Deep Learning

M. Mettauer

October 25, 2017

Contents

1 Machine Learning Basics

1.1 Learning Algorithms p. 96

Mitchell (1997) provides the definition "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P , if its performance at tasks in T , as measured by P ,improves with experience E ."

1.2 The Task, T p. 96

Classification In this type of task, the computer program is asked to specify which of k categories some input belongs to.

Classification with missing inputs Classification becomes more challenging if the computer program is not guaranteed that every measurement in its input vector will always be provided. When some of the inputs may be missing, the learning algorithm must learn a set of functions (example: medical diagnosis)

Regression In this type of task, the computer program is asked to predict a numerical value given some input.

Transcription In this type of task, the machine learning system is asked to observe a relatively unstructured representation of some kind of data and transcribe it into discrete, textual form. For example, in optical character recognition.

Machine translation In a machine translation task, the input already consists of a sequence of symbols in some language, and the computer program must convert this into a sequence of symbols in another language.

Structured output Structured output tasks involve any task where the output is a vector (or other data structure containing multiple values) with important relationships between the different elements. Example: Parsing, mapping a natural language sentence into a tree that describes its grammatical structure and tagging nodes of the trees as being verbs, nouns, or adverbs, and so on.

Anomaly detection In this type of task, the computer program sifts through a set of events or objects, and flags some of them as being unusual or atypical. An example of an anomaly detection task is credit card fraud detection.

Synthesis and sampling In this type of task, the machine learning algorithm is asked to generate new examples that are similar to those in the training data. Synthesis and sampling via machine learning can be useful for media applications where it can be expensive or boring for an artist to generate large volumes of content by hand (For example, video games).

Imputation of missing values The algorithm must provide a prediction of the values of the missing entries.

Denoising The learner must predict the clean example $\mathbf{x}$ from its corrupted version $\tilde{\mathbf{x}}$.

Density estimation or probability mass function estimation In the density estimation problem, the machine learning algorithm is asked to learn a function $p_{model}(x) : \mathbb{R}^n \rightarrow \mathbb{R}$, where $p_{model}(x)$ can be interpreted as a probability density function (if x is continuous) or a probability mass function (if x is discrete) on the space that the examples were drawn from.

1.3 The Performance Measure, P p. 100

Usually this performance measure P is specific to the task T being carried out by the system. For tasks such as classification, classification with missing inputs, and transcription, we often measure the accuracy of the model.

1.4 The Experience, E p. 104

Machine learning algorithms can be broadly categorized as unsupervised or supervised by what kind of experience they are allowed to have during the learning process. Most of the learning algorithms can be understood as being allowed to experience an entire dataset. Supervised: We have labels. Unsupervised: We don't have labels, instead we try to learn the entire probability distribution that generated a dataset.

1.5 Capacity, Overfitting and Underfitting p. 107

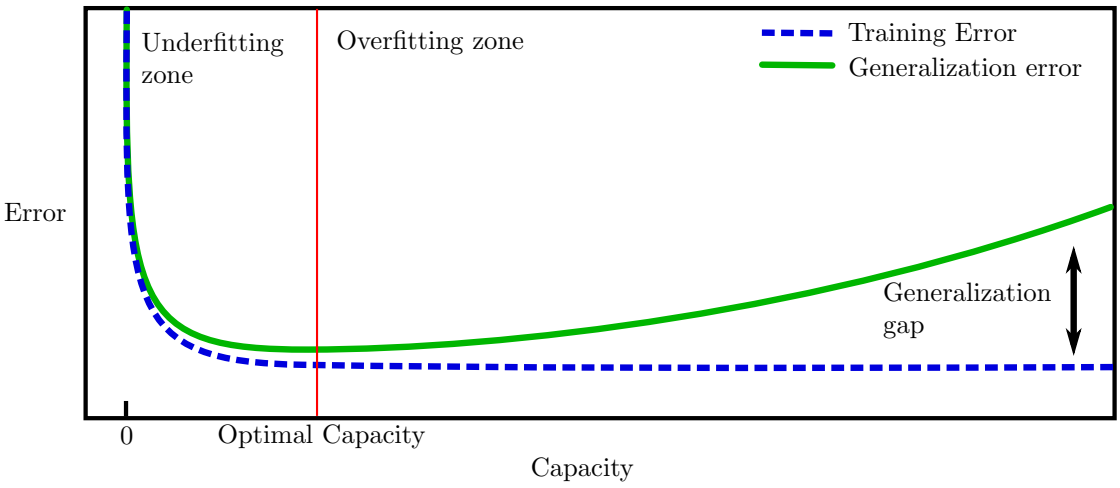
The central challenge in machine learning is that we must perform well on new, previously unseen inputs—not just those on which our model was trained. The ability to perform well on previously unobserved inputs is called *generalization*.
The factors determining how well a machine learning algorithm will perform are its ability to:

1. Make the training error small.

2. Make the gap between training and test error small.

Underfitting occurs when the model is not able to obtain a sufficiently low error value on the training set. *Overfitting* occurs when the gap between the training error and test error is too large. We can control whether a model is more likely to overfit or underfit by altering its *capacity*.

