Introduction to Machine Learning

by dcamenisch

1 Introduction

WIP

- f^* ground truth function
- $\hat{f}$ prediction model
- ϵ noise

2 Regression

In this first part we are gonna focus on fitting lines to datapoints. For this we will introduce the **machine learning pipeline**. It consists of three parts and has the goal to find the optimal model $\hat{f}$ for given data D, that we can use to predict new data.

The three parts of the ML Pipeline are the function class **F**, the loss function ℓ and the optimization method.

2.1 Linear Regression

Given the data (x_i, y_i) we use models of the form $f(x) = w^\top x + b$ to fit the data. To find the optimal values for w and b we try to reduce the **squared loss**:

$$\ell(y, f(x)) := \frac{1}{n}\sum(y_i - f(x_i))^2 = \frac{1}{n}||y - Xw||_2^2$$

In the matrix notation b is part of w. The closed form solution for linear regression is given by the normal equation $Ax - b \Rightarrow x = (A^\top A)^{-1}A^\top y$:

$$\hat{w} = (X^\top X)^{-1}X^\top y$$

We can also get the closed form solution by using the fact that the squared loss is a convex function and $\hat{w}$ is the global minima of this function. Therefore we can calculate the gradient $\nabla\ell(y, f(x))$ and solve for 0 to find $\hat{w}$. Later, we will see a more efficient way of finding $\hat{w}$.

2.1.1 Different Loss Functions

The square loss penalizes over- and underestimation the same. Further it puts a large penalty on outliers (grows quadratically). While this is often good, we might want a different loss function, some possibilities are:

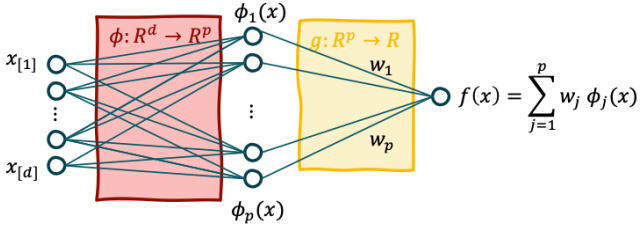
- Huber loss - ignores outliers ($a = y - f(x)$)

$$\ell_\delta(y, f(x)) := \begin{cases} \frac{1}{2}a^2 & \text{for } |a| \le \delta \\ \delta \cdot (|a| - \frac{1}{2} \cdot \delta) & \text{otherwise} \end{cases}$$

- Asymmetric losses - weigh over- and underestimation differently

2.2 Nonlinear Functions

Linear functions helped us to keep the calculations "simple" and find good solutions. But often there are problems that are more complex and would require nonlinear functions. The avoid using nonlinear functions we introduce feature mapping.



From out input vector x we extract a **feature vector** $\phi(x)$ by using a fixed mapping ϕ that can consist of any nonlinear function. On this feature vector we can use the already known methods for linear functions to find a solution.

2.3 Regularization

We will later see that too complex models are not always good, as the use too many features. If we want to reduce the number of features, we can encourage sparsity by introducing a penalty term. We commonly use:

- **Lasso Regression**: $\underset{w\in\mathbb{R}^d}{\text{argmin}}||y - \Phi w||^2 + \lambda||w||_1$
- **Ridge Regression**: $\underset{w\in\mathbb{R}^d}{\text{argmin}}||y - \Phi w||^2 + \lambda||w||_2$

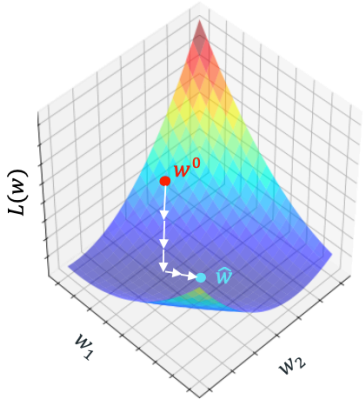
Lasso regression sets a lot of weights to zero, while ridge regression just puts the focus on lower weights.

3 Optimization

If the closed form is not available or desirable, as calculating it is expensive, we use the **gradient descent** algorithm. It works by initializing w^0 and iteratively moving it towards the optimal solution. We choose the direction by calculating $\nabla\ell(w)$ and then multiply it by the stepsize / learning rate η:

$$w^{t+1} = w^t - \eta_t \cdot \nabla\ell(w^t)$$

Convergence is only guaranteed for the convex case, else we might get stuck at any stationary point.



