Chapter 5

CNN, Autoencoder and GAN

Before we start to talk about convolutional neural networks, generative adversarial networks and autoencoders, we will take a closer look at the ReLU activation function.

5.1 Rectified linear unit (ReLU)

The rectifier activation function is defined as,

$$g(x) = \max(0, x)$$

and is one of the most popular activation functions for deep neural networks, especially for convolutional neural networks. A smooth approximation to the ReLU is given by the so called *softplus* function,

$$g(x) = ln(1 + e^x)$$

where the derivative is given by the logistic function, $g'(x) = e^x/(e^x + 1) = 1/(1 + e^{-x})$. Both ReLU and softplus are shown in figure 5.1. Why has ReLU become so popular? There are some definite advantages of using this activation function, such as,

- Sparse activation. Suppose we have normalized inputs and uniform initialization of the weights. For a given input pattern about 50% of the nodes will have a zero output and the rest will be linear.
- No vanishing gradients.
- Fast computation.
- More biological plausible.

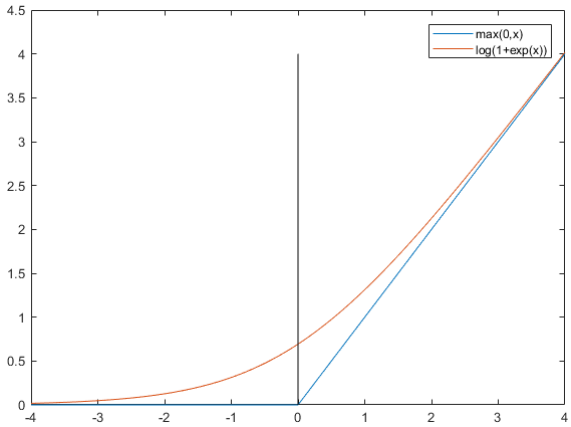


Figure 5.1: The ReLU activation function and a smooth version called softplus.

There are also potential problems with this activation function. There could sometimes be a problem of the function being unbounded. Dying nodes means that sometimes ReLU nodes can be pushed into "regions" where they are inactive (i.e. zero) for almost all inputs. This means that there will be no gradient pushing the node back to being active. Such nodes "dies" and are no longer used.

The universal approximation theorem that was discussed in connection with the MLP needed the activation function to be sigmoidal. There is however a newer version of the approximation theorem that allows for ReLU activation functions. This means that a neural network with ReLU nodes is also a universal "approximator".

5.2 Convolutional neural networks (CNN)

(See also chapter 9 in the Deep Learning Book).

CNNs are mostly used for image analysis. We can of course think of images in both 1D and in 3D, but the most applications work with "ordinary" 2D images. Each pixel in the images can however consist of several values. For a black and white image each pixel, denoted $I(i, j)$, is a scalar with only two possible numbers. In a gray-scale image each pixel is still a scalar, but can take a range of numbers, typically $[0, 255]$. For color images each pixel is represented by more than one value, where a common scheme is the RGB encoding, hence each pixel is a 3-dimensional vector.

5.2.1 Filters

To understand how the CNN architecture works we are going to start with the concept of image filters. Assume the our image is given by $I(i,j)$, for simplicity we can assume that each pixel is a scalar. A "filtered" image is mathematically described by the process of convolution. The filter is represented by a *kernel*, $K(n,m)$ and the convolved (filtered) image $h(i,j)$ is given by,

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i-m, i-n)K(m,n)$$

When implementing convolution for neural networks one often use another related function called *cross-correlation*,

$$h(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, i+n)K(m,n)$$

Here is a small numerical example where a 4x4 image matrix is filtered by a 2x2 kernel.

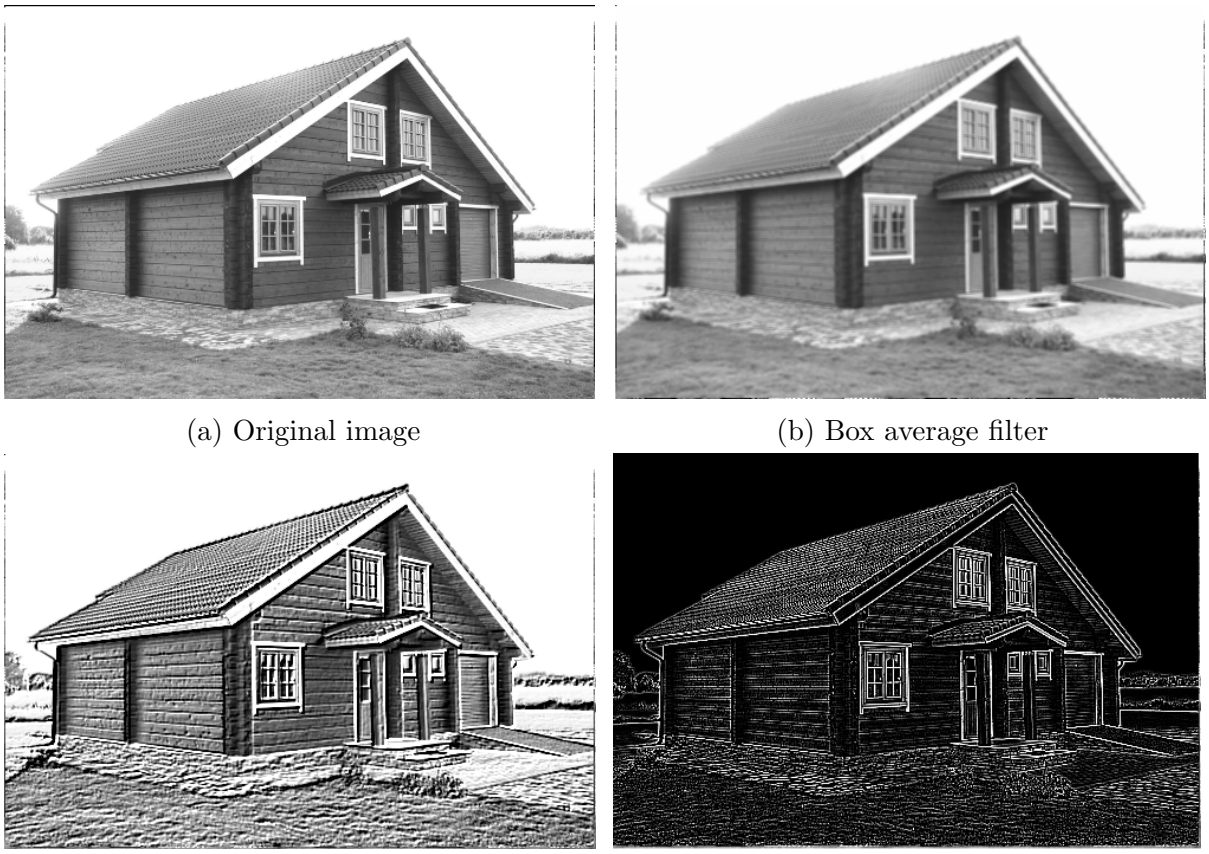
$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix} * \begin{bmatrix} -1 & 1 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} (-1+2+0+6) & (-2+3+0+7) & (-3+4+0+8) \\ (-5+6+0+10) & (-6+7+0+11) & (-7+8+0+12) \\ (-9+10+0+14) & (-10+11+0+15) & (-11+12+0+16) \end{bmatrix}$$

$$= \begin{bmatrix} 7 & 8 & 9 \\ 11 & 12 & 13 \\ 15 & 16 & 17 \end{bmatrix}$$

Note that the filtered image is smaller than the starting image. One can deal with the edges in different ways and here we placed the filter (kernel) inside the image. We will discuss various techniques to handle the edges later. What can we do with filters? Figures 5.2d - 5.2b show an experiment where an image of a house (Fig. 5.2a) has been filtered by different 3x3 kernels. The first filter is a box average filter, the second is a so called emboss filter and the third one is an "outline" filter that highlight large difference in pixel values. The actual kernels are specified here,

$$\text{box average} = \frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}, \qquad \text{emboss} = \begin{bmatrix} -2 & -1 & 0 \\ -1 & 1 & 1 \\ 0 & 1 & 2 \end{bmatrix}, \qquad \text{outline} = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

The e.g. "outline" kernel will make the image dark in areas where all pixels are the same and where neighbor pixels differ the image will appear white. We can have kernels that detects horizontal and vertical differences (edges), such as the Sobel kernels and kernels that



(a) Original image

(b) Box average filter

(c) Emboss filter

(d) Outline filter

Figure 5.2: A selection of three different filters applied on the top left image.

