Artificial Neural Networks
Lecture Notes

Prof. Dr. Sen Cheng

Winter Semester 2019/20

10 Deep neural networks

A deep neural network is simply a neural network with many layers. There is no standard definition of when a network counts as deep. Sometimes a network with $m = 4$ layers is already considered deep. The main challenges with deep neural networks, as compared to shallow neural networks, is that 1. they have a very large number of parameters that need to be trained, and 2. the gradients can vanish/explode as errors are progagated through the many layers. Therefore, special techniques have been developed to deal with these specific issues. These techniques can mostly be applied to shallow networks as well to make training more efficient.

10.1 Convolutional neural network

The more layers there are in a network and the more units there are in a layer, the more complex a function the network can represent. However, one cannot train multiple, fully-connected layers because there are simply too many parameters. Hence, *convolutional neural networks (CNN)* were developed to address these issues. This type of network was inspired by two properties of visual information processing in the brain since these networks are mostly used for image processing.

First, lower layers of the visual processing hierarchy process lower-level representations that are highly variable and hence less informative. For instance, when an image shows a line, the pixel values change with illumination and the specific values of the pixels don't really matter. By contrast, higher layers of the visual processing hierarchy represent higher-level features that are more abstract and more invariant such as the orientation of bar, shapes or semantic categories. Second, units in higher layers process information from a larger visual field, i.e., their receptive fields are larger. In CNN, these two properties, feature extraction and integration, are modelled by *convolution* and *max pooling*, respectively. Together these two operations can reduce network complexity and extract higher-order features that are useful for image recognition. An example of these increasingly complex and abstract features can be seen in Fig. 1.

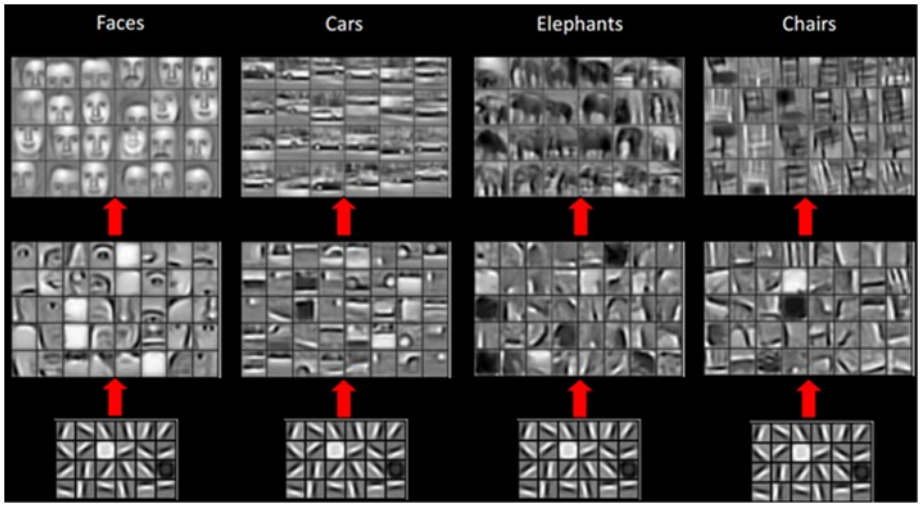


Figure 1: Network represents increasingly complex and abstract features (Siegel et al., 2016, Fig.3).

10.1.1 Convolution

Mathematically, a *convolution* between two one-dimensional functions f and g is defined as

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau) g(t - \tau)\, d\tau \tag{1}$$

One of the functions, e.g. f, is usually considered a *filter* and is a localized function. Discrete variant

$$(f * g)_i = \sum_{i'=0}^{n-1} f_{i'} g_{i-i'} \tag{2}$$

where n is the size of the convolution filter. For instance, calculating a moving average of a time series g (Fig. 2) is like convolving it with a box filter

$$f_i = \begin{cases} \frac{1}{n} & 0 \leq i < n \\ 0 & i \geq n \end{cases} \tag{3}$$

Figure 2: Moving average in 1-d.