BY-NC-SA



1.6 Maximum Likelihood Estimation p. 128

Let $p_{\text{model}}(\boldsymbol{x};\boldsymbol{\theta})$ be the function that estimates the true probability $p_{data}(\boldsymbol{x})$. A maximum likelihood estimation tries to find an optimum for the parameter $\boldsymbol{\theta}$.

An example for the true (unknown) probability $p_{data}(\boldsymbol{x})$ could be the probability that on a 256x256x8 grayscale picture is a banana, given every pixel ($\boldsymbol{x}$). So $p_{\text{model}}(\boldsymbol{x};\boldsymbol{\theta})$ could be any form of multidimensional density function, showing the probability of $\boldsymbol{x}$ on every possible $\boldsymbol{\theta}$. The form of this probability function depends on the structure (connections, activation-functions etc.) of our neuronal network.

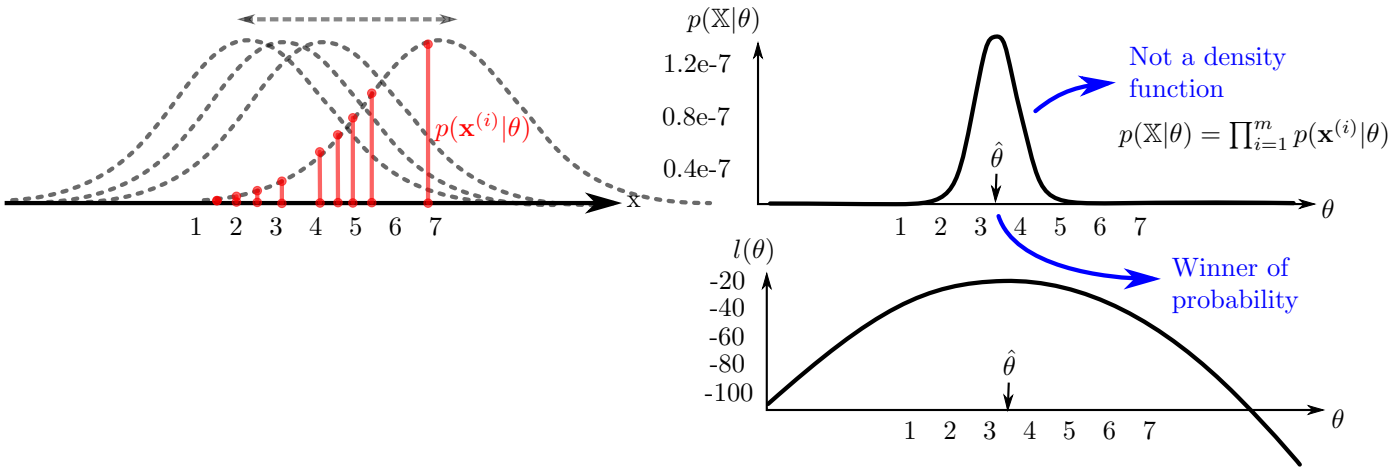
$\mathbb{X} = \{\boldsymbol{x}^{(1)}, ..., \boldsymbol{x}^{(m)}\}$	These are m experiments (samples) from the true (unknown) data-generating distribution $p_{\text{data}}(\boldsymbol{x})$
$\boldsymbol{\theta}$	Our parameters or weights
$p(\boldsymbol{\theta})$	Density function of the weights. If we assume it uniform (no information in it), then we maximize $p_{\text{model}}(\boldsymbol{x}\|\boldsymbol{\theta})$ while we are maximizing $p_{\text{model}}(\boldsymbol{x};\boldsymbol{\theta})$
$p_{\text{model}}(\boldsymbol{x}\|\boldsymbol{\theta})$	Is the probability getting the sample $\boldsymbol{x}$, when $\boldsymbol{\theta}$ is given.
$p_{\text{model}}(\boldsymbol{x};\boldsymbol{\theta})$	Multidimensional density function $= p_{\text{model}}(\boldsymbol{x}\|\boldsymbol{\theta})p(\boldsymbol{\theta})$

The likelihood of $\boldsymbol{\theta}$ with the set of samples $\mathbb{X}$ is defined as $p(\mathbb{X};\boldsymbol{\theta}) = \prod_{i=1}^{m} p_{\text{model}}(\boldsymbol{x}^{(i)};\boldsymbol{\theta})$ (samples are i.i.d. - independent & identically distributed).

Recipe with log-likelihood:

1. Log-likelihood: $l(\boldsymbol{\theta}) \equiv \ln(p(\mathbb{X};\boldsymbol{\theta})) = \sum_{k=1}^{m} \ln(p(\boldsymbol{x}^{(i)};\boldsymbol{\theta}))$ $\quad \left(\frac{d}{dx}\ln(x) = \frac{1}{x}\right)$
2. Gradient: $\nabla_{\boldsymbol{\theta}} l = \sum_{k=1}^{n} \nabla_{\boldsymbol{\theta}} \ln(p(\boldsymbol{x}^{(i)};\boldsymbol{\theta}))$
3. $\nabla_{\boldsymbol{\theta}} l = 0$: find solution $\hat{\boldsymbol{\theta}} = \arg\max_{\boldsymbol{\theta}} l(\boldsymbol{\theta})$

