Prof. Dr. Jochen Garcke
Dr. Bastian Bohn
Dr. Ivan Lecei

4 DEEP NEURAL NETWORKS

Send your solutions to this chapter's tasks until June 3rd.

After having considered unsupervised learning methods on sheet 3, we now come back to supervised learning once more. Recall that we are given data $\mathcal{D} := \{(\mathbf{x}_i, y_i) \in \Omega \times \Gamma \mid i = 1, \dots, n\}$ drawn i.i.d. according to some measure μ and we are looking for a function f, which approximately achieves $f(\mathbf{x}) = y$ for $(\mathbf{x}, y) \sim \mu$. As before, we tacitely assume $\Omega \subset \mathbb{R}^d$ and $\Gamma \subset \mathbb{R}$. We now turn to the model class of *(artificial) neural networks* and especially *deep neural networks* (DNN). This class is very popular in machine learning nowadays and a vast zoo of specific types of DNNs exists, which are used for many different tasks such as speech recognition, automated video sequencing, graph learning or image generation, see e.g. http://www.asimovinstitute.org/neural-network-zoo/ and [**bengio**, **bronstein**].

The basic idea of a neural network is to model the way in which information is propagated between neurons in the human brain. Specifically, they are built on the analogon of sending an electrical signal along a neuron synapse. In an artificial neural network model, certain neurons are connected to each other and - based on the state of a neuron - a signal is passed along these connections to adjacent neurons. Depending on how important a connection is, it is given a certain weight. We will stick here to the class of *feedforward* networks, which means that information is passed only in one direction.

4.1 A SINGLE-LAYER FEEDFORWARD NETWORK

A feedforward neural network can directly be modeled as a specific directed acyclic graph. In the easiest case, we are dealing with a *single-layer* neural network[1], see figure 4.1. For this simple network model, the i-th neuron of the *input layer* contains the component z_i of an input vector $\mathbf{z} \in \mathbb{R}^d$. Then it propagates this information to the single *output layer* neuron by multiplying it with the connection weight w_i. At the output neuron, the propagated information of all input neurons is summed up and a *bias* b is added. The result is

$$f(\mathbf{z}) = \sum_{i=1}^{d} w_i z_i + b.$$

1 Note that the output layer is usually not counted. Therefore, we refer to this as a single- or one-layer neural network.

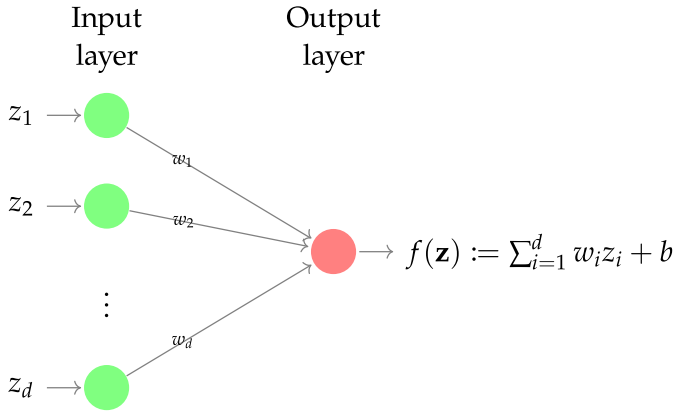

Figure 4.1: A one-layer neural network, which gets a vector $\mathbf{z} = (z_1, \ldots, z_d)$ as input and computes $f(\mathbf{z})$ by propagating it to the output layer.

Viewing the weights w_i for $i = 1, \ldots, d$ and the bias b as degrees of freedom, we already know the model class of f very well: This is exactly the class of affine linear functions. Thus, the model class that can be represented by this most simple neural network is the class of affine linear functions. To obtain a classifier, usually a nonlinear *activation function* $\phi : \mathbb{R} \to \mathbb{R}$ is applied to the result in the output layer. The most simplest one in the case of two classes $\Gamma = \{0, 1\}$ would be the heaviside function

$$\phi(t) = \begin{cases} 1 & \text{if } t > 0 \\ 0 & \text{else} \end{cases}$$

for which we obtain the so-called *perceptron* neural network, which was invented by F. Rosenblatt in 1957, see [**rosenblatt**]. It resembles the first step for machine learning with neural networks. While this is a nice fact per se, we already exhaustively dealt with this model class on the first sheets. To create a broader model class, we will now add more layers (so-called *hidden layers*) in between the input and the output layer to the network.

