DAT300 CHAPTER 15 – Classyfying Images with Deep Convolutional Neural Networks

Jorid Holmen
Applied Robotics, Norwegian University of Life Sciences

This chapter is about **convolutional neural networks (CNNs)**, which is used for image and pettern recognition (image classification). In the beginning we will talk about the basic building blocks of CNN, using a bottom-up approach. Then we will take a deeper dive into the architecture and explore how to implement CNNs in TensorFlow.

Contents

1 The building blocks of CNNs

CNNs were originally inspired by how the visual cortex of the human brain works when recognizing objects.

In the following sections we will discuss why convolutional architectures are often described as "feature extraction layers", and then delve deeper into the theoretical definitions of the type of convolution operation that is commonly used in CNNs and walk through examples of computing convolutions in one and two dimensions.

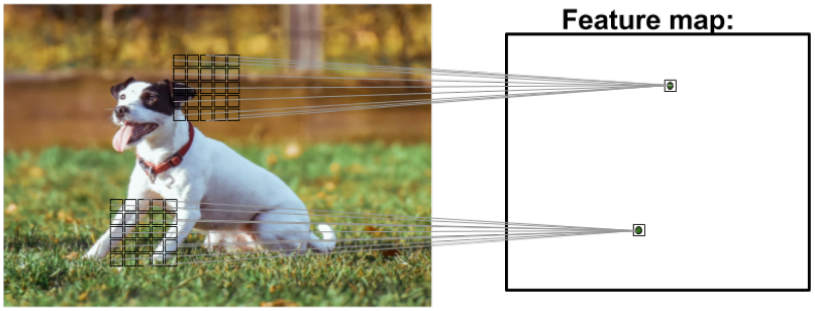
1.1 Understanding CNNs and feature hierarchies

Succesfully extracting **important or relevant features** is key to the performance of any machine learning algorithm and traditional machine learning models. This is especially true for models that depend on input features provided by domain experts or generated through computational feature extraction methods. CNNs are able to automatically learn the features from raw data that are most useful for a particular task. Therefore it is common to view CNN layers as feature extraction:

- The early layers (those right after the input layers) extract **low-level features** from raw data. Also called **low-level features**. The low level features are for exampe edges and blobs.
- The later layers (often **fully connencted layers** like in a multilayer perceptron (MLP)) use these features to predict a contionuous target value or class label. Also called **high-level features**. High level features are for example the general contour of a building.

Certain types of multilayer NNs, and in particual, deep convolutional NNs, construct a so-called **feature hierarchy** by combining the low-level features in a layer wise fashion to form high-level features.

The CNN computes feature maps from an input image, where each element in the map comes from a local path of pixels in the input image. The local path of pixels is referred to as the **local receptive field**.



CNNs usually perform very well on image-related tasks, largely due to:

- **Sparse connectivity**: a single element in the feature map is connected to only a small patch of pixels. This is different from i.e. perceptrons where the whole input image is connected.
- **Parameter-sharing**: the same weights are used for different patches of the input image.

As a direct consequence of these two ideas, replacing a conventional, fully connected MLP with a convolution layer substantially decreases the number of weights/parameters in the network and we will see an improvement in the ability to capture *salient (relevent) features*. In the context of the image data, it makes sense to assume that nearby pixels are typically more relevant to each other than pixels that are far away from each other.

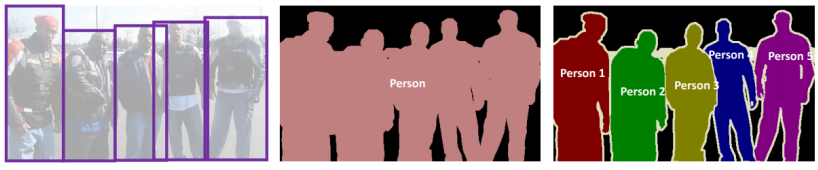
Typically, CNNs are composed of several **convolutional** and **subsampling** layers that are followed by one or more fully connected layers at the end. The fully connected layers are essentually an MLP, where every input unit, i, is connected to every output unit, j, with weight w_{ij}.

Both the convolutional and the fully connencted layers have weights and biases that are optimized during training. However, the subsampling layers, also known as pooling layers which will be explained more later, do not have any learnable paramters.

1.1.1 Image classification tasks

CNN's are useful for many tasks, including these:

- Classification and localization: Identify the main object in the image and calculate a bounding box.
- Object detection: Detection of multiple objects in an image.
- Semantic segmentation: Assign a class to every pixel in an image, identity of objects is disregarded. Also referres to as dense prediciton. Labeling is pixel-wise istead of image-wise.



Object Detection Semantic Segmentation **Instance Segmentation**

1.2 Performing discrete convolutions

A **discrete convolution** (or simply convolution) is a fundemental operation in CNN. Therefore, it is important to understand how this operation works. This section will cover the mathematical definitions and some of the **naive** algorithms to compute convolutions of one-dimensional tensors (vectors) and two-dimensional tensors (matrices).

1.2.1 Discrete convolutions in one dimension

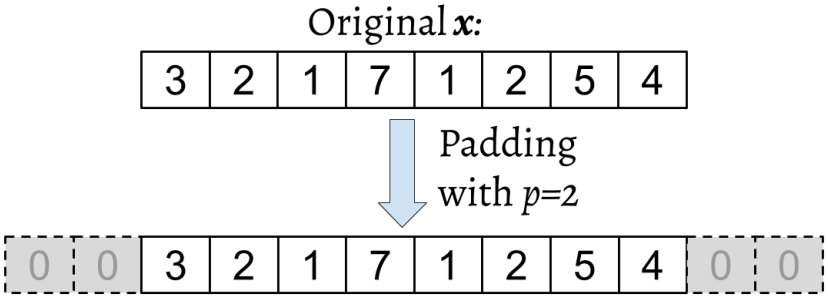
To understand convolutional operations it is best to start with convolution in one dimension, which is sometimes used for working with certain types of sequence data, such as text. After that, we can start with two-dimensional, which is often used for two-dimensional images.

A discrete convolution for two vectors, $\mathbf{x}$ and $\mathbf{w}$ is denoted by $\mathbf{y} = \mathbf{x} * \mathbf{w}$, in which vector $\mathbf{x}$ is input (sometimes called **signal**) and $\mathbf{w}$ is called the **filter** or **kernel** (filter consists of weights). *Not confuse the convolutional operator, *, with dot product, etc.* A discrete convolution is mathematicaly defined as follows:

$$\mathbf{y} = \mathbf{x} * \mathbf{w} \longrightarrow y[i] = \sum_{k=-\infty}^{+\infty} x[i-k]w[k] \tag{1}$$

Each element in the output is the product of a subset of the input and each filter coefficient. The brackets [] are used to denote the indexing for the vector elements. The index, i, runs through each element of the output vector, $\mathbf{y}$.

The sum runs through indices from $-\infty$ to ∞. To correctly compute the summation, it is assumed that $\mathbf{x}$ and $\mathbf{w}$ are filled with zeros. This will result in $\mathbf{y}$ also having an infite size. This is not pratical, so $\mathbf{x}$ is only padded with a finite number, p, on each side. This is called **zero-padding**, or simply **padding**.