An example with a gauss maximum likelihood estimation where the only parameter θ is the mean and nine samples $\mathbb{X}$:



BY-NC-SA

1.7 Conditional Log-Likelihood p. 129

In the training situation, we have also (all) our training targets $\mathbb{Y}$ to all inputs $\mathbb{X}$. We have $P(\boldsymbol{y}|\boldsymbol{x};\boldsymbol{\theta})$, in order to predict $\boldsymbol{y}$ given $\boldsymbol{x}$ and $\boldsymbol{\theta}$:

$\boldsymbol{\theta}_{ML} = \arg\max_{\boldsymbol{\theta}} P(\mathbb{Y}|\mathbb{X};\boldsymbol{\theta})$

$\boldsymbol{\theta}_{ML} = \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{m} \log P(\boldsymbol{y}^{(i)}|\boldsymbol{x}^{(i)};\boldsymbol{\theta})$

This conditional log-likelihood shows the probability that our network generated the output $\mathbb{Y}$ given the input $\mathbb{X}$ and the parameters $\boldsymbol{\theta}$). The log-likelihood reaches the maximum at the point $\boldsymbol{\theta}_{ML}$.

1.8 Stochastic Gradient Descent p. 147

The dataset to train a neuronal network normally is too large to calculate the gradient for the whole dataset. Instead we use the Stochastic Gradient Descent Algorithm (SGD) to optimize our loss-function. Assuming we have m data-samples to train, the entire loss-function would be:

$J(\boldsymbol{\theta}) = \mathbb{E}_{\boldsymbol{x},y\sim\hat{p}_{\text{data}}} L(\boldsymbol{x},y,\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m} L(\boldsymbol{x}^{(i)},y^{(i)},\boldsymbol{\theta})$ where

$\mathbb{E}_{x\sim P}[f(x)] = \sum_{x} P(x)f(x)$ means the expectation of the function $f(x)$ with respect to a prob. distribution $P(x)$.

L is the example loss $L(\boldsymbol{x}^{(i)},y^{(i)},\boldsymbol{\theta}) = -\log p(y|\boldsymbol{x};\boldsymbol{\theta})$ for the negative log-likelihood case.

The gradient for the whole data-loss-function would be: $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m} \nabla_{\boldsymbol{\theta}} L(\boldsymbol{x}^{(i)},y^{(i)},\boldsymbol{\theta})$

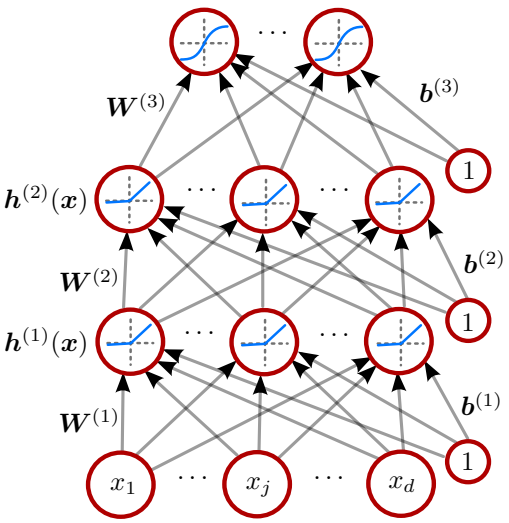
Instead we use a minibatch of m' examples **drawn uniformly** from the entire trainings-set: $\mathbb{B} = \{\boldsymbol{x}^{(1)},\ldots,\boldsymbol{x}^{(m')}\}$

The estimate of the gradient is formed as: $\boldsymbol{g} = \frac{1}{m'}\sum_{i=1}^{m'} \nabla_{\boldsymbol{\theta}} L(\boldsymbol{x}^{(i)},y^{(i)},\boldsymbol{\theta})$

The SGD Algorithm then follows the estimated gradient downhill: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon\boldsymbol{g}$

2 Deep Feedforward Networks

Deep feedforward networks are also called feedforward neural networks or multilayer perceptrons (MLPs).



Feedforward networks are mapping an input $\boldsymbol{x}$ to an output $\boldsymbol{y}$ with a function $\boldsymbol{y} = f(\boldsymbol{x};\boldsymbol{\theta})$. $\boldsymbol{\theta}$ are all adjustable parameters like for example the weight-matrices $\boldsymbol{W}^{(i)}$ or the bias $b^{(i)}$.
For a network with L hidden layers we can use the formulas:
Layer pre-activation for $k > 0$: $(\boldsymbol{h}^{(0)}(\boldsymbol{x}) = \boldsymbol{x})$

$$\boldsymbol{a}^{(k)}(\boldsymbol{x}) = \boldsymbol{b}^{(k)} + \boldsymbol{W}^{(k)}\boldsymbol{h}^{(k-1)}(\boldsymbol{x})$$

Hidden layer activation for k from 1 to L:

$$\boldsymbol{h}^{(k)}(\boldsymbol{x}) = \boldsymbol{g}(\boldsymbol{a}^{(k)}(\boldsymbol{x}))$$

Output layer activation for $k = L+1$:

$$\boldsymbol{h}^{(L+1)}(\boldsymbol{x}) = \boldsymbol{o}(\boldsymbol{a}^{(L+1)}(\boldsymbol{x})) = \boldsymbol{f}(\boldsymbol{x})$$

