possible by projecting the inputs non-linearly to a higher-dimensional space. The strength of this method and a theoretical rationale for its effectiveness will be outlined later.

AND/OR gate

It is useful to think about how classical techniques can be replicated by perceptrons. As an example, we consider AND as well as OR gates with $\mathfrak{p}$ inputs $\boldsymbol{x} \in \{0,1\}^{\mathfrak{p}}$. Both can be reproduced with a single perceptron as shown in Figure 13.2, whereby the logistic function is used as activation.

For the AND gate, all inputs have to be true ($= 1$) in order to output true. This is achieved by setting a negative bias so that the result of the linear transformation is only positive if $\boldsymbol{x} = \mathbf{1}_{\mathfrak{p}}$. Since we are using a sigmoid function as activation function, the arguments of the non-linearity should be far from 0 to get outputs close to 0 and 1. Taking these considerations into account, we can, for example, choose $\boldsymbol{w} = 100 \cdot \mathbf{1}_{\mathfrak{p}}$ and $b = -\mathfrak{p} \cdot 100 + 50$. If $\boldsymbol{x} = \mathbf{1}_p$, the perceptron outputs $\sigma(50) \approx 1$. In all other cases, the linear transformation yields $\boldsymbol{w}^\top \boldsymbol{x} + b \leq -50$ such that the output is approximately 0.

The OR gate with $\mathfrak{p}$ inputs outputs true if at least one of the inputs is true. Thus, we can choose a bias that does not depend on the number of inputs. Again, taking into account the above considerations, we can choose $b = -50$ and $\boldsymbol{w} = 100 \cdot \mathbf{1}_{\mathfrak{p}}$. If no input is true, the perceptron returns $\sigma(-50) \approx 0$, otherwise the argument of the sigmoid is at least 50 and the output is approximately 1.

13.2 Separation of arbitrary unions of polyhedra

We now have all the tools to constructively prove that a 3-layer perceptron can separate classes from arbitrary unions of polyhedra.

We first look at a two-class example shown in Figure 13.5. To allow for visual representation, the data are in $\mathbb{R}^2$. The first class 1 is represented by circles, the second class by crosses. We can draw a (non-convex) polyhedron that separates both classes. We call this polyhedron pure, as all points of class 2 are on the inside and all points of class 1 are on the outside.

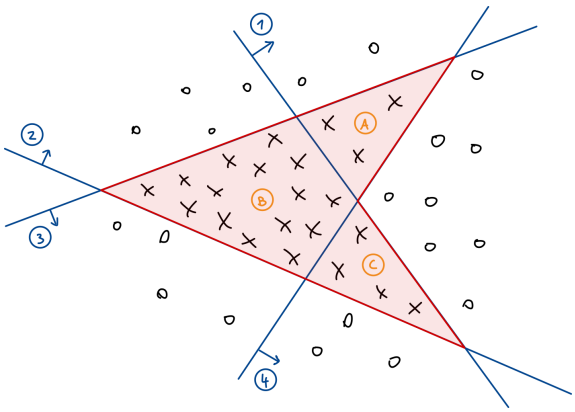


Figure 13.5: Sample data of two classes in $\mathbb{R}^2$.

The MLP architecture to classify the two classes is shown in Figure 13.6. It consists of three layers. The perceptrons of the first layer can separate the input space into two regions. In our current example, each of the four perceptrons of the first layer split $\mathbb{R}^2$ in into two half-planes. We can choose the parameters of the perceptrons so that the output transitions steeply between two half-planes. In other words, we can assume that the output is 0 on one side of the straight and 1 on the other. The half-plane that gets mapped to 1 is depicted by a small arrow in Figure 13.5.

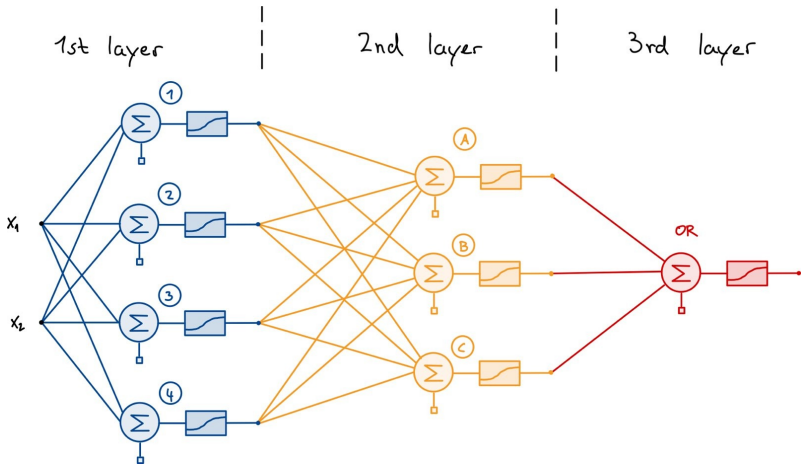


Figure 13.6: MLP to classify our toy data.

With the first layer, we non-linearly projected the two-dimensional input to a four-dimensional space. Each of the individual regions are in the input space is mapped to one corner of a four-dimensional hypercube, i.e., the output of the first layer is in $\{0,1\}^4$. For example, region A gets mapped to $[1,1,1,0]^\top$, region B to $[0,1,1,0]^\top$ and region C to $[0,1,1,1]^\top$. The hypercube has 16 corners, but there are only 11 regions. That's because some regions like $[1,0,0,1]$ do not exist.

Using n perceptrons in the first layer, what is the maximal number $R(n)$ of regions? If no lines are parallel, the n-th line intersects all $n-1$ previous lines. In doing so, it creates n new regions, $R(n) = R(n-1) + n$. Using $R(1) = 2$, we obtain

$$R(n) = 2 + \sum_{i=2}^{n} i = 1 + \frac{n(n+1)}{2} \tag{13.4}$$

regions. The n-dimensional hypercube $[0,1]^n$ has 2^n corners.

The second layer is used to select the appropriate regions. The parameters of the perceptrons in this layer can be chosen similarly to the AND/OR gate. For our concrete example, we need three perceptrons that each select one of the regions A, B and C. All that is left to do in the third layer is to use an OR perceptron to combine all regions. It is not necessary that the regions are adjacent. The above MLP architectures also work if one class is an arbitrary union of polyhedra.

We can summarize our steps in a constructive proof that a 3-layer perceptron can separate classes from arbitrary unions of polyhedra:

Algorithm 13.1: Separation of arbitrary unions of polyhedra

Partition feature space such that all resulting polyhedra are pure;
Identify polyhedra containing only samples of class 2. For each of these polyhedra, introduce a second layer perceptron that responds positively to the corresponding corner of the hypercube;
Add an OR perceptron to the third layer.

Literature

- Ian Goodfellow, Yoshua Bengio, and Aaron Courville, *Deep learning*. 2016, [GBC16]

14 Function counting theorem

In the last section, we have mapped points in the training/test set non-linearly to higher dimensions. Our hope is to make them linearly separable in the case of classification or linearly approximatable in the case of regression. But why should this work? The function counting theorem stated in [Cov65] provides a theoretical rationale for the case of classification. It is also known as Cover's theorem.

Theorem (Function counting theorem)**.** *The number of homogeneously linearly separable dichotomies (HLSD) of* $\mathfrak{n}$ *points in general position in* $\mathfrak{p}$ *dimensions is*

$$C(\mathfrak{n}, \mathfrak{p}) = 2 \sum_{k=0}^{\mathfrak{p}-1} \binom{\mathfrak{n}-1}{k}. \tag{14.1}$$

To understand the theorem, we first have to explain a few terms. Firstly, a *dichotomy* of a (finite) set is a partition into two disjoint sets. In other words, we assign each element one of two classes. If the set contains $\mathfrak{n}$ points, there are $2^{\mathfrak{n}}$ dichotomies. A set of points is in *general position* in $\mathfrak{p}$ dimensions if no subset of $\mathfrak{p}$ points lies on a $(\mathfrak{p}-1)$-dimensional hyperplane. The points are *linearly separable*, if there exists a $(\mathfrak{p}-1)$-dimensional hyperplane that separates the two classes of the dichotomy. The separation is *homogeneous*, if the hyperplane goes through the origin. We can represent a homogeneous linear separation by a vector $\boldsymbol{w} \in \mathbb{R}^{\mathfrak{p}}$ that is orthogonal to the hyperplane. The hyperplane separates the two classes if $\boldsymbol{w}^\top \boldsymbol{x} > 0$ for all points $\boldsymbol{x}$ in class 1 and $\boldsymbol{w}^\top \boldsymbol{y} < 0$ for all points $\boldsymbol{y}$ in class 2.

Consider for example three points in two-dimensional space, 6 out of 8 dichotomies are homogeneously linearly separable (see Figure 14.1). In the above arrangement, the three points are homogeneously separable exactly when all points belong to the same class or when an outer point belongs to a separate class. The former contributes two dichotomies, the latter four. The remaining

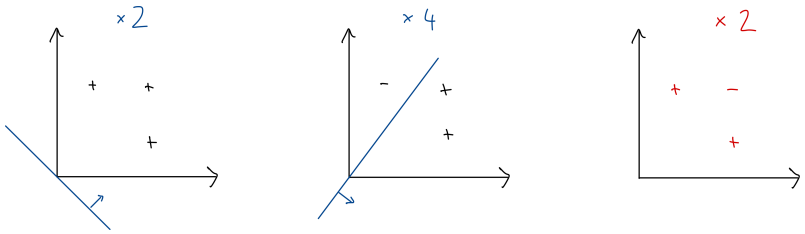


Figure 14.1: Dichotomies of three points in two-dimensional space. The dichotomy on the right is not homogeneously separable.

two dichotomies, where the middle point belongs to a different class than the neighboring points, are not homogeneously separable. We note that the number of HLSD does not depend on the arrangement of points, as the theorem holds in general.

How many HLSD are there in general? Let's say there are $C(\mathfrak{n}, \mathfrak{p})$ HLSD. We can track what changes if the number of points increases. If we add one more point, we obtain

$$C(\mathfrak{n}+1, \mathfrak{p}) = \underbrace{C(\mathfrak{n}, \mathfrak{p})}_{(1)} + \underbrace{C(\mathfrak{n}, \mathfrak{p}-1)}_{(2)} \tag{14.2}$$

for $\mathfrak{n}, \mathfrak{p} \in \mathbb{N}$. We explain the two contributions separately.

(1) This is the number of HLSD we already had. If the new point is simply assigned to a class according to the "old" decision boundary, we obtain a separable dichotomy. This corresponds to "Assign label that works in any case".

(2) Sometimes also the class that is opposite to the previous separation can be assigned to the new point. To obtain the number of these dichotomies, we consider the set of $\mathfrak{p}-1$ dimensional hyperplanes which include the origin and the new point. If the other $\mathfrak{n}$ points can still be separated by one of the constrained hyperplanes in the set, we can nudge the respective decision boundary such that the new point falls either in the region for class 1 or class 2. This is possible due to the assumption that the points are in general position.

The constraint removes one degree of freedom so that the additional number of dichotomies is $C(\mathfrak{n}, \mathfrak{p}-1)$. Another way to think about this is to project all points onto a hyperplane along the direction of the new point. If the other $\mathfrak{n}$ points are homogeneously separable in this $(\mathfrak{p}-1)$-dimensional space, we obtain a separation in the original $\mathfrak{p}$-dimensional space that goes through the additional point.

We can use Equation 14.2 to recursively reduce the number of points inside the expression $C(\mathfrak{n}, \mathfrak{p})$ to $\mathfrak{n} = 1$.

$$\begin{aligned} C(\mathfrak{n}, \mathfrak{p}) &= C(\mathfrak{n}-1, \mathfrak{p}) + C(\mathfrak{n}-1, \mathfrak{p}-1) \\ &= C(\mathfrak{n}-2, \mathfrak{p}) + 2C(\mathfrak{n}-2, \mathfrak{p}-1) + C(\mathfrak{n}-2, \mathfrak{p}-2) = \dots \end{aligned} \tag{14.3}$$

We need to apply the relation $(\mathfrak{n}-1)$ times. The number of dimensions $\mathfrak{p}$ either stays constant or gets decreased by one. To decrease the dimension by k, we need to choose the term (2) k out of $(\mathfrak{n}-1)$ times. Thus, in the final result, the term $C(1, \mathfrak{p}-k)$ occurs $\binom{\mathfrak{n}-1}{k}$ times. The dimension can be reduced to one at most, since no separable dichotomies exist in zero-dimensional space, so k runs form 0 to $\mathfrak{p}-1$. Together, we obtain

$$C(\mathfrak{n}, \mathfrak{p}) = \sum_{k=0}^{\mathfrak{p}-1} \binom{\mathfrak{n}-1}{k} C(1, \mathfrak{p}-k) \tag{14.4}$$

$$= 2 \sum_{k=0}^{\mathfrak{p}-1} \binom{\mathfrak{n}-1}{k}, \tag{14.5}$$

where we have used $C(1, d) = 2$ for $d \in \mathbb{N}$, as all dichotomies with on point in d dimensions are homogeneously linearly separable. This proves the theorem.

To realize the importance of the function counting theorem, we will take a look at Figure 14.2. It shows the fraction of HLSD of all possible dichotomies for different ratios of $\mathfrak{n}/\mathfrak{p}$. This fraction

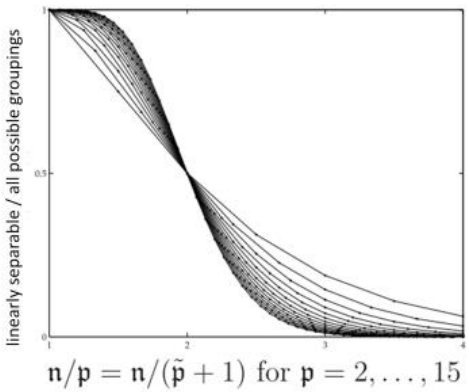


Figure 14.2: Fraction of HLSD of all possible dichotomies. The bigger $\mathfrak{p}$, the more S-shaped the curve gets.

is above one half for $\mathfrak{n}/\mathfrak{p} < 2$ and converges to one if $\mathfrak{n}/\mathfrak{p} \to 1$. This implies that projecting points into higher dimensions increases the probability that the points are linearly separable. In summary, the function counting theorem tells us to

- decrease the $\mathfrak{n}/\mathfrak{p}$ ratio,
- put the points in general position.

14.1 Activation functions

So far, we only used a sigmoid function as activation function for MLPs. But the activation function of a hidden layer can be almost everything, as long as it is non-linear. Possible choices are displayed in Figure 14.3. One important candidate is ReLU, which we will look at in more

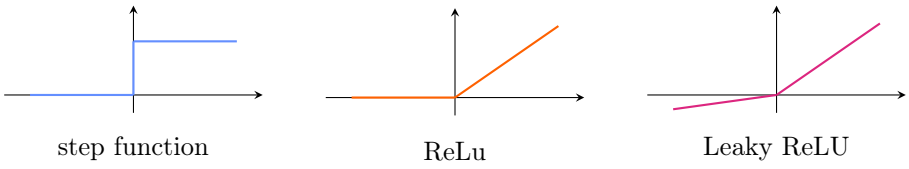


Figure 14.3: Graph of possible activation functions.

detail below. It stands for "Rectified Linear Unit" and is defined as $\mathrm{ReLU}(x) = \max(0, x)$. Other examples are the absolute function, leaky ReLU, step function, tanh, $\exp(\mathrm{ReLU})$, ...

The activation function of the last layer can't be chosen freely. If we use an MLP as a regression model, we use the ℓ_2 loss. The corresponding activation function of the output layer is linear. If we are dealing with classification, a suitable loss is the cross-entropy. In this case, the corresponding activation function of the last layer is soft (arg) max.

14.2 Linear regions of a ReLU network

If we choose ReLU as the activation function, the resulting MLP will give output that is a piecewise function of its input. Figure 14.4 shows a three layer MLP, its parameters and the output after each layer.

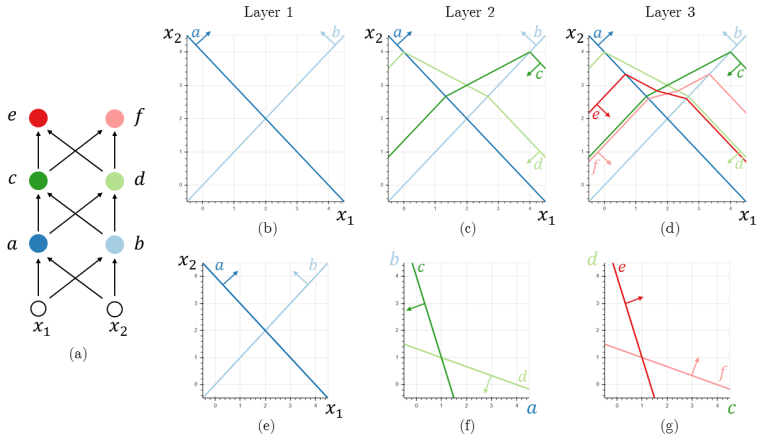


Figure 14.4: ReLU MLP. Figure from [STR17].

Figure 14.5 shows an MLP with the absolute value as activation function. Every layer folds its input into one quadrant. That way, the preimage of the line in the final output are 16 lines in input space.

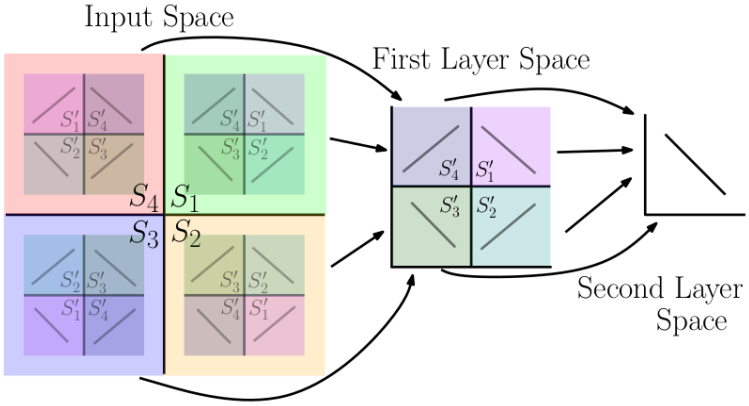


Figure 14.5: MLP with absolute value as activation function. Figure from [Mon+14].

The two examples above illustrate, that the number of linear regions grows exponentially with the number of layers. This leads us to the important insight that for a given number of parameters, deep MLPs induce more polytopes than shallow MLPs.

Literature

- Thomas M. Cover, "Geometrical and Statistical Properties of Systems of Linear Inequalities with Applications in Pattern Recognition". 1965, [Cov65]

15 Training of neural networks

When we talked about MLPs so far, we assumed that the optimal parameters were given. This is obviously not the case in real world scenarios. Instead, the parameters $\boldsymbol{w}$ are obtained by minimizing the objective

$$\arg\min_{\boldsymbol{w}} \sum_{i \in \mathcal{D}} \mathcal{L}\left(\boldsymbol{y}_i, \boldsymbol{f}(\boldsymbol{x}_i; \boldsymbol{w})\right) + \Omega_{\text{architecture}}(\boldsymbol{w}) + \Omega_{\text{output}}(\boldsymbol{f}(\boldsymbol{x}_i; \boldsymbol{w})). \tag{15.1}$$

We explain the individual terms of Equation 15.1 in more detail below. Since MLPs show a correspondence with biological models of brains, they are also called neural networks (NN). This also fits with the fact that optimizing parameters based on training data $\mathcal{D}$ is called "learning". From now on, we will use the name neural network instead of multilayer perceptrons.

15.1 Architecture

All network architectures considered here are feed-forward NNs. There, outputs of a layer can only connect to following layers. Feed-forward NNs can be described as directed acyclic graphs. This is in contrast to recurrent NNs where outputs can be used as inputs of the current or previous layers. Their graph representation contains cycles. A popular example is the Long short-term memory (LSTM).

Another import part of the architecture are the activation functions. One popular choice as of December 2022 is to use the Gaussian Error Linear Unit (GELU). It is defined as

$$\text{GELU}(x) = x \cdot \Phi(x) \quad \text{where} \quad \Phi(x) = \int_{-\infty}^{x} \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{y^2}{2}\right) \mathrm{d}y. \tag{15.2}$$

GELU can be seen as a smooth version of ReLU. Both functions are plotted in Figure 15.1.

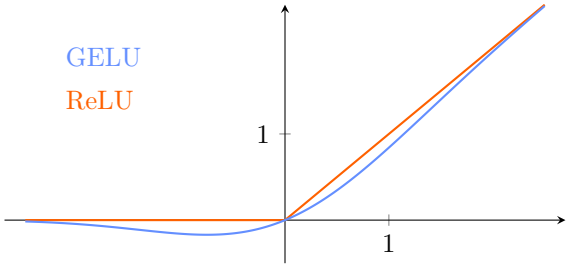


Figure 15.1: Plot of popular activation functions GELU and ReLU.

On the one hand, the right choice of activation function can improve the performance of the NN. But on the other hand, perceptrons can replicate the behavior of other activation. For instance, the absolute function can be expressed by two perceptrons, as shown in Figure 15.2. In fact, a neural

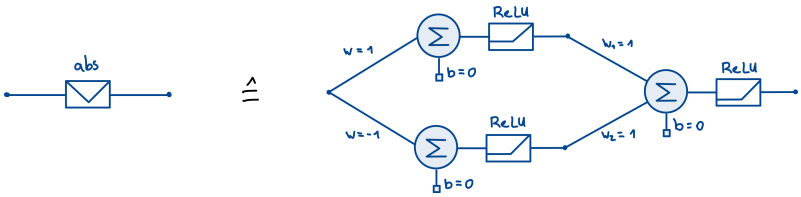


Figure 15.2: Absolute value expressed by ReLU perceptrons.

network using ReLU activation can approximate every function by linear regions. We justify this claim in the one-dimensional case, the extension to higher dimensions follows similar arguments.

The derivative of ReLU is a step function. The step can be mirrored by a negative weight and shifted by the bias. A perceptron in a second layer scales the steps and adds them up. The output of the perceptron in the second layer is a function whose derivative has n steps. Thus, the function itself is a piecewise linear function. Finally, a constant can be added by the bias of the second layer.

15.2 Loss function

As discussed before, the appropriate loss for regression is the ℓ_2-norm. If data is to be classified, the cross-entropy

$$-\sum_{i=1}^{\mathfrak{K}} [\boldsymbol{y}]_i \cdot \ln [\boldsymbol{f}(\boldsymbol{x}; \boldsymbol{w})]_i = -\boldsymbol{y}^\top \cdot \ln \boldsymbol{f}(\boldsymbol{x}; \boldsymbol{w}) \tag{15.3}$$

is to be used. Here, $\boldsymbol{f}(\boldsymbol{x}; \boldsymbol{w})$ is the prediction of the model and $\boldsymbol{y}$ the ground truth. The ground truth of a classification problem is usually encoded in one-hot vectors, but the cross-entropy works for every $\boldsymbol{y} \in \mathbb{R}^{\mathfrak{K}}$. The technique of assigning a certain probability to several classes instead of using a one-hot vector is called "label smoothing".

Regularizers

Just as with linear regression, regularization can help the model make better predictions and not overfit. Equation 15.1 contains two regularizers. $\Omega_{\text{architecture}}$ regularizes the weights and biases and prevents them from diverging during training, for instance. A common example is the weight decay, given by

$$\Omega_{\text{architecture}} = k \cdot \|\boldsymbol{w}\|_2, \tag{15.4}$$

where k is a hyperparameter. It is also possible to regularize the output of the model. The term

$$\Omega_{\text{output}} = -\eta \cdot H(\boldsymbol{f}(\boldsymbol{x}, \boldsymbol{w})) \tag{15.5}$$

was proposed in [Per+17]. Since entropy is maximized by a uniform distribution, Ω_{output} is minimized by a uniform output. Therefore, one-hot predictions are penalized and Ω_{output} is called "confidence penalty".

Data set

Training of neural networks requires large sets of labeled training data. It is often difficult to acquire enough labeled data to train a model for a specific problem. This problem gave rise to the development of the following techniques:

- pretraining
- self-supervision
- data augmentation

15.3 Optimizer

The optimizer is responsible for the actual learning. The following methods update the weights and biases (both included in the parameters $\boldsymbol{w}$) in an iterative process. A training cycle over all training data is called *epoch*. Out of simplicity, we consider an unregularized model.

- *Gradient descent*: In each iteration, compute the gradient of the loss with respect to the parameters. Move the parameters in the direction of the negative gradient.

$$\begin{aligned} \boldsymbol{w}^{t+1} &= \boldsymbol{w}^t - \alpha \boldsymbol{g}^t \\ \boldsymbol{g}^t &= \nabla_{\boldsymbol{w}} \sum_{i \in \mathcal{D}} \mathcal{L}(\boldsymbol{y}_i, \boldsymbol{f}(\boldsymbol{x}_i; \boldsymbol{w}^t)) \end{aligned} \tag{15.6}$$

  The hyperparameter $\alpha > 0$ is called "learning rate".

- *Stochastic gradient descent (SGD)*: Instead of using the entire data set $\mathcal{D}$ at once, it is divided randomly into *mini-batches* $\mathcal{B} \subset \mathcal{D}$ in each epoch. The gradient is computed after processing the data of one mini-batch $\mathcal{B}$. Thus, the parameters get updated multiple times per epoch. SGD is empirically better than standard gradient descent, as it can escape local minima. The reason for that is that the path of the parameters undergoes a "jittering" due to the changing loss surface.

- *Momentum*: Rather than always moving in the direction of the new negative gradient, just modify the previous direction. This introduces the analogy of a momentum.

$$\begin{aligned} \boldsymbol{w}^{t+1} &= \boldsymbol{w}^t - \alpha \boldsymbol{m}^t \\ \boldsymbol{m}^t &= \beta \boldsymbol{m}^{t-1} + (1-\beta)\boldsymbol{g}^t \end{aligned} \tag{15.7}$$

  The momentum exponentially smooths the gradient in time. An adequate value for the new hyperparameter is $\beta \sim 0.9$.

- *ADAM*: Rescale the learning rate for each single parameter.

$$\begin{aligned} \boldsymbol{w}^{t+1} &= \boldsymbol{w}^t - \alpha \frac{\hat{\boldsymbol{m}}^t}{\sqrt{\hat{\boldsymbol{v}}} + \varepsilon} \\ \boldsymbol{m}^t &= \beta \boldsymbol{m}^{t-1} + (1-\beta)\boldsymbol{g}^t \\ \boldsymbol{v}^t &= \gamma \boldsymbol{v}^{t-1} + (1-\gamma)(\boldsymbol{g}^t)^2 \end{aligned} \tag{15.8}$$

  Here, $(\boldsymbol{g}^t)^2$ is the element-wise square. The vector $\boldsymbol{v}^t$ has the effect of reducing the learning rate for steep gradients. The hyperparameters are constrained by $\beta, \gamma \in [0, 1)$; typical values are $\beta \sim 0.9$ and $\gamma \sim 0.999$. The initialization $\boldsymbol{m}^0 = 0$, $\boldsymbol{v}^0 = 0$ introduces bias of both quantities towards 0, especially for the first few iterations. We can counteract this initialization bias by using the corrected quantities

$$\hat{\boldsymbol{m}}^t = \frac{\boldsymbol{m}^t}{1-(\beta)^t}, \qquad \hat{\boldsymbol{v}}^t = \frac{\boldsymbol{v}^t}{1-(\gamma)^t}, \tag{15.9}$$

  where again $(\beta)^t$ and $(\gamma)^t$ are actual exponents. ADAM trains faster than SDG, but there are indications that the trained NN generalizes worse.

Although not optimizers themselves, the following techniques help to improve the model.

- *Dropout*: Remove random subset of neurons in each epoch approximates an ensemble of NNs and increases the robustness.

- *Batch normalization*: Normalize the output of layers to zero mean and unit variance. As this can reduce the expressivity of the model, each output of the normalization is modified by a learned affine transformation. Thus, the batch normalization can replicate identity and the model theoretically doesn't lose expressivity. The original paper [IS15] suggests implementing the normalization before the activation, but there are indications that the results are better the other way round.

- *Initialization*: For instance Layer-sequential unit variance (LSUV), whereat the weights are divided by the square root of the empirical variance.

Literature

- Ian Goodfellow, Yoshua Bengio, and Aaron Courville, *Deep learning*. 2016, [GBC16]

16 Backpropagation

The training of neural Networks involves the calculation of gradients. A very efficient way to do that is by the use of backpropagation, which is a special case of automatic differentiation. It is not equivalent to numerical differentiation

$$\frac{\partial f}{\partial x} \approx \frac{f(x+h) - f(x)}{h}, \tag{16.1}$$

and not equivalent to symbolic differentiation.

Backpropagation relies on computational graphs. They are similar to the diagrams we use to visualize the architecture of NNs, but they include basic mathematical operations such as multiplication and addition. Figure 16.1 shows an example of a computational graph of a very small NN.

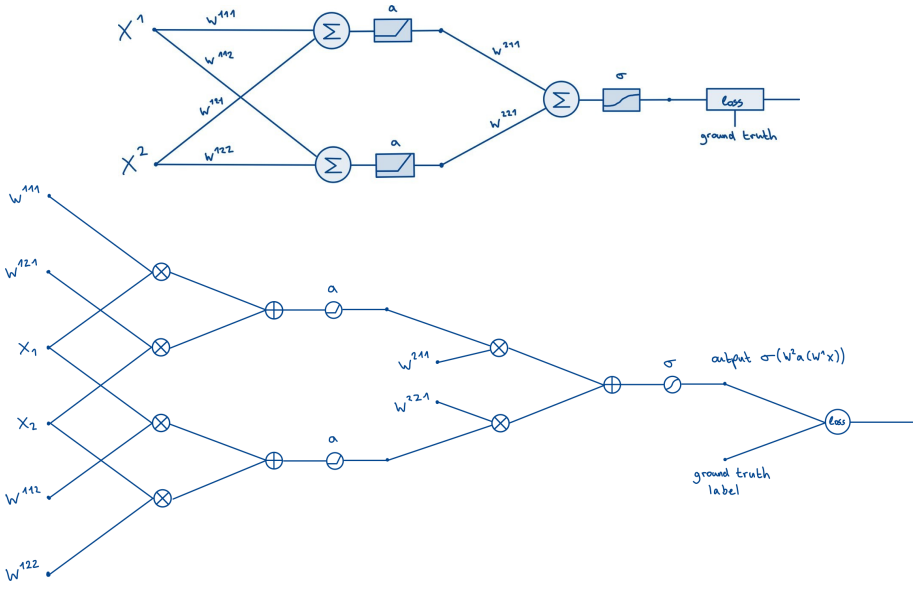


Figure 16.1: Top: Diagram of a small NN. Bottom: Corresponding computational graph.

