Deep Learning (2IMM10)
Spatially distributed data (CNN)

Vlado Menkovski

Spring 2020

4 Models for spatially distributed data (CNN)

The learning outcomes of this chapter:

- Be able to develop models for data with local spatial correlations using convolutional neural networks

4.1 Spatially distributed data

We refer to data as spatially distributed when the data manifests with useful correlations of the features based on their position relative to the other features. For example if we take image data where the features are the pixel color and intensities. We can observe patterns such as strokes and edges based on locally correlated groups of pixels (Figure 1).

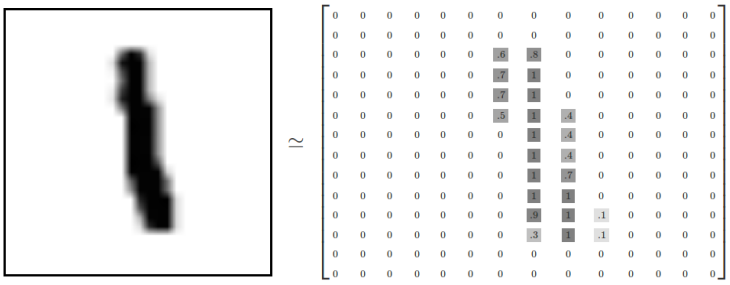


Figure 1: MNIST image with an array representation

For such data it is important to preserve this information and hence we structure it as an image. In contrast you can think about a table as a data structure, where each column is a feature. In a table we typically to not expect that information will be lost if the column are shuffled, but in an image we do.

Images can have different dimesionality. Photographs are typically two-dimensional, however there are images of one dimension, such as speech (Figure 7) as well as higher dimensional image (such as Magnetic Resonance Image or Computer Tomography medical scans).

Note: a color 2D image is typically organized as a 3D array where the third axis holds the encoding for the different colors of the pixels. In an RGB encoded image, we would have one 2D image for the red color of the pixels one for the green color and one for the blue color. Even though the image is represented in a 3D array we still refer to it as a 2D image because the spatial information is distributed in 2 dimensions (i.e. we can re-arrange the color planes without losing information).

Spatially distributed data often also has correlations that are localized in small regions of the image. For example to detect a bicycle in an image it would be useful to have a detector for a wheel. This detector could then be used to detect both of the wheels and combined with a detector for the frame could be combined in a detection mechanisms for bicycles. Note, however, that a wheel can take a small part of an image and can appear in different locations. Neural networks are particularly well suited for building such hierarchical representations, however the MLP architectures discussed so far are not efficient at detecting local patterns in images. In this chapter we study the convolutional neuron the the convolutional neural network models that use this mechanisms to learn to detect local patterns more efficiently.

4.2 Convolutional neural networks

Convolutional neuron

Let us consider the following task. We develop a model that detects the sequence '1011' in a 1D image (Figure 2). We can formulate this as binary classification. With a single artificial neuron (Figure 3) we could develop a model to detect the pattern. However, since this pattern is shorter than the length of the image it could in principle appear in different locations in the image. The problem with our single neuron is that it looks at the whole image. We also refer to this type of neurons as 'fully connected'. To detect the pattern at different locations we would need more neurons, each focusing on a different location. Then would need a second layer with a neuron combining the output of the neurons or the first layer. The second layer neuron would basically need to implement the binary 'OR' operation to test whether any of first layer neurons detecting the pattern.

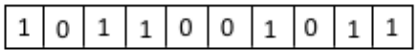


Figure 2: An input sequence in which we want to detect and locate the sub-sequence 1011.

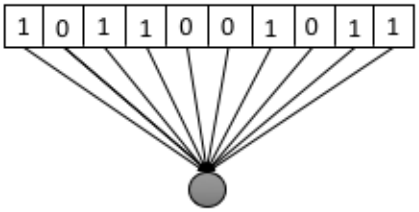


Figure 3: The way a regular artificial neuron sees its input.

This model would work as expected, but it will also have a number of parameters that is higher than what we could have optimally. It would be much more efficient to have a neuron that does not look at the whole image but has a narrower 'field of view', ideally as wide as the pattern.

In fig. 4 such a neuron is depicted. This neuron sees only a part of the image. For a model as this to process the whole image we would need to slide it across the image. As such we can re-use it on different locations. Such as neuron is referred to as a convolutional neuron. The output would then not be a single activation, but rather an activation at each location. This 'activation map' would then need to be processed by the second layer that would make the final decision. In our case we can acutally use the same second layer neuron as before. A single neuron that implements the binary OR operation on the output of the convolutional neuron.

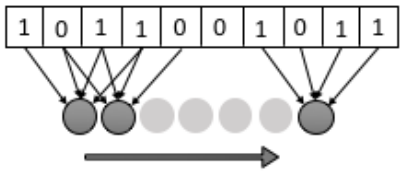


Figure 4: A schematic overview of our neurons field of view sliding along the input sequence.

The important aspect is that the convolutional neuron re-uses the paramters at each location and as a consequence the new model will have significantly less parameters than the fully connected model.

As with the original artificial neuron, the convolutional neuron has a weight for every input in its

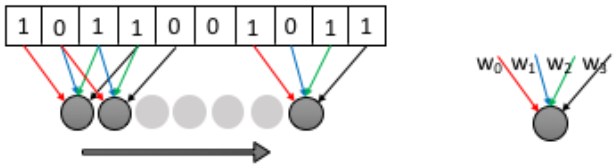


Figure 5: An illustration of the neuron operation described in eq. (2).

field of view. These weights together are called the *kernel* of the neuron, and the size of its field of view referred to as its kernel-size. The operation the neuron implements can be described in terms of the mathematical operation of "convolution". Convolution of two sequences is typically[1]defined as follows:

$$\begin{aligned}(a * b)_n &= \sum_{i=-\infty}^{\infty} a_i b_{n-i} \\ &= \sum_{i=-\infty}^{\infty} a_{n-i} b_i\end{aligned} \tag{1}$$

where finite sequences are padded with zeros, and convolution of higher dimensional arrays is defined analogously. A visual representation of this can be seen in Figure 5. Note that the second equality means that convolution is a commutative operation. If the neuron from our example has kernel $k = (w_3, w_2, w_1, w_0)$, and has bias b and activation function ϕ, then if we apply it to an input sequence $x = (x_0, \ldots, x_N)$, the output is

$$o_\theta(x) = \phi(x * k + \mathbf{b}), \tag{2}$$

where $\mathbf{b}$ is the vector with b at every index. Note that in order for the weights to be applied as in Figure 5 we have arranged them in reversed order to form the kernel.

The main goal of using convolutions is to re-use parameters of the model which in turn makes the model significantly more efficient both for training and inference. As depicted in Figure 5 the parameters w_0, w_1, w_2, and w_3 on the corresponding colored edges (red, blue, green and black) are re-used at each position where the neuron is placed during the convolution operation. In contrast, if we used a different neuron for each location, or one neuron taking the whole image as input, the number of parameters needed to achieve the same detection would be significantly larger as the parameters cannot be re-used.

This has a very significant impact on the success of the training such models. Not only because the model is less efficient, and we need to train more parameters, but we would also need to have examples of the pattern appearing in all locations in our training dataset. This is a key challenge in many settings, as we cannot expect to have training data available that covers the entire natural distribution of the data. Therefore, one of the main challenges of Deep Learning is to develop methods that are efficient in generalizing from a small number of examples.

[1]In physics, mathematics, and engineering, multiplication is typically denoted by $\cdot$ (`\cdot` in LaTeX) and $*$ is used for convolution. In some computer-science literature one can also find $\star$ being used to denote convolution. Outside of computer-science this $\star$ operation is usually used to denote the related cross-correlation operation, which is essentially convolution with the order of weights in the kernel swapped. Fun fact: TensorFlow actually performs cross-correlation instead of convolution.

Figure 6: A selection of images form the MNIST dataset. Ask yourself the following questions: What kind of patterns might be present in various locations in an image from the MNIST set? What kind of patterns could help you classify these images?

Figure 7: A sound fragment of someone speaking. Besides phonemes, what might be some local structures that could help a network identify what is being said?

Padding

There is one problem with our description in eq. (2) of what the neuron does. We said that for the theoretical definition of convolution to work, we simply pad our sequences with zeros. This however begs the question how long our output sequence should be. Clearly we cannot have an infinitely long output sequence. Far away from the actual (non-zero) input sequence the output is zero, so these outputs can be ignored and obviously these zeros are not computed — see Figure 8 for a visual representation of this. On the interior of the original input sequence, i.e. where the neurons field of view lies entirely within the original input sequence, we are certainly interested in the output, so there the output clearly should be computed — see Figure 9.

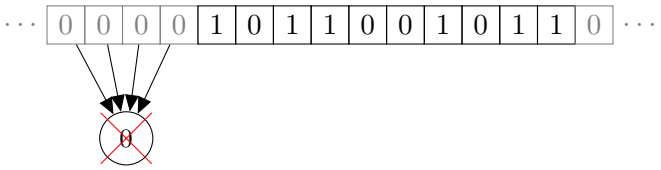


Figure 8: Theoretical zeros in (1) far away from the input are ignored and outputs there are not computed. The padded zeros are drawn in gray.

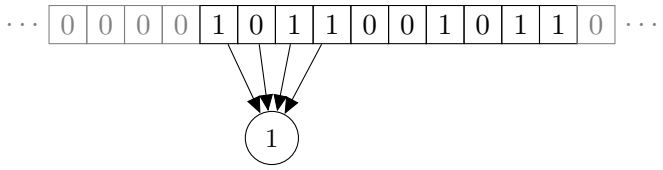


Figure 9: On the interior of the original input sequence outputs clearly need to be calculated. The padded zeros are drawn in gray.

However, the question remains what we should do at the edge of our input sequence — see Figure 10. This depends on the task at hand. In our current example it doesn't make much sense to compute the corresponding outputs since we are looking for locations of the entire pattern. On the other hand, suppose we are training our neuron[2]to recognize bicycles in pictures. In that case we would very well be interested in bicycles that are only partly in the picture, and we would want to pad our picture so that the output of our convolutional neuron has the same size as the original picture.

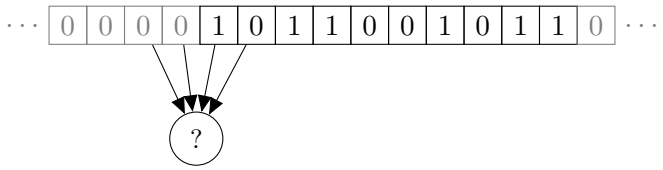


Figure 10: What should happen at the edge of the input sequence?