As the slope gets smaller, we want to decrease η, so that we do not overshoot. For the linear regression case we have:

$$||w^t - w^*||_2 \le \rho^t||w^0 - w^*||_2, \quad \rho = ||I - \eta X^\top X||_{op}$$

Where ρ is the convergence speed for constant stepsize η. This leads to an optimal fixed stepsize of:

$$\eta = \frac{2}{\lambda_{\min} + \lambda_{\max}}$$

We stop when new iterations does not change anymore (below a certain threshold).
To make gradient descent more stable / robust against ill-conditioned landscapes we might add momentum:

$$w^{t+1} = w^t + \gamma\Delta w^{t-1} - \eta_t\nabla\ell(w^t)$$

3.1 Stochastic Gradient Descent

When we have a lot of data, it is costly to compute the gradient, so we only use a minibatch S of the dataset D (randomly sampled without replacement). Now the update step looks like this:

$$w^{t+1} = w^t - \eta_t \cdot \nabla \ell_S(w^t)$$

Where the loss is only calculated over the minibatch S. This method also gives us a chance to escape saddle points.

4 Model Error

We generally want to minimize the estimation error $\ell(\hat{f}(x), f^*(x))$, since we do not know f^* we can not actually compute this value. Instead, we usually observe $y_i = f^*(x_i) + \epsilon_i$. For each observed sample we can compute the **prediction error** $\ell(\hat{f}(x), y)$, in fact we are often interested in the average prediction error or **generalization error**:

$$R(\hat{f}) := \mathbb{E}_{x,y}[\ell(\hat{f}(x), y)] = \mathbb{E}_x[\ell(\hat{f}(x), f^*(x))] + \epsilon$$

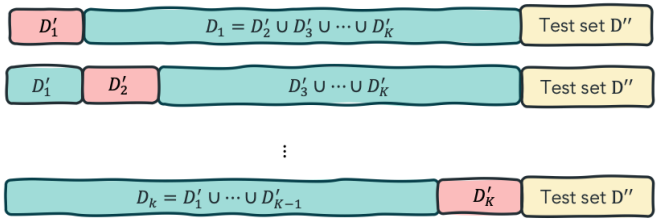
The generalization error computed over all possible (x, y) pairs weighted by how likely each is.
The training loss is often to optimistic to approximate the generalization error. To get a better approximation we split our data into training and test set.

By only using the training set to fit our model, we have the test data to get a better estimate of the generalization error.

4.1 Cross-Validation

When choosing between different model, we might choose the model with the lowest test set error, this may introduce a systematic bias. To prevent this from happening we can split the training set again, creating a validation set. Now the idea is to choose the model with the best validation error and use the test set only to get the estimate for the generalization error.

Setting aside so much data can be wasteful. So we introduce k**-fold cross-validation**



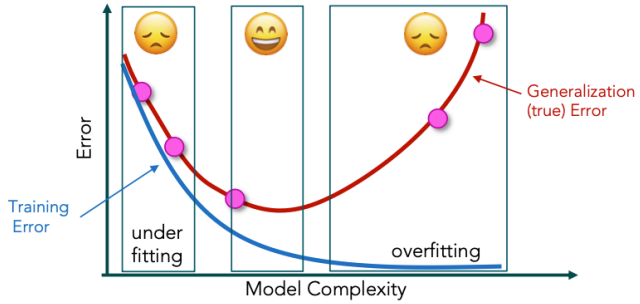
We proceed as follows:

1. For all folds $k = 1, ..., K$:
   (a) Train $\hat{f}_k$ on $D' - D'_k$
   (b) Compute val. error $R_k = \frac{1}{|D'_k|} \sum_{x,y} \ell(\hat{f}_k(x), y)$
2. Compute cross-validation error $CV = \frac{1}{K} \sum_{i=1}^{K} R_i$
3. Pick model with lowest cross-validation error CV
4. Evaluate the model using the test set D''

For K very large, we can get the best approximation, if $K = |D'|$ we call it leave-one-out cross-validation (LOOCV).

4.2 Model Complexity

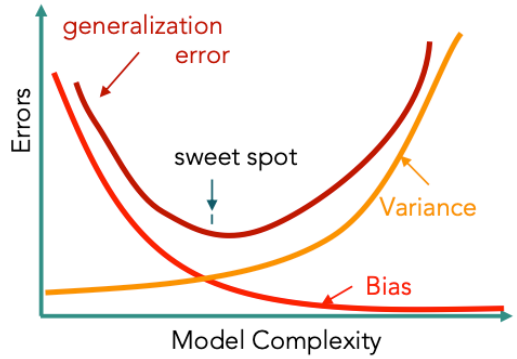
Model complexity is closely related to training and generalization error.