BY-NC-SA

2.1 Cost Functions p. 172

Mostly we learn with maximum likelihood:	$J(\boldsymbol{\theta}) = -\mathbb{E}_{\boldsymbol{x},\boldsymbol{y}\sim\hat{p}_{\text{data}}} \log p_{\text{model}}(\boldsymbol{y}\|\boldsymbol{x};\boldsymbol{\theta})$
Another possibility is to learn with mean squared error cost:	$J(\boldsymbol{\theta}) = \frac{1}{2}\mathbb{E}_{\boldsymbol{x},\boldsymbol{y}\sim\hat{p}_{\text{data}}}\|\boldsymbol{y} - f(\boldsymbol{x};\boldsymbol{\theta})\|^2$
When we us the L1 norm (mean absolute error):	$J(\boldsymbol{\theta}) = \mathbb{E}_{\boldsymbol{x},\boldsymbol{y}\sim\hat{p}_{\text{data}}}\|\boldsymbol{y} - f(\boldsymbol{x};\boldsymbol{\theta})\|_1$

Unfortunately, mean squared error and mean absolute error often lead to poor results when used with gradient-based optimization (some output units that saturate produce very small gradients).

2.2 Output Units p. 175

The last units in the network chain are the output units. There are several types or output units:

Linear Units: Are often used to produce a mean of a distribution function: $\hat{\boldsymbol{y}} = \boldsymbol{W}^\top \boldsymbol{h} + \boldsymbol{b}$, with the given features $\boldsymbol{h}$.

Sigmoid Units: (For Bernouille Output Distributions) We can think the sigmoid output unit as having two components. First a linear layer, then a sigmoid activation function to convert to a probability between 1 and 0. So we have $\hat{\boldsymbol{y}} = \sigma(\boldsymbol{W}^\top \boldsymbol{h} + \boldsymbol{b})$ with the logistic sigmoid function: $\sigma(x) = \frac{1}{1+\exp(-x)}$

Softmax Units: (For Multinoulli Output Distributions) To have a distribution with multiple outputs who sum to one, we use softmax. First we need a linear layer predicting unnormalized log probabilities: $\boldsymbol{z} = \boldsymbol{W}^\top \boldsymbol{h} + \boldsymbol{b}$. Then we can calculate the softmax to every element of $\boldsymbol{z}$: $\text{softmax}(\boldsymbol{z})_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$

The log in the log-likelihood can undo the exp of the softmax to receive: $\log \text{softmax}(\boldsymbol{z})_i = z_i - \log \sum_j \exp(z_j)$.

Note that in softmax we can always add some constant: $\text{softmax}(\boldsymbol{z}) = \text{softmax}(\boldsymbol{z} + c)$.
To be numerically stable we do: $\text{softmax}(\boldsymbol{z}) = \text{softmax}(\boldsymbol{z} - \max_i z_i)$.

2.3 Hidden Units p. 185

For the hidden unit activation function we normally use **Rectified Linear Units** (ReLU), which are defined as:

$$g(z) = \max\{0, z\}$$

When we use ReLUs it can be a good practise to initialize all elements of $\boldsymbol{b}$ with a small positive value, such as 0.1, so the units are more likely to be active at the beginning of the training.

Other ReLU types:

With a adjustable slope when $z_i < 0$:	$g(\boldsymbol{z}, \boldsymbol{\alpha})_i = \max(0, z_i) + \alpha_i \min(0, z_i)$
With $\alpha_i = -1$ we get the **absolute value rectification**:	$g(z) = \|z\|$ (used for image recognition)
With a small value like $\alpha_i = 0.01$ we get the **leaky ReLU**	

A **PReLU** (parametric) treats α_i as a learn-able parameter. The non-zero slope is an advantage while optimizing the network with SGD.

Other output units (in general not used any-more in feedforward networks):
Sigmoid activation function: $g(z) = \sigma(z)$
Hyperbolic tangent activation function: $g(z) = \tanh(z) = \frac{\exp(z)-\exp(-z)}{\exp(z)+\exp(-z)} = 2\sigma(2z) - 1$
Softplus activation function: $\zeta(x) = \log(1 + \exp(x))$

BY-NC-SA

2.4 Back-Propagation p. 197

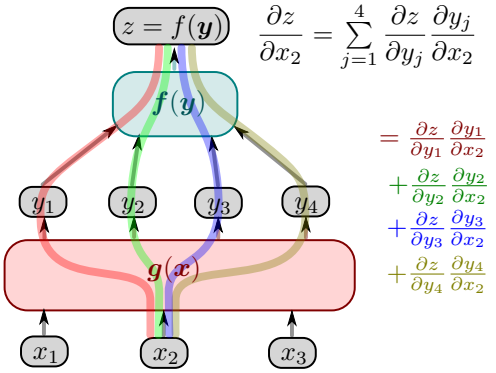
We do back-propagation with the chain rule of calculus: Suppose that $y = g(x)$ and $z = f(g(x)) = f(y)$ where x, y, z are scalars, the chain rule states that:

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$

Suppose that $\boldsymbol{x} \in \mathbb{R}^m, \boldsymbol{y} \in \mathbb{R}^n$, $\boldsymbol{g}$ maps from $\mathbb{R}^m$ to $\mathbb{R}^n$ and f from $\mathbb{R}^m$ to $\mathbb{R}$. $\boldsymbol{y} = \boldsymbol{g}(\boldsymbol{x})$ and $z = f(\boldsymbol{y})$

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i} \qquad \nabla_{\boldsymbol{x}} z = \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\right)^\top \nabla_{\boldsymbol{y}} z$$

To the right the same in vector notation for all x_i, where $\frac{\partial \boldsymbol{x}}{\partial \boldsymbol{y}}$ is the $n \times m$ Jacobian matrix of g.

