Neural Networks

Implementation and Application

Marius Smytzek

March 19, 2019

Contents

1 Linear Algebra

1.1 Eigendecomposition

The eigendecomposition of a $m \times n$ matrix A consists of eigenvectors v_i, $1 \leq i \leq n$, and the corresponding eigenvalue λ_i, s.t. $Av_i = \lambda_i v_i$. Let $V = \begin{pmatrix} | & & | \\ v_1 & \dots & v_n \\ | & & | \end{pmatrix}$ and $\lambda = \begin{pmatrix} \lambda_1 & & 0 \\ & \ddots & \\ 0 & & \lambda_n \end{pmatrix}$, then the following holds for the eigendecomposition $A = V\lambda V^{-1}$.

Example Eigendecomposition of $A = \begin{pmatrix} 5 & 3 \\ 3 & 5 \end{pmatrix}$. Solve the following:

$$\det(A - \lambda\mathbb{1}) = \det\left(\begin{pmatrix} 5 & 3 \\ 3 & 5 \end{pmatrix} - \begin{pmatrix} \lambda & 0 \\ 0 & \lambda \end{pmatrix}\right) = \det\begin{pmatrix} 5-\lambda & 3 \\ 3 & 5-\lambda \end{pmatrix} = 0$$

$$(5-\lambda)^2 - 3^2 = 25 - 10\lambda + \lambda^2 - 9 = \lambda^2 - 10\lambda + 16 = (\lambda - 8)(\lambda - 2) = 0$$

It follows $\lambda_1 = 8, \lambda_2 = 2$. Solve the equations:

$$Av_1 = \lambda_1 v_1 \iff \begin{pmatrix} 5v_{11} + 3v_{12} - \lambda_1 v_{11} \\ 3v_{11} + 5v_{12} - \lambda_1 v_{12} \end{pmatrix} = \begin{pmatrix} -3v_{11} + 3v_{12} \\ 3v_{11} - 3v_{12} \end{pmatrix} = 0$$

$$Av_2 = \lambda_2 v_2 \iff \begin{pmatrix} 5v_{21} + 3v_{22} - \lambda_2 v_{21} \\ 3v_{21} + 5v_{22} - \lambda_2 v_{22} \end{pmatrix} = \begin{pmatrix} 3v_{21} + 3v_{22} \\ 3v_{21} + 3v_{22} \end{pmatrix} = 0$$

It follows $V = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$.

Read Eigenvalues: All lines add up to 8, therefore 8 is the first eigenvalue. $\text{trace}(A) - 8 = 10 - 8 = 2$.

1.2 Singular Value Decomposition

An $m \times n$ matrix A can be written as the product of three matrices, U $(m \times m)$, V $(n \times n)$, and a diagonal matrix D $(m \times n)$, s.t. $A = UDV^T$. U (left-singular vectors) and V (right-singular vectors) are orthogonal matrices. The elements along the diagonal of D are called singular values. U are the eigenvectors of AA^T. V are the eigenvectors of A^TA. The singular values are the square roots of the eigenvalues of A^TA and AA^T.

2 Machine Learning Basics

Machine learning is the ability of a computer to learn concepts without explicit implementation. The general objective is to learn a task (T), from experience (E), based on a performance measure (P).

The experience E is mostly given by a set of training data which could be labeled (supervised learning), unlabeled (unsupervised learning), or a mix of both (semi-supervised learning).

The task T can vary from learning a function $f : \mathbb{R}^n \leftarrow \{1, \ldots, k\}$ for classification, or a function $f : \mathbb{R}^n \leftarrow \mathbb{R}$ for regression, or for instance learn which patterns could be observed in the data (clustering).

The performance measure P is the objective we should be maximized, for instance the accuracy.

2.1 Capacity, Overfitting and Underfitting

A model's capacity is its ability to t a wide variety of functions. A model with low capacity may struggle to fit the data. A model with high capacity may memorize properties of the data which are not qualified for the general case.

If the model's capacity is too low it underfits the data. If its top high it overfits the data. In general if a model underfits it does not represent the data. If it overfits it represents the data too much. Both cases are bad for the general case, see Figure 2.1.

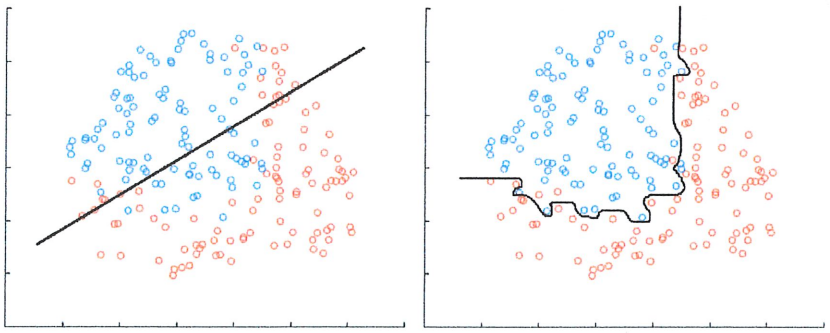


Figure 2.1: The left model underfits, while the right model overfits.

The optimal capacity can be found by plotting the training and generalization errors as a function of the capacity, see Figure 2.2.

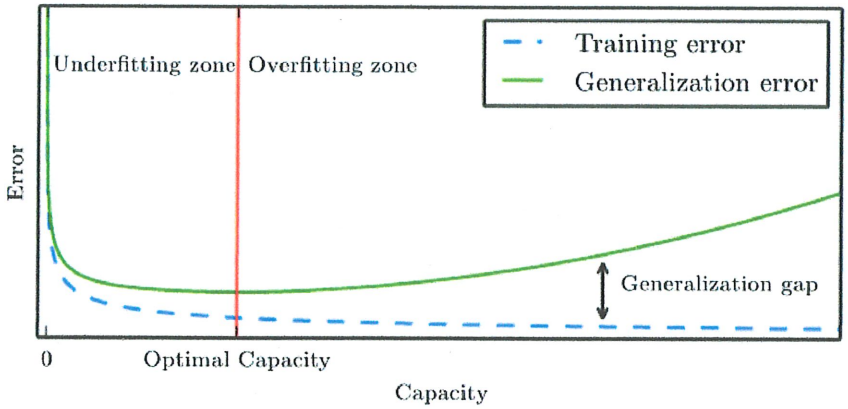


Figure 2.2: The training and generalization errors as a function of capacity.

There are different approaches to prevent under-/overfitting.

- Underfitting
  - Mine/Get new features.
  - Create polynomial features of already existing one.
  - Decrease the regularization parameters (λ).
- Overfitting
  - Get more training examples.
  - Use a smaller set of features.
  - Increase the regularization parameters (λ).

2.1.1 Bias-Variance-Trade-Off

The bias is a model describes how close the model gets too the target. The variance describes how strong distribution of the single results are around the average. **??** describes the bias and variance of a model.

A model with high bias underfits. A model with high variance overfits. A model with high bias and variance is unable to represent the data at all. A model with low bias and variance is called sweet spot.

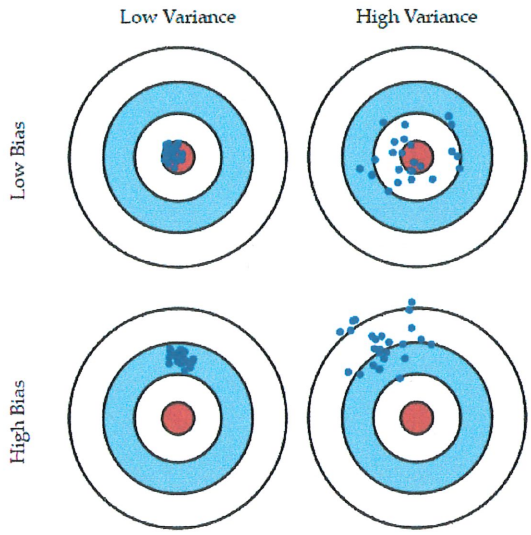


Figure 2.3: The bias and variance of a model.