4.3 Bias and Variance

For different datasets $D_1, ..., D_K$ we define:

- **Bias** - distance of the average model $\bar{f} = \frac{1}{K} \sum_{i=1}^{K} \hat{f}_i$ to the ground truth $\mathbb{E}_x[\ell(\bar{f}(x), f^*(x))]$
- **Variance** - average distance of the models to the average model $\mathbb{E}_x[\frac{1}{K} \sum_{i=1}^{K} \ell(\hat{f}_i(x), \bar{f}(x))]$



5 Classification

Instead of predicting $y \in \mathbb{R}$, we limit y to be in a finite, discrete set Y (e.g. $\{-1, +1\}$). When looking at binary classification we often use the labels $-1, +1$ and let the predicted value be equal to $\hat{y} = \text{sgn}\hat{f}(x)$. Similar to regression we care about the generalization error:

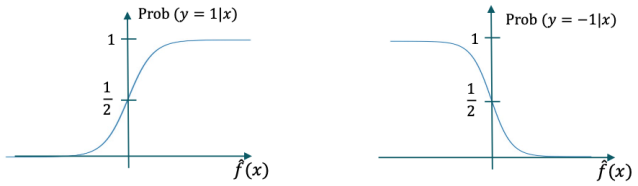
$$R(\hat{f}) = \mathbb{P}_{x,y}[y \neq \text{sgn}\hat{f}(x)] = \mathbb{E}_{x,y}[\ell_{0-1}(\hat{f}(x), y)]$$

Where we call $\ell_{0-1}(\hat{f}(x), y) = \mathbb{I}_{y \neq \text{sgn}\hat{f}(x)}$ the **zero-one loss**. Since this loss is neither convex nor continuous, we can not efficiently minimize the training error with it. Therefore we introduce different type of loss functions:

- **Exponential loss**: $\ell_{\exp}(\hat{f}(x), y) = e^{y\hat{f}(x)}$
- **Logistic loss**: $\ell_{\log}(\hat{f}(x), y) = \log(1 + e^{y\hat{f}(x)})$
- **Hinge loss**: $\ell_{\text{hinge}}(\hat{f}(x), y) = \max(0, 1 - y\hat{f}(x))$
- **Linear loss**: $\ell_{\text{lin}}(\hat{f}(x), y) = y\hat{f}(x)$

We will mainly focus on the logistic loss (also called **logistic regression**), as in practice it is the most used. We can derive that the logistic loss is the negative conditional log likelihood $\mathbb{P}[y = +1|x]$ or $\mathbb{P}[y = -1|x]$ that is parameterized by $\hat{f}(x)$ via the **softmax transformation**. We define (similar for $y = -1$):

$$\mathbb{P}[y = +1|x] = \frac{1}{1 + e^{-\hat{f}(x)}}$$



Using this we can define the probability vector:

$$\hat{p}(x) = (\mathbb{P}[y = -1|x],\ \mathbb{P}[y = +1|x])$$

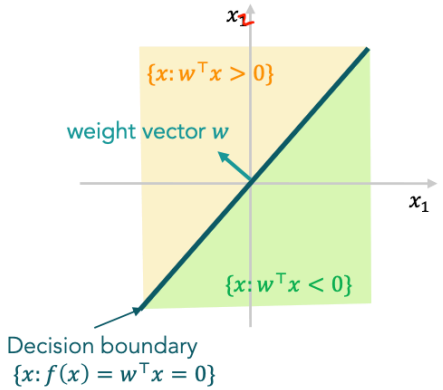
If we want to extend the log loss to multiple classes, we define a vector $\tilde{f}(x) = (\hat{f}_1(x), ..., \hat{f}_K(x))$ and transform it using softmax:

$$\hat{p}_k = \frac{e^{\hat{f}_k(x)}}{\sum_{i=1}^{K} e^{\hat{f}_j(x)}}$$

For the multiclass case we choose the classifier error to be the maximal entry of $\hat{p}$ if $y \neq \hat{y}$.

5.1 Linear Classifiers

Linear classifiers use functions form the class $F = \{f \mid f(x) = w^\top x,\ w \in \mathbb{R}^d\}$. We already know that this class of functions makes training and prediction simple. The decision boundary of the function is given by $\{x \mid f(x) = 0\}$.



To train our classifier we can use gradient descent. The gradient of the logistic loss is given by:

$$\nabla \ell(\hat{f}(x), y) = \frac{y_i x_i}{1 + e^{y_i \hat{f}(x)}}$$

For linearly separable data, gradient descent on the logistic loss converges to the direction w_{MM} that maximizes the minimum ℓ_2-distance between the decision boundary and y_i. We call this the **maximum-margin** solution.
In particular we can write:

$$w_{\text{MM}} = \underset{||w||_2=1}{\text{argmax}} \min_i y_i w^\top x_i) = \underset{||w||_2=1}{\text{argmax}}\ \text{margin}(w)$$

Instead of just linear functions, we can again use feature mapping to receive nonlinear classifiers.