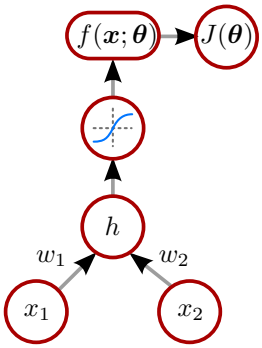


Drawing for $m = 3$ and $n = 4$

Example of back-propagation on a simple net where $\{w_1, w_2\} \in \boldsymbol{\theta}$, and the loss $J(\boldsymbol{\theta}) = \frac{1}{2}\mathbb{E}_{\boldsymbol{x},\boldsymbol{y}\sim\hat{p}_{\text{data}}}\|\boldsymbol{y} - f(\boldsymbol{x};\boldsymbol{\theta})\|^2$:

$\frac{\partial}{\partial x}\sigma(x) = \frac{\exp(x)}{(1+\exp(x))^2}$

1. Derive the Loss with f: $\frac{\partial L}{\partial f} = \frac{\partial}{\partial f}(\frac{1}{2}(y-f)^2) = f - y$
2. Back-propagate to h: $\frac{\partial L}{\partial h} = \frac{\partial L}{\partial f}\frac{\partial f}{\partial h} = (f-y)\left(\frac{\exp(h)}{(1+\exp(h))^2}\right)$
3. Back-propagete to w_1: $\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial h}\frac{\partial h}{\partial w_1} = (f-y)\left(\frac{\exp(h)}{(1+\exp(h))^2}\right)x_1$
4. Back-propagete to w_2: $\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial h}\frac{\partial h}{\partial w_2} = (f-y)\left(\frac{\exp(h)}{(1+\exp(h))^2}\right)x_2$



3 Regularisation for Deep Learning

3.1 Parameter Norm Penalties p. 223

Many regularization approaches are based on limiting the capacity of neuronal network models by adding a parameter norm penalty $\Omega(\boldsymbol{\theta})$ to the objective function J. The norm penalty tries to reduce the norm of the weight vector $\boldsymbol{w}$, such that we have small numbers as weights. This has also numerical (stability) benefits. We denote the regularized objective function by $\tilde{J}$:

$$\tilde{J}(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) = J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) + \alpha\Omega(\boldsymbol{\theta})$$

where α is an adjustable parameter, to adjust the influence of the norm penalty and $\boldsymbol{w} \in \boldsymbol{\theta}$ are the weight parameters.

3.1.1 L^2 Parameter Regularisation p. 224

The L^2 parameter norm penalty is commonly known as **weight decay**, where we add the regularization term $\Omega(\boldsymbol{\theta}) = \frac{1}{2}\|\boldsymbol{w}\|_2^2$.

Note: The fraction $\frac{1}{2}$ is just to get a simpler derivation.

BY-NC-SA

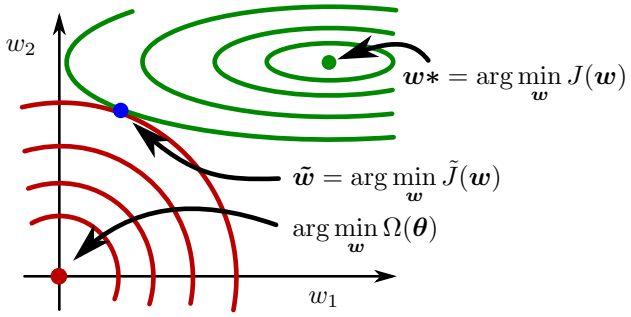
The weight decay model has the following objective function:

$$\tilde{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = \frac{\alpha}{2}\boldsymbol{w}^\top \boldsymbol{w} + J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y})$$

The gradient of $\tilde{J}$ is given by $\nabla_{\boldsymbol{w}}\tilde{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = \alpha\boldsymbol{w} + \nabla_{\boldsymbol{w}} J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y})$.
If take a single gradient step to update the weights we can write this with:

$$\boldsymbol{w} \leftarrow (1 - \epsilon\alpha)\boldsymbol{w} - \epsilon\nabla_{\boldsymbol{w}} J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y})$$

We see that with every step, we reduce the size of the weights proportional to the actual entries $\boldsymbol{w}_i$, and subtract the gradient to improve the model.



The best weights $\tilde{\boldsymbol{w}}$ for $\tilde{J}$ are now shifted from $\boldsymbol{w}*$ to the point zero considering the function $J(\boldsymbol{w})$. Normally the point $\boldsymbol{w}*$ would be a place where the model is completely over-fitted, except we have very much training data.