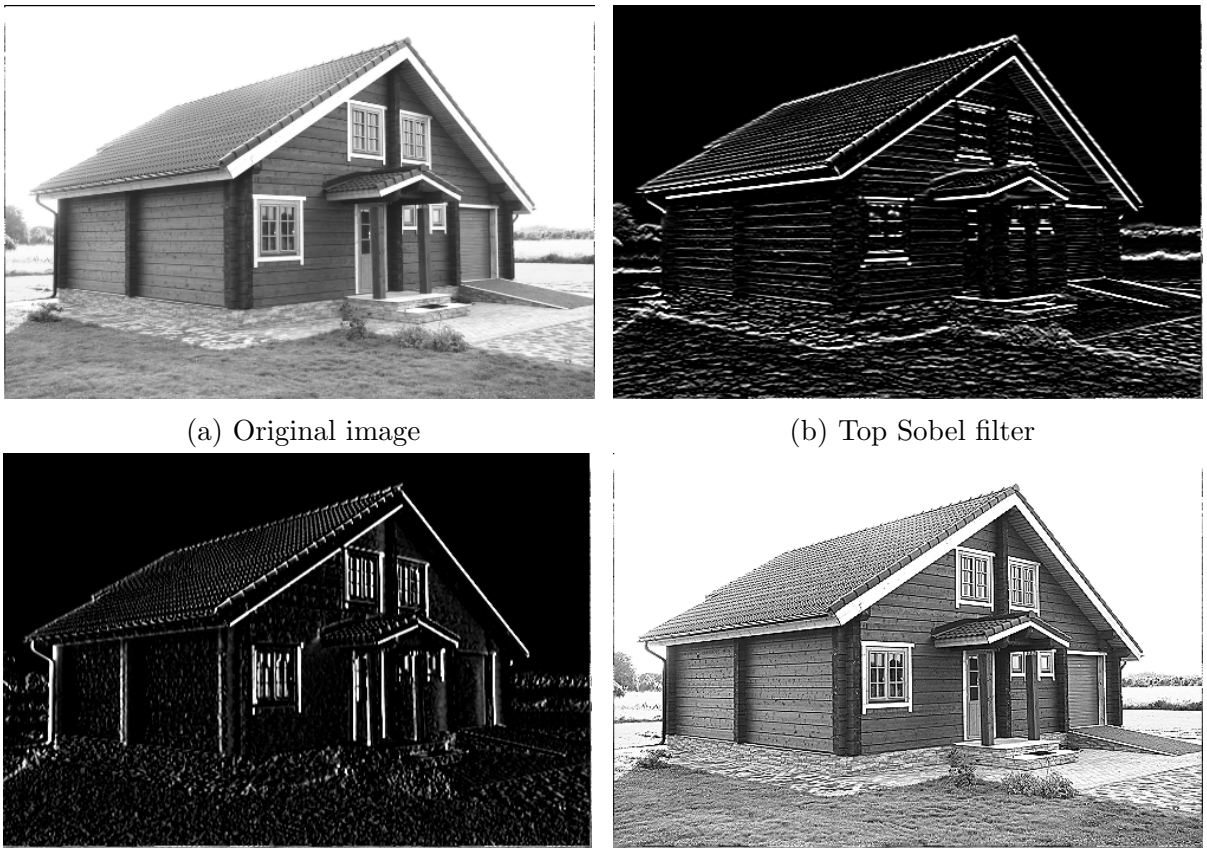
appears to "sharpen" the image. Below are examples of such kernels and the corresponding effect on the house can be seen in figures 5.3d - 5.3b.

$$\text{top Sobel} = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}, \qquad \text{left Sobel} = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}, \qquad \text{sharpen} = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

We can understand that using filters can be a good way analyzing images. A convolutional neural network is using trainable filters as the basic tool for the analysis. Next we will see how this is implemented in the network architecture.

5.2.2 Sparse connectivity and parameter sharing

The first property of the filters is that they only see a small part of the image. We can accomplish this in a network using a sparsely connected networks. To begin lets illustrate



(a) Original image

(b) Top Sobel filter

(c) Left Sobel filter

(d) Sharpen filter

Figure 5.3: A selection of three different filters applied on the top left image.

this using an ordinary MLP. Figure 5.4 shows the two versions of the input-to-hidden layer of an MLP. The top network is fully connected the usual way. Here a hidden node is connected to all of the input nodes. The nodes that affect a given hidden node is typically called the receptive field of this hidden node. The lower network in figure 5.4 shows the same input-to-hidden network, but with fewer connections. The receptive field for each of the hidden nodes is now smaller. The hidden node h_3 is only receiving input from inputs (x_2, x_3, x_4). This network has fewer number of weights compared to a fully connected network as a result of the sparse connectivity. We can interpret the hidden nodes activity with a filtered 1D images, except for one issue. In a filter process all kernel parameters stay constant as the kernel is moving over the image. In our 1D example in figure 5.4 the weights are not necessary the same for each hidden node.

Figure 5.5 shows the same 1D example as before, but now with a lot of weights being shared across the layer. All weights with the same color are being shared between the different hidden nodes. This means that e.g. h_2 is computing a weighted sum of (x_1, x_2, x_3) exactly

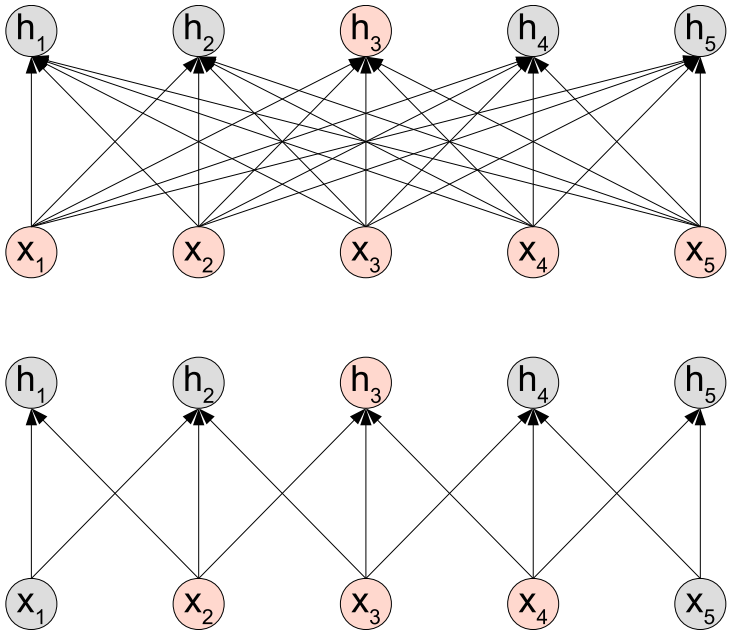


Figure 5.4: (Top) A fully connected input-to-hidden layer in an MLP. The highlighted hidden node (h_3) is receiving input from all input nodes. It has a large receptive field. (Bottom) The same MLP layer, but with fewer weights. The h_3 hidden node is now only receiving input from three input nodes. It has a smaller receptive field.

the same way as e.g. h_4 is weighting together (x_3, x_4, x_5). We have now accomplished exactly

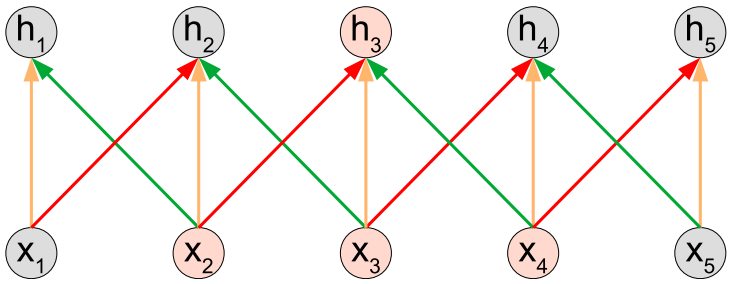


Figure 5.5: Sparse connectivity and **shared** weights. Weights with the same color are shared between the hidden nodes.

the same process of the image filter, but formulated as network with sparse connections and shared weights. The hidden activity is our filtered image. So far we have only worked in 1D, but it is rather trivial to arrange the input nodes of the network as a 2D grid of inputs (i.e. the pixel values of an image) and similarly arrange the hidden nodes as a 2D grid (i.e. the filtered image). Figure 5.6 shows an illustration of a 2D input layer and a 2D hidden layer. This *convolution* stage described above is fundamental to all CNNs. There are however more

computational blocks when constructing a complete CNN. We will talk about these blocks next.

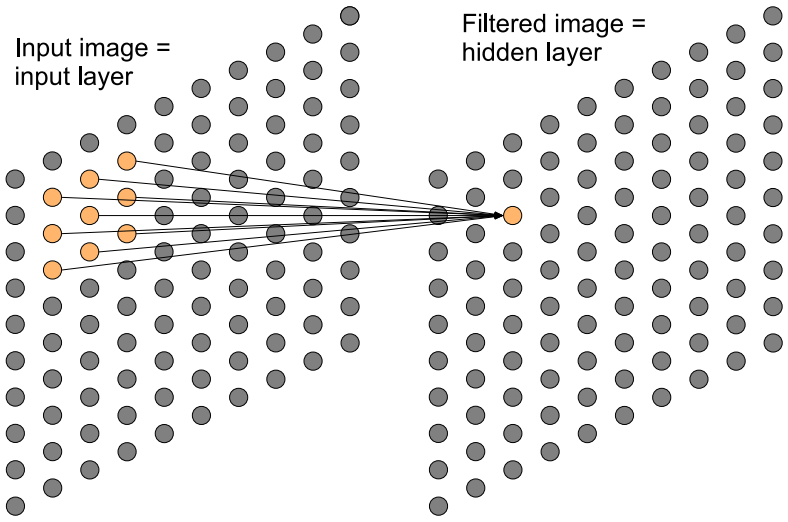


Figure 5.6: The input layer of an MLP where the inputs have been arranged in a 2D grid, representing the pixel values of an image. The hidden layer is also arranged as a 2D grid, representing the filtered image. Each hidden node is only connected to nine inputs in the input layer (the receptive field of the hidden node). This is visualized by the orange hidden and input nodes. The kernel in the convolution process used when computing the filtered image is here represented by the weight values of a given hidden node. Note: each hidden node share the weights with all other hidden nodes.