5.2 Support Vector Machines

For general w that correctly separates the data, $\frac{\text{margin}(w)}{||w||_2}$ is the min. distance of any point to the decision boundary. If we use general w the solution is not unique anymore. But we can rescale any unit norm w by $\alpha = \frac{1}{\text{margin}(w)}$ such that $\alpha w = \tilde{w}$. So instead of searching within unit norm w to find w_{MM} with maximum margin, we can search within all $\tilde{w}$ with $\text{margin}(\tilde{w}) = 1$ to find the one that maximizes:

$$\frac{\text{margin}(\tilde{w})}{||\tilde{w}||_2} = \frac{1}{||\tilde{w}||_2}$$

This is how support vector machines work. More formal:

$$\hat{w} = \min_w ||w||_2 \quad \text{s.t. } y_i w^\top x_i \geq 1 \text{ for all } i = 1, ..., n$$

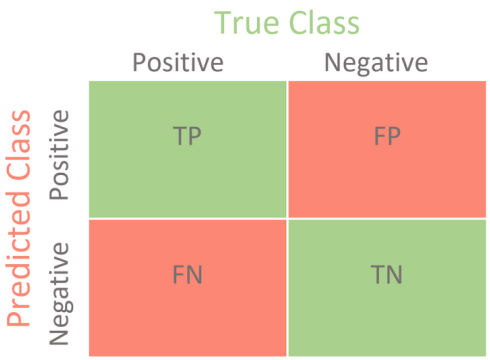
If the data is not linearly separable, we might want to use a **soft-margin SVM**. Since not all constraints can hold, we want to allow some "slack" in the constraints:

$$\hat{w} = \min_{w,\xi} \frac{1}{2}||w||_2^2 + \lambda \sum_{i=1}^{n} \max(0, 1 - y_i w^\top x_i)$$

The later part penalizes any margin violations. To find the optimal λ one might use cross-validation.

6 Hypothesis Testing

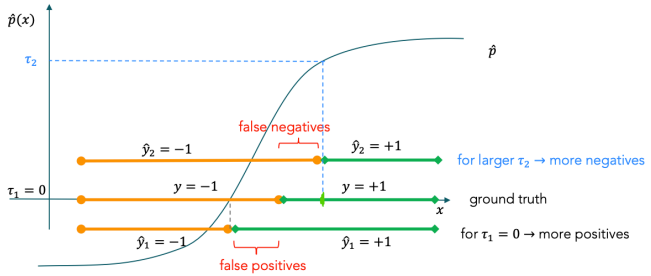
We focused a lot to derive good surrogate losses for the 0-1 loss. But is this error really a good metric? Hypothesis testing is a way to express asymmetry in classification tasks. For this we introduce the confusion matrix:



Further we define:

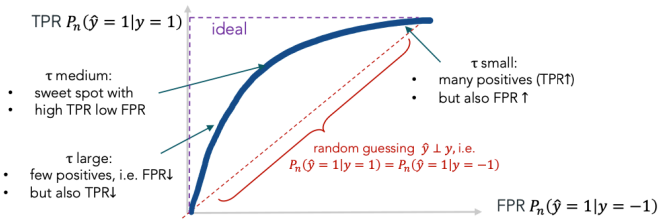
$$\text{error}_1/\text{FPR} = \frac{FP}{TN + FP}, \quad \text{error}_2/\text{FNR} = \frac{FN}{TP + FN}$$

We want to find a test that minimizes the FPR, while controlling the FNR. This can be viewed as defining a null hypothesis $H_0(x)$ and then deciding to accept or reject it (H_0 is always the positive class). When choosing H_0 we want it to represent the more crucial class one to get right, e.g. it is more important to truly classify a person as sick than to classify them as healthy. To decide it we accept or reject H_0 we fix τ, where we accept $H_0(x)(\hat{y} = -1)$ if $\hat{p}(x) < \tau$ and the opposite way around.



6.1 AUROC

We want to have a large recall $\frac{TP}{\#[y=+1]}$ but also a small FPR. Based on these metrics we can draw the ROC curve by varying τ.



We can either choose our model by caring about a specific point, e.g. TPR @ FPR = 5%, or we choose whichever courve gets closer to the ideal curve, that is maximizing the area under the curve.

7 Kernels

We have previously seen how we can get nonlinear functions via feature maps ϕ. But there are limits to these feature maps, they can introduce a lot of computational complexity (feature explosion) and there are also infinite feature maps we can not get this way. If we want to avoid these limitations we use the **kernel trick**. It consists of two steps:

1. We know that the solution $\hat{w}$ is in the column space of $\Phi^\top$. Therefore among the global minimizers one has the form $\hat{w} = \Phi^\top \hat{\alpha}$ with $\hat{\alpha} \in \mathbb{R}^n$ so that:

$$\hat{f}(x) = \hat{w}^\top \phi(x) = \hat{\alpha}^\top \Phi \phi(x) = \sum_{i=1}^{n} \hat{\alpha}_i \cdot \phi(x_i)^\top \phi(x)$$

   Notice that $\hat{\alpha}$ only depends on x_i via inner products $\phi(x_i)^\top \phi(x_j)$. Using this we can define a symmetric kernel function $k(x,z) = \phi(x)^\top \phi(z)$ and a corresponding kernel matrix $K = \Phi\Phi^\top$.