For one-dimensional convolution, there are three common ways of padding:

Valid padding: No padding happens at all, i.e. all potential outputs for which the field of view does not lie entirely within the input sequence, are discarded.

Same padding: The input sequence is padded in such a way that the output sequence has the same size as the original input sequence.

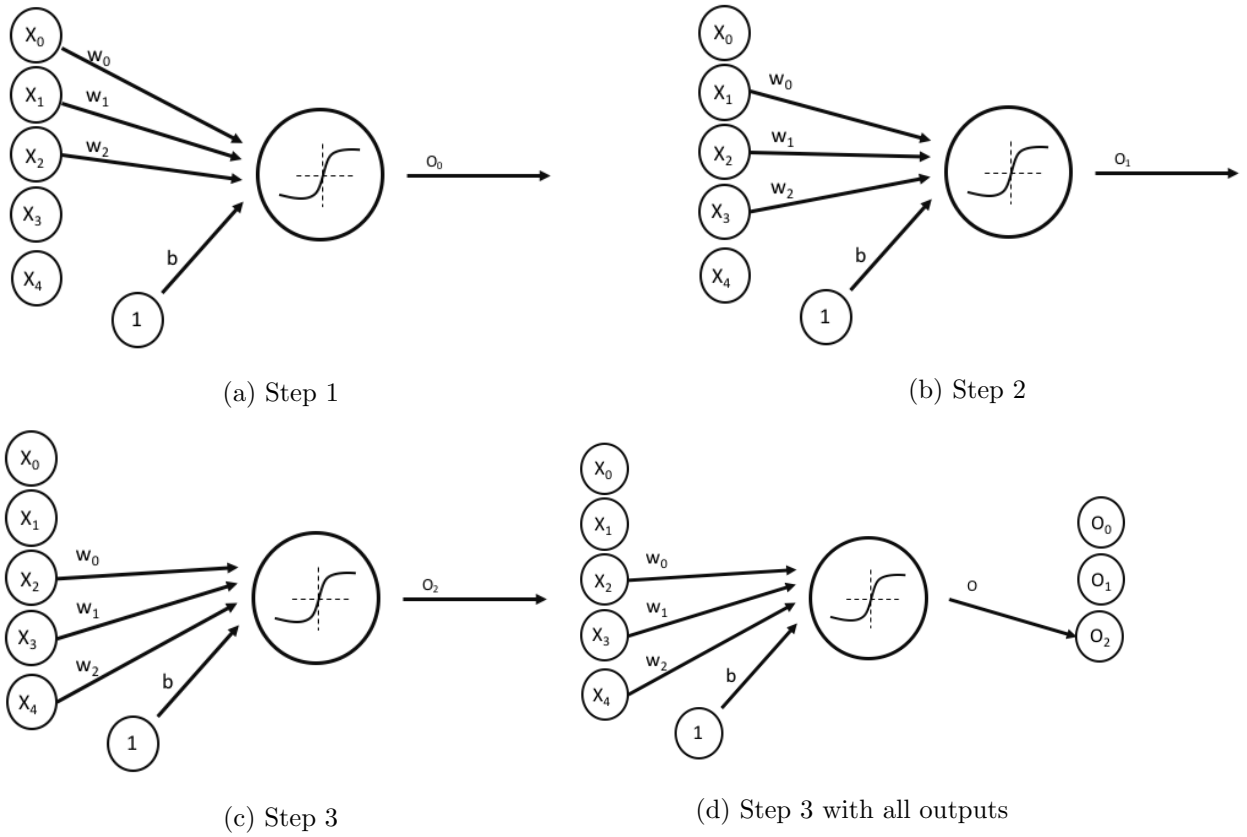
Causal padding: Padding is applied in such a way that the output at index i does not depend on input variables at any later index.

For higher-dimensional convolution, valid padding and same padding are the two common ways of padding.

Backpropagation

To be able to use a new component in a neural network model we need to make sure that it will not prevent us from training our model. To do that the new component needs to allow us to compute the gradient of its output with respect to the inputs. Specifically for the convolutional neuron in addtion to computing the gradient of the output with respect to its input we also need to compute the gradient of the its output with respect to its parameters such that we can update these parameters during training.

[2]Of course a single neuron is a bit too simple for this task, but we just use this to make a clear point.



(a) Step 1

(b) Step 2

(c) Step 3

(d) Step 3 with all outputs

Figure 11: Forward pass of convolutional neuron

As before we take a forward step with the neuron and a backward step to compute the output values. In this case the difference is that both the forward and the backward steps involve a number of convolutional steps.

In Figure 11 we can observe the forward steps. The neuron processes an input with a a size of 5 $(x_0, ..x_4)$, with a kernel of length 3 (w_0, w_1, w_2). In the first step at the initial position the neuron computes the ouput o_0 Figure 11a. It the second and third steps outputs O_1 Figure 11b and o_2 Figure 11c are computed.

We can represent these outputs as a activation map vector Figure 11d

We assume that the component under investigation is part of a larger model and as such during the backwards pass it receives gradients from the end of the model back to its output (Figure 12).

Let us consider the compute graph of the convolutional neuron at each step of the convolution to compute the backward pass values Figure 13. In this case it is more effective to look at separate compute graphs for each step in the convolution Figure 13a and Figure 13b, and Figure 13c. For each of these steps we can compute a gradient update to a selected parameter (in the figure for w_1) or for an input compute coming from a previous layer (x_1 in Figure 13d and Figure 13e).

The same operations can be expresses in a condensed way using vector representations Figures 14a

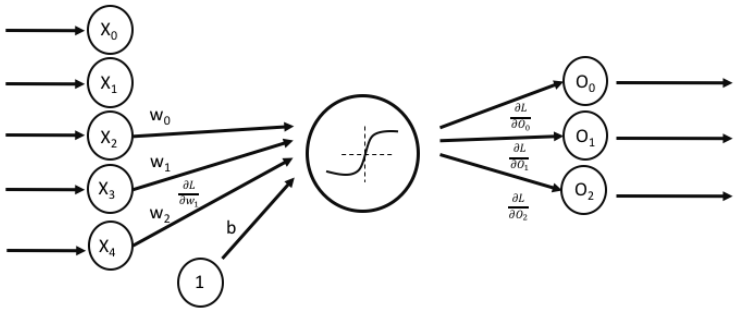


Figure 12: Convolutional neuron receives gradient updates from subsequent layers for each output

and 14b. We can notice in the figures that computing the backward pass can be achieved also by a convolution. In this case we convolve the transposed kernel with the gradients coming from the end of the model.

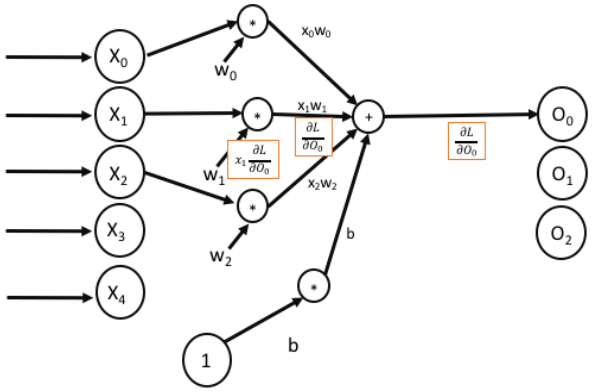
To express these operations more formally, the details of how this works exactly depend a on the type of padding we perform. However, for a theoretical analysis, the type of padding comes down to how much you shift the indices of the output sequence, and at what points you cut the output sequence off. Therefore, if our neuron has kernel

$$\begin{aligned}\mathbf{k} &= (k_0, \ldots, k_K) \\ &= (w_K, \ldots, w_0),\end{aligned}$$

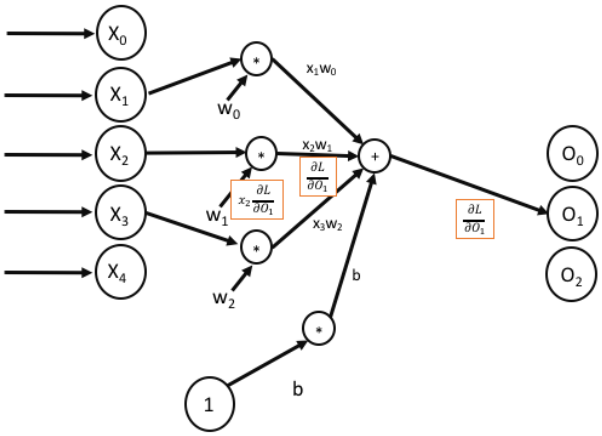
and bias b, and is applied to some input $\mathbf{x} = (x_0, \ldots, x_N)$, we can without loss of generality say that the output of our neuron is given by

$$\begin{aligned} o_i &= \phi(y_i + b) && \text{if an output is required, otherwise 0,} \\ y_i &= \sum_{j=-\infty}^{\infty} x_{i-j} k_j \\ &= \sum_{j=-\infty}^{\infty} x_j k_{i-j}. \end{aligned}$$

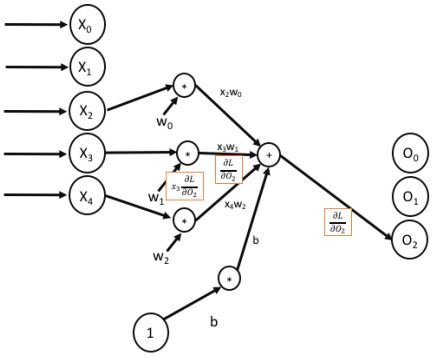
Now if we have a loss function L, which is a function of $\mathbf{o}$, then we can calculate the gradient of L with respect to the parameters of the neuron, and with respect to the input to our neuron as follows:



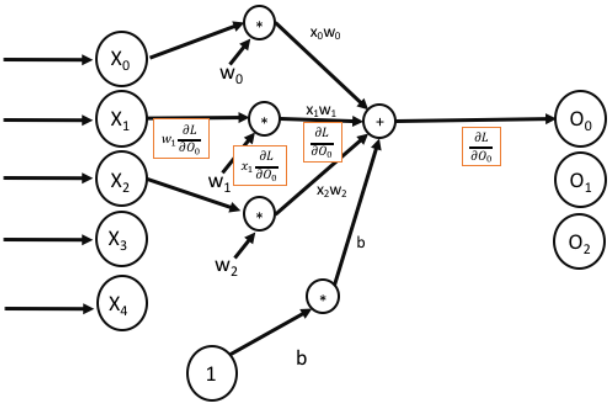
(a) Backpropagating the update value for w_1 at step 1