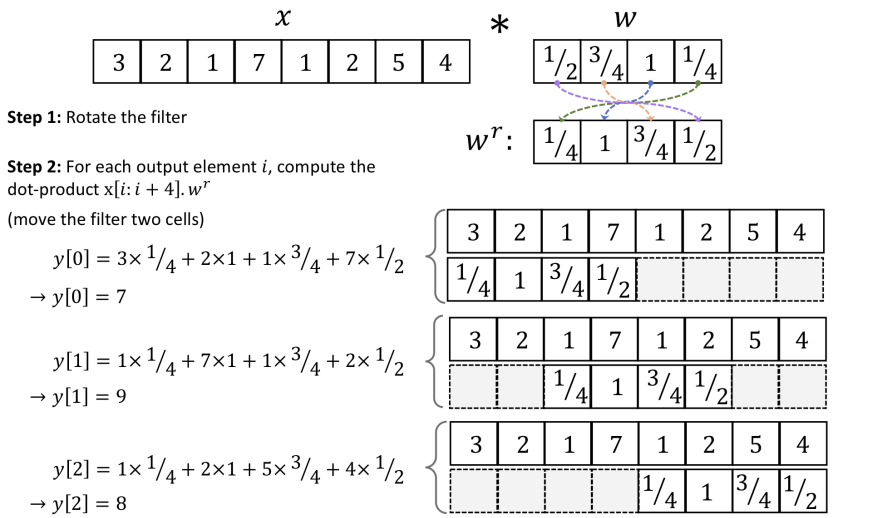
If the original input $\mathbf{x}$ and the vector $\mathbf{w}$ have n and m elements, where $m \leq n$, and the padded vector $\mathbf{x}^p$ has size $n + 2p$, we get:

$$\mathbf{y} = \mathbf{x} * \mathbf{w} \longrightarrow y[i] = \sum_{k=0}^{k=m-1} x^p[i+m-k]w[k], \tag{2}$$

where p is the amount of padding.

Another issue is indexing $\mathbf{x}$ with $i+m-k$. $\mathbf{x}$ and $\mathbf{w}$ are indexed in different directions. Computing the sum with one index going in the reverse direction is equivalent to computing the sum with both indices in the forward direction after flipping one of those vectors, $\mathbf{x}$ or $\mathbf{w}$, after they are padded. Let's assume we flip (rotate) the filter, $\mathbf{w}$, to get the rotated filter $\mathbf{w}^r$. Then, we can simply compute the dot product:

$$y[i] = x[i : i+m].w^r \tag{3}$$



The rotated filter $\mathbf{w}^r$, is shifted with two cells each time we **shift**. The shift is another hyperparameter of a convolution called **stride**. The stride has to be a positive number smaller than the size of the input vector.

In this image the input is $n = 8$, the filter size is $m = 4$, the padding is $p = 0$ and the stride is $s = 2$.

1.2.2 Padding inputs to control the size of the output feature maps

So far we have only used zero-padding in convolutions to compute finite-sized output vectors. Techincally padding can be applied with any positive number. Depending on the choice of p, boundary cells may be treated differently than the cells located in the middle of $\mathbf{x}$.

Consider an example where $n = 5$, $m = 3$ and $p = 0$. Then $x[0]$ is only used in computing one output element, like $y[0]$. While $x[1]$ is used is used in two output elements, like $y[0]$ and $y[1]$. In this way the elements in the middle will have more say in the outputs. This can be avoided with padding.

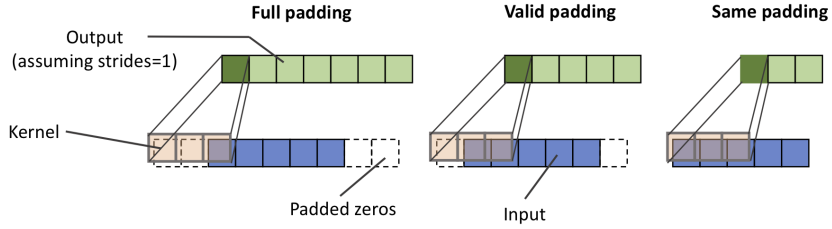
The size of the output $\mathbf{y}$ depends on the choice of the padding strategy we use. There are three modes of padding that are commonly used in practice:

- *Full*: the padding parameter, p, is set to $p = m$ - 1. Full padding increases the dimensions of the output, therefore it is rarely used in CNN architectures. The size results in an output larger than the input. Usually used for signal processing applications where it is important to minimize boundary effects.

- *Same*: usually used to ensure that the output vector has the same size as the input vector, $\mathbf{x}$. In this case, the padding parameter, p, is computed according to the filter size, along with the requirement that the input size and output size are the same. The most commonly used, and also the reccomended one. Preserves the height and width of input images and decreases the spatial size via pooling layers instead.

- *Valid*: $p = 0$ – no padding. A disadvantage is that the volume of tensors will decrease substantially in NNs with many layers, which can be detrimental to the network performance.

Typically full padding and same padding is recommended, possibly together with pooling for downsampling.



1.2.3 Determining the size of the convolution output

The output size of a convolution is determined by the total number of times that we shift the filter, $\mathbf{w}$, along the input vector. If we have input vector, $\mathbf{x}$, of size n and filter, $\mathbf{w}$, of size m, then the output size, o, resulting from $\mathbf{y} = \mathbf{x} * \mathbf{w}$ with padding, p, and stride, s, will be:

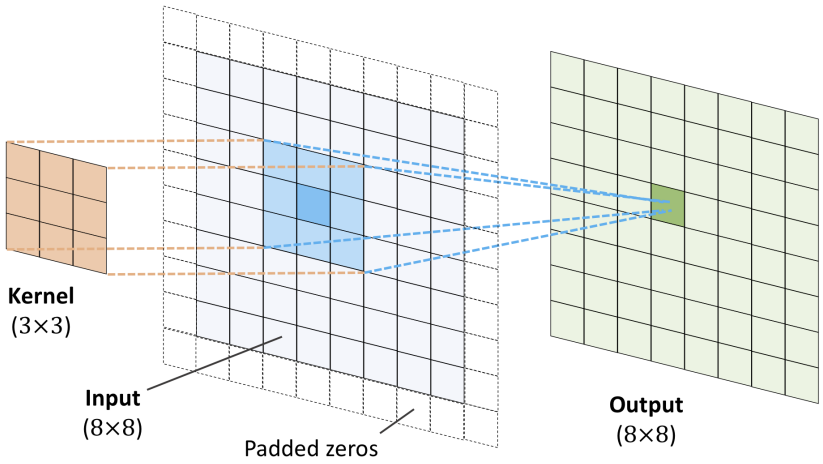
$$o = \left[\frac{n+2p-m}{s}\right] + 1 \tag{4}$$

1.2.4 Performing a discrete convolution in 2D

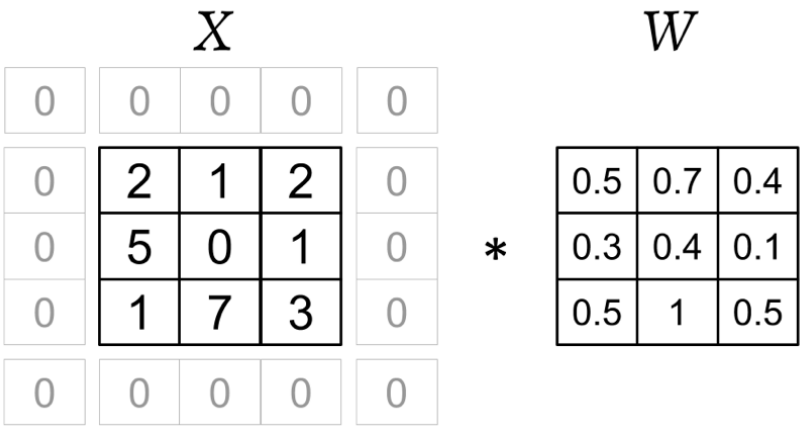
The concepts from previous sections are easily extendible to 2D. When we deal with 2D inputs such as matrix $\mathbf{X}_{n_1 x n_2}$ and filter matrix $\mathbf{W}_{m_1 x m_2}$, where $m_1 \leq n_1$ and $m_2 \leq n_2$. Then the matrix $\mathbf{Y} = \mathbf{X} * \mathbf{W}$ is the result of a 2D convolution between X and Y:

$$\mathbf{Y} = \mathbf{X} * \mathbf{W} \longrightarrow Y[i,j] = \sum_{k_1=-\infty}^{+\infty} \sum_{k_2=-\infty}^{+\infty} X[i-k_1, j-k_1] w[k_1, k_2] \tag{5}$$

Zero-padding, rotating the filter matrix, the use of stride, etc. are applicable to 2D convolutions, provided they are extended to both dimensions independently. The following figure demonstrates 2D convolution of an input matrix of 8 X 8, using kernel of size 3 X 3. The input is padded with zeros of $p = 1$. The result will be 8 X 8.