2. Sometimes we can more efficiently compute the inner products / evaluate the kernel function, e.g. for the feature vector $\phi(x) = [1, \sqrt{2}x_1, \sqrt{2}x_2, x_1^2, x_2^2, \sqrt{2}x_1x_2]$, the inner product is:

$$\phi(x)^\top \phi(z) = (1 + x_1z_1 + x_2z_2)^2 = (1 + x^\top z)^2 = k(x,z)$$

   This kernel function is a lot less expensive to compute.

7.1 Example for Ridge Regression

Remember $w = \Phi^\top \alpha$ and $K = \Phi\Phi^\top$, applying this to ridge regression we get:

$$\begin{aligned}\frac{1}{n}||y - \Phi w||_2^2 + \lambda||w||_2^2 &= \frac{1}{n}||y - \Phi\Phi^\top\alpha||_2^2 + \lambda||\Phi^\top\alpha||_2^2 \\ &= \frac{1}{n}||y - K\alpha||_2^2 + \lambda\alpha^\top K\alpha\end{aligned}$$

7.2 Different Kernels

A valid kernel must have the following properties:

- K is symmetric because of the inner products: $k(x,z) = k(z,x)$
- K is positive-semidefinite for any choice of inputs $x_1, ..., x_n$, i.e. $z^\top K z \geq 0$

Common kernel choices are:

- **linear**: $k(x,z) = x^\top z$
- **polynomial**: $k(x,z) = (x^\top z + 1)^m$
- **rbf**: $k(x,z) = \exp\left(-\frac{||x-z||_\alpha}{\tau}\right)$

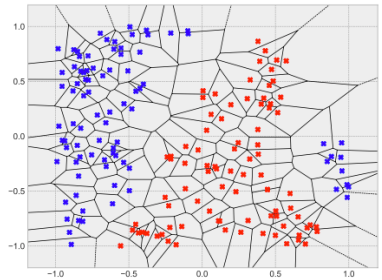
An RBF kernel with $\alpha = 2$ is also called a gaussian kernel and one with $\alpha = 1$ is a laplacian kernel. Special about the RBF kernel is that it corresponds to infinite dimensional features.
Given valid kernels we can compose new ones by conserving kernel convexity:

- $k = k_1 + k_2$
- $k = k_1 \cdot k_2$
- $k = c \cdot k_1 \quad \forall c > 0$
- $k = f(k_1) \quad \forall f$ convex

Mercers Theorem: Any valid kernel can be decomposed into a linear combination of inner products.

8 Other Nonlinear Methods

8.1 KNN Classification

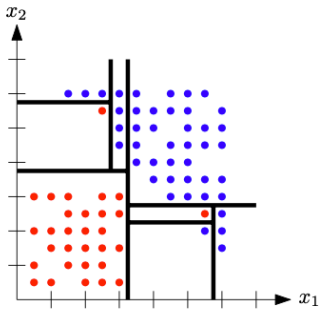


This method does not need any training and classification is done during test time. For a given training set D it works as follows:

1. Pick k and distance metric d
2. For given x, find among $x_1, ..., x_n \in D$ the k closest to $x \to x_{i_1}, ..., x_{i_k}$
3. Output the majority vote of labels $y_{i_1}, ..., y_{i_k}$

This method is very sensitive to k and becomes unstable in high dimensions. We might need large n for good results but computation can be reduces when allowing for some error probability.

8.2 Decision Trees



A decision tree returns a partition of X with sets aligned with the main axis. A given x is assigned the majority class of the partition it lands in. The partitions can be modelled as leaf nodes of a binary tree. Single trees can easily overfit to noise, we have to choose the depth of the tree carefully.

9 Neural Networks

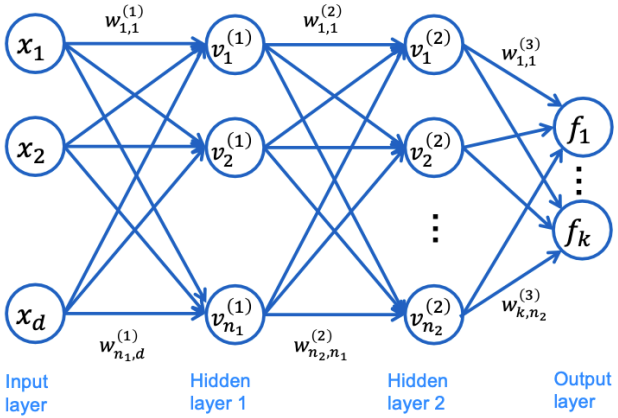
Success in learning crucially depends on the quality of the features. The key idea of neural networks is to parameterize the feature maps and optimize over the parameters.
We want to build a complex model out of simple components:

$$\phi(x, \theta) = \varphi(\theta^\top x)$$

Hereby, $\theta \in \mathbb{R}^d$ are the weights and $\varphi : \mathbb{R} \mapsto \mathbb{R}$ is a non-linear **activation function**. Possible activation functions are:

- **Identity**: $\varphi(z) = z$
- **Sigmoid**: $\varphi(z) = \frac{1}{1+\exp(-z)}$
- **Tanh**: $\varphi(z) = \tanh z = \frac{\exp(z)-\exp(-z)}{\exp(z)+\exp(-z)}$
- **ReLU**: $\varphi(z) = \max(0, z)$

Nesting these components we create networks of the form:



Where $v_i = \varphi(z_i)$ and z_i is the sum of inputs times their weight. To deal with biases we introduce a "constant" 1 feature to each layer. Note that we can have as many layers as we want and use different activation functions per layer. Such networks are typically trained via SGD.
By the universal approximation theorem, we can approximate any arbitrary smooth target function, given at least one layer with sufficient width.

9.1 Forward Propagation

This is the process of calculating the output for a given input.

- For input layer
$$v^{(0)} = [x; 1]$$
- For each hidden layer $1 : L-1$
$$z^{(l)} = W^{(l)} v^{(l-1)} \quad \text{and} \quad v^{(l)} = [\varphi(z^{(l)}); 1]$$
- For output layer
$$f = W^{(L)} v^{(L-1)}$$

9.2 Backpropagation

We can use the loss functions we already know to compute the loss. For multi output networks, we use the sum of per-output for regression tasks and cross-entropy loss for classification tasks. As mentioned we use SGD to fit our neural network. We want to jointly optimize over all weight for all layers. This is generally a non-convex optimization problem. Nevertheless, we can try to find a local optimum. In order to apply SGD, need to compute $\nabla_W \ell(W; x, y)$ w.r.t. each weight $w_{i,j}^{(l)}$:

$$\left(\nabla_{W^{(L)}} \ell\right)^T = \frac{\partial \ell}{\partial W^{(L)}} = \frac{\partial \ell}{\partial f} \frac{\partial f}{\partial W^{(L)}}$$

$$\left(\nabla_{W^{(L-1)}} \ell\right)^T = \frac{\partial \ell}{\partial W^{(L-1)}} = \frac{\partial \ell}{\partial f} \frac{\partial f}{\partial z^{(L-1)}} \frac{\partial z^{(L-1)}}{\partial W^{(L-1)}}$$

$$\left(\nabla_{W^{(L-2)}} \ell\right)^T = \frac{\partial \ell}{\partial W^{(L-2)}} = \frac{\partial \ell}{\partial f} \frac{\partial f}{\partial z^{(L-1)}} \frac{\partial z^{(L-1)}}{\partial z^{(L-2)}} \frac{\partial z^{(L-2)}}{\partial W^{(L-2)}}$$

$$\vdots$$

$$\left(\nabla_{W^{(i)}} \ell\right)^T = \frac{\partial \ell}{\partial W^{(i)}} = \frac{\partial \ell}{\partial f} \frac{\partial f}{\partial z^{(L-1)}} \frac{\partial z^{(L-1)}}{\partial z^{(L-2)}} \cdots \frac{\partial z^{(i+1)}}{\partial z^{(i)}} \frac{\partial z^{(i)}}{\partial W^{(i)}}$$

Notice that we can reuse calculations from **the previous layer** , **forwards pass** and only have to compute **the gradient** for each layer.
Since the optimization problem is non-convex the initialization of the weights matters. With inappropriate weights we can run into exploding or vanishing gradients. To avoid this we randomly initialize the weights based on some distribution assumption for the activation function.

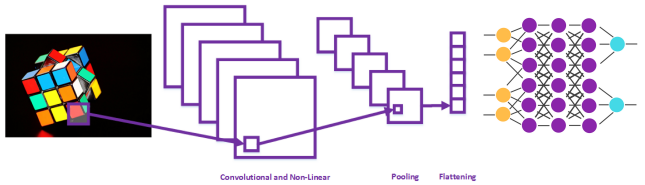
9.3 Overfitting

Since any deep neural network has a lot more parameters then data points to train on, overfitting can happen easily. To avoid this we use:

- **Regularization**: add a penalty on the weights to the cost function
- **Early Stopping**: stop training once validation error stop to decrease
- **Dropout**: randomly ignore hidden units during training with probability p, after training all units are used and weights are multiplied by p
- **Batch Normalization**: normalize the input data (mean 0, variance 1) in each layer

9.4 Convolutional Neural Networks

CNN are a specialized architecture for neural networks. The idea is that predictions should be unchanged under some transformations of the data, e.g. rotation of images.