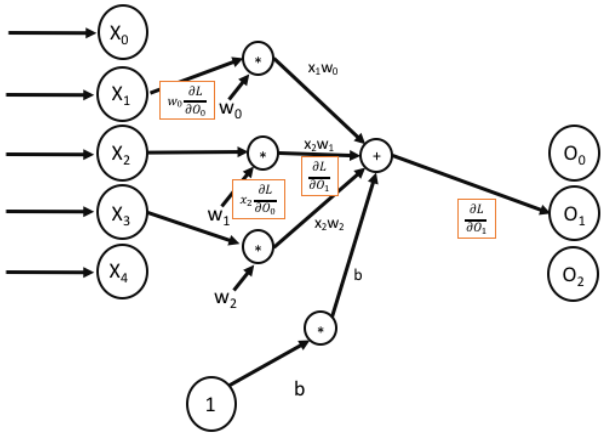
(b) Backpropagating the update value for w_1 at step 2



(c) Backpropagating the update value for w_1 at step 3

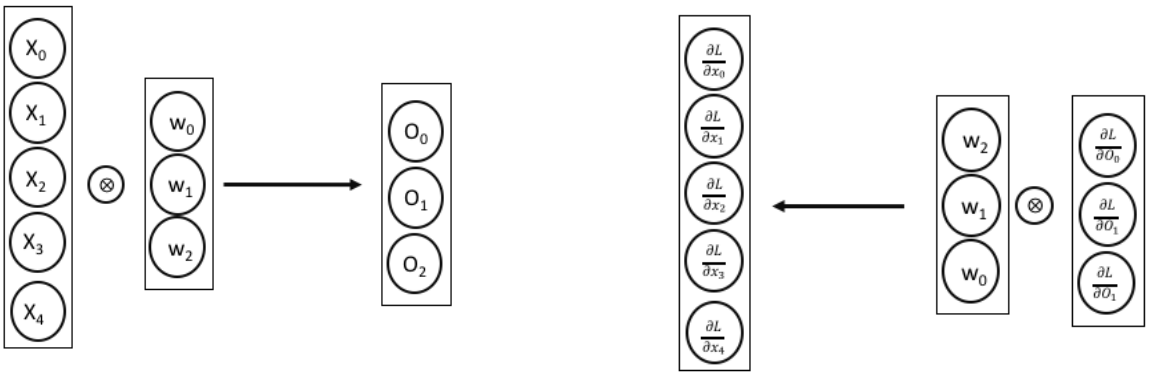


(d) Backpropagating the update value for x_1 at step 1



(e) Backpropagating the update value for x_1 at step 2

Figure 13: Computing the backpropagation of the convolutional neuron for the parameter w_1 and the signal x_1 using its computation graphs



(a) Forward pass

(b) Backward pass (notice the transposed kernel)

Figure 14: Backpropagation convolutional neuron (vector form)

$$
\begin{aligned}
\frac{\partial L}{\partial b} &= \frac{\partial L}{\partial \mathbf{o}} \frac{\partial \mathbf{o}}{\partial b} = \frac{\partial L}{\partial \mathbf{o}} \phi'(\mathbf{y} + \mathbf{b}) \\
\frac{\partial L}{\partial \mathbf{y}} &= \frac{\partial L}{\partial \mathbf{o}} \frac{\partial \mathbf{o}}{\partial \mathbf{y}} = \frac{\partial L}{\partial \mathbf{o}} \phi'(\mathbf{y} + \mathbf{b}) \\
\frac{\partial y_i}{\partial k_j} &= x_{i-j} \\
\frac{\partial L}{\partial k_j} &= \sum_i \frac{\partial L}{\partial y_i} \frac{\partial y_i}{\partial k_j} \qquad (3) \\
\frac{\partial y_i}{\partial x_l} &= k_{i-l} \\
\frac{\partial L}{\partial x_l} &= \sum_i \frac{\partial L}{\partial y_i} \frac{\partial y_i}{\partial x_l}. \qquad (4)
\end{aligned}
$$

where ϕ' is applied element-wise. Here equations (3) and (4) can be summarized as

$$
\begin{aligned}
\frac{\partial L}{\partial k_j} &= \sum_i \frac{\partial L}{\partial y_i} x_{i-j} \\
\frac{\partial L}{\partial x_l} &= \sum_i \frac{\partial L}{\partial y_i} k_{i-l}
\end{aligned}
$$

which is essentially just convolution with the order of the right sequence ($\mathbf{x}$ or $\mathbf{k}$) reversed[3] (and some shifting of indices).

Question: the indexing in all of this can be a bit confusing. Can you work out the formulas if $\mathbf{x} = (x_0, x_1, x_2)$, $\mathbf{k} = (k_0, k_1) = (w_1, w_0)$, *padding is of the "same" type,* $b = 0$, ϕ *is the identity, and the output is indexed as* $\mathbf{o} = (o_0, o_1, o_2)$*?*

[3]This is called cross-correlation.

Channels

So far we have looked at the case where the input is an array of numbers over which we want to do convolution. As discussed earlier we may have an image with different components that exist in the image in parallel such as different colors is an image. We do not want to do convolution over that dimension as the data is not spatially distributed in that dimension. We refer to the different values in that axis as channels. To be able to use a convolutional neuron in such an image we need the kernel to also have this additional axis such that we can learn unique parameters for each of the different channels.

In practice the tensors that represent the color images are organized according to two main conventions for channels, the most common being "channels last", and the other one being "channels first". This means the input to a convolutional neuron has the following shape:

1-D convolution: $L \times C$ for channels last, or $C \times L$ for channels first;

2-D convolution: $L \times M \times C$ for channels last or $C \times L \times M$ for channels first;

3-D convolution: $L \times M \times N \times C$ for channels last or $C \times L \times M \times N$ for channels first.

If we take the channels last format, we can view all of this as a 1, 2, or 3-dimensional array of vectors, and our kernel should be such an array too. The product in eq. (1) then becomes an inner product, so in the case of 1-dimensional convolution this becomes

$$\begin{aligned}(\mathbf{a} * \mathbf{b})_n &= \sum_{i=-\infty}^{\infty} \mathbf{a}_i^\top \mathbf{b}_{n-i} \\ &= \sum_{i=-\infty}^{\infty} \sum_{j=0}^{C-1} a_{i,j} b_{n-i,j}.\end{aligned}$$

The analysis of backpropagation for a convolutional neuron holds with the necessary changes.

Convolutional layers

In the same way as we benefited from using multiple neurons and forming layers in the MLP model we can create layers of convolutional neurons to create model with high capacity. Such layers of convolutional neurons are refrred to as convolutional layers.

In this context, a convolutional neuron is typically refered to as a "filter". A convolutional layer has multiple filters with the same activation function, the same kernel-size, and the same kind of padding. The outputs of the filters are stacked together so that if a 2-dimensional filter transforms its input to an $N \times M$ tensor, a layer with K filters will transform the input to an $N \times M \times K$ tensor.[4]The output of

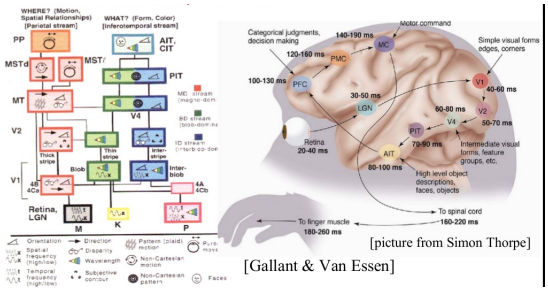


Figure 15: Processing visual information in a sequence of detections of low level concepts to high level concepts is also observed in biological systems as the human visual system.

one convolutional layer can be used as the input for another, allowing us to stack these layers. For a graphical representation of how convolutional neurons are grouped together into layers which can be stacked, see Figure 16.

These models that contain layers of convolutional neurons are referred to as convolutional neural networks (CNN). Typically CNN models have a number of hidden convolutional layers. Each layer detects localized patterns that are combined into more complex features in the subsequent layers. Since we know how to compute the gradient of a convolutional neuron, we can use back propagation and gradient-based optimization algorithms to train these models.

CNN Models

The architecture of the of a CNN as an extension of the MLP model includes a number of convolutional layers typically as hidden layers. These layers detect spatially localized features in the images. The sequence of convolutional layers combines these patterns in increasingly complex patterns, or higher level features. For many tasks these geographically distributed patterns need to be combined and mapped to a target variable. Suppose for example we would like to assign a label to an image. To achieve this we *flatten* the output of our last convolutional layer and feed that to a (sequence of) *dense* or *fully connected* layers. In other words the convolutional layers are followed by an MLP model — see fig. 17a for an example. For the output layer of the CNN the same principles hold as for the MLP. Both convolutional layers and dense layers have an activation function. These activation functions can also be specified and implemented as a separate layer, see fig. 17b for an example of this.

Sub-sampling When a CNN processes a large image, the activation map at each layer is proportionally large. This can place a significant computational complexity burden both on training and inference — especially when each convolutional layer has more filters than the previous. These activation maps give a precise location of where a feature has been detected. In many cases, however, we do not need to know the precise location, but rather only if the feature has been located or not. Therefore, we can benefit from decreasing the size of the activation maps, which will result in improved the efficiency of our models. There are two ways of doing this: the first is using *strided convolutions*[5], the second is using *pooling layers*.

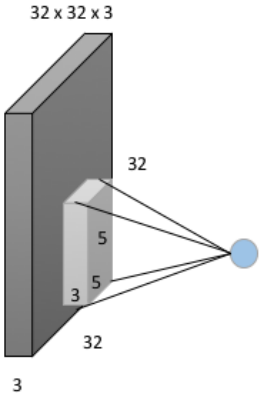
Sub-sampling with Strided convolution

Especially when the field of view of a neuron is large, two neighboring outputs will have largely overlapping inputs. One way of reducing our output size could therefore be to only pick every n^{th} output for some n. This is called strided convolution and n is the size of our strides. For higher dimension, we could also specify different stride sizes for the different dimensions. A graphical representation of strided 1-D convolution is given in Figure 18.