5.2.3 CNN building blocks

Following the terminology used in the Deep Learning Book, a convolutional layer consists of three blocks (see figure 5.7), the convolution stage, the activation stage and the pooling stage. The first two are vital, but the pooling stage can many times be omitted.

The activation stage typically consists of using the rectified linear (ReLU) activation for the hidden nodes. It it fair to say that most CNNs today use ReLU as activation function. The pooling stage consists of replacing the hidden layer with a new layer that can be said to summarize the neighborhood of a given hidden node. A very common pooling operation is *max pooling*. Max pooling consists of replacing the hidden node value by the maximum value of a small neighborhood around the hidden node. Figure 5.8 (left graph) is showing max pooling on a hidden layer of an (1D) MLP. Here the hidden layer is summarized by taking the max of the hidden node values. Each of the pooling nodes has a receptive field (three in this example) and the pooling node values is simply the maximum of the hidden node

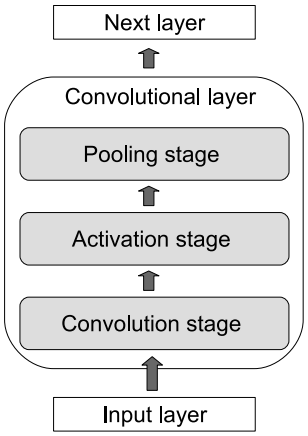


Figure 5.7: The three building blocks of a convolutional layer in a CNN. The convolutional stage, the activation stage and the pooling stage. The first two are fundamental, but the pooling stage can sometimes be omitted.

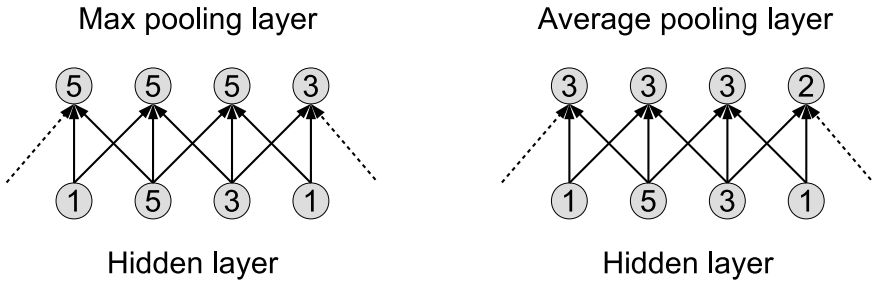


Figure 5.8: Illustration of two different pooling stages. The left graph is showing max pooling with a receptive field of three nodes. (The dotted line at the border indicates that other hidden nodes may be part of the receptive field, but they are not part of the calculations). The right graph is showing average pooling with also a receptive field of three nodes.

values. In the right graph of figure 5.8 an example of average pooling is illustrated. Here the average value of the hidden nodes, within the receptive field of a given pooling node, is used as the pooling layer output.

The pooling layer have two main advantages, first the pooling helps to make the representation approximately invariant to small translations of the input. The invariance means that if the input is moved by a small amount, the values of the pooling layer stays approximately the same. This is useful if we are interested in whether a specific feature exists, and not so much exactly where it is. Another advantage of using a pooling layer is that it can be used to downsample the filtered image. Because the pooling summarizes the hidden nodes with a specified receptive field, it is possible to use fewer nodes in the pooling layer. Figure 5.9 shows an example of max pooling with downsampling. This effectively makes the filtered

images smaller. You can find more detailed discussions on the convolutional- and the pooling

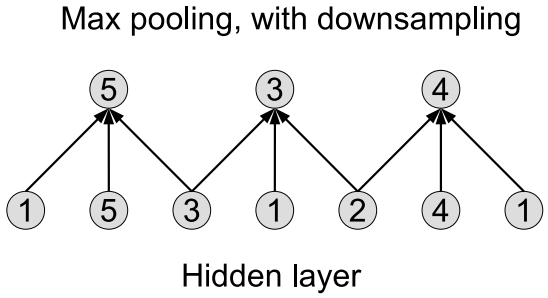


Figure 5.9: Max pooling with downsampling. We can think of the pooling operation as moving a filter along the hidden layer. The filter has a size of three, and is moved two steps (stride) after each operation. Here the seven hidden nodes are replaced by three nodes after pooling.

layer in chapter 9.1 - 9.8 in the Deep Learning Book.

5.2.4 Building a CNN network

Here follows more details of how to construct a complete CNN model. We will start by taking a closer look at the filter stage. Assume that we have a 7x7 input image (at the moment, only one value for each pixel). If we now have a 3x3 kernel we will produce a 5x5 filtered image, as illustrated in figure 5.10. The first thing we see is that the filtered image

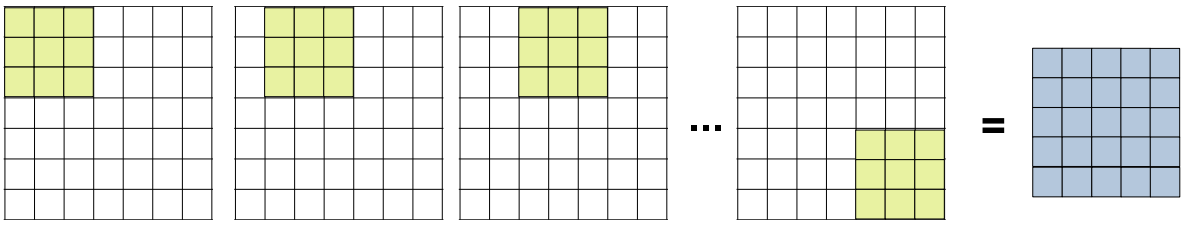


Figure 5.10: The image is 7x7 pixels and the kernel (green) is 3x3. Moving the kernel (fully inside) the image will produce a 5x5 filtered images (blue).

is smaller than the original image due to the choice of having the kernel operating within the image. For CNNs there is also a choice of moving the filter more than one pixel when convolving the image. The amount of pixels the filter is moved is called **stride**. Figure 5.11 shows the example above, but now with stride **2**. As we can see, the resulting filtered image is now even smaller (3x3). We will run into a problem if we use stride=3 for our 7x7 image with the kernel size 3x3, because the operation does not "fit". How can we solve

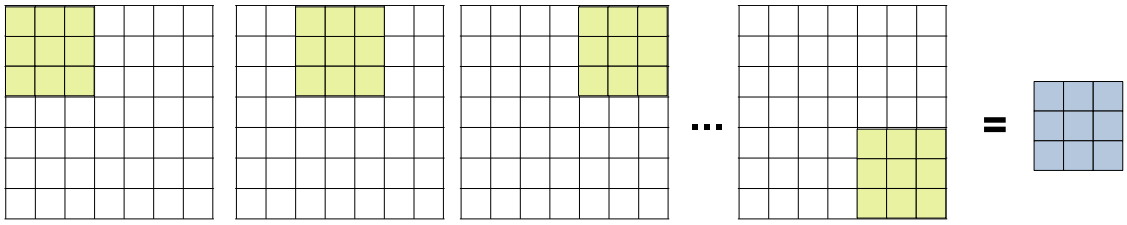


Figure 5.11: The image is 7x7 pixels and the kernel (green) is 3x3. Moving the kernel (fully inside) the image, but now with stride=2, will produce a 3x3 filtered images (blue).

this? Also, how can we make sure that the filtered image has the same dimension as the original image. The solution is called **padding** where zero-padding being the most common strategy. Zero-padding consists of adding a boundary of zeros to the original image so that the resulting filtered images has the desired dimensions. Figure 5.12 shows the how to zero pad the 7x7 image as to obtain a 7x7 filtered image with the 3x3 kernel. For the CNN it is quite common to work with 3x3 kernels, stride=1 and zero-padding=1. In many libraries it is common to write *padding='same'*, to indicate that the filtered image should have the same dimensions as the input image.

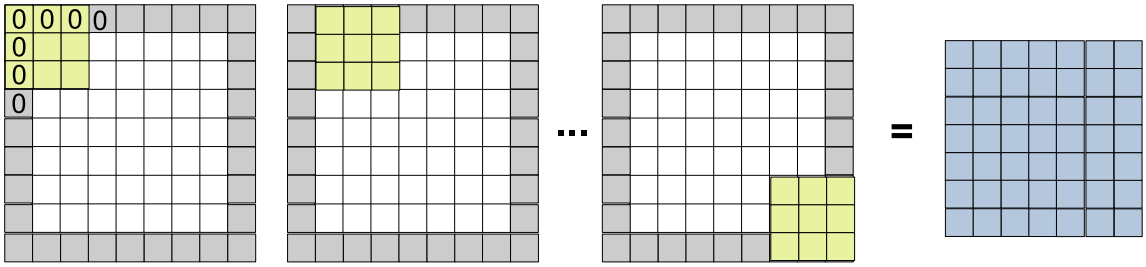


Figure 5.12: Zero-padding the images with a boundary of zeros will make sure that the filtered image has the same size as the original image.