As NNs/MLPs are compositions of functions, any derivative will heavily use the chain rule. We will look at the derivative of the loss function with respect to the weight w^{111}, see Figure 16.1.

$$\frac{\partial \mathcal{L}\left(y, \sigma(\boldsymbol{W}^2 a(\boldsymbol{W}^1 \boldsymbol{x}))\right)}{\partial w^{111}} = \overbrace{\underbrace{\frac{\partial \mathcal{L}}{\partial \sigma} \cdot \frac{\partial \sigma}{\partial a} \cdot \frac{\partial a}{\partial \boldsymbol{W}^1 \boldsymbol{x}} \cdot \frac{\partial \boldsymbol{W}^1 \boldsymbol{x}}{\partial w^{111}}}_{\text{reverse mode AD}}}^{\text{forward mode AD}}. \tag{16.2}$$

The forward mode automatic differentiation (AD) computes the derivative starting with the innermost derivative. At first, we select the variable according to which we want to derive; here w^{111}. We then calculate the derivative of all following quantities that depend on w^{111}. This corresponds to a sweep through the computational graph from left to right. We obtain the derivative of all outputs with respect to one input variable (in our example, there is only one output). Note that the partial derivatives in Equation 16.2 are Jacobian matrices.

The backward mode AD computes the derivative starting with the outer derivative. We need to pick an output variable that is derived with respect to all input variables. The details of the sweep from right to left are described below. If we consider few inputs and many outputs, the forward mode should be preferred, because here the number of sweeps needed is equal to the number of inputs. If we instead have many inputs and few outputs (e.g., loss), the backward mode should be preferred.

In contrast to numerical differentiation, the derivatives at each node of the computational graph are exact. The nodes consist of fundamental mathematical operations like addition, multiplication and functions such as exp and sin. Their derivatives can be implemented together with the operations themselves. This means that the partial derivatives or Jacobian matrices occurring in Equation 16.2 are known. In contrast to symbolic differentiation, we never have a mathematical expression for the entire derivative. Also, no derivation rules like the product rule are used. A symbolic differentiation is too expensive for more complicated functions.

16.1 Backward mode AD

We will now have a closer look at the backward mode AD. The technique assigns a so-called "adjoint" $\bar{v}_i$ to each intermediate result v_i at node i of the computational graph. It is given by the partial derivative of the output y with respect to the node,

$$\bar{v}_i = \frac{\partial y}{\partial v_i}. \tag{16.3}$$

Thus, the adjoint of the output is one. Using the chain rule, we obtain

$$\bar{v}_i = \sum_{j \in \text{children}(i)} \bar{v}_j \cdot \frac{\partial v_j}{\partial v_i} \tag{16.4}$$

for the other nodes. The values of v_i needed to evaluate the partial derivatives $\partial v_j / \partial v_i$ are also called "primals". They don't depend on the upstream adjoints $\bar{v}_j$ and thus can be stored during the forward pass. The forward pass is just the evaluation of the computational graph and should not be confused with forward mode AD. The sum over the children depends on the upstream adjoints and needs to be computed in the backward pass.

As an example, we will look at the backpropagation of the product

$$f(x) = a(x) \cdot b(x). \tag{16.5}$$

The computational graph is depicted in Figure 16.2. The backpropagation starts by assigning an

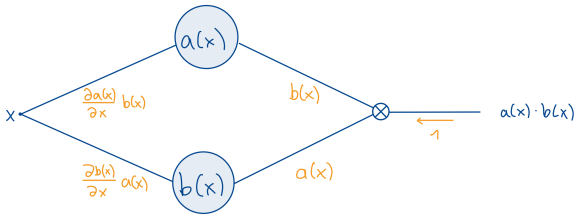


Figure 16.2: Computational graph of a product. The gradient along edges $\bar{v}_j \frac{\partial v_j}{\partial v_i}$ is colored in orange.

adjoint of 1 to the last node, which in our case is the result of a product. The corresponding partial derivatives of this multiplication are

$$\frac{\partial f}{\partial a} = b(x), \qquad \frac{\partial f}{\partial b} = a(x). \tag{16.6}$$

The nodes a and b only have one child whose upstream gradient is 1. Therefore, their adjoints are given by

$$\bar{a} = 1 \cdot \frac{\partial f}{\partial a} = b(x), \qquad \bar{b} = 1 \cdot \frac{\partial f}{\partial b} = a(x). \tag{16.7}$$

Using Equation 16.4, the derivative of f with respect to x is thus obtained by

$$\frac{\partial f}{\partial x} = \bar{a} \cdot \frac{\partial a}{\partial x} + \bar{b} \cdot \frac{\partial b}{\partial x} = b(x)a'(x) + a(x)b'(x). \tag{16.8}$$

This of course replicates the product rule. We summarize the procedure of backpropagation:

1) build computational graph

2) forward pass, store primals

3) topological sorting (order nodes such that they appear ahead of their dependencies)

4) starting from last node, propagate messages backwards.

16.2 Physics-informed ML

There are several approaches to "include physics" into machine learning. We list them sorted by the integration depth of the physical laws.

1) Observational bias: Feed the NN with plenty of examples. For example, the ground state energy of a molecule should be invariant of its rotation. To achieve this, molecules can the NN can be fed with rotated versions of the same molecule, see Figure 16.3.

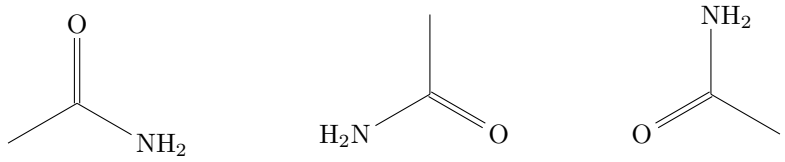


Figure 16.3: Different rotations of molecules should have the same ground state energy.

2) Learning bias: Add a soft constraint to loss. This means that the NN can still produce unphysical results, but they are penalized.

3) Inductive bias: "Bake" physical laws into NN (for example using translation invariant CNNs, see section 17).

Physics-informed NNs were introduced in [RPK19]. They can be used to solve partial differential equations (PDEs) numerically. The differential operators that occur in the process can be evaluated in automatic differentiation. We consider a PDE of the form

$$\frac{\partial u(\boldsymbol{x},t)}{\partial t} + N(u(\boldsymbol{x},t)) = 0, \quad \boldsymbol{x} \in \Omega,\ t \in [0,\tau], \tag{16.9}$$

where N is any (non-linear) differential operator. The problem also includes the known initialization $h(\boldsymbol{x})$,

$$u(\boldsymbol{x},0) = h(\boldsymbol{x}), \quad \boldsymbol{x} \in \Omega \tag{16.10}$$

and boundary conditions $g(\boldsymbol{x})$,

$$U(\boldsymbol{x},t) = g(\boldsymbol{x},t), \quad \boldsymbol{x} \in \partial\Omega,\ t \in [0,\tau]. \tag{16.11}$$

We want to model $u(\boldsymbol{x},t)$ by means of an NN $f(\boldsymbol{x},t;\boldsymbol{w})$. Therefore, we discretize the domain by introducing a lattice with points $(\boldsymbol{x}_i,t_i)$. To train our model, an appropriate loss function is needed. We could use the ansatz

$$\mathcal{L}(\boldsymbol{w}) = \underbrace{\sum_j \|f(\boldsymbol{x}_j,t_j;\boldsymbol{w}) - u(\boldsymbol{x}_j,t_j)\|^2}_{\text{data term}} + \underbrace{\sum_i \left\| \frac{\partial f(\boldsymbol{x}_i,t_i;\boldsymbol{w})}{\partial t} + N(f(\boldsymbol{x}_i,t_i;\boldsymbol{w})) \right\|^2}_{\text{PDE/physics term}}. \tag{16.12}$$

The loss function is visualized in Figure 16.4. The values of $u(\boldsymbol{x}_j,t_j)$ are either the initial/boundary conditions or obtained by another method, for instance by measurement or a more expensive solution/computation. The physics term contains derivatives that can be computed using automatic differentiation. In summary, physics-informed NNs can afford an efficient solution if the coefficient of the PDE are known.

Literature

- M. Raissi, P. Perdikaris, and G.E. Karniadakis, "Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations". 2019, [RPK19]

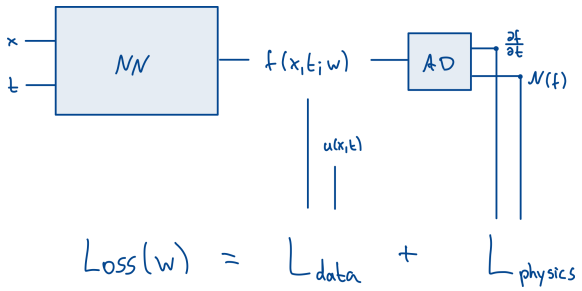


Figure 16.4: Visualization of the loss function.

17 Convolutional neural networks

The neural networks considered so far have a big disadvantage: The number of parameters is very large for high-dimensional inputs. A one megapixel image, for example, has 10^6 pixels/dimensions. If there is only one layer of the same size as the input (see Figure 17.1 on the left), the number of parameters is already very large: There are 10^6 biases and $10^6 \cdot 10^6 = 10^{12}$ weights. This type of network is also called fully connected NN, since every perceptron of a layer is connected to every node of the previous layer.

Figure 17.1: Left: Comparison of a fully connected network (left) and a locally connected network (right).

One way to reduce the number of parameters is by using a locally connected NN (see Figure 17.1 on the right). In this architecture, perceptrons are only connected to "neighboring" outputs of the previous layer. In our one-dimensional example, the perceptrons are only connected to the three closest inputs (except for the two outermost perceptrons). This means that the weight matrix is sparse. The number of parameters reduced to $10^6(3+1) = 4 \cdot 10^6$.

The number of parameters can be decreased even further. Weight Sharing NNs use the same weights and biases for every perceptron of a layer. That way, there are only $3 + 1 = 4$ parameters left. The mathematical operation of passing from one to the next layer is a convolution, which is why this architecture is also called *Convolutional Neural Network* (CNN). CNNs are shift/translation equivariant which makes them an example of inductive bias (see subsection 16.2). They are particularly useful for image analysis as translation invariance is exactly what we would expect when, e.g., looking at the problem of classifying specific entities within an image. The linear shift invariant operators known from signal processing also correspond to convolutions. Example 17.1 shows an application of CNNs.

Example 17.1: Super Resolution Microscopy

The resolution of microscopy is limited due to the diffraction of light. A single point of the observed object does not produce a point on the screen of a microscope, but instead results in an *Airy disk* if a circular aperture is used. If two points are too close together, the microscope can't resolve them, see Figure 17.2.

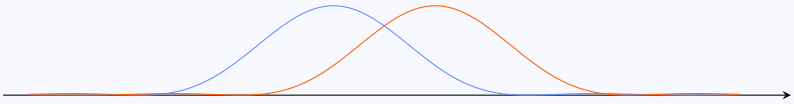


Figure 17.2: Cross-section of two overlapping Airy disks.

In [Spe+20], a method was introduced that allows to localize molecules with higher accuracy. The tool called DECODE (deep context dependent) processes temporal sequences of images by means of a convolutional neural network. It predicts detection probabilities, subpixel spatial coordinates, intensities and a background. It also provides uncertainties of the coordinates and the brightness (see Figure 17.3).

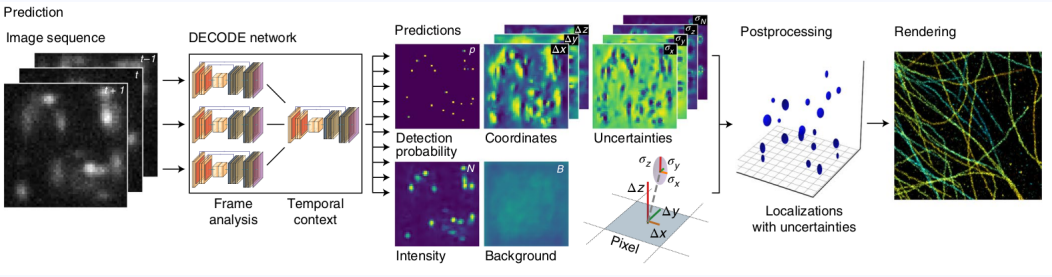


Figure 17.3: Architecture of DECODE. Figure from [Spe+20].

Since there is no data with the envisaged precision, the model had to be trained on simulated data. The simulator generates random emitters that are treated as ground truth (GT). The image sequence is rendered using an image formation model which takes care of camera effects and the mapping using a point spread function (PSF). The predictions of the DECODE network are evaluated by a loss that compares the count of emitters, their location and the background. The training pipeline is displayed in Figure 17.4.

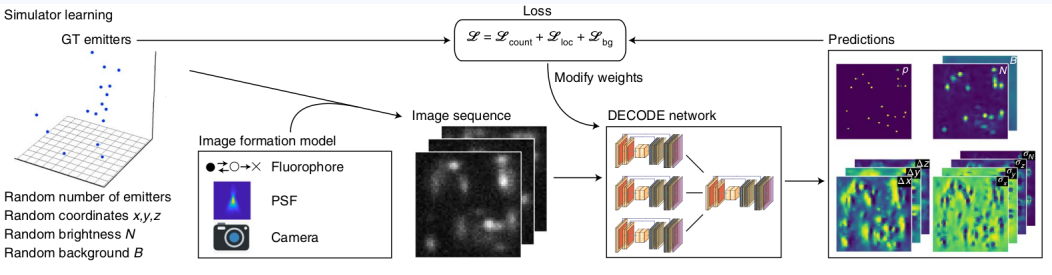


Figure 17.4: Training pipeline of DECODE. Figure from [Spe+20].

Figure 17.5 compares the DECODE network to other benchmarks. We see that the DECODE network shows great performance on every dataset modality in every regime (low/high density, low/high signal-to-noise ratio).

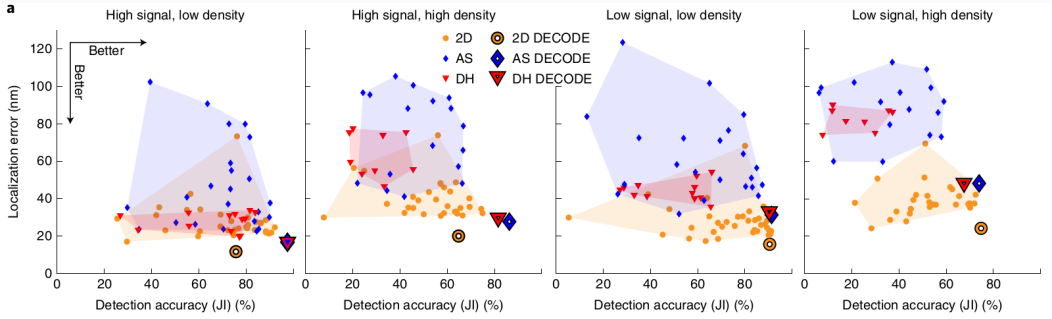


Figure 17.5: Comparison of DECODE to other benchmarks for different regimes (low/high density, low/high signal-to-noise ratio) and modalities (2D, AS: astigmatic, DH: double helix). Figure from [Spe+20].

We now want to take a look at the two-dimensional case since it is relevant for images. In this case, the weights and biases are shared across the neighbors of a pixel in the image $\boldsymbol{I}$. This means that the weights form a matrix of the size of that neighborhood, called *filter* or *kernel* $\boldsymbol{K}$. The process of propagating to the next layer is thus a convolution. In the mathematical sense, a convolution of two-dimensional discrete inputs $\boldsymbol{I}$ and $\boldsymbol{K}$ is defined by

$$[\boldsymbol{I} * \boldsymbol{K}]_{ij} = \sum_{k,l} [\boldsymbol{I}]_{i-k,j-l} \cdot [\boldsymbol{K}]_{kl} \,, \tag{17.1}$$

where the sum goes over all entries of the kernel. For a simpler illustration, however, we use the

definition

$$[\boldsymbol{I} * \boldsymbol{K}]_{ij} = \sum_{k,l} [\boldsymbol{I}]_{i+k,j+l} \cdot [\boldsymbol{K}]_{kl}\,, \tag{17.2}$$

which is also called cross-correlation. The only difference is the sign in the indices. The two definitions transition into each other when the kernel is mirrored along both axes or equivalently rotated by 180°. Figure 17.6 illustrates the convolution of matrices $\boldsymbol{I}$ and $\boldsymbol{K}$. The result of the convolution is obtained by laying the kernel over the image, multiplying entries that lie on top of each other and adding up the products. Afterward, the kernel slides along the image dimension to calculate the next value. In a variant of convolutions, the kernel doesn't move to the very next position, but skips a few steps. The length of this jump is called *stride*.

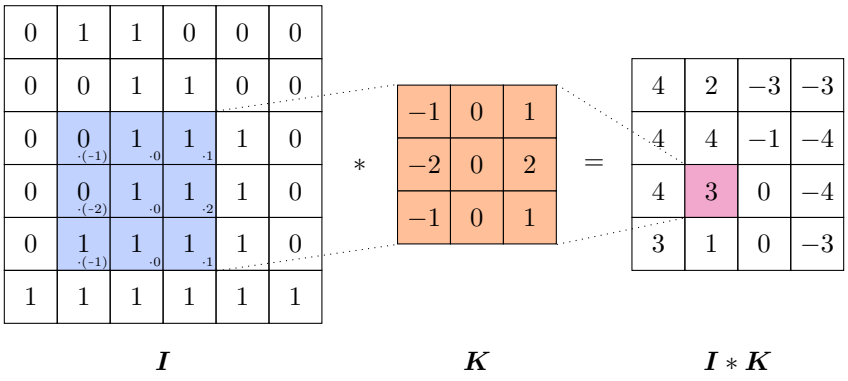


Figure 17.6: Example of a two-dimensional convolution.

The size of the kernel affects the size of the output $\boldsymbol{I} * \boldsymbol{K}$. For larger kernels we need less shift steps to cover the whole image. As the output has one pixel for each shifted kernel position the output gets smaller. If it is necessary to obtain a result with the same size as the output, the original image can be padded with zeros or extended by reflections along its borders. In Figure 17.7, we perform a convolution on a black-and-white image. The result has high values if the image changes from dark to bright in the horizontal direction. Thus, the kernel (also known as Sobel operator) detects edges in vertical direction.

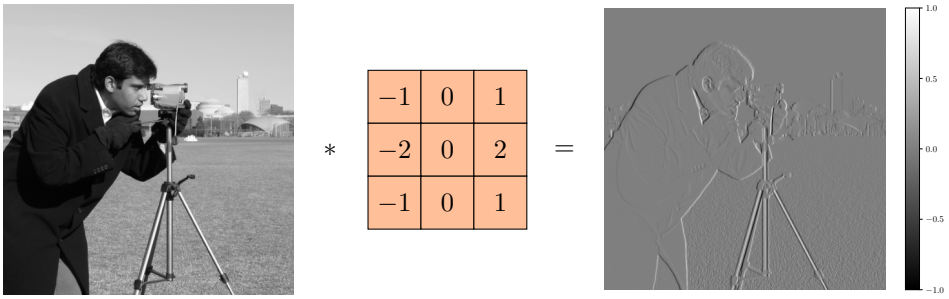


Figure 17.7: Example of a convolution with a kernel that detects vertical edges. The result is scaled so that the maximum absolute value is 1.

In CNNs, the parameters of the kernel are learned. The idea is that a convolutional layer learns to detect features such as edges. Subsequent layers can use these inputs to infer more and more complex structures.

The number of parameters in CNN layers is so low, that we can introduce multiple *channels* to each layer. Instead of one convolution, there are many convolutions with separate weights and biases. They can be used to detect different characteristics (vertical/horizontal edges, points, ...). If the inputs are colored images, they already have multiple channels. Let's say the images have height h, width b and c_{in} channels (often red, green and blue). One kernel with size $k \times k$ has $c_{\text{in}}k^2$

weights. The output of one convolution is a matrix of size $(w - k + 1) \times (h - k + 1)$. We apply multiple convolutions with different parameters to receive c_{out} output channels. Also considering the biases, we obtain

$$\#\text{parameters} = (c_{\text{in}} k^2 + 1) \cdot c_{\text{out}}. \tag{17.3}$$

17.1 VGG-16

The so-called VGG-16 model is an example of a neural network that relies on convolutional layers [SZ14]. It is named after the Visual Geometry Group from Oxford University and the fact that it uses 16 convolutional/fully connected layers. Its architecture is displayed in Figure 17.8. The

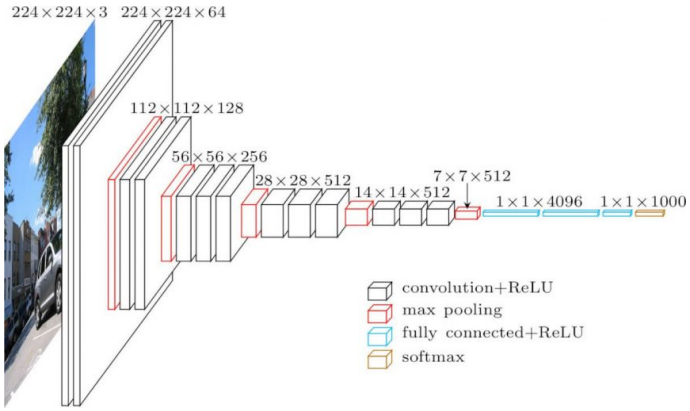


Figure 17.8: Architecture of VGG-16 model. Figure from [NDB18].

inputs are images with three color channels and a resolution of 244×244. The convolutional layers have 3×3-kernels and use padding to obtain the same image dimensions. After two convolutional layers, the model uses the technique of *max pooling*. The input matrix is divided into a grid of 2×2 squares, and only the maximum of every four values is passed to the next layer (see Figure 17.9). Both width and height are halved.

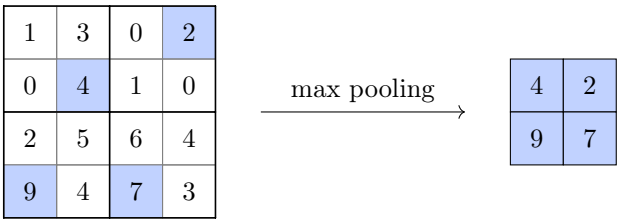


Figure 17.9: Example of a 2×2 max pooling.

The model successively reduces the image dimensions and increases the number of channels. Following the last max pooling, the data is flattened and transformed using three fully connected layers. All layers (convolution and fully connected) use ReLU as activation. The model is used for classification, so the softmax is applied in a final step. The dimensions have reduced to 1000 values that signify the image classes of the ImageNet Large-Scale Visual Recognition Challenge (ILSVRC)[7]. The VGG-16 was the second-best classification model of the challenge in 2014.

17.2 DeepDream

An entertaining application of CNNs is called DeepDream. Instead of using labeled training data to update the weights and improve the model's predictions, we change the input image itself to maximize the prediction of a certain class. The necessary technique is already provided with backpropagation.

[7] https://www.image-net.org/

This method requires a trained model in order to obtain useful results. We can choose an arbitrary image (or even just noise) and select one of the classes the model can discriminate. As before, we want to minimize the loss between the image and our desired class label. The difference is that we keep all weights and biases fixed and allow the image itself to be updated.

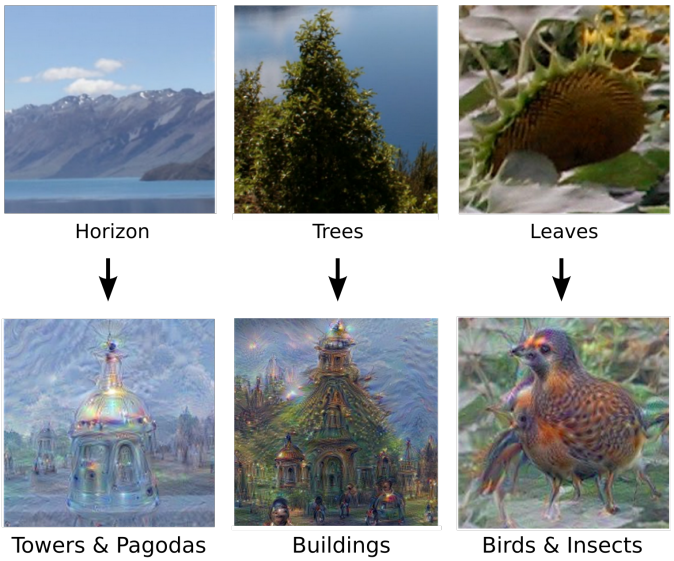


Figure 17.10: Examples of images and the results of DeepDream. Figure from [MOT15].

Figure 17.10 shows the results of DeepDream with the original image and the chosen image class. The notion of dreaming comes from the fact that the model creates images using images seen so far. This is an example of image generation enabled by machine learning. We will look at another approach in the section on diffusion models.

Literature

- Ian Goodfellow, Yoshua Bengio, and Aaron Courville, *Deep learning*. 2016, Chap. 9 [GBC16]

18 Applications of CNNs

18.1 Semantic segmentation

Until now, we considered classification problems that categorize an entire image into one of many classes. Semantic segmentation assigns a class to every single pixel of an image. The number of classes is much lower. An image of a segmentation problem together with its ground truth is given in Example 18.1.

Example 18.1: Cityscape

A popular data set for semantic segmentation is called Cityscape [Cor+16]. It contains 25000 annotated images of urban street scenes in 50 cities, see Figure 18.1. The purpose of the data set is to promote the development of deep learning models that pave the way to autonomous driving.

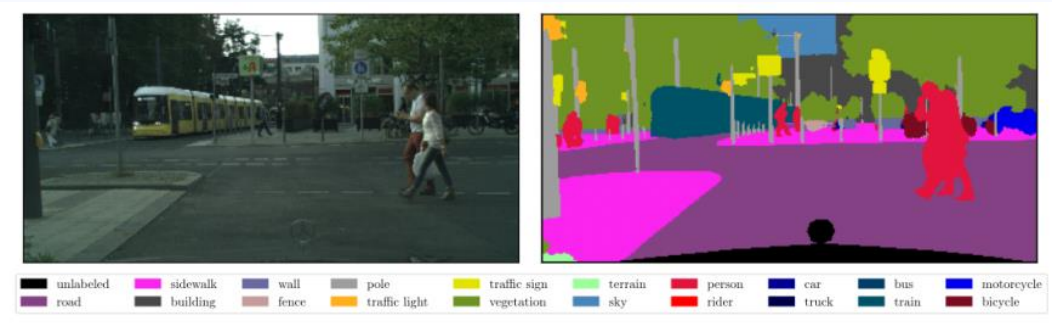


Figure 18.1: Example of the data set "Cityscape" used for semantic segmentation. Figure from [Kaw18].

The *U-Net* is a CNN architecture that is tailored to segmentation problems. It was introduced in [RFB15] and gained great popularity since. The visualization in Figure 18.2 makes the origin of the name apparent. The information from the input image can influence the segmentation map

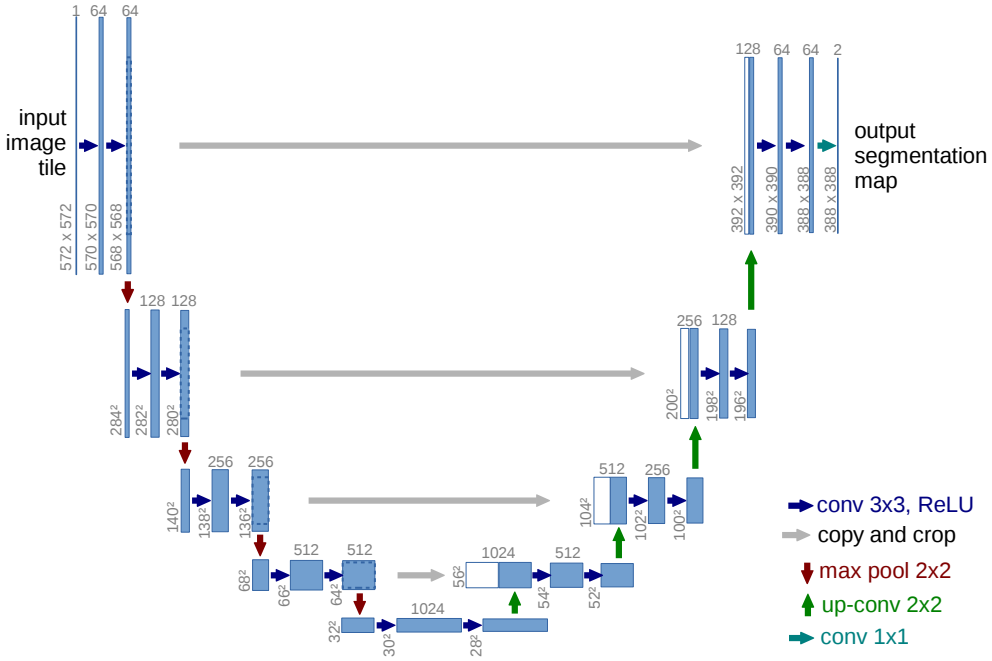


Figure 18.2: Diagram of the U-Net. Figure from [RFB15].

output through multiple pathways. The 2×2 max poolings reduce the spatial resolution. The

idea is that on the lower path, the information of what class is visible is detected on a coarse scale ("semantic pathway"). The special thing about the U-Net are the horizontal connections where the cropped input is copied. The cropping is necessary as edge pixels get lost in the convolutions of the semantic pathway. The upper paths convey where exactly a certain structure is located ("geometric pathway").

In the original U-Net, each of the both pathways are combined in a concatenation. For this, the output of the max-pooling and convolutions must be scaled up again to meet the same resolution. This is achieved by a technique called *up-convolution* or *transposed convolution*. It consists out

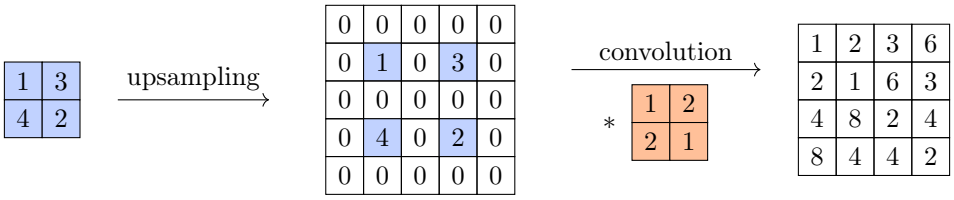


Figure 18.3: Example of a 2×2 up-convolution.

of an upsampling step followed by a convolution with learned parameters, see Figure 18.3. The up-convolutions halve the number of channels so that the total number of values is the same as for the copied part. The last step is a 1×1 convolution that maps each 64-dimensional feature vector to the required number of classes. Note that the image dimension of the output is smaller than the input image. The model only predicts classes for the center part of an image. Images that are larger than the 572×572 pixels can be tiled into several regions and fed into the U-Net separately.