Definition of convolution in 2-d:

$$(f * g)_{i,j} = \sum_{i'=0}^{n_v-1} \sum_{j'=0}^{n_h-1} f_{i',j'} g_{i-i',j-j'} \tag{4}$$

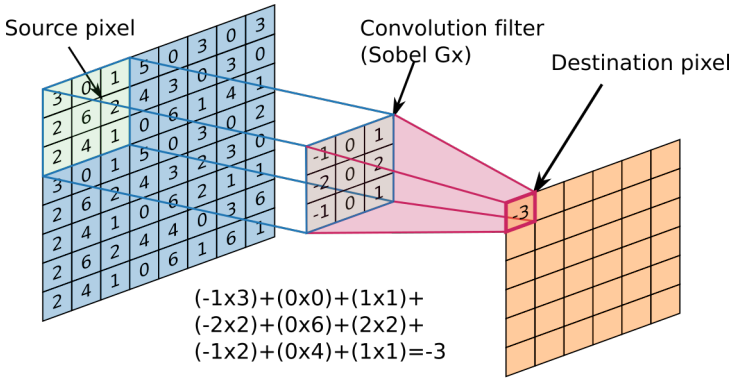
where n_v and n_h are the sizes of the convolution filter in the vertical and horizontal direction, respectively. Example of 2-d convolution Fig. 3a. Just like in 1-d, convolution can be used for smoothing in 2-d, although in image processing it's frequently called blurring instead (Fig. 3b).

Convolutions can be used to extract more abstract features, such as outlines (Fig. 3c) or edges. The latter is called *edge detection*. This is useful for image recognition because these features are more abstract, invariant and informative. In addition, the number of parameters is reduced from the number of pixels in the image to the number of elements in the filter.

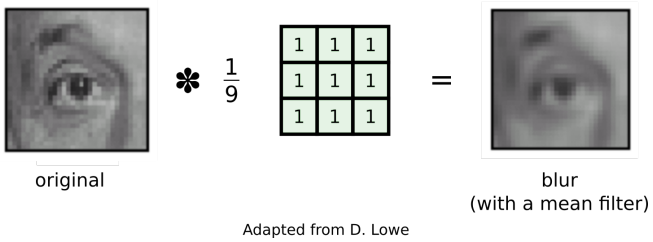
Usually, there are multiple, n_f^l sets of features that are calculated in one convolutional layer, e.g., to represent a color image in RGB space the red, green and blue channels are separately encoded a two dimensional matrix. Each color channel contributes a feature map.

Note that the convolution is defined with a minus sign $i - i', j - j'$. As a result, there is a disconnect between the visualization of the convolution filter $f_{i',j'}$ and its effect in the convolution. In mathematics, the minus sign is required so that the convolution operation is symmetric in f and g. However, in image processing, this creates a lot of confusion. That's why we use a plus sign in *convolutional layers* in an ANN, even though the unit doesn't, strictly speaking, perform a convolution operation. It is called a *correlation* operation. In 2-d, the difference between the kernels in convolution and correlation is a rotation by 180°.

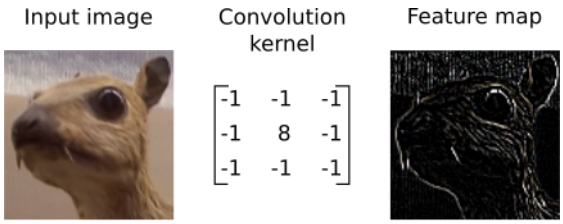
$$a_{i,j,k}^{l+1} = b_k^l + \sum_{i'=0}^{n_i^l-1} \sum_{j'=0}^{n_j^l-1} \sum_{k'=1}^{n_f^l} w_{i',j',k,k'}^l h_{is_v+i',js_h+j',k'}^l \tag{5a}$$



(a) Illustration of 2-d convolution.



(b) Blurring convolution.



(c) Outline convolution.

Figure 3: 2-d convolutions in image processing.

To generate the output, an activation function is applied to each unit.

$$h^l_{i,j,k} = \phi^l\left(a^l_{i,j,k}\right) \tag{5b}$$

- $w^l_{i',j',k,k'}$ convolution kernel. Note that the indices i and j have very different meaning from earlier use; they index the units within the input layer/ kernel in the vertical and horizontal dimension, respectively. Earlier, they referred to units in the input and output layers, respectively. This change is necessary because the weights in CNNs are not uniquely connecting pairs of units (see below).
- n^l_v, n^l_h: size of filter in vertical and horizontal direction, respectively
- n^l_f: number of feature maps
- s^l_v, s^l_h: strides in vertical and horizontal direction, respectively
- b^l_k: overall brightness of the feature map, bias
- m^l_v, m^l_h: size of image/feature map in vertical and horizontal direction, respectively

In Eq. 5a, there is a summation across the feature dimension, no convolution.