4.2 A TWO-LAYER FEEDFORWARD NETWORK

Let us now consider a more involved two-layer network, see figure 4.2. This network consists of an input layer with $d_1 := d$ neurons, one hidden layer with d_2 neurons and an output layer with a single neuron. For an input $\mathbf{z} \in \mathbb{R}^d$ the network does the following:

- Neuron i of the input layer gets the data z_i of the input vector.
- The information z_i is passed on by neuron i of the input layer to neuron j of the subsequent layer multiplied by the weight $w_{i,j}^{(1)}$.

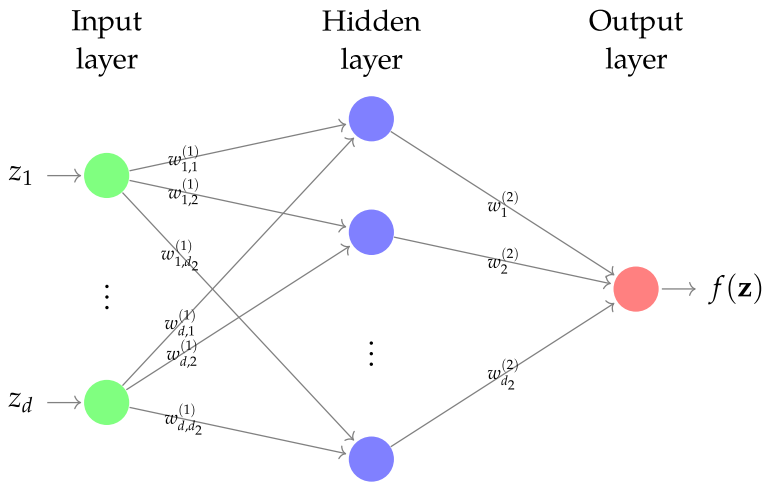

Figure 4.2: A fully-connected two-layer neural network (one hidden layer), which gets a vector $\mathbf{z} = (z_1, \ldots, z_d)$ as input and computes $f(\mathbf{z})$ by propagating the input through the network architecture. Here, the hidden layer has d_2 neurons.

- All the information $w^{(1)}_{i,j} \cdot z_i$ for all $i = 1, \ldots, d$ that arrives in neuron j of the hidden layer is summed up and a bias $b^{(2)}_j$ is added to create the so-called *net* sum:

$$\text{net}^{(2)}_j := \sum_{i=1}^{d} w^{(1)}_{i,j} z_i + b^{(2)}_j.$$

- The net sum is taken as input for an activation function $\phi^{(2)} : \mathbb{R} \to \mathbb{R}$. Thus, $o^{(2)}_j := \phi^{(2)}(\text{net}^{(2)}_j)$ is the information computed (and stored) in neuron j of the hidden layer.

- Now each neuron j of the hidden layer passes its information $o^{(2)}_j$ to each neuron of the next layer. In our case, this is the output layer, which only consists of a single neuron. Again, the information is multiplied by the corresponding weight $w^{(2)}_j$.

- The information that arrives in the output layer is summed up to

$$\text{net}^{(3)} := \sum_{j=1}^{d_2} w^{(2)}_j o^{(2)}_j + b^{(3)},$$

  where $b^{(3)}$ is the bias of the output neuron.

- We apply a final activation function $\phi^{(3)}$ to obtain

$$\begin{aligned} f(\mathbf{z}) = o^{(3)} &:= \phi^{(3)}(\text{net}^{(3)}) = \\ &= \phi^{(3)}\left(\sum_{j=1}^{d_2} w^{(2)}_j \cdot \phi^{(2)}\left(\sum_{i=1}^{d} w^{(1)}_{i,j} z_i + b^{(2)}_j\right) + b^{(3)}\right). \end{aligned}$$

This is called *forward propagation* of the information/input $\mathbf{z}$ and it computes the output $f(\mathbf{z})$ according to the neural network f defined by the architecture from figure 4.2. Note that, for regression, we usually choose the final activation function to be $\phi^{(3)} := \mathrm{id}$, so $o^{(3)} = \mathrm{net}^{(3)}$. Obviously, the model class from which f stems is now much more involved than in the single-layer case.

4.3 DEEP NEURAL NETWORKS

We can directly see that, the more hidden layers we add to the network, the more involved and complicated the structure of f gets. Networks with more than one hidden layer are usually referred to as *deep* neural networks.