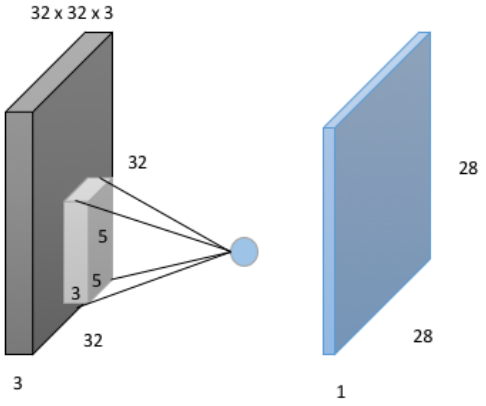
To see what the output size will be for a 2-D convolutional layer with padding and strides we can use the following overview:

[4]The order actually depends on choice of channels convention. If we use the channels first convention we get a $K \times N \times M$ tensor.

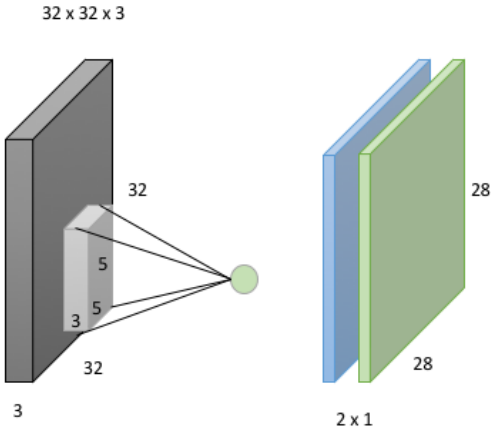
[5]Not to be confused with fractionally strided convolutions, which is a related but different concept also referred to as "transposed convolution".



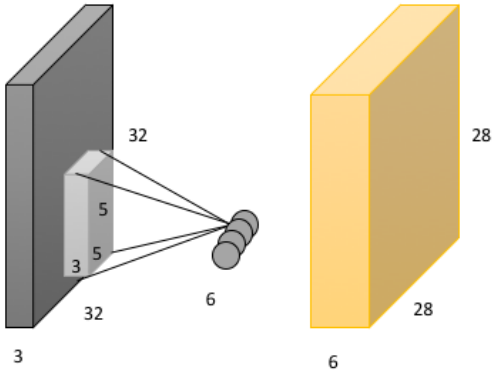
(a) We want to apply a neuron with a $5 \times 5(\times 3)$ kernel to a $32 \times 32 \times 3$ image.



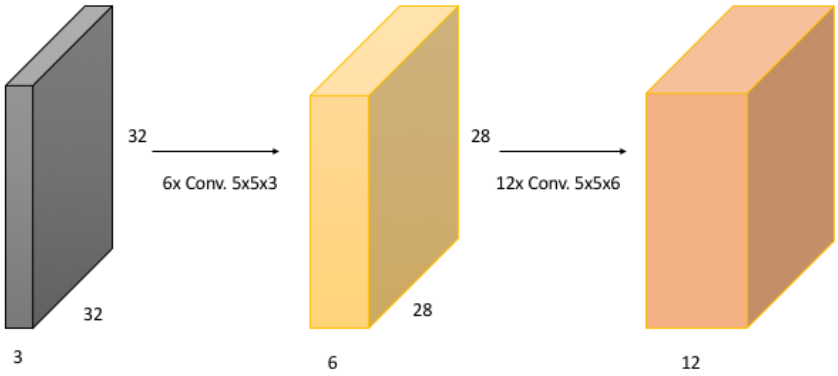
(b) Convolving the neuron over the entire image using "valid" padding, we get a $28 \times 28 \times 1$ output.



(c) We can convolve another neuron with the same dimensions over the image and stack the outputs to get a $28 \times 28 \times 2$ output.

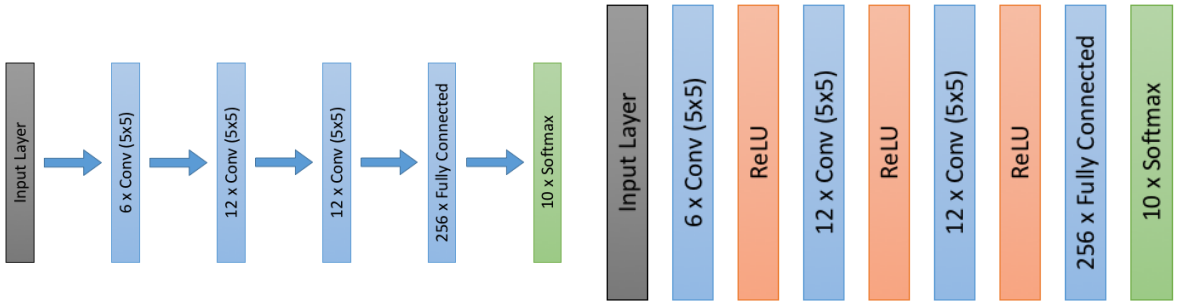


(d) We can stack more neurons this way to get a convolutional layer with 6 filters, giving a $28 \times 28 \times 6$ output.



(e) We can in turn stack such convolutional layers to get a deeper model.

Figure 16: A graphical representation of how we stack convolutional neurons to form convolutional layers.



(a) Depiction of an example CNN architecture

(b) Depiction of an example CNN architecture with activations explicitly mentioned

Figure 17: Two depictions of an example CNN architecture.

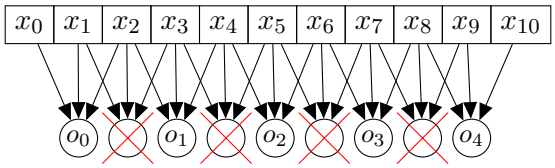


Figure 18: Strided convolution with a stride size of two,and a kernel of size three.

- Accepts:
  - $W_1 \times H_1 \times D_1$
- Outputs:
  - $W_2 = (W_1 - F + 2P)/S + 1$
  - $H_2 = (H_1 - F + 2P)/S + 1$
  - $D_2 = K$
- Where:
  - F is the filter size
  - P is the padding size
  - S is the stride
  - K is the layer depth (number of neurons)

Software packages implementing these convolutional layers can usually infer the output shape of a layer given its input shape, and you can access these details if you need to know what the shapes of your tensors at some point in your network are.

Sub-sampling with Pooling layers

The other way we can reduce the output size is by using a *pooling* layer. Pooling layers 'pool' the values of a local region in an image into a single value, such that the dimensionality of their output is reduced. There are different pooling layers such as 'Average Pooling', 'Maximum Pooling'

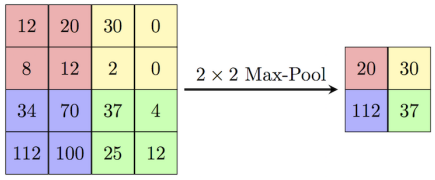


Figure 19: Subsampling - maxpooling

and 'Minimum Pooling'. The 'Maximum Pooling' or maxpooling layer is commonly used in image processing. This layer takes a specified windows size (e.g. 2 by 2) and produces as output for that location the maximum of the values in that window — see Figure 19.

Adding a pooling layer to the model requires that we can also backpropagate using that layer. Specifically we need to be able to compute the gradient of the output with respect to the inputs. For this we have the following formulas:

$$a(x) = \max_{i \in \{0,\ldots,m-1\}} x_i \qquad \text{has derivatives}^6$$

$$\frac{\partial a(x)}{\partial x_i} = \begin{cases} 1 & \text{if } x_i = \max(x) \\ 0 & \text{otherwise,} \end{cases}$$

and

$$a(x) = \frac{1}{m} \sum_{i=0}^{m-1} x_i \qquad \text{has derivatives}$$

$$\frac{\partial a(x)}{\partial x_i} = \frac{1}{m}.$$

Question: The derivative for an average pooling layer is smaller than one. Do you expect using average pooling in combination with convolutional layers to cause problems with vanishing gradients? Why or why not?

In Figure 20 we can see an example architecture of a CNN model with maxpool layers. In this example a maxpool layer is used after every convolution. For deeper and larger models it is also common to apply maxpool after every second or third convolutional layer instead.

Note that the components we have discussed so far are already sufficient to build a broadly applicable class of CNN models consisting of a bunch of convolutional layers with pooling layers in between, and a fully connected network on top of that.

4.3 Image Analysis with CNN

Digit classification with MNIST

Now that we have the basic building blocks of Convolutional Neural Networks, let us look at how we can use them to do image analysis. We first consider the classification task on the MNIST dataset.

[6] This is in the case of a unique maximum. If the maximum is attained at two or more distinct indices we only have one-sided derivatives for those indices.

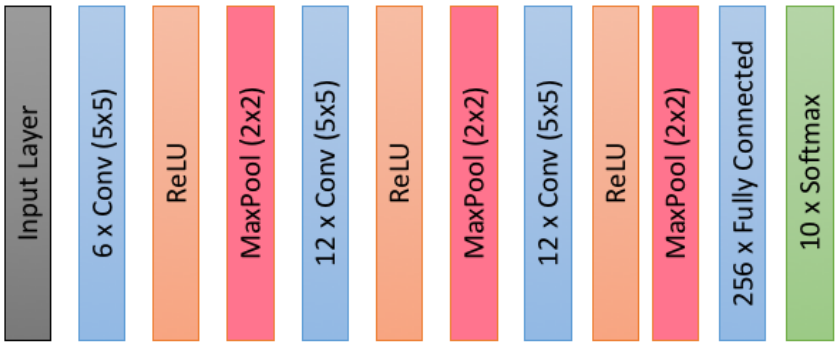


Figure 20: CNN - architecture with maxpool layers

Then we look at what kind of other image analysis tasks we might encounter, and finally we look at some existing image analysis models and how we can use those to solve specific problems.

In **??** we talked about how the sharing of weights allows the network to recognize learn a pattern in one place and recognize it everywhere.

In order to do classification we again make the labels into a one-hot encoding

$$y \mapsto e_y = (\delta_{y,j})_{j=0,\ldots,9}.$$

We use a Softmax layer to interpret the output of our network as a probability distribution:

$$\text{softmax} : (v_0, \ldots, v_9) \mapsto \left(\frac{e^{v_j}}{\sum_{i=0}^{9} e^{v_i}} \right)_{j=0,\ldots,9}.$$

and for our loss we use

$$\begin{aligned} L(y, \mathbf{p}) &= -\sum_{i=0}^{9} e_{y,i} \log(p_i) \\ &= -\log(p_y). \end{aligned}$$

we can view as either the negative log-likelihood of the correct label according to the probability distribution, $\mathbf{p}$, that our network gives as its output, or as the cross-entropy of the probability distribution that our network gives as its output relative to the correct (empirical) distribution represented by the one-hot encoding of the label.