The size of the resulting feature map. Another difference to strict convolution is when the strides s^l_i and s^l_j are larger than 1. This reduces the size of the image and is useful when memory during training is limited. This is equivalent to computing the entire convolution, but skipping $s^l_i - 1$ (respectively $s^l_j - 1$) output units when generating the output. That's why the size of the output layer is roughly $s^l_i s^l_j$-times smaller than the full convolution. Image reduction with pooling layers (see below) appear to work better than using larger strides to reduce the size of the image representation. It is therefore more common to use $s^l_i = s^l_j = 1$.

Even when using 1 for the stride, the size of the feature map is smaller than the size of its input: $m^{l+1}_v = m^l_v - n^l_v + 1$ and $m^{l+1}_h = m^l_h - n^l_h + 1$. That is because the convolutional filter is not allowed to stick out. This reduction is undesirable in image processing because that means that the image representation will shrink in every layer and it's always the information around the edges that is lost. To avoid this loss of information, *padding* is used. The image in the input, or the feature map in the hidden layer, is enlarged by $(n^l_v - 1)/2$ and $(n^l_h - 1)/2$, respectively, around

the edges. The additional pixels or the additional hidden units are either set to 0 *zero padding* or the new pixels are filled with copies of the neighboring edge. Padding the input preserve the size of the image representation after the convolution and the image information around the edges.

Weight sharing. In a fully connected network, all the units in layer l are connected to all the units in layer $l+1$. Hence, there are $n^l \times n^{l+1}$ weights that need to be estimated. Each weight connects exactly one pair of input and output units. The reason why CNN use dramatically fewer parameters is *weight sharing*. There is one convolution kernel $w^l_{i',j',k,k'}$ that is shared between all units to compute the features in each layer. In addition, the kernel is usually much smaller than the input image itself, further saving on parameters.

Additional material:

- "Convolutional Neural Network – In a Nut Shell" by Muhammad Rizwan
- "An intuitive guide to Convolutional Neural Networks" by Daphne Cornelisse

10.1.2 Backpropagation in convolutional neural networks

Because of weight sharing, each entry in the weight matrix, i.e. convolution kernel, $w^l_{i',j',k,k'}$ influences every unit's activity in the next layer $h^{l+1}_{i,j,k}$, therefore the gradient has to sum over all these influences.

$$\frac{\partial L}{\partial w^l_{i',j',k,k'}} = \underbrace{\sum_{i=1}^{m_v^{l+1}} \sum_{j=1}^{m_h^{l+1}}}_{\text{weight sharing}} \underbrace{\frac{\partial L}{\partial a^{l+1}_{i,j,k}}}_{\delta^{l+1}_{i,j,k}} \underbrace{\frac{\partial a^{l+1}_{i,j,k}}{\partial w^l_{i',j',k,k'}}}_{h^l_{is_v+i',js_h+j',k'}} = \sum_{i=1}^{m_v^{l+1}} \sum_{j=1}^{m_h^{l+1}} \delta^{l+1}_{i,j,k} h^l_{is_i+i',js_j+j',k'} \tag{6a}$$

$$h^l_{i^*,j^*,k'} \qquad w_{i',j',k,k'} \qquad a^{l+1}_{i,j,k} \tag{6b}$$

If the loss function is differentiable w.r.t. y, then the gradient

$$\delta^m_{i,j,k} = \frac{\partial L}{\partial h^m_{i,j,k}} \frac{\partial h^m_{i,j,k}}{\partial a^m_{i,j,k}} = \frac{\partial L}{\partial h^m_{i,j,k}} (\phi^m)' \left(a^m_{i,j,k}\right) \tag{6c}$$

is defined. For the hidden layers, we have to sum over all activations in layer $a^{l+1}_{i,j,k}$ through which the activation $a^l_{i^*,j^*,k'}$ can propagate to the output. Eq. 5a shows that this occurs if the indices satisfy certain relationships:

$$i^* = is_v + i' \qquad , i' = 0, \ldots, n_v - 1 \tag{6d}$$

$$j^* = js_h + j' \qquad , j' = 0, \ldots, n_h - 1 \tag{6e}$$

Because the convolution limits the spatial range through which one pixel can affect the next layer, we only have to sum over the width of the filter:

$$\delta^l_{i^*,j^*,k'} = \frac{\partial L}{\partial a^l_{i^*,j^*,k'}} = \sum_{i'=0}^{n_v^l-1} \sum_{j'=0}^{n_h^l-1} \sum_{k=1}^{n_f^{l+1}} \frac{\partial L}{\partial a^{l+1}_{i,j,k}} \frac{\partial a^{l+1}_{i,j,k}}{\partial a^l_{i^*,j^*,k'}} \tag{6f}$$

$$= \sum_{i'=0}^{n_v^l-1} \sum_{j'=0}^{n_h^l-1} \sum_{k=1}^{n_f^{l+1}} \delta^{l+1}_{i,j,k} \frac{\partial a^{l+1}_{i,j,k}}{\partial a^l_{i^*,j^*,k'}} \tag{6g}$$

The last derivative is:

$$\frac{\partial a^{l+1}_{i,j,k}}{\partial a^l_{i^*,j^*,k'}} = \frac{\partial a^{l+1}_{i,j,k}}{\partial h^l_{i^*,j^*,k'}} \frac{\partial h^l_{i^*,j^*,k'}}{\partial a^l_{i^*,j^*,k'}} \tag{6h}$$

$$= \left(\phi^l\right)' \left(a^l_{i^*,j^*,k'}\right) w_{i',j',k,k'} \tag{6i}$$

$$\delta^l_{i^*,j^*,k'} = \left(\phi^l\right)' \left(a^l_{i^*,j^*,k'}\right) \sum_{i'=0}^{n^l_v-1} \sum_{j'=0}^{n^l_h-1} \sum_{k=1}^{n^{l+1}_f} \delta^{l+1}_{(i^*-i')/s_v,(j^*-j')/s_h,k} w_{i',j',k,k'} \tag{6j}$$

Notice that the equations for backpropagation in CNN are very similar to the one for backpropagation in fully connected networks. This is because both use only linear operations. The differences between the networks consists in how units are indexed and how inputs are summed. That's why the differences in the backpropagation equations differ in precisely these aspects.

Additional material:

- "Backpropagation in Convolutional Neural Networks" by Jefkine Kafunah

10.1.3 Pooling

A convolution layer extracts features from the input image, however, the size of the feature map remains as large as the input image itself, if padding is used. Since we want the network to learn more abstract, lower-dimensional representations of the input, we have to reduce the size of the image representation. A *pooling layer* accomplishes this by reducing the inputs in a rectangluar, mostly square, receptive field to a single number. Most commonly, the receptive fields are fairly small such as 2×2 with a stride of 2 because usually multiple pooling layers are used in a deep network and the image size reduces exponentially. The input-output function that appears to work the best in most situation is the maximum. This is a so-called *max pooling* layer (Fig. 4a). Other choices include averaging over the inputs (*average pooling*.

(a) Max pooling (b) Average pooling

Figure 4: Example of pooling operations.

Another desirable result of pooling is *translational invariance*. The identity of objects is invariant to translation (and rotation and many other image transformations). Humans and other animals have learned these invariances and can recognize objects under these transformations. However, the mathematical representation of a shifted image I' is very different from the original image I, i.e., a distance measure such as $l_2(I, I')$ would be quite large – in fact in many cases it would be larger than changing the identity of the object. The ANN has to learn these invariances, i.e. map the very different input images I and I' to the same label. Pooling helps create these invariances because the output of the pooling unit is not sensitive to the spatial location of the information within the receptive field.

Since a pooling layer has no parameters, there is nothing to be learned. The pooling layer does have to keep track of from which input units the output came from in the forward pass and route the error to those input units during the backward pass.

- **max pooling**: only the winning unit contributes to the output, so the error is passed onto the winning unit, the other inputs units receive zero error.
- **average pooling**: the error is divided by the size of the receptive field and this error is routed to all input units alike

10.1.4 Putting it all together

A deep neural network generally consists of multiple instances of convolutional layers and pooling layers (Fig. 5). Because of how the activations are defined in the forward pass and how the errors are computed in the backward pass,