4.3.1 *Computing point evaluations of f – forward propagation*

For a given vector $\mathbf{z} \in d$, we want to compute $f(\mathbf{z})$, where f is the function given by the neural network. The generalization from the two-layer case to the L-layer case with $L > 2$ is straightforward: Given the values $o_i^{(l)}$ for $i = 1, \ldots, d_l$ computed in the neurons of the l-th layer[2], we obtain the values in the $l+1$-th layer by computing the net sum of the j-th neuron by

$$\mathrm{net}_j^{(l+1)} := \sum_{i=1}^{d_l} w_{i,j}^{(l)} o_i^{(l)} + b_j^{(l+1)}$$

and applying the activation function of the $l+1$-th layer to get

$$o_j^{(l+1)} := \phi^{(l+1)}(\mathrm{net}_j^{(l+1)}).$$

This procedure is then iterated until we reach the output layer. Note that we can also write this in a matrix-vector-fashion

$$\vec{\mathrm{net}}^{(l+1)} := \left(\mathbf{W}^{(l)}\right)^T \cdot \vec{o}^{(l)} + \vec{b}^{(l+1)}$$

with weight matrix entries $\mathbf{W}_{ij}^{(l)} := w_{i,j}^{(l)}$. Often - by abusing notation - you will see the application of the activation function written as

$$\vec{o}^{(l+1)} := \phi^{(l+1)}(\vec{\mathrm{net}}^{(l+1)}),$$

where the application of $\phi^{(l+1)}$ is meant component-wise.

Apart from classic activation functions such as $\phi^{(l)}(z) = \tanh(z)$ or $\phi^{(l)}(z) = \frac{1}{1+e^{-z}}$, the most famous one in recent artificial neural networks is the so-called *rectified linear unit* or just *ReLU*-function $\phi^{(l)}(z) = \mathrm{ReLU}(z) := \max(0, z)$.

2 For the case $l = 1$, we just set $o_i^{(1)} := z_i$ for $i = 1, \ldots, d_1$ with $d_1 = d$.

While deep neural networks have already been studied in the 1960s, their popularity in machine learning emerged only in the last 10 years, due to the fact that adequate hardware and efficient training algorithms to determine the weights and biases have been missing earlier.

4.3.2 *Least-squares error minimization*

Finally, let us have a look at how to train a neural network, i.e. how to determine the weights and biases. To this end, we will again aim to minimize the least-squares loss function

$$\frac{1}{n}\sum_{i=1}^{n} C_i(f) := \frac{1}{n}\sum_{i=1}^{n}(f(\mathbf{x}_i) - y_i)^2 \tag{4.1}$$

for the neural network model f. For $L > 2$ the minimization problem is nonlinear and (usually) nonconvex, which makes the mathematical and numerical treatment much harder than in the case of linear models. If a minimizer of (4.1) exists, it is usually not even unique and - for deep networks with large L - many local minimizers exist. Nevertheless, many numerical experiments in the last decades have shown that gradient-based minimization algorithms such as quasi-Newton algorithms or even simple descent methods lead to very good results when employed to solve (4.1). Note that we already employed a gradient descent algorithm for the most simple neural network model class, namely the affine linear one, on sheet 1. To compute the gradient w.r.t. the weights and biases of (4.1), we will use the so-called *backward propagation* or *backprop* algorithm, which is based on the chain rule.

4.3.3 *Computing the gradients of C_i – backward propagation*

Since the one-point loss C_i for $i = 1, \dots, n$ is just a random instance of $C(f) := (f(\mathbf{x}) - y)^2$ for $(\mathbf{x}, y) \sim \mu$, we will focus on computing

$$\frac{\partial C(f)}{\partial w_{i,j}^{(l)}} \text{ and } \frac{\partial C(f)}{\partial b_j^{(l+1)}} \quad \forall\, i = 1, \dots, d_l \text{ and } j = 1, \dots, d_{l+1}$$

for $l = 1, \dots, L$.

Note that the chain rule gives us

$$\begin{aligned}\frac{\partial C(f)}{\partial w_{i,j}^{(l)}} &= \frac{\partial C(f)}{\partial o_j^{(l+1)}} \cdot \frac{\partial o_j^{(l+1)}}{\partial \operatorname{net}_j^{(l+1)}} \cdot \frac{\partial \operatorname{net}_j^{(l+1)}}{\partial w_{i,j}^{(l)}} \\ &= \frac{\partial C(f)}{\partial o_j^{(l+1)}} \cdot \left(\phi^{(l+1)}\right)' \left(\operatorname{net}_j^{(l+1)}\right) \cdot o_i^{(l)}.\end{aligned}$$

