Back-Propagation and Algorithms for Training Artificial Neural Networks

Gero F. Kauerauf

RWTH Aachen University
gero.kauerauf@rwth-aachen.de

Abstract. In this paper, we elaborate on key concepts in training neural networks. We start by explaining the fundamentals of feedforward networks. We then consider basic algorithms such as back-propagation and stochastic gradient descent, which are widely used for training. The result is a compact overview of standard methods.

Keywords: Deep Learning · Back-Propagation · Stochastic Gradient Descent · Feedforward Networks

1 Introduction

In recent years neural networks have become very popular. They are used in many fields for different tasks. Therefore, the efficient training of these networks has become very important.

In this paper we focus on *feedforward networks* and *supervised learing*, for which we describe these algorithms in great detail. For better understanding, this paper itself is structured in the same order the algorithms and methods to train are used in. Key elements are:

- Structure of *feedforward networks* and *forward propagation*
- *Cost functions* and *computational graphs* for these networks
- *Backward-propagation*
- *Stochastic gradient descent*

We make use of examples and graphics at key points for better understanding. Furthermore, we have simplified unnecessarily complex practical parts, that are not essential for a fundamental understanding. This paper is mostly based on the standard textbook by **Goodfellow et al.** [1].

2 Problem description

Since computers are getting better every year, one would expect that we could solve more and more problems. This may be true in some fields, however, there are still problems that are very hard to solve.

Oftentimes, these problems are the ones that are hard to describe mathematically. Take for example an image of a *cat* and a *dog*. A human can intuitively

recognize a cat on an image and is able to distinguish between those two animals with ease. However, up to this day, it is not possible to describe the key characteristics of a *cat* mathematically, in such a way, that one can build an algorithm that solves this problem.

Another class of problems is the one with problems that can be described mathematically perfectly but are too complex. These are for example games like chess. We have no problem with describing the rules of chess. However, due to the sheer amount of possible moves and positions in chess, we have trouble analyzing games perfectly. Writing an algorithm that "brute-forces" a game of chess is no problem, but in practice our computers are too slow. Even worse, computers might never be fast enough!

This is where *neural networks* come into play. For these kinds of problems, they are currently the best solution we have.

3 Feedforward Networks

Deep feedforward networks, also called *feedforward neural networks* or *multilayer perceptrons* (MLPs) are basic deep learning models. Their main purpose is to approximate some kind of function $f^*(x)$. Take for example a classification function $y = f^*(x)$, where every input x maps to a class y. A feedforward network now aims to learn the best parameters Θ for a function $y = f(x, \Theta)$ that approximates $f^*(x)$.

These models are *networks* due to the fact that they can be represented by *directed graphs*. They are called *feedforward networks* because the input x flows through these functions without any *feedback* connections. This means that the graph is *acyclic*. If the graph is *not acyclic* and therefore $feedback$ connections exist, we have a *recurrent neural networks*.

For example let $f(x) = (f^3 \circ f^2 \circ f^1)(x) = f^3(f^2(f^1(x)))$. We now define the *depth* of a *feedforward network* f as the number of functions that it is composed of. Our example has a *depth* of 3. These kinds of chains are very typical in *neural networks*, as each function in the chain represents a layer in the *directed acyclic graph*. The *domain* of the first function in the chain (in our example f^1) defines the inputs of the network. The corresponding first layer is therefore called *input layer*. Likewise, the last function in the chain (in our example f^3) maps the input to its final value. The last layer is therefore called *output layer*. If we think of f as a classifier function the elements of the *codomain* of f^3 are the classes. Layers between the *input* and *output* layer are called *hidden layers*.

3.1 Feedforward network graphs

As proposed earlier we can also think of *neural networks* as *acyclic directed graphs*. Loosely inspired by neuroscience, *neurons* are interpreted as *nodes* and *synapses* as *edges*. Each *node* now receives input from an arbitrary amount of neurons in the previous layer and computes an output with its own activation

function. An *edge* on the other hand has a scalar *weight*. These *weights* can be adjusted to change the output of the neural network.