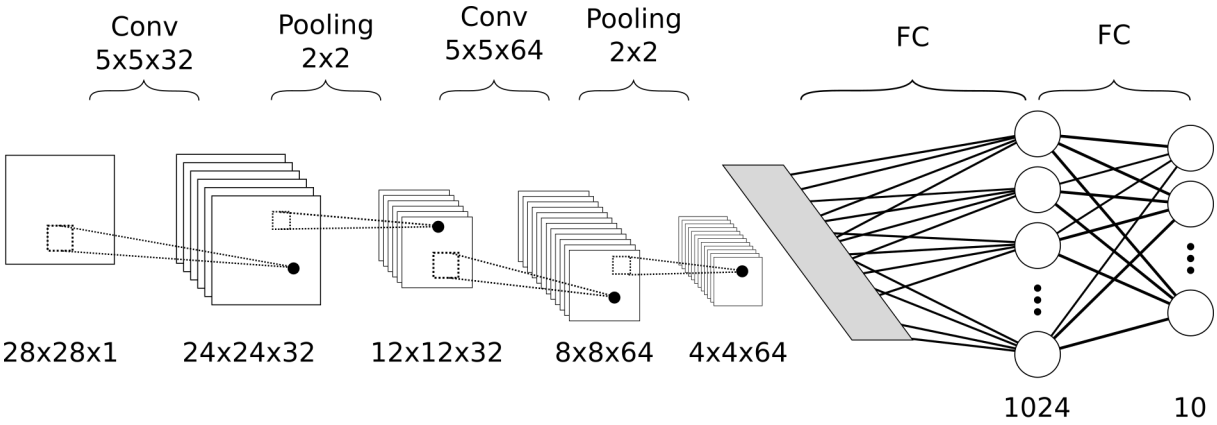


Figure 5: Example of convolutional neural network (CNN) with seven layers.

each layer l in a deep network interacts only with the preceding layer $l-1$ and the following layer $l+1$. That's why ANN are modular and any type of layer can be combined with any other type of layer, as long as the fan-in and fan-out factors match.

For image recognition, the architecture of the deep network is inspired by the architecture of the visual systems in the brain. The lower layers consist of convolutional and pooling layers that generate feature maps that are local, compressed and abstract representations of the input image. The highest layers consist of fully connected layers that integrate all feature information to generate the classification (Fig. 5).

10.2 Recent technological breakthroughs

CNN have become hugely popular only in 2012. However, the ideas behind CNN have been known since the 1990's (Lecun et al., 1998). What happened in that time? Earlier CNN were used, but not very successful in solving image recognition problems. Their performance was far from human performance (Fig. 6). This changed because of three technical advances. First, CNN have to be trained on massive amounts of data. These were generated by Fei-Fei Li

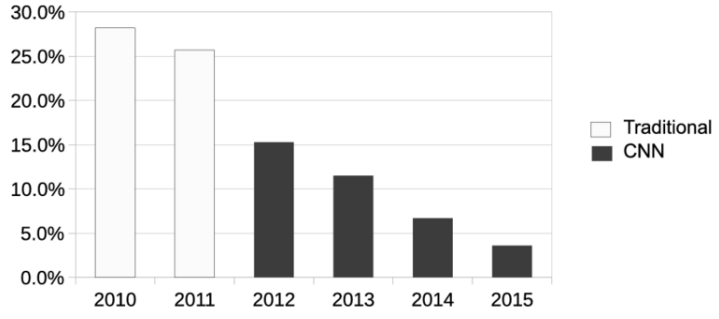


Figure 6: Performance on ImageNet competition over the years (Bottou et al., 2018, Fig. 2.4). A human achieves an error rate of roughly 5%.

and colleagues (Deng et al., 2009) in their ImageNet database. Using Amazon Mechanical Turk they had more than 14 million images hand-annotated by people to indicate what objects are pictured in more than 20,000 categories. The ImageNet competition started 2010. The second technical advance were graphical processing units (GPUs), which made it possible to train deep neural networks on massive amounts of data. In 2012, a deep convolutional

neural net called AlexNet (Krizhevsky et al., 2012) trained using GPUs achieved an error rate of 16%, which was a huge improvement over the 2nd best-performing algorithm. That achievement sparked huge interest in CNNs. Nevertheless, AlexNet was not the first deep CNN trained using GPUs. That distinction goes to DanNet (Ciresan et al., 2011), which AlexNet is very similar to.

The wide-spread popularity of deep learning would not be possible without the third technological advance: the availability of software frameworks that automate the otherwise tedious implementation, training and testing of deep networks. Several frameworks have been developed over the years and include: Caffe, CNTK, TensorFlow, Theano, and Torch. To make the usage even simpler, Keras provides another abstraction layer on top of CNTK, TensorFlow, or Theano.