However, there is a trade-off between bias and variance. With increasing model capacity, the variance increases and the bias decreases. Therefore, the optimal solution can neither have the lowest bias nor the lowest variance. Figure 2.4 shows this trade-off.

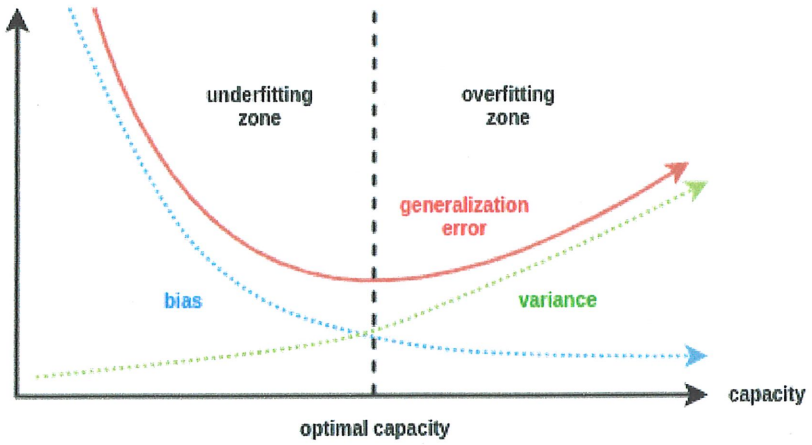


Figure 2.4: The bias and variance as a function of the model's capacity.

2.2 Loss Functions

Loss: $\mathcal{L}^1$-Loss / Mean Absolute Error (MAE)
Formula: $L(y,x) = |y - f(x)|$
Normalized: $\frac{1}{m}\sum_{i=1}^{m}|y_i - f(x_i)|$
$\frac{\delta}{\delta f(x)}L(y,x) = \begin{cases} +1 & f(x) > y \\ -1 & f(x) < y \end{cases}$

Loss: $\mathcal{L}^2$-Loss / Mean Squared Error (MSE)
Formula: $L(y,x) = (y - f(x))^2$
Normalized: $\frac{1}{m}\sum_{i=1}^{m}(y_i - f(x_i))^2$
$\frac{\delta}{\delta f(x)}L(y,x) = 2(y - f(x))$

Loss: Cross-Entropy (Binary)
Formula: $L(y,x) = -(y\log(f(x)) + (1-y)\log(1-f(x)))$
Normalized: $\frac{1}{m}\sum_{i=1}^{m} -(y_i\log(f(x_i)) + (1-y_i)\log(1-f(x_i)))$
$\frac{\delta}{\delta f(x)}L(y,x) = -(\frac{y}{f(x)} - \frac{1-y}{1-f(x)}) = \frac{1-y}{1-f(x)} - \frac{y}{f(x)} = \frac{f(x)-y}{f(x)(1-f(x))}$

Loss: Cross-Entropy (Multiclass)
Formula: $L(y,x) = -\sum_{c=1}^{C} y_c \log(f_c(x))$
Normalized: $\frac{1}{m}\sum_{i=1}^{m} -\sum_{c=1}^{C} y_{i,c}\log(f_c(x_i))$
Note that y is a one-hot encoded label.
$\frac{\delta}{\delta f(x)}L(y,x) = -\sum_{c=1}^{C}\frac{y_c}{f_c(x)}$

2.3 Supervised Learning

2.3.1 Linear Regression

Linear regression is the task which aims to learn a linear function based on examples. This can be represented as the function $y = w^T x(+b)$, where w is a weight vector, which includes weights for each feature. The objective of linear regression is to minimize the MSE, which is

$$w^* = \arg\min_{w\in\mathbb{R}^m} \frac{1}{m}||y - wx||_2^2$$

. This objective can be used to derive a normal equation. The minimum for w has to be an extreme point of the objective.

$$\nabla_w \frac{1}{m}||y - Xw||_2^2 = 0 \iff \frac{1}{m}\nabla_w||y - Xw||_2^2 = 0$$

$$\nabla_w (y - Xw)^T(y - Xw) = 0 \iff \nabla_w(y^T - w^T X^T)(y - Xw) = 0$$

$$\nabla_w y^T y - y^T Xw - w^T X^T y + w^T X^T Xw = 0 \iff \nabla_w y^T y - 2w^T X^T y + w^T X^T Xw = 0$$

$$2X^TXw - 2X^Ty = 0 \iff X^TXw - X^Ty = 0 \iff w = (X^TX)^{-1}X^Ty$$

Since MSE is convex it has exactly one minimum, therefore, this is the optimal solution. However, the inverse could not exist. Linear regression can also be solved by other optimizations.

2.3.2 Ridge Regression

Ridge regression is an expansion to linear regression by using $\mathcal{L}^2$-regularization. Its objective is to minimize $J(w)$.

$$J(w) = MSE_{\text{train}} + \lambda w^T w$$

2.3.3 Logistic Regression

While linear regression is used to predict functions, logistic regression is used for classification. Logistic regression computes the probability the a sample can be classified as one of two classes. This is done by the logistic-sigmoid (or sigmoid, σ) function.

$$p(y = 1|x; w) = \sigma(w^T x) = \frac{1}{1 + e^{-w^T x}}$$

The normal decision boundary is 0.5 (which is not fixed). The objective is to minimize the cross-entropy loss.

$$-(y \log(\sigma(w^T x)) + (1 - y) \log(1 - \sigma(w^T x))$$

This concept can also be applied to multiclass, by using the softmax function.

$$p(y|x; w) = \text{softmax}(w^T x)$$

The objective is still to minimize the cross-entropy loss.

$$-\sum_{c=1}^{C} y \log(\text{softmax}(w^T x)_c)$$

2.3.4 Support Vector Machines (SVM)

The goal of a support vector machine is to search a function which separates two (or more) classes, such that the vector to the nearest points of two classes are maximized. These vectors are called support vectors. Instead of training a parameter w we train the parameters $\alpha_1, \ldots, \alpha_m$. We rewrite the linear regression formula:

$$w^T x + b = b + x^T w$$

w is then defined as $w = \sum_{i=1}^{m} \alpha_i x^{(i)}$. This results in the formula for the SVM.

$$b + \sum_{i=1}^{m} \alpha_i x^T x^{(i)}$$

A more general approach is to use a kernel $k(x, x^{(i)})$:

$$b + \sum_{i=1}^{m} \alpha_i k(x, x^{(i)})$$

Example for kernels are:

- Polynomial kernel: $k(x, x^{(i)}) = (x^T x^{(i)})^d$
- Gaussian kernel: $k(x, x^{(i)}) = e^{-\gamma ||x - x^{(i)}||^2}$ with $\gamma > 0$
- Define a feature function $\Phi(x)$: $k(x, x^{(i)}) = \Phi(x)^T \Phi(x^{(i)})$

2.4 Cross Validation

There are different approaches for cross validation. The simplest one is the hold-out method. For this validation we split the data into a training, validation, and test set. The training set is used to train the model, the validation set is used to measure the performance of the model, and the test set is used to determine if the model trained well after the training is finished (important: never mix these sets). A typical split is 60%/20%/20%. If the set is extremely large, we could decrease the size of the validation and test set.

A more advanced technique is k-fold cross validation. For this approach we split the data in k folds which have the same size. Then we train the model k times iterating the k folds and using one fold as the test/validation set and the other $k - 1$ folds as the training set. We can use this to determine the parameters of a model. Typical k's are 3 or the most common 5.

These two methods are often used together. We start with the hold-out method and then applying k-fold validation to the training set.