So far we have only worked with single valued pixels, e.g. grayscale images. How do we handle *multichannel* images such as colored images using the RGB encoding scheme. Instead of a single image we now instead have three images, one for each color. We can view the image as having a certain *depth*. The filter will have the same depth as the images it is suppose to filter. This means that for RGB color images all filters in the first convolutional layer have depth three. This is illustrated in figure 5.13 where an 10x10x3 image is filtered using a 3x3x3 filter. We can think of this as having a weight vector ω (the filter) of size $3x3x3 = 27$ and a small chunk of the image represented by $\mathbf{x}$, also of size 27. The filter process is then just taking the dot product between these two vector. As usual we also add a bias b. The output, i.e. the hidden node value, before activation, for one particular "pixel" of the filtered image is then,

$$h = \omega^T \mathbf{x} + b$$

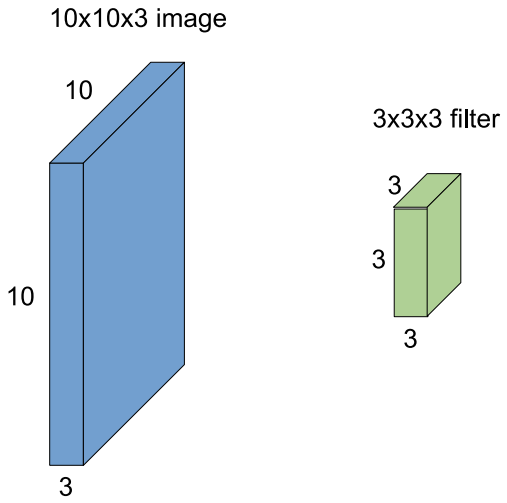


Figure 5.13: A 10x10x3 images and a filter of size 3x3x3. The important thing to remember is that filters always extends to the full depth on the input image.

We can now work with multichannel inputs, but what if we want to create multichannel outputs, i.e. more than one filtered image. Here we simply create more filters, for each filter we apply to the input image an additional channel is added to the filtered image, as illustrated in figure 5.14. Each of the filters will learn different features useful for the task

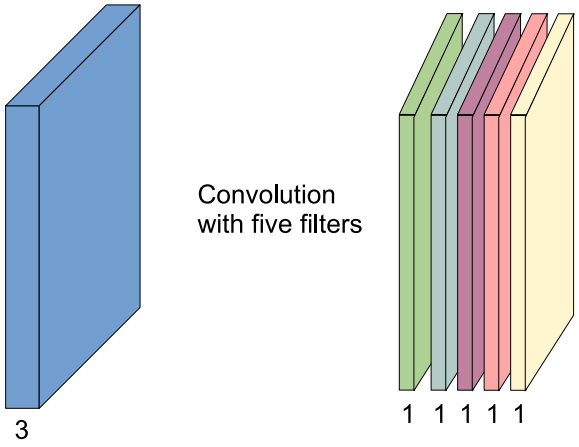


Figure 5.14: Here we have an input image of depth 3 (e.g. RGB image). Using five different filters will result in an filtered image of depth five (five channels).

in questions. We can imagine that filters learn to detect various kind of edges in the image, some filters may also learn to detect that specific colored edges may be of importance.

We are now at the point of building deeper CNNs. The next step is simply to use output

of the pooling stage (for all created filters) and apply yet another convolutional layer (remember figure 5.7). By repeating this step we can build a deep CNN. Figure 5.15 shows the architecture of one of the first successful models, LeNet, used for classification of handwritten characters. This paper[1] was published 1998. The last layer in the LeNet CNN (figure 5.15)

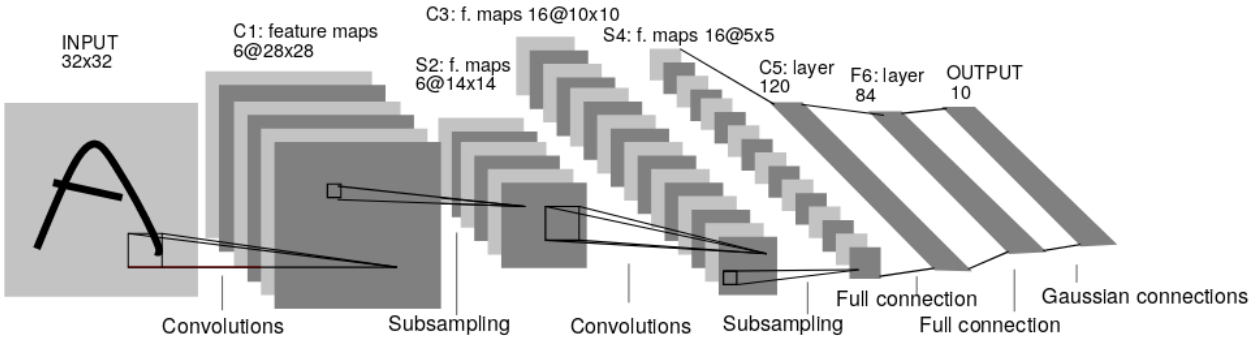


Figure 5.15: The CNN model of LeNet-5. This architecture was used to classify digits, hence the 10 output nodes. The term sub-sampling used in the figure is the same thing as the pooling layer.

consists of 16x(5x5) filters. These are then connected to a number of fully connected hidden layers, before feeding into the output layers consisting of 10 nodes. Taking the 16x5x5 filter nodes and rearranging then into a 1D vector is often referred to as a "Flatten" operation. This is the term used in Keras.

5.2.5 Further reading

There are many, many ways of building a CNN model and they can be used for many different problems. There is also a lot of research activity around CNNs, meaning that new models and new tools for understanding them are produced as we speak.

There are also many sites that provide good explanations of how CNN models work and how to use them. This site [2] provides an good overview of a few important CNN models.

[1] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. Proceedings of the IEEE, november 1998.

[2] https://adeshpande3.github.io/adeshpande3.github.io/The-9-Deep-Learning-Papers-You-Need-To-Know-About.html

5.3 The autoencoder

Now we are going to look at a very special architecture. It is called an autoencoder and is visualized in figure 5.16 In this network the size of the output layer is the same as the

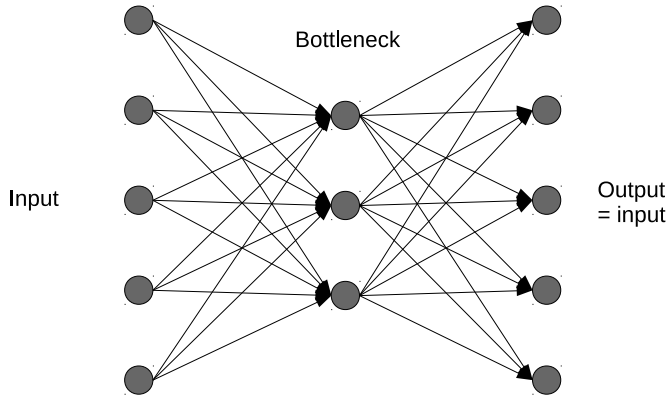


Figure 5.16: This special MLP, called an autoencoder, has the same number of output nodes as the inputs. The hidden layer, called bottleneck, always have a smaller size compared to the input size.

input layer. For the network shown in figure 5.16 we have one hidden layer. For a general autoencoder we can have more than one hidden layer (see below). The (middle) hidden layer, called the bottleneck layer, always have fewer hidden nodes than the input layer. In the figure we have indicated that the output is equal to the input, and this is the main idea of the autoencoder. The output should predict the input! This is a trivial problem to solve if we allow for the bottleneck layer to be at least as large as the input layer. However, the bottleneck should have fewer nodes than the input size, as in figure 5.16. Now it is not a trivial problem anymore. We can view the first half of the network as an encoder, where the input is encoded by the hidden layer. The second part of the network is then the decoder, where the hidden representation is decoded into the original input. As a conclusion we can think of the autoencoder as a network for finding effective representations of data. Note that we do not use any labels of the data.

To train an autoencoder we use the following error function,

$$E(\boldsymbol{\omega}) = \frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{P}\left(y_k\left(\mathbf{x}_n, \boldsymbol{\omega}\right) - x_{nk}\right)^2 \tag{5.1}$$

where N is the number of data in the training dataset and P is the number of inputs. Here we assume that the inputs $\mathbf{x}(n)$ are continuous rather than binary, so we can treat it as a regression problem. This is the reason for having the mean square error function. On the other hand if all inputs are binary we can change the error function appropriately. Typically, the error $E(\boldsymbol{\omega})$ is minimized using some standard stochastic gradient descent approach.

Interpretation By minimizing the error we force the encoder part of the network to find an efficient representation such that the decoder can re-create the data with no error. There is of course no guarantee that this will work. However depending on the type of activation function used we can decide what kind of "compression" to expect. In fact one can show that when using linear activation functions for the hidden (bottleneck) layer the autoencoder is doing a principal component analysis (PCA). If the hidden layer consists of M nodes, then by training the autoencoder the M hidden node weight vectors will correspond to the M principal component directions with the largest variance of the training data. If the data contains a lot of linear correlations among the input variables, then we can be successful by training an autoencoder using linear activation functions.