It is shown in [2], [3], that in most feedforward networks a *rectifier linear units* (reLU) ($f(x) = \max(0, x)$) activation function works best. However, in certain cases other activation functions might perform better. Such a case for example would be the *swish* activation function ($f(x) = x * \text{sigmoid}(x)$), that tends to perform better than reLU on deeper networks [4].

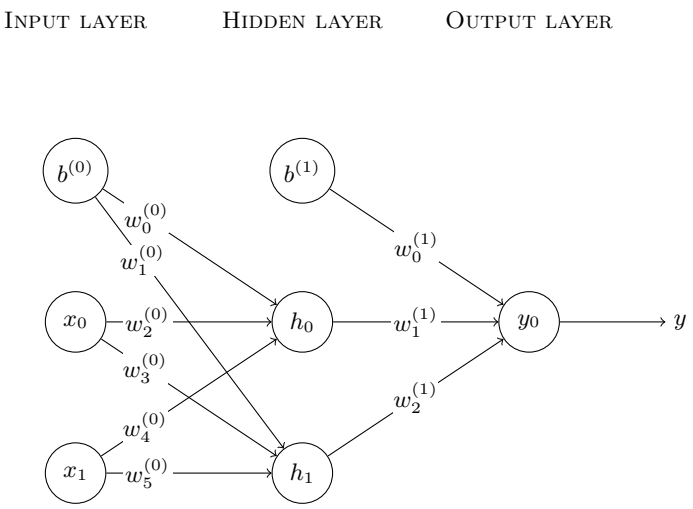

Fig. 3.1. Example DNN with a single hidden layer

The network shown in [fig. 3.1] can be described by its *weight matrices*.

We can store the weights of the edges between two layers in a single *weight matrix* $\mathbf{W} \in \mathbb{R}^{m \times n}$. The input can then be represented as a vector $x \in \mathbb{R}^n$. Calculating the output of a neural network can therefore be achieved by iterative *matrix-vector multiplication* and component-wise usage of the activation function on each layer.

For [fig. 3.1] the weight matrices are

$$\mathbf{W}^{(0)} = \begin{pmatrix} w_2^{(0)} & w_4^{(0)} \\ w_3^{(0)} & w_5^{(0)} \end{pmatrix}, \quad \mathbf{W}^{(1)} = \begin{pmatrix} w_1^{(1)} & w_2^{(1)} \end{pmatrix}. \tag{3.1.1}$$

With the corresponding bias vectors

$$\mathbf{b}^{(0)} = \begin{pmatrix} w_0^{(0)} \\ w_1^{(0)} \end{pmatrix}, \quad \mathbf{b}^{(1)} = \begin{pmatrix} w_0^{(1)} \end{pmatrix}. \tag{3.1.2}$$

In order to get more “flexible” we add a *bias* node to each layer. This bias node always has an input of 1 and describes the constant in a linear equation.

Let $f(x)$ now be the activation function and

$$\mathbf{x} = \begin{pmatrix} x_0 \\ x_1 \end{pmatrix}, \tag{3.1.3}$$

the input vector. The output vector for [fig. 3.1 on the preceding page] can be calculated by evaluating the following equations.

$$\mathbf{x}^{(0)} = f(W^{(0)}x + b^{(0)}) = f(\begin{pmatrix} w_2^{(0)} & w_4^{(0)} \\ w_3^{(0)} & w_5^{(0)} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \end{pmatrix} + \begin{pmatrix} w_0^{(0)} \\ w_1^{(0)} \end{pmatrix}) \tag{3.1.4}$$

$$\hat{\mathbf{y}} = f(W^{(1)}\mathbf{x}^{(0)} + b^{(1)}) = f(\begin{pmatrix} w_1^{(1)} & w_2^{(1)} \end{pmatrix} \mathbf{x}^{(0)} + \begin{pmatrix} w_0^{(1)} \end{pmatrix}) \tag{3.1.5}$$

Where the activation function f is used component-wise.

4 Deep learning

Deep learning refers to the broader family of machine learning algorithms, that are based on artificial neural networks. Feedforward networks fall into the family of *deep neural networks* (DNNs).