For the first term we have

$$\frac{\partial C(f)}{\partial o_j^{(l+1)}} = \begin{cases} 2(f(\mathbf{x}) - y) & \text{if } l = L, \\ \sum_{i=1}^{d_{l+2}} \frac{\partial C(f)}{\partial \operatorname{net}_i^{(l+2)}} \cdot \frac{\partial \operatorname{net}_i^{(l+2)}}{\partial o_j^{(l+1)}} & \text{else.} \end{cases}$$

Since $\frac{\partial \operatorname{net}_i^{(l+2)}}{\partial o_j^{(l+1)}} = w_{j,i}^{(l+1)}$ and

$$\frac{\partial C(f)}{\partial \operatorname{net}_i^{(l+2)}} = \frac{\partial C(f)}{\partial o_i^{(l+2)}} \cdot \frac{\partial o_i^{(l+2)}}{\partial \operatorname{net}_i^{(l+2)}} = \frac{\partial C(f)}{\partial o_i^{(l+2)}} \cdot \left(\phi^{(l+2)}\right)' \left(\operatorname{net}_i^{(l+2)}\right),$$

we can calculate $\frac{\partial C(f)}{\partial w_{i,j}^{(l)}}$ starting at the final layer and iterate backwards step by step. This process is called *backward propagation* or simply *backprop*. To this end, let us iteratively define

$$\delta_j^{(l)} := \begin{cases} 2(f(\mathbf{x}) - y) = 2(o_1^{(L+1)} - y) & \text{if } l = L, \\ \sum_{i=1}^{d_{l+2}} \delta_i^{(l+1)} \cdot \left(\phi^{(l+2)}\right)' \left(\operatorname{net}_i^{(l+2)}\right) \cdot w_{j,i}^{(l+1)} & \text{else.} \end{cases}$$

Then, it follows

$$\frac{\partial C(f)}{\partial w_{i,j}^{(l)}} = \delta_j^{(l)} \cdot \left(\phi^{(l+1)}\right)' \left(\operatorname{net}_j^{(l+1)}\right) \cdot o_i^{(l)}$$

for $l = 1, \ldots, L$. Analogously, one can show

$$\frac{\partial C(f)}{\partial b_j^{(l+1)}} = \delta_j^{(l)} \cdot \left(\phi^{(l+1)}\right)' \left(\operatorname{net}_j^{(l+1)}\right).$$

Again, we can write this down in a matrix-vector format by

$$\vec{\delta}^{(l)} := \begin{cases} 2(f(\mathbf{x}) - y) & \text{if } l = L, \\ \mathbf{W}^{(l+1)} \cdot \left(\vec{\delta}^{(l+1)} \odot \left(\phi^{(l+2)}\right)' \left(\vec{\operatorname{net}}^{(l+2)}\right)\right) & \text{else,} \end{cases}$$

which gives us the derivatives

$$\begin{aligned} \nabla_{\mathbf{W}^{(l)}} C(f) &= \vec{o}^{(l)} \cdot \left(\vec{\delta}^{(l)} \odot \left(\phi^{(l+1)}\right)' \left(\vec{\operatorname{net}}^{(l+1)}\right)\right)^T && \in \mathbb{R}^{d_l \times d_{l+1}}, \\ \nabla_{\vec{b}^{(l+1)}} C(f) &= \vec{\delta}^{(l)} \odot \left(\phi^{(l+1)}\right)' \left(\vec{\operatorname{net}}^{(l+1)}\right) && \in \mathbb{R}^{d_{l+1}}, \end{aligned}$$

where $\odot$ denotes the Hadamard product.

This shows us, how the derivatives for one data tuple $(\mathbf{x}, y)$ can be computed. Doing this for all $(\mathbf{x}_i, y_i)$ with $i = 1, \ldots, n$ gives us the derivatives of (4.1) w.r.t. the weights and biases. This enables us to employ a gradient descent algorithm.

4.3.4 *Training the network – stochastic (minibatch) gradient descent*

Let us now introduce an advanced variant of the gradient descent optimizer we had on sheet 1. Instead of using a standard gradient descent optimizer, which can be costly for large data sets, we will use a stochastic variant, where a subset $B \subset \{1, \ldots, n\}$ is chosen randomly

in each iteration step and the gradient $\nabla_{w,b}$ w.r.t. all weights and biases of

$$C_B(f) := \frac{1}{|B|} \sum_{i \in B} C_i(f)$$

is computed instead of the gradient of (4.1). This is much cheaper if $|B| \ll n$ and gives an unbiased estimate of the gradient of $C(f)$ since the data is drawn i.i.d. according to μ. The overall stochastic minibatch gradient descent algorithm for the minimization of (4.1) is given in algorithm 4.4.