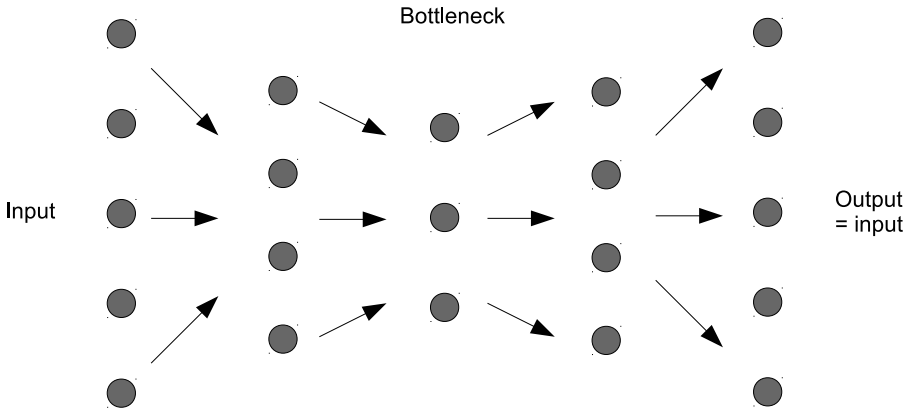


Figure 5.17: An example of an autoencoder with 3 hidden layers.

If the data is more complex we will need a more complex encoder to efficiently represent the inputs by the activations of the bottleneck layer. The first choice is to use non-linear activation functions, such as the tanh() or the sigmoid. Using non-linear activation functions will make the encoder a bit more complex, but to really improve the mapping capabilities we will need more hidden layers. Figure 5.17 shows an autoencoder with 3 hidden layers. Note the symmetry in terms of the number of hidden nodes in each layer. This is a very common architecture of the autoencoder, namely an odd number of hidden nodes, the number of nodes in the first hidden layer is the same as the number of hidden nodes in last hidden layer, and so on. We can call this a *butterfly* design.

Figure 5.18 shows an example where a simple 2D dataset is being trained by an autoencoder. Even though the data is two dimensional it could effectively be represented by a single coordinate, since the data nicely follows a curve. Here we cannot use a PCA approach since the curve is not "linear". Instead we will use an autoencoder with three hidden layers. The error function was MSE and training was accomplished using standard SGD. In the left graph the output from the autoencoder is shown as "reconstructed" curve. The size of the autoencoder was $2-2-1-2-2$ with tanh() for the hidden layers. We can see that it is

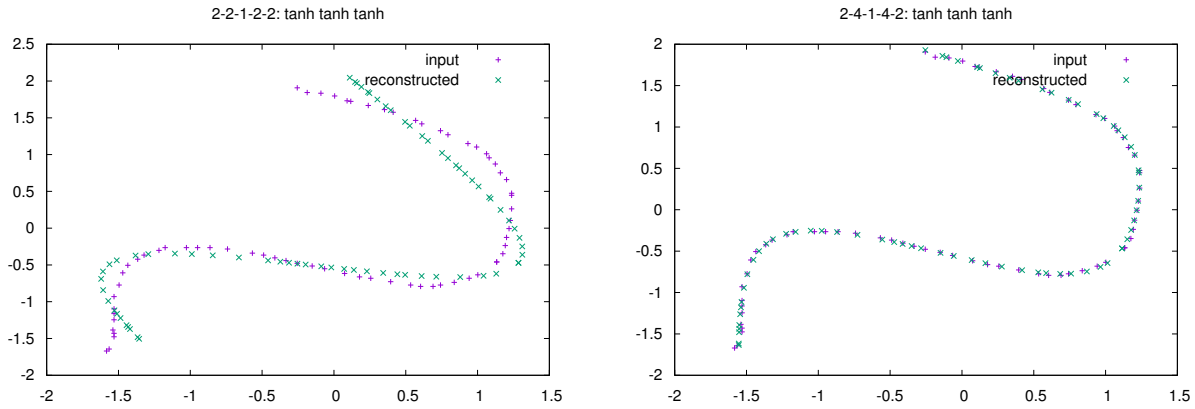


Figure 5.18: Training an autoencoder on a 2D dataset. (left) The input "curve" is feed into an autoencoder of size (2-2-1-2-2) and the output is the curve marked "reconstructed". (right) Here 4 nodes where used in the first and last hidden layer. Now the output is perfectly matching the input.

almost being able to reconstruct the input curve. Note here that the bottleneck layer only has one hidden node. This means that we effectively represent the 2D input by a single value. In the right graph we have an $2-4-1-4-2$ autoencoder. Now the output fits perfect to the input.

In conclusion, we can view the autoencoder as a very efficient tool for finding good representations of data, without the need for labels.

5.3.1 Stacked denoising autoencoder

The stacked denoising autoencoder (SdAE) is modern version (i.e. deep learning version) of the autoencoder described above. It introduces two new concepts, one that you are already familiar with and one that we have not yet talked about. They are,

- Denoising: This simply means that when we train the autoencoder we use the dropout technique on the input layer. The main reason is to avoid overtraining when we have a deep autoencoder with many weights. Also, the denoising of the inputs makes the autoencoder not focus on noise in the input data, rather on general features.
- Pre-training: This we have not used before! It means that we do not train the full network from scratch (i.e. random initialization of the weights). Instead each layer is pre-trained one by one and in the end a fine-tuning is performed using standard methods.

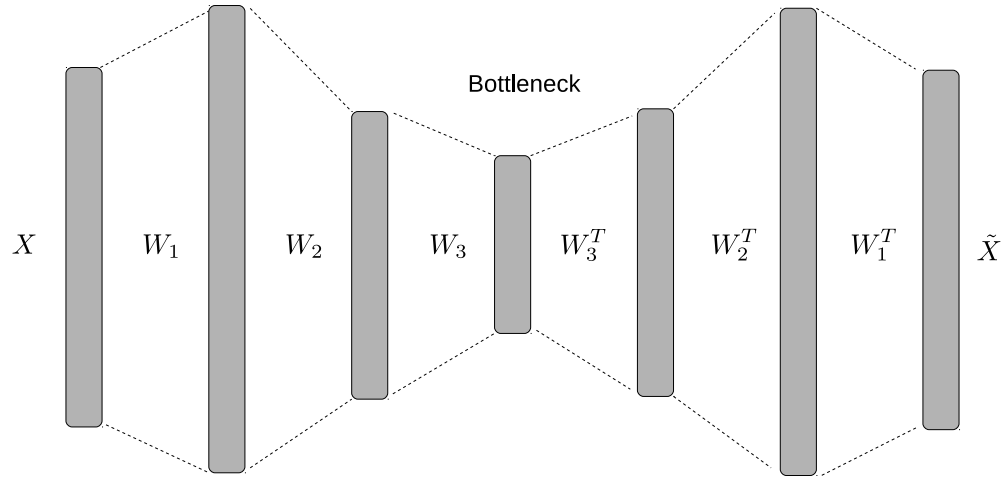


Figure 5.19: The SdE network architecture. The equally sized encoder and decoder part share the bottleneck layer. Tied weights are used indicated by the set of weights (W_1, W_1^T), (W_2, W_2^T) etc. The output layer $\tilde{\mathbf{x}}$ is the autoencoder reconstruction of the input X.

Let's look at the SdAE in a bit more detail. The SdAE autoencoder often follows a butterfly construction with equal sized encoder and decoder parts, see Figure 5.19. Each layer in the encoder performs a mapping from layer input $\mathbf{h}$ to layer output $\mathbf{h}'$ with parameters $\mathbf{W}$ and $\mathbf{b}$,

$$\mathbf{h}' = f(\mathbf{W}\mathbf{h} + \mathbf{b}),$$

where $f(\cdot)$ is the mapping (or activation) function. The weight matrix $\mathbf{W}$ and the bias vector $\mathbf{b}$ are trainable parameters for each layer of the encoder. The autoencoder is also using tied weights, where the decoders weights are the transpose of the encoder weights. For instance if $\mathbf{W}'_L$ denotes the last weight matrix of the decoder, then $\mathbf{W}'_L = \mathbf{W}_1^T$, the transposed of the first weight matrix of the encoder (see Fig. 5.19).

Training the SdAE Training the autoencoder is accomplished in two steps, first pre-training of the individual layers of the autoencoder followed by fine tuning step where all weights and biases are optimized together. The pre-training step is done layer by layer. Let's take the first layer of the SdAE. To get an initial set of weights we train a simple autoencoder with only one hidden layer, as described previously. When training is done we take the first layer of weights of the this training and use that as the first layer of weights for the SdAE. We can now run all data through this first layer and generate a set of output values for the first hidden nodes. This is now the input data to a second simple autoencoder that will be trained to recover this input data. This results in a second simple autoencoder, with weights that will be put on the second layer of the SdAE. And so on! During training dropout is used on the input layer. This denoising is important to avoid identity mapping, especially if the size of the hidden layer is larger than the number of inputs. The amount of

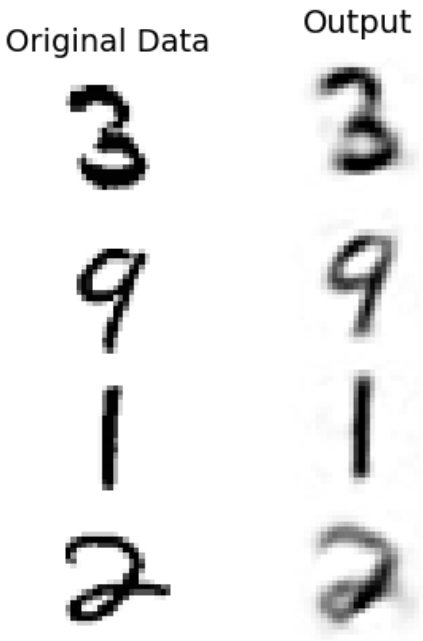


Figure 5.20: An example of original (left) and reconstructed images from the MNIST database, using a SdAE of size $[784] - [1000] - [400] - [20] - [400] - [1000] - [784]$.

denoising used is a tunable hyper-parameter. Furthermore, the pre-training of each layer is also using tied weights of the corresponding simple autoencoder. When the pre-training is finished we have an initial set of weights for the full SdAE. Now training proceeds as usual by minimizing the autoencoder error function (e.g. eqn. 5.1).