The typical procedure when training a feedforward network is to define a *cost function* for it. The *cost function* is dependent on the *weights* of the neural network and describes the *error* of the network. Remember, for a given input $\mathbf{x}$ the network computes the output $\hat{\mathbf{y}}$. The cost function now measures the distance of $\hat{\mathbf{y}}$ to the real $\mathbf{y}$, this distance is called the *error*. In the context of *supervised learning*, we know $\mathbf{y}$, because a human has previously "classified" the input. A minimum in the *cost function* therefore minimizes the *error* of the network. In order to minimize the *cost function* we have to compute its gradient and then *adjust* the weights. The computation of the gradient is performed by the *back-propagation algorithm*. The *adjustment* is then performed by an *optimizing algorithm*.

4.1 Cost Functions

A very important aspect of neural networks is the choice of a cost function. In most cases we can make use of *maximum likelihood* principle [5], from which we can than can derive a good cost function. As thoroughly explained by **Goodfellow et al.** [1], we know the following:

Let $p_{\text{model}}(\boldsymbol{x}; \boldsymbol{\theta})$ map any configuration $\boldsymbol{x}$ to a real number that estimated the true probability $p_{data}(\boldsymbol{x})$. Then the maximum likelihood estimator for $\boldsymbol{\theta}$ is defined by

$$\boldsymbol{\theta}_{ML} = \arg\max_{\boldsymbol{\theta}} p_{\text{model}}(\mathbb{X}; \boldsymbol{\theta}) \tag{4.1.1}$$

$$= \arg\max_{\boldsymbol{\theta}} \prod_{i=1}^{m} p_{\text{model}}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \tag{4.1.2}$$

where $\mathbb{X}$ is a random variable, and $\boldsymbol{x}^{(i)}$ is an observed outcome of it. We can take the logarithm of this product to obtain a convenient sum, that is less prone to numerical underflow.

$$\boldsymbol{\theta}_{\mathbf{ML}} = \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{m} \log p_{\text{model}}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}) \tag{4.1.3}$$

Dividing this by m yields

$$\boldsymbol{\theta}_{\mathbf{ML}} = \arg\max_{\boldsymbol{\theta}} \mathbb{E}_{\mathbf{x}\sim\hat{p}_{\text{data}}} \log p_{\text{model}}(\boldsymbol{x}; \boldsymbol{\theta}). \tag{4.1.4}$$

We can than generalize this to predict $\boldsymbol{y}$ for a given $\boldsymbol{x}$ by estimating a conditional probability $P(\boldsymbol{Y}|\mathbf{x}; \boldsymbol{\theta})$. Let $\boldsymbol{X}$ be our inputs and $\boldsymbol{Y}$ our observed outputs. The conditional maximum likelihood estimator is

$$\boldsymbol{\theta}_{\text{ML}} = \arg\max_{\boldsymbol{\theta}} P(\boldsymbol{Y}|\boldsymbol{X}; \boldsymbol{\theta}). \tag{4.1.5}$$

This further simplifies to

$$\boldsymbol{\theta} = \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{m} \log P(\boldsymbol{y}^{(i)}|\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \tag{4.1.6}$$

if we assume that the examples are independent and identically distributed. The maximum likelihood principle gives us a method to find a good *cost function*.

4.2 Computational graphs

Another important concept we want to introduce are *computational graphs*. A *computational graph* describes a mathematical expression. Such a *directed graph* consists out of nodes and edges. A node represents an arbitrary variable, that may be a scalar, a vector, a matrix, or of any other type. Then there are also *operations*. An *operation* can be informally described as a label on a node. This label defines the operation that is performed on the input. Whereas the input is given by the *directed edges* to that node. An operation can take an arbitrary amount of inputs and stores the result in its node.

An example is given by [fig. 4.2 on the next page].