Let us consider the architecture shown in Figure 21.

As an input we take $28 \times 28 \times 1$ tensors and we pass them to a two dimensional convolutional layer with 6 filters and a 5×5 kernel and valid padding. The output of this first convolutional layer then is a $24 \times 24 \times 6$ tensor. As activation we apply the ReLU function to every entry of that tensor. Next we send this tensor to the second convolutional layer with 12 filters, a 5×5 kernel, valid padding, and again ReLU activation. This gives us $20 \times 20 \times 12$ tensor on which we perform maxpooling with a 2×2 window to obtain a $10 \times 10 \times 12$ tensor. Next we send this tensor to the third and final convolutional layer which again has 12 filters, a 5×5 kernel, valid padding, and a ReLU activation which gives s a $6 \times 6 \times 12$ tensor to which we again apply maxpooling with a 2×2 pool to get a $3 \times 3 \times 12$ tensor. We flatten this into a 108-dimensional vector which we send to a fully connected layer with 256 neurons and ReLU activation. To obtain an output we send this 256-dimensional vector to a fully connected layer with 10 units whose output is fed to a softmax function in order to obtain a probability distribution.

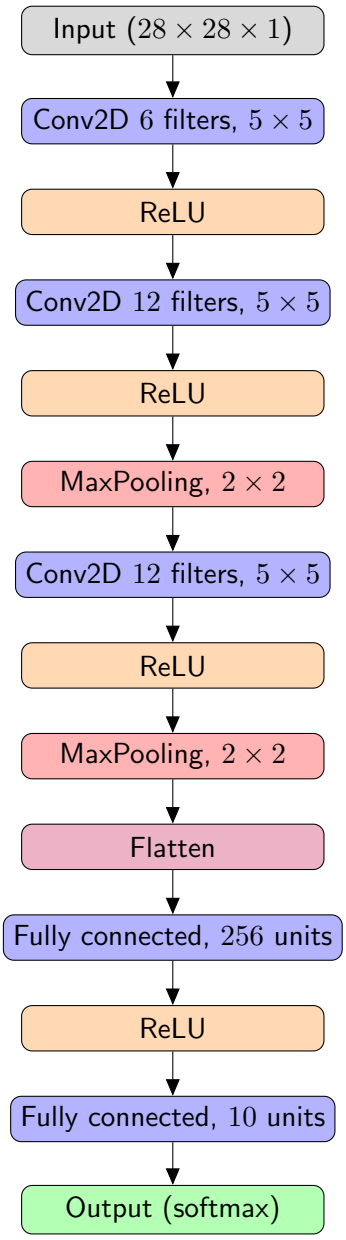


Figure 21: The architecture performing classification on the MNIST dataset.

Note: Convolutional layers give us a natural way to develop model that are invariant translation of the local patterns. You can imagine that our model can benefit from invariances to other transformations as well. We can also achieve this by modifying the data we train the model on. Before feeding the data to the network in our training loop we can randomly shift, rotate, flip, shear deform, and zoom in on the images. This essentially increases the variations that our training data covers from the input space. This approach is referred to as *data augmentation* and is particularly valuable in regimes with limited amount of data available. For this we do need to understand well what kind of transformations should our model be invariant to. For example we can not use a horizontal flip transformation on a task where we would like to distinguish the left from the right hand in an x-ray image, unless we also change the label accordingly.

State-of-the-art CNN classifiers

Beyond the foundational aspects of CNN models discussed so far, state-of-the-art models include many further developments. We discuss this evolution by highlighting a number of impactful solutions.
Historically the CNN model was introduced with the LeNet architecture[7]. In Figure 22, you can

[7]Though earlier versions were already published as early as 1989 (LeCun, Y.; Boser, B.; Denker, J. S.; Henderson, D.; Howard, R. E.; Hubbard, W.; Jackel, L. D. (1989). "Backpropagation Applied to Handwritten Zip Code Recognition". Neural Computation. MIT Press - Journals. 1 (4): 541–551.) with the core concepts published even earlier (Fukushima, Kunihiko (1980). "Neocognitron: A Self-organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position")

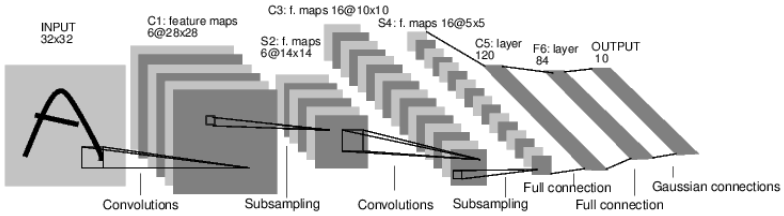


Figure 22: LeNet model. LeCun, Yann, et al. "Gradient-based learning applied to document recognition." Proceedings of the IEEE 86.11 (1998): 2278-2324.

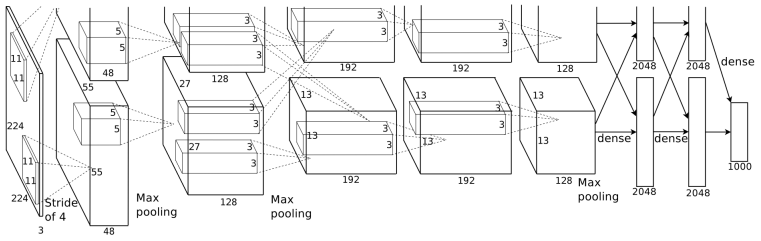


Figure 23: The AlexNet: Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." Advances in neural information processing systems. 2012.

see on depiction of LeNet where rather than layers and neurons you can see the activation maps and the kernels. You can work out the architecture based on this activation maps. For example you can see that the first convolutional layer has 6 neurons based on the dimensionality of the first activation map.

A more recent model and arguably one of the models that is most responsible for the increased attention to this field is AlexNet. AlexNet won the ImageNet Large Scale Visual Recognition Challenge in 2012 with a very large margin and set the stage for further developments in deep learning. The ImageNet dataset consists of color images with resolution of 224 by 224 labeled with 1000 different classes. The model is depicted in fig. 23 with the activation maps.

Following the success of AlexNet a number of models the rate of development of new and improved models rapidly increased, particularly on the ImageNet task. One of these models is VGGNet Creffig:vgg-network. VGGNet introduced models with much larger complexity. The model also reduced the number of decisions that you would need to take by standardizing the size of the kernels to 3 by 3. This model (or variations) of it is still very useful today. For many tasks that do not present complexity as large as ImageNet a variation of VGGNet (typically with fewer blocks) can deliver satisfactory performance. Due to its simplicity in implementation it can be a good choice as a first attempt at a suitable task.

One of the main drawbacks of VGGNet was the large number of parameters. Relative to its depth VGGNet uses many parameters, particularly after the last convolutional layer, the dense layer that follows uses a large about of parameters to process that last activation map.

The GoogleNet model (Figure 25) that relies on the inception block addressed this challenge. This architecture is deeper and uses less parameters that VGGNet. It also delivers superior performance.

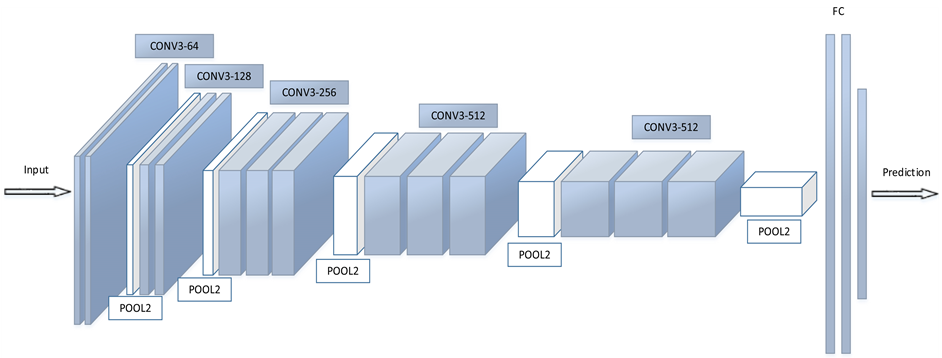


Figure 24: VGGNet: Simonyan, Karen, and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition." arXiv preprint arXiv:1409.1556 (2014).

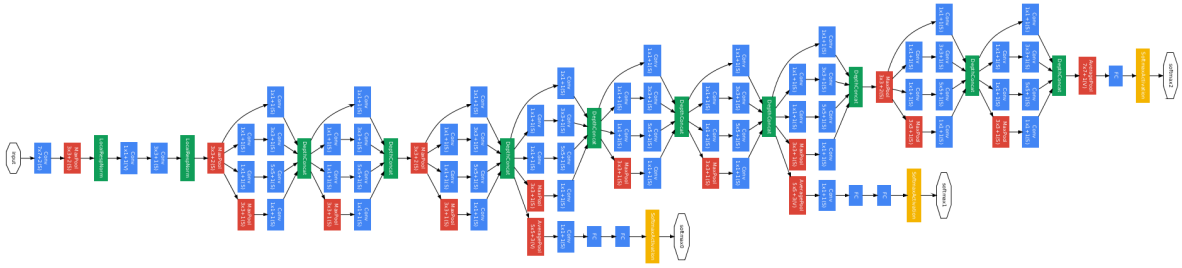


Figure 25: GoogleNet: Szegedy, Christian, et al. "Going deeper with convolutions." Cvpr, 2015.

The architecture consists of a number of inception blocks that are combined together. The model also uses multiple output layers to improve the flow of the gradient updates deeper in the model and in turn improve the training time.

The inception block, deals with the choice of the size of the kernel by actually using multiple kernel sizes in parallel. Specifically, the inception model uses 1×1 or 3×3 or 5×5 kernels Figure 26.

Note: What is the point of using a 1×1 kernel? Such a filter certainly can not learn spatial patterns. However, with such a kernel we can combine all the activation on a particular location, or specifically we can learn how to combine them. One other way to think of the 1×1 kernel is that it offers a way to change the depth of the activation map. This is very closely related to the computional complexity of the model. A 1×1 kernel can enable us to change the depth of activation maps. This trade off between capacity and complexity enables us to optimize for different settings.

Even though a number of advances were introduced in the models so far (ReLU activations, multiple output layers), the vanishing gradient effects still limits the performance of very deep models. The ResNet model has brought a new innovation that enabled much deeper models. The ResNet architecture introduces the residual block (Figure 28). The residual block introduces the skip connection, an identity map between the input and the output. The map allows for gradients to flow without passing through the non-linear layers as they are positioned in parallel to the skip connection.