Figure 5.20 shows original and reconstructed images of the MNIST database using a SdAE. Each image in the MNIST database is represented as a single numerical vector of size 784. There are 60 000 images in the training dataset. The architecture used in this example was,

$$[784] - [1000] - [400] - [20] - [400] - [1000] - [784]$$

and training was run for 50 epochs with a batch size of 500 images. Note that the bottleneck only contains 20 hidden nodes, meaning that each image is represented by a numerical vector of size 20. The left graph in figure 5.20 shows 4 digits and the right graph shows the corresponding reconstructed images. The reconstruction is not perfect, but the bottleneck is on other hand rather small.

Applications of the SdAE The autoencoder can be used to pre-train deep networks. In fact this was one of the first an important applications of the autoencoder. Other applications areas include imputation of missing data and detection of outliers. For the former a SdAE

can be trained on both complete and incomplete data. At the test situation a missing data can be reconstructed using the SdAE. The idea of using a SdAE to detect outliers is based on finding specific data points that have large reconstruction errors, i.e. data points that do not fit into the trained model represented by the SdAE. Such data points could be potential outlier data points.

5.4 The variational autoencoder

The variational autoencoder (VAE) is a generalization of the autoencoder in terms of its ability to generate data. We call VAE a *generative* model. What do we mean by that? Let's first describe our standard MLP network in terms of a *discriminative* model, meaning that it generates target variables given a set of observed variables. Simply, we have inputs that we feed to the MLP and out comes the target variables. On the other hand the *generative* model generates both inputs and outputs. Most times we do not distinguish between input and output, we simple have a set of variables. Generative models are typically probabilistic, specifying a probability distribution of the variables.

The autoencoders we have talked about so far are not generative models. They are trained on a dataset with the aim of being a good model for this data. But we cannot sample from the trained autoencoder to generate new data. The VAE can do that! The details of the VAE are a bit technical, so here we will just look at the overall idea of this method. Figure 5.21 shows the standard autoencoder design, where an input is being encoded by the encoder part of the network ending up with a representation given by the bottleneck, here called the latent vector. The decoder part is then taking the latent vector and reconstructs the input. The VAE does almost the same thing. However, the "bottleneck" is divided into two vectors called *mean* vector and *standard deviation* vector, see figure 5.22. The mean and the standard deviation vectors are parameters specifying a normal distribution. This

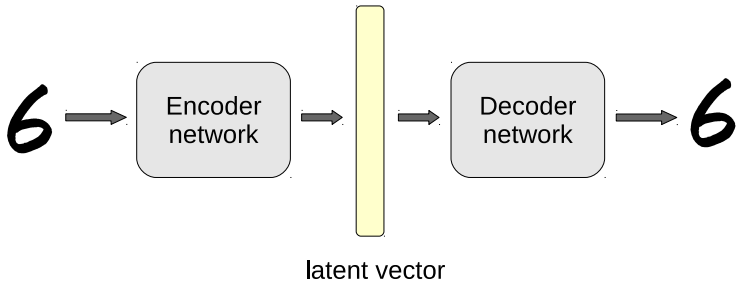


Figure 5.21: The general design of an autoencoder. There is an encoder part, followed by the bottleneck, here called latent vector. The decoder is then taking the latent vector and reconstructs the input.

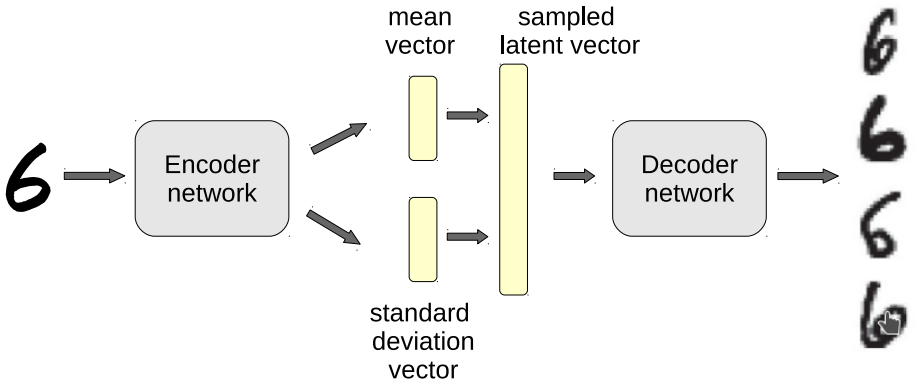


Figure 5.22: The general design of a variational autoencoder. The bottleneck is now divided into two vectors, the mean and the standard deviation vectors. These are specifying a normal distribution that can be sampled from. Each sample is the used as the input to the decoder, the is now *generating* and new output.

normal distribution is then sampled from. Each specific sample provides an input to the decoder network that is now generating an output. This is illustrated in figure 5.22 where the input image of the digit "six" is feed to encoder part of the VAE. This results in fixed mean and standard deviation vectors that in turn defines a normal distribution. Sampling from this normal distribution gives rise to latent vectors that in turn are decoded by the decoder network to finally produce new images.

In this example we had a specific input to the VAE. Can we generate data without feeding an initial input? The training of the VAE is designed to produce mean and standard deviation vectors that are close to zero and one, respectively. If one would average all mean and standard deviation vectors over all of the training data one should obtain value close to zero and one. This means that in order to generate data that are similar to the training data, one simply sample from a normal distribution with zero mean and unit variance. The VAE has become a generative model. How to actually carry out the training and the details of the error function to use is not covered here. It is bit technical!

We can train a VAE on the MNIST database. Figures 5.23 and 5.24 shows the result of such a model. The architecture of this model was,

$$[784] - [400] - [20] - [3 + 3] - [20] - [400] - [784]$$

meaning that the bottleneck consists of a 3-dimensional normal distribution, specified by three values for the mean and three values for the standard deviation. (We can also say that we have three 1-dimensional normal distributions, each with a mean and a standard deviation value.) Figure 5.23 shows images that are generated when leftmost image (digit 8) is fed into the VAE and gives rise to a specific mean and standard deviation vector. Sampling

Figure 5.23: The leftmost image is the input to a VAE trained on the MNIST database. The four images to the right are then generated by the VAE.

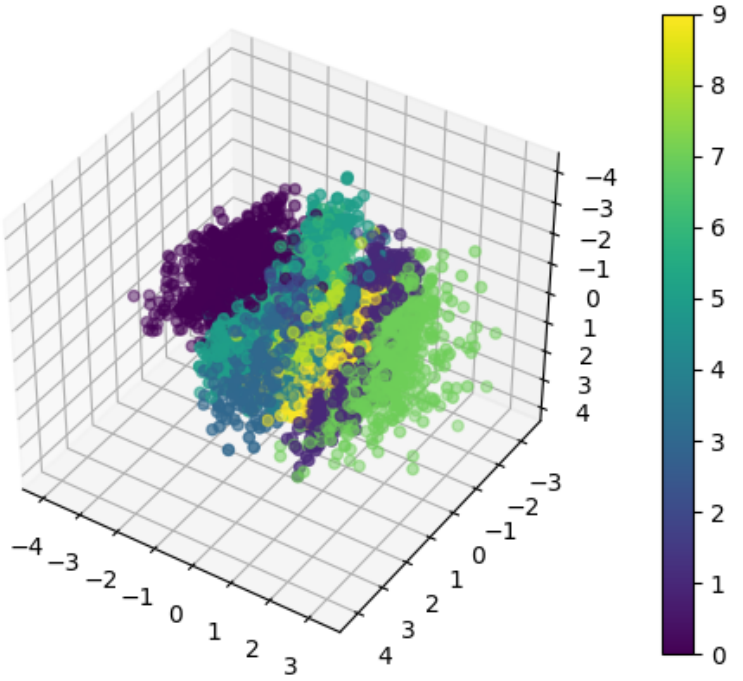


Figure 5.24: A visualization of the latent space for a VAE trained on the MNIST database. Each point in the figure corresponds the the mean vector on the VAE. The color of the points denote the digit of the input image, as explained by the colorbar to the right.

from the corresponding normal distributions generates the 4 images to the right. Again, note that we now have a really small bottleneck. Since we only have three mean values for each image we can actually plot them in a 3D plot. Figure 5.24 shows a subset of the images from the test MNIST data. The colors of the different points represent the 10 digits, as shown by the colorbar to the right. We can see how the different digits separates in the plot, even though there is overlap between the "classes".