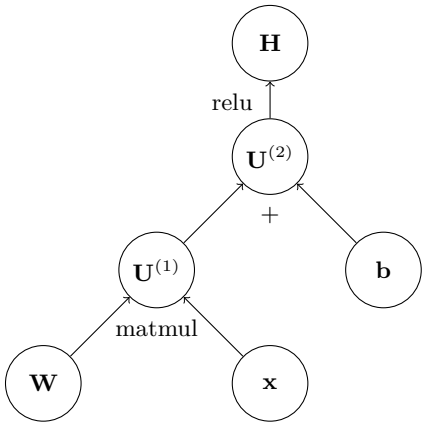

Fig. 4.1. Example of a computational graph

This specific computational graph describes the expression $H = \max\{0, \mathbf{W}\mathbf{x} + \mathbf{b}\}$, where "relu" refers to the rectifier linear unit function $f(x) = \max\{0, x\}$. Therefore this describes the forward propagation of a vector $\mathbf{x}$ through one layer, where the activation function ("relu") is used component-wise as introduced in [section 3 on page 2].

We can now construct a computational graph for any given cost function. On this graph we can then use the *back-propagation algorithm*.

4.3 Back-Propagation

Often mistaken for the whole learning algorithm, back-propagation just refers to the algorithm for computing the gradient of a computational graph. It is called back-propagation because we evaluate the gradient for the output layer back to the input layer. In some way, it can be understood as a counter part to the forward propagation. As already discussed, forward propagation describes the process of calculating the output $\hat{\mathbf{y}}$ of a feedforward network for a given input $\mathbf{x}$. Back-propagation on the other hand now works it way from the output layer to the input layer in order to compute the gradient for a given input $\mathbf{x}$, a calculated output $\hat{\mathbf{y}}$ and a "wanted" output $\mathbf{y}$.

A common misbelief is that the back-propagation is performed on the feed-forward network itself, whereas it is rather done on the computational graph of the cost function. Optimizing the *weights* in the graph of the cost function also optimizes them in the network! Since the back-propagation only computes the gradient, the "real" learning is performed by an *optimizing algorithm*, such as *stochastic gradient descent* [section 4.4 on page 9].

Chain Rule of Calculus. The chain rule of calculus describes how to calculate the derivative of a function that is composed out of other functions. Let $f : \mathbb{R} \to \mathbb{R}$

and $g : \mathbb{R} \to \mathbb{R}$ both be differentiable. If $y = g(x)$ and $z = f(y) = f(g(x))$, then

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}. \tag{4.3.1}$$

This can of course also be written as

$$(f(g(x)))' = f'(g(x))g'(x). \tag{4.3.2}$$

This can now be extended to arbitrary dimension. Let $g : \mathbb{R}^m \to \mathbb{R}^n$ and $f : \mathbb{R}^n \to \mathbb{R}$. If $\mathbf{y} = g(\mathbf{x})$ and $z = f(g(\mathbf{x})) = f(\mathbf{y})$, then

$$\frac{\partial z}{\partial \mathbf{x}_i} = \sum_j \frac{\partial z}{\partial \mathbf{y}_j}\frac{\partial \mathbf{y}_j}{\partial \mathbf{x}_i}. \tag{4.3.3}$$

Using the ∇ operator for the gradient, this can also be written as

$$\nabla_{\mathbf{x}} z = \left(\frac{\partial \mathbf{y}}{\partial \mathbf{x}}\right)^T \nabla_{\mathbf{y}} z, \tag{4.3.4}$$

where $\frac{\partial \mathbf{y}}{\partial \mathbf{x}}$ is the $n \times m$ Jacobian of g. Since feedforward networks often work with tensors of arbitrary dimension and not vectors, we would have to change our notation a bit to deal with them. We would simply "flatten" the tensor and use multiple coordinates as indices for the dimensions. However, I am not going to introduce this here, since the concept can already be understood with vectors.

The general idea of the back-propagation algorithm is simple. If we want to compute the gradient of z with respect to x we start at the node z. We observe that the gradient of z with respect to z is 1, since $\frac{\partial z}{\partial z} = 1$. We then compute the gradient with respect to each parent of z by multiplying the current gradient by the Jacobian that produced z. This means we propagate backward through the graph by multiplying with Jacobians until we reach x. If there are multiple paths, we simply take them one after another.