2.5 Maximum Likelihood Estimate (MLE)

The MLE is the approximation of a parameter of the distribution. We consider a training sample $x^{(1)}, \ldots, x^{(m)}$ and a model $p_{\text{model}}(x; \theta)$. We want to determine a parameter, s.t. the probability of the real data is maximized. For instance:

$$\theta_{ML} = \arg\max_{\theta} \prod_{i=1}^{m} p_{\text{model}}(x^{(i)}; \theta)$$

More formally this can be applied to all loss functions. In this case $L(\theta) = \prod_{i=1}^{m} p_{\text{model}}(x^{(i)}; \theta$. To solve this we can use the first derivative to get an extreme point and the second to determine that it is a maximum. A useful trick is to compute the logarithm instead of the product:

$$\theta_{ML} = \arg\max_{\theta} \prod_{i=1}^{m} p_{\text{model}}(x^{(i)}; \theta) = \arg\max_{\theta} \sum_{i=1}^{m} \log(p_{\text{model}}(x^{(i)}; \theta))$$

. Example: Compute the MLE of θ for $p(x;\theta) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\theta)^2}{2\sigma^2}}$.

$$\theta_{ML} = \arg\max_\theta \prod_{i=1}^{m} p_{\text{model}}(x^{(i)};\theta) = \arg\max_\theta \sum_{i=1}^{m} \log(p_{\text{model}}(x^{(i)};\theta))$$

$$= \arg\max_\theta \sum_{i=1}^{m} \log(\frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x^{(i)}-\theta)^2}{2\sigma^2}}) = \arg\max_\theta \sum_{i=1}^{m} \log(1) - \log(\sqrt{2\pi\sigma^2}) + \log(e^{-\frac{(x^{(i)}-\theta)^2}{2\sigma^2}})$$

$$= \arg\max_\theta \sum_{i=1}^{m} -\frac{1}{2}\log(2\pi\sigma^2) - \frac{(x^{(i)}-\theta)^2}{2\sigma^2}$$

We can eliminate all static terms that do not use θ:

$$= \arg\max_\theta \sum_{i=1}^{m} -(x^{(i)}-\theta)^2 = \arg\min_\theta \sum_{i=1}^{m} (x^{(i)}-\theta)^2$$

Compute the first derivative equal to 0:

$$0 \stackrel{!}{=} \frac{\delta}{\delta\theta} \sum_{i=1}^{m} (x^{(i)}-\theta)^2 = \sum_{i=1}^{m} 2\theta - 2x^{(i)} = 2m\theta - 2\sum_{i=1}^{m} x^{(i)}$$

$$\iff m\theta - \sum_{i=1}^{m} x^{(i)} = 0 \iff \theta = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}$$

Compute second derivative:

$$\frac{\delta}{\delta\theta} 2m\theta - 2\sum_{i=1}^{m} x^{(i)} = 2m > 0$$

This is a minimum, therefore $\theta_{ML} = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}$ is the MLE of θ.

2.5.1 Distribution Estimate

If the expected value or variance are unknown, they can be approximated by the Taylor expansion:

$$E[f(X)] = E(f(\theta_{ML}+(X-\theta_{ML}))) \approx E[f(\theta_{ML})+f'(\theta_{ML})(X-\theta_{ML})+\frac{1}{2}f''(\theta_{ML})(X-\theta)^2]$$

The term could be expanded by more terms of the Taylor expansion. The formal definition of the Taylor expansion is:

$$\sum_{i=1}^{\infty} \frac{1}{i!} f^{(i)}(\theta)(X-\theta)^i$$

The variance is then:

$$\text{var}[f(x)] \approx (f'(E[X]))^2\text{var}[X] \approx (f'(E[X]))^2\text{var}[X] + \frac{(f''(E[X]))^2}{2}(\text{var}[X])^2$$

2.6 Maximum a Posterior Estimate (MAP)

MAP uses a prior distribution $p(\theta)$ to accomplish the estimation. By applying Bayes rule, it follows:

$$p(\theta|X) = \frac{p(X;\theta)p(\theta)}{p(X)}$$

In most cases the denominator could be ignored because it does not contain the estimated parameter. MAP estimation's goal is then to maximize this probability. For instance:

$$\theta_{MAP} = \arg\max_{\theta} p(\theta|X) = \arg\max_{\theta} \frac{p(X;\theta)p(\theta)}{p(X)} = \arg\max_{\theta} p(X;\theta)p(\theta)$$

$$= \arg\max_{\theta} \left[\prod_{i=1}^{m} p(x^{(i)};\theta)\right] p(\theta) == \arg\max_{\theta} \left[\sum_{i=1}^{m} \log(p(x^{(i)};\theta))\right] + \log(p(\theta))$$

Example: Compute the MLE of θ for $p(x;\theta) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\theta)^2}{2\sigma^2}}$ and $p(\theta) = \frac{1}{\sqrt{2\pi\sigma_0^2}} e^{-\frac{(\theta-\theta_0)^2}{2\sigma_0^2}}$.

$$\theta_{MAP} = \arg\max_{\theta} p(\theta|X) = \arg\max_{\theta} \left[\sum_{i=1}^{m} \log(p(x^{(i)};\theta))\right] + \log(p(\theta))$$

$$= \arg\min_{\theta} \frac{1}{2\sigma^2} \left[\sum_{i=1}^{m} (x^{(i)} - \theta)^2\right] + \frac{1}{2\sigma_0^2}(\theta - \theta_0)^2$$

Set the first derivative equal to 0:

$$0 \stackrel{!}{=} \frac{\delta}{\delta\theta} \frac{1}{2\sigma^2} \left[\sum_{i=1}^{m} (x^{(i)} - \theta)^2\right] + \frac{1}{2\sigma_0^2}(\theta - \theta_0)^2 = \frac{1}{\sigma_0^2}(\theta - \theta_0) - \frac{1}{\sigma^2} \left[\sum_{i=1}^{m} (x^{(i)} - \theta)\right]$$

$$= \frac{1}{\sigma_0^2}(\theta - \theta_0) - \frac{1}{\sigma^2} \left[\sum_{i=1}^{m} x^{(i)}\right] + \frac{m}{\sigma^2}\theta = \frac{1}{\sigma_0^2}\theta - \frac{1}{\sigma_0^2}\theta_0 - \frac{1}{\sigma^2} \left[\sum_{i=1}^{m} x^{(i)}\right] + \frac{m}{\sigma^2}\theta$$

$$\iff \frac{1}{\sigma_0^2}\theta + \frac{m}{\sigma^2}\theta = \frac{1}{\sigma_0^2}\theta_0 + \frac{1}{\sigma^2} \left[\sum_{i=1}^{m} x^{(i)}\right]$$

$$\iff \frac{\sigma^2 + m\sigma_0^2}{\sigma_0^2\sigma^2}\theta = \frac{1}{\sigma_0^2}\theta_0 + \frac{1}{\sigma^2} \left[\sum_{i=1}^{m} x^{(i)}\right] \iff \theta = \frac{\sigma^2}{\sigma^2 + m\sigma_0^2}\theta_0 + \frac{\sigma_0^2}{\sigma^2 + m\sigma_0^2} \left[\sum_{i=1}^{m} x^{(i)}\right]$$

For MAP estimation with this Gaussian model the following holds:

	$m = 0$	$m \to \infty$
θ_{MAP}	θ_0	θ_{ML}
σ^2_{MAP}	σ_0^2	0

To be a prior, p has to be a probability density function. This means that the area under the function is 1.

$$\int_{-\infty}^{\infty} p(\theta)\delta\theta \stackrel{!}{=} 1$$

To accomplish this we could use a normalization factor for the prior. Example: Compute the normalization factor c, s.t. $f(x) = \begin{cases} cx & 0 \leq x \leq 1 \\ 0 & \text{otherwise} \end{cases}$ is a probability density function.

$$\int_{-\infty}^{\infty} f(x)\delta x = \int_{0}^{1} f(x)\delta x = [\frac{c}{2}x^2]_0^1 = \frac{c}{2} = 1 \iff c = 2$$

2.7 Lagrange Multiplier

Sometime we could encounter constraint objective where we have to find minima and maxima under certain circumstances. The objective is then formalized as:

$$\text{maximize/minimize} f(x, y)$$

$$\text{subject to: } g(x, y) = c$$

The Lagrange multiplier is then:

$$\mathcal{L}(x, y, \lambda) = f(x, y) - \lambda(g(x, y) - c)$$

If the objective is constrained to several subject, we simply add more multiplier:

$$\text{maximize/minimize} f(x, y)$$

$$\text{subject to: } g(x, y) = c, h(x, y) = d$$

The Lagrange multiplier is then:

$$\mathcal{L}(x, y, \lambda_1, \lambda_2) = f(x, y) - \lambda_1(g(x, y) - c) - \lambda_2(h(x, y) - d)$$

We can then minimize or maximize this multiplier. Example: Find the critical point of $f(x, y) = x^3 + xy^2$ under the constrained $2x + y^2 = 2$.