Algorithm 4.4 Stochastic gradient descent algorithm to determine a minimizer of (4.1).

Input: Data set $\mathcal{D}$, learning rate $\nu > 0$, minibatch size K, number of steps S.

for all $s = 1, \dots, S$ **do**
  Draw a random set $B \subset \{1, \dots, n\}$ of size K.
  Calculate $f(\mathbf{x}_i)\ \forall\, i \in B$ via forward propagation.
  Calculate $\nabla_{w,b} C_B(f)$ via backprop.
  Update the weights and biases
    $\mathbf{W}^{(l)} \leftarrow \mathbf{W}^{(l)} - \nu \cdot \nabla_{\mathbf{W}^{(l)}} C_B(f)$,
    $\vec{b}^{(l+1)} \leftarrow \vec{b}^{(l+1)} - \nu \cdot \nabla_{\vec{b}^{(l+1)}} C_B(f)$
  for all $l = 1, \dots, L$.
end for

Task 4.1. *Implement a class* `TwoLayerNN`*, which represents a (fully-connected, feed-forward) two-layer neural network, i.e.* $L = 2$*. The activation functions should be* $\phi^{(2)} = \mathrm{ReLU}$ *and* $\phi^{(3)} = \mathrm{id}$*. The weights and biases can be initialized by drawing i.i.d. uniformly distributed random numbers in* $(-1, 1)$*. The class should contain a method* `feedForward` *to calculate the point evaluations of* f *for a whole minibatch at once and a method* `backprop` *to calculate* $\nabla_{w,b} C_B(f)$*. To this end, avoid using for-loops over the minibatch and use linear algebra operations (on vectors, matrices or tensors) instead.*

Task 4.2. *Augment the* `TwoLayerNN` *class by implementing methods to randomly draw a minibatch data set and a method to perform the stochastic minibatch gradient descent algorithm.*

Task 4.3. *Test your implementation by drawing* 250 *uniformly distributed points* $\mathbf{x}_i$ *in* $\mathbb{R}^2$ *with norm* $\|\mathbf{x}_i\| \leq 1$ *and label them by* $y_i = -1$*. Now draw* 250 *uniformly distributed points* $\mathbf{x}_i$ *in* $\mathbb{R}^2$ *with* $1 < \|\mathbf{x}_i\| \leq 2$ *and label them by* $y_i = 1$*. Use your two-Layer neural network with* $d_2 = 20$ *hidden layer neurons,* $S = 50000$ *iterations and* $K = 20$ *to classify the data. Try different learning rates* ν*. Output the least-squares error every* 5000 *iterations. After* S *iterations, make a scatter plot of the data and draw the contour line of your learned classifier. What do you observe? What happens if you increase* S*?*

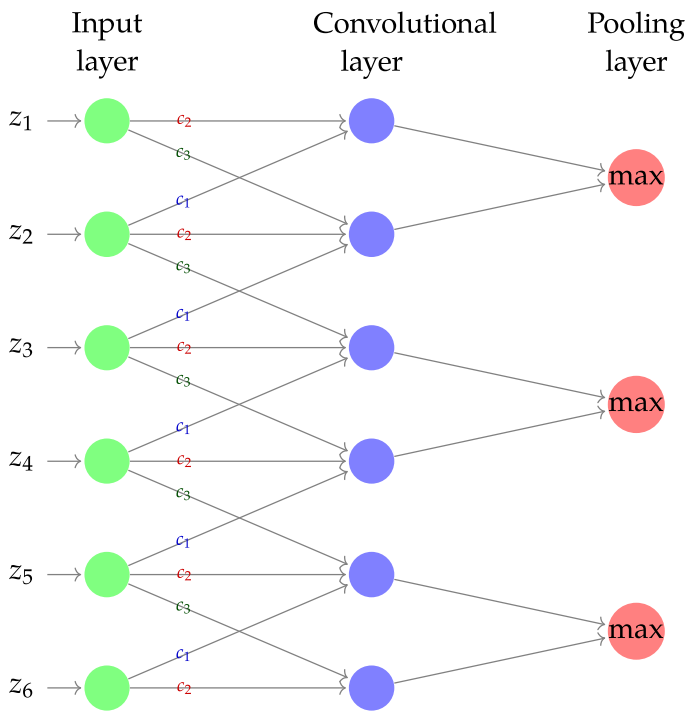

Figure 4.3: A convolutional layer with three (shared) weights c_1, c_2, c_3 applied to $\mathbf{z} = (z_1, \ldots, z_6)$ followed by a downsampling/max-pooling layer with 3 neurons.