5.5 Generative adversarial networks

The Generative adversarial network (GAN) also belong to the category of generative models. It was invented 2014 by Goodfellow et al. Just to continue the discussion on generative models. Creating a generative model means here using a training dataset, that contains samples from some unknown distribution p_d, and being able to come up with an estimate p_m of this unknown distribution. We can do this by explicitly stating how p_m looks like or we can provide a model that can generate data from p_m. The main usage of the GAN is to be able to generate samples.

The GAN actually consists of two networks, called the *generator* network and the discriminator network. The purpose of the generator network is to create samples that looks like they are coming from the training dataset. The discriminator network is trying to separate generated samples from "real" samples, i.e. it is just a binary classification network. Figure 5.25 shows an illustration of the design of the GAN model. One can think of this in terms of playing a game. The generator is trying fool the discriminator by making as good "fake" samples as possible, while the discriminator network is trying to becomes as good as possible to tell whether it is a fake sample or not. Both the generator and the discriminator can be simple MLPs, think of the generator as the decoder part of an autoencoder and the discriminator as being an ordinary MLP for binary classification. However, GANs are mostly used in connection with images and then both the generator and the discriminator are CNN type

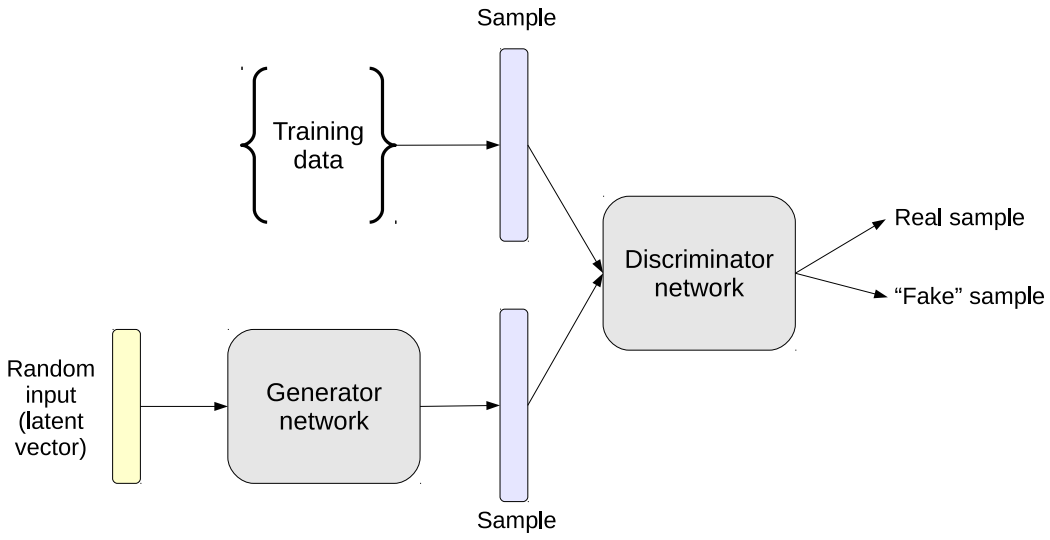


Figure 5.25: An illustration of the GAN model. Using a random input vector (latent vector) the generator is producing a sample with the aim of being similar to samples coming from the training dataset. The discriminator network on the other hand is taking both real and generated samples as inputs with the task of separating them from each other.

of networks.

Training the GAN So how does one accomplish to play this game, i.e. how do we train the networks? Let's look at the original problem formulation. Training the GAN is the following max-min problem,

$$\min_{\text{G}} \max_{\text{D}} V(D, G)$$

where V is defined as follows,

$$V(D, G) = \mathbb{E}_{x \sim p_{data}(x)}[\log D(x)] + \mathbb{E}_{z \sim p_z(z)}[\log(1 - D(G(z))] \tag{5.2}$$

Here $D(x)$ represents the discriminator and $G(z)$ the generator. The output of discriminator network $D(x)$ is the probability that x is a real sample. The random variable z is drawn from the distribution $p_z(z)$ that generates inputs to the generator network. And x are samples drawn from the $p_{data}(x)$, represented by our training dataset. If we look at the max of V with respect to D we see that it is our usual cross entropy "error", but now being maximized since it is positive. For the first term all data comes from the training dataset and maximizing $\log D(x)$ means that $D(x)$ should be one. For the second term all data are coming from the generator network, hence $D(G(z))$ should be zero, i.e. maximizing $\log(1 - D(G(z)))$. Now, the min of V with respect to G does the opposite. It wants to make generated samples look like the real samples, meaning that the discriminator should classify a generated sample as real, i.e. minimize,

$$\mathbb{E}_{z \sim p_z(z)}[\log(1 - D(G(z))]$$

Note that the first term of eqn. 5.2 is independent of G and is therefore just a constant with respect to the minimization of V. It turns out that training a GAN model is not a trivial process. Both the generator and the discriminator needs to improve in a similar pace, otherwise one can for example end up in a local minimum where the discriminator is perfect and the generator looses its generative ability and just outputs some fixed average sample.

Some example of GAN models There is a lot of research going on at the moment concerning GAN models. How to improve the training procedure and how to evaluate them are such topics. Below are a few titles where GAN models are being used in applications:

- Generative Adversarial Text to Image Synthesis
- Image-to-Image Translation with Conditional Adversarial Networks
- Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network

5.5.1 Cycle GAN

There are many extensions to the original GAN model presented above. The GAN Zoo contains a list of some of these extensions and modifications. We will look at one extension called *Cycle GAN*. This model is suitable for image-to-image translation. Suppose we want to transfer a landscape image taken a summer day into the same landscape, but now on a winter day. We can imagine that this problem could be solved using a large set of paired images, e.g. a lot of landscapes taken both on a summer day and a winter day. Then we could train a CNN to do the mapping of summer to winter landscape. The main limitation with this approach comes from the requirement of having a paired dataset. This can be difficult, and in some cases even impossible, to obtain. The cycle GAN model do not have that requirement and can work with a set of unpaired images. For the cycle GAN we can use a set of summer landscapes images and a unrelated set om winter landscape images. Figure 5.26 shows an example of paired and unpaired image sets.

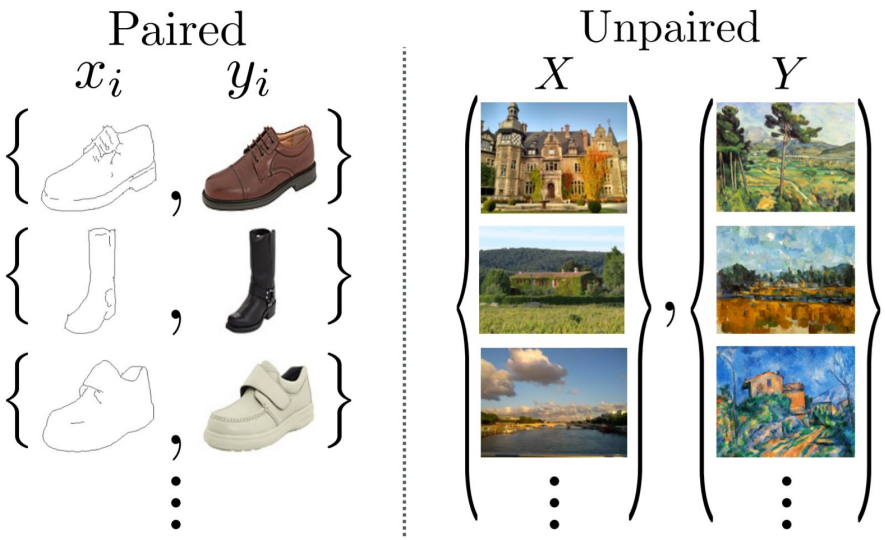


Figure 5.26: (Left) A set of paired images where there exists a correspondence between each pair of images. (Right) A set of unpaired images, where no correspondence exists. (This figure is taken from the original cycle GAN paper.)

Assume we have training data from the domain X and one set of training data from the domain Y. The cycle GAN uses two generators $G : X \to Y$ that convert images from domain X to domain Y, and $F : Y \to X$ that converts from Y to X. We also have two discriminators D_X and D_Y where D_X discriminates x from $F(y)$ and D_Y distinguishes y from $G(x)$. The process is illustrated in Figure 5.27. There is an additional image generation process for the

cycle GAN, each generated image is also fed into the reverse generator as to "cycle" back to the original image.