The problem is:

$$\nabla_{x,y} f(x, y) = x^3 + xy^2 \stackrel{!}{=} 0$$

$$\text{subject to: } 2x + y^2 = 2$$

The Lagrange multiplier is:

$$\mathcal{L}(x, y, \lambda) = x^3 + xy^2 - 2\lambda x + \lambda y^2 - 2\lambda$$

Compute the derivatives:

I.

$$\frac{\delta}{\delta x}\mathcal{L}(x,y,\lambda) = 3x^2 + y^2 - 2\lambda \stackrel{!}{=} 0$$

II.

$$\frac{\delta}{\delta y}\mathcal{L}(x,y,\lambda) = 2xy - 2\lambda y \stackrel{!}{=} 0$$

III.

$$\frac{\delta}{\delta \lambda}\mathcal{L}(x,y,\lambda) = 2x + y^2 - 2 \stackrel{!}{=} 0 \iff x = 1 - S\frac{y^2}{2}$$

III. in II. and $y \neq 0$:

$$2\left(1 - \frac{y^2}{2}\right)y - 2\lambda y = 0 \iff \lambda 1 - \frac{y^2}{2}$$

In I.:

$$3\left(1 - \frac{y^2}{2}\right)^2 + y^2 - 2\left(1 - \frac{y^2}{2}\right) \iff \frac{3}{4}y^4 - y^2 + 1 = 0$$

This does not have a solution, because it is always greater than 0. Left is $y = 0$. $y = 0$ in III.:

$$2x - 2 \iff x = 1$$

In I.:

$$3 - 2\lambda = 0 \iff \lambda = \frac{3}{2}$$

In II.:

$$0 = 0$$

$(x, y) = (1, 0)$ is the only critical point of $f(x, y)$ under the constraint $2x + y^2 = 2$.

3 Principal Component Analysis (PCA)

PCA is a technique for dimensional reduction. It is used to compress/encode data, to remove less relevant dimensions by projection, get a two dimensional representation of multidimensional set of points. There are several objectives for PCA:

- Let $U_n = c + V_n = c + \left\{ \sum_{j=1}^{n} \alpha_j u_j | u_j{}_{j=1}^{n} \text{ONS}, c \in \mathbb{R}^d, \alpha_j \in \mathbb{R} \right\}$ where $u_j{}_{j=1}^{n}$ is an orthonormal system.

$$\underset{Z_i \in V_n, c \in \mathbb{R}}{\arg\min} \frac{1}{m} \sum_{i=1}^{m} ||Z_i + c - X_i||_2^2$$

This means to minimize the MSE of the projection and the original points. We could get rid of the c in the objective. It could be seen as the origin of U_n.

$$\nabla_c \frac{1}{m} \sum_{i=1}^{m} ||Z_i + c - X_i||_2^2 = 2 \sum_{i=1}^{m} (Z_i - X_i) + 2mc \iff c = \frac{1}{m} \sum_{i=1}^{m} (X_i - Z_i)$$

Since the optimal c depends on Z_i:

$$\underset{Z_i \in V_n, c \in \mathbb{R}}{\arg\min} \frac{1}{m} \sum_{i=1}^{m} ||Z_i + c - X_i||_2^2 = \underset{Z_i \in V_n}{\arg\min} \frac{1}{m} \sum_{i=1}^{m} ||Z_i - (X_i - \frac{1}{m} \sum_{j=1}^{m} X_j)||_2^2$$

$$= \underset{Z_i \in V_n}{\arg\min} \frac{1}{m} \sum_{i=1}^{m} ||Z_i - (X_i - \bar{X})||_2^2 = \underset{Z_i \in V_n}{\arg\min} \frac{1}{m} \sum_{i=1}^{m} ||Z_i - \tilde{X}_i||_2^2$$

Let $Z_i = P\tilde{X}_i$ the orthogonal projection of $\tilde{X}_i$. It holds that,

$$||Z_i - \tilde{X}_i||_2^2 = ||P\tilde{X}_i - \tilde{X}_i||_2^2 = ||P\tilde{X}_i - P\tilde{X}_i||_2^2 + ||P\tilde{X}_i - \tilde{X}_i||_2^2$$

$$= ||Z_i - P\tilde{X}_i||_2^2 + ||P\tilde{X}_i - \tilde{X}_i||_2^2$$

$$\underset{Z_i \in V_n}{\arg\min} \frac{1}{m} \sum_{i=1}^{m} ||Z_i - \tilde{X}_i||_2^2 = \underset{P \in \mathbb{R}^d \to \mathbb{R}^d}{\arg\min} \frac{1}{m} \sum_{i=1}^{m} ||P\tilde{X}_i - \tilde{X}_i||_2^2 = \underset{P \in \mathbb{R}^d \to \mathbb{R}^d}{\arg\min} \frac{1}{m} \sum_{i=1}^{m} ||(P - \mathbb{1})\tilde{X}_i||_2^2$$

$$= \underset{P \in \mathbb{R}^d \to \mathbb{R}^d}{\arg\min} \frac{1}{m} \sum_{i=1}^{m} \tilde{X}_i^T (P - \mathbb{1})^T (P - \mathbb{1}) \tilde{X}_i = \underset{P \in \mathbb{R}^d \to \mathbb{R}^d}{\arg\min} \frac{1}{m} \sum_{i=1}^{m} \tilde{X}_i^T (P^T P - P - P^T + \mathbb{1}) \tilde{X}_i$$

Because P is a orthogonal projection, $P = P^T$ and $P^2 = P$:

$$= \underset{P \in \mathbb{R}^d \to \mathbb{R}^d}{\arg\min} \frac{1}{m} \sum_{i=1}^{m} \tilde{X}_i^T (\mathbb{1} - P) \tilde{X}_i = \underset{P \in \mathbb{R}^d \to \mathbb{R}^d}{\arg\min} \frac{1}{m} \sum_{i=1}^{m} \tilde{X}_i^T \tilde{X}_i - \tilde{X}_i^T P \tilde{X}_i$$

$$= \underset{\{u_i\}_{i=1}^{m} ONS}{\arg\min} \frac{1}{m} \sum_{i=1}^{m} \tilde{X}_i^T \tilde{X}_i - u_i^T \left(\sum_{j=1}^{m} \tilde{X}_j^T \tilde{X}_j \right) u_i$$

Introduce $C = \sum_{i=1}^{m} \tilde{X}_i^T \tilde{X}_i$. The objective is the minimized by using the n largest eigenvectors of C:

$$\underset{\{u_i\}_{i=1}^{m} ONS}{\arg\min} \; C - \frac{1}{m} \sum_{i=1}^{m} u_i^T C u_i$$

- Another objective is to minimize the reconstruction error using the projection D.

$$\underset{D \in \mathbb{R}^{n \times l}, l \leq n}{\arg\min} \frac{1}{m} \sum_{i=1}^{m} ||X_i - DD^T X_i||_2$$

This is the one needed in NNIA.