Since the proposal of the U-Net in 2015, many modifications were made to improve the performance. To train a U-Net, following hints may be helpful (courtesy of Alberto Bailoni in 2020):

- Definitely include some kind of normalization (GroupNorm or BatchNorm): for example Conv+GroupNorm+ReLU (but avoid common mistake of applying normalization before the last sigmoid/softmax).
- Use skip connections in each U-Net block (explained below).
- Sum features coming from the U-Net skip-connections, instead of concatenating them: it saves some GPU-memory and usually gives equivalent performances.
- Use larger kernels (e.g., 5×5 or 7×7) at the very beginning (applied to the input image) and at the very end (just before applying the final sigmoid).
- Use "same" convolutions (padded with zero) to simplify architecture description. Boundary artifacts are usually not a problem during training.
- Adam optimizer, learning rate 10^{-4} with decay are still the default options.

Example 18.2: Pose estimation

Semantic segmentation can be used to estimate the pose of people in images. [Cao+16] proposed the method shown in Figure 18.4. It uses the input image to predict Part Confidence Maps (e.g., shoulder and elbow) and Part Affinity Fields (e.g., pointing in the direction of the upper arm). Both outputs are matched to estimate body poses. The method also works in real-time (the video result is available at `https://youtu.be/pW6nZXeWlGM`).

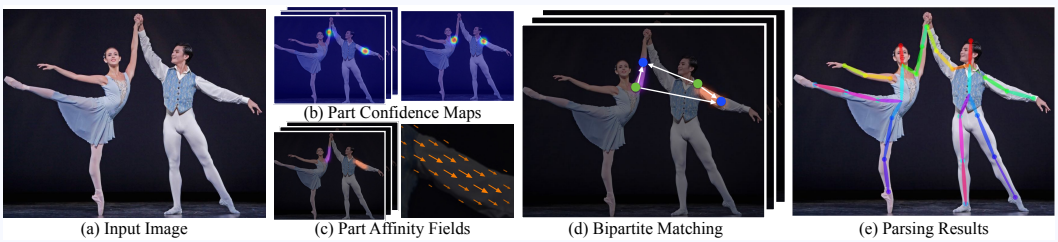
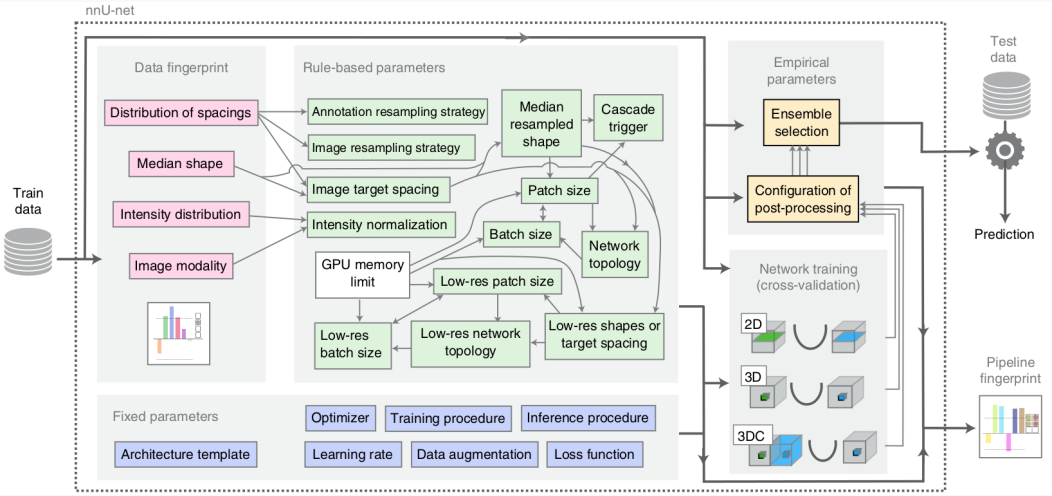


Figure 18.4: Pose estimation pipeline. Figure from [Cao+16].

Example 18.3: No New U-net

The No New U-net (nnU-Net) proposed in [Ise+21] is a method that automatically sets up semantic segmentation pipelines for biological applications. It designs the preprocessing, network architecture, training and post-processing based on the data set and hardware properties. The pipeline of the nnU-Net is shown in Figure 18.5, its performance on various data sets in Figure 18.6.



Design choice	Required input	Automated (fixed, rule-based or empirical) configuration derived by distilling expert knowledge (more details in online methods)
Learning rate	–	Poly learning rate schedule (initial, 0.01)
Loss function	–	Dice and cross-entropy
Architecture template	–	Encoder–decoder with skip-connection ('U-Net-like') and instance normalization, leaky ReLU, deep supervision (topology-adapted in inferred parameters)
Optimizer	–	SGD with Nesterov momentum ($\mu = 0.99$)
Data augmentation	–	Rotations, scaling, Gaussian noise, Gaussian blur, brightness, contrast, simulation of low resolution, gamma correction and mirroring
Training procedure	–	1,000 epochs × 250 minibatches, foreground oversampling
Inference procedure	–	Sliding window with half-patch size overlap, Gaussian patch center weighting
Intensity normalization	Modality, intensity distribution	If CT, global dataset percentile clipping & z score with global foreground mean and s.d. Otherwise, z score with per image mean and s.d.
Image resampling strategy	Distribution of spacings	If anisotropic, in-plane with third-order spline, out-of-plane with nearest neighbor Otherwise, third-order spline
Annotation resampling strategy	Distribution of spacings	Convert to one-hot encoding → If anisotropic, in-plane with linear interpolation, out-of-plane with nearest neighbor Otherwise, linear interpolation

Image target spacing	Distribution of spacings	If anisotropic, lowest resolution axis tenth percentile, other axes median. Otherwise, median spacing for each axis. (computed based on spacings found in training cases)
Network topology, patch size, batch size	Median resampled shape, target spacing, GPU memory limit	Initialize the patch size to median image shape and iteratively reduce it while adapting the network topology accordingly until the network can be trained with a batch size of at least 2 given GPU memory constraints. for details see online methods.
Trigger of 3D U-Net cascade	Median resampled image size, patch size	Yes, if patch size of the 3D full resolution U-Net covers less than 12.5% of the median resampled image shape
Configuration of low-resolution 3D U-Net	Low-res target spacing or image shapes, GPU memory limit	Iteratively increase target spacing while reconfiguring patch size, network topology and batch size (as described above) until the configured patch size covers 25% of the median image shape. For details, see online methods.
Configuration of post-processing	Full set of training data and annotations	Treating all foreground classes as one; does all-but-largest-component-suppression increase cross-validation performance? Yes, apply; reiterate for individual classes No, do not apply; reiterate for individual foreground classes
Ensemble selection	Full set of training data and annotations	From 2D U-Net, 3D U-Net or 3D cascade, choose the best model (or combination of two) according to cross-validation performance

Figure 18.5: Pipeline of nnU-Net. Figure from [Ise+21].

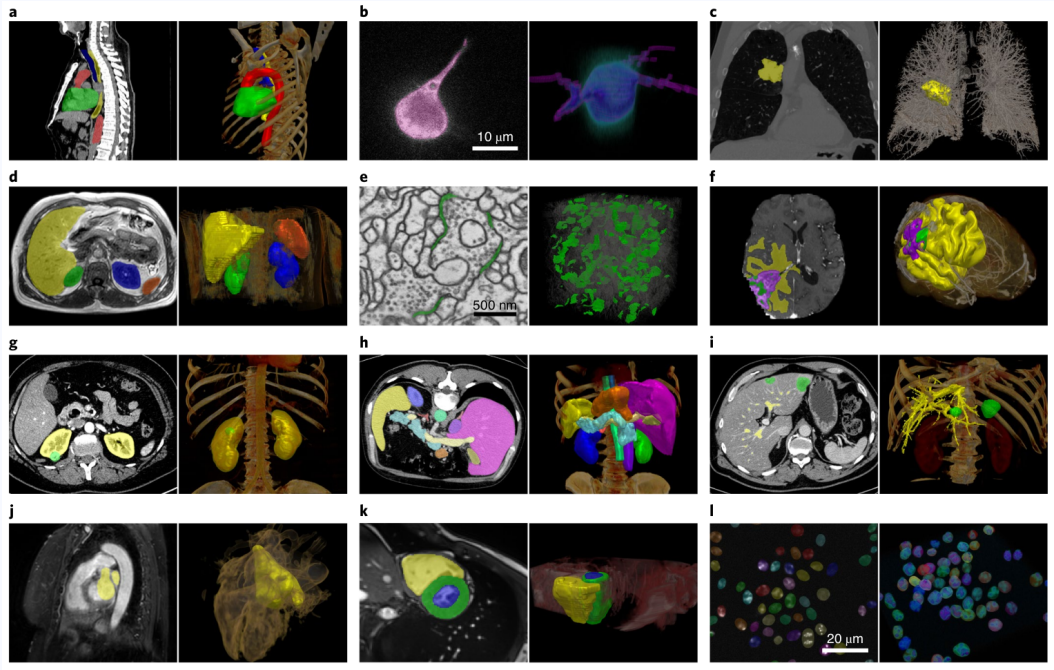


Figure 18.6: Application of the nnU-Net to various segmentation tasks rendered in 2D and 3D. Figure from [Ise+21].

18.2 Skip connections

When training very deep neural networks, the gradient for updating parameters in the first few layers may be very small. This can happen because the gradient is multiplied with the derivative of the activation function in backpropagation. Many activation functions such as ReLU or sigmoid have large regions where they are almost flat, so that there is a high probability of multiplying the gradient with zero. To counteract vanishing gradients, *skip connections* were introduced. They help to "push" the gradient across deep networks. We have already seen their use in the U-Net. Other popular realizations are the ResNet (residual neuronal network) and the DenseNet.

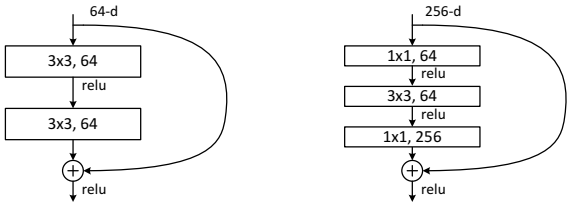


Figure 18.7: Building block of ResNet. Figure from [He+15].

Skip connections bypass at least one non-linear transformation and forward it untransformed. In backpropagation, the gradient is passed directly through these identity functions. A graphical representation of skip connections in ResNet is displayed in Figure 18.7. Their implementation into a deep network and comparisons to architectures without skip connections is shown in Figure 18.9. The data of the skip connections is either added or concatenated to the transformed features. The architecture of the DenseNet proposed in [Hua+16] contains so-called Dense Blocks (see Figure 18.8). In these blocks, all possible skip connections are implemented. Since the inputs of the skip connections are concatenated in the DenseNet, a layer receives the features of all preceding layers within a Dense Block. Since the size of the features remains the same inside the Dense Blocks, so-called transition layers are necessary. They apply convolutions and max poolings to reduce the dimension so that the model can make predictions for classifications.

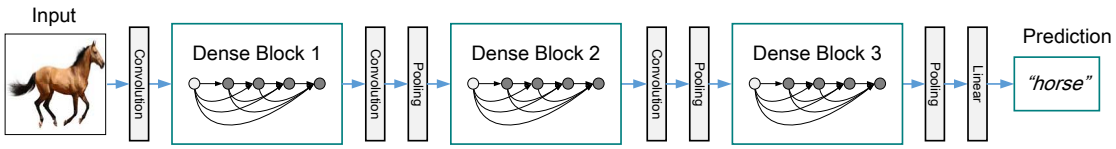


Figure 18.8: DenseNet architecture. Figure from [Hua+16].

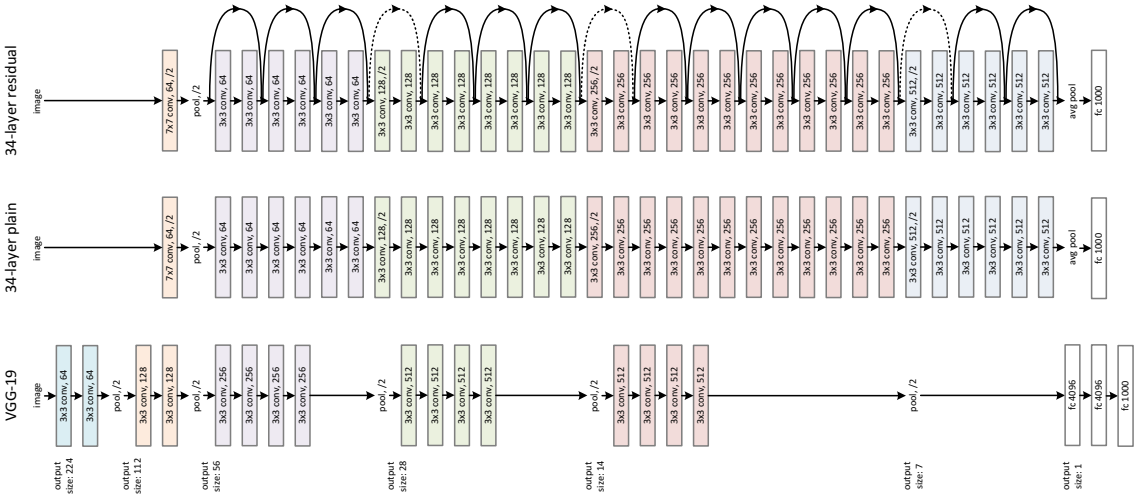


Figure 18.9: ResNet architecture compared to other models. Figure from [He+15].

18.3 Loss surface

Skip connections can also greatly affect the loss surface. Figure 18.10 shows the loss of ResNet-56 with and without skip connections. The network with skip connections has a loss surface that

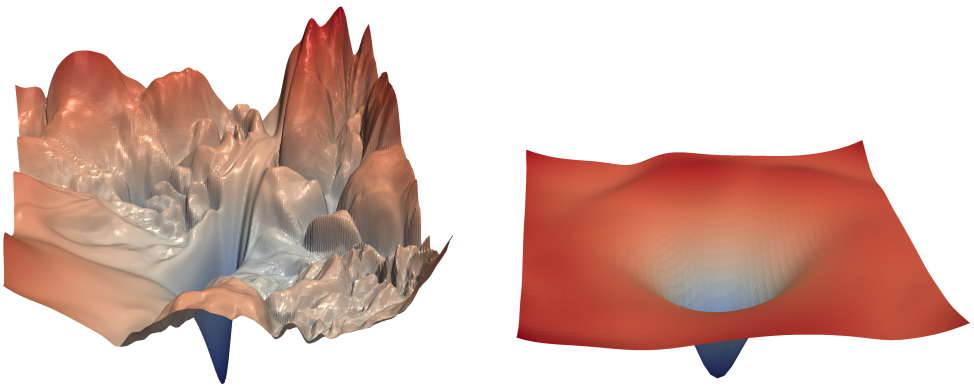


Figure 18.10: Loss surface of ResNet-56. Left: Without skip connections. Right: With skip connections. Figure from [Li+17].

looks a lot smoother than the one for the network without them. Finding the minimum in the right image is much easier than in the left one. The loss is displayed as a function

$$f(x, y) = \mathcal{L}(\boldsymbol{w}^* + x\boldsymbol{\delta} + y\boldsymbol{\eta}) \tag{18.1}$$

around a center point $\boldsymbol{w}^*$. The two directions $\boldsymbol{\delta}$ and $\boldsymbol{\eta}$ in weight space are *sampled randomly* from a Gaussian distribution.

These plots can't directly be used to compare different losses or network architectures, as networks can be invariant to scaling. If ReLU activation is used, the output stays constant after all weights of one layer are multiplied by a positive factor and all weights of the next layer are divided by that factor. If batch normalization is used, even the output of a single layer is invariant under scaling.

The scaling of plots such as Figure 18.10 is very important, because it decides whether something looks smooth. Perturbations of weights by one unit will only have minor effects if the weights are on a large scale while it might completely change the loss if the weights are on a small scale. But due to the scale invariance, both models with small or large weights could be equivalent. Therefore, [Li+17] proposes to rescale the individual components of the random directions $\boldsymbol{\delta}, \boldsymbol{\eta}$ so that all components pertaining to a single perceptron have the same Frobenius norm as the weights pertaining to that perceptron.

Minima are manifolds?

While Figure 18.10 suggests that minima of the loss are points in weight space, it has been found empirically that these minima can actually be manifolds.

In [Dra+18], paths connecting minima of a network are constructed so that the loss along these paths is minimal. Figure 18.11 shows an example of such a path for a DenseNet. The loss along this path is much smaller than the loss along a linear interpolation.

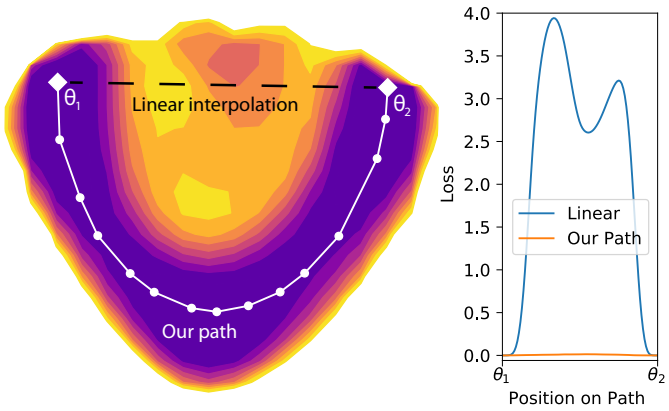


Figure 18.11: Slice though the loss function. The curved path connecting the minima θ_1 and θ_2 runs through a manifold of low loss. Figure from [Dra+18].

A tentative explanation of why deep NNs don't exhibit boundaries between local minima is given by the following illustration. We consider the XOR (exclusive OR) problem visualized in Figure 18.12. To perfectly classify the four points, two perceptrons, say Alice and Bob, are needed. Their output is combined by an AND-perceptron in a second layer as seen before. The MLP

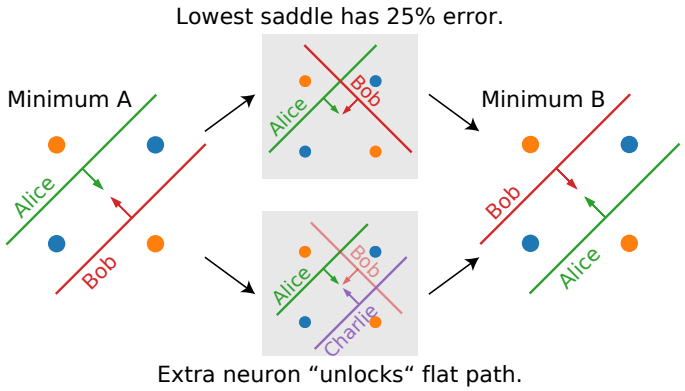


Figure 18.12: On a XOR data set, a continuous transition without misclassification between the two outer minima is only possible with three perceptrons. Figure from [Dra+18].

still classifies the data perfectly if Alice and Bob are interchanged; this is shown on the right of Figure 18.12. These two networks both correspond to minima of the loss surface. But they cannot continuously be transformed into each other without increasing the loss along the way (upper center in Figure 18.12).

Only if a third perceptron is introduced, both minima can be transformed into each other via a path along which the loss is minimal. Such a transformation corresponds to a curve as the one displayed in Figure 18.11. In the lower center of Figure 18.12, Charlie is introduced to the MLP. He can take over the function of Bob so that Bob can rotate to Alice. After Bob has replaced Alice, she can move over to Bob's original place and replace Charlie to arrive at the other minimum. The switching on and off of Alice, Bob and Charlie is done by the perceptron in the second layer. This would explain why wider networks result in minima that are manifolds. A similar argument can be made for the increase of the depth.

A different approach to explore the minimum manifold is to use a cyclic learning rate schedule as proposed in [Gar+18]. The learning rate is periodically decreased and increased as shown in Figure 18.13. This periodicity is reflected in the test error as the network jigs around too much at large learning rates and ideally converges to a different local minimum at small learning rates. The period of increasing learning rate is also called "exploration phase" while the period of decreasing learning rate is known as "exploitation phase". The lower diagram shows the distance from the initial parameters in weight space and verifies that the model indeed converges to different minima.

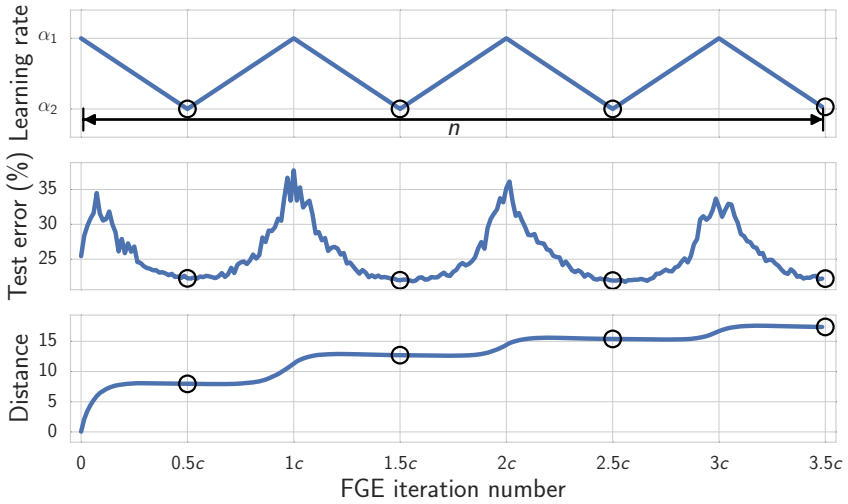


Figure 18.13: Using a cyclic learning rate to explore the minimum manifold. Figure from [Gar+18].

If we have obtained several points of the manifold of minimum loss, we can use them as a starting point for an *ensemble method* as already discussed for random forests (tree ensembles). The Fast Geometric Ensembling (FGE) that [Gar+18] derives from the ensemble obtained by the cyclic learning rate shows better performance than the separate models or the combination of the two models at the endpoints of the path through the manifold. This indicates that the ensembles of neuronal networks in the manifold of low loss might generalize better.

18.4 Side tasks and self-supervision

Supervised deep neural networks require huge amounts of labeled training data, but large sets of labeled data are often not available and very expensive to acquire. Unsupervised methods, however, are often unable to capture the visual semantics required for problems such as object recognition and therefore fall behind supervised methods [DZ17]. *Self-supervised learning* tries to utilize the vast abundance of unlabeled training data such as images and videos. The challenge is to formulate tasks so that we can easily measure performance, even without knowing a ground truth. Ideally, the tasks are set in such a way that they can only be solved if the model understands some structure of the data. Possible examples of such tasks are listed below.

- denoising / inpainting
- compression (auto-encoding)
- figuring out rotations of patches, arrangement of patches
- contrastive learning tasks

- predicting next frame in a video
- estimating optical flow in a video

Models pre-trained with a self-supervised task can be re-trained to solve other problems with less labeled training data. Self-supervised models can also be advantageous themselves, such as in Example 18.4.

Example 18.4: Noise2Void

Denoising of images is a problem that can be well addressed by CNNs. Usually, the CNNs can be trained on noisy images with their noise-free ground truths. A method called Noise2Noise was introduced in [Leh+18] to denoise images without ground truth images. Instead, independent pairs of noisy images are used for training. If even such pairs cannot be obtained, a method called Noise2Void proposed in [KBJ19] can be used as it is a self-supervised model.

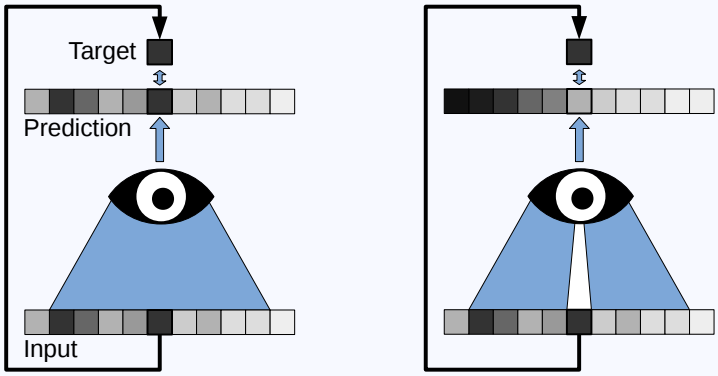


Figure 18.14: Degenerating conventional versus blind spot network. Figure from [KBJ19].

Noise2Void relies on the fact that pixel values in images are normally not independent but are influenced by a certain region of pixels. Instead of predicting the entire denoised image from the noisy image, the value of single pixels can be modelled as a function of this region known as the *receptive field*. If the ground truth image is known, a network can be trained with the denoised pixel value as the target.
If no ground truth is known, the noisy pixel could naively be used as target. This situation, shown on the left in Figure 18.14, would lead to a degenerated network, as the model would simply learn the identity. Noise2Void instead implements a "blind spot network" shown on the right in Figure 18.14. The current pixel is not visible to the model such that it can be used as target.

Literature

- Kaiming He et al., *Deep Residual Learning for Image Recognition.* 2015, [He+15]

19 Diffusion models

Diffusion models are a type of generative machine learning model. Popular examples include DALL-E2 and Stable Diffusion, which are both used to generate images from text prompts.

Intuition

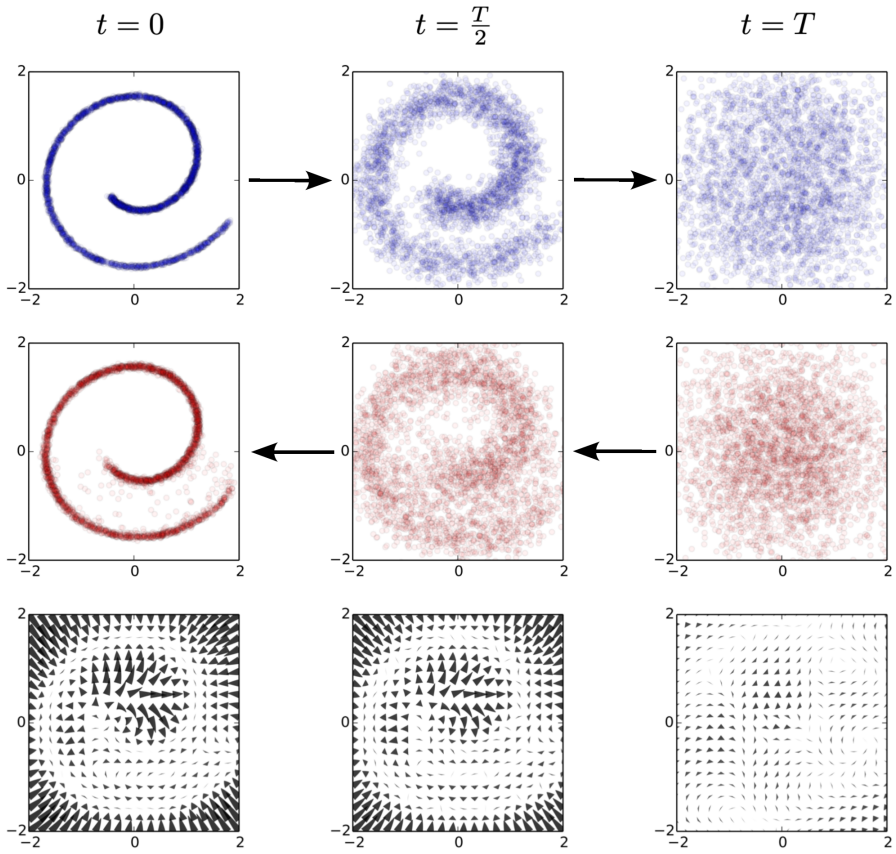


Figure 19.1: Illustration for a diffusion process applied to a two dimensional toy dataset. The top row shows the samples of the dataset for different iterations of the forward process. The middle row depicts the sample distributions for the same time steps in the backward process. The bottom row shows the corresponding score functions. Figure adapted from [Wen21].

Suppose that we have a set of (gray) images, and we want to generate new images with similar content as the original ones. We can represent those images in a (typically high dimensional) vector space, where each dimension corresponds to the gray value of one pixel. Large parts of this space will correspond to meaningless conglomerations of pixels, and the plausible images will only lie on a low dimensional manifold. This situation is illustrated in the top left panel of Figure 19.1 for a two-dimensional example, where the points on the blue spiral represent plausible samples.

In order to generate new data, we have to somehow get a hold on the distribution of original images $p^*(\boldsymbol{x})$. $\boldsymbol{x}$ has dimensions equal to the number of pixels and each entry encodes the gray value of the corresponding pixel. It is beneficial to parameterize the distribution as follows

$$p(\boldsymbol{x};\boldsymbol{w}) = \frac{e^{-\beta f(\boldsymbol{x};\boldsymbol{w})}}{\int e^{-\beta f(\boldsymbol{x};\boldsymbol{w})} \mathrm{d}\boldsymbol{x}} = \frac{e^{-\beta f(\boldsymbol{x};\boldsymbol{w})}}{Z(\boldsymbol{w})}. \tag{19.1}$$

In this way we ensure that the learned probability density is always positive. This expression is called the Gibbs measure which should be familiar from statistical mechanics. In this form $f(\boldsymbol{x};\boldsymbol{w})$ takes the role of the energy, β is the inverse temperature and $Z(\boldsymbol{w})$ is the partition function. A model of the type of Equation 19.1 is also called *energy based model*.

Learning $p^*(\boldsymbol{x})$ thus involves learning both the energy and the partition function. This is however problematic because the calculation of $Z(\boldsymbol{w})$ involves integrating/summing over all possible pixel values which for normal sized images is computationally intractable.