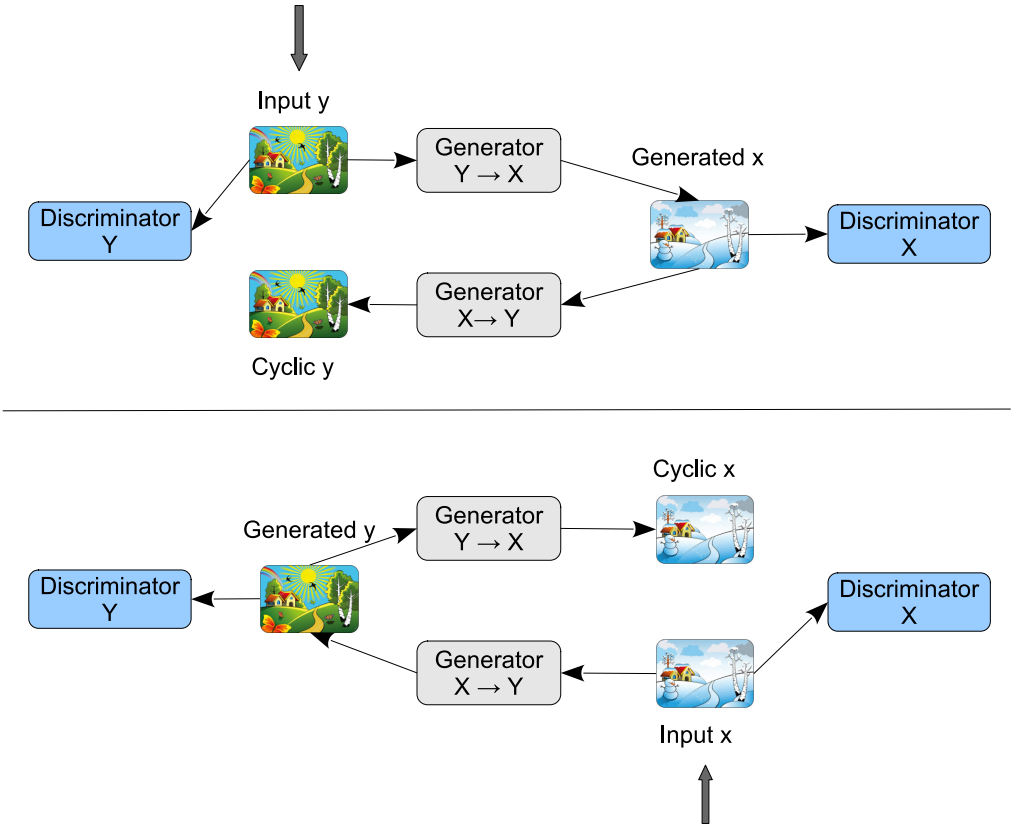


Figure 5.27: An illustration of the Cycle GAN model. (Top) An input y (summer landscape) is translated into an x (winter landscape) and then cycled back into the original image. Discriminator D_Y is taking the y image and separates that one from the generated y image (from bottom). (Bottom) An input x (winter landscape) is translated into an y (summer landscape) and then cycled back into the original image. Discriminator D_X is taking the x image and separates that one from the generated x image (from top).

The objective function for the cycle GAN consists of two parts. The first part is the same as the objective for the original GAN (*adversarial loss*), but we need one for each pair of generator and discriminator. For the G mapping and its discriminator D_Y we have the objective,

$$V_{\text{GAN}}(G, D_Y) = \mathbb{E}_{y \sim p_{data}(y)}[\log D_Y(y)] + \mathbb{E}_{x \sim p_{data}(x)}[\log(1 - D_Y(G(x))] \tag{5.3}$$

where G tries to generate images $G(x)$ that look similar to images from domain Y, while D_Y will try to distinguish between translated samples $G(x)$ and real samples y. We have

similar objective for the F mapping and D_X,

$$V_{\text{GAN}}(F, D_X) = \mathbb{E}_{x \sim p_{data}(x)}[\log D_X(x)] + \mathbb{E}_{y \sim p_{data}(y)}[\log(1 - D_X(F(y))] \tag{5.4}$$

Training the cycle GAN would then translate into the following max-min problem,

$$\min_{\text{G,F}} \max_{D_y, D_x} V_{\text{GAN}}(G, D_Y) + V_{\text{GAN}}(F, D_X)$$

There is however a problem with this approach. There is nothing that "forces" the translated images to look the same. A winter landscape of a lake could easily be translated to a summer landscape of a forest. To address this problem the authors introduced the *cycle consistency* loss. This loss ensures that translating a image from X to Y and then translating it back to X again should result in the same starting image, i.e. $F(G(x)) \approx x$ and $G(F(y)) \approx y$. The suggested loss function was,

$$V_{\text{cyc}}(G, F) = \mathbb{E}_{x \sim p_{data}(x)}[||F(G(x)) - x||_1] + \mathbb{E}_{y \sim p_{data}(y)}[||G(F(y)) - y||_1] \tag{5.5}$$

The final objective function is a combination of adversarial losses (Eqn. 5.3 and 5.4) and the cycle consistency loss (Eqn. 5.5),

$$V(G, F, D_X, D_Y) = V_{\text{GAN}}(G, D_Y) + V_{\text{GAN}}(F, D_X) + \lambda V_{\text{cyc}}(G, F) \tag{5.6}$$

In their orginial paper λ was set to 10. The training then results in two generators G^* and F^* as found by,

$$G^*, F^* = \min_{\text{G,F}} \max_{D_y, D_x} V(G, F, D_Y, D_X)$$

Here are a couple of examples from the original cycle GAN paper.

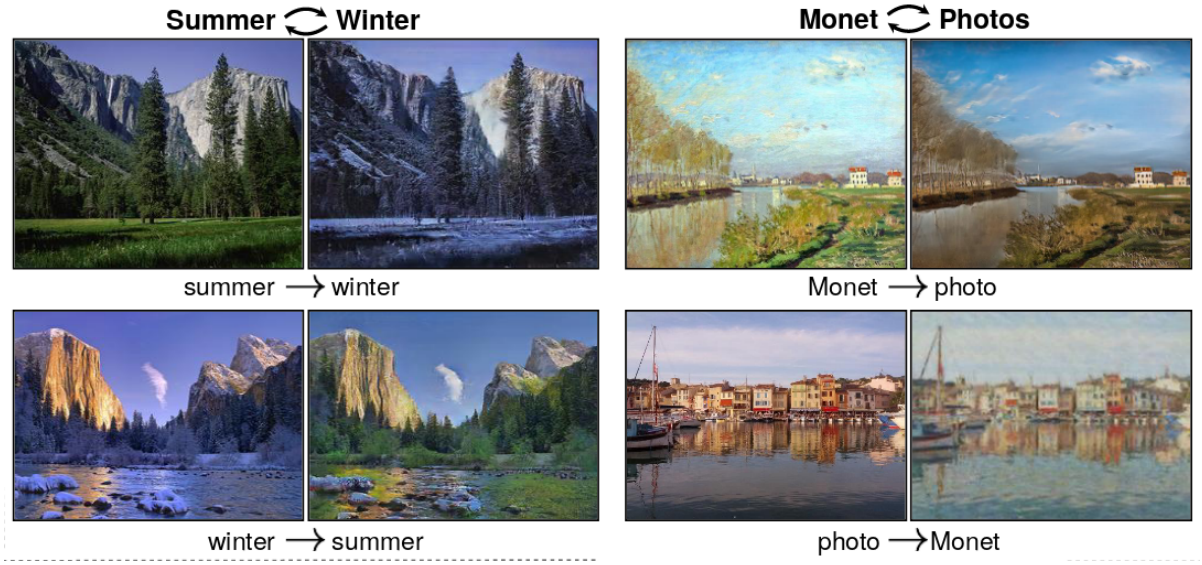


Figure 5.28: Translation of summer and winter images and taking a photo and turning it into a Monet painting.

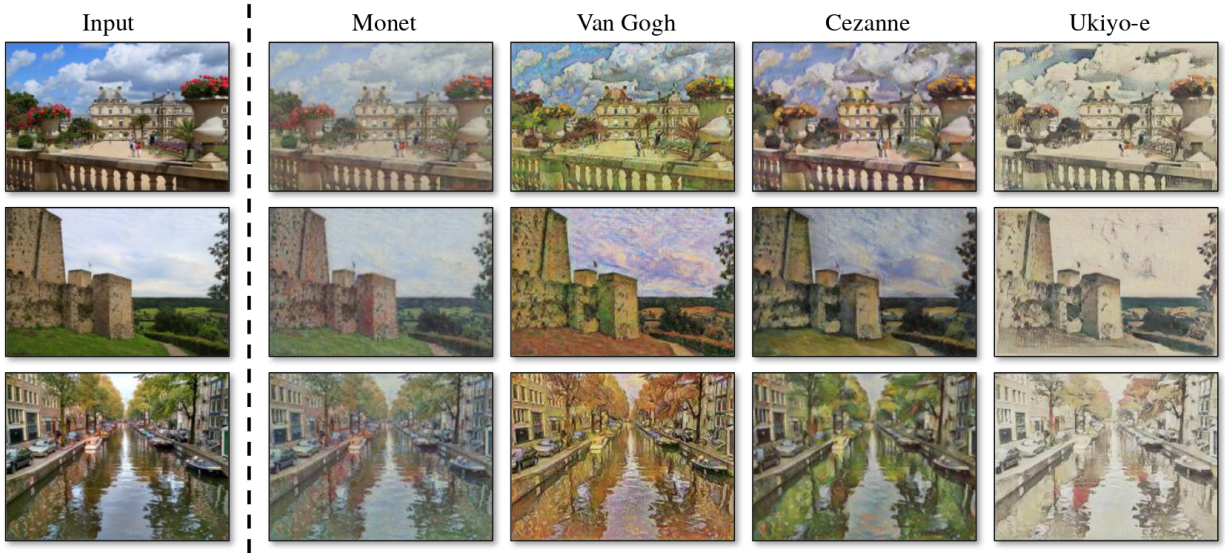


Figure 5.29: More of images to paintings.