PCA is an algorithm which following these steps:

1. Compute the covariance matrix of X. That is $B = X - \bar{X}$, where $\bar{X}$ is the mean of the columns/features, and $C = \frac{1}{m} B^T B$
2. Apply eigendecomposition to C which yield $\lambda_1, \ldots, \lambda_n$ and $v_1, \ldots, v_n$. Select the l eigenvectors $v_1', \ldots, v_l'$ which have the greatest eigenvalues (sorted for large to small).
3. Set $D = \begin{pmatrix} | & & | \\ v_1 & \ldots & v_l \\ | & & | \end{pmatrix}$
4. Compute $P = BD$.

To clarify: $B \in \mathbb{R}^{m \times n}$, $C \in \mathbb{R}^{n \times n}$, $D \in \mathbb{R}^{n \times l}$, and $P \in \mathbb{R}^{m \times l}$.
PCA computes the subspace which contains most of the variance.

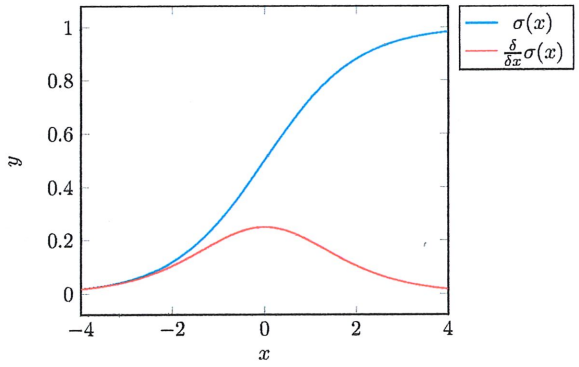
3.1 Kernel PCA

Instead of using normal PCA we could also use a PCA with kernels. For them we do not compute the eigendecomposition for the covariance matrix, but instead for a kernel matrix:

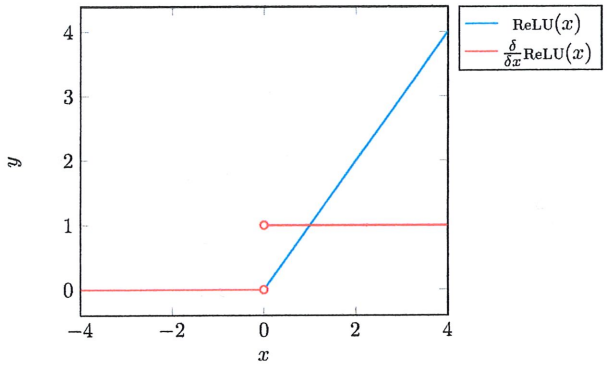
$$C = \frac{1}{m} \sum_{i=1}^{m} \Phi(X_i) \Phi(X_i)^T = \frac{1}{m} \Phi(X)^T \Phi(X)$$

4 Activation Functions

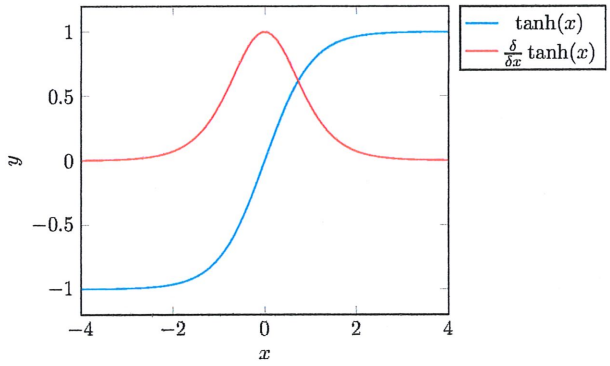
Activation: Sigmoid (σ)
Formula: $\sigma(x) = \frac{1}{1+e^{-x}}$
Derivative: $\frac{\delta}{\delta x}\sigma(x) = \sigma(x) - \sigma^2(x) = \sigma(x)(1-\sigma(x))$
$\frac{\delta}{\delta x}\sigma(x) = -\frac{1}{(1+e^{-x})^2}(-e^{-x}) = \frac{e^{-x}}{(1+e^{-x})^2} = \frac{1+e^{-x}-1}{(1+e^{-x})^2}$
$= \frac{1+e^{-x}}{(1+e^{-x})^2} - \frac{1}{(1+e^{-x})^2} = \frac{1}{1+e^{-x}} - \frac{1}{(1+e^{-x})^2} = \sigma(x) - \sigma^2(x)$



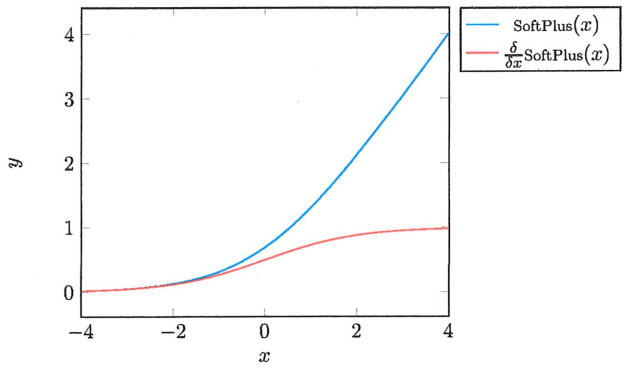
Activation: ReLU
Formula: $\text{ReLU}(x) = \max(0, x)$
Derivative: $\frac{\delta}{\delta x}\text{ReLU}(x) = \begin{cases} 1 & x > 0 \\ 0 & x < 0 \end{cases}$, $\frac{\delta}{\delta x}x = 1$ and $\frac{\delta}{\delta x}0 = 0$



Activation: tanh
Formula: $\tanh(x) = 2\sigma(2x) - 1$
Derivative: $\frac{\delta}{\delta x}\tanh(x) = 1 - \tanh^2(x)$
$\frac{\delta}{\delta x}\tanh(x) = 2(\sigma(2x) - \sigma^2(2x)) * 2 = 4\sigma(2x) - 4\sigma^2(2x)$
$= 1 - (4\sigma^2(2x) - 4\sigma^2(2x) + 1) = 1 - (2\sigma(2x) - 1)^2 = 1 - \tanh^2(x)$



Activation: SoftPlus
Formula: $\text{SoftPlus}(x) = \log(1 + e^x)$
Derivative: $\frac{\delta}{\delta x}\text{SoftPlus}(x) = \sigma(x)$
$\frac{\delta}{\delta x}\text{SoftPlus}(x) = \frac{e^x}{1+e^x} = \frac{e^x e^{-x}}{e^{-x}+e^x e^{-x}} = \frac{e^{x-x}}{e^{x-x}+e^{-x}} = \frac{1}{1+e^{-x}} = \sigma(x)$



Activation: Softmax

Formula: $\text{softmax}(x)_i = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$

This formula can have under- or overflows because of the large exponents. A stable version is: $\text{softmax}(x)_i = \frac{e^{x_i - \max(x)}}{\sum_{j=1}^{n} e^{x_j - \max(x)}}$

Derivative: $\frac{\delta}{\delta x_k}\text{softmax}(x)_i = \begin{cases} \text{softmax}(x)_i(1 - \text{softmax}(x_i)) & i = k \\ -\text{softmax}(x)_i\text{softmax}(x)_k & i \neq k \end{cases}$

If $i = k$: $\frac{\delta}{\delta x_k}\text{softmax}(x)_i = \frac{e^{x_i}\sum_{j=1}^{n} e^{x_j} - e^{x_i}e^{x_i}}{\left(\sum_{j=1}^{n} e^{x_j}\right)^2} = \frac{e^{x_i}\sum_{j=1}^{n} e^{x_j}}{\left(\sum_{j=1}^{n} e^{x_j}\right)^2} - \frac{e^{x_i}e^{x_i}}{\left(\sum_{j=1}^{n} e^{x_j}\right)^2}$

$= \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}} - \left(\frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}\right)^2 = \text{softmax}(x)_i - \text{softmax}^2(x)_i$