More formally this means that each node in the computational graph corresponds to a variable (in our case a vector). Each variable $\mathbf{v}$ has now the following subroutines:

- `operation(v)`, gets the mathematical operation that defines $\mathbf{v}$, which is represented by the edges flowing into $\mathbf{v}$ and the relation, e.g. matrix multiplication
- `children(v)`, returns an array that contains the variables that are children of $\mathbf{v}$ in the graph
- `parents(v)`, returns an array that contains the variables that are parents of $\mathbf{v}$ in the graph

Furthermore every operation `op` is associated with a `back_prop` operation, that computes the product described by [equation (4.3.4)]. Let $\mathbf{C} = \mathbf{AB}$, and the gradient of a scalar z with respect to $\mathbf{C}$ be $\mathbf{g} \in \mathbb{R}^n$. Then `back_prop` has to

define the gradient of matrix multiplication for $\mathbf{A}$ and $\mathbf{B}$. This means that `op.back_prop(parents(..), A, g)` evaluates to $\mathbf{g}\mathbf{B}^T \in \mathbb{R}^n$, because $\frac{\partial \mathbf{C}}{\partial \mathbf{A}} = \mathbf{B}^T$. Correspondingly, the call of `op.back_prop(parents(..), B, g)` evaluates to $\mathbf{A}^T\mathbf{g} \in \mathbb{R}^n$. Formally the call of `op.back_prop(parents(..), V, g)` returns

$$\sum_i \big(\nabla_{\mathbf{V}} \texttt{op.f(parents(..))}_i \mathbf{G}_i\big), \tag{4.3.5}$$

where `op.f` is the mathematical relation (e.g. matrix multiplication). The two dots in the call of the `parent(..)` function refer to the current node we are at.

It is of great importance that `op.back_prop` always pretends that all of its inputs are distinct. If the multiplication operator $\text{mul}(a, b) = a \cdot b$, gets x for both inputs, and therefore computes $\text{mul}(x, x) = x \cdot x = x^2$, `back_prop` should for both inputs return x. It will later add those two x together to receive the correct derivate of $2x$.

By making use of the `back_prop` function like this, we receive a very general back-propagation algorithm.

Back-Propagation example. With the back-propagation we can determine the gradient of [fig. 4.3] easily.

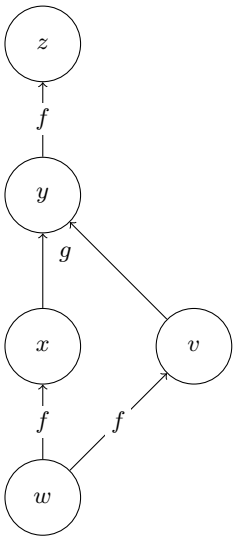

Fig. 4.2. Example graph for back-propagation

This means that $x = f(w), v = f(w), y = g(x, v), z = f(y)$. Therefore

$$z = f\Big(g\big(f(w), f(w)\big)\Big). \tag{4.3.6}$$

If we now want to compute $\frac{\partial z}{\partial w}$, we can do so by applying the chain rule of calculus recursively. It follows that

$$\frac{\partial z}{\partial w} = \frac{\partial z}{\partial y}\left(\frac{\partial y}{\partial x}\frac{\partial x}{\partial w} + \frac{\partial y}{\partial v}\frac{\partial v}{\partial w}\right) \tag{4.3.7}$$

$$= \frac{\partial z}{\partial y}\left(\frac{\partial y}{\partial x} + \frac{\partial y}{\partial v}\right)\frac{\partial x}{\partial w}, \quad \text{because } \frac{\partial x}{\partial w} = \frac{\partial v}{\partial w} \tag{4.3.8}$$

$$= f'(w) \cdot \Big(g^{(1,0)}\big(f(w), f(w)\big) + g^{(0,1)}\big(f(w), f(w)\big)\Big) \cdot f'\big(f(w), f(w)\big). \tag{4.3.9}$$

Where [equation (4.3.7)] describes, what the back-propagation algorithm would do.