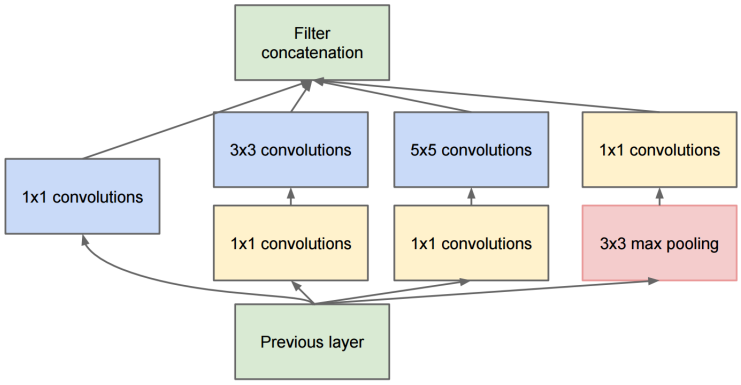
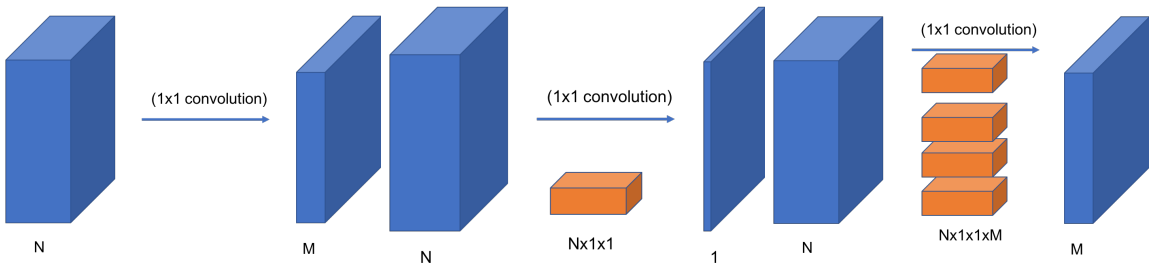


Figure 26: Inception module



(a) $1x1$ convolution (b) $1x1$ convolution (c) $1x1$ convolution

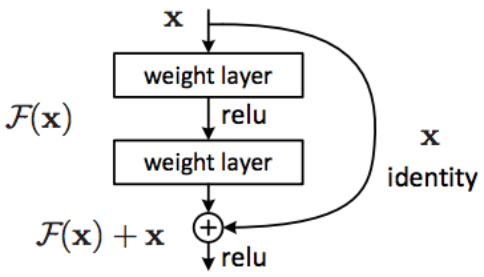


Figure 28: Residual block
8

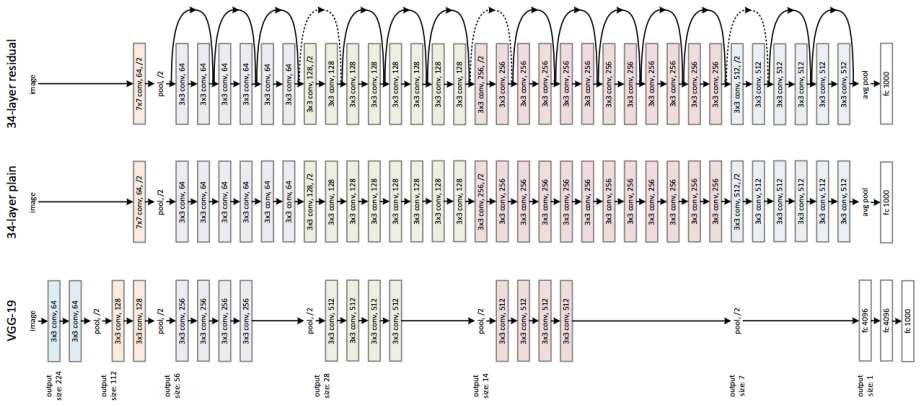


Figure 29: Residual Network
9

Using the residual block the ResNet model can achieve improved performance by using a deep architecture. In Figure 29 a 34-layer deep architecture is depicted in comparison to a 19-layer deep architecture of VGGNet.

Some of these advancements have later on been combined, such the Inception blocks with skip connections that further improved the performance of the models. These particular architectures are currently no longer state-of-art and have been surpassed by even newer architectures, but have certainly been influential and very commonly used on many tasks. We leave even more recent developments out of the scope of this paper.

Besides classifying images there are other tasks we can solve with CNN models in the domain of image analysis. Each such task comes with decisions regarding the input and output encoding, the model output, the loss function(s), model architecture and training configuration. We continue with a overview that is by no means complete, but should give you an idea of what kind of tasks you may encounter and what kind of choices you could make.

Image localization

The task of image localization, extends the task of object detection into object detection and localization in the image. Typically, we formulate the task of object detection as a machine learning classification task as given in the MNIST example above. For image localization we have the following formulation; detecting the type of object is formulated as classification and identifying the location of that object is well suited for the regression formulation. As such, the model has multiple outputs that have different roles and accordingly the loss function has multiple terms corresponding to each output and its role.

On a conceptual level we define the notion of a *bounding box*. A bounding box can be represented by four numbers: an x-coordinate, a y-coordinate, its width w, and its height h. The content of the bounding box can again be encoded as a one-hot vector, and the prediction is again a distribution over the possible classes. We then use a bounding box to localize an object in the image (Figure 30). As such this task requires that we have corresponding annotations for each image. For each object in the image we need an annotation of its bounding box. Our model needs to be able to output a

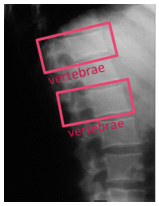
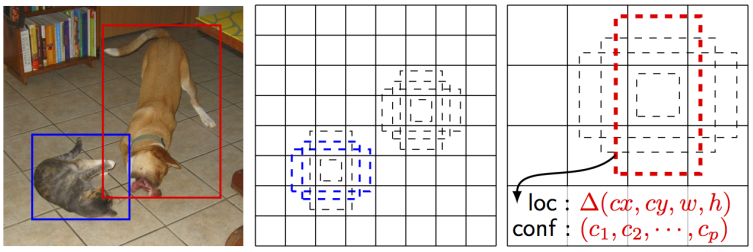


Figure 30: Localization of objects



(a) Image with GT boxes (b) 8×8 feature map (c) 4×4 feature map

Figure 31: YOLO - you only look once

desired number of bounding boxes.

We can then use the following loss functions for each bounding box:

- MSE for the location and size of the boxes;
- Cross-entropy for the content of the boxes.

In practice we would have a CNN model with four outputs for each bounding box that do regression and one output that does classification.

Question: How can we use these losses to deal with a shortage or surplus of bounding boxes?

State-of-the-art CNN localization

The "You only look once" (YOLO) architecture[10] develops a solution for image localization by splitting the image in cells and assign each cell an number of bounding boxes. Each bounding box has (x, y, w, h, l) and distribution over the type of objects present (**??**).

The details of the architecture are given in Figure 32.

Image segmentation

Image segmentation means partitioning the pixels of an image into segments. This can be used for object detection, but is also useful for other recognition tasks. It comes down to classifying each pixel, so our output is a width $\times$ height $\times$ c tensor of values between 0 and 1 where c is the number of classes. In order to get the right output format (a probability distribution per pixel) we can

[10]Redmon, Joseph, et al. "You only look once: Unified, real-time object detection." Proceedings of the IEEE conference on computer vision and pattern recognition. 2016.

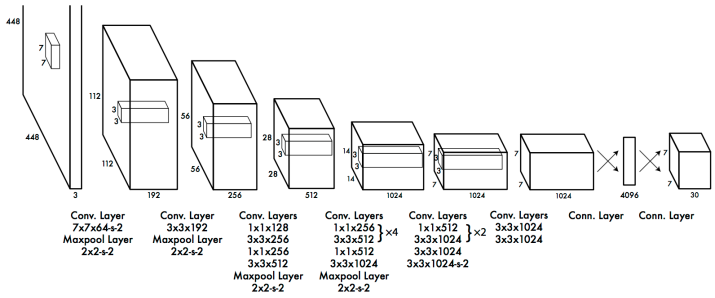


Figure 32: YOLO - you only look once

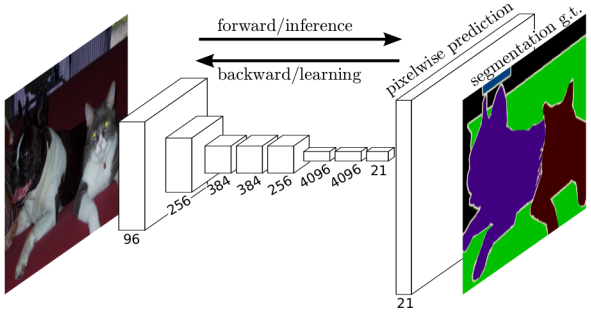


Figure 33: Image Segmentation

apply the softmax function over the last axis of this tensor. A good loss function for training a model for image segmentation is pixel-wise categorical cross-entropy (Figure 33.

State-of-the-art CNN for segmentation

One highly successful architecture of image segmentation is UNet (Figure 35). The difficulty in image segmentation is that information about the regions that need to be segmented in the image are present both globally in the image and locally in small patches. The global context brings information about the type of object with respect to other objects in the image and the local information gives the ability to precisely separate the borders of the object. Looking at a medical imaging example in Figure 34. The different organs are segmented in the image. The type of the object is needs to be determined from the global human anatomy present is such images, while for the border of each image we need precise pixel information such that we can delineate the edges.

This challenge is addressed by the UNet architecture. The model processes the input image in stages by iterativelly bringing the information into a more compact representation. This series of compatifications form a global context at different levels. This forms the bottom U-shaped channel of the model. To carry the local information needed for detecting the edges of the object the model introduces horizontal 'skip' connections where context and different levels is being mixed the high level information.