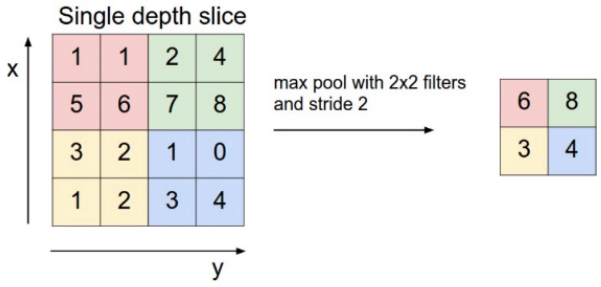
Each layer is not fully connected but structured. The activation function is applied to the element-wise convolution:

$$\varphi(W * v^{(l)})$$

The output dimension when applying m different $f \times f$ filters to an $n \times n$ image with padding p and stride s is:

$$l = \frac{n + 2p - f}{s} + 1$$

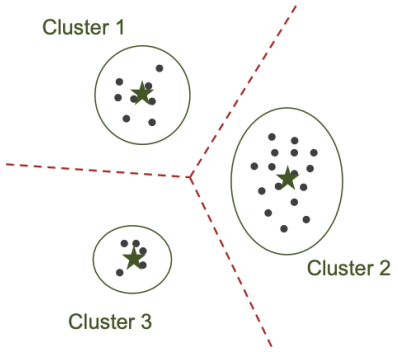
Additionally we might use average or max pooling layers to aggregate several units into a single one, or use stride layers to skip units to decrease size.



10 Unsupervised Learning

10.1 k-Means Clustering

Given an unlabelled dataset, we try to learn feature similarities based on proximity in feature space. Data points with similar features then should be grouped into the same cluster. k-Means tries to represent each cluster by a single (center) point μ_i.



Each data points is assigned by:

$$z_j = \underset{i}{\text{argmin}} ||x_j - \mu_i||_2, \quad z_j \in \{1, ..., k\}$$

To pick the optimal centers we try to minimize the sum of squared distances:

$$\hat{R}(\mu) = \sum_{i=1}^{n} \min_{j \in \{1,...,k\}} ||x_i - \mu_j||_2^2$$

This is a non-convex optimization problem and NP-hard. One way of finding a good solution is Lloyd's heuristics:

1. Initialize cluster centers $\mu^{(0)}$
2. While not converged:
   (a) Assign each point to closest center:
   $$z_i \leftarrow \underset{j \in \{1,...,k\}}{\text{argmin}} ||x_i - \mu_j^{(t-1)}||_2$$
   (b) Update centers as mean of assigned data points:
   $$\mu_j^{(t)} \leftarrow \frac{1}{n_j} \sum_{i|z_i=j} x_i$$

This guarantees to monotonically decrease the average squared distance in each iteration and converges to a local optimum. This local optimum is strongly dependent on the initialization. One way to initialize the centers is **k-Means++**:

1. Start with random data point as center $\mu_1 = x_i$ where $i \sim \text{Unif}\{1, ..., n\}$
2. Add centers $2, ..., k$ randomly, proportionally to the squared distance to closest selected center:
   $$\text{given } \mu_{1:j} \text{ pick } \mu_{j+1} = x_i$$
   $$\text{where } p(i) = \frac{1}{z} \min_{l \in \{1,...,j\}} ||x_i - \mu_l||_2^2$$

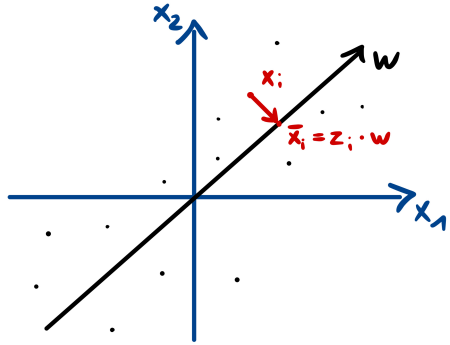
To find the optimal number of clusters k can not be done by cross-validation, as the loss keeps decreasing with larger k. We can either keep increasing k until we reach a negligible decrease in loss or we can use regularization to add a penalty term for larger k.

10.2 Principal Component Analysis

PCA is used for dimensionality reduction. Given data $x_i \in \mathbb{R}^d$ we want to obtain a low-dimensional representation $z_i \in \mathbb{R}^k$ where $k < d$. One of the benefits of low-dimensional representation is that we can visualize data that we otherwise could not. Feature discovery is another use case for PCA, it can help us to discover features from data, e.g. Eigenfaces. We assume that our data is centered around the origin.
Our goal is to learn the function $f(x) = Ax$ that maps the high dimensional data to the lower dimensions, while minimizing the reconstruction error. First we will look at the case $k = 1$.

$$\min_{w,z} \sum_{i=1}^{n} ||x_i - z_i w||_2^2 \quad \text{s.t. } ||w||_2 = 1$$

We limit w to be of unit length to guarantee a unique solution.