To solve this issue, we introduce the so-called *score function*

$$s(\boldsymbol{x};\boldsymbol{w}) = \nabla_x \log p(\boldsymbol{x};\boldsymbol{w}) = -\nabla_x f(\boldsymbol{x};\boldsymbol{w}) - \underbrace{\nabla_x \log Z(\boldsymbol{w})}_{=0} = -\nabla_x f(\boldsymbol{x};\boldsymbol{w}), \tag{19.2}$$

and notice that it no longer depends on the partition function, which makes it a tractable quantity. The score function of our toy example is shown in the lower left panel of Figure 19.1. Since the score is calculated as the negative gradient of the energy $f(\boldsymbol{x};\boldsymbol{w})$, we can think of it as a force field which pushes arbitrary images towards regions of more plausible images. If we knew the true score function, we could in theory sample data from a known prior distribution, e.g., a Gaussian and iteratively move those samples along the direction specified by the score function until we end up with the desired images. However, there are additional complications. Since our training data is usually very sparse within the high dimensional vector space, the score function can only be reconstructed locally in regions where the density of training data is large enough as exemplarily shown in Figure 19.2. An accurate global reconstruction is however crucial if we want to start from an isotropic distribution like the Gaussian.

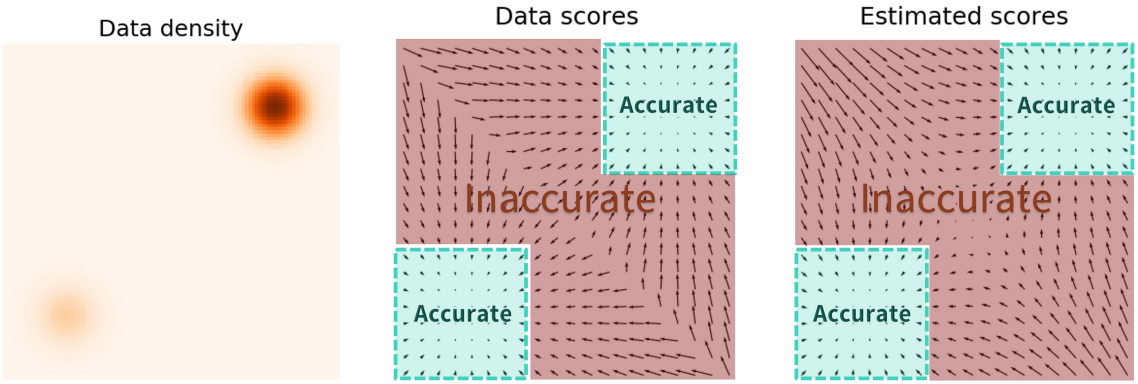


Figure 19.2: Example where the reconstruction of the score of the true probability density fails due to regions of high sparsity. Figure from [Son21].

This finally leads to the idea of diffusion: Instead of calculating just one score function, we iteratively add noise to our training data and learn a score function for each of the intermediate distributions. The added noise is chosen such that we eventually end up with Gaussian noise, i.e., we have lost all information of the original distribution. If the noise in each step is small enough, we are able to reverse the process, starting from Gaussian noise and iteratively updating the samples according to the corresponding intermediate score function until we end up with the original distribution. In this way, we ensure that the score function we use in each step is meaningful, in contrast to our naive approach before. The whole process is illustrated in Figure 19.1.

19.1 Stochastic processes

Let us for the moment assume that we already know the score function of the true probability density $\nabla_{\boldsymbol{x}} f^*(\boldsymbol{x})$. We can then describe a process which evolves some initial probability distribution p_0 into the Gibbs distribution (Equation 19.1) with the following equation:

$$\frac{\mathrm{d}p(\boldsymbol{x},t)}{\mathrm{d}t} = \mathrm{div}(\nabla_{\boldsymbol{x}} f^*(\boldsymbol{x}) \cdot p(\boldsymbol{x},t)) + \frac{1}{\beta}\Delta_{\boldsymbol{x}} p(\boldsymbol{x},t). \tag{19.3}$$

It is called *Fokker-Planck equation* and describes the change of the overall probability distribution with time. The first term in the equation will increase the probability density in places where $f^*(\boldsymbol{x})$ is low which corresponds to the fact that the system strives for the state of minimum energy. The second term on the other hand is a diffusion term which accounts for the finite temperature $\frac{1}{\beta}$ of the system. As already mentioned, in the limit $t \to \infty$, $p(\boldsymbol{x},t)$ will converge to the Gibbs

distribution which is the density which minimizes the free energy given by

$$\int p(\boldsymbol{x}) f^*(\boldsymbol{x}) \mathrm{d}\boldsymbol{x} + \frac{1}{\beta} \int p(\boldsymbol{x}) \log p(\boldsymbol{x}) \mathrm{d}\boldsymbol{x}. \tag{19.4}$$

A one-dimensional example for such a process is illustrated in Figure 19.3.

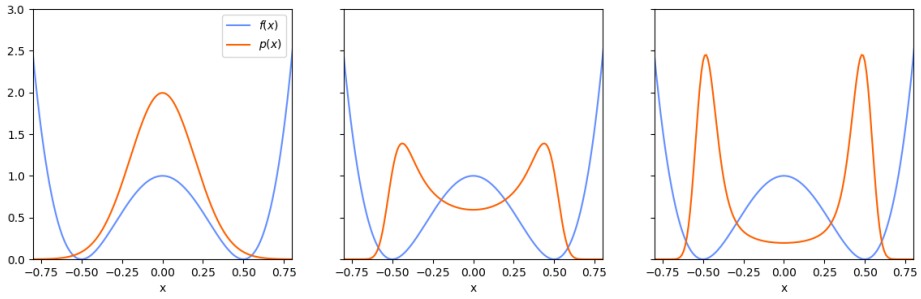


Figure 19.3: Temporal change (from left to right) of an initial Gaussian probability density $p(\boldsymbol{x}, t)$, in the presence of an energy distribution $f^*(\boldsymbol{x})$, according to the Fokker-Planck equation (Equation 19.3). For later times the probability density will be largest around the minima of the energy.

We can equivalently describe the process using the following equation:

$$\mathrm{d}\boldsymbol{x} = -\nabla_{\boldsymbol{x}} f^*(\boldsymbol{x}) \mathrm{d}t + \sqrt{\frac{2}{\beta}} \mathrm{d}\boldsymbol{w}, \tag{19.5}$$

which is a so-called "stochastic differential equation (SDE)". It is distinct from other types of differential equations due to the appearance of the term $\mathrm{d}\boldsymbol{w}$. It can be thought of as infinitesimal white noise and thus introduces non-determinism into the equation. Instead of describing the evolution of the whole probability distribution the SDE specifies how individual samples move according to the process. It is thus more useful for our problem of generating individual images.

The former discussion applies to actual physical systems and utilizes only one score function. As discussed at the beginning of this chapter we however need to use an adapted score function at each of the intermediate timesteps when we want to apply the stochastic process in diffusion models. Instead of only looking at the "backward process" which yields the reconstruction of the probability distribution as we have done above, we also have to consider the "forward process" which we use to learn the intermediate score functions. We require that this process transforms an arbitrary probability distribution into a Gaussian normal distribution. This can for example be achieved by the following SDE:

$$\mathrm{d}\boldsymbol{x} = -\frac{1}{2} \alpha(t) \boldsymbol{x} \mathrm{d}t + \sqrt{\alpha(t)} \mathrm{d}\boldsymbol{w}, \tag{19.6}$$

where the first them is a drift term which pulls the samples towards $\boldsymbol{x} = 0$ and the second term is a diffusion term. The time dependent coefficient $\alpha(t)$ controls how much noise is injected at each time step. Examples for the forward diffusion process are given in Figure 19.4 and Figure 19.5.

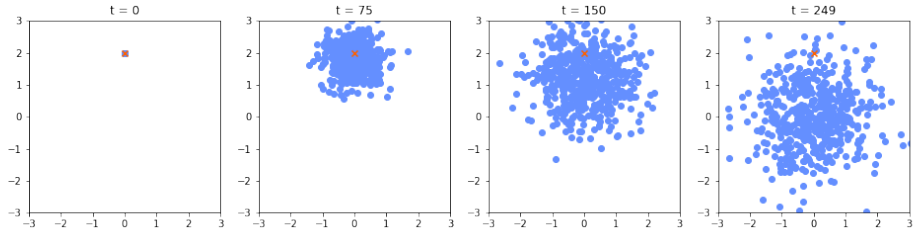


Figure 19.4: Forward diffusion applied to a set of points which all start at the coordinates marked with the red 'x'. The distribution of samples converges to a standard normal.

It is important to note that the above is only one example of a possible implementation of the forward diffusion process. There exist also other and more general approaches.

Formulating the process in terms of an SDE also allows us to explicitly give an expression for the corresponding reverse SDE which we need to describe the backward process:

$$\mathrm{d}\boldsymbol{x} = \left[-\frac{1}{2}\alpha(t)\boldsymbol{x} - \alpha(t)\nabla_{\boldsymbol{x}} \log p(\boldsymbol{x}, t; \boldsymbol{w})\right] \mathrm{d}t + \sqrt{\alpha(t)}\mathrm{d}\boldsymbol{w}, \tag{19.7}$$

where $\nabla_{\boldsymbol{x}} \log p(\boldsymbol{x}, t; \boldsymbol{w})$ is the estimated score function for the distribution at time t. This is the reverse in the sense that applying the forward stochastic process described in Equation 19.6 to samples from a distribution p_0 in some time interval $[t_0, t_1]$ and subsequently applying the reverse stochastic process in Equation 19.7 backwards in time i.e. from t_1 to t_0 the resulting samples will again be distributed according to p_0.

We note that the above expression only requires knowledge of the score function for all time steps, which we can learn according to the scheme described below. In case we already know all the score functions, we can sample new images by solving the reverse SDE for Gaussian noise samples as initial conditions, using an Euler-Maruyama method.

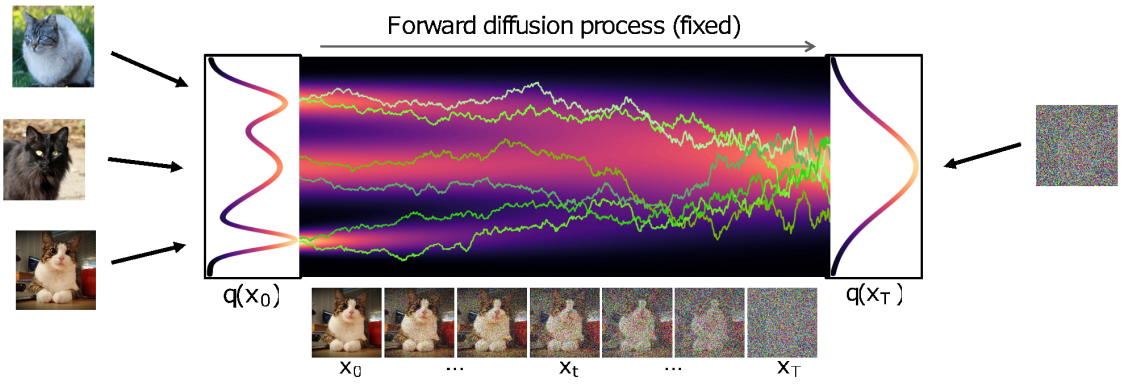


Figure 19.5: Schematic representation of a diffusion process applied to image samples. Figure from [Son22].

19.2 Score matching

The only remaining problem is how exactly we can learn the score functions from the data. To this end we first look at how we can estimate one individual score function for fixed time t. To begin we have to come up with an appropriate loss function. An intuitive measure for the dissimilarity between predicted and true score is given by the so-called "Fisher divergence":

$$\mathcal{L}(\boldsymbol{w}) = \frac{1}{2}\int p^*(\boldsymbol{x}) \| \underbrace{\nabla_x \log p(\boldsymbol{x}; \boldsymbol{w})}_{\text{predicted score}} - \underbrace{\nabla_x \log p^*(\boldsymbol{x})}_{\text{true score}} \|^2 \mathrm{d}\boldsymbol{x}. \tag{19.8}$$

This expression still depends on the true distribution $p^*(\boldsymbol{x})$, which is of course undesirable. Fortunately there is a theorem by [HD05] which allows us (under mild conditions) to rewrite the above expression to

$$\mathcal{L}(\boldsymbol{w}) = \int p^*(\boldsymbol{x}) \sum_{j=1}^{\mathrm{p}} \left[\frac{1}{2}\left(\frac{\partial \log p(\boldsymbol{x}; \boldsymbol{w})}{\partial x_j}\right)^2 + \frac{\partial^2 \log p(\boldsymbol{x}; \boldsymbol{w})}{\partial x_j^2}\right] \mathrm{d}\boldsymbol{x} + \text{const.} \tag{19.9}$$

Additionally, we can approximate the integral with a sum over the training data, so that we finally arrive at

$$\mathcal{L}(\boldsymbol{w}) = \sum_{i=1}^{\mathrm{n}} \sum_{j=1}^{\mathrm{p}} \left[\frac{1}{2}\left(\frac{\partial \log p(\boldsymbol{x}_i; \boldsymbol{w})}{\partial x_j}\right)^2 + \frac{\partial^2 \log p(\boldsymbol{x}_i; \boldsymbol{w})}{\partial x_j^2}\right] + \text{const.} \tag{19.10}$$

Since the training data contains implicit information about the true data distribution, we could get rid of all terms which explicitly dependent on $p^*(\boldsymbol{x})$.

In the following, we illustrate the idea of the proof of the above theorem for one dimension.

$$\mathcal{L}(\boldsymbol{w}) = \frac{1}{2}\int p^*(x)\|\nabla_x \log p(x;\boldsymbol{w}) - \nabla_x \log p^*(x)\|^2 \mathrm{d}x \tag{19.11}$$

$$= \frac{1}{2}\int p^*(x)\left[\left(\frac{\partial \log p(x;\boldsymbol{w})}{\partial x}\right)^2 - 2\frac{\partial \log p(x;\boldsymbol{w})}{\partial x}\frac{\partial \log p^*(x)}{\partial x} + \underbrace{\left(\frac{\partial \log p^*(x)}{\partial x}\right)^2}_{\text{independent of } \boldsymbol{w}}\right]\mathrm{d}x \tag{19.12}$$

$$= \frac{1}{2}\int p^*(x)\left(\frac{\partial \log p(x;\boldsymbol{w})}{\partial x}\right)^2 \mathrm{d}x - 2\int p^*(x)\frac{\partial \log p(x;\boldsymbol{w})}{\partial x}\frac{1}{p^*(x)}\frac{\partial p^*(x)}{\partial x}\mathrm{d}x + \text{const.} \tag{19.13}$$

$$= \int p^*(x)\left[\frac{1}{2}\left(\frac{\partial \log p(x;\boldsymbol{w})}{\partial x}\right)^2 + \frac{\partial^2 \log p(x;\boldsymbol{w})}{\partial x^2}\right]\mathrm{d}x - \underbrace{2\left[\frac{\partial \log p(x;\boldsymbol{w})}{\partial x}p^*(x)\right]_{-\infty}^{\infty}}_{=0 \text{ assumption}} + \text{const.} \tag{19.14}$$

$$= \int p^*(x)\left[\frac{1}{2}\left(\frac{\partial \log p(x;\boldsymbol{w})}{\partial x}\right)^2 + \frac{\partial^2 \log p(x;\boldsymbol{w})}{\partial x^2}\right]\mathrm{d}x + \text{const.} \tag{19.15}$$

19.3 Training and sampling procedure

With the previous considerations, we are now able to describe one possible approach for training a diffusion model and sampling data from it:

1. Apply forward diffusion to the training data (with $T \sim 1000$ discrete steps).
2. Learn the score function $\nabla_x \log p_t(\boldsymbol{x})$ for each of those timesteps by training an appropriate NN model on the resulting data of every diffusion step with the score matching objective

$$\sum_{t=1}^{T}\sum_{i=1}^{\mathrm{n}}\sum_{j=1}^{\mathrm{p}}\lambda(t)\left[\frac{1}{2}\left(\frac{\partial \log p_t(\boldsymbol{x}_i;\boldsymbol{w})}{\partial x_j}\right)^2 + \frac{\partial^2 \log p_t(\boldsymbol{x}_i;\boldsymbol{w})}{\partial x_j^2}\right]. \tag{19.16}$$

   λ is a positive time dependent weighting function. When adding more noise to the original distribution its score function gets smoother (see e.g. Figure 19.1), which makes it easier for the model to learn it. As a consequence also the overall magnitude of the loss decreases for later times. $\lambda(t)$ is thus chosen such that it increases for later times to balance the decreasing loss magnitudes.
3. Sample data from a prior distribution $\sim \mathcal{N}(0, \boldsymbol{I})$.
4. Solve the reverse SDE for sampled data as initial values, using the previously predicted score functions.

It should be noted that this is not the only way to implement score matching. Furthermore, there are other approaches to diffusion models which do not rely on the score function at all but instead iteratively estimate the unknown denoised image from the noisy images or alternatively the noise that was added to the noise-free image (see e.g. [HJA20]).

Literature

- Yang Song et al., "Score-based generative modeling through stochastic differential equations". 2020, [Son+20]
- Aapo Hyvärinen and Peter Dayan, "Estimation of non-normalized statistical models by score matching." 2005, [HD05] (Detailed derivation of score matching)

20 Attention

We start this chapter with a few words from ChatGPT[8]:

Write a short introductory paragraph for a chapter about attention/transformers in a Machine Learning lecture script, highlighting ChatGPT as motivational example.

In this chapter, we will delve into the topic of attention mechanisms and transformers, which have revolutionized the field of Natural Language Processing. One of the most remarkable examples of these models in action is OpenAI's ChatGPT, a large language model that can generate human-like responses to text inputs. This model is based on the transformer architecture and its ability to attend to different parts of the input, allowing it to understand the context and generate a relevant response. Through this chapter, we will gain a deeper understanding of attention and transformers, and see how they can be applied to various NLP tasks.

Figure 20.1: ChatGPT introduces itself

Attention is part of a new class of models, which we have not encountered yet. So far, we have always predicted categories or numbers for fixed input data one at a time. Considering the case of natural language processing (NLP), our output is usually a series of words, the length of which is typically not known a priori. Thus, in text generation one usually predicts one word after the other, taking only the input and the previous predictions into account. Such a model is called *autoregressive* and has the following general structure:

$$\boldsymbol{y}_t = f(\boldsymbol{x}_t, \boldsymbol{x}_{t-1}, \dots, \boldsymbol{y}_{t-1}, \boldsymbol{y}_{t-2}, \dots; \boldsymbol{w}), \tag{20.1}$$

where the $\boldsymbol{x}_i$ are the inputs and $\boldsymbol{y}_t$ the predictions of the model.

Also, with regard to the input, there are several crucial differences to conventional machine learning tasks:

- As mentioned above, the input is often a sentence or multiple sentences and as such the number of words i.e., the size of the input can vary.
- An isolated item (word) is often not informative, but must be regarded under consideration of the other words in the input.
- The order of items is assumed to be arbitrary. Of course, if we think about language, the relative positioning of words does indeed matter. However, as we will see, this information usually isn't reflected in the model architecture but has to be included into the input.

Before we can understand how attention handles these challenges, we have to clarify how exactly we represent language mathematically. This is certainly a huge topic on its own. For us, it suffices to know that words are usually embedded in a vector space, which respects the semantics of the words in the sense that e.g., words with a similar meaning also lie close to each other.

The following discussion is completely general, but it might still be helpful to keep a specific example of language processing in mind. Think for example about the problem of language translation, i.e., we have a sentence in one language as input and want to output the sentence in another language (which then might be of different size). We can then represent our input $\boldsymbol{X}$ as an $\mathfrak{p} \times \mathfrak{n}$ matrix, where $\mathfrak{n}$ is the number of items or words in the sentence and $\mathfrak{p}$ is the dimension of our word embedding space. It is important to note the difference to our usual convention where $\mathfrak{n}$ denotes the number of training samples. Here, all $\mathfrak{n}$ items make up one single sample.

[8] https://openai.com/blog/chatgpt

Attention addresses the problem of interdependence between words by updating the features of each word based on the features of all other words in the input. In this way, a new representation of each word in embedding space is created, which now depends on the meaning of the word in the given sentence. Additionally, attention provides a score for how important individual words are, similarly as we humans focus on specific parts of a sentence when interpreting it.

To illustrate this, let us look at the following two sentences:

1. The animal didn't cross the street because it was too tired.
2. The animal didn't cross the street because it was too wide.

The word "it" in the first sentence clearly refers to the animal, where in the second sentence it refers to the street. Capturing such a dependence is essential, e.g., for the case where we want to translate the sentence to German, since here the grammatical gender of the animal (das Tier) and the street (die Straße) are different. Figure 20.2 shows the results of a machine learning model using attention on these two input sentences. The intensity of the background color of each word indicates how much it influences the new representation of the word "it". We see that the model captures the dependence correctly, which we can take as a sign that attention is indeed a useful concept.

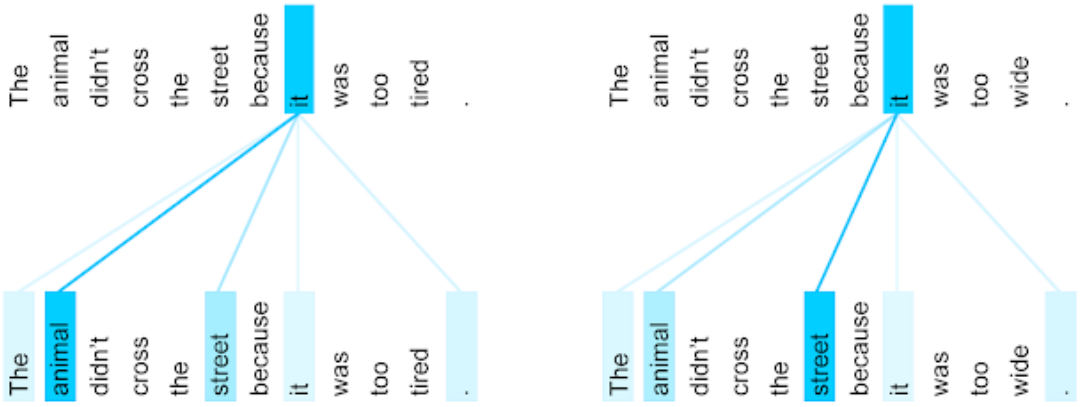


Figure 20.2: Example of how attention can capture dependencies between different words in a sentence.Figure from [Usz].

Implementation

We can imagine our input as a graph where each node corresponds to the representation of one word and edges between nodes indicate a relation between the corresponding words. In the whole discussion we look at a complete graph, i.e., we assume that a priori each word can influence the meaning of each other word. An example is shown in Figure 20.3.

In theory, there are now different ways how one could combine the representations of words to update the existing ones:

- Convolutional

$$\boldsymbol{x}_i' = \bigoplus_{j \neq i} \underbrace{c_i}_{1\times 1} \cdot \underbrace{\boldsymbol{\psi}(\boldsymbol{x}_j; \boldsymbol{w})}_{\mathfrak{p}\times 1} \tag{20.2}$$

- Attention

$$\boldsymbol{x}_i' = \bigoplus_{j \neq i} \underbrace{a(\boldsymbol{x}_i, \boldsymbol{x}_j; \boldsymbol{w}_a)}_{1\times 1} \cdot \underbrace{\boldsymbol{\psi}(\boldsymbol{x}_j; \boldsymbol{w}_t)}_{\mathfrak{p}\times 1} \tag{20.3}$$

- Message passing

$$\boldsymbol{x}_i' = \bigoplus_{j \neq i} \boldsymbol{\psi}(\boldsymbol{x}_i, \boldsymbol{x}_j; \boldsymbol{w}) \tag{20.4}$$

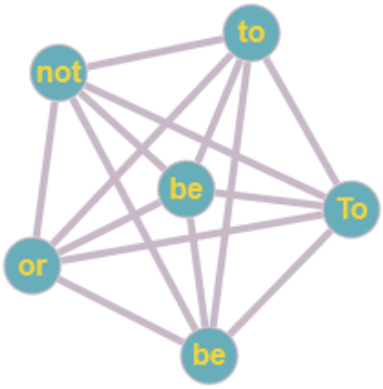


Figure 20.3: Graph representing the sentence "To be or not to be".

Here, $\bigoplus$ denotes a general aggregation operation like summing or averaging. ψ and a are functions, which are usually parameterized by a neural network. The function a is called "attention". In practice it turns out that the convolutional approach is too restrictive and not expressive enough, whereas message passing is too data hungry and computaionally expensive. Attention is thus the best compromise between expressiveness and computational efficiency.

One of the simplest examples for attention is the *dot product attention* given by

$$a(\boldsymbol{x}_i, \boldsymbol{x}_j) = \text{soft (arg) max} \left(\boldsymbol{x}_i^T \boldsymbol{x}_j\right). \tag{20.5}$$

This is however very limited as it has no learnable parameters at all.

20.1 Multi-head attention

We will instead look at the so-called *multi-head attention* (MHA) which is used in many NLP models including ChatGPT.

We start by defining the quantities

$$\begin{array}{lrcl} \text{Query} & \boldsymbol{Q} & = & \boldsymbol{W}_\text{Q} \quad \boldsymbol{X} \\ \text{Key} & \boldsymbol{K} & = & \boldsymbol{W}_\text{K} \quad \boldsymbol{X} \\ \text{Value} & \underbrace{\boldsymbol{V}}_{\mathfrak{p}\times\mathfrak{n}} & = & \underbrace{\boldsymbol{W}_\text{V}}_{\mathfrak{p}\times\mathfrak{p}} \quad \underbrace{\boldsymbol{X}}_{\mathfrak{p}\times\mathfrak{n}}. \end{array} \tag{20.6}$$

$\boldsymbol{W}_\text{Q}$, $\boldsymbol{W}_\text{K}$ and $\boldsymbol{W}_\text{V}$ are weight matrices, which are learnable parameters and hence increase the expressiveness of the model. The reason for those definitions as well as the naming will become clear soon. A usual choice is to set $\boldsymbol{W}_\text{V} = \boldsymbol{I}$. In the following, we will thus also refer to $\boldsymbol{V}$ as the original features.

The next step is to calculate the so called *scaled dot-product attention* given by

$$\boldsymbol{A} = \text{soft (arg) max} \left(\boldsymbol{K}^T \boldsymbol{Q} \frac{1}{\mathfrak{p}^{1/2}}\right). \tag{20.7}$$

The result is an $\mathfrak{n} \times \mathfrak{n}$ matrix, which loosely speaking encodes how compatible the keys $\boldsymbol{K}$ of individual items are with the questions $\boldsymbol{Q}$ asked by the other items (see also Figure 20.4). The normalization by $\mathfrak{p}^{1/2}$ leads to more stable gradients for training.

Next, we multiply our attention matrix $\boldsymbol{A}$ onto our original features $\boldsymbol{V}$ to obtain a new representation (see also Figure 20.5):

$$\boldsymbol{Z} = \boldsymbol{V}\boldsymbol{A} \tag{20.8}$$

These steps form the core of multi-head attention. We are already able to transform the representation of our original items meaningfully. However, the expressiveness of this model is still not large enough to reproduce the impressive results we have seen at the beginning of the chapter. In order to at least make a further step in this direction, we make the following improvements:

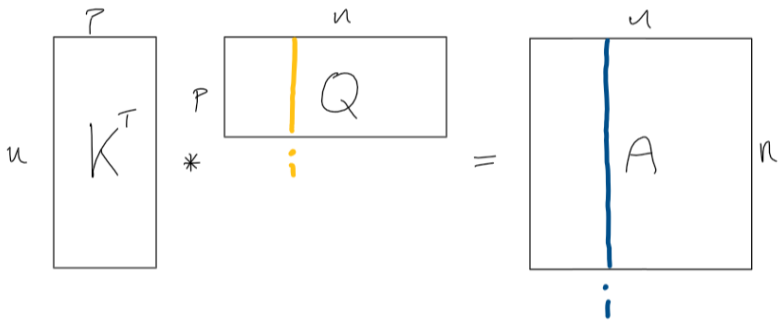


Figure 20.4: Illustration of the matrix multiplication performed in the scaled dot-product attention. The entries of $\boldsymbol{Q}$ marked in orange correspond to "the question asked by the i-th item". The blue entries of the resulting matrix correspond to "how compatible all keys are with the question asked by the i-th item".

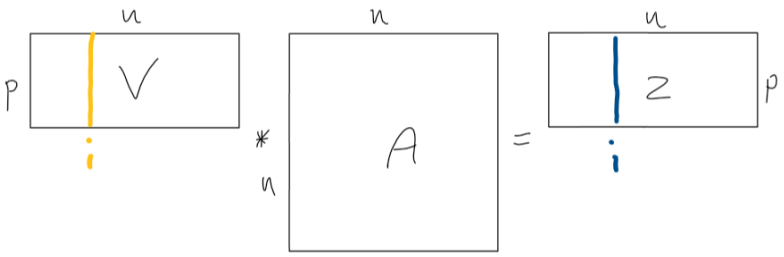


Figure 20.5: Illustration of the matrix multiplication performed when calculating the new representation $\boldsymbol{Z}$. The entries of $\boldsymbol{V}$ marked in orange correspond to the old representation of the i-th item, since usually one chooses $\boldsymbol{W}_{\mathrm{V}} = \boldsymbol{I}$ and thus $\boldsymbol{V} = \boldsymbol{X}$. The entries of $\boldsymbol{Z}$ marked blue are the new representation of item i.

- Make the model deeper. Instead of updating the representation of items only once, we add several layers to or network and repeat the procedure from above in each of them. This has already proven to be very efficient for models discussed earlier like CNNs.
- Add multiple attention heads. By this we mean that in each layer we learn multiple sets of $\{\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}\}$ and calculate the attention for each of them. The resulting representations are then merged and propagated to the next layer. This in theory gives the model the possibility to focus on different parts of the sentence and entangle different relations with each of the heads.

The final model then looks like this (see also Figure 20.6):

1. Define

$$\begin{aligned} \text{Query} \quad & \boldsymbol{Q}^{l,h} &&= \boldsymbol{W}_{\mathrm{Q}}^{l,h} && \boldsymbol{X}^{l} \\ \text{Key} \quad & \boldsymbol{K}^{l,h} &&= \boldsymbol{W}_{\mathrm{K}}^{l,h} && \boldsymbol{X}^{l} \\ \text{Value} \quad & \underbrace{\boldsymbol{V}^{l,h}}_{\mathfrak{p}^{\mathrm{aH}} \times \mathfrak{n}} &&= \underbrace{\boldsymbol{W}_{\mathrm{V}}^{l,h}}_{\mathfrak{p}^{\mathrm{aH}} \times \mathfrak{p}} && \underbrace{\boldsymbol{X}^{l}}_{\mathfrak{p} \times \mathfrak{n}} . \end{aligned} \tag{20.9}$$

   The superscript l indicates the layer, whereas h indicates the index of the head. Note also that the weight matrices are now of the dimension $\mathfrak{p}^{\mathrm{aH}} \times \mathfrak{p}$ with $H \cdot \mathfrak{p}^{\mathrm{aH}} = \mathfrak{p}$ and H the number of heads. This means that each of the heads operates in a lower feature dimension than in the original representation.