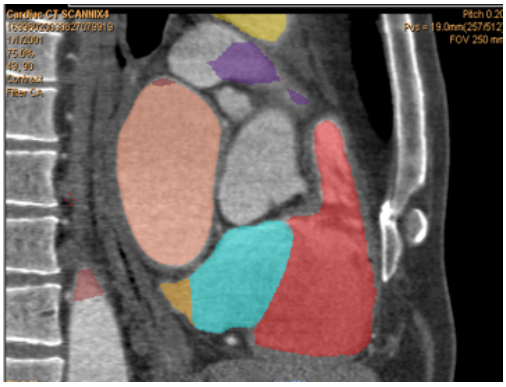


Figure 34: Image Segmentation

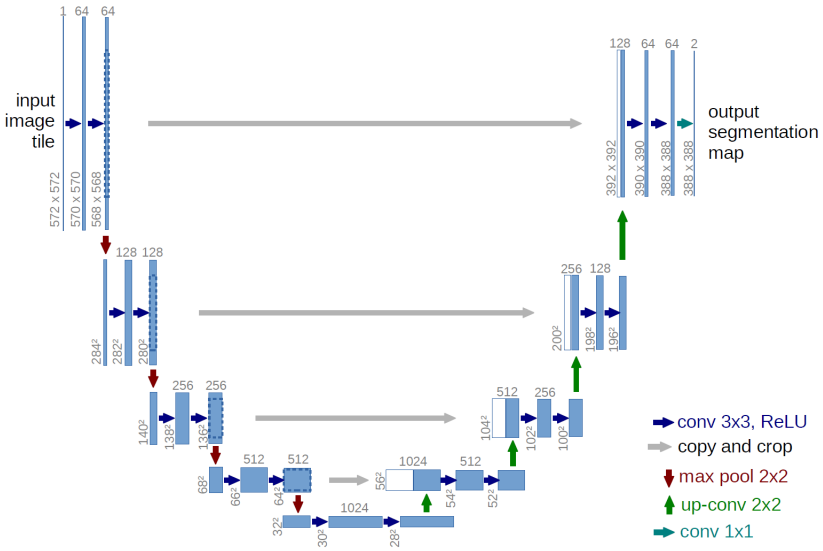


Figure 35: UNet CNN for image segmentation: Ronneberger, Olaf, Philipp Fischer, and Thomas Brox. "U-net: Convolutional networks for biomedical image segmentation." International Conference on Medical image computing and computer-assisted intervention. Springer, Cham, 2015.

Figure 36: Input image → output (transformed) image (Gatys, Leon A., et al. "A neural algorithm of artistic style. arXiv 2015." arXiv preprint arXiv:1508.06576 (2015).)

Filtering

There are multiple approaches to removing noise from images with neural networks. One simple approach is to have a network that gives an output the same size as the input, and for which the loss during training is the mean squared error between the predicted image and the original noiseless image. A particularly efficient architecture for such a task is the fully convolutional model introduced before. The concept of introducing a noise to the input data and training a model to reconstruct the orignial image (without noise) is also related to the denoising autoencoders, which we discuss in more detail later in the chapter. Filtering, nevertheless, has also a broader set of applications. We do not go in more details about such applications here, but we introduce one architecture that should illustrate the broad range of possibilities.

In the work Gatys et al. a model is presented that can copy the artistic style of a given image an apply it to another image (Figure 36). This method is a good illustration of how a CNN model can be used in a task where we need to generate an image.

Transfer learning

Some of the models that we have discussed so far are very large and have been trained extensively in order to get very high levels of accuracy. When we want to do image analysis in practice training these models from start is often not feasible for several reasons:

- We have much less data than what is typically used to get these state of the art results.
- We don't have the expensive hardware or the large amount of time required to train these models from start.
- The task we want to perform is slightly different, e.g. we want to do classification with a different number of classes.

Fortunately, it is possible to re-use (parts of) models trained on one dataset and one task on a different task or dataset. This is called transfer learning.

For CNNs, this shouldn't come as a surprise if we think about our motivation for using convolutional layers: the aim was for filters to pick up on increasingly high level local patterns that are relevant in a broader context. Thus the features learned by the convolutional layers in a CNN trained on image classification should be useful for doing image segmentation as well.

In Section 4.1 we talked about how many CNN architectures consist roughly of two parts: a convolutional part and an MLP stacked on top of that[11]. A basic way of doing transfer learning with such models is the following. If we have trained a full CNN with such a two-part architecture on a dataset D_1 with task T_1, and we want to use what it has learned to perform another task T_2 on a dataset D_2, we can simply take the (trained) convolutional part of the model, and build a new fully connected network on top of that. The new network can than be trained for a relatively short period of time on D_2 to perform T_2. This way the representations the first CNN has learned on the first task are transferred to the new model and can be used for the second task. Alternatively, in many cases it is possible to even keep most of the original MLP and only change the final (output) layer.

When training the new model we can either freeze (parts of) the transferred layers or train the full model, so called *fine tuning*. Fine tuning can help get better results, but when there is little data available in D_2 it can also increase the risk of overfitting. Moreover, when the model is very large you have insufficient computational resources, freezing the transferred part of the model can make training it much cheaper because you don't have to do back-propagation over those layers. *Question: why is this? And under what circumstances does this not hold?*

4.4 Representation learning of images

In chapter 1 we talked about how we want to automatically learn meaningful features from complex data so that we can use those features for various (machine learning) tasks. Then in Section 4.3 we discussed how we can use layers of a CNN trained on one task to build models that perform some other task. This strongly suggests that a CNN indeed learns meaningful, or at least useful, features. In this section we will explore this property of CNNs further.

Autoencoding

So far the tasks that we considered fall into the category of supervised learning. In a broader context we can imagine a scenario where we do not have annotations for part or all of the datapoints. Even for such a scenario there are task for which need more efficient representation of the data. For example an image retrieval task would benefit from an efficient representation of the data where computing the distance between the datapoints is more meaningful (but also more efficient) then computing the distance in the image space. Furthermore, in light of the transfer learning discussion from above, we may have a scenario where we would like to develop features in unsupervised setting that we can transfer in a supervised setting and therefore improve the efficiency of the supervised training specifically with respect to the amount of annotations.

Such reprsentations can be learning with the autoencoder architectures. The autoencoder learns a compressed representation of the data by learning how to reconstruct the data after passing it though some kind of inofmration bottleneck. Specifically, an autoencoder consists of two parts: an encoder f and a decoder g (fig. 37) . These two parts are trained such that the decoder is an approximate inverse of the encoder on the target data, i.e.

$$g(f(x)) = x$$

[11] Of course this second part doesn't necessarily have to be an MLP, or in general even a fully connected network. The point is mainly that the second part is either non-local, or task specific.

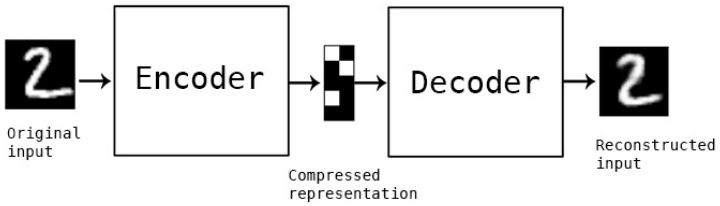


Figure 37: A conceptual representation of an autoencoder (https://blog.keras.io/building-autoencoders-in-keras.html)

for all x in the data set. The encoded data,

$$h = f(x),$$

is then the representation learned by the autoencoder. However, we don't want $g \circ f$ to be the identity map on all possible input data, otherwise it would not have to learn a useful representation. Therefore we need to introduce some kind of restriction to the autoencoder.

An autoencoder architecture was introduced in fig. 38. A convolutional version of an autoencoder also inclused convolutional layers fig. 39.

To train the autoencoder model we need to use a loss function that computes the precision of the reconstruction compared to the input. As a basic loss function we can use $L(x, g(f(x)) = \|x - g(f(x))\|^2$, or scale this by the number of pixels to get the mean squared pixel-wise error.

Using a different loss function allows us to bias the model towards a particular representation. We can represent our domain knowledge about what in the dataset is signal and what is noise via the loss function. For example, we may only care about the intensity of the pixel values, but not for their color. We can then define a loss function that first converts both images into black and white encoding and then computes the mean square error of the black and white values.

There are different approaches to introduce a bottleneck in the autoencoder models. The most direct approach is to reduce the number of neurons in one or more layers in the autoencoder. In this approach the narrowest layer defines the bottleneck of the capacity of the model. The capacity of the learned representation is determined by this layer, which is typically the representation layer. This models reduces the dimensionality of the input to the dimensionality of the representation layer. Note: In the chapter on generative models we discuss at another technique for regularizing autoencoders called the "variational autoencoder". There we use the encoder and decoder to parameterize conditional probability distributions on the spaces of data, and on the space of representations, the so called "latent space". Even though, such a model also fits the definition of an autoencoder, it is more suitable to study this model in the context of generative models.

Sparse Autoencoder We can induce particular properties of the representation of the autoencoder model by introducing specific a loss function on the latent space. This gives us regularized autoencoders with a loss function of the form

$$\text{Loss}(x, g, f) = L(x, g(f(x)) + \Omega(f(x)) = \|x - g(f(x))\|^2 + \Omega(f(x))$$

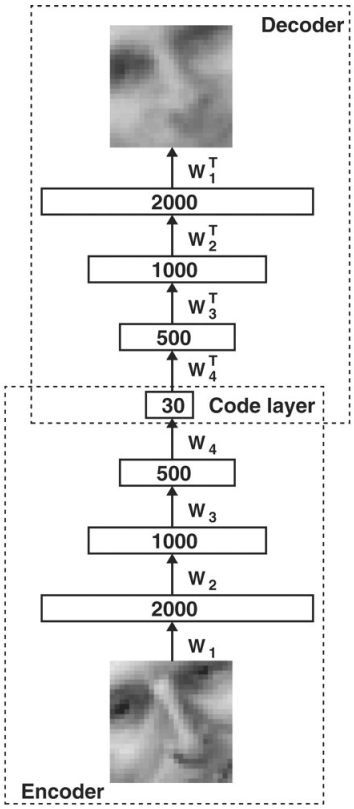


Figure 38: MLP Autoencoder: Hinton, Geoffrey E., and Ruslan R. Salakhutdinov. "Reducing the dimensionality of data with neural networks." science 313.5786 (2006): 504-507.