10.3 Regularization

Large neural nets and small datasets usually lead to overfitting. The model learns the statistical noise in the training data and cannot generalize to new data. Earlier in this class regularization was introduced as a method to avoid overfitting. A regularization term was introduced into the loss function to penalize large values of the model parameters.

$$L(\theta, X, Y) = L_0(\theta, X, Y) + \lambda R(\theta_i) \tag{7}$$

This is also known as *shrinkage*. Some regularization techniques discussed below are quite different from shrinkage. Why are these techniques then considered forms of regularization? Because they impose features onto the model that we know that the model should posses: in the case of shrinkage we "know" that the parameter values should be fairly small. In a similar vein, we can introduce other types of knowledge to constrain the model. Goodfellow et al. (2016) therefore propose:

> Regularization is any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error.

In other words, regularization reduces model complexity.

How to detect that your network is overfitting? Traditional methods for model selection are difficult to use with deep networks because they have so many parameters and take a long time to train. Instead, to catch overfitting in a deep network, look at the accuracy of the model on the training data (training accuracy) and the accuracy on the test data (test accuracy) over the training steps Fig. 7. The accuracy on the training set will always increase, when you use gradient descent. For stochastic gradient descent that will only be true on average. If the network starts to overfit the data, the test accuracy will diverge from the training accuracy and might even start to drop, if overfittig is severe.

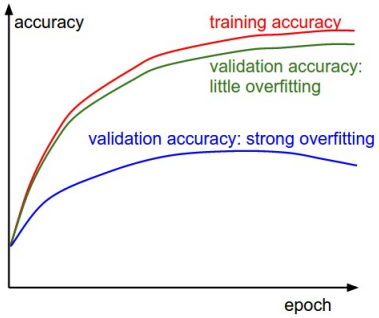


Figure 7: Detecting overfitting in your dataset.

10.3.1 Early stopping

A very simple strategy to avoid overfitting the data, is to stop training process when the test accuracy starts to drop. One way to implement this is to compute test accuracy every T training steps. If the current test accuracy is lower than the previous one, then return the previous snapshot of the network and stop the optimization. If not, store the current test accuracy and snapshot of the network and repeat.

10.3.2 Weight decay

Weight decay sounds like a new regularization method, but it is mathematically equivalent to gradient descent on an l_2 regularization term. The term describes the update equation that is used: after each update, the weights are multiplied by a factor $\beta < 1$, which means that the weights "decay" with time and, hence, prevents the weights from getting too large.

$$w \leftarrow \beta w \tag{8}$$

Assuming that the original loss function that is being minimized is L_0, one can add an l_2 regularization term to form a new loss function:

$$L = L_0 + \frac{1}{2}\lambda \sum_i \theta_i^2 \tag{9}$$

This new loss function will penalize large values of the parameters and therefore limit overfitting (see earlier lecture). The regularization parameter λ determines how the original loss function L_0 is traded off with the regularization term. The gradient of the new loss function is:

$$\frac{\partial L}{\partial \theta_i} = \frac{\partial L_0}{\partial \theta_i} + \lambda \theta_i \tag{10}$$

When using this gradient in gradient descent, the second term means that the weights should decrease proportionally to its value, which is equivalent to multiplying it by a factor $\beta < 1$.

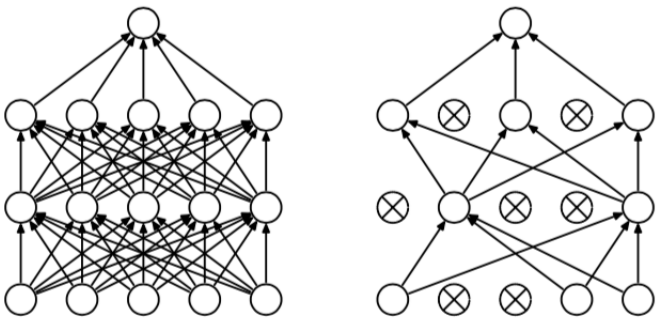
l_2 regularization, or weight decay, effectively reduces the capacity of the network. So deeper and wider networks might have to be used to compensate for the reduced capacity.

10.3.3 Dropout

Overfitting occurs because all items in the training set are burned into the network, so that the network doesn't generalize well to new data. One problem is complex co-adaptation in the network, i.e., units depend on fine differences between the activity of other units to generate their outputs. Complex *co-adaptation* are sensitive to small changes in the input and therefore do not generalize well. The second problem is that all units in the network are trained on the training data. So all units adapt to the idiosyncrasies in the training data (Hinton et al., 2012; Srivastava et al., 2014).