4.4 Stochastic gradient descent

Stochastic gradient descent (SGD) is the main *optimization algorithm* that is used in *deep learning*. In *TensorFlow* [6] many of the *optimizers* that come within the deep learning module *keras* are based on the SGD algorithm. As mentioned earlier, the optimizing algorithm is what "really" trains the network by updating the weights.

The underlying gradient descent is an iterative optimization algorithm for nonlinear multi-variable vector-valued functions of arbitrary dimension. Its main idea is to start at an arbitrary point and from there on step into a direction, that minimizes the function value. This way, it gets closer to a minimum with each step. Suppose we have a function $f : \mathbb{R}^2 \to \mathbb{R}$ that is differentiable. For any point x_0 we can evaluate its derivative $f'(x_0)$. And since the derivative describes the slope of the function, it tells us if we have to increase or decrease x_0 in order to make $f(x_0)$ smaller.

Therefore we can reduce $f(x)$ by taking a small step into the opposite direction of the derivative [7], since

$$f(x - \epsilon \operatorname{sign}(f'(x))) < f(x), \qquad \text{for a small enough } \epsilon. \tag{4.4.1}$$

In the following, we will expand this beyond the 2-dimensional case, so that it can be used for neural networks of arbitrary size.

Let $x \in \mathbb{R}^n$ and $f : \mathbb{R}^n \to \mathbb{R}^m$. The directional derivative of f in direction u, a unit vector, is the slope of f in direction u. The directional derivative of $f(x)$ is defined as

$$\frac{\partial}{\partial \alpha} f(x + \alpha u) = u^T \nabla_x f(x)\big|_{\alpha=0}. \tag{4.4.2}$$

We would now like to know in which direction f decreases the fastest. This can be found out by using the directional derivative.

$$\min_{u,u^T u=1} u^T \nabla_x f(x) \tag{4.4.3}$$

$$= \min_{u,u^T u=1} \|u\|_2 \|\nabla_x f(x)\|_2 \cos\theta \tag{4.4.4}$$

$$= \min_{u,u^T u=1} \|\nabla_x f(x)\|_2 \cos\theta, \tag{4.4.5}$$

where $\cos\theta$ is the angle between the gradient and u. By ignoring the gradient, which does not depend on u, this simplifies further to $\min_u \cos\theta$. Since $\cos(\pi) = -1$, this is minimized if the gradient and the unit vector point in opposite directions. This means the negative gradient points the steepest way downhill.

Therefore taking a step in direction of $-\nabla_x f(x)$ is an improvement. It follows that

$$\|f(x - \epsilon\, \nabla_x f(x))\| < \|f(x)\|, \qquad \text{for a small enough } \epsilon, \tag{4.4.6}$$

expandes the gradient descent algorithm to arbitrary dimension. This is also called *method of steepest descent*. Notice that we have to calculate the gradient for each step. For functions that take a lot of inputs (e.g. DNNs), this is computationally expensive.

Stochastic gradient descent. As its name proposes, this algorithm takes a stochastic approach to avoid the complete calculation of the gradient for functions that can be expressed as a sum.

In the context of training DNNs, we often have *cost functions* that decompose into sums. We can exploit this, to only calculate the gradient for a partial sum of the cost function.

A *minibatch* is a small subset of training examples, of usually just a few hundred. We can then calculate the gradient only for the partial cost function defined by this *minibatch*. By approximating our gradient this way, we may fit a model onto millions of examples way cheaper.

Let $Q(w)$ be a function of the form $Q(w) = \sum_{i=1}^{n} Q_i(w)$, where we want to estimate w, so that it minimizes $Q(w)$.

An estimated gradient for a *minibatch* of size m could then formed as

$$\boldsymbol{g} = \eta \nabla \sum_{i=1}^{m} Q_i(w), \tag{4.4.7}$$

where η describes the step size, in the context of deep learning we call η the *learning rate*.