$= \text{softmax}(x)_i(1 - \text{softmax}(x)_i)$

If $i \neq k$: $\frac{\delta}{\delta x_k}\text{softmax}(x)_i = \frac{0 * \sum_{j=1}^{n} e^{x_j} - e^{x_i}e^{x_k}}{\left(\sum_{j=1}^{n} e^{x_j}\right)^2} = -\frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}\frac{e^{x_k}}{\sum_{j=1}^{n} e^{x_j}}$

$= -\text{softmax}(x)_i\text{softmax}(x)_k)$

The softmax function has an easy derivative. It is mostly used as the output layer for multiclass neural networks. Therefore, it is common to use it in combination with cross entropy loss. In this case the derivative $\frac{\delta}{\delta x_i}L(y, \text{softmax}(x)) = \text{softmax}(x)_i - y_i$

$\frac{\delta}{\delta x_i}L(y, \text{softmax}(x)) = \frac{\delta}{\delta x_i} - \sum_{c=1}^{m} y_c \log(\text{softmax}(x)_c)$

$= -\sum_{c=1}^{m} y_c \frac{\delta}{\delta x_i}\log(\text{softmax}(x)_c) = -\sum_{c=1}^{m} y_c \frac{1}{\text{softmax}(x)_c}\frac{\delta}{\delta x_i}\text{softmax}(x)_c$

$= -\frac{y_i}{\text{softmax}(x)_i}\frac{\delta}{\delta x_i}\text{softmax}(x)_i - \sum_{c \neq i}^{m} \frac{y_c}{\text{softmax}(x)_c}\frac{\delta}{\delta x_i}\text{softmax}(x)_c$

$= -\frac{y_i}{\text{softmax}(x)_i}\text{softmax}(x)_i(1 - \text{softmax}(x)_i) + \sum_{c \neq i}^{m} \frac{y_c}{\text{softmax}(x)_c}\text{softmax}(x)_i\text{softmax}(x)_c$

$= -y_i(1 - \text{softmax}(x)_i) + \sum_{c \neq i}^{m} y_c\text{softmax}(x)_i$

$= -y_i + y_i\text{softmax}(x)_i) + \text{softmax}(x)_i \sum_{c \neq i}^{m} y_c$

$= -y_i + \text{softmax}(x)_i \sum_{c=1}^{m} y_c = \text{softmax}(x)_i - y_i$

4.1 Gradient Problems

Vanishing and Exploding Gradients Some activation functions have always a very small gradient, for instance $\sigma(x)$, if the gradient is multiplied several times (backpropagation) it starts to vanish, i.e. always approximately equal to 0. However, if the gradient is always too large it starts to explode, i.e. always a tremendous number.

Dying Gradients Some activation functions have a gradient of 0, for instance $\text{ReLU}(x)$. This means if the gradient reaches this point it is most likely to not recover and, therefore, it dies. This is bad for optimization because it relies on varying gradients.

5 Deep Neural Networks

A neural network consists of layers of neurons. Each layer have an activation function. In a fully-connected neural network all neurons of a layer are connected to all neurons of the previous layer. We train a network by using weights W and biases b. The result of a layer i could be computed by:

$$a_i = W_i^T h_{i-1} + b_i$$

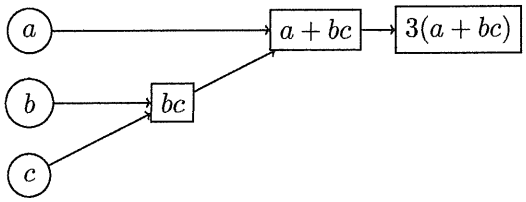
$$h_i = h(a_i)$$

where $h()$ is the activation function. The neural network is evaluated using a loss function. Most likely the cross-entropy is used.

5.1 Computation Graph

Computation graph give a simple introduction on how neural networks work. We draw a computation graph by extracting all simple operations from a term.

Example: Draw the computation graph of $3(a + bc)$.



The general idea is to use a forward pass to compute the result and a backward pass to determine the impact of each parameter on the result. For the forward pass we follow the edges and compute for each node its result. For the backward pass we compute the derivative of the results using the chain rule. Figure 5.1 demonstrates this computation.

5.2 Forward Propagation

The forward computation is just a forward pass on the neural network. The result is then used to compute the loss.

5.3 Backward Propagation

The back propagation is a backward pass to compute the impact of all W_i and b_i onto the loss of the neural network. Then an optimization algorithm is used to update the

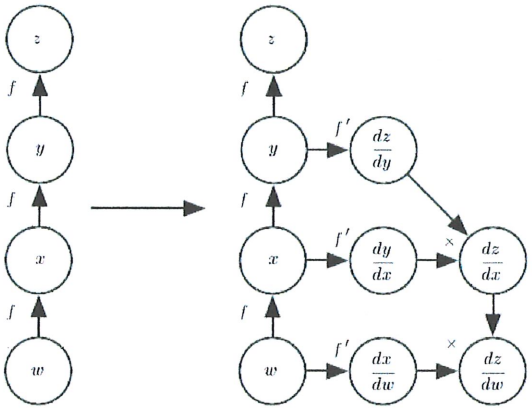


Figure 5.1: Compute the derivative based on a computation graph.

weights and biases. The general formula to update a weight or bias a is:

$$\frac{\delta(L)}{\delta a} = \sum_{i..j \in \text{Path}(L,a)} \frac{\delta L}{\delta u^{(i)}} \cdots \frac{\delta u^{(j)}}{\delta a}$$

Figure 5.2 demonstrates the back propagation on a neural network.

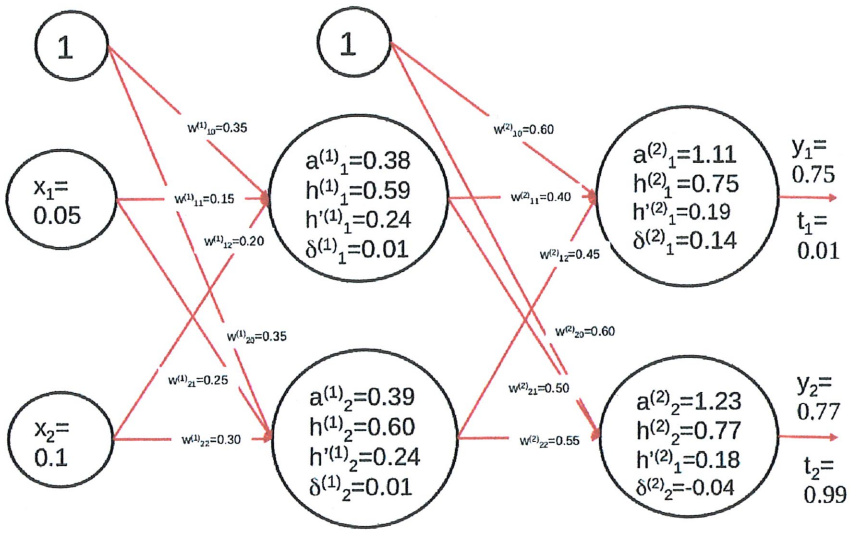


Figure 5.2: The back propagation on a neural network.

6 Optimization

6.1 Problems for Optimization

There are several problems which optimizations could encounter:

Local Minima The optimizations needs to skip local minima (gradient = 0) in order to converge t0 a global minimum.

Plateaus and Saddle Pints The optimizations needs to skip plateaus and saddle points (gradient = 0) in order to converge to a global minimum.

Cliffs and Exploding Gradients The optimizations needs to handle non-linearity and the resulting rise to high gradients. This problem could be fixed by clipping the gradient.

Vanishing and Exploding Gradients If the gradient vanishes it has no impact on the optimization if it explodes the impact is too high.

Initialization If the optimizations is not initialized correctly it could fail to find a solution at all.

6.2 Gradient Descent (GD)

For the normal gradient descent we compute the average gradient over all training examples.

```
def gd(θ, ϵ):
    while stopping criterion not met do
        g = (1/m) ∇_θ Σ_{i=1}^{m} L(f(x^(i); θ), y^(i))  (Compute average gradient)
        θ = θ − ϵg  (update)
    end while
```

The problem with GD is that we need to compute the gradient for all samples, i.e. it may take longer to converge to a minimum. Additionally it is likely to fall in a local minimum.