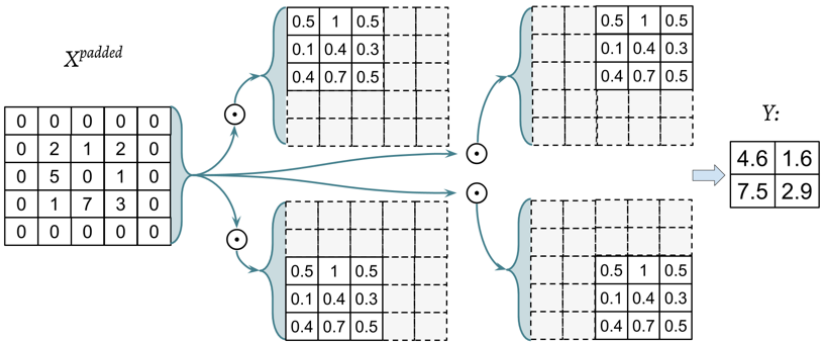
The following example illustrates the computation of 2D convolution between an input matrix $\mathbf{X}_{3x3}$ and a kernel matrix $\mathbf{W}_{3x3}$, using padded $p = (1, 1)$ and strise $s = (2, 2)$. According to the specified padding, one layer of zeros is added on each side of the input matrix, which results in a padded matrix X_{5x5}^{padded}, as follows:



The rotated filter will be

$$W^r = \begin{bmatrix} 0.5 & 1 & 0.5 \\ 0.1 & 0.4 & 0.3 \\ 0.4 & 0.7 & 0.5 \end{bmatrix} \tag{6}$$

Note that this rotation is not the same as the transpose matrix. Now, the rotated filter matrix can be shifted along the padded input matrix, like a sliding window and compute the sum of the element-wise product:

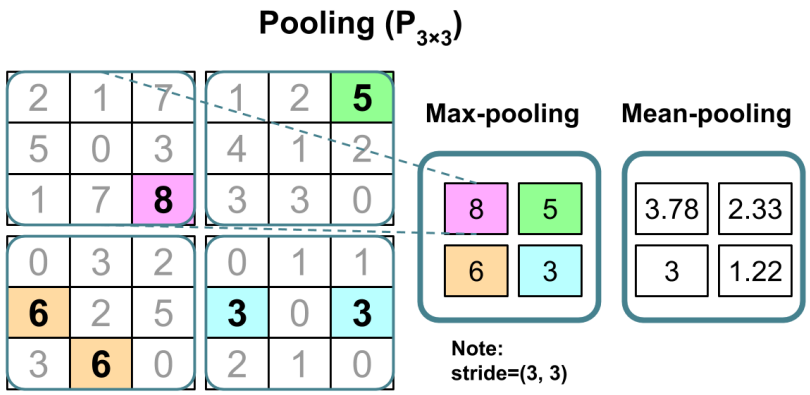


1.3 Subsampling layers

Subsampling is typically applied in two forms of pooling operations in CNNs: **max–pooling** and **mean–pooling** (also known as **average–pooling**). Pooling helps reduce the spatial dimension and introduces robustness to small changes due to noise or other minor variations. It works as a dimensional reduction for higher computational efficiency and reduced overfitting. Usually the pooling-size is the same as the stride-size, so there is no overlapping.

The pooling layer is usually denoted by $P_{n_1 x n_2}$. The subscript determines the size of the neighborhood (the number of adjacent pixels in each dimension) where the max or mean operations is performed. This neighborhood is called **pooling size**.

The operation is described in the following figure. Here, max-pooling takes the maximum value from a neighborhood of pixels, and mean-pooling computes their average:



The advantage of pooling is:

- Max-pooling introduces a local variance. This means that small changes in a local neighborhood do not change the result of max-pooling. Therefore, it helps generating features that are more robust to noise in the input data.

- Pooling decreases the size of the features, which results in higher computational efficiency. It may reduce the degree of overfitting as well.

Typically, pooling is said to be non-overlapping, which is done by setting the stride equal to the pooling size. However, overlapping pooling can occur.

Many researchers also use convolutional layers with a stride of 2 to reduce the feature size, instead of pooling. You can think of it as a pooling layer with learnable weights.

1.4 Transposed convolutions

Transposed convolutions play a crucial role in various aspects of convolutional neural networks. While standard convolutions are used for detecting features or patterns with a specified stride, transposed convolutions serve additional purposes:

- Detecting Features/Patterns (Stride = 1): Similar to ordinary convolutions, transposed convolutions are effective for detecting features and patterns when the stride is set to 1.

- Down-sampling, Reducing Resolution (Stride > 1): Transposed convolutions can be utilized for down-sampling, contributing to the reduction of resolution when the stride is greater than 1.

- Up-sampling, Increasing Resolution (Transposed Convolutions or Fractional Convolutions): One of the key roles of transposed convolutions is up-sampling, which involves increasing resolution. This is particularly useful in tasks such as image generation and semantic segmentation.

1.5 Working with multiple input or color channels

An input to a convolutional layer may contain one or more 2D arrays or matrices with dimensions $N_1 x N_2$. These $N_1 x N_2$ are called channels. Conventional implementations of convolutional layers expect a rank-3 tensor representation as an input, for example a three dimensional array, $\mathbf{x}_{N_1 x N_2 x c_{in}}$, where c_{in} is the number of input channels. For an RGB image we would have $c_{in} = 3$ for red, green and blue color channels. Gray images gives $c_{in} = 1$.

To incorporate multiple input channels in the convolution operation, we perform the convolution operation for each channel seperately and then add the results together using the matrix summation. The convolution associated with each channel (c) has its own kernel matrix as W[:,:,c].

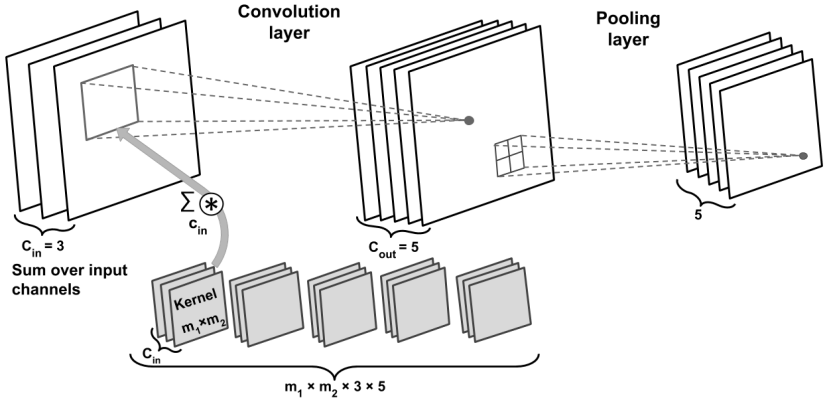
The total pre-activation result is computed in the following formulas, given an example $X_{n_1 \mathrm{X} n_2 \mathrm{X} c_{in}}$, a kernel matrix $W_{m_1 \mathrm{X} m_2 \mathrm{X} c_{in}}$ and a bias value b:

$$\begin{gathered} Z^{Conv} = \sum_{c=1}^{C_{in}} W[:,:,c] * X(:,:,c) \\ \text{Pre-activation: } Z = Z^{Conv} + b_c \\ \text{Feature map: } A = \Phi(Z) \end{gathered} \tag{7}$$

The finale result, A, is a feature map. Usually, a convolutional layer of CNN has more than one feature map. If we use multiple feature maps, the kernel tensor becomes four-dimensional: *width* x *height* x c_{in} x c_{out}. *width* x *height* is the kernel size, c_{in} is the number of input channels, C_{out} is the number of feature maps. Given an example $X_{n_1 \mathrm{X} n_2 \mathrm{X} c_{in}}$, a kernel matrix $W_{m_1 \mathrm{X} m_2 \mathrm{X} c_{in} \mathrm{X} C_{out}}$ and a bias vector $b_{c_{out}}$, we update the preceeding formulas:

$$\begin{gathered} Z^{Conv}[:,:,k] = \sum_{c=1}^{C_{in}} W[:,:,c,k] * X[:,:,c,k] \\ \text{Pre-activation: } Z[:,:,k] = Z^{Conv}[:,:,k] + b[k] \\ \text{Feature map: } A[:,:,k] = \Phi(Z[:,:,k]) \end{gathered} \tag{8}$$

In the following image you can see a convolutional layer, followed by a pooling layer. There are three input channels, and the kernel tensor is four-dimensional. Each kernel tensor is denoted as $m_1 x m_2$, and there are three of them, one for each input channel. Furthermore, there are five such kernels, accounting for five output feature maps. Finally, there is a pooling layer subsampling the feature maps.