2. Calculate the scaled dot-product attention

$$\boldsymbol{A}^{l,h} = \text{soft (arg) max} \left(\left(\boldsymbol{K}^{l,h} \right)^{T} \boldsymbol{Q}^{l,h} \frac{1}{\mathfrak{p}^{1/2}} \right) . \tag{20.10}$$

3. Calculate new representations based on the attention

$$\boldsymbol{Z}^{l,h} = \boldsymbol{V}^{l,h}\boldsymbol{A}^{l,h}. \tag{20.11}$$

4. Concatenate the representations of all attention heads along the feature dimension to form one unified representation

$$\boldsymbol{Z}^{l} = \left[\boldsymbol{Z}^{l,1}; \ldots; \boldsymbol{Z}^{l,H}\right]. \tag{20.12}$$

5. Pass the result through a simple linear NN with weights $\boldsymbol{W}^l$ and a residual connection, which is then used as input for the next layer,

$$\boldsymbol{X}^{l+1} = \text{norm}\left(\boldsymbol{W}^{l}\boldsymbol{Z}^{l} + \boldsymbol{X}^{l}\right). \tag{20.13}$$

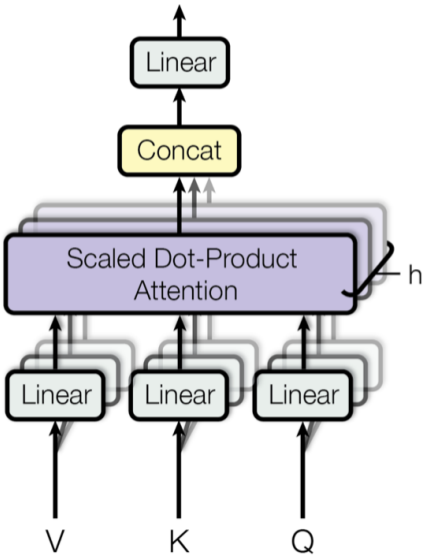


Figure 20.6: Architecture of a multi-head attention block. Figure from [Vas+17].

Literature

- Ashish Vaswani et al., "Attention is all you need". 2017, [Vas+17]

21 Transformer

We now want to see how we can use the multi-head attention mechanism in a network, which we can apply to an actual NLP problem.

21.1 Positional encoding

In the previous chapter, we saw that the response of a MHA block is invariant under a permutation of input tokens. Language however is sensitive to the relative positioning of words in a sentence, hence we need a way to encode this information in our input. Of course, we could just append our feature space by one dimension, which we use to pass the word position as an integer to the network. However, this creates several issues:

- Since the number of tokens in the input is usually not constrained, the integer representing word position can get arbitrarily high. This is undesired in deep neural networks.
- The number of feature dimensions is typically much larger than 1, which does not reflect the relative importance between word embedding and word position. The network thus likely has a very weak response to the word position.

These problems are solved by *positional encoding*. The idea is to spread the positional information over several neurons. How exactly do we do this? To answer this question, let us first look at the binary number system. In fact, this number system does something very similar, namely representing a single number using multiple digits. In this specific case, each digit may only take the values 0 or 1. So encoding the position of a token as a binary number, where each neuron represents one digit, would already be a step in the right direction as the values of individual neurons are constrained and the higher number of neurons needed enhances the importance of the positional information. Since the neurons are however not naturally constrained to binary values, we can go even one step further and allow the neurons to take continuous values in the interval $[-1, 1]$. This can be done by realizing that during counting, binary digits show an oscillating behavior. The first digit flips between 0 and 1 in each counting step, the second digit in every second step, the third digit in every fourth step, etc. One could say that the different digits oscillate with different frequencies. A natural analogue in the continuous domain are $\sin(\omega t)$ and $\cos(\omega t)$ with different frequencies ω. This is the motivation to define a positional embedding as

$$E_{(2k),j} = \sin\left(j \cdot \exp\left(-\frac{2k \cdot \log(10000)}{\mathfrak{p}}\right)\right), \tag{21.1}$$

$$E_{(2k+1),j} = \cos\left(j \cdot \exp\left(-\frac{2k \cdot \log(10000)}{\mathfrak{p}}\right)\right). \tag{21.2}$$

Here, $E_{n,m}$ specifies the value of the n-th feature in the positional encoding vector of a token in the m-th position and $\mathfrak{p}$ is the number of dimensions of the positional encoding vector. The factor of 10000 is arbitrary and makes sure that the positional encodings are unique for a sufficiently large number of token positions. The resulting matrix $\boldsymbol{E}$ is shown in Figure 21.1. Note that there are other possible ways to realize positional encodings.

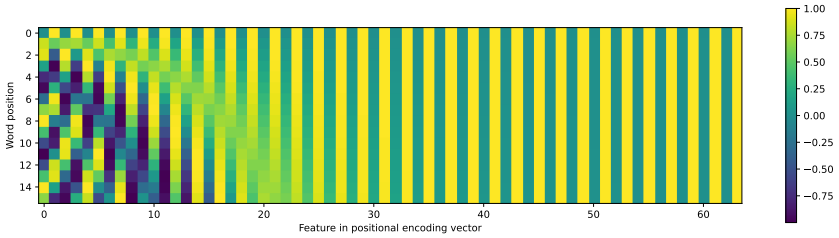


Figure 21.1: Visualization of the positional encoding matrix $\boldsymbol{E}$. The rows correspond to different token positions, whereas the columns represent different features in the embedding vectors.

Another nice way to think about the positional encoding is to notice that every $\sin(j\omega)$, $\cos(j\omega)$ (j being the token position) pair with the same frequency defines a point on the unit circle S^1.

With increasing token position, this point rotates around the unit circle with fixed angular velocity ω. Taking all sin, cos pairs together, they can thus be interpreted as many hands of a clock, all rotating at different velocities. And just as each configuration of two ordinary watch hands uniquely defines a time within a 12h interval, the configuration of our watch hands uniquely defines a word position. For more details on the intuition behind positional encoding refer e.g. to [Kaz19].

The only remaining question is how we combine the positional information with the token features. There are in theory two sensible approaches: 1. We could simply append the positional encoding to the feature vector of each token. 2. We could add the positional embedding vector onto the feature vector of each token. In practice, it is most common to choose the second option, as it requires less memory and performs similarly well.

21.2 Architecture

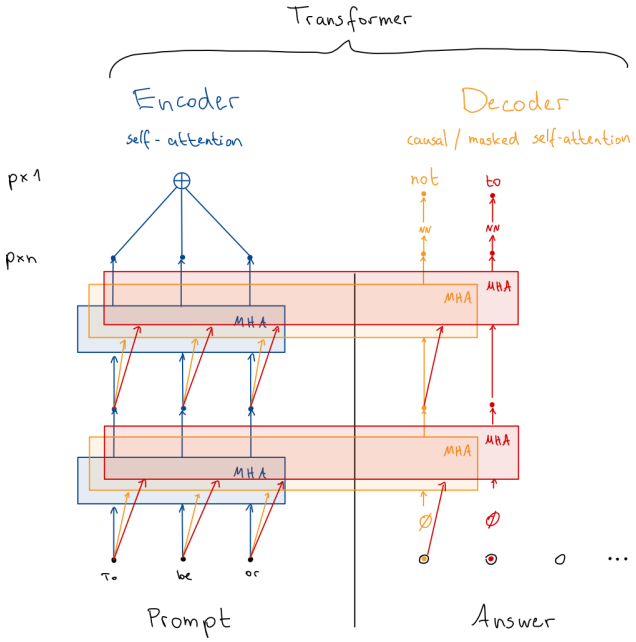


Figure 21.2: Sketch of the transformer architecture.

We will now look at a transformer architecture similar to the one from the original paper [Vas+17]. The architecture is illustrated in Figure 21.2. The transformer consists of an encoder and a decoder block. The encoder takes a series of input tokens also called the prompt and passes them through a positional encoding layer which adds the positional information to each token as discussed in the previous paragraph. The tokens are then further passed through multiple multi-head attention layers. Finally, the output of those layers is aggregated into one vector by some aggregation function which could, e.g., be the mean, sum or max function. This final aggregated output can for example be used as guide for a generative image model, which is further described below.

The decoder is used to generate new text based on the input, which we will also call the answer. Here the output is generated iteratively token by token. In the first iteration, the tokens processed by the encoder as well as a special token positioned after the input tokens, which does not carry any semantic information but only indicates that the transformer response should be saved there, are fed into the first MHA layer. However, only the outputs corresponding to tokens not present in the input are passed on to the next MHA layer. The inputs for the remaining positions are taken from the outputs of the corresponding encoder layer (see Figure 21.2). In this way, the tokens are passed through multiple MHA layers, where the final one yields a prediction for the first token of the output. In the next iteration, this output as well as a new empty token are used as decoder input. The process is then continued until the decoder prediction yields a predefined stop token, which terminates the process. Initially, the length of the input fed to each MHA layer in the decoder grows with each iteration. However, for long prompts and answers, it is common to restrict the decoder input to a certain size. If this limit is reached, the first input token is dropped

every time a new output token is generated.

This is only one way of realizing a transformer model. In particular, there are also examples which only use an encoder like e.g., "BERT" or only a decoder like e.g., "GPT".

21.3 Guided image synthesis

One of the most prominent examples of a text guided generative image model is OpenAI's DALL-E2. The user can describe an image with words, and the model generates an image based on this prompt. The model mainly relies on three parts:

1. A text encoder in the form of a transformer model
2. An image classification model e.g., a CNN
3. A diffusion model for the actual image generation process

For training, a large amount of text-image pairs are used. One way to acquire this data is to take pictures from the internet and assign a text snippet to them, which is for example taken from the image caption. During training, the text and images are separately encoded with the respective encoding/classification model. Crucially, the same target space is chosen for both the text and image representation. In this way one can compute the outer product between both representation vectors as shown in Figure 21.3. The loss function is then chosen such that larger values on the diagonal of the resulting matrix reduce the loss, whereas non-vanishing off-diagonal elements are penalized. In other words, the model learns to represent the encoded text and encoded image of a matching pair as the same (or at least similar) feature vector in the same abstract space.

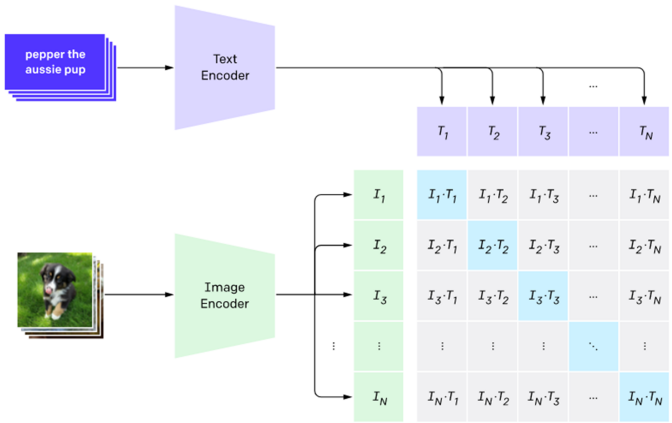


Figure 21.3: Visualization of the forward pass used during training of a text guided image synthesis model.

For the generation of an image, the user prompt is encoded with the trained transformer model. The encoded representation can then be used as a guide in a separately trained diffusion model, together with a subsequent image decoder to produce the actual image corresponding to the user prompt. This procedure is sketched in Figure 21.4.

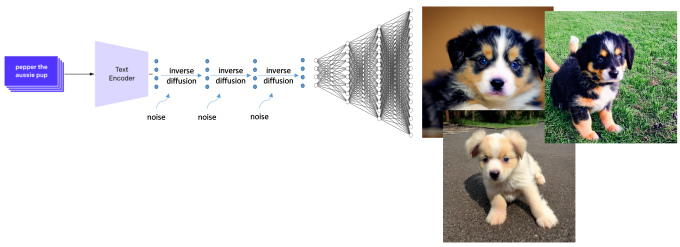


Figure 21.4: Diagram of the image generation process in a text guided image synthesis model.

Example 21.1: Prediction of chemical reactions

Transformers also have exciting and surprising applications outside the field of natural language processing. One example is the prediction of the products of a chemical reaction given the reactants. Machine learning approaches in this area commonly rely on a method called SMILES (simplified molecular-input line-entry system, see [Wei88]). The basic idea is to convert chemical formulas into text strings, as shown exemplarily in Figure 21.5.

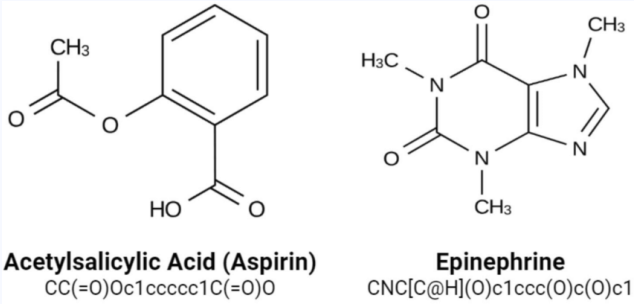


Figure 21.5: Examples of two chemical formulas with their respective SMILES string. Figure from [Yas20].

Now that the chemical information is available as text, one can treat the problem of reaction prediction as an effective machine translation problem. The reactants are passed as inputs to a transformer model, which then outputs a prediction for the products of the reaction in the form of another SMILES representation. This procedure is explained in detail, e.g., in [Sch+19].
Another similar problem is atom mapping. If there are e.g., multiple fluorine atoms in the reactants and products, the question is how exactly these atoms rearranged, or in other words, which of the fluorine atoms in the products corresponds to which fluorine atom in the reactants. A possible approach to this problem is described in [Sch+20], which makes direct use of a self attention mechanism. Essentially the authors find that attention weights between different tokens in the SMILES string of the reaction, learned by an appropriate transformer, contain all the relevant mapping information. An example is shown in Figure 21.6, showcasing the correct mapping between two fluorine atoms in a chemical reaction.

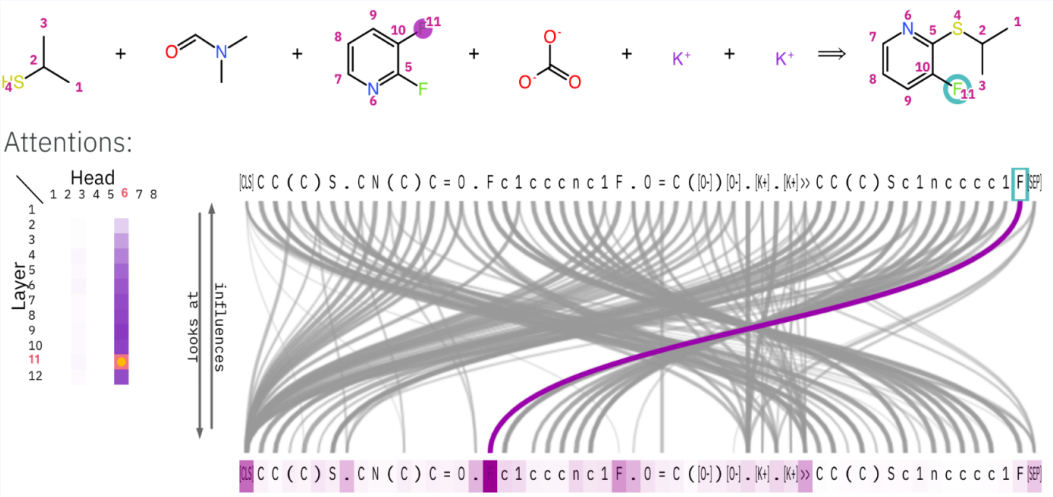


Figure 21.6: The top of the figure shows the chemical reaction which was fed into the model, including the predicted mapping (same numbers correspond to mapped atoms). On the bottom, the attention weights of one attention head are shown. The intensity of the purple background on the lower SMILES string indicates the probability distribution for the corresponding atom to be mapped to the selected fluorine atom in the top string. The purple line indicates the most probable mapping, which is in fact also the correct one. Figure from [Sch+20].

Both methods described above are particularly powerful as they are unsupervised. The model learns the laws governing chemical reactions purely based on observations without the need for labeling done by a human, which is quite impressive.

Literature

- Ashish Vaswani et al., "Attention is all you need". 2017, [Vas+17]
- William L Hamilton, *Graph representation learning*. 2020, Chap. 5 [Ham20]

22 Graph neural networks

In section 20, we mentioned that we can imagine text input as a complete graph. This means that we have already dealt with neural networks on graphs in our treatment of transformers. In this section, we will formalize the idea of *graph neural networks* (GNNs). The main difference to normal MLPs is that we no longer have just one large feature vector which is passed through the individual layers. Instead, the features are structured according to an underlying graph where feature vectors are assigned to the nodes, the edges or the entire graph and "the whole graph is passed through the network".

22.1 Equivariance/invariance

Before we can take a look at architectures of GNNs, we want to define the properties of equivariance and invariance. They are used to state that the functions are consistent with the underlying symmetry of the data. This concept is important in practice. For example, the prediction of the toxicity of a molecule should be invariant under rotations of the molecule. The specific definition of equivariance/invariance depends on the domain of the function and the required symmetry.

Functions on sets

Given a set with $\mathfrak{n}$ elements, a function on this set has the form $f\colon \mathbb{R}^{\mathfrak{p}\times\mathfrak{n}} \to \mathbb{R}^{\mathfrak{p}\times\mathfrak{n}}$, where $\mathfrak{p}$ is the number of feature dimensions of each individual item. These functions occur, for example, in multi-agent reinforcement learning or in the context of economics. As the elements of a set don't have a specific order, we usually want a function that is equivariant/invariant to a permutation $\boldsymbol{P} \in \mathbb{R}^{\mathfrak{n}\times\mathfrak{n}}$ of inputs. A function f_{eq} is permutation equivariant if

$$f_{\text{eq}}(\boldsymbol{X}\boldsymbol{P}) = f_{\text{eq}}(\boldsymbol{X})\boldsymbol{P} \tag{22.1}$$

holds for any input $\boldsymbol{X} = [\boldsymbol{x}_1, \dots, \boldsymbol{x}_{\mathfrak{n}}] \in \mathbb{R}^{\mathfrak{p}\times\mathfrak{n}}$. An example of an equivariant function is $f(\boldsymbol{X}) = [\psi(\boldsymbol{x}_1), \dots, \psi(\boldsymbol{x}_n)]$, where ψ is an arbitrary function on a single sample. We note that equivariance is similar to "transforms as a vector" in the language of physics.

A function f_{inv} is permutation-invariant if

$$f_{\text{inv}}(\boldsymbol{X}\boldsymbol{P}) = f_{\text{inv}}(\boldsymbol{X}) \tag{22.2}$$

holds. An example that fulfills this property is $f(\boldsymbol{X}) = \phi(\bigoplus_i \psi(\boldsymbol{x}_i))$, where ψ and ϕ are arbitrary functions and $\bigoplus_i$ signifies an aggregator, i.e., any commutative operator such as the sum, mean, minimum or maximum.

Functions on graphs

Graphs are the ordered pair $G = (\mathcal{V}, \mathcal{E})$ consisting of vertices $\mathcal{V}$ and edges $\mathcal{E}$. We restrict ourselves here to the case where only vertices are endowed with features, since the extension to edge features is straightforward. Functions on graphs will depend both on the node features $\boldsymbol{X}$ and

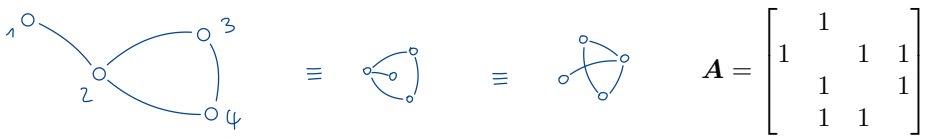


Figure 22.1: Example of a graph with two equivalent representations and its adjacency matrix.

the adjacency matrix $\boldsymbol{A}$. Note that the features $\boldsymbol{X}$ are a $\mathfrak{p} \times \mathfrak{n}$-matrix, where $\mathfrak{n}$ now is the number of vertices. This means that already the dimension of a single sample is $\mathfrak{p}\cdot\mathfrak{n}$. The adjacency matrix encodes the information of which vertex is connected to which vertex. In our case of undirected graphs, the adjacency matrix is symmetric, $\boldsymbol{A}^\top = \boldsymbol{A}$. An example is given in Figure 22.1.

As in the case of functions on sets, we consider permutations $\boldsymbol{P}$ of nodes. Thereby, the node properties are transformed like $\boldsymbol{X} \to \boldsymbol{X}\boldsymbol{P}$ and the adjacency matrix like $\boldsymbol{A} \to \boldsymbol{P}^\top\boldsymbol{A}\boldsymbol{P}$. For a

function to be permutation equivariant on a graph, it must satisfy the condition

$$f_{\mathrm{eq}}^{\mathrm{G}}(\boldsymbol{X}\boldsymbol{P}, \boldsymbol{P}^\top \boldsymbol{A}\boldsymbol{P}) = f_{\mathrm{eq}}^{\mathrm{G}}(\boldsymbol{X}, \boldsymbol{A})\boldsymbol{P}. \tag{22.3}$$

A function on a graph is permutation-invariant if it fulfills

$$f_{\mathrm{inv}}^{\mathrm{G}}(\boldsymbol{X}\boldsymbol{P}, \boldsymbol{P}^\top \boldsymbol{A}\boldsymbol{P}) = f_{\mathrm{inv}}^{\mathrm{G}}(\boldsymbol{X}, \boldsymbol{A}). \tag{22.4}$$

Equivariant functions on graphs can be constructed using a locally invariant neighborhood function $g(\boldsymbol{x}_i, \boldsymbol{X}_{\mathcal{N}_i})$. The function g only depends on the feature $\boldsymbol{x}_i$ and the features $\boldsymbol{X}_{\mathcal{N}_i}$ of the graph neighbors $\mathcal{N}_i$ of the vertex i. The global equivariant function is then given as $f(\boldsymbol{X}, \boldsymbol{A}) = [g(\boldsymbol{x}_i, \boldsymbol{X}_{N_1}), \dots, g(\boldsymbol{x}_n, \boldsymbol{X}_{N_n})]$, whereby the dependence on the adjacency matrix $\boldsymbol{A}$ is implicit in $\mathcal{N}_i$. An example of a permutation-invariant function on the entire graph is $f(\boldsymbol{X}, \boldsymbol{A}) = \phi(\bigoplus_i \psi(\boldsymbol{x}_i))$, where ϕ and ψ are arbitrary functions, for instance neural networks. The function f no longer depends on the structure of the graph.

There are several popular ways to construct a locally invariant neighborhood function g. In our current setting, they are formulated as follows:

convolutional	$g\left(\boldsymbol{x}_i, \bigoplus_{j\in\mathcal{N}_i} c_{ij} \cdot \psi(\boldsymbol{x}_j)\right)$	graph convolutional NN (GCN)
attentional	$g\left(\boldsymbol{x}_i, \bigoplus_{j\in\mathcal{N}_i} a(\boldsymbol{x}_i, \boldsymbol{x}_j) \cdot \psi(\boldsymbol{x}_j)\right)$	graph attention transformer (GAT)
message-passing	$g\left(\boldsymbol{x}_i, \bigoplus_{j\in\mathcal{N}_i} \psi(\boldsymbol{x}_i, \boldsymbol{x}_j)\right)$	physics engine

We have already encountered the above methods in the section on attention, since transformer encoders are nothing more than GATs on complete graphs. So GATs generalize this idea by allowing graphs that are not complete.

22.2 Graph convolutional networks

Graph convolutional networks were introduced by Kipf and Willing in 2016 [KW16]. A sketch of the architecture is shown in Figure 22.2. From layer ℓ to layer $\ell + 1$, the features $\boldsymbol{X}$ are changed by

$$\boldsymbol{X}^{\ell+1} = \sigma\left(\boldsymbol{W}^\ell \cdot \boldsymbol{X}^\ell \cdot \boldsymbol{S}\right). \tag{22.5}$$

The non-linearity originates from the activation function σ, where for example ReLU can be chosen. The weight matrices $\boldsymbol{W}^\ell \in \mathbb{R}^{\mathfrak{p}\times\mathfrak{p}}$ are the layer-specific linear transformations that "mix"

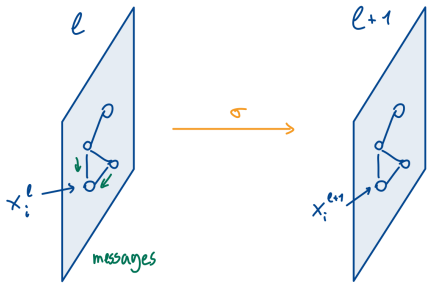


Figure 22.2: Transformation between two layers in a GCN.

features. Note that their size is independent of the number of vertices, as we would expect. The multiplication with $\boldsymbol{W}$ corresponds to a single MLP layer (without activation) applied to the $\mathfrak{p}$ features of each node individually.

By multiplication with $\boldsymbol{W}$, the features of the nodes did not influence each other. The *propagation of features* along the edges of the graph is provided by the "normalized adjacency matrix

with self-loops" $\boldsymbol{S} \in \mathbb{R}^{\mathtt{n}\times\mathtt{n}}$. It entails the structure of the graph and is the same for every layer. It is defined as

$$\boldsymbol{S} = \boldsymbol{D}^{-\frac{1}{2}}\,(\boldsymbol{A}+\boldsymbol{I})\,\boldsymbol{D}^{-1/2}, \quad \text{with} \quad \boldsymbol{D} = \operatorname{diag}((\boldsymbol{A}+\boldsymbol{I})\mathbf{1}_{\mathtt{n}}). \tag{22.6}$$

The exponent $-\frac{1}{2}$ can be understood element-wise, as the degree matrix $\boldsymbol{D}$ is diagonal. It contains the degrees of the adjacency matrix with self-loops, $\boldsymbol{A}+\boldsymbol{I}$ (the degree of a vertex is the number of connected edges). We can write the transformation of a single node feature in Equation 22.5 as

$$\boldsymbol{x}_i^{\ell+1} = \sigma\left(\boldsymbol{W}^{\ell}\sum_{j\in\mathcal{N}_i}\frac{\boldsymbol{x}_j^{\ell}}{\sqrt{|\mathcal{N}_i|\cdot|\mathcal{N}_j|}}\right). \tag{22.7}$$

This illustrates that features are being passed from neighboring nodes and added up. In Figure 22.2, this is represented by the small green arrows. As the matrix multiplication is associative, we note that

$$\left(\boldsymbol{W}^{\ell}\boldsymbol{X}^{\ell}\right)\boldsymbol{S} \quad = \quad \boldsymbol{W}^{\ell}\left(\boldsymbol{X}^{\ell}\boldsymbol{S}\right) \tag{22.8}$$

$$\begin{matrix}\text{first transform,}\\ \text{then propagate}\end{matrix} \quad = \quad \begin{matrix}\text{first smooth,}\\ \text{then transform}\end{matrix} \tag{22.9}$$

which means that the order of transformation and propagation/smoothing does not play a role.

A limitation of GCNs is the oversmoothing that is observed in practice. Oversmoothing here means that interacting notes have very similar features after the application of a few layers. Unlike other models seen previously, increasing the depth in GCNs does not make the model better, but rather worsens the problem of oversmoothing. This makes the application most useful when neighboring nodes have similar labels/responses (also known as homophily). Possible solutions to the problem such as dropouts were suggested.

We have not encountered this problem with CNNs, as there the filters have more degrees of freedom. In GCNs, the propagation along edges of the graph is not weighted with learned parameters and the same linear transformation $\boldsymbol{W}^{\ell}$ is used for every node. One possible approach to solve this problem would be to simply convert the graph into an image. However, information about the equivariance/invariance would then be lost.

Simple(r) GCN

In order to reduce complexity and avoid possible redundant computation, a simpler model was introduced in [Wu+19], called simple graph convolution. This approach can speed up calculations while only slightly reducing accuracy in many applications. Instead of applying the non-linearity σ every layer, it is only used in the final one. That means that the expressions

$$\boldsymbol{X}^{\ell+1} = \sigma\left(\boldsymbol{W}^{\ell}\sigma\left(\boldsymbol{W}^{\ell-1}\boldsymbol{X}^{\ell-1}\boldsymbol{S}\right)\boldsymbol{S}\right) \tag{22.10}$$

get replaced by

$$\boldsymbol{X}^{\ell+1} = \sigma\left(\boldsymbol{W}^{\ell}\boldsymbol{W}^{\ell-1}\boldsymbol{X}^{\ell-1}\boldsymbol{S}^{2}\right). \tag{22.11}$$

We collapse all linear transformations and call it $\boldsymbol{W}$. Thus, we obtain the output of L layers as

$$\boldsymbol{Y} = \sigma\left(\boldsymbol{W}\boldsymbol{X}\boldsymbol{S}^{L}\right), \tag{22.12}$$

where $\boldsymbol{X}$ are the input features. The loss of non-linearity results in even lower expressivity, but the model works well on data that shows a high degree of homophily. The expressivity can be increased by increasing the depth of the model.

GCNII

In [Che+20], the so-called GCNII was proposed, where the methods of "initial residual" and "identity mapping" were added to the standard GCN. The transformation between two layers is given by

$$\boldsymbol{X}^{\ell+1} = \sigma\big((\beta^{\ell}\boldsymbol{W}^{\ell} + \underbrace{(1-\beta^{\ell})\boldsymbol{I}}_{\text{identity mapping}})((1-\alpha^{\ell})\boldsymbol{X}^{\ell}\boldsymbol{S} + \underbrace{\alpha^{\ell}\boldsymbol{X}^{0}}_{\text{initial residual}})\big), \tag{22.13}$$

where β^ℓ and α^ℓ are layer-dependent parameters. A common choice for β is $\beta^\ell = \lambda/\ell$, where λ is a hyperparameter. While it appears that the identity mapping simply redefines the weights $\boldsymbol{W}^\ell$, it actually has an effect if weight regularization is applied. The initial residual is inspired by the skip connections that are used in ResNet, for instance. The skip connections are visualized in

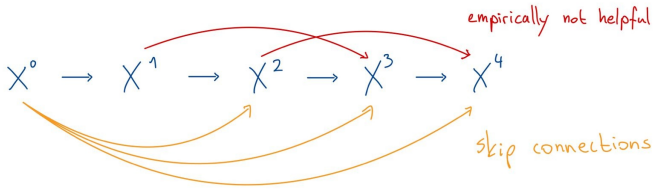


Figure 22.3: Sketch of the initial residuals used in GCNII. The other skip connections have proven empirically not to be helpful.