Since for a given w the minimal distance vector $\bar{x}_i - x_i$ is perpendicular to w, we find that the optimal solution for $z_i = w^\top x_i$. We can now substitute z_i and receive the following optimization goal:

$$\hat{w} = \underset{||w||_2=1}{\text{argmin}} \sum_{i=1}^{n} ||x_i - ww^\top x_i||_2^2$$

Which again can be reformulated as:

$$\hat{w} = \underset{||w||_2=1}{\text{argmax}} \sum_{i=1}^{n} (w^\top x_i)^2 \text{ or } \hat{w} = \underset{||w||_2=1}{\text{argmax}} \, w^\top \Sigma w$$

Where $\Sigma = \frac{1}{n} \sum_{i=1}^{n} x_i x_i^\top$ is the empirical covariance. Since we still have an argmax this is not a minimization problem anymore and we can not find a solution like in previous problems. There still exists a closed form solution given by the principal eigenvector of Σ, i.e. $w = v_1$ where for $\lambda_1 \geq ... \geq \lambda_d \geq 0$:

$$\Sigma = \sum_{i=1}^{d} \lambda_i v_i v_i^\top$$

At this place we will omit the proof for the closed form solution.

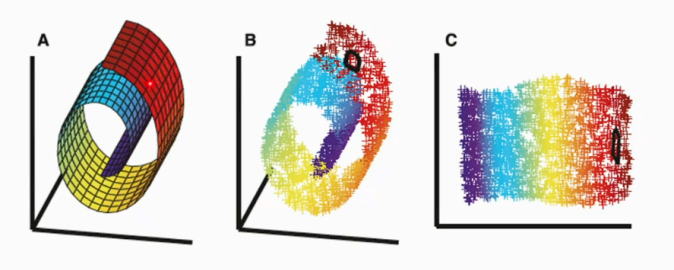
Up until now everything was for $k = 1$. For $k > 1$ we have to change the normalization from $||w||_2 = 1$ to $W^\top W = I$ everything else is basically the same, we just take the first k principal eigenvectors so that $W = [v_1, ..., v_k]$.
Choosing the optimal k is different depending on our goal, for feature induction we use cross-validation else we often pick k so that the variance of our data is mostly explained (other dimensions would add little information).

10.2.1 PCA through SVD

Another way of obtaining the PCA is through singular-value decomposition. Recall that we can represent any data matrix X as $USV^\top$ where S is a diagonal matrix containing the singular values (singular values being the square root of eigenvalues). Now the top k principal components are exactly the first k columns of V.

10.2.2 Kernel PCA

Again we run into problems trying to work with complex arrangements of data, e.g. circles, swiss roll, etc.



Similar to supervised learning where we worked with kernels, we can take the same approach for unsupervised learning. Since it holds $\Sigma = \frac{1}{n}\sum_{i=1}^{n} x_i x_i^\top = X^\top X$ we can apply the kernel trick. We start by assuming $w = \Phi^\top \alpha$, plugging this into our objective and the constraint we end up with:

$$\hat{\alpha} = \underset{\alpha}{\text{argmax}} \frac{\alpha^\top K^\top K \alpha}{\alpha^\top K \alpha}$$

We arrive at the general closed form solution:

$$\alpha^{(i)} = \frac{1}{\sqrt{\lambda_i}} v_i \quad K = \sum_{i=1}^{n} \lambda_i v_i v_i^\top \quad \lambda_1 \geq ... \geq \lambda_n \geq 0$$

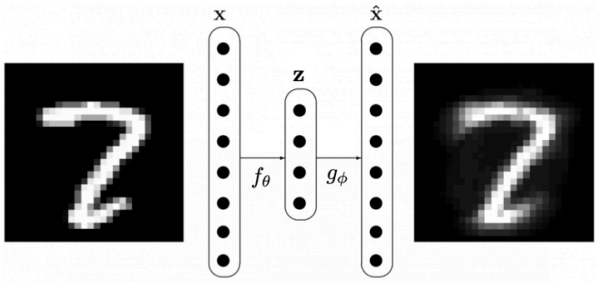
Given this, a new point x is projected as z where:

$$z_i = \sum_{j=1}^{n} \alpha_j^{(i)} k(x_j, x)$$

10.3 Autoencoders

Autoencoders are neural networks with a bottleneck layer and $d_{in} = d_{out}$. We want to minimize $\frac{1}{n}\sum_{i=1}^{n} ||x_i - \hat{x}_i||_2^2$. The idea is to learn the identity function:

$$\hat{x} = f(x; \theta) \text{ where } f(x; \theta) = f_{dec}(f_{enc}(x, \theta_{enc}); \theta_{dec})$$



If linear activation functions and the square loss between input and output are used, then the encoder learns PCA. Otherwise it learns some nonlinear embedding z of the features.