The convolutional operations are carried out by treating an input image with multiple color channels as a stack of matrices; that is, we perform the convolution on each matrix seperatly and then add the result. The convolutions can also be extended to 3D.

2 Putting everything together – implementing CNN

The most important operation in traditional NN is matrix multiplication, like $z = Wx + b$, where x is a column vector and W is a weight matrix. In CNN, this is replaced by a convolution operation, as in $Z = W * X + b$, where X is a matrix representing the pixels in a *height x width* arrangement.

In both cases, the pre-activations are passed on to an acivation function to obtain the activation, $A = \phi(Z)$, of a hidden unit. Furthermore you will recall that subsampling is another building block of CNN, which appear in the form of pooling.

2.1 Preprocessing

Preprocessing in Convolutional Neural Networks (CNNs) involves preparing the input data before feeding it into the network for training. This step is crucial for ensuring that the network can effectively learn from the data and make accurate predictions. Here are some common preprocessing techniques used in CNNs:

- **Normalization**: Normalizing the input data helps in bringing all features to a similar scale, which can speed up convergence during training and prevent the dominance of one feature over others. Common normalization techniques include scaling the pixel values to a range between 0 and 1 or standardizing them to have a mean of 0 and a standard deviation of 1.
- **Resizing**: Images in a dataset may have different sizes, but CNNs typically require inputs of fixed dimensions. Resizing involves adjusting the dimensions of all images in the dataset to a consistent size, usually by interpolation.
- **Data Augmentation**: Data augmentation techniques are used to artificially increase the size and diversity of the training dataset by applying transformations such as rotations, flips, shifts, random crops and zooms to the input data. The goal is to create new, varied examples from the original data. This helps in improving the generalization capability of the model and reducing overfitting. Also adjust the brightness, contrast, saturation and blur.
- **Noise Reduction**: In some cases, the input data may contain noise or artifacts that could negatively impact the performance of the CNN. Preprocessing techniques such as denoising filters or image enhancement algorithms can be used to remove or reduce noise from the input data.

In addition to these, one must also think of the classical machine learning preprocessing steps, like cleaning up missing values and outliers.

2.1.1 Adversial attacks

Adversial attacks are images where there is added carefully constructed noise to trick the deep learning model into classifying it as something else. Sometimes the noise is not even visible to the human eye, but still highly confuses the model.

To prevent this there are two options, but they are not fool proof. There first is adversial training, where you generate a number of adversarial examples against your own network, and then explicitly train the model to not be fooled by them. This improves the generalization of the model but has not been able to provide a meaningful level of robustness — in fact, it just ends up being a game of whack-a-mole where attackers and defenders are just trying to one-up each other.

In defensive distillation, we train a secondary model whose surface is smoothed in the directions an attacker will typically try to exploit, making it difficult for them to discover adversarial input tweaks that lead to incorrect categorization. The reason it works is that unlike the first model, the second model is trained on the primary model's "soft" probability outputs, rather than the hard true labels from the real training data. This technique was shown to have some success defending initial variants of adversarial attacks but has been beaten by more recent ones, like the Carlini-Wagner attack, which is the current benchmark for evaluating the robustness of a neural network against adversarial attacks.

2.2 Initialization of weights

The initialization of weights is a crucial aspect in the training of neural networks, often influencing convergence and overall performance. There are several considerations, which contribute to the effective initialization of weights in neural networks, optimizing their learning process and overall performance.

- **Truncated Normal Distributions:** Variations of truncated normal distributions are commonly employed. Values exceeding $> \pm 2$ standard deviations are redrawn to avoid extreme initializations.
- **Default Initializer for Dense and Conv2D:** In the case of Dense and Conv2D layers, the Glorot normal initializer is used by default. It employs a truncated normal distribution with a standard deviation calculated as $\sqrt{\frac{2}{\text{fan\_in}+\text{fan\_out}}}$, where fan_in represents the number of input units, and fan_out represents the number of output units in the weight tensor.
- **Biases Initialization:** Biases are typically initialized with zeros.
- **Matching Feature Scales:** It's important for feature scales to align with weight initializations and any potential weight restrictions. For instance, uint8 values are often rescaled by $\frac{1}{255.0}$.

2.3 Regularizing an NN

Choosing the size of a network, whether we are dealing with a traditional NN or CNN, has always been a challenging problem. A simple network without any hidden layers could only capture a linear decision boundary, which is not sufficient for dealing with an exclusive or similar problem.

The capacity of a network refers to the level of complexity of the function that it can learn to approximate. Small networks have low capacity and are therefore likely to underfit, and large networks may overfit. When dealing with real-world machine learning we do not know how large the network should be.

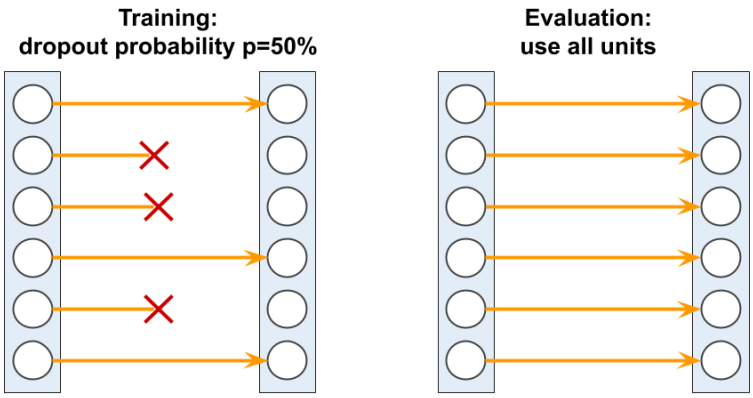
A way to deal with this problem is to build a network with a relatively large capacity to do well on the training dataset. Then, to prevent overfitting, we can apply one or multiple regularization schemes, like L1 or L2, to achieve a good generalization performance on new data. Other methods are norm constrain on weights and adding gaussian noise to weights.

2.3.1 Dropout

Dropout is also a regularization technique. It is often applied to hidden units of higher layers and works as follows: During the training of an NN, a fraction of the hidden units are randomly dropped at every iteration with the user defined probability p_{drop}. The common choice is $p_{drop} = 0.5$. The weights and the activation function associated with the remaining neurons are rescaled to account for the missing/dropped neurons. Dropout is only applied during the training phase, not during inference.

The effect of this random dropout is that the network is forced to learn a redundant representation of the data, which means it learns more general and robust patterns from the data. Therefore, the network cannot rely on an activation of any set of hidden units, since they may be turned off at any time during training.

This random dropout can effectively prevent overfitting. The following figure shows an example of applying probability $p = 0.5$ during the training phase, whereby half the neurons will become inactive randomly. However, during prediction, all neurons will contribute to computing the pre-activations of the next layer.



For evaluation, all units must be active. To ensure that the overall activations are on the same scale during training and prediction, the activations of the active neurons have to be scaled appropriately, for example by halving the activation if $p_{drop} = 0.5$. However, since it inconvenient to always scale activations when making predictions, tensorflow and other tools can scale the activations during training, commonly referred to as inverse dropout.