Figure 22.3. Only the connections from the initial features are used, as the other connections have not proven useful in practice.

Even more expressive?

There are various ideas in a growing literature on how to make GNNs even more expressive. A few prominent ideas are:

- Use attention.
- Tune the aggregation.
- Apply normalization.

Geometric deep learning

We have been working with models that take into account the symmetry of the data for more than just this section. For example, a CNN obeys translational symmetry on a Cartesian grid and MHA is equivariant under permutations. A compilation of symmetries and the adapted architectures is shown in Table 22.1.

Domain	Symmetry	Architecture
Cart. lattice	translation	CNN
sets	permutation	MHA
graphs	permutation	GNN
sphere	SO(3)	equivariant NN

Table 22.1: Symmetry of several domains with suitable machine learning model.

23 Probabilistic graphical models

In this chapter we will take a closer look at probabilistic graphical models (PGMs). In simple terms, these models allow us to establish conditional relationships among a set of random variables. More specifically, PGMs represent the following prior knowledge we have of a given system:

- Conditional independencies between random variables through a graph
- Conditional probabilities through factors associated with nodes of that graph

PGMs can be divided into two groups:

1. Undirected models, e.g., spin glasses and Markov random fields
2. Directed models, e.g., Hidden Markov Models (HMMs), Gaussian mixture models, Kalman filter, Markov decision processes (used in reinforcement learning, see next chapter)

As the names suggest, the difference between them is that the underlying graph of the model is undirected in the first and directed in the second case. We will focus on directed graphical models which can formally be defined as follows:

A directed graphical model (DGM) or equivalently belief network or Bayesian network is given by a directed acyclic graph (DAG) with

- one node per random variable
- conditional probability distributions $p(X_i|X_{\mathrm{pa}(i)})$
- a joint probability distribution $p(X_1, \ldots, X_n) = \prod_i p(X_i|X_{\mathrm{pa}(i)})$

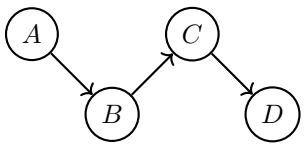
where $\mathrm{pa}(i)$ denotes the parents of node i. To better understand the meaning of this definition, let us look at an example of four random variables A, B, C, D. In general, the joint probability distribution of those random variables can be decomposed according to the chain rule of probability,

$$\begin{aligned} p(A, B, C, D) &= p(A|B, C, D) \cdot p(B|C, D) \cdot p(C|D) \cdot p(D) && (23.1)\\ &= p(D|A, B, C) \cdot p(C|A, B) \cdot p(B|A) \cdot p(A) && (23.2)\\ &= \ldots && (23.3) \end{aligned}$$

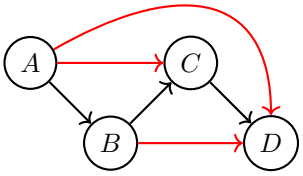
A given DGM allows us to simplify this expression of the joint. For example if we consider the DGM given in Figure 23.1a we can write

$$p(A, B, C, D) = p(D|C) \cdot p(C|B) \cdot p(B|A) \cdot p(A). \tag{23.4}$$

In contrast, the decomposition of the joint as in Equation 23.2 would be represented by the graph in Figure 23.1b.



(a) Example of a simple DGM, representing the conditional dependencies given in Equation 23.4.



(b) DGM representation of Equation 23.2. Edges marked red represent redundant relations in the DGM of Figure 23.1a.

In words the diagram states that D is conditionally independent of A and B given C. This can also be written as:

$$D \perp A, B \mid C. \tag{23.5}$$

23.1 Bayesian inference for the Gaussian

As preparation for the discussion of the Kalman filter, we consider the problem of Bayesian inference for Gaussian distributions, similar to what we did in section 8 for the Bayesian inference of Bernoulli distributions. Given a likelihood $p(\boldsymbol{x}|\boldsymbol{z})$ which follows a Gaussian distribution, we want to obtain its conjugate prior. In this case the conjugate prior is Gaussian again so that we find

$$\underbrace{p(\boldsymbol{z}|\boldsymbol{x})}_{\sim\mathcal{N}(\boldsymbol{\mu}_{\boldsymbol{z}|\boldsymbol{x}},\boldsymbol{\Sigma}_{\boldsymbol{z}|\boldsymbol{x}})} \propto \underbrace{p(\boldsymbol{x}|\boldsymbol{z})}_{\sim\mathcal{N}(\boldsymbol{C}\boldsymbol{z},\boldsymbol{\Sigma}_{\boldsymbol{x}|\boldsymbol{z}})} \cdot \underbrace{p(\boldsymbol{z})}_{\sim\mathcal{N}(\boldsymbol{\mu}_{\boldsymbol{z}},\boldsymbol{\Sigma}_{\boldsymbol{z}})}. \tag{23.6}$$

Here we choose the mean of $p(\boldsymbol{x}|\boldsymbol{z})$ to be $\boldsymbol{C}\boldsymbol{z}$ as this is one of the assumptions of the Kalman filter (see below). The task is now to find expressions for $\boldsymbol{\mu}_{\boldsymbol{z}|\boldsymbol{x}}$ and $\boldsymbol{\Sigma}_{\boldsymbol{z}|\boldsymbol{x}}$ depending on the means and covariances of the likelihood and prior. To this end, we take the natural logarithm on both sides of Equation 23.6 which yields

$$\begin{aligned} -2\ln p(\boldsymbol{z}|\boldsymbol{x}) &= \text{const.} + (\boldsymbol{x}-\boldsymbol{C}\boldsymbol{z})^T\boldsymbol{\Sigma}_{\boldsymbol{x}|\boldsymbol{z}}^{-1}(\boldsymbol{x}-\boldsymbol{C}\boldsymbol{z}) + (\boldsymbol{z}-\boldsymbol{\mu}_{\boldsymbol{z}})^T\boldsymbol{\Sigma}_{\boldsymbol{z}}^{-1}(\boldsymbol{z}-\boldsymbol{\mu}_{\boldsymbol{z}}) && (23.7)\\ &= \text{const.} + (\boldsymbol{z}-\boldsymbol{\mu}_{\boldsymbol{z}|\boldsymbol{x}})^T\boldsymbol{\Sigma}_{\boldsymbol{z}|\boldsymbol{x}}^{-1}(\boldsymbol{z}-\boldsymbol{\mu}_{\boldsymbol{z}|\boldsymbol{x}}). && (23.8)\end{aligned}$$

In order to find $\boldsymbol{\Sigma}_{\boldsymbol{z}|\boldsymbol{x}}$, we isolate all terms quadratic in $\boldsymbol{z}$:

$$\begin{aligned} \boldsymbol{z}^T\boldsymbol{C}^T\boldsymbol{\Sigma}_{\boldsymbol{x}|\boldsymbol{z}}^{-1}\boldsymbol{C}\boldsymbol{z} + \boldsymbol{z}^T\boldsymbol{\Sigma}_{\boldsymbol{z}}^{-1}\boldsymbol{z} &= \boldsymbol{z}^T\boldsymbol{\Sigma}_{\boldsymbol{z}|\boldsymbol{x}}^{-1}\boldsymbol{z} && (23.9)\\ \boldsymbol{z}^T\left(\boldsymbol{C}^T\boldsymbol{\Sigma}_{\boldsymbol{x}|\boldsymbol{z}}^{-1}\boldsymbol{C} + \boldsymbol{\Sigma}_{\boldsymbol{z}}^{-1}\right)\boldsymbol{z} &= \boldsymbol{z}^T\boldsymbol{\Sigma}_{\boldsymbol{z}|\boldsymbol{x}}^{-1}\boldsymbol{z} && (23.10)\end{aligned}$$

$$\implies \boldsymbol{\Sigma}_{\boldsymbol{z}|\boldsymbol{x}} = \left(\boldsymbol{C}^T\boldsymbol{\Sigma}_{\boldsymbol{x}|\boldsymbol{z}}^{-1}\boldsymbol{C} + \boldsymbol{\Sigma}_{\boldsymbol{z}}^{-1}\right)^{-1} \tag{23.11}$$

This result can be summarized by saying that the precisions of the distributions add up to the precision of the posterior.

Similarly, to find $\boldsymbol{\mu}_{\boldsymbol{z}|\boldsymbol{x}}$, we can isolate all terms that are linear in $\boldsymbol{z}$,

$$\begin{aligned} -2\boldsymbol{z}^T\boldsymbol{C}^T\boldsymbol{\Sigma}_{\boldsymbol{x}|\boldsymbol{z}}^{-1}\boldsymbol{x} - 2\boldsymbol{z}^T\boldsymbol{\Sigma}_{\boldsymbol{z}}^{-1}\boldsymbol{\mu}_{\boldsymbol{z}} &= -2\boldsymbol{z}^T\boldsymbol{\Sigma}_{\boldsymbol{z}|\boldsymbol{x}}^{-1}\boldsymbol{\mu}_{\boldsymbol{z}|\boldsymbol{x}} && (23.12)\\ \boldsymbol{\Sigma}_{\boldsymbol{z}|\boldsymbol{x}}\left(\boldsymbol{C}^T\boldsymbol{\Sigma}_{\boldsymbol{x}|\boldsymbol{z}}^{-1}\boldsymbol{x} + \boldsymbol{\Sigma}_{\boldsymbol{z}}^{-1}\boldsymbol{\mu}_{\boldsymbol{z}}\right) &= \boldsymbol{\mu}_{\boldsymbol{z}|\boldsymbol{x}} && (23.13)\end{aligned}$$

$$\implies \boldsymbol{\mu}_{\boldsymbol{z}|\boldsymbol{x}} = \frac{\boldsymbol{C}^T\boldsymbol{\Sigma}_{\boldsymbol{x}|\boldsymbol{z}}^{-1}\boldsymbol{x} + \boldsymbol{\Sigma}_{\boldsymbol{z}}^{-1}\boldsymbol{\mu}_{\boldsymbol{z}}}{\boldsymbol{C}^T\boldsymbol{\Sigma}_{\boldsymbol{x}|\boldsymbol{z}}^{-1}\boldsymbol{C} + \boldsymbol{\Sigma}_{\boldsymbol{z}}^{-1}}. \tag{23.14}$$

This is a slight abuse of notation since we "divide" by a matrix where the order of operations matters. The mean of the posterior is thus a weighted sum of the prior mean $\boldsymbol{\mu}_{\boldsymbol{z}}$ and the observation $\boldsymbol{x}$.

23.2 State space models and Kalman filter

We now introduce an important class of DGMs, so-called state space models. They consist of latent state variables $\boldsymbol{z}_t$ and observables $\boldsymbol{x}_t$ with conditional dependencies shown in Figure 23.2.

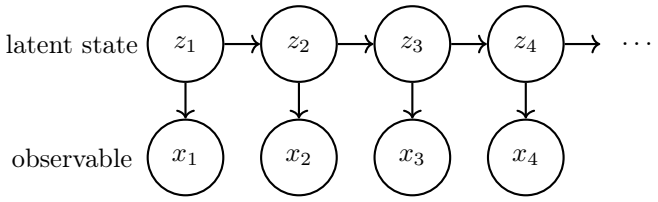


Figure 23.2: The graph of a state space model

The latent state variables describe the "true" state of a given system which evolves in time. Each evolution step of the system only depends on the previous state of the system $z_t \perp z_{1:t-2} \mid z_{t-1}$.

This is also called "Markov property". Usually the true state of the system is not known and can only be determined through certain experiments. This fact is modeled by the observables x_t which are random variables describing the outcome of some experiment given a current latent state z_t. Notably those observables are independent of future latent states and conditionally independent of previous system states given the current one.

An important example of a state space model is the Kalman filter. It assumes both the evolution of latent states and the measurement process to be linear with Gaussian noise, i.e.,

$$\boldsymbol{z}_t = \boldsymbol{A}\boldsymbol{z}_{t-1} + \boldsymbol{w}_t, \quad \boldsymbol{w}_t \sim \mathcal{N}(0, \boldsymbol{\Gamma}); \tag{23.15}$$

$$\boldsymbol{x}_t = \boldsymbol{C}\boldsymbol{z}_t + \boldsymbol{v}_t, \qquad \boldsymbol{v}_t \sim \mathcal{N}(0, \boldsymbol{\Sigma}). \tag{23.16}$$

Additionally, the prior for $\boldsymbol{z}_1$ is also assumed to be Gaussian

$$p(\boldsymbol{z}_1) \sim \mathcal{N}(0, \boldsymbol{V}_0). \tag{23.17}$$

The goal of the Kalman filter is to predict the latent states of the system given only the previously measured observables, i.e. $\boldsymbol{z}_t|\boldsymbol{x}_{1:t}$. As a side note, there is also a method called Kalman smoothing which infers $\boldsymbol{z}_t|\boldsymbol{x}_{1:T}$ for $t \in 1, \ldots, T$.

A system which has the properties described above is for example a mechanical system where the position and velocity of a tracked object are measured. To see this we can bring the equations of motion in the form of Equation 23.15 by defining

$$\boldsymbol{z} = \begin{bmatrix} \text{pos.}x \\ \text{pos.}y \\ \text{vel.}x \\ \text{vel.}y \\ \text{acc.}x \\ \text{acc.}y \end{bmatrix}; \quad \boldsymbol{A} = \begin{bmatrix} 1 & 0 & \Delta t & 0 & 1/2\Delta t^2 & 0 \\ 0 & 1 & 0 & \Delta t & 0 & 1/2\Delta t^2 \\ 0 & 0 & 1 & 0 & \Delta t & 0 \\ 0 & 0 & 0 & 1 & 0 & \Delta t \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}. \tag{23.18}$$

In particular, a simultaneous position and velocity measurement in x-direction is of the form of Equation 23.16 with the matrix

$$\boldsymbol{C} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}. \tag{23.19}$$

We now derive an expression for $p(\boldsymbol{z}_t|\boldsymbol{x}_{1:t})$:

$$p(\boldsymbol{z}_t|\boldsymbol{x}_{1:t}) = \int p(\boldsymbol{z}_{t-1}, \boldsymbol{z}_t|\boldsymbol{x}_{1:t}) \mathrm{d}\boldsymbol{z}_{t-1} \tag{23.20}$$

$$\propto \int p(\boldsymbol{x}_t|\boldsymbol{z}_{t-1}, \boldsymbol{z}_t, \boldsymbol{x}_{1:t-1}) \cdot p(\boldsymbol{z}_{t-1}, \boldsymbol{z}_t|\boldsymbol{x}_{1:t-1}) \mathrm{d}\boldsymbol{z}_{t-1} \tag{23.21}$$

In the first line we have marginalized over $\boldsymbol{z}_{t-1}$ and in the second line we have used Bayes rule. We can now make use of the structure of the underlying state space model by realizing that $\boldsymbol{x}_t$ is conditionally independent of $\boldsymbol{z}_{t-1}$ and $\boldsymbol{x}_{1:t-1}$ given $\boldsymbol{z}_t$. We then get

$$p(\boldsymbol{z}_t|\boldsymbol{x}_{1:t}) \propto \int p(\boldsymbol{x}_t|\boldsymbol{z}_t) \cdot p(\boldsymbol{z}_{t-1}, \boldsymbol{z}_t|\boldsymbol{x}_{1:t-1}) \mathrm{d}\boldsymbol{z}_{t-1} \tag{23.22}$$

$$= p(\boldsymbol{x}_t|\boldsymbol{z}_t) \int p(\boldsymbol{z}_{t-1}, \boldsymbol{z}_t|\boldsymbol{x}_{1:t-1}) \mathrm{d}\boldsymbol{z}_{t-1} \tag{23.23}$$

$$= p(\boldsymbol{x}_t|\boldsymbol{z}_t) \int p(\boldsymbol{z}_t|\boldsymbol{z}_{t-1}, \boldsymbol{x}_{1:t-1}) \cdot p(\boldsymbol{z}_{t-1}|\boldsymbol{x}_{1:t-1}) \mathrm{d}\boldsymbol{z}_{t-1} \tag{23.24}$$

$$= p(\boldsymbol{x}_t|\boldsymbol{z}_t) \underbrace{\int p(\boldsymbol{z}_t|\boldsymbol{z}_{t-1}) \cdot p(\boldsymbol{z}_{t-1}|\boldsymbol{x}_{1:t-1}) \mathrm{d}\boldsymbol{z}_{t-1}}_{=: \mathcal{N}(\boldsymbol{m}_t, \boldsymbol{P}_t)} \tag{23.25}$$

The final expression consists of the following three conditional probabilities:

- $p(\boldsymbol{x}_t|\boldsymbol{z}_t)$ the measurement
- $p(\boldsymbol{z}_t|\boldsymbol{z}_{t-1})$ the forecast for the latent state without the new measurement

- $p(\boldsymbol{z}_{t-1}|\boldsymbol{x}_{1:t-1})$ the previous best estimate for the latent state of the system

Expressions of the form $p(\boldsymbol{z}_t|\boldsymbol{x}_{1:t})$ appear both on the left and right-hand side of Equation 23.25 for points in time which are one time step apart. The equation thus defines an iterative update rule for $p(\boldsymbol{z}_t|\boldsymbol{x}_{1:t})$.

For the exact evaluation of Gaussian expressions we can use our results from the previous paragraph on Bayesian inference for Gaussian distributions. If we define

$$p(\boldsymbol{z}_t|\boldsymbol{x}_{1:t}) \sim \mathcal{N}(\boldsymbol{\mu}_t, \boldsymbol{V}_t) \tag{23.26}$$

we find

$$\boldsymbol{V}_t = \left(\boldsymbol{C}^T\boldsymbol{\Sigma}^{-1}\boldsymbol{C} + \boldsymbol{P}_t^{-1}\right)^{-1} \tag{23.27}$$

$$\boldsymbol{\mu}_t = \boldsymbol{V}_t\left(\boldsymbol{C}^T\boldsymbol{\Sigma}^{-1}\boldsymbol{C} + \boldsymbol{P}_t^{-1}\boldsymbol{m}_t\right) \tag{23.28}$$

The above expressions still depend on $\boldsymbol{m}_t$ and $\boldsymbol{P}_t$, the evaluation of which requires the computation of the integral in Equation 23.25. After some calculation one finds

$$\boldsymbol{m}_t = \boldsymbol{A}\boldsymbol{\mu}_{t-1}, \tag{23.29}$$

$$\boldsymbol{P}_t = \boldsymbol{\Gamma} + \boldsymbol{A}\boldsymbol{V}_{t-1}\boldsymbol{A}^T. \tag{23.30}$$

An example of Kalman filtering for a mechanical system is shown in Figure 23.3. The first step

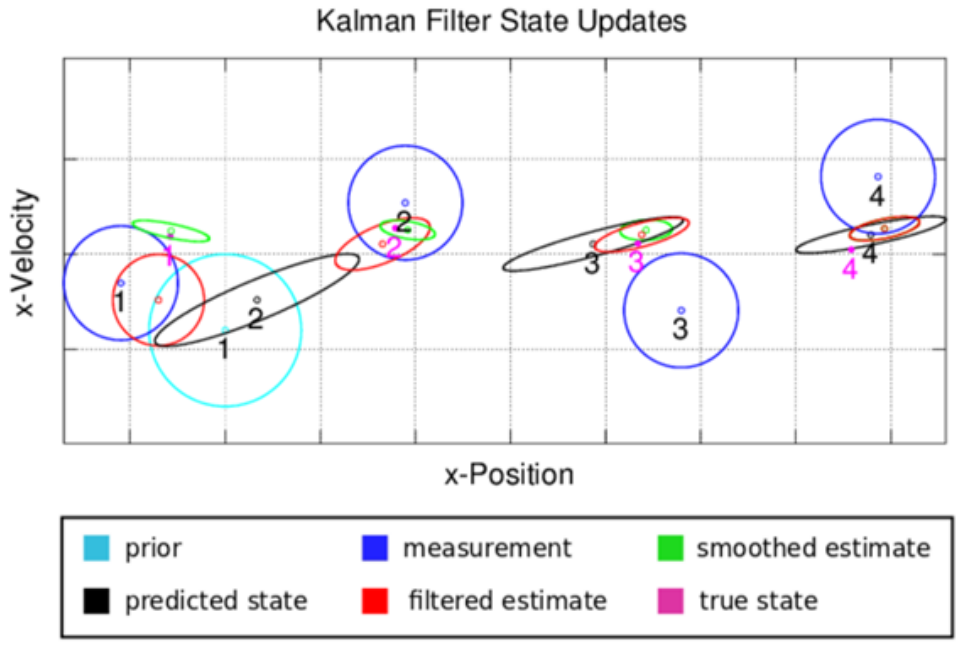


Figure 23.3: Example of the Kalman filter used to track a mechanical object

is to combine the prior $p(\boldsymbol{z}_1)$ with the measurement $p(\boldsymbol{x}_1|\boldsymbol{z}_1)$ to yield the filtered estimate for the first latent state $p(\boldsymbol{z}_1|\boldsymbol{x}_1)$. This result is then propagated according to Equation 23.29 and Equation 23.30 to yield the prediction for $\boldsymbol{z}_2$ without taking into account the new measurement. Note that in this step the former isotropic distribution gets distorted since an x-velocity which is higher than the mean will also result in a greater x-position after a time step Δt and vice versa for x-velocities lower than the mean. The whole process is then iterated using the newly predicted state as prior distribution. The smoothed estimate which is shown in the figure can be obtained similarly by not considering only past measurements but all measurements for the prediction of each latent variable.

24 Reinforcement learning

Another use case of directed graphical models is *reinforcement learning* (RL). A classical example of this is a chess game where we want to train a model to take player actions based on the current board state such that it maximizes its win rate. In formal terms, a reinforcement learning problem consists of the following three parts:

1. The *environment*. It defines the general setting of the problem, involved objects, physical laws/game rules, etc. In the example of the chess game, the environment is the game board, including all figures along with the game rules, e.g., how figures are allowed to move.

2. The *agent*[9]. This term is used for the model we train. The agent can take actions based on the current state of the environment which in turn is influenced by the chosen action. In the chess example, the agent takes the role of one player and during each turn acts by moving one figure according to the game rules.

3. The *reward*. It represents the learning objective of the problem. A reward is attributed to each player action. The agent is trained to maximize the overall reward obtained. The reward is a handcrafted score and often not straightforward to choose, like in the example of the chess game where objectively we can only assign a positive reward for winning the game and a negative one for loosing. The quality of intermediate actions which do not lead to the end of the game is much harder to quantify.

Reinforcement learning, with its learning objective to maximize a certain reward, represents the third machine learning paradigm besides supervised and unsupervised learning. It has many fields of applications ranging from improving the locomotion of robots over game playing with the popular example of Google's AlphaGo to even natural language processing. An example for the latter is ChatGPT which we already discussed in section 20. The training discussed in this chapter is used as an unsupervised pre-training step. Reinforcement learning is subsequently used to fine-tune the model such that it is able to perform its dedicated task, in this case generating relevant responses to user prompts.

One "run" of our model (i.e., one game of chess in our example) can be described by the DGM shown in Figure 24.1. A process which can be described by such a DGM is also called *Markov decision process* (MDP).

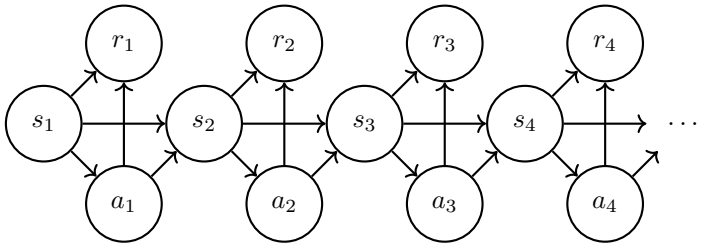


Figure 24.1: The DGM corresponding to a Markov decision process

From the previous chapter we know how to write down the joint probability distribution for the states s_t, actions a_t and rewards r_t:

$$p(s_1, ..., s_4, a_1, ..., a_4, r_1, .., r_3; \boldsymbol{w}) =: p(\tau|\boldsymbol{w}) \tag{24.1}$$

$$= \prod_{t=1}^{T} p(r_t|a_t, s_t) \cdot p(a_t|s_t; \boldsymbol{w}) \cdot p(s_t|s_{t-1}, a_{t-1}) \tag{24.2}$$

where we define $p(s_1|s_0, a_0) = \mu(s)$. The terms $p(r_t|a_t, s_t)$ and $p(s_t|s_{t-1})$ are determined by the environment and are also referred to as *world model*. The term $p(a_t|s_t; \boldsymbol{w})$ is also called the *policy* of the agent and is the function which is actually optimized during training. This expression of the joint allows us to calculate the expected reward of a given policy, which is the objective we want to maximize.

[9]Note that there is also multiagent reinforcement learning which involves several agents. Here we only consider the single agent case.

An important class of MDPs are finite or episodic MDPs (as opposed to infinite MDPs), i.e., processes which do not run forever but have a limited number of steps before they terminate. For those processes it can be optimal to use different policies at different times.

Furthermore, there are so-called partially observable MDPs, where the latent states s_t of the system are not known but are only accessible through observables x_t. These are obtained through some process $p(x_t|s_t)$ similar to what we discussed for state space models in the previous chapter.

24.1 Gradient-free policy optimization

For the training of an RL model we first consider gradient-free policy optimization. Here we assume no prior knowledge of the world model. The basic idea is to try out random policies and iteratively invent new policies based on the previous best. In pseudocode this could be written as follows:

Algorithm 24.1: Gradient-free policy optimization

repeat
  sample $\boldsymbol{w}_i \sim \mathcal{N}(\boldsymbol{\mu}, \sigma^2 \boldsymbol{I})$;
  roll out policy, find rewards $R(\boldsymbol{w}_i)$;
  select top 10% of $\boldsymbol{w}_i$ with highest $R(\boldsymbol{w}_i)$;
  fit $\boldsymbol{\mu} :=$ mean(top 10% of $\boldsymbol{w}_i$);
until *tired*;

This is a very simple but also very expensive implementation. It is mainly useful for problems with low-dimensional $\boldsymbol{w}$.

24.2 Policy gradient methods

A more sophisticated approach which makes use of the fact that we know the structure of the overall joint probability distribution are *policy gradient* methods. Here we are only looking at episodic MDPs.

The basic idea is to use gradient ascent on the policy parameters to increase the expected reward. The latter is calculated by rolling out the current policy multiple times and average the reward over all conducted runs. In this way, we again do not need any knowledge on the world model.

The gradient of the expected reward can be written as

$$\nabla_{\boldsymbol{w}} \mathbb{E}_\tau[R(\tau)] = \nabla_{\boldsymbol{w}} \int p(\tau|\boldsymbol{w}) R(\tau) \mathrm{d}\tau = \int \nabla_{\boldsymbol{w}} p(\tau|\boldsymbol{w}) R(\tau) \mathrm{d}\tau \tag{24.3}$$

$$= \int p(\tau|\boldsymbol{w}) \frac{\nabla_{\boldsymbol{w}} p(\tau|\boldsymbol{w})}{p(\tau|\boldsymbol{w})} R(\tau) \mathrm{d}\tau = \int p(\tau|\boldsymbol{w}) \nabla_{\boldsymbol{w}} \log p(\tau|\boldsymbol{w}) R(\tau) \mathrm{d}\tau \tag{24.4}$$

$$= \mathbb{E}_\tau[R(\tau) \nabla_{\boldsymbol{w}} \log p(\tau|\boldsymbol{w})] \approx \frac{1}{\#\text{roll-outs}} \sum_{i=1}^{\#\text{roll-outs}} R(\tau) \nabla_{\boldsymbol{w}} \log p(\tau^i|\boldsymbol{w}) \tag{24.5}$$

$$= \frac{1}{\#\text{roll-outs}} \sum_{i=1}^{\#\text{roll-outs}} R(\tau) \nabla_{\boldsymbol{w}} \sum_{t=1}^{T} \log p(a_t^i|s_t^i; \boldsymbol{w}) \tag{24.6}$$

In the last step we have used Equation 24.2 and the fact that only the policy term depends on $\boldsymbol{w}$.

The policy term can, e.g., be parametrized by a neural network. The gradients required in Equation 24.6 can then easily be obtained from automatic differentiation. This approach is not biased but has a high variance.

24.3 Proximal policy optimization (PPO)

Proximal policy optimization is one of many approaches how to get less noisy estimates than from the policy gradient method, which is taken from [Sch+17].

The objective used in the policy gradient method is

$$\mathbb{E}_\tau[R(\tau) \nabla_{\boldsymbol{w}} \log p(\tau|\boldsymbol{w})]. \tag{24.7}$$

The authors of the paper approximate this by omitting the logarithm and normalizing $p(\tau|\boldsymbol{w})$ by the current best fit $p(\tau|\boldsymbol{w}^{\text{old}})$

$$\mathbb{E}_\tau\left[R(\tau)\frac{p(\tau|\boldsymbol{w})}{p(\tau|\boldsymbol{w}^{\text{old}})}\right] = \mathbb{E}_\tau\left[R(\tau)\underbrace{\sum_t \frac{p(a_t^\tau|s_t^\tau;\boldsymbol{w})}{p(a_t^\tau|s_t^\tau;\boldsymbol{w}^{\text{old}})}}_{=:r}\right] \tag{24.8}$$

It turns out that using optimization on this objective tends to overshoot. There are again several ways to prevent this, which include:

- Penalize large changes in the policy.
- Clip the ratio r in the objective from Equation 24.8, which is what is used with PPO (see Figure 24.2).

The clipping used always prevents reducing the reward but does not honor overshooting.

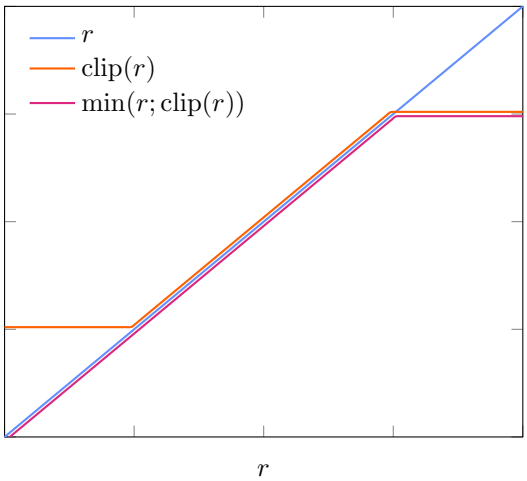


Figure 24.2: Clipped version of the ratio r from the objective in Equation 24.8. The red graph represents the clipping of r used in [Sch+17]

Literature

- John Schulman et al., "Proximal policy optimization algorithms". 2017, [Sch+17]

25 Graph partitioning

Much of the data found in machine learning can be naturally represented as a graph. We introduced the notion of a graph in subsection 2.3. Social networks, for example, can be understood as graphs. Here, the users are the vertices $\mathcal{V}$ and the edges $\mathcal{E}$ can encode whether user $x \in \mathcal{V}$ follows user $y \in \mathcal{V}$.