4.4 RELATION TO OTHER METHODS / MODELS

Finally, let us remark some analogies of (deep) neural networks to other models/methods.

4.4.1 *A relation to kernel methods*

While the weights and the biases are degrees of freedom of our model class, the (possibly nonlinear) activation functions $\phi^{(l)}$ are usually fixed a priori. Let $\phi^{(L+1)} = \mathrm{id}$. Since we are only dealing with a single output neuron, we can rewrite our model as

$$f(\mathbf{z}) = \left(\vec{w}^{(L)}\right)^T \cdot \psi(\mathbf{z}) + b^{(L+1)}$$

for a vector-valued function $\psi : \mathbb{R}^d \to \mathbb{R}^{d_L}$ depending on the weights, biases and activation functions of the previous $l = 1, \ldots, L-1$ layers. In this way, we have a direct analogy to SVM or kernel methods in general, where ψ reflects the feature map which is chosen to transform the data. However, the difference between the SVM and the hidden-layer neural network model is that ψ—or equivalently the kernel K—has been chosen a priori for SVM, whereas here ψ depends on the degrees of freedom (namely the weights and biases of the hidden

layers). Furthermore, there also exist hybrids between kernel methods and deep neural networks called *deep kernel networks*, see e.g. [**bohn**].

4.4.2 *A relation to ordinary differential equations (ODEs)*

A special class of neural networks are so-called *residual networks* (ResNets). Here, the forward propagation step can be written in the form

$$\vec{o}^{(l+1)} = \vec{o}^{(l)} + \Delta t \cdot \phi^{(l+1)} \left(\left(\mathbf{W}^{(l)} \right)^T \cdot \vec{o}^{(l)} + \vec{b}^{(l+1)} \right),$$

where $\Delta t > 0$ is some positive scaling parameter. If we assume that the activation function is the same in each iteration step, this can be written as

$$\frac{\vec{o}^{(l+1)} - \vec{o}^{(l)}}{\Delta t} = \phi \left(\left(\mathbf{W}^{(l)} \right)^T \cdot \vec{o}^{(l)} + \vec{b}^{(l+1)} \right). \tag{4.2}$$

Therefore, we can reinterpret this as a time-explicit Euler discretization with step width Δt for the ODE

$$\dot{\vec{o}}(t) = \phi \left(\mathbf{W}^T(t) \vec{o}(t) + \vec{b}(t) \right). \tag{4.3}$$

Therefore, a stable forward propagation can only be guaranteed if the ODE itself admits stable solutions. From ODE theory, it is well-known that this holds if the real part of the eigenvalues of the Jacobian J of the RHS of (4.3) are non-positive. Furthermore, it is needed that the explicit Euler scheme is stable, which holds if

$$|1 + \Delta t \cdot \lambda_i(\mathbf{J}^{(l)})| \leq 1 \quad \forall\, l = 1, \ldots, L-1$$

is fulfilled for all eigenvalues $\lambda_i(\mathbf{J}^{(l)})$ of the l-th layer Jacobian $J^{(l)}$ of the RHS of (4.2). This reinterpretation motivates the creation of new forms of neural networks which allow for stable evaluations, e.g. Hamiltonian-based networks with a leapfrog-type discretization, see [**haber**].

4.5 CONVOLUTIONAL NEURAL NETWORKS

By many experts, the breakthrough in deep learning is considered to be a model published in 2012. It was submitted to the *ImageNet Large Scale Visual Recognition Challenge*[3] (ILSVRC) by a team around Alex Krizhevsky [**krizhevsky2012imagenet**]. One remarkable observation about this model is, that it used a machine learning technique which

In the literature the model is referred to as AlexNet.

3 An annual challenge using a dataset of 1.2 million images from various sources. Each image depicts objects from 1000 categories which were hand–labeled using Amazon Turk. The collection of the ImageNet dataset is a great achievement by itself.

was not very popular at that time. Therefore, none of the other entries did use it. The second remarkable thing is, that it did beat its competitors by over ten percent points[4] in the error rate.

The model is built using a class of neural networks called *(Deep) Convolutional Neural Networks* (DCNN), which we will focus on in this section.

When using the HoG features in section 3.4.1 we saw that the differentiation could be implemented using an image convolution with a kernel matrix. In a CNN such convolutions are used as layers in a neural network. While the weights for the convolution in the HoG features were chosen by hand, the weights in a CNN are free parameters and therefore subject to change during training.