In machine learning it is common to use ensemble learning to get a better result, but this is computationally expensive in deep learning, since it involves training several models and collecting and averaging the output of multiple models. The dropout offers a work around, with an efficient way to train many models at once, corresponding each dropout set to a model and the final prediction is the average over these.

The relationship between model ensembles and dropout is not immediately obvious. However, consider that in dropout, we have a different model for each mini-batch, due to setting the weights to zero randomly during each forward pass. Then, via iterating over the mini-batches, we essentially sample over $M = 2^h$ models, where h is the number of hidden units.

The restriction aspect that distinguishes dropout from regular ensembling, however, is that we share the weights over these "different models", which can be seen as a form of regularization. Then, during "interference", like predicting the labels in the test dataset, we can average over all these different models that we sampled over during training. This is very expensive, though.

Then, averaging the models, that is, computing the geometric mean of the classmembership probability that is returned by a model, i, can be computed as follows:

$$P_{Ensemble} = \left[\prod_{j=1}^{M} p^i\right]^{\frac{1}{M}} \tag{9}$$

Now the trick behind dropout is that this geometric mean of the model ensembles (here, M models) can be apporximated by scaling the predictions of tha last/final model sampled during training by a factor of 1/(1 - p), which is

much cheaper than computing the geometric mean explicitly using the previous equation. In fact, the approximation is exactly equivalent to the true geometric mean if we consider linear models.

Filter-wise dropout of convolutions are possible in some implementations, in the form of drop convolution filters instead of weights. If a layer does not affect sizes, layer-dropout is also technique (drop a whole layer), typically in long chains of similar layers.

2.3.2 Batch normalization

Batch Normalization is a technique commonly employed in neural networks to enhance training stability and performance. It operates by rescaling the activations of a layer to maintain their mean close to 0 and standard deviation close to 1. This normalization process is particularly beneficial in networks with numerous layers, as it helps mitigate issues related to vanishing or exploding gradients.

The mechanism behind why batch normalization has a positive effect is not fully understood. However, its practical advantages in improving convergence speed and allowing for more aggressive learning rates have made it a widely adopted practice in the training of deep neural networks. By normalizing the inputs of each layer during training, Batch Normalization contributes to the overall stability and efficiency of the learning process, promoting faster and more reliable convergence.

Using batch normalization in conjunction with activation functions can replace the need for regularization techniques, such as dropout.

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

2.3.3 Gradient clipping

Gradient clipping is a technique that tackles exploding gradients. The idea of gradient clipping is: if the gradient gets to large, we rescale it to keep it small. Gradient clipping is not really a part of regularization or optimisation, but rather a complementary method to be applied during optimization.

Gradient clipping is a technique used during the training of neural networks to prevent the exploding gradient problem. The exploding gradient problem occurs when the gradients of the loss function with respect to the model parameters become very large during training. This can lead to unstable training and difficulty in converging to an optimal solution.

Gradient clipping addresses this issue by imposing a threshold on the gradients. If the norm (magnitude) of the gradients exceeds this threshold, they are scaled down proportionally so that their norm is reduced to the specified threshold. This prevents the gradients from becoming too large and destabilizing the training process.

2.4 Activation functions

There are different kinds of activation functions such as ReLU, sigmoid and tanh. Some of these activation functions, like ReLU are mainly used in intermediate/hidden layers of an NN to add non-linearities to our model. But others, like sigmoid (for binary) and softmax (for multiclass), are added at the last/output layer, which results in class membership probabilities as the output of the model.

2.4.1 Softmax activation

The softmax activation function is commonly used in the output layer of a neural network for multi-class classification problems. It takes a vector of real numbers as input and transforms them into a probability distribution. The class activations sum to 1 and the formula for the softmax activation function is given by:

$$\sigma(a)_i = \frac{e^{a_i}}{\sum_{j=1}^{K} e^{a_j}} = h_i \tag{10}$$

$\sigma(a)_i$ is the output of the softmax function for the i-th class
a_i is the input value for the i-th class
K is the total number of classes

2.4.2 ReLu activation

ReLU stands for Rectified Linear Unit, and it is a popular activation function in deep learning.

The ReLU activation function is defined as follows:

$$\sigma(x) = \max(0, x) \tag{11}$$

In other words, if the input x is positive, the output is equal to x; if x is negative, the output is zero. This function introduces non-linearity to the model, allowing it to learn complex patterns and representations in the data. ReLU has become a standard choice in many neural network architectures due to its simplicity and effectiveness in mitigating the vanishing gradient problem.

However, it's worth noting that ReLU is not without its challenges. One issue is that neurons can become "dead" during training, meaning they always output zero. This can happen if the input to a ReLU neuron is always negative, and the weights are adjusted in a way that the neuron never activates. To address this, variations of ReLU, such as Leaky ReLU and Parametric ReLU, have been proposed.

2.5 Swish activation

The Swish activation function is defined as follows:

$$\sigma(x) = x \cdot sigmoid(x) = \frac{x}{1 + e^{-x}} \tag{12}$$

where σ is the sigmoid activation function and β is a learnable parameter.

In mathematical terms, the sigmoid function introduces a smoothness to the Swish activation, and the β parameter controls the shape of the function. Swish has been reported to perform well in various deep learning tasks, often outperforming other activation functions like ReLU. It is believed to combine some of the favorable properties of both sigmoid and ReLU activations.

The Swish activation function tends to allow more expressive power in the network due to its non-monotonicity, which can help the optimization process during training. However, it's worth noting that the effectiveness of activation functions can depend on the specific characteristics of the task at hand, and empirical testing is often necessary to determine the most suitable activation function for a given scenario.

2.6 Backpropagation in CNN

When we perform a convolution, we slide the filter across the input feature map. For a given position of the filter, we get a single value in the output feature map. To perform backpropagation through the convolution, we consider each position of the filter seperately, when computing the gradients. For each position, you compute how the output would change with a small change to the filter value at that position. As a result of the filter sliding accross the input, and being involved in the computation of many output values, the gradient for a specific weight of the filter is accumulated over all the positions where that weight was used.

2.6.1 Forward pass

The forward pass is the process by which the input data is passed through the network layer by layer, ultimately producing an output. The output is then compared to the actual target, and the error is used to update the networks parameters during the backpropagation phase.

2.6.2 Filter gradient

The filter gradient is determined by summing up local contributions, where each element in the patch is multiplied by the corresponding element in the output gradient. It's noteworthy that every weight in the filter contributes to each pixel in the output map. Consequently, any alteration in a weight within the filter induces changes in all output pixels. These modifications cumulatively affect the final loss, allowing for the straightforward calculation of derivatives. The derivatives are computed by considering the collective impact of each weight on every pixel in the output map. This process facilitates the network's understanding of how adjustments to individual weights in the filter influence the overall loss, aiding in the optimization of the model during training.

2.6.3 Atomatic tuning

Automatic tuning, facilitated by tools like Keras tuner, streamlines the optimization of hyperparameters in neural networks. This process involves exploring a range of hyperparameters to enhance the model's performance. Techniques employed in automatic tuning include:

- **Random Search:** This approach involves randomly sampling hyperparameter combinations over a specified grid or interval. Common hyperparameters include the number of units, learning rate, and the depth of layers.
- **Hyperband Optimization:** This method optimizes the use of resources by running multiple configurations in parallel, eliminating less promising ones and focusing resources on more promising candidates. It efficiently explores the hyperparameter space.
- **Bayesian Optimization:** Bayesian optimization leverages probabilistic models to guide the search for optimal hyperparameters. It adapts to the information gained during the tuning process, making it an intelligent and effective strategy.

However, it's crucial to exercise caution when using automatic tuning. The process can be computationally intensive and time-consuming, especially if the search space is wide. Running automatic tuning too extensively without proper consideration of computational resources can lead to prolonged execution times, potentially spanning days. Therefore, a balance must be struck to ensure effective tuning without excessive computational demands.