This algorithm is also called batch GD (BGD).

6.2.1 Stochastic GD (SGD)

We could use instead a stochastic GD where we apply the updates for each training example. However, this could not find a minimum at all, because of this we use a learning rate decay, i.e. the learning rate gets smaller over time.

```
def sgd(θ, τ, ϵ0, ϵτ):
        k = 0
    while stopping criterion not met do
        x, y = random sample of a corresponding pair in (X, Y)
        g = ∇θ L(f(x; θ), y) (Compute gradient)
        ϵk = (1 − k/τ)ϵ0 + (k/τ)ϵτ (learning rate decay)
        θ = θ − ϵk g (update)
        k++
    end while
```

We could also implement this algorithm, s.t. it iterates all examples. In this case a shuffle of the examples is needed.

In NNIA MBGD is called SGD.

6.2.2 Mini-Batch GD (MBGD)

A combination of both is the mini-batch GD. Instead of training with all or one example, we train with a batch of samples which has size m'. If $m' = 1$ the algorithm is equivalent to SGD. If $m' = m$ the algorithm is equivalent to BGD.

```
def mbgd(θ, m', τ, ϵ0, ϵτ):
    k = 0
    while stopping criterion not met do
        Sample a minimatch of corresponding pairs in (X, Y) of size m'
        g = (1/m') ∇θ Σ_{i=1}^{m'} L(f(x^(i); θ), y^(i)) (Compute average gradient)
        ϵk = (1 − k/τ)ϵ0 + (k/τ)ϵτ (learning rate decay)
        θ = θ − ϵk g (update)
        k++
    end while
```

We could also implement this algorithm, s.t. we create batches at the beginning of each iteration and iterate these batches. In this case a shuffle of the examples is needed.

In NNIA MBGD is called SGD.

6.2.3 Momentum

However, there is still the possibility that these optimizations will converge to a local minimum or stop at a saddle point. In order to decrease this possibility we use momentum for our algorithms. The momentum keeps track of previous updates. The momentum should be less than 1 and greater or equal to 0. If the momentum is equal to 0, the algorithm is equivalent to the one without momentum.

add paremeters v (velocity), and α (momentum)
change the update to:
$v = \alpha v - \epsilon g$
$\theta = \theta + v$

6.3 Adaptive Learning Rates

6.3.1 Newton

Instead of using a fixed learning rate we could also use the second derivative as an implicit learning rate. However, if the algorithm gets to a saddle point, plateau, or an intermediate point it fails to find a solution because the second derivative is 0.

$g = f'(x)$
$\theta = \theta - \frac{1}{f''(x)}g$

6.3.2 AdaGrad

AdaGrad uses an accumulated square regularization for the learning rate.

def adagrad(θ, m', ϵ, $\delta = 10^{-7}$):
    r = 0
    while stopping criterion not met do
        Sample a minimatch of corresponding pairs in (X,Y) of size m'
        $g = \frac{1}{m'}\nabla_\theta \sum_{i=1}^{m'} L(f(x^{(i)};\theta), y^{(i)})$ (Compute average gradient)
        $r = r + g \circ g$ (accumulate squared gradient)
        $\Delta\theta = -\frac{\epsilon}{\delta+\sqrt{r}} \circ g$ (compute update)
        $\theta = \theta + \Delta\theta$ (update)
    end while

6.3.3 RMSProp

RMS uses an accumulated square regularization for the learning rate with decay ρ.

def rmsprop(θ, m', ϵ, $\delta = 10^{-6}$, ρ):
    r = 0
    while stopping criterion not met do
        Sample a minimatch of corresponding pairs in (X,Y) of size m'
        $g = \frac{1}{m'}\nabla_\theta \sum_{i=1}^{m'} L(f(x^{(i)};\theta), y^{(i)})$ (Compute average gradient)
        $r = \rho r + (1-\rho) g \circ g$ (accumulate squared gradient with decay)
        $\Delta\theta = -\frac{\epsilon}{\sqrt{\delta+r}} \circ g$ (compute update)
        $\theta = \theta + \Delta\theta$ (update)
    end while

7 Regularization

In the following we show some regularization methods for the training of neural networks. Such methods could also be combined.

7.1 Normalization

Sometimes features have different ranges and scalings. this could influence the learning, s.t. the result will be distorted or the training needs a long time to find a good solution.

Therefore, we want to normalize the features by making all features 0 centered and in the range $[-1, 1]$

$\mu = \frac{1}{m}\sum_{i=1}^{m} x_i$ (compute mean)
$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu)^2$ (compute empirical variance)
$\hat{x}_i = \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}}$ (normalize data)

7.1.1 Batch Normalization

With this we can normalize the input of neural network. However, it could also help to normalize between each layer of the neural network. Because we do not know the global mean or variance during the training with mini-batches, we could use batch normalization.

$\mu_{\mathcal{B}} = \frac{1}{m}\sum_{i=1}^{m} x_i$ (compute mean of the mini-batch)
$\sigma^2_{\mathcal{B}} = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2$ (compute empirical variance of the minibatch)
$\hat{x}_i = \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}}$ (normalize the mini-batch)
$y_i = \gamma\hat{x}_i + \beta \equiv BN_{\gamma,\beta}(x_i)$ (scale and shift)

This approach introduce two new hyper-parameters which should also be trained. These two are trained, s.t. that the means and variance over all samples are 0 and 1. If γ and β were not included SGD (or another approach) would train the weights itself to satisfy this requirement, but this would lead to distorted weights.

We can also use batch normalization for the input for instance in online learning where we get a stream of examples.

Batch normalization allows us to use higher learning rates because no activation function is gone to have a really high or low output. Additionally it reduces overfitting because it has a regularization effect on the weights. Because the outputs are small the weight will get small to. Moreover it adds noise to training which helps also to prevent overfitting.

In conclusion batch normalization helps us to train a network faster and reduces overfitting.

7.2 Initialization

It is hard to initialize the weights with the best parameters because the optimization is not fully understood. It is important that they break the symmetry. Typical a uniform or Gaussian distribution is used. For a fully connected layer with m inputs and n outputs the interval $\left[-\frac{6}{\sqrt{m+n}}, \frac{6}{\sqrt{m+n}}\right]$ is used. Another approach is sparse initialization, where each layer is initialized with k non-zero weights. However, it is important to have a low variance for the initialization of the weights.

For the biases a initialization with 0 is good. For ReLU it could be better to initialize with 0.1

It could also be better to pre-train each layer on its own.

7.3 Parameter Norm Penalties

Parameter norm penalties extend the loss function by a norm penalty $\Omega(\theta)$ to regularize the objective:

$$\hat{L}(\theta; X, y) = L(\theta; X, y) + \lambda\Omega(\theta)$$

Typically only the weights are regularized not the biases. This regularized loss let the network prefer smaller over larger weights and prevents, therefore, overfitting. This is because larger weights mean a larger variance and, therefore, the model is more likely to overfit.

7.3.1 $\mathcal{L}^2$-Regularization

$\mathcal{L}^2$-regularization is the most common norm penalty:

$$\Omega(w) = \frac{1}{2}||w||_2^2$$

This results in the corresponding gradient:

$$\nabla_w \hat{L}(w; X, y) = \lambda w + \nabla_w L(w; X, y)$$

7.3.2 $\mathcal{L}^1$-Regularization

$\mathcal{L}^1$-regularization is another common norm penalty:

$$\Omega(w) = ||w||_1 = \sum_i |w_i|$$

This results in the corresponding gradient:

$$\nabla_w \hat{L}(w; X, y) = \lambda \text{sign}(w) + \nabla_w L(w; X, y)$$

7.3.3 Tangent Prop

The tangent prop algorithm is a penalty which uses the tangent of the activation function to reduce the variance:

$$\Omega(w) = \sum_i (\nabla_x f(x)^T v(i))$$

The effectiveness depends here on the type of the activation function f.