Images can also be considered as graphs. The nodes are either pixels or regions of pixels also known as *superpixels*, and the (undirected) edges are determined by the adjacency of the nodes in the image. As long as we are not interested in distinguishing different classes, the task of partitioning this graph is one of pure *image partitioning*. In partitioning, we seek to decompose an image into non-overlapping segments whose union constitutes the entire image. Segments are not associated with different types of objects as in instance segmentation (see subsection 18.1), and so one may also call the process *instance segmentation without semantics*. Example 25.1 shows the partitioning of tissue into single cells using microscopic images.

Obtaining a valid partitioning based on conflicting local estimates of connectedness, or lack thereof, of pairs of pixels turns out to be an NP-complete[10] problem [BBC04]. As such, it is an interesting case study of a hard combinatorial optimization problem.

Example 25.1: Connectome

A connectome is a map of the neural connections in a brain. To create such a map at the microscale, we can partition the tissue into single brain cells, as in Figure 25.1. Given a volume of electron microscopic images of a nervous system, we trace each and every cell in the volume. The images are beautiful, the task is difficult though not impossible, and the result promises to give a great boost to the neurosciences.

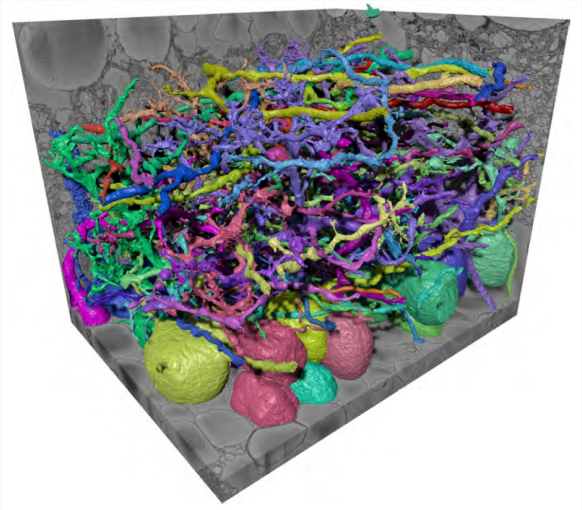


Figure 25.1: Partitioning of brain cells into single neurons. Figure from [And+12].

25.1 Graph partitioning as a multicut problem

Assume that we have obtained, by some means, a set of statements indicating whether a pair of nodes is to end up in the same cluster – we will call this an *attractive interaction* – or whether a pair should end up in different clusters – a *repulsive interaction*. In social media, these interactions can, e.g., be inferred by the number of likes and dislikes. We ignore, for the time being, how exactly these predictions were made.

Now if our mechanism predicting attractive or repulsive interaction were flawless, the inference step would be trivial: we would simply group all nodes with an attractive interaction, and would end up with a set of clusters with the following property: each pair of nodes within a cluster has

[10] Broadly speaking, this means while it is easy to verify the quality of a solution, it can be exceedingly hard to find an optimal solution.

either no or an attractive interaction; and each pair of nodes residing in distinct clusters has either no or a repulsive interaction.

Note that it is trivial to make predictions which are consistent: for instance, if all interactions are repulsive, then each node will prefer to make up a cluster by itself; or if all interactions are attractive, then all nodes prefer to be part of a single happy family. These predictions are consistent but obviously meaningless. The art is to make predictions that are both meaningful (informed by the underlying data) *and* consistent.

If the interactions are not self-consistent, as in Figure 25.2 (b), we will look for the least hurtful compromise: we will look for a configuration that minimizes total frustration. To that end, let us define an energy as follows:

$$\tilde{\varepsilon}(\boldsymbol{c}) = \underbrace{\sum_i \sum_j -w_{ij}^+ \delta(c_i, c_j)}_{\text{reward}} + \underbrace{\sum_i \sum_j w_{ij}^+ (1 - \delta(c_i, c_j))}_{\text{penalty}} + \overbrace{\sum_i \sum_j -w_{ij}^- (1 - \delta(c_i, c_j))}^{} + \overbrace{\sum_i \sum_j w_{ij}^- \delta(c_i, c_j)}^{}. \tag{25.1}$$

The indices i, j run over all vertices $\mathcal{V}$ in our graph. The factors $w_{ij}^{+/-} \in \mathbb{R}^+$ are the positive/negative interaction strengths. The label c_i indicates to which cluster node i belongs, and $\delta(c_i, c_j)$ is the Kronecker delta. In other words, for a given clustering $\boldsymbol{c}$, this function rewards any attractive interaction within a cluster and any repulsive interaction between clusters; and it penalizes all others.

Rewriting the first term $\sum_{i,j} -w_{ij}^+ \delta(c_i, c_j)$ as $-\sum_{i,j} w_{ij}^+ + \sum_{i,j} w_{ij}^+ (1 - \delta(c_i, c_j))$ and proceeding similarly for the last term, we obtain a new expression

$$\tilde{\varepsilon}(\boldsymbol{c}) = 2 \sum_i \sum_j (w_{ij}^+ - w_{ij}^-)(1 - \delta(c_i, c_j)) + \sum_i \sum_j (w_{ij}^- - w_{ij}^+). \tag{25.2}$$

The last term of this expression does not depend on the chosen clustering. Omitting it and dividing by two, we end up with a measure that considers only edges between vertices that are in different clusters: such edges are also said to be *cut*. For each cut edge, it gauges the repulsive vs. the attractive interaction. If the former outweighs the latter, this is rewarded by an energy decrease, and vice versa:

$$\varepsilon(\boldsymbol{c}) = \sum_i \sum_j (w_{ij}^+ - w_{ij}^-)(1 - \delta(c_i, c_j)) \tag{25.3}$$

This is what we will call the *multicut energy*, where lower is better.

Figure 25.2 shows the smallest nontrivial example of a graph, involving three nodes. It will be obvious that the number of consistent partitionings, also known as the *Bell number*, increases quickly with the number of nodes. For three nodes, we have five possible clusterings, all of which are shown in Figure 25.2. For fifteen nodes, that number is already 1382958545, and it keeps growing quickly.

Remember that our goal is to find a clustering that honors, to the extent possible, a given set of attractive and repulsive interactions. In view of how quickly the Bell numbers grow with graph size, a brute force approach that enumerates all possible partitionings and evaluates their energy is doomed to fail. We need something better, described next.

25.2 Integer linear programs (ILP)

In the previous section, we used an intuitive formulation of the energy, which however relied on an auxiliary variable function $c\colon \mathcal{V} \mapsto \{1, \ldots, |\mathcal{V}|\}$ mapping nodes to clusters. The resulting energy is invariant under permutations: calling two distinct clusters 1 and 2 or 2 and 1 amounts to the same thing. In response, we will try to find a formulation that does not exhibit such permutation invariance. Specifically, we would like to use a set of binary indicator variables, one per edge, that indicates whether an edge $e \in \mathcal{E}$ is *cut* ($z_e = 1 - \delta(c_i, c_{j \neq i}) = 1$) or not ($z_e = 0$). In the former case, the nodes incident to e end up in separate clusters, in the latter case they become part of the same cluster.

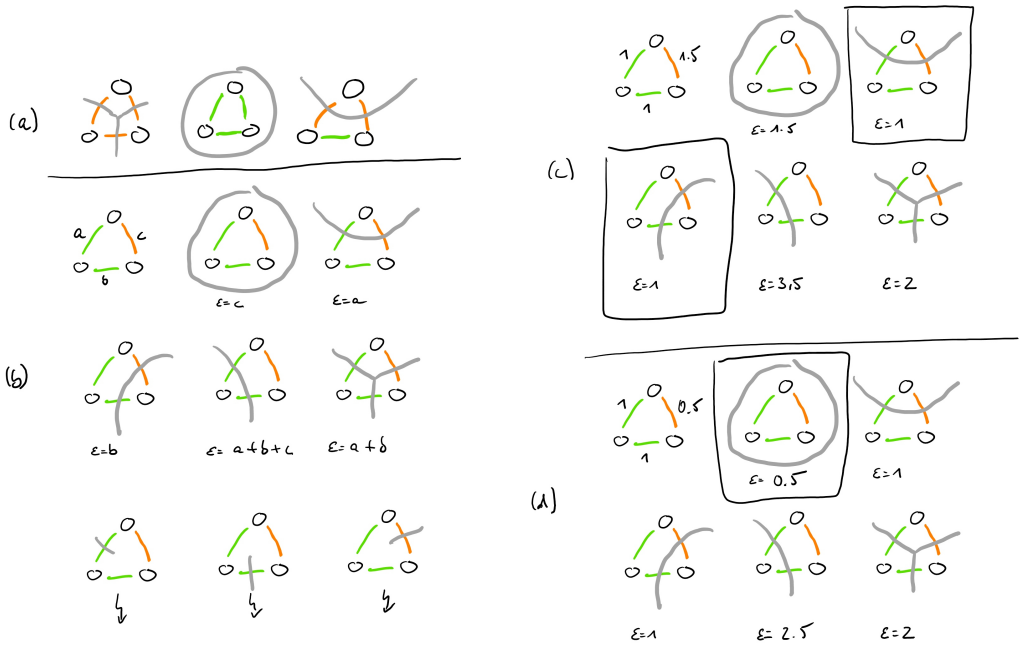


Figure 25.2: (a) Sample graphs with three nodes. Attractive and repulsive interactions are color coded (green and orange, respectively) and, in these examples, consistent. The optimal partitioning induced by these interactions is indicated by the gray lines. (b) A graph with attractive interactions of strength a and b and a repulsive interaction of strength c. Shown are the energies ε of all consistent partitionings, along with three cuts that do not amount to consistent partitionings. (c, d) As above, but with concrete values. The minimal energy configurations are highlighted by a box. The optimum configuration depends on the relative strength of the interactions.

From the illicit configurations in Figure 25.2 (b) we can deduce a set of constraints that a valid partitioning must fulfill – namely, for each cycle, and for each pivot edge e on that cycle, the sum of indicators of all the other edges must be no less than the indicator at edge e. These are also known as the *cycle inequalities*, defining the set

$$\mathcal{C} = \left\{ \boldsymbol{z} \in \mathbb{R}^{|\mathcal{E}|} \;\middle|\; z_e \leq \sum_{f \in C \setminus e} z_f \quad \forall e \in C,\, \forall C \in \{\text{simple cycles in } G\} \right\}. \tag{25.4}$$

For the small graph from Figure 25.2, we can write down the cycle inequalities explicitly,

$$\begin{aligned} z_{(1,2)} &\leq z_{(2,3)} + z_{(1,3)}, \\ z_{(2,3)} &\leq z_{(1,2)} + z_{(1,3)}, \\ z_{(1,3)} &\leq z_{(1,2)} + z_{(2,3)}. \end{aligned} \tag{25.5}$$

If the conditions in Equation 25.4 were equalities, then each one would specify a plane in Hesse normal form. To see this, pick a particular simple cycle C and note that $z_e = \sum_{f \in C \setminus e} z_f$ can be rewritten $\boldsymbol{z}^\top \boldsymbol{a}_{C,e} = 0$ where $\boldsymbol{a}_{C,e} \in \{-1, 0, +1\}^{|\mathcal{E}|}$ is a vector that has a -1 entry associated with edge e, a $+1$ entry for all other edges that are part of simple cycle C, and 0 elsewhere. As the expressions are in fact inequalities and not equalities, each one of them rules out a half-space. The intersection of the viable half-spaces yields the *feasible region* in which, if non-empty, a solution must lie.

The feasible region of the cycle inequalities is unbounded, so we additionally impose $\{z_e \leq 1 \mid e \in E\}$ to obtain indicator variables in the unit cube. Taken together, these equations define a convex *polytope*, shown in Figure 25.3.

Our toy graph had three edges, so we have 2^3 possible integer values for the indicators; but there are only five valid partitionings (see Figure 25.2 (b)) and it is reassuring that our polytope does have corners precisely at the five integer coordinates that correspond to these valid partitionings.

While the polytope for our small toy graph can easily be visualized in three dimensions, it is also unfortunately misleading: indeed, our toy graph, the triangle, is a so-called *series-parallel*

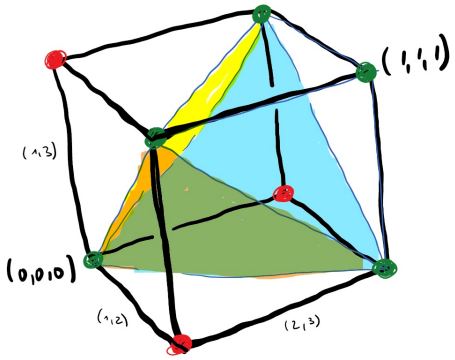


Figure 25.3: Visualization of all cycle inequalities for the graph from Figure 25.2. There are five feasible integral vertices (marked by green dots) corresponding to the five valid partitionings from Figure 25.2.

graph[11], a family of graphs whose modular structure affords efficient computation also on certain problems, including the multicut, that are otherwise difficult. Indeed, the polytope defined by $\mathcal{C} \cap \{\boldsymbol{z} \leq 1\}$ happens to have only integral corners, and this holds true for all series-parallel graphs [Cho94]. Unfortunately, the graphs arising from many applications typically are not so benign, and their cycle inequality polytopes will have many fractional vertices that do not correspond to valid partitionings: see Figure 25.4 for a sketch.

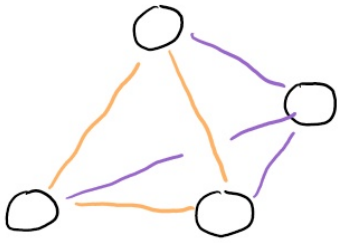


Figure 25.4: A tetrahedron is the smallest graph that is not series-parallel. Consider an edge labeling in which the orange edges are cut (labeled with $z = 1$) and purple edges are neither cut nor uncut ($z = 1/2$). This edge labeling satisfies all cycle inequalities and indeed corresponds to a corner of the polytope $\mathcal{C} \cap \{\boldsymbol{z} \leq 1\}$; yet it does not correspond to a valid graph partitioning.

As a consequence, we need to impose integrality constraints. Overall, the optimization problem becomes

$$\begin{aligned} \min_{\boldsymbol{z}} \ & \boldsymbol{w}^\top \boldsymbol{z} \\ \text{s.t. } & \boldsymbol{z} \in \mathcal{C} \\ & \boldsymbol{z} \in \{0,1\}^{|\mathcal{E}|} \end{aligned} \tag{25.6}$$

The above is one instance of an *integer linear program (ILP)*. It is difficult to overstate the importance of this class of optimization problems. Indeed, most discrete optimization problems can be cast as an ILP, and in consequence there have been massive efforts to try to find ways of solving these efficiently. We must not set our hopes too high, for the very reason that NP-hard problems such as the multicut can be written as an ILP. So we must expect to encounter even modestly-sized ILP instances that prove exceedingly hard to solve. Still, highly optimized software relying on some form of branch-and-bound has been developed and can often find exact optima for real-world

[11] Geometrically speaking, a graph is series-parallel if it can be obtained from a single edge by sequentially replacing lines with serial or parallel connections as in electrical circuits.

problems. Such software expects the ILP to be cast in a canonical form, such as

$$\min_{\boldsymbol{z}} \boldsymbol{w}^\top \boldsymbol{z} \qquad \text{(semi-)convex in } \boldsymbol{z} \tag{25.7}$$

$$\text{s.t. } \boldsymbol{A}\boldsymbol{z} \leq b \qquad \text{convex in } \boldsymbol{z} \tag{25.8}$$

$$\boldsymbol{z} \geq 0 \qquad \text{convex in } \boldsymbol{z} \tag{25.9}$$

$$\boldsymbol{z} \in \mathbb{Z}^n \qquad \text{non-convex in } \boldsymbol{z} \tag{25.10}$$

You can easily verify that Equation 25.7 is a special case.

Because ILPs are so important a class of optimization problems, let's study their polyhedral geometry in terms of a concrete example. Figure 25.5 shows an integer linear program defined over two variables, call them z_1 and z_2. A weight vector $\boldsymbol{w}$, illustrated by a blue arrow, indicates how the cost $\boldsymbol{w}^\top \boldsymbol{z}$ depends on $\boldsymbol{z}$. The blue shading illustrates the cost associated with each configuration $\boldsymbol{z}$. However, we have three additional linear inequality constraints, as embodied by Equation 25.8. Each of them rules out one half-space. Each inequality is indicated by a black line delimiting a hatched half-space that is ruled out. At this point, the triangle formed by the half-spaces is the feasible region, and the solution with minimal cost is indicated by a turquoise star. This is the solution of the linear programming relaxation. However, this tentative solution is not integral, in violation of Equation 25.10. The latter dictates that only points from the Cartesian lattice be eligible. Integral solutions that are compatible with the integrality constraints are indicated by green dots, all others are marked in red. We now are left with only five feasible integral solutions, and the one with the smallest cost amongst them is marked by a green star: it is the optimal solution to our integer linear program.

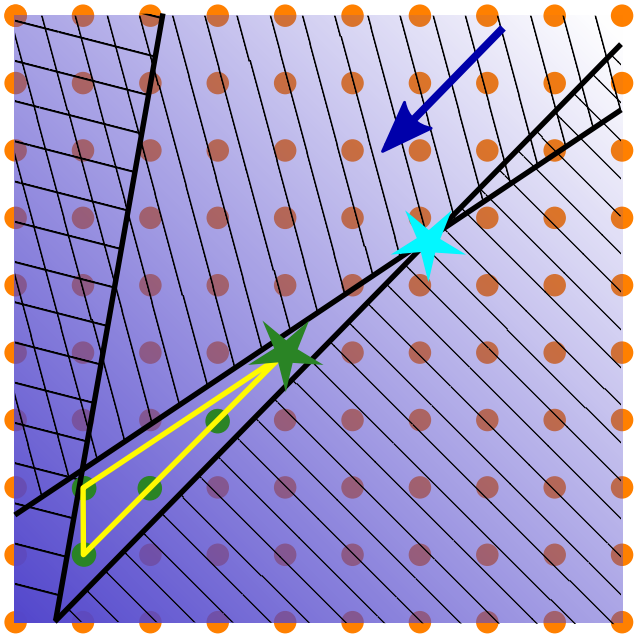


Figure 25.5: Example ILP problem over two variables. The blue arrow indicates the direction of increasing cost, and blue shading indicates the cost associated with each solution. Three constraints rule out a half-plane each (indicated by hatched areas). Taken together, they define a polyhedron. Amongst all integer solutions, only the ones inside the polyhedron are feasible (indicated by green circles). The LP solution, ignoring integrality constraints, is shown by a turquoise star. The best feasible integral solution is indicated by a green star; it cannot be obtained by mere rounding of the LP solution. Finally, the convex hull of all feasible integral solutions is shown in yellow.

Note that finding that best integral solution can be extremely hard. This is in spite of the objective having the simplest conceivable structure (the cost is a mere linear function of the variables)! Mathematically speaking, the objective is a (semi-) convex function. Optimizing convex functions over a convex set of constraints such as Equation 25.8, Equation 25.9 is easy [BBV04]. It is the innocent-looking Equation 25.10 that is the culprit. The set of integers is non-convex, and searching over it is what makes the problem combinatorial. Note, in particular, that simple

rounding strategies do not work well in general. In our example, if we try to round the solution of the linear programming relaxation (the turquoise star) to any of the closest integer solutions, we find that all of them are infeasible! In this example, the optimal ILP solution is only the 17th (or so) nearest integral neighbor of the relaxed solution. This kind of problem is exacerbated in higher dimensions. Finally, note that the optimal ILP solution is the corner of a more compact polyhedron, shown in yellow: the convex hull of all feasible integral solutions. Optimizing over this polyhedron would be easy, if only we knew it. Alas, finding the correct convex hull of feasible integral solutions is itself a hard combinatorial problem. For certain ILPs, a subset of the faces of the convex hull may be easier to find. These are the coveted *facet-defining* constraints.

Modularity clustering

One way to define the interactions if there are only attractive interactions in a graph is with *modularity*. It was introduced by M. E. J. Newman and is defined by

$$w_{ij} = a_{ij} - \frac{\text{degree}_i - \text{degree}_j}{2|\mathcal{E}|} \qquad \forall i, j \in \mathcal{V}, \tag{25.11}$$

where a_{ij} is the adjacency and the degree of a node is the number of connecting edges. In this expression, we subtract the expected connectivity $(\text{degree}_i - \text{degree}_j)/(2|\mathcal{E}|)$ from the actual connectivity a_{ij}. Modularity is an instance of multicut partitioning.

Approximations

In order to deal with big graphs, we can find a solution of the linear program without the cycle inequalities. Afterward, we look for violations of the inequalities, add these as constraints and solve the newly obtained program. For huge problems, one can use one of the following approximations, which we will only mention here:

- Kernighan-Lin,
- Louvain clustering,
- Leiden clustering.

Literature

- Bernhard H Korte and Jens Vygen, *Combinatorial optimization*. 2011, [KV11]
- M. E. J. Newman and Tiago P. Peixoto, "Generalized Communities in Networks". 2015, [NP15]

26 Optimal transport

Let's say we want to predict the spectra of molecules. We could use a transformer-encoder to make predictions for molecules that are given in the SMILES representation (see also Example 21.1). To train the model, we need to compare different predictions to the ground truth, as shown in

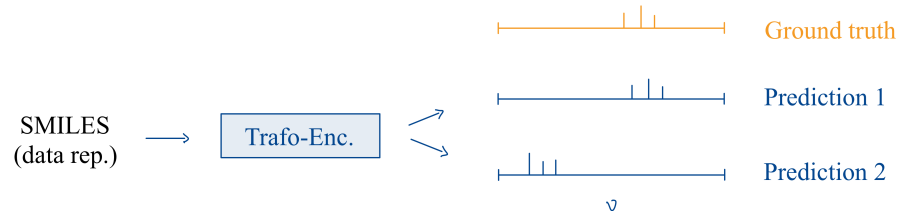


Figure 26.1: Two different predictions of molecular spectra from a transformer-encoder are compared to the ground truth.

Figure 26.1. Simply using the least squares method would only evaluate the quadratic difference for each frequency ν. Thus, both predictions shown would be equally good, as none matches the frequencies exactly. But prediction 1 is intuitively much better than prediction 2.

To get a meaningful statement about the agreement, we could in a first attempt smooth the spectra and calculate a norm like ℓ_2 afterward. One downside of the smoothing is that we have to decide which kernel and kernel parameters we want to use. Instead, we will use the *earth mover's distance* (EMD), also known as *Wasserstein-1-metric*. Illustratively, it transforms one distribution into the other and penalizes the movement of (probability) mass scaled with the distance of the movement.

26.1 Discrete optimal transport

We start with rephrasing the problem in the language of discrete optimal transport (discrete OT). Given n *sources* a_i and m *sinks* b_j, we want to evaluate

$$\min_{\boldsymbol{P}} \sum_{i=1}^{n} \sum_{j=1}^{m} P_{ij} C_{ij} \tag{26.1}$$

by finding the optimal *transportation plan* $\boldsymbol{P} \in \mathbb{R}^{n \times m}$ under the constraints that

$$a_i = \sum_{j=1}^{m} P_{ij} \quad \text{“capacity”}, \tag{26.2}$$

$$b_j = \sum_{i=1}^{n} P_{ij} \quad \text{“demand”}, \tag{26.3}$$

$$P_{ij} \geq 0. \tag{26.4}$$

We can understand the element P_{ij} of the transport plan as the mass shipped from source i to sink j (see Figure 26.2). The factor C_{ij} is the cost of shipping one unit of mass from i to j. The constraints ensure that the capacity of the sources is fully utilized (Equation 26.2), that the demand of the sinks is covered (Equation 26.3) and that no “antimatter” is transported (Equation 26.4). These constraints can only be fulfilled if $\sum_i a_i = \sum_j b_j$.

This optimization problem can be formulated as a linear program. We define the scalar product of two matrices $\boldsymbol{A}, \boldsymbol{B} \in \mathbb{R}^{n \times m}$ as

$$\langle \boldsymbol{A}, \boldsymbol{B} \rangle := \operatorname{tr}(\boldsymbol{A}\boldsymbol{B}^\top) = \sum_{i,j} A_{ij} B_{ij} \tag{26.5}$$

which is equivalent to the normal inner product after a vectorization of the matrices. The constraints

$$\underset{n \times m}{\boldsymbol{P}}\, \mathbf{1}_m = \boldsymbol{a} \quad \text{and} \quad \underset{m \times n}{\boldsymbol{P}^\top} \mathbf{1}_n = \boldsymbol{b} \tag{26.6}$$

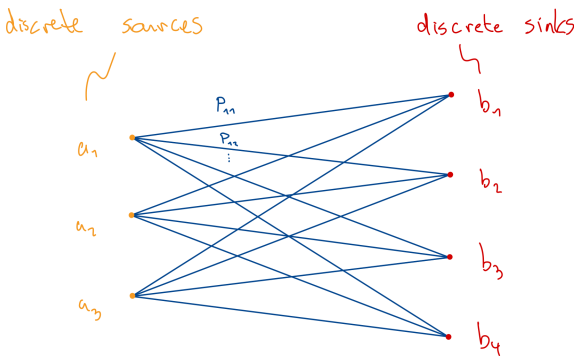


Figure 26.2: Graph of discrete OT.

can be rewritten as two inequalities, for instance

$$\boldsymbol{P} \cdot \mathbf{1}_m = \boldsymbol{a} \iff \begin{Bmatrix} \boldsymbol{P} \cdot \mathbf{1}_m \geq \boldsymbol{a} \\ -\boldsymbol{P} \cdot \mathbf{1}_m \geq -\boldsymbol{a} \end{Bmatrix} \iff \begin{bmatrix} \boldsymbol{P} \\ -\boldsymbol{P} \end{bmatrix} \mathbf{1}_m \geq \begin{bmatrix} \boldsymbol{a} \\ -\boldsymbol{a} \end{bmatrix}. \tag{26.7}$$

Together, we obtain the linear program

$$\begin{aligned} \min_{\boldsymbol{P}} \; & \langle \boldsymbol{P}, \boldsymbol{C} \rangle \\ \text{s.t.} \; & \begin{bmatrix} \boldsymbol{P} \\ -\boldsymbol{P} \end{bmatrix} \mathbf{1}_m \geq \begin{bmatrix} \boldsymbol{a} \\ -\boldsymbol{a} \end{bmatrix} \\ & \begin{bmatrix} \boldsymbol{P}^\top \\ -\boldsymbol{P}^\top \end{bmatrix} \mathbf{1}_n \geq \begin{bmatrix} \boldsymbol{b} \\ -\boldsymbol{b} \end{bmatrix} \\ & \boldsymbol{P} \geq 0. \end{aligned} \tag{26.8}$$

The constraints on $\boldsymbol{P}$ define a region in $\mathbb{R}^{n \times m} \simeq \mathbb{R}^{n \cdot m}$ which is referred to as *transport polytope*. In the special case of $\boldsymbol{a} = \boldsymbol{b} = \mathbf{1}_n$, the feasible region is also known as the *Birkhoff polytope*.

If we want to apply the discrete OT as a loss in the above scenario, we can define the sources a_i to be the prediction and the sinks b_j to be the ground truth. The loss is then given by the minimum in Equation 26.8. As the parameters $\boldsymbol{w}$ of the encoder change, the feasible polytope of the optimization problem changes as well. Thus, the optimal transport plan and thereby the optimal parameters $\boldsymbol{w}$ of the encoder can transition suddenly.

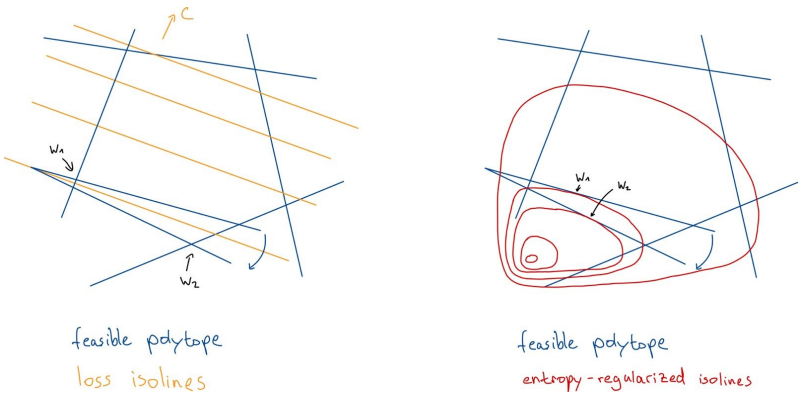


Figure 26.3: Feasible polytope and cost isolines in the standard and regularized case.

Such a case is visualized in Figure 26.3 on the left. It shows two feasible polytopes that correspond to the parameters $\boldsymbol{w}_1$ and $\boldsymbol{w}_2$. The minimal loss changes in a non-differentiable way if we transition the parameters from $\boldsymbol{w}_1$ to $\boldsymbol{w}_2$. This is of course problematic if we want to apply OT as a loss function and train a model with gradient descent.

To counteract this behavior, we regularize the objective with the aid of the entropy. Using the regularization strength $\gamma > 0$, we change our objective to

$$\min_{\boldsymbol{P}} \langle \boldsymbol{P}, \boldsymbol{C} \rangle + \gamma \sum_{i,j} P_{ij} \left(\log P_{ij} - 1 \right). \tag{26.9}$$

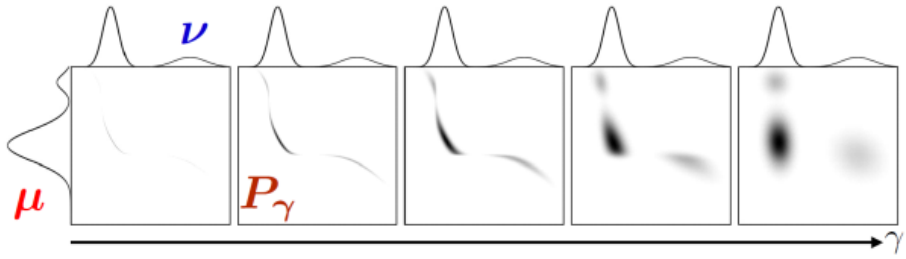


Figure 26.4: Optimal transport plan $\boldsymbol{P}_\gamma$ for the sources $\boldsymbol{\mu}$ and the sinks $\boldsymbol{\nu}$ for different regularization strengths γ. For $\gamma \to \infty$, the optimal transport plan converges to the independent joint distribution. Figure from [Cut13].