2.7 Loss functions for classification

Focusing on classification problems, depending on the type of problem (binary versus multiclass) and type of output (logits versus probabilities), we should choose the appropriate loss function to train our model. **Binary cross-entropy** is the loss function for binary classification, and **categorical cross-entropy** is the loss function for multiclass classification. In keras there are two types of categorical cross-entropy, depending on the ground thruth labels being one-hot encoded or integers.

The following table describes three loss functions available in keras for dealing with all three cases: binary classification, multiclass with one-hot encoded ground thruth labels, and multiclass with integer/sparse labels. Each one of these three loss functions also has the option to recieve the predictions in the form of logits or class member probabilities:

Loss function	Usage	Examples	
		Using probabilities	Using logits
		from_logits=False	*from_logits=True*
`BinaryCrossentropy`	Binary classification	`y_true:` 1 `y_pred:` 0.69	`y_true:` 1 `y_pred:` 0.8
`CategoricalCrossentropy`	Multiclass classification	`y_true:` 0 0 1 `y_pred:` 0.30 0.15 0.55	`y_true:` 0 0 1 `y_pred:` 1.5 0.8 2.1
`Sparse CategoricalCrossentropy`	Multiclass classification	`y_true:` 2 `y_pred:` 0.30 0.15 0.55	`y_true:` 2 `y_pred:` 1.5 0.8 2.1

3 Optimization

Optimization is a research subject of its own. All AI models use stochastic gradient descent (SGD) on some level in the core. A good optimizer leads to faster training, which leads to a faster model, which leads to a more robust model. A machine learns through optimization.

In standard optimization the learning rate is the only hyperparameter.

Important things to note is that optimization in not the same as backpropagation. You should also know about the chain rule in regards to derivatives, since derivation is a huge part of optimization.

3.1 Gradient descent

In machine learning, like adaline, we compute the gradient descent based on the whole training dataset and update the weights of the model by taking a step in the opposite direction of the gradient. In order to find the optimal weights of the model, we optimized an objective function (for example the sum of squared errors (SSE)) as the cost function $J(w)$. In addition the gradient is multiplied by the **learning rate** η. Furthermore we defined the **activation function** $\Phi()$,

that uses the **net input** z. Lastly we use the activation function to implement a threshold function to perform the binary classification.

Gradient descent is an iterative optimization algorithm used for finding the minimum of a function, typically the loss function in the context of machine learning. The idea is to adjust the parameters of a model iteratively in the direction opposite to the gradient of the objective function with respect to those parameters.

The learning rate is a crucial hyperparameter that influences the size of the steps taken during the optimization process. Too small learning rate may lead to slow convergence and getting stuck in a local minimum, too large learning rate can lead to skipping the minimum.

The update rule for gradient descent is:

$$\theta = \theta - \alpha \nabla J(\theta)$$

where:

- θ represents the parameter vector (or set of parameters),
- α is the learning rate,
- $\nabla J(\theta)$ is the gradient of the cost or loss function $J(\theta)$ with respect to the parameters θ.

3.2 Stochastic Gradient Descent (SGD)

Stochastic gradient descent (SGD) optimization accelerates the model learning. It learns faster, but also has a noisy nature, that luckily is beneficial when training multilayer NNs with nonlinear activation functions. The added noise will help to escape the local cost minima. SDG is computationally more efficient, especially for large datasets.

Instead of using the entire dataset, like gradient descent, SGD randomly selects a single datapoint (or a small sub-set, often called mini-batch) from the training set. It calculates the gradient of the loss function based on this batch and updates the parameters accordinly. This process is repeated for each batch in the training set.

The update rule for a parameter θ in SGD is given by:

$$\theta = \theta - \alpha \nabla J(\theta; x^{(i)}, y^{(i)})$$

where:

- θ is the parameter being updated,
- α is the learning rate,
- $J(\theta; x^{(i)}, y^{(i)})$ is the loss function for a single data point $(x^{(i)}, y^{(i)})$,
- $\nabla J(\theta; x^{(i)}, y^{(i)})$ is the gradient of the loss function with respect to θ for the single data point.

3.2.1 Batch size

(For reference)

- Larger batches provide a more accurate estimate of the gradient, but with less than linear returns (computation complexity).
- Multicore architectures are usually underutilized by extremely small batches. This motivates using some absolute minimum batch size, below which there is no reduction in the time to process a minibatch.
- If all examples in the batch are to be processed in parallel (as is typically the case), then the amount of memory scales with the batch size. For many hardware setups this is the limiting factor in batch size.
- Some kinds of hardware achieve better runtime with specific sizes of arrays. Especially when using GPUs, it is common for power of 2 batch sizes to offer better runtime. Typical power of 2 batch sizes range from 32 to 256, with 16 sometimes being attempted for large models.
- Small batches can offer a regularizing effect (Wilson and Martinez, 2003), perhaps due to the noise they add to the learning process. Generalization error is often best for a batch size of 1. Training with such a small batch size might require a small learning rate to maintain stability due to the high variance in the estimate of the gradient. The total runtime can be very high due to the need to make more steps, both because of the reduced learning rate and because it takes more steps to observe the entire training set.

3.3 Momentum

The classical momentum method is a technique for accelerating gradient descent that accumulates a velocity vector in directions of persistent reduction in the objective across iterations. Momentum affects the convergence most in the transient phase, i.e. before tuning.

Momentum optimization is a technique used to accelerate the training of machine learning models, especially in the context of gradient-based optimization algorithms like stochastic gradient descent (SGD). The idea behind momentum is to add a fraction of the previous update to the current update, allowing the optimization process to build up momentum and dampen oscillations.

Momentum is designed to decrease the dependency of the learning rate. Picture a ball rolling down the slope, and the ball is supposed to just roll over the local minima using momentum. When the slope is negativ, the momentum increases. When the slope is changing or is positive, the momentum is decreasing. With momentum optimization the gradient is computed to update the parameters, and then the momentum term is added.

The update rule for the momentum optimization can be expressed as follows:

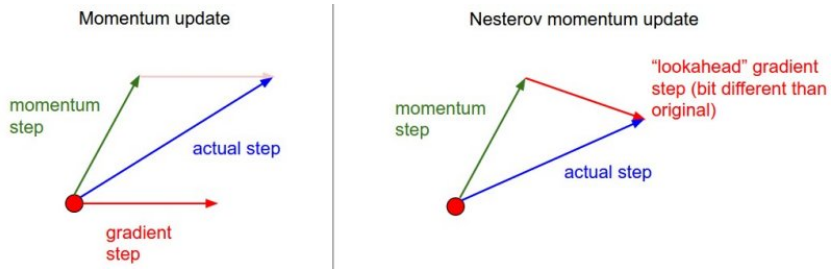
$$\nu_t = \gamma\nu_{t-1} + \eta\nabla_\theta J(\theta), \tag{13}$$

γ: controls how much is remembered. Is often around 0.9.
ν_{t-1}: the memory from previous iterations

3.4 Nesterov momentum

Momentum may still be large near the optimum, but nesterov momentum adds a partial update of the gradient before adding momentum to the weight update. First push is anticipatory, guessing a likely gradient. Nesterov momentum first applies the momentum term to the current parameters and then computes the gradient on the adjusted parameters. This adjustment can help in reducing the oscillations typically observed with standard momentum.

$$\nu_t = \gamma\nu_{t-1} + \eta\nabla\theta J(\theta - \gamma\nu_{t-1}) \tag{14}$$



Use nesterov momentum when there are sharp narrow valleys, noisy gradient, for stability/robustness, and as a default.

3.5 Decay

Where momentum accellerates in promising directions, decay reduces the learning rate. Learning rate decay is used to fine-tune the optimization process and improve convergence. The idea is to start with a relatively large learning rate and gradually decrease it over time.

Learning rate decay can help address challenges in the optimization process. Initially, a higher learning rate allows the model to make larger updates and explore the parameter space more quickly. As training progresses, reducing the learning rate can help fine-tune the model and converge to a more accurate solution.