3.1.2 L^1 Parameter Regularisation p. 226

Formally, L^1 regularization on the model parameter $\boldsymbol{w}$ is defined as: $\Omega(\boldsymbol{\theta}) = \|\boldsymbol{w}\|_1 = \sum_i |w_i|$

thus the regularized objective function is given by: $\tilde{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = \alpha\|\boldsymbol{w}\|_1 + J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y})$

with the corresponding gradient: $\nabla_{\boldsymbol{w}}\tilde{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = \alpha\,\text{sign}(\boldsymbol{w}) + \nabla_{\boldsymbol{w}} J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y})$

So on every gradient-step we reduce the weights $\boldsymbol{w}$ additionally with the constant α. In comparison to L^2, the regularization contribution to the gradient no longer scales linear with each w_i. L^1 regularization results also in a solution that is more sparse than L^2. Sparsity in this context refers to the fact that some parameteres have an optimal value of zero.

3.2 Dataset Augmentation p. 233

The best way to make a machine learning model generalize better is to train it on mor data. One way to get around this problem is to create fake data and add it to the training set. A few ways to generate more data:

- Add (small) noise to the input.
- Translating (rotate,scale and shift) data for example on images.
- Generate data from a model (can also be another neuronal network).

3.3 Noise Robustness p. 235

In the general case, it is important to remember that noise injection can be much more powerful than simply shrinking the parameters, especially when the noise is added to the hidden units.

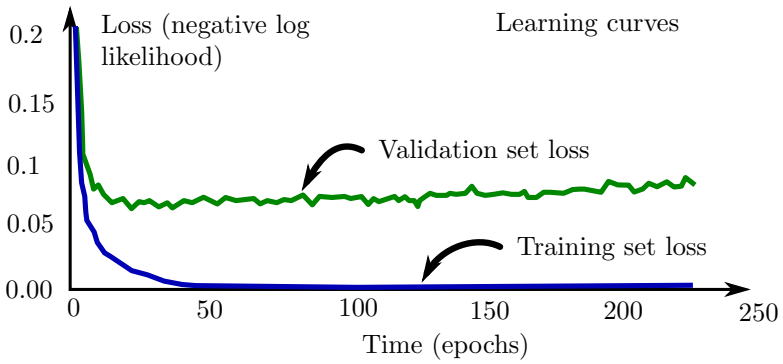
Another way is to add noise to the weights (recurrent neuronal networks).

Noise at the Output Targets: Sometimes we have also errors in our labels. This can be harmful if we maximize $\log p(y|\boldsymbol{x})$. One way to prevent this ist to explicityly model the noise on the labels (label smoothing on the output).

BY-NC-SA

3.4 Early Stopping p. 239

When training models with sufficient representational capacity to overfil the task, we often observe that training error decreases steadily over time, but validation set error begins to rise again.
We can obtain a model with better validation set error (and thus, hopefully better test set error) by returning to the parameter setting at the point with the lowest validation set error.



3.5 Parameter Tying and Parameter Sharing p. 246

While parameter norm penalty is one way to regularize parameters to be close to one another, the more popular way is to use constraints: *to force sets of parameters to be equal.* This has the significant advantage that we use only a subset of parameters to optimize (less memory, Convolutional NN).

3.6 Bagging p. 249

Bagging (short: bootstrap aggregating) is a technique for reducing generalization error by combining serveral (different) models. Bagging is an example of a general strategy in machine learning called „model averaging“. Techniques empliying this strategy are known as „ensemble methods“.

If all models are not biased, and do not make the same errors on the test set (small covariances), baggig can be a powerful method.

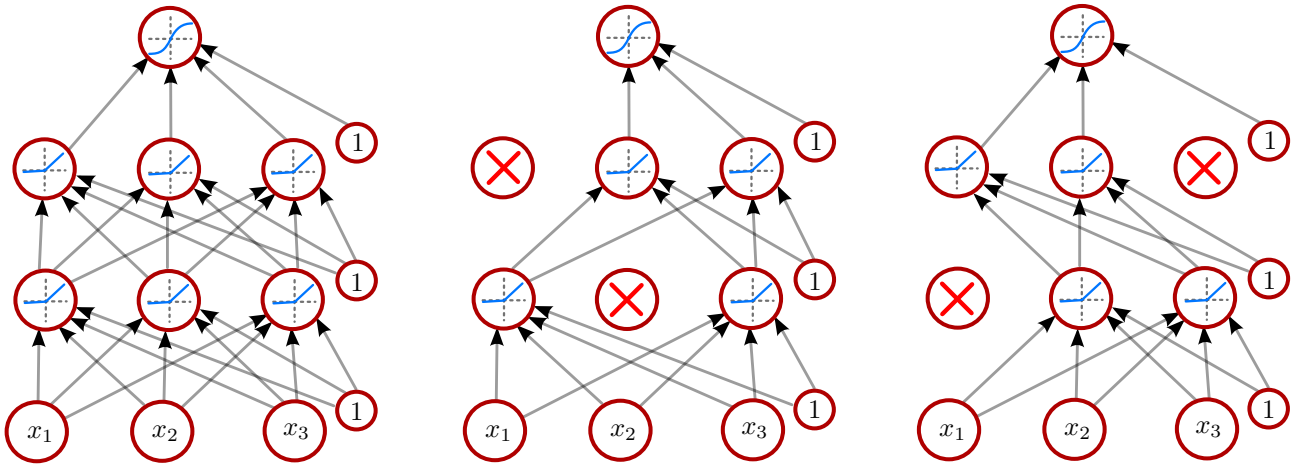
In general good methodes have similar variances. In this case we can take the average. If not we have to weight with the variances.