7.4 Dataset Augmentation

We can manipulate existing data to create new fake data. Common techniques are:

- The shift of audio by a few samples.
- The shift, flip, or rotation of training images (caution: rotation by 180° or flips could cause confusion, e.g. 6 and 9).
- Add noise to data (e.g. Gaussian noise to images).
- Multiply by noise (could be equivalent to dropout).

All these techniques create more artificial data. More data means that the model is more likely to not overfit. However, the created data needs to be meaningful, s.t. the network learns the right task and not a completely different one.

7.5 Early Stopping

If we train the model long enough it may occur that the model starts to overfit because it learns to represent the data. Because of this it could help to stop the training before it actually terminates. This is called early stopping. A common early stopping criterion is that we stop the training as soon as the validation error would increase again. Another approach is to save a copy of the network for the best scored validation error.

Early stopping and $\mathcal{L}^2$-regularization can have the same impact if $\lambda \approx \frac{1}{\tau\epsilon}$. Where τ is the iteration at which the early stopping would occur and ϵ is the learning rate.

7.6 Parameter Tying and Parameter Sharing

If we want to train two task we could tie the parameters of these tasks together for instance using $\mathcal{L}^2$-regularization:

$$\Omega(w_1, w_2) = ||w_1 - w_2||_2^2$$

This would reduce the variance of both tasks.

Moreover parameter sharing would set the parameters equal. This would lead to significant memory reduction because less parameters need to be stored and updated. This does not mean that this is always a great idea. It may lead all networks to fail their tasks.

7.7 Ensemble Methods

7.7.1 Bootstrap Aggregating (Bagging)

For bagging we create m new training sets by sampling uniformly from the original training data. Then we train a new model for each of the m training sets. The result is than computed by average (regression) or majority vote (classification). Each of the models is than able to learn simpler differences between its examples. This method increases the robustness and since the capacity for each model is smaller it decreases the variance, and as a result it decreases the probability that the model overfits. For classifiers if the trained models are completely independent, the error decrease by $\frac{1}{k}$ where k is the number of independent classes.

7.7.2 Dropout

Because we have to create m new training sets bagging is really expensive. However, we could approximate it using dropout. For dropout we do not create new sets but instead excluding random neurons from the learning. This reduces co-dependencies between neurons which reduces the variance and the probability of overfitting.

Dropout could be divided in training and testing. In the training phase we ignore a neuron with a probability of p. In the testing phase we use all neurons, to apply everything which is learned. However, we need to scale the results by the factor p because otherwise the results would be distorted. We want to match the expected testing results to the actual training results, therefore, this is needed.

This is hard to implement because we need to decide if we need to scale or not in the forward pass, therefore we need to jump to different code blocks. Because of this a common method is to implement inverted dropout. For inverted dropout we scale the parameters during the training by the keep probability q ($q = 1 - p$), s.t. the network can implicitly learn the scaling.

8 Recurrent Neural Networks (RNN)

The neural networks discussed so far take only one example and produce one output. However, we could also get a sequence of data and produce one or multiple outputs to it, for instance a sequence of integers where we should guess the next integer.

In the case of sequence we could share the parameters if all positions of the sequence behave in the same way.

The recurrence of these network can occur at any time for instance after the activation, before, after the computation of the loss, etc. The recurrence is then linked to the next element of the sequence. Such an RNN could compute the following:

$a^{(t)} = b + Wh^{(t-1)} + Ux^{(t)}$
$h^{(t)} = \sigma(a^{(t)})$
$o^{(t)} = c + Vh^{(t)}$
$y = \text{softmax}(o^{(t_{\max})})$

Basically RNNs are small neural networks which contain loops.

8.1 Bidirectional RNNs

Instead of having only a connection to the next element RNNs could also receive information from the next one. This is called a bidirectional RNN. The idea is that the past as well as the future could influence the present.

8.2 Long Short-Term Memory (LSTM)

A typical RNN has the problem that for each forward pass only the information of the previous element (or the few previous elements) of the sequence is available. The greater the gap between to elements the less they could influence each other and thus the RNN will be unable to connect such information. This could be solved by giving each RNN cell long-term dependencies.

The standard for accomplishing this are LSTM cells. These cell keep track of a long-term dependency to previous cells.

Figure 8.1 demonstrates the difference between RNN and LSTM cells. The LSTM cell uses in addition of the h lane the C lane which is the long-term dependency.

Figure 8.2 demonstrates a forward pass through an LSTM cell.

A problem for LSTM cells are that they are hard to train. We could not apply a simple back propagation because we could have different results. Additionally if we propagate back through the cells we could encounter vanishing or exploding gradients.

These RNN and LSTM cells are often used in language processing.

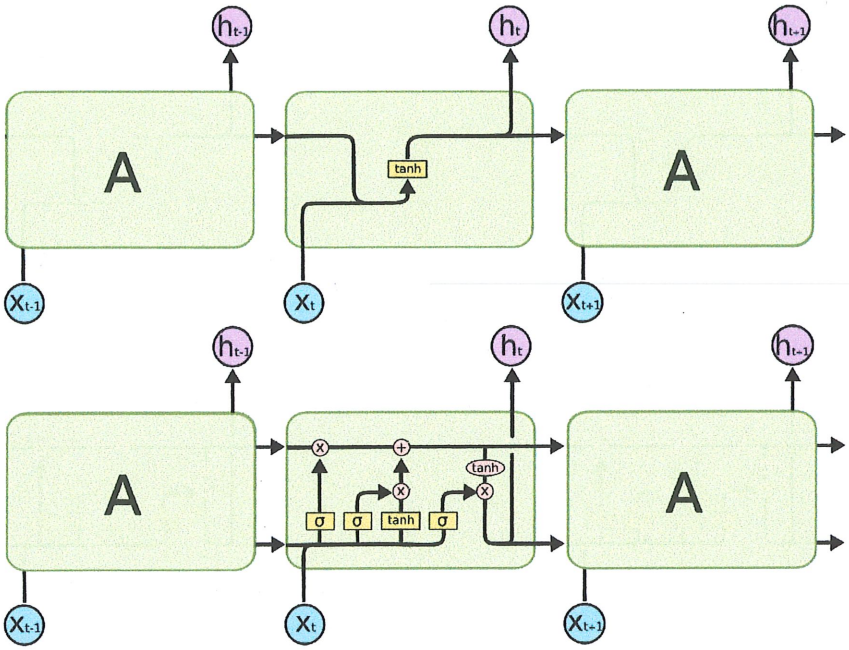


Figure 8.1: Difference between a simple RNN cell (top) and a basic LSTM cell (bottom).

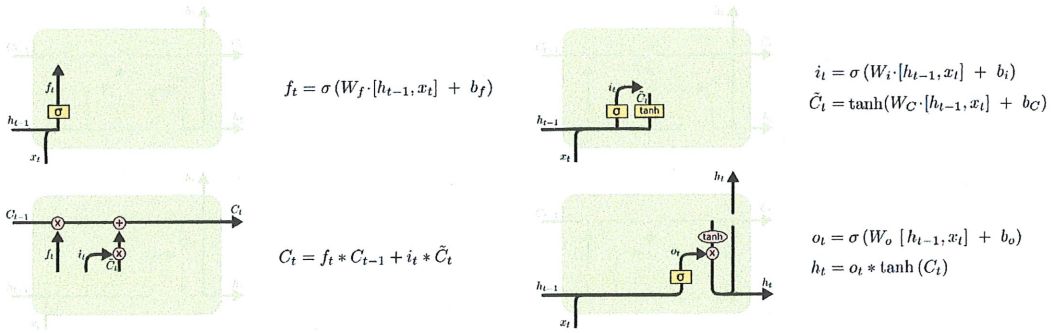


Figure 8.2: Forward pass through an LSTM cell.