The choice of the decay strategy and its hyperparameters (such as decay factor or decay schedule) often involves experimentation and tuning for specific datasets and models. Learning rate decay is a common technique to improve the training of neural networks and other machine learning models. You need to shrink the learning rate graduelly, since too fast decays leads to never finding the minimum. Decay with restart and more adaptive decays may be useful.

3.6 Algorithms with adaptive learning rates

Momentum reduces the dependency on the learning rate, by adding another hyperparameter, which could be seen as against it purpose. Algorithms with adaptive learning rate have seperate learning rates for each parameter, and updates the learning rate automatically thoughout the course of the learning. It is recommended to just use Adam, since it combines the idea of momentom and RMSProp.

3.6.1 AdaGrad

Update rule: AdaGrad adapts the learning rates of each parameter individually based on the past gradients. It divides the learning rate by the square root of the sum of squared gradients for each parameter.

Learning rate adaption: It has a self-adaptive learning rate. Parameters that receive high gradients will have a lower effective learning rate, and parameters with low gradients will have a higher effective learning rate.

Advantage: It is particularly useful in scenarios where the features have very different scales, as it can automatically adjust the learning rates.

Disadvantage: Over time, the learning rates can become very small, which can lead to slow convergence

3.6.2 RMSProp

Update rule: RMSProp also adapts the learning rates of each parameter based on the past gradients, but it uses an exponentially decaying average of squared gradients rather than their sum.

Learning rate adaption: It addresses the diminishing learning rate problem of AdaGrad. By using an exponentially decaying average, RMSProp avoids having learning rates that become too small over time.

Advantage: It works well in a wide range of scenarios and addresses some of the limitations of AdaGrad.

Disadvantage: It still may not be as effective in scenarios with very noisy gradients.

3.6.3 Adam

Update rule: Adam combines the ideas of both Momentum and RMSProp. It maintains two moving averages for each parameter: the first moment (mean) of the gradients and the second moment (uncentered variance) of the gradients.

Learning rate adaption: It adapts the learning rates based on the first and second moments. This allows it to have separate learning rates for different parameters and to adjust them dynamically during training.

Advantage: Adam is considered a very effective and versatile optimization algorithm. It's widely used in practice and often provides fast convergence.

Disadvantage: It may sometimes be sensitive to the choice of hyperparameters and can sometimes exhibit erratic behavior.

4 Achitectures

In Convolutional Neural Networks, architecture refers to the overall structure and design of the network. It encompasses the arrangement of layers, the types of layers used (such as convolutional, pooling, and fully connected layers), the connectivity patterns between layers, and any additional components like normalization layers or skip connections.

CNN architectures vary depending on the specific task they're designed for, such as image classification, object detection, or segmentation. These architectures differ in terms of their depth, the arrangement of layers, the use of techniques like skip connections or inception modules, and other design choices.

Architectures are often optimized for specific tasks or datasets, and researchers continually propose new architectures or modify existing ones to improve performance on various tasks or to make them more efficient in terms of computational resources or memory usage.

The architectures could be non-sequential network topology, modular - blocks of convolutions used several times, pre-built or pre-trained models or you would need keras to implement it.

4.1 Non-Sequential Network Topology

In the realm of neural network architectures, non-sequential network topology deviates from the traditional sequential flow of layers. Instead, it introduces innovative connections and structures to enhance model performance. Some notable features include:

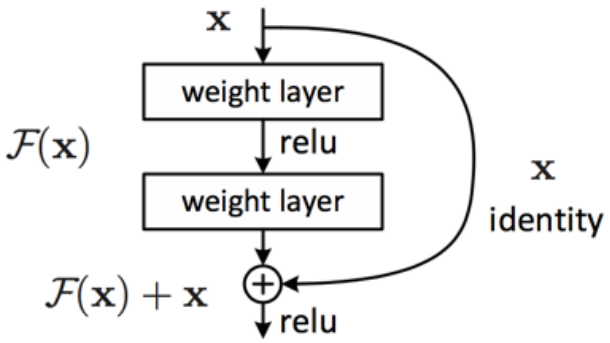
- **Connections between non-neighbour layers**, for example
  - Residual Networks (ResNets) and Skip-Connections: These architectures introduce connections that bypass one or more layers, allowing the model to learn residual information. This helps in mitigating the vanishing gradient problem and facilitates the training of deeper networks.

– DenseNet (Densely Connected Convolutional Networks): DenseNet takes this concept further by establishing connections to all subsequent layers. Each layer receives direct input from all preceding layers, promoting feature reuse and fostering dense connectivity within the network.

- **Parallel filter groups**, e.g. series of convolutions in parallell. An example of parallell filter groups are inception cell. This approach enables the model to capture features at multiple spatial scales, enhancing its ability to learn complex patterns.
- **Extra input and/or output layers in the network**, providing supplementary information to the network. These layers contribute to the overall flexibility and functionality of the model.

4.1.1 Residual networks

A residual network adds activation from a layer to the pre-activation of a later layer.



Imagine you are trying to model or approximate a function $H(x)$ with your neural network. The network learns the difference or the residual between the input and the desired output. The residual is denoted as $F(x)$. The network is effectively trying to learn $F(x)$ such that when it is added to the input x, we get closer to $H(x)$.

$$F(x) = H(x) - x \tag{15}$$

In ResNet, each residual block (a couple of layers in the network) aims to learn the residual function $F(x)$. The output block is then $F(x) + x$. This addition is facilitated by the skip or shortcut connection, which carries x over the block and adds it to the output. Think of it as you are trying to teach

someone a new topic. Instead of explaining everything from scratch, you first identify what they already know (the input x) and then focuses on teaching what they are missing or what is different (the residual $F(x)$). This often makes the learning process more efficient. The problem ResNet primarily adresses is the lack of data.

Which layer is essential in ResNet to create the shortcut connections? Identity or projection layer What is the main purpose of the skip connections in ResNet? To facilitate the training of deeper networks by allowing gradient flow

4.1.2 Inception networks

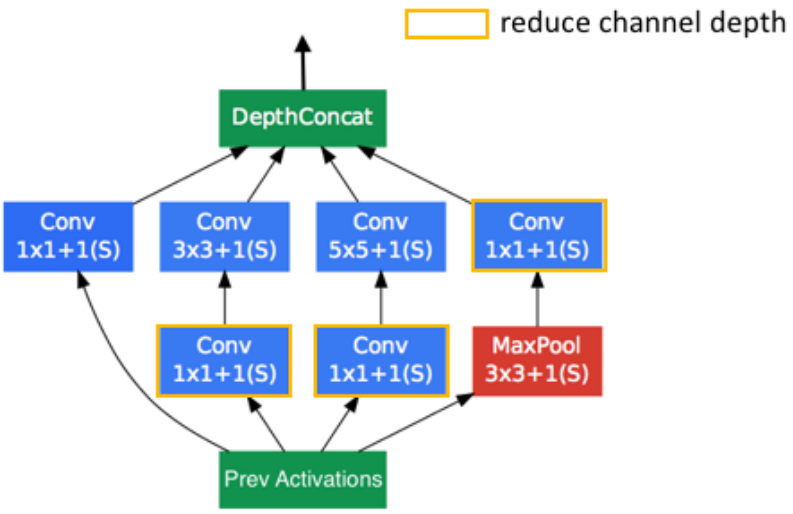
Inception networks, known for their inventive inception cells, represent a major leap in neural network design, especially valued in computer vision. They're tailored to capture features at different scales and cleverly combine them, enhancing the model's ability to understand complex patterns in the input data.

The relevant characteristics of Inception cells include:

- **Feature extraction on different scales:** Inception modules employ convolutional layers with different kernel sizes within the same module. This forces the network to extract features at multiple levels of details, so the cell really understand what is going on in the input data.
- **Concatenation of output:** The outputs from convolutional layers with distinct kernel sizes are concatenated. This fusion of features ensures that the model can harness information from diverse scales, contributing to its proficiency in discerning complex structures within the data.
- **Use of same-padding:** To maintain consistency in spatial dimensions during concatenation, Inception modules typically utilize same padding in their convolutional layers. This strategy preserves the sizes of feature maps, facilitating seamless integration of information from different scales.