The subtraction of 1 inside the entropy is an irrelevant constant offset that simplifies subsequent expressions. The isolines of the entropy-regularized case are shown in Figure 26.3 on the right. The term $P_{ij} \log P_{ij}$ has a slope of negative infinity at $P_{ij} = 0$, so the entropy-regularization encourages non-sparse solutions (see also Figure 26.4). Our new optimization goal is a differentiable function of the parameters $\boldsymbol{w}$. This makes it a useful tool for machine learning. Below, we describe a method to quickly calculate the optimal transport plan.

26.2 Sinkhorn iterations

To obtain the optimal transport of the entropy-regularized case, we define a Lagrange function to combine the objective (Equation 26.9) with the constraints (Equation 26.6),

$$\mathcal{L}(\boldsymbol{P}, \boldsymbol{\alpha}, \boldsymbol{\beta}) = \sum_{i,j} (P_{ij} C_{ij} + \gamma P_{ij} (\log P_{ij} - 1)) + \boldsymbol{\alpha}^\top (\boldsymbol{P}\mathbf{1}_m - \boldsymbol{a}) + \boldsymbol{\beta}^\top (\boldsymbol{P}\mathbf{1}_m - \boldsymbol{b}), \tag{26.10}$$

where we introduced the Lagrange multipliers $\boldsymbol{\alpha} \in \mathbb{R}^n$ and $\boldsymbol{\beta} \in \mathbb{R}^m$. At a minimum, the derivative of the Lagrange function with respect to the independent variables P_{ij} vanishes,

$$\begin{aligned} 0 \stackrel{!}{=} \frac{\partial \mathcal{L}}{\partial P_{ij}} &= C_{ij} + \gamma(\log P_{ij} - 1) + \gamma P_{ij} \frac{1}{P_{ij}} + \alpha_i + \beta_j \\ &= C_{ij} + \gamma \log P_{ij} + \alpha_i + \beta_j. \end{aligned} \tag{26.11}$$

We can rewrite this as

$$\log P_{ij} = \frac{1}{\gamma} (-\alpha_i - C_{ij} - \beta_j) \tag{26.12}$$

$$\implies P_{ij} = \underbrace{\exp\left(-\frac{\alpha_i}{\gamma}\right)}_{=:u_i} \cdot \underbrace{\exp\left(-\frac{C_{ij}}{\gamma}\right)}_{=:K_{ij}} \cdot \underbrace{\exp\left(-\frac{\beta_j}{\gamma}\right)}_{=:v_j} \tag{26.13}$$

$$\implies \boldsymbol{P} = \mathrm{diag}(\boldsymbol{u}) \cdot \boldsymbol{K} \cdot \mathrm{diag}(\boldsymbol{v}). \tag{26.14}$$

Using the constraints

$$\begin{aligned} \boldsymbol{P}\mathbf{1}_m \stackrel{!}{=} \boldsymbol{a} \quad &\implies \quad \boldsymbol{a} = \boldsymbol{u} \odot \boldsymbol{K}\boldsymbol{v} \\ \boldsymbol{P}^\top \mathbf{1}_m \stackrel{!}{=} \boldsymbol{b} \quad &\implies \quad \boldsymbol{b} = \boldsymbol{v} \odot \boldsymbol{K}^\top \boldsymbol{u}, \end{aligned} \tag{26.15}$$

we derive the *iterated Sinkhorn equations*

$$\begin{aligned} \boldsymbol{u} &= \boldsymbol{a} \oslash \boldsymbol{K}\boldsymbol{v} \\ \boldsymbol{v} &= \boldsymbol{b} \oslash \boldsymbol{K}^\top \boldsymbol{u}, \end{aligned} \tag{26.16}$$

where $\odot$ and $\oslash$ are element-wise multiplication and division. We can use Equation 26.16 to iteratively find a fixed point. As shown in Figure 26.5, this can be calculated efficiently on a GPU.

This method provides a superfast and differentiable means to compute the optimal discrete regularized transport. These two characteristics make it a useful new tool in the machine learning toolbox.

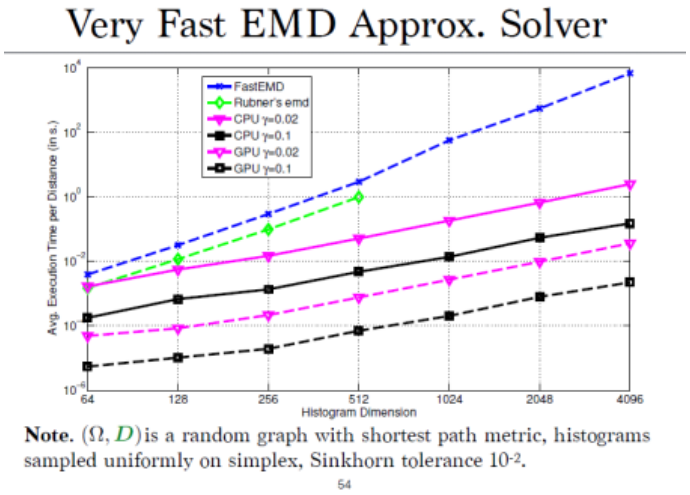


Figure 26.5: Computation time of optimal transport problems with different algorithms and parameters. Note the log-scale of both axes. Figure from [Cut13].

26.3 Extensions

If we have found an optimal transport plan, we can use it to interpolate between two distributions. The transport plan defines how much mass should be transported from a source in the start distribution to a sink in the end distribution of the interpolation. This mass P_{ij} can then be moved from site i to site j. Figure 26.6 compares linear interpolation to an interpolation powered by OT.

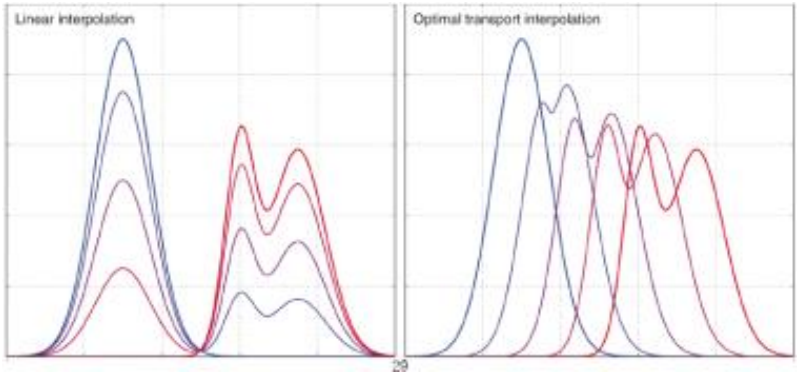


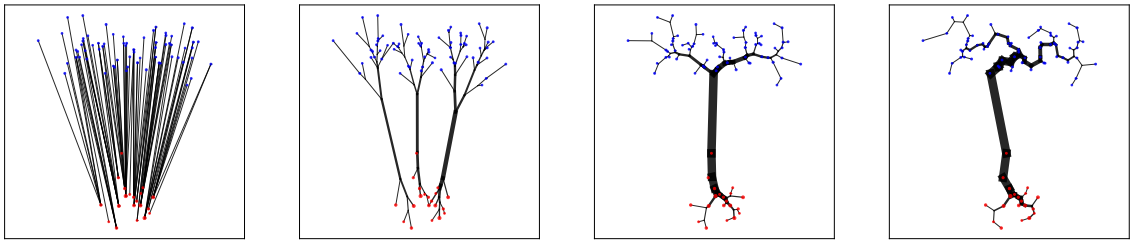
Figure 26.6: The interpolation that uses optimal transport (right) transitions the distributions in a more natural way than the linear interpolation (left). Figure from [Cut].

We can also extend OT to include that transporting a lot of mass along the same path can be cheaper than transporting it in several batches. For that, we introduce a parameter $\alpha \in [0, 1]$ and make use of the inequality

$$(x + y)^\alpha \leq x^\alpha + y^\alpha \qquad x, y \in [0, \infty). \tag{26.17}$$

We replace the minimization objective $\sum_{i,j} P_{ij} C_{ij}$ with $\sum_{i,j} P_{ij}^\alpha C_{ij}$ to provide an incentive to transport mass "together". We also need to introduce *branch points* $p \in \mathcal{B}$ where transports of mass can merge or split. Here, the sources, sinks and branch points are points in $\mathbb{R}^d$ and the cost C_{ij} is given by the Euclidean distance of the points i and j.

The problem can adequately be described by a graph. The nodes consist out of the sources, sinks and branch points $\mathcal{B}$. We need to optimize $\mathcal{B}$, the set of edges $\mathcal{E}$, the coordinates of the



$\alpha = 1$
Optimal Transport

$\alpha = 0.95$

$\alpha = 0.5$

$\alpha = 0$
Euclidean Steiner Tree

Figure 26.7: Branched optimal transport for different values of α on a toy example of sources (red) and sinks (blue). The width of an edge indicates the mass transported along the edge. Figure from [LFH22].

branch points $\boldsymbol{x}_{\mathcal{B}}$ and of course the transport plan $\boldsymbol{P}$ to minimize the objective

$$\begin{aligned}
&\underset{\mathcal{B}, \mathcal{E}, \boldsymbol{x}_{\mathcal{B}}, \boldsymbol{P}}{\arg\min} \sum_{(i,j) \in \mathcal{E}} P_{ij}^{\alpha} \left\| \boldsymbol{x}_i - \boldsymbol{x}_j \right\|_2, \text{ subject to} \qquad (26.18)\\
&\text{supply} \quad a_i = \sum\nolimits_k P_{ik} - \sum\nolimits_k P_{ki} \quad \text{at each source } i,\\
&\text{demand} \quad b_j = \sum\nolimits_k P_{kj} - \sum\nolimits_k P_{jk} \quad \text{at each sink } j,\\
&\text{conservation} \quad \sum\nolimits_k P_{kp} = \sum\nolimits_k P_{pk} \quad \text{at each branch point } p \in \mathcal{B}.
\end{aligned}$$

The sources and sinks are no longer ends of the transportation but can be an intermediate stop. Examples of branched OT for different α are shown in Figure 26.7. The smaller the value of α, the bigger the incentive to transport mass together. The case $\alpha = 0$ is also known as a Steiner tree.

References

[And+12] Bjoern Andres et al. "Globally Optimal Closed-Surface Segmentation for Connectomics". In: *Computer Vision – ECCV 2012*. Ed. by Andrew Fitzgibbon et al. Springer Berlin Heidelberg, 2012, pp. 778–791. ISBN: 978-3-642-33712-3.

[BBC04] Nikhil Bansal, Avrim Blum, and Shuchi Chawla. "Correlation clustering". In: *Machine learning* 56 (2004), pp. 89–113.

[BBK22] Jan Niklas Böhm, Philipp Berens, and Dmitry Kobak. "Attraction-repulsion spectrum in neighbor embeddings". In: *Journal of Machine Learning Research* 23.95 (2022), pp. 1–32.

[BBV04] Stephen Boyd, Stephen P Boyd, and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[BN03] Mikhail Belkin and Partha Niyogi. "Laplacian Eigenmaps for Dimensionality Reduction and Data Representation". In: *Neural Computation* 15.6 (2003), pp. 1373–1396. DOI: 10.1162/089976603321780317.

[BN06] Christopher M Bishop and Nasser M Nasrabadi. *Pattern recognition and machine learning*. Vol. 4. 4. Springer, 2006.

[Can+19] Askery Canabarro et al. "Unveiling phase transitions with machine learning". In: *Physical Review B* 100.4 (2019), p. 045129.

[Cao+16] Zhe Cao et al. *Realtime Multi-Person 2D Pose Estimation using Part Affinity Fields*. 2016. DOI: 10.48550/ARXIV.1611.08050. URL: https://arxiv.org/abs/1611.08050.

[car12] (https://stats.stackexchange.com/users/2970/cardinal) cardinal. *Is it possible to have a pair of Gaussian random variables for which the joint distribution is not Gaussian?* Cross Validated. URL:https://stats.stackexchange.com/q/30205 (version: 2019-12-02). 2012. eprint: https://stats.stackexchange.com/q/30205. URL: https://stats.stackexchange.com/q/30205.

[Che+20] Ming Chen et al. "Simple and Deep Graph Convolutional Networks". In: *Proceedings of the 37th International Conference on Machine Learning*. Ed. by Hal Daumé III and Aarti Singh. Vol. 119. Proceedings of Machine Learning Research. PMLR, 2020, pp. 1725–1735. URL: https://proceedings.mlr.press/v119/chen20v.html.

[Che95] Yizong Cheng. "Mean shift, mode seeking, and clustering". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 17.8 (1995), pp. 790–799. DOI: 10.1109/34.400568.

[Cho94] Sunil Chopra. "The graph partitioning polytope on series-parallel and 4-wheel free graphs". In: *SIAM journal on discrete mathematics* 7.1 (1994), pp. 16–31.

[col20] LHCb collaboration. *Simulated jet samples for quark flavour identification studies*. 2020. DOI: 10.7483/OPENDATA.LHCB.N75T.TJPE.

[Cor+16] Marius Cordts et al. "The Cityscapes Dataset for Semantic Urban Scene Understanding". In: *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016.

[Cov65] Thomas M. Cover. "Geometrical and Statistical Properties of Systems of Linear Inequalities with Applications in Pattern Recognition". In: *IEEE Trans. Electron. Comput.* 14 (1965), pp. 326–334.

[CSB13] J.H. Conway, N.J.A. Sloane, and Bannai. *Sphere Packings, Lattices and Groups*. Grundlehren der mathematischen Wissenschaften. Springer New York, 2013. ISBN: 9781475722499. URL: https://books.google.de/books?id=hoTjBwAAQBAJ.

[Cut] Marco Cuturi. *Computational Optimal Transport*. URL: https://optimaltransport.github.io/resources/ (visited on 04/10/2023).

[Cut13] Marco Cuturi. "Sinkhorn distances: Lightspeed computation of optimal transport". In: *Advances in neural information processing systems* 26 (2013).

[Dab19] Fabian Dablander. *Two properties of the Gaussian distribution*. 2019. URL: https://fabiandablander.com/statistics/Two-Properties.html (visited on 03/30/2023).

[Dam+22] Sebastian Damrich et al. "Contrastive learning unifies t-SNE and UMAP". In: *arXiv preprint arXiv:2206.01816* (2022).

[dev] scikit-learn developers. *Linear and Quadratic Discriminant Analysis*. URL: https://scikit-learn.org/stable/modules/lda_qda.html (visited on 04/06/2023).

[DGL13] Luc Devroye, László Györfi, and Gábor Lugosi. *A probabilistic theory of pattern recognition*. Vol. 31. Springer Science & Business Media, 2013.

[DH97] Pierre Demartines and Jeanny Hérault. "Curvilinear component analysis: A self-organizing neural network for nonlinear mapping of data sets". In: *IEEE Transactions on neural networks* 8.1 (1997), pp. 148–154.

[DHS00] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern Classification*. Second. Nov. 2000, p. 688. ISBN: 978-0-471-05669-0.

[Dra+18] Felix Draxler et al. "Essentially No Barriers in Neural Network Energy Landscape". In: (2018). DOI: 10.48550/ARXIV.1803.00885. URL: https://arxiv.org/abs/1803.00885.

[DZ17] Carl Doersch and Andrew Zisserman. "Multi-task Self-Supervised Visual Learning". In: *2017 IEEE International Conference on Computer Vision (ICCV)*. 2017, pp. 2070–2079. DOI: 10.1109/ICCV.2017.226.

[Gar+18] Timur Garipov et al. "Loss surfaces, mode connectivity, and fast ensembling of dnns". In: *Advances in neural information processing systems* 31 (2018).

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[glS20] glS. *What do the level sets of the Shannon entropy look like?* 2020. URL: https://math.stackexchange.com/questions/3791203/what-do-the-level-sets-of-the-shannon-entropy-look-like (visited on 03/27/2023).

[GOV22] Léo Grinsztajn, Edouard Oyallon, and Gaël Varoquaux. "Why do tree-based models still outperform deep learning on typical tabular data?" In: *Advances in Neural Information Processing Systems* 35 (2022), pp. 507–520.

[GP17] Bolin Gao and Lacra Pavel. "On the properties of the softmax function with application in game theory and reinforcement learning". In: *arXiv preprint arXiv:1704.00805* (2017).

[HA03] Fred A. Hamprecht and Erik Agrell. "Exploring a space of materials: Spatial sampling design and subset selection". In: *Experimental Design for Combinatorial and High Throughput Materials Development*. Ed. by James N. Cawse. New York: Wiley, 2003, pp. 227–307. DOI: https://doi.org/10.1002/anie.200385086.

[Ham20] William L Hamilton. *Graph representation learning*. Morgan & Claypool Publishers, 2020.

[Han09] Bruce E. Hansen. *Lecture Notes on Nonparametrics*. University of Wisconsin, 2009.

[Hau21] Manuel Haussmann. "Bayesian Neural Networks for Probabilistic Machine Learning". PhD thesis. Universität Heidelberg, 2021.

[HD05] Aapo Hyvärinen and Peter Dayan. "Estimation of non-normalized statistical models by score matching." In: *Journal of Machine Learning Research* 6.4 (2005).

[He+15] Kaiming He et al. *Deep Residual Learning for Image Recognition*. 2015. DOI: 10.48550/ARXIV.1512.03385. URL: https://arxiv.org/abs/1512.03385.

[Hfe11] Hferee, Wikimedia Commons. *File:Delaunay Voronoi.svg*. 2011. URL: https://commons.wikimedia.org/w/index.php?title=File:Delaunay_Voronoi.svg&oldid=651563886 (visited on 03/16/2023).

[HJA20] Jonathan Ho, Ajay Jain, and Pieter Abbeel. "Denoising diffusion probabilistic models". In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 6840–6851.

[HR02] Geoffrey E Hinton and Sam Roweis. "Stochastic neighbor embedding". In: *Advances in neural information processing systems* 15 (2002).

[HTF09] Trevor Hastie, Robert Tibshirani, and Jerome H Friedman. "The elements of statistical learning: data mining, inference, and prediction". In: vol. 2. Springer, 2009. Chap. Classification Trees. ISBN: 9780387848570.

[Hua+16] Gao Huang et al. *Densely Connected Convolutional Networks*. 2016. DOI: 10.48550/ARXIV.1608.06993. URL: https://arxiv.org/abs/1608.06993.

[IS15] Sergey Ioffe and Christian Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. 2015. DOI: 10.48550/ARXIV.1502.03167. URL: https://arxiv.org/abs/1502.03167.

[Ise+21] Fabian Isensee et al. "nnU-Net: a self-configuring method for deep learning-based biomedical image segmentation". In: *Nature methods* 18.2 (2021), pp. 203–211.

[J R12a] J. Rodal, Wikimedia Commons. *Differential Entropy Beta Distribution for alpha and beta from 0.1 to 5*. 2012. URL: https://en.wikipedia.org/wiki/File:Differential_Entropy_Beta_Distribution_for_alpha_and_beta_from_0.1_to_5_-_J._Rodal.jpg (visited on 03/31/2023).

[J R12b] J. Rodal, Wikimedia Commons. *Differential Entropy Beta Distribution for alpha and beta from 1 to 5*. 2012. URL: https://en.wikipedia.org/wiki/File:Differential_Entropy_Beta_Distribution_for_alpha_and_beta_from_1_to_5_-_J._Rodal.jpg (visited on 03/31/2023).

[Kaw18] Daniel Kawetzki. "Semantische Segmentierung von urbanen Szenen mittels Deep Learning". M.Sc. thesis. Universität Heidelberg, 2018.

[Kaz19] Amirhossein Kazemnejad. *Transformer Architecture: The Positional Encoding*. 2019. URL: https://kazemnejad.com/blog/transformer_architecture_positional_encoding/.

[KBJ19] Alexander Krull, Tim-Oliver Buchholz, and Florian Jug. "Noise2void-learning denoising from single noisy images". In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2019, pp. 2129–2137.

[Kob20] Dmitry Kobak. twitter thread. 2020. URL: https://twitter.com/hippopedoid/status/1318917878364672001.

[Kra22] Scott Kravitz. *Searching for Dark Matter with Machine Learning*. 2022. URL: https://indico.physics.lbl.gov/event/1968/attachments/3220/4228/Searching_for_dark_matter_with_machine_learning_Phys290E_Feb_2022.pdf (visited on 03/31/2023).

[Kru14] John Kruschke. "Doing Bayesian data analysis: A tutorial with R, JAGS, and Stan". In: (2014).

[Kru64] Joseph B Kruskal. "Nonmetric multidimensional scaling: a numerical method". In: *Psychometrika* 29.2 (1964), pp. 115–129.

[KV11] Bernhard H Korte and Jens Vygen. *Combinatorial optimization*. Vol. 1. Springer, 2011.

[KW16] Thomas N. Kipf and Max Welling. "Semi-supervised classification with graph convolutional networks". In: *arXiv preprint arXiv:1609.02907* (2016). DOI: 10.48550/ARXIV.1609.02907.

[Leh+18] Jaakko Lehtinen et al. "Noise2Noise: Learning Image Restoration without Clean Data". In: *CoRR* abs/1803.04189 (2018). arXiv: 1803.04189. URL: http://arxiv.org/abs/1803.04189.

[LFH22] Peter Lippmann, Enrique Fita Sanmartín, and Fred A Hamprecht. "Theory and Approximate Solvers for Branched Optimal Transport with Multiple Sources". In: *Advances in Neural Information Processing Systems* 35 (2022), pp. 267–279.

[Li+17] Hao Li et al. *Visualizing the Loss Landscape of Neural Nets*. 2017. DOI: 10.48550/ARXIV.1712.09913. URL: https://arxiv.org/abs/1712.09913.

[LP16] Russell Lyons and Yuval Peres. *Probability on Trees and Networks*. Vol. 42. Cambridge Series in Statistical and Probabilistic Mathematics. Available at https://rdlyons.pages.iu.edu/. Cambridge University Press, New York, 2016, pp. xv+699. ISBN: 978-1-107-16015-6. DOI: 10.1017/9781316672815. URL: http://dx.doi.org/10.1017/9781316672815.

[Mac03] David J.C. MacKay. *Information Theory, Inference and Learning Algorithms*. Cambridge University Press, 2003. ISBN: 9780521642989. URL: https://books.google.de/books?id=AKuMj4PN%5C_EMC.

[Mei03] Marina Meilă. "Comparing Clusterings by the Variation of Information". In: *Annual Conference Computational Learning Theory*. 2003.

[Men+09] Bjoern H Menze et al. "A comparison of random forest and its Gini importance with standard chemometric methods for the feature selection and classification of spectral data". In: *BMC bioinformatics* 10 (2009), pp. 1–16.

[MHM20] Leland McInnes, John Healy, and James Melville. *UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction*. 2020. arXiv: 1802.03426 [stat.ML].

[MHV07] Markus Maier, Matthias Hein, and Ulrike Von Luxburg. "Cluster identification in nearest-neighbor graphs". In: *Algorithmic Learning Theory: 18th International Conference, ALT 2007, Sendai, Japan, October 1-4, 2007. Proceedings 18*. Springer. 2007, pp. 196–210.

[Mon+14] Guido Montúfar et al. *On the Number of Linear Regions of Deep Neural Networks*. 2014. DOI: 10.48550/ARXIV.1402.1869. URL: https://arxiv.org/abs/1402.1869.

[MOT15] Alexander Mordvintsev, Christopher Olah, and Mike Tyka. *Inceptionism: Going Deeper into Neural Networks*. 2015. URL: https://ai.googleblog.com/2015/06/inceptionism-going-deeper-into-neural.html (visited on 03/08/2023).

[MS+91] Thomas Martinetz, Klaus Schulten, et al. "A" neural-gas" network learns topologies". In: (1991).

[Mur22] Kevin P Murphy. *Probabilistic machine learning: an introduction*. MIT press, 2022.

[NDB18] Will Nash, Tom Drummond, and Nick Birbilis. "A review of deep learning in the study of materials degradation". In: *npj Materials Degradation* 2.1 (2018), p. 37.

[NP15] M. E. J. Newman and Tiago P. Peixoto. "Generalized Communities in Networks". In: *Phys. Rev. Lett.* 115 (8 Aug. 2015), p. 088701. DOI: 10.1103/PhysRevLett.115.088701. URL: https://link.aps.org/doi/10.1103/PhysRevLett.115.088701.

[OHa] Tom O'Haver. *Multicomponent Spectroscopy*. URL: https://terpconnect.umd.edu/~toh/spectrum/CurveFittingB.html (visited on 06/04/2023).

[Per+17] Gabriel Pereyra et al. "Regularizing neural networks by penalizing confident output distributions". In: *arXiv preprint arXiv:1701.06548* (2017).

[RFB15] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. *U-Net: Convolutional Networks for Biomedical Image Segmentation*. 2015. DOI: 10.48550/ARXIV.1505.04597. URL: https://arxiv.org/abs/1505.04597.

[Rhy20] H.I. Rhys. *Machine Learning with R, the tidyverse, and mlr*. Manning Publications, 2020. ISBN: 9781617296574. URL: https://books.google.de/books?id=BoeryQEACAAJ.

[Rip07] Brian D Ripley. *Pattern recognition and neural networks*. Cambridge university press, 2007.

[RPK19] M. Raissi, P. Perdikaris, and G.E. Karniadakis. "Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations". In: *Journal of Computational Physics* 378 (2019), pp. 686–707. ISSN: 0021-9991. DOI: https://doi.org/10.1016/j.jcp.2018.10.045.

[Sam69] J.W. Sammon. "A Nonlinear Mapping for Data Structure Analysis". In: *IEEE Transactions on Computers* C-18.5 (1969), pp. 401–409. DOI: 10.1109/T-C.1969.222678.

[Sch+17] John Schulman et al. "Proximal policy optimization algorithms". In: *arXiv preprint arXiv:1707.06347* (2017).

[Sch+19] Philippe Schwaller et al. "Molecular transformer: a model for uncertainty-calibrated chemical reaction prediction". In: *ACS central science* 5.9 (2019), pp. 1572–1583.

[Sch+20] Philippe Schwaller et al. "Unsupervised attention-guided atom-mapping". In: (2020).

[Set09] Burr Settles. "Active learning literature survey". In: (2009).

[Son+20] Yang Song et al. "Score-based generative modeling through stochastic differential equations". In: *arXiv preprint arXiv:2011.13456* (2020).

[Son21] Yang Song. https://yang-song.net/blog/2021/score/. 2021.

[Son22] Yang Song. https://cvpr2022-tutorial-diffusion-models.github.io/. 2022.

[SP96] Jochen Stutz and Ulrich Platt. "Numerical analysis and estimation of the statistical error of differential optical absorption spectroscopy measurements with least-squares methods". In: *Applied optics* 35.30 (1996), pp. 6041–6053.

[Spe+20] Artur Speiser et al. "Deep learning enables fast and dense single-molecule localization with high accuracy". In: *Nature methods* 18 (2020), pp. 1082–1090.

[Sta+89] Thomas A. Stamey et al. "Prostate Specific Antigen in the Diagnosis and Treatment of Adenocarcinoma of the Prostate. II. Radical Prostatectomy Treated Patients". In: *The Journal of Urology* 141.5 (1989), pp. 1076–1083. ISSN: 0022-5347. DOI: https://doi.org/10.1016/S0022-5347(17)41175-X. URL: https://www.sciencedirect.com/science/article/pii/S002253471741175X.

[STR17] Thiago Serra, Christian Tjandraatmadja, and Srikumar Ramalingam. *Bounding and Counting Linear Regions of Deep Neural Networks*. 2017. DOI: 10.48550/ARXIV.1711.02114. URL: https://arxiv.org/abs/1711.02114.

[SZ14] Karen Simonyan and Andrew Zisserman. *Very Deep Convolutional Networks for Large-Scale Image Recognition*. 2014. DOI: 10.48550/ARXIV.1409.1556. URL: https://arxiv.org/abs/1409.1556.

[Tan20] M. Farhan Tandia. *Visualization Method: SNE vs t-SNE*. 2020. URL: https://www.linkedin.com/pulse/visualization-method-sne-vs-t-sne-implementation-using-tandia (visited on 03/31/2023).

[Tim17] (https://stats.stackexchange.com/users/35989/tim) Tim. *Choosing between uninformative beta priors*. Cross Validated. URL:https://stats.stackexchange.com/q/298176 (version: 2017-11-14). 2017. eprint: https://stats.stackexchange.com/q/298176. URL: https://stats.stackexchange.com/q/298176.

[TYD77] Yoshio Takane, Forrest W Young, and Jan De Leeuw. "Nonmetric individual differences multidimensional scaling: An alternating least squares method with optimal scaling features". In: *Psychometrika* 42 (1977), pp. 7–67.

[Usz] Jakob Uszkoreit. https://ai.googleblog.com/2017/08/transformer-novel-neural-network.html.

[Van14] Laurens Van Der Maaten. "Accelerating t-SNE using tree-based algorithms". In: *The journal of machine learning research* 15.1 (2014), pp. 3221–3245.

[Vas+17] Ashish Vaswani et al. "Attention is all you need". In: *Advances in neural information processing systems* 30 (2017).

[VH08] Laurens Van der Maaten and Geoffrey Hinton. "Visualizing data using t-SNE." In: *Journal of machine learning research* 9.11 (2008).

[Wei88] David Weininger. "SMILES, a chemical language and information system. 1. Introduction to methodology and encoding rules". In: *Journal of chemical information and computer sciences* 28.1 (1988), pp. 31–36.

[Wen21] Lilian Weng. https://lilianweng.github.io/posts/2021-07-11-diffusion-models/. 2021.

[Wu+19] Felix Wu et al. "Simplifying Graph Convolutional Networks". In: *Proceedings of the 36th International Conference on Machine Learning*. Ed. by Kamalika Chaudhuri and Ruslan Salakhutdinov. Vol. 97. Proceedings of Machine Learning Research. PMLR, 2019, pp. 6861–6871. URL: https://proceedings.mlr.press/v97/wu19e.html.

[Yas20] Jacob Yasonik. "Multiobjective de novo drug design with recurrent neural networks and nondominated sorting". In: *Journal of Cheminformatics* 12.1 (2020), p. 14.