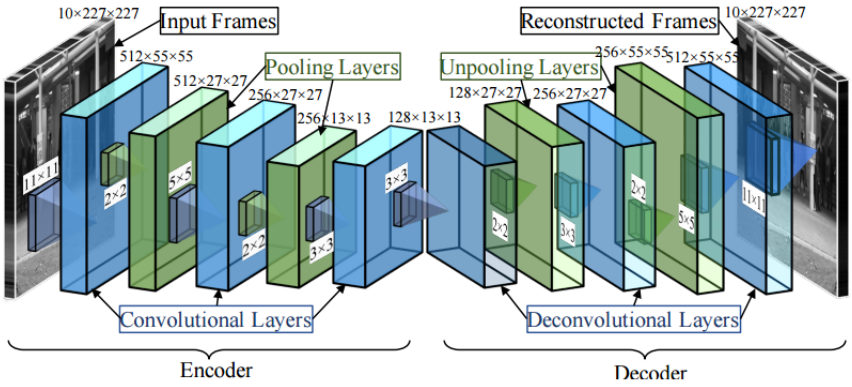


Figure 39: Convolutional Autoencoder

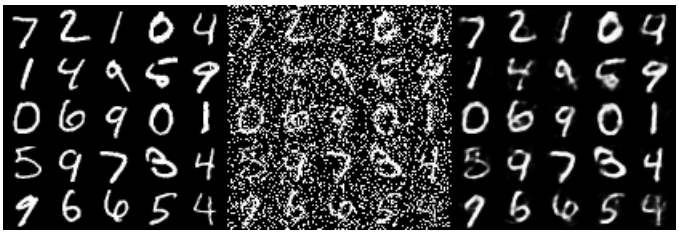


Figure 40: De-noising Autoencoder - Bengio, Yoshua, et al. "Generalized denoising auto-encoders as generative models." Advances in Neural Information Processing Systems. 2013.

For example, we may aim to develop a sparse representation[12] of our data. Such a sparse autoencoder can be achieved by introducing L^1 regularization on the output of the representation layer:

$$\Omega(h) = \sum_i |h_i|.$$

De-noising Autoencoder Another way to avoid learning the identify function is to introduce noise to the input — see Figure 40. The noise acts as a bottleneck as the capacity of the model is now determined by the signal-to-noise ratio. In other words, the variance of the noise determines how close two values in the signal can be discerned from each other. If we introduce through some (stochastic) function $n(x)$, the loss we try to minimize during training is given by

$$L(x, g(f(n(x))).$$

Moreover, the denoising autoencoder can also be interpreted as a generative model, as the noise forces the f and g to implicitly learn $p_{data}(x)$. We will look at generative models in more detail in the last chapter of this course.

One-shot/metric learning

Metric learning techniques develop a distance metric between the data points in the dataset. There are a number of tasks that can benefit from a distance metric that captures the semantic properties of the data. We already mentioned the retrieval task in the context of the autoencoder. However, the autoencoder as an unsupervised model can only offer a compressed representation of the data. Metric learning techniques allow us to learn various distance metrics based on available supervised information about the data. A model that produces such a metric enables solving various targeted retrieval or recommendation tasks. Moreover, these techniques closely correspond to the one/few-shot classification task. The typical classification task includes a fixed number of classes and a sufficient amount of examples per class. In contrast to this we can have a task that has a large number of classes and only a few (or even one) example per class. In this section we study in details the methods for this types of tasks.

One-shot learning In the supervised learning setting, the model eventually learns the most salient features that predict the values for the target variables. When we only have very few (even one) example datapoint per class, there is little opportunity for the right patterns to be distinguished from the rest of the information in the data. Take for example the task of facial recognition. For

[12]Sparse representations are representations where the vast majority of entries are zero.

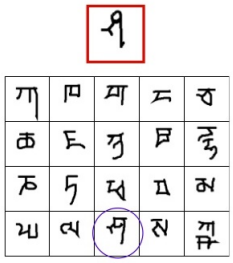


Figure 41: One-shot learning

this task we may have a large number of training points, but we also have a large number of classes, as we may only have one photo per person. Furthermore, this task will likely require a solution that can adapt to adding new people without the high computational cost of re-training the whole model.

So, now rather than developing a model that can detect specific classes, we are better off training a model that can learn the characteristic features for faces and then introduce the downstream task of detection. One technique that allows developing more general representations of data is metric learning. Metric learning methods develop a model that maps the data to a metric space where the distance captures a given semantic property. Here we will be looking at deep metric learning, where the mapping is given by some deep neural network.

In our face detection example, the goal of the model would be to represent the data in a space where images of faces belonging to the same person would be closer together than images of different people.

In this setup, the model has a better chance to develop features that distinguish faces of people and use them to generalize even when given a single example of the face of a person. Furthermore, adding a new person to the dataset may require little or even no additional training.

Metric learning The goal is to learn a distance metric d and some value τ such that for images of the same class we have

$$d(I^{(1)}_{c=1}, I^{(2)}_{c=1}) < \tau$$

and for images of different classes we have

$$d(I^{(1)}_{c=1}, I^{(2)}_{c\neq 1}) > \tau.$$

One method to develop such a metric is based on the Siamese network model — see Figure 42. Here we learn a distance metric d of the form

$$d(x_1, x_2) = \|f(x_1) - f(x_2)\|^2 \tag{5}$$

for some function f parameterized by a deep neural network. The goal is thus to train the network f such that

- if x_i and x_j are from the same class $\|f(x_i) - f(x_j)\|^2$ is small;

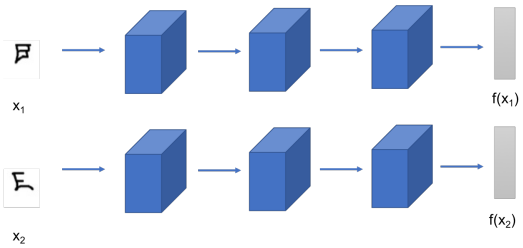


Figure 42: Siamese Network: Taigman, Yaniv, et al. "Deepface: Closing the gap to human-level performance in face verification." Proceedings of the IEEE conference on computer vision and pattern recognition. 2014.

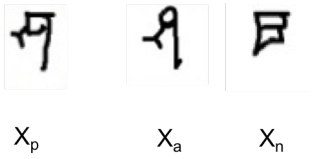


Figure 43: Triplet Network[15]

- if x_i and x_j are not from the same class $\|f(x_i) - f(x_j)\|^2$ is large.

[13] To achieve a better robustness to the margin parameter, τ, the triplet network method introduces a more robust training setup including three datapoints and a three component loss function.

Triplet Network The idea behind the triplet network — see Figure 43 — is to show three images to the same network: two images from the same class x_p and x_a, and an image from a different class x_n. We call x_a the anchor (or baseline) input, and x_p and x_n the positive and negative input respectively. The goal is then to learn a metric such that the distance between the positive input and the anchor is less then that between the negative input and the anchor. I.e.

$$d(x_p, x_a) \leq d(x_n, x_a)$$

or in terms of a learned metric (5)

$$\|f(x_p) - f(x_a)\|^2 \leq \|f(x_n) - f(x_a)\|^2 .$$

We can rewrite this as

$$\|f(x_p) - f(x_a)\|^2 - \|f(x_n) - f(x_a)\|^2 \leq 0.$$

Now in order to build in more robustness, we want this inequality to hold by a margin $\alpha > 0$, i.e.

$$\|f(x_p) - f(x_a)\|^2 - \|f(x_n) - f(x_a)\|^2 + \alpha \leq 0.$$

In order to achieve this we use the *triplet loss* given by:

$$L(x_p, x_n, x_a) = \max(\|f(x_p) - f(x_a)\|^2 - \|f(x_n) - f(x_a)\|^2 + \alpha, 0).$$

[13] Hoffer, Elad, and Nir Ailon. "Deep metric learning using triplet network." International Workshop on Similarity-Based Pattern Recognition. Springer, Cham, 2015.

[14]Selecting the triplets is important and has a significant impact on the efficiency of the metric learning. This is an active research area and out of the scope of this document.

Triplet Selection

The number of all possible triplets is k combination of n, where k is 3, and n is the number of datapoints in the dataset. Training with all of the triplets is typically not feasible. Nevertheless, to achieve a good embedding (i.e. learning a discriminative distance metric), we need only a small fraction of all possible triplets. However, the quality of the embedding depends significantly on the selected triplets as not all triplets as equally informative.

Based on the definition of the loss, there are three categories of triplets.[15]

easy triplets: triplets which have a loss of 0, because:

$$\|f(x_p) - f(x_a)\|^2 - \|f(x_n) - f(x_a)\|^2 + \alpha \leq 0.$$

hard triplets: triplets where the negative is closer to the anchor than the positive, i.e.

$$\|f(x_n) - f(x_a)\|^2 < \|f(x_p) - f(x_a)\|^2$$

semi-hard triplets: triplets where the negative is not closer to the anchor than the positive, but which still have positive loss:

$$\|f(x_p) - f(x_a)\|^2 < \|f(x_n) - f(x_a)\|^2 < \|f(x_p) - f(x_a)\|^2 + \alpha$$

Each of these definitions depends on where the negative is, relatively to the anchor and positive. We can, therefore, extend these three categories to the negatives: hard negatives, semi-hard negatives or easy negatives. The figure below shows the three corresponding regions of the embedding space for the negative as depicted in Figure 44.

Easy negatives are less informative and will contribute no gradients to our training. Hard negatives contribute most to training but sometimes this selection method might result that too many triplets are consisting of mislabeled and poorly imaged samples, specifically it might lead to a collapsed model (i.e. $f(x) = 0$). Semi-hard negatives selection, as a trade-off option, select relatively more informative triples as well as avoid model collapsing.

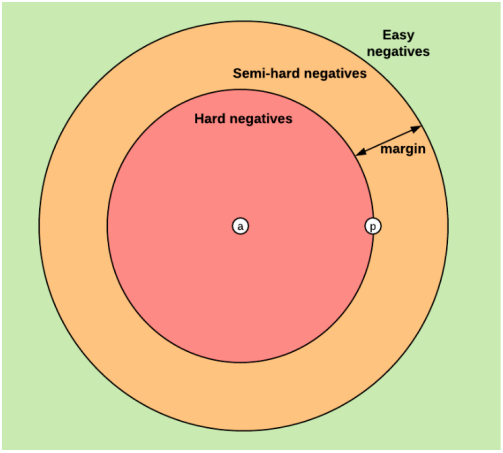


Figure 44: The three types of negatives, given an anchor and a positive

[14]Hermans, Alexander, Lucas Beyer, and Bastian Leibe. "In defense of the triplet loss for person re-identification." arXiv preprint arXiv:1703.07737 (2017).

[15]Schroff, Florian, Dmitry Kalenichenko, and James Philbin. "Facenet: A unified embedding for face recognition and clustering." Proceedings of the IEEE conference on computer vision and pattern recognition. 2015.