A solution to the first problem is to add stochasticity, a solution to the second problem is to train different networks on different subsets of the training data and then to average the predictions across the ensemble of networks. Even though every network might overfit the dataset it was trained on, each network will learn a different input-output relationship and therefore generate different predictions on the test data. A problem with this approach is that many networks have to be trained and stored. If the networks are large, this required a long time.

Dropout is a regularization method that adds stochasticity and approximates the training of a large number of different neural networks in parallel. During training, each unit is randomly ignored, or dropped, with a probability p. Typical values for the dropout rate are 0.5 and 0.2.



(a) Standard Neural Net (b) After applying dropout.

Figure 8: Illustration of dropout in a neural network (Srivastava et al., 2014, Fig. 1).

Dropout introduces noise into the training process and forces nodes to probabilistically take on more or less responsibility for the inputs, hence averting complex co-adaptation. Each iteration of dropout creates a different

network with different numbers of units in the layers and different connectivity. Each update to the weights is applied only to the units that are not dropped. The weights are shared between the different configurations of the network.

The number of different effective networks that can be generated by dropout is computed as follows

units dropped out	**number of conf.**
1	n^l
2	$n^l(n^l-1)/2$
$\vdots$	$\vdots$
k	$\binom{n^l}{k}$

Since the number of configurations reaches a maximum at $k = n^l/2$, a dropout rates of $p = 0.5$ is the most common choice. However, for a given problem other rates may yield a better performance.

Dropout reduces the effective capacity of the model, because the effective size of the network is smaller than the number of available units. Therefore a deeper and wider architectures has to be used to compensate for the lower capacity.

10.3.4 Data augmentation

Visual object recognition should be *invariant* under several transformations of the image, because the identity of objects does not change, if the object is translated, rotated, scaled, stretched (to a certain extend), discolored (to a certain extend), damaged (to a certain extend), or occluded. In addition, the object identity is invariant to changes in our view of the object, i.e., whether we are near to or far from the object, how it's lighted, noise in our perception (e.g. smoke or fog) or changes of the background behind the object. Humans and other animals have learned these invariances and can recognize objects under all these transformations. However, the mathematical representation of a transformed image I' is very different from the original image I, i.e., a distance measure such as $l_2(I,I')$ would be quite large – in fact in many cases it would be larger than changing the identity of the object. The ANN has to learn these invariances, i.e. map the very different input images I and I' to the same label. In the early days of computer vision, it was tried to built models that explicitly took into account these invariances (see pooling above for an example).

In deep learning, these invariance are trained into the network by *data augmentation*. All images are presented as they are, but also transformed in all the ways that one would like while leaving object identity, i.e. the label, intact. The network then learned to associate very different input images I and I' with the same label. It has been shown that deep networks *generalize* the invariance to novel objects that were not shown before.

Another advantage of data augmentation is that it increases the size of the dataset tremendously, because each transformed image I' and its label form a new item that the network learns from.

10.4 Speeding up convergence

10.4.1 Transfer learning

When training a very large neural network, it is helpful to start with an existing network that was trained on a similar task to the one at hand, and to reuse the lower layers of that network in the new network. This so-called *transfer learning* works because the lower layers in ANN generally learn representations that are closer to the inputs, and these representations are generally more generic, i.e., they are useful for a number of different tasks. So, the lower layers can be transferred to another network to perform a similar tast. The closer to the output layer a layer is, the more specific its representation is for the task the network is performing. That's why the upper layers have to be trained specifically.

Transfer learning will speed up training and require less training data. Since the weights in the lower layer are already close to correct, there are fewer network parameters that still need to be trained. In addition, since backpropagation starts at the output layer, the training signal is always the most informative in the upper layers and the weights there converge the fastest. It takes longer for the weights in the lower layers to converge. In transfer learning, those weights are nearly correct already and so the more time-consuming part of training can be dramatically shortened.