$$y_i = \frac{\mathrm{Var}(y_{i_2})}{\mathrm{Var}(y_{i_1}) + \mathrm{Var}(y_{i_2})} y_{i_1} + \frac{\mathrm{Var}(y_{i_1})}{\mathrm{Var}(y_{i_1}) + \mathrm{Var}(y_{i_2})} y_{i_2}$$

3.7 Dropout p. 251

Dropout provides a computationally inexpencive but powerful method of regularizing a broad family of models. Dropout aims to approximate the process of bagging, but with an exponentially large number of neural networks. To train with dropout, we use a minibatch-based learning algorthm that makes small steps. Each time we load a new minibach, we randomly sample a different binary mask, to activate or deactivate the different neurons.

BY-NC-SA



entire model training minibatch n training minibatch n+1

At aplication time (inference) we use the entire model. To correct the produced error from dropout, we use the **weight scaling inference rule**. Because we usually use an inclusion probability of $\frac{1}{2}$, the weight scaling rule usually amounts to dividing the weights by 2 at the end of training.

4 Optimization for Training Deep Models

4.1 Basic Algorithms p. 286

In the normal case we use SGD. A crucial parameter for the SGD algorithm is the learning rate. It can be a fixed value. In practice, it is necessary to gradually decrease the learning rate over time (iterator k).

A common way is, to decay the learning rate linearly until iteration τ, with $\alpha = \frac{k}{\tau}$:

$$\epsilon_k = (1-\alpha)\epsilon_0 + \alpha\epsilon_\tau$$

After iteration τ , it is common to leave ϵ constant.

4.1.1 Momentum p. 288

The method of momentum is designed to accelerate learning, especially in the face of high curvature, small but consistent gradients, or noisy gradients.

Formally, the momentum algorithm introduces a variable $\boldsymbol{v}$ that plays the role of velocity. It is the direction and speed at which the parameters move through parameter space. $\boldsymbol{v}$ accumulates the gradient elements weightet with ϵ and α.

$$\boldsymbol{v} \leftarrow \alpha\boldsymbol{v} - \epsilon\nabla_{\boldsymbol{\theta}}\left(\frac{1}{m}\sum_{i=1}^{m} L(\boldsymbol{f}(\boldsymbol{x}^{(i)};\boldsymbol{\theta}),\boldsymbol{y}^{(i)})\right)$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$$

The larger α is relative to ϵ, the more previous gradients affect the current direction.

4.1.2 Nestrov Momentum p. 226

Compared with the standart momentum, here the gradient is evaluated after the current velocity is applied.

$$\boldsymbol{v} \leftarrow \alpha\boldsymbol{v} - \epsilon\nabla_{\boldsymbol{\theta}}\left(\frac{1}{m}\sum_{i=1}^{m} L(\boldsymbol{f}(\boldsymbol{x}^{(i)};\boldsymbol{\theta}+\alpha\boldsymbol{v}),\boldsymbol{y}^{(i)})\right)$$

BY-NC-SA

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$$

With SGD the Nestrov Momentum seams not improve the learning progress.

4.2 Parameter Initialisation Strategies p. 292

Very important is the initialisation of the weights before we start training. An ill conditioned initalisation (e.g. all zero) can be a reason that we can not improve the parameters with more training (or only with very slow progress). If possible, take a pretrained network. If not use this formula:

$$W_{i,j} \sim U\left(-\frac{6}{\sqrt{m+n}}, \frac{6}{\sqrt{m+n}}\right)$$

Where m and n are the width and the length of the parameter space.

4.3 Algorithms with Adaptive Learning Rates p. 298

4.3.1 AdaGrad p. 299

The AdaGrad algorithm individually adapts the learning rates of all model parameters by scaling them inversely proportional to the square root of the sum of all of their historical squared gradient values. AdaGrad performs well for some but not all deep learning models.

1. Compute gradient: $\boldsymbol{g} = \nabla_{\boldsymbol{\theta}} \left(\frac{1}{m} \sum_{i=1}^{m} L(\boldsymbol{f}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)}) \right)$
2. Accumulate squared gradient: $\boldsymbol{r} \leftarrow \boldsymbol{r} + \boldsymbol{g} \odot \boldsymbol{g}$
3. Compute update: (Division and square root applied element-wise) $\Delta\boldsymbol{\theta} \leftarrow -\frac{\epsilon}{\delta + \sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$
4. Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$

$\boldsymbol{A} \odot \boldsymbol{B}$: Element-wise (Hadamard) product of $\boldsymbol{A}$ and $\boldsymbol{B}$.