For example, let $c_1, \ldots, c_{2n+1}$ denote weights, and suppose we have d input nodes $z_1, \ldots, z_d$. If the first hidden layer is a convolutional layer its inputs are computed by

$$\text{net}_j^{(2)} := \sum_{k=-n}^{n} z_{j+k} c_{n+1-k}$$

where the values z_l for $l > d$ or $l < 1$ are set to zero. In this way, weights are shared in comparison to a *fully-connected layer* as previously described. An example for $d = 6$ and $n = 1$ is depicted in figure 4.3. Usually more than one convolutional layer is used in parallel, resulting in a multidimensional output. Also, the input dimension can be arbitrary.

CNNs are especially popular for images. While we previously flattened the image and used, for example, the pixels row-wise as input to a classifier, we can now keep the 2D representation. The advantage is that locality information is available to the net in this way.

This can be observed empirically.

In many cases a stack of convolutional layers is used at the beginning of a network to learn features. Here, one expects the layers to learn higher and higher abstractions down the network. For example one convolution at the beginning might encode lines in certain directions, another one encodes circles, while deeper layers use these pieces of information to recognize, say, body parts.

A certain type of activation functions is usually used for convolutional layers. Here, a technique called *subsampling* is employed. The idea is to apply a function to the convolution outputs in order to make the network invariant under certain transformations. For example, the common *max-pooling* selects the maximum values among all (possibly overlapping) square regions of given size. In this way the network is expected to be stable under small translations. Other options are for example ℓ_1 or ℓ_2 averages. A neat side effect is that this decreases the number of neurons if the regions do not overlap too much. In practice

4 The results can be found here: http://www.image-net.org/challenges/LSVRC/2012/results.html.

max-pooling is implemented as an additional layer, see figure 4.3 for a non-overlapping max-pooling example.

After applying some convolutional layers, the input is often flattened into a one-dimensional feature vector and then fed into a final fully-connected layer for an M-class classification problem ($M = |\Gamma|$). The output layer is then made of M nodes, representing a probability distribution . This is achieved for example by the *softmax* activation function,

The labels are also expected to be in this form.

$$o_j^{(L+1)} := \frac{\exp\left(\text{net}_j^{(L+1)}\right)}{\sum_{i=1}^{d_{L+1}} \exp\left(\text{net}_i^{(L+1)}\right)},$$

where $d_{L+1} = M$.

Finally, a suitable choice has to be made for the loss function (optimization objective). Matching the softmax function, *log-likelihood* is a popular choice leading to maximum likelihood estimation (MLE)

$$\max_{w,b} \sum_{i=1}^{M} \log(\mathbb{P}[y_i|\mathbf{x}_i, w, b]).$$

We will use the so called *cross entropy* loss, which is related to MLE.

4.5.1 *Keras*

For the final tasks we use the KERAS library [**chollet2015keras**]. It provides a high-level and easy to use abstraction for popular deep learning backends such as Tensorflow.

Defining the network from task 4.3 could be done like this

```
import keras
import keras.layers as layers

model = keras.models.Sequential()
model.add(layers.Dense(20, input_shape=(2,),
          activation='relu'))
model.add(layers.Dense(1))
```

You can print a summary with `print(model.summary())`.

Continue with compiling the model

`mse` stands for mean squared error, cf. eq. (4.1), `sgd` is an abbreviation for algorithm 4.4.

```
model.compile(loss='mse', optimizer='sgd',
              metrics=['accuracy'])
```

and then start training by

```
K = 20
S = 50000
history = model.fit(X_train, Y_train, batch_size=K,
                    epochs=S / (500 / K),
                    verbose=True)
```

The gradient implementation[5] is derived using automatic differentiation. In contrast to numeric differentiation (derivative approximated, e.g. by computing the difference quotient) here the symbolic knowledge about the network is used. This works similar to a computer algebra system (CAS) like Mathmematica, Maple, Octave, etc.

KERAS has support for several gradient descent variants, which can also be configured (e.g. change step size). Of course, the final layer can have an activation function too, e.g. `softmax` when used for classification.

A convolutional layer with a ReLU activation can be added with

```
model.add(layers.Conv2D(16, kernel_size=(3, 3),
                        activation='relu'))
```

which adds 16 parallel layers (i.e. 16 layers of the same shape at the same position[6]) each with a 3×3 convolutional matrix for 2D input. To flatten the result for classification use

```
model.add(layers.Flatten())
```

4.5.2 *Regularization*

To prevent over-fitting many techniques are known, but most of them are not well understood. A very popular technique is called *dropout*, where – during each training step – one neglects nodes in a layer with a given probability p. In this way, random sub-nets are trained. To add dropout regularization to a layer use

```
model.add(layers.Dropout(p))
```

after it.