10.4.2 Learning rate schedule

Training the network critically depends on the learning rate (see earlier lecture). If it is much too high, the optimization diverges Fig. 9. If it is somewhat high, optimization will converge very fast initially, but fail to reach the minimum. If it is too low, convergence will be very slow. A good itermediate learning rate achieves a medium convergence speed initially and a low asympototic loss. With a *learning rate schedule*, one can have the best of both worlds. A fast

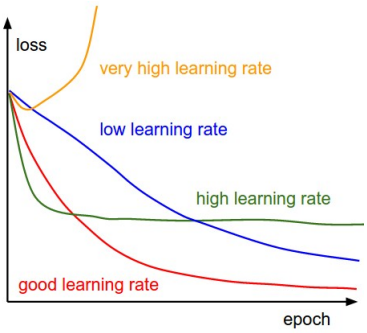


Figure 9: The effect of choosing different learning rates on network performance.

initial convergence with an high initial learning rate, and good asymptotic performance in later update steps. So, the learning rate changes during the course of training.

$$\eta_t = r(t,\xi) \tag{11}$$

There are several different choices for the learning rate schedule:

- predetermined piecewise constant learning rate: learning rate stays constant until certain predetermined epochs, in which the learning rate is reduced
- performance scheduling: every T steps measure the validation error and reduce the learning rate by a factor β when the error changes $< \varepsilon$.
- expotential scheduling: $\eta_t = \eta_0 d^{-t/r}$
- power scheduling: $\eta_t = \eta_0 (1 + t/r)^{-c}$

10.4.3 Momemtum optimization

Gradient descent can be very slow, especially when the some activation values are very small. In addition, each update step is independent of the previous ones. To speed up convergence, a metaphor from physics can be adopted for optimization. Imagine that you are describing a ball rolling down a hill. The parameter values θ correspond to the position of the ball. The loss function corresponds to the potential energy of the ball. Then, gradient descent equates the velocity of the ball to the gradient of the loss function, i.e., the gradient describes how the position of the ball changes.

$$\theta \leftarrow \theta - \eta \nabla L(\theta) \tag{12}$$

If instead we assume that the gradient describes the acceleration of the ball, then the gradient describes how the velocity changes:

$$v \leftarrow \mu v - \eta \nabla L(\theta) \tag{13a}$$

and the velocity then describes how the position, i.e.,

$$\theta \leftarrow \theta + v \tag{13b}$$

This method is called *momentum optimization*. The parameter μ is often called the *momentum* because it describes the tendency of the optimization to continue in the same direction as previously. For $\mu = 0$, there is no momentum

and the optimization forgets the previous direction immediately. That is the normal gradient descent. For $\mu = 1$, the optimization never forgets any previous step. The only way to change the direction, is when the gradient changes direction. This is a bad idea since around a local minimum of the loss function, the gradient vanishes, so the velocity will only change once the optimization has overshot. This leads to oscillations. A good trade-off between these two extremes is $\mu = 0.9$, but the optimal value is different in every application.

A small modification improves the performance of momentum optimization, because in Eq. 13a the gradient is evaluated at θ, even though the gradient is added to the parameters at $\theta + \mu v$. Nesterov Accelerated Gradient (NAG)

$$v \leftarrow \mu v - \eta \nabla L(\theta + \mu v) \tag{14a}$$

$$\theta \leftarrow \theta + v \tag{14b}$$

There are many other optimization methods that exploit certain properties of the gradient around the local minimum to speed up convergence. We cannot mention then all here. One that is used and mentioned quite frequently is *Adam*, which stands for *adaptive moment estimation*. It combines two different techniques and works very well for many problems. However, in some cases it might lead to solutions that generalize poorly.

In addition, some optimization methods make use of the second derivative, i.e., the Hessian. However, those are expensive and for a network with millions of parameters, the Hessians cannot even be stored in computer memory.

References

Bottou, L., Curtis, F. E., & Nocedal, J. (2018). Optimization Methods for Large-Scale Machine Learning. *SIAM Rev.*, *60*(2), 223–311.

Ciresan, D. C., Meier, U., Masci, J., Gambardella, L. M., & Schmidhuber, J. (2011). Flexible, High Performance Convolutional Neural Networks for Image Classification. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence*, volume 2, (pp. 1237–1242)., Barcelona, Spain.

Deng, J., Dong, W., Socher, R., Li, L.-j., Li, K., & Fei-fei, L. (2009). Imagenet: A large-scale hierarchical image database. In *In CVPR*.

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. The MIT Press.

Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. R. (2012). Improving neural networks by preventing co-adaptation of feature detectors. *ArXiv12070580 Cs*.

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*.

Lecun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proc. IEEE*, *86*(11), 2278–2324.

Siegel, C., Daily, J., & Vishnu, A. (2016). Adaptive Neuron Apoptosis for Accelerating Deep Learning on Large Scale Systems.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *J. Mach. Learn. Res.*, *15*, 1929–1958.