4.3.2 RMSProp p. 299

The RMSProp algorithm modifies AdaGrad to perform better in the non-convex setting by changing the gradient accumulation into an exponentially weighted moving average. Empirically, RMSProp has been shown to be an effective and practical optimization algorithm for deep neural networks. To use RMSProp we need:

- Global learning rate ϵ
- Decay rate ρ
- Small constant δ, usually 10^{-6}, used to stabilize division by small numbers

1. Compute gradient: $\boldsymbol{g} = \nabla_{\boldsymbol{\theta}} \left(\frac{1}{m} \sum_{i=1}^{m} L(\boldsymbol{f}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)}) \right)$
2. Accumulate squared gradient: $\boldsymbol{r} \leftarrow \rho\boldsymbol{r} + (1-\rho)(\boldsymbol{g} \odot \boldsymbol{g})$
3. Compute update: (Division and square root applied element-wise) $\Delta\boldsymbol{\theta} \leftarrow -\frac{\epsilon}{\delta + \sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$
4. Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$

BY-NC-SA

4.3.3 Adam p. 301

Adam „adaptive moments“can be seen as a variant on the combination of RMSProp and momentum with a few important distinctions. Adam includes an estimation for the first and the second order ($\boldsymbol{s}$ and $\boldsymbol{r}$) which are initialised with $\mathbf{0}$. Further there is a bias correction for the initialisation of this moments. Adam is generally regarded as being fairly robust to the choice of hyperparameters, though the learning rate sometimes needs to be changed from the suggested default.

For Adam we need:

- Step size ϵ (Suggested default: 0.001)
- Exponential decay rates for moment estimates, ρ_1 and $\rho_2 \in [0, 1[$ (Suggested defaults: 0.9 and 0.999 respectively)
- Small constant δ, usually 10^{-8}, used to stabilize division by small numbers
- Initial parameters $\boldsymbol{\theta}$:
  Initialize 1st and 2nd moment variables $\boldsymbol{s} = \mathbf{0}$, $\boldsymbol{r} = \mathbf{0}$
  Initialize time step t = 0

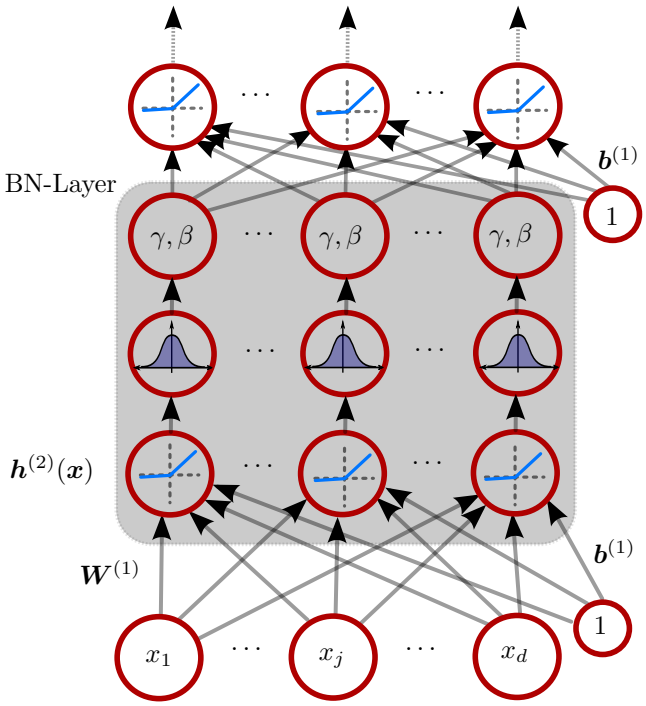
1. Compute gradient and increment time: $\boldsymbol{g} = \nabla_{\boldsymbol{\theta}} \left(\frac{1}{m} \sum_{i=1}^{m} L(\boldsymbol{f}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)}) \right) \qquad t \leftarrow t + 1$
2. Update biased first moment estimate: $\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1 - \rho_1)\boldsymbol{g}$
3. Update biased second moment estimate: $\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1 - \rho_2)(\boldsymbol{g} \odot \boldsymbol{g})$
4. Correct bias in first moment: $\hat{\boldsymbol{s}} = \frac{\boldsymbol{s}}{1-\rho_1^t}$
5. Correct bias in second moment: $\hat{\boldsymbol{r}} = \frac{\boldsymbol{r}}{1-\rho_2^t}$
6. Compute update: (Division and square root applied element-wise) $\Delta\boldsymbol{\theta} \leftarrow -\epsilon \frac{\hat{\boldsymbol{s}}}{\delta + \sqrt{\hat{\boldsymbol{r}}}} \odot \boldsymbol{g}$
7. Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$

4.4 Batch Normalization p. 309

Batch Normalization is a method to improve deep models in serveral ways:

- Improves the gradientflow through the network
- Allows higer learning rates
- Reduces the strong dependence on initialisation
- Acts as a form of regularization and slightly reduces the need for dropout

When we use Batch Normalisation, we insert an additional layer on every hidden layer. During the training we normalize the mean and the standard deviation of one minibatch, on every neuron. So we force to have a unit gaussian on every output. In after the training we replace this numbers with data collected during the training. In addition we insert two new lernable parameters per neuron such that the network can undo the Batch Normalisation if it wants to.



BY-NC-SA