Keras also has support for using regularization terms in the loss function, similar to what we have seen when discussing SVMs.

Task 4.4. *Use KERAS to build a classifier for the MNIST dataset (see template notebook).*

(a) Build a model with the following layers:

- *Fully-connected layer (`Dense`) with 128 output nodes + ReLU.*
- *Fully-connected layer with 128 output nodes + ReLU.*
- *Fully-connected layer with 10 output nodes + softmax.*

Use the SGD optimizer with a batch size of 128 and the categorical crossentropy loss (`loss="categorical_crossentropy"`), train for 20 episodes. Use the accuracy metric (set in `model.compile`) and provide the test data as validation data to `model.fit` (set the parameter `validation_data` to `(X_test, Y_test)`). Plot the fit history (return value of `model.fit`).

5 The network is at least *piece-wise* differentiable.

6 This can also be understood in the following way: Each neuron of the convolutional layer contains a sixteen-dimensional vector.

(b) *Build a new network by adding dropout ($p = 0.3$) to the first and second layer of the model from (a). Train it for 250 episodes.*

(c) *Build a third network by using the model from (b) with the optimizer* `"adam"` *instead of SGD. Be prepared to roughly explain what this optimizer does. Train for 20 episodes.*

Task 4.5. *Build a CNN with* KERAS *with the following layers:*

- *16 parallel conv. layer with kernel size* 3×3 + *ReLU.*
- *32 parallel conv. layer with kernel size* 3×3 + *ReLU.*
- *A 2D max pooling layer of size* 2×2*, non-overlapping + dropout* ($p = 0.25$).
- *Flatten + fully-connected layer with 128 outputs + ReLU + dropout* ($p = 0.5$)
- *Fully-connected layer with output size 10 + softmax.*

Train for 15 episodes, the other parameters should be the same as in the previous model.

Feel free to change the network/training in order to improve the error.

Task 4.6. *(**Optional bonus task**) Use a CNN to learn features for pedestrian classification. Proceed as follows:*

- *Design and train a CNN for pedestrian classification (use the data from section 3.4). You can start with the network from task 4.5.*
- *Use the output after the flattening (see* KERAS*' FAQ on how to do this) as a feature vector for a linear SVM, together with the HoG features.*

Try to use PCA in order to improve the accuracy, also tweak the HoG parameters. Make sure not to over-fit (the pedestrian dataset is small in deep learning standards). You can also install the AUGMENTOR *library for* PYTHON *in order to enlarge the dataset (common technique in deep learning). We did obtain a 93.5% error rate, but there is definitely room for improvement! You might also want to take a look at the images which got classified wrongly.*

4.6 OUTLOOK

GPUS Keras (more precisely its backends) can take advantage of a GPU (graphics card) in order to speed up the training. If you did the tasks your training was probably using the CPU, even if your machine has a GPU. Setting this up can be challenging. But if successful, the performance gain is usually significant. We did run task 4.5 on six Xeon 3.6 GHz (Sandy Bridge) cores and on a Tesla P100 GPU. For the CPU

one episode of training took 30 seconds (a 2012 Quad-Core laptop took over a minute), while on the GPU an episode was finished in three seconds. Even higher speedups are common (our network is probably too small).

Training a neural network is a lot of try-and-error and needs experience and patience. Modern networks can only be trained (in reasonable time) on a GPU or dedicated hardware. For example, AlexNet had 60 million parameters and was trained over six days on two GPUs.

REFERENCES ON GENERAL NEURAL NETWORKS AND CNNS For further programming resources the free course at fast.ai[7] could be interesting. See also the book [**bengio**] or the review [**rawat2017deep**] on CNNs. There you will also find references to attempts on describing the history of neural networks.

ADVERSARIAL NEURAL NETWORKS Another interesting branch of literature is the one on attacks against neural networks. For example [**brown2017adversarial**] designed a sticker that you can patch onto (or near) objects to make a CNN classify everything as a toaster.[8] In [**evtimov2017robust**] the authors investigate how robust a CNN can detect traffic stop signs when the sign has graffiti on it ("stop eating animals" or fake SOP signs). Finally, [**papernot2016practical**] provides examples for manipulated but visually indistinguishable images of traffic signs which get classified differently by a CNN.

7 http://www.fast.ai/

8 A top Reddit comment argues that the CNN rightfully does so, and that, in fact, the sticker does look like a toaster...or a Hunter S. Thompson version of a toaster.