Algorithm 1: Stochastic gradient descent

Input: $Q(w) = \sum_{i=1}^{n} Q_i(w)$
Output: w that minimizes $Q(w)$
Choose initial w;
while $\|Q(w)\| \geq \epsilon$ **do**
  Randomly shuffle dataset entries;
  for $i \leftarrow 1$ **to** n **do**
    $w \leftarrow w - \eta \nabla Q_i(w)$;
  end
end

The SGD algorithm does not only allow more efficient training, but furthermore to train an existing model, which is possible due to the minibatch procedure.

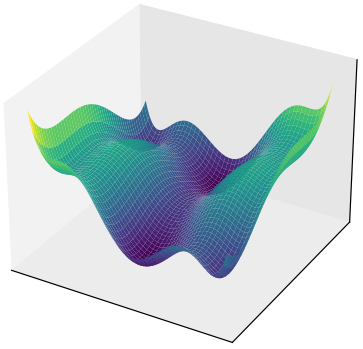

Fig. 4.3. Plot of the six hump camel function: $f(\mathbf{x}) = \left(4 - 2.1x_1^2 + \frac{x_1^4}{3}\right)x_1^2 + x_1x_2 + (-4 + 4x_2^2)x_2^2$

In [fig. 4.4] we see a nonlinear 3-dimensional function, with global and local minima. The difference between gradient descent and its stochastic extension can be easily visualized. Gradient descent always finds the shortest, steepest path, whereas SGD takes a longer path with many turns. A great analogy is one of a drunken man, who tries to find his way down from a mountain into a valley. Gradient descent would then be a sober man.

5 State of the art

Modern neural network implementations such as Keras in TensorFlow [6] are used in a broad variety of fields. Research interest in neural networks has been on a rise for the last 40 years and it seems like this trend is not going to stop anytime soon. With computers getting constantly faster and new different types of neural networks coming up, the number of fields in which neural networks are used also increases.

Especially in the last years, there has been a huge interest and development in convolutional neural network. Modern optimization algorithms such as *Ftrl*

[8] and new activation functions like *swish* [4] by Google are increasing the performance of them even more.

6 Conclusion

We have elaborated on the structure of feedforward networks and the *linear algebra* behind forward propagation. From there we have taken a look at the maximum likelihood principle, that helps us with finding a good cost function. The gradient of the cost function can then be computed with the back-propagation algorithm. This gradient can then be passed to an optimizer such as stochastic gradient descent, that adjusts the weights in order to minimize the error of the DNN. However, this is just the theoretical side of things. A real problem with neural networks is their practical nature. It is often a "trial and error" to determine what works well. Theoretically analyzing why it works well, is most of the time incredibly laborious and very complex.

References

1. Goodfellow, I., Bengio, Y., Courville, A.: Deep Learning. MIT Press (2016), http://www.deeplearningbook.org
2. Nair, V., Hinton, G.E.: Rectified linear units improve restricted boltzmann machines. In: ICML (2010), https://www.cs.toronto.edu/~fritz/absps/reluICML.pdf
3. Glorot, X., Bordes, A., Bengio, Y.: Deep sparse rectifier neural networks. vol. 15 (01 2010), https://mila.quebec/wp-content/uploads/2019/08/glorot11a.pdf
4. Ramachandran, P., Zoph, B., Le, Q.V.: Searching for activation functions. CoRR **abs/1710.05941** (2017), http://arxiv.org/abs/1710.05941
5. Cramer, E., Kamps, U.: Grundlagen der Wahrscheinlichkeitsrechnung und Statistik : ein Skript für Studierende der Informatik, der Ingenieur- und Wirtschaftswissenschaften; 3., überarb. Aufl. SpringerLink: Springer e-Books, Springer, Berlin (2014). https://doi.org/10.1007/978-3-662-43973-9, https://publications.rwth-aachen.de/record/444796
6. Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G.S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., Zheng, X.: TensorFlow: Large-scale machine learning on heterogeneous systems (2015), http://tensorflow.org/, software available from tensorflow.org
7. Cauchy, A.: Méthode générale pour la résolution de systèmes d'équations simultanées (1847)
8. H. Brendan McMahan, Gary Holt, D.S.: Ad click prediction: a view from the trenches https://static.googleusercontent.com/media/research.google.com/de//pubs/archive/41159.pdf