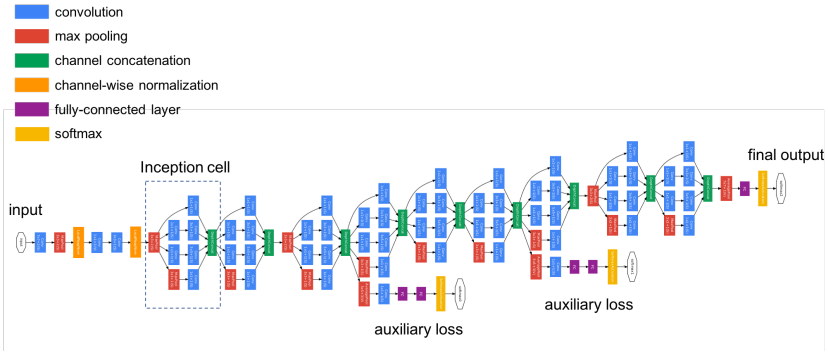
Inception networks are often strategically combined with bottlenecks to address the challenge of reducing channel depth, thereby minimizing the number of parameters in the network. This integration involves convolutional layers that sum over input channels post-convolution, resulting in a more parameter-efficient architecture.

The teamwork of Inception cells/modules and bottlenecks works great in lots of computer vision jobs. It helps extract and put together features from different scales in neural networks.



One might wonder: won't using multiple filters of varying sizes dramatically increase computational costs? The answer is yes - but here's where the Inception architecture introduces a clever trick. Before applying larger filters like 5x5, the network first reduces the depth (number of channels) of the input using 1x1 convolutions. This process, often called bottleneck layers, compresses the information without losing too much of it, and thus reduces computational costs.

In Inception modules, what is the purpose of the 1x1 convolutions before the 3x3 and 5x5 convolutions? To reduce computational cost by decreasing the number of channels



4.2 Building on a pretrained network

Building on a pretrained neural network is a powerful strategy in the field of deep learning, allowing the reuse of established models for new or fine-tuned purposes. In frameworks like Keras, this practice is often referred to as "applications".

When dealing with neural networks applied to image-related tasks, the primary objectives are to generate meaningful features, combine them into recognizable objects, and distinguish between different types of objects. Leveraging a pretrained network provides a head start in achieving these goals.

The typical strategy involves several key steps:

- **Reuse existing networks (applications in Keras):** Pretrained models, such as those available in Keras, offer a variety of architectures that have been trained on large datasets for tasks like image classification. These networks serve as excellent starting points for related tasks.
- **Task-specific modification:**
  - **Strip final dense layer(s):** The final dense layer, often equipped with a softmax activation for classification, is removed. This step is crucial as it is specific to the original task for which the network was pretrained.
  - **Freeze network parameters:** The parameters of the pretrained layers are frozen, preventing them from being updated during subsequent training. This preserves the learned features from the original task.
  - **Train new dense layers:** New dense layers are added to the network, serving as task-specific heads. These layers are then trained to adapt the network to the nuances of the new or fine-tuned task. The frozen layers act as powerful feature extractors, while the new layers specialize in capturing task-specific information.

By building on pretrained networks in this manner, practitioners can benefit from transfer learning, speeding up the training process, and achieving effective results with limited data for specific applications.

Some available networks in Keras are:

- Xception
- VGG16
- VGG19
- ResNet (50, 101, 152; v1, v2)
- Inception V3
- Inception ResNet V2
- MobileNet (v1, v2, v3Large, v3Small)
- DenseNet (121, 169, 201)
- NASNet (Large, Mobile)
- EfficientNet (B0, ..., B7)

4.3 Kernel stacking

Kernel stacking is a fundamental strategy in the design of convolutional neural networks (CNNs), offering insights into optimizing feature extraction and computational efficiency.

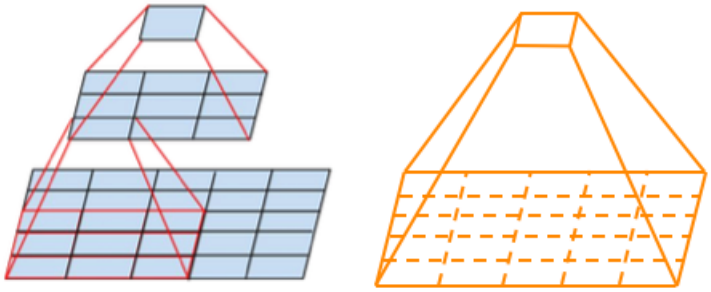
Larger spatial filters exhibit increased expressiveness, allowing for the extraction of features at a more extensive scale, resulting in a larger receptive field. This capability enables the network to capture broader patterns and relationships within the input data.

An effective approach in kernel stacking involves using multiple 3x3 filters to approximate the impact of a larger filter size. For instance, the spatial coverage of two stacked 3x3 filters matches that of a single 5x5 filter. This not only enhances computational efficiency but also simplifies the model. The parameter reduction is notable:

$$\text{Parameters for a 5x5 filter: } 5 \times 5 \times \text{channels} = 25 \times \text{channels}$$

$$\text{Parameters for two stacked 3x3 filters: } 2 \times 3 \times 3 \times \text{channels} = 18 \times \text{channels}$$

Stacking filters in this manner intensifies the focus on the central region of the larger filters. This concentration enhances the network's ability to capture detailed information at the center of the receptive field, potentially improving its capability to recognize intricate patterns.



Building on the concept of kernel stacking, considerations extend to the activation functions applied within the network. For the most accurate representation of features, linear activation functions (or no activation) within the stacked layers are recommended. This ensures a direct and unaltered transformation of the input.

While linear activation provides accurate representations, incorporating activation functions (such as Rectified Linear Unit - ReLU) between stacked lay-

ers enhances the network's performance. Activation functions introduce non-linearity, facilitating the extraction of complex patterns and improving the model's ability to learn and generalize.

These considerations highlight the nuanced choices involved in kernel stacking, encompassing both spatial filter strategies and activation functions to achieve an optimal balance between accuracy and efficiency in CNNs.

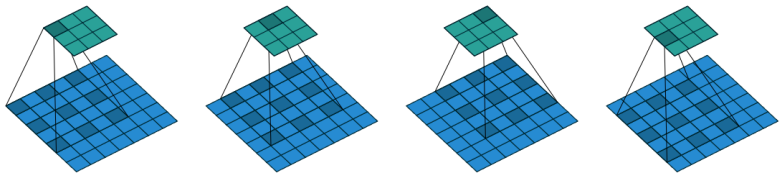
4.4 Atrous/dilated convolutions

Atrous/dilated convolutions are a convolutional operation that enhances the receptive field by spreading the convolutional kernel.

Atrous convolutions achieve an enlarged receptive field by introducing gaps (or dilations) between the weights of the convolutional kernel. This allows the network to capture information from a broader spatial context.

Consider a 3x3 kernel applied over a 5x5 area on a 7x7 image with no padding and a stride of 1. In this scenario, the atrous convolution effectively expands the spatial context, enabling the network to gather information from a larger region without increasing the number of parameters.

In Keras, specifying a dilation rate of 2x2 for a 3x3 filter extends the effective receptive field to a 5x5 area on the input. It's important to note that although the spatial context expands, the number of parameters remains the same, as the kernel retains its original 3x3 weights.



4.5 Early stopping and continued optimization

In machine learning, **early stopping** is a technique employed during model training to prevent overfitting and improve generalization. Regardless of the optimization strategy used, the loss function may reach a plateau, indicating that further training may not result in significant improvements. Early stopping involves halting the training process based on certain criteria:

- **Minimum loss change over epochs:** Training iterations are stopped if the change in loss over consecutive epochs falls below a predefined threshold. This suggests that the model has likely converged, and additional training may not yield substantial benefits.
- **Reached a certain threshold:** The training process is terminated if the loss value reaches a specified threshold. This threshold is typically determined based on the desired level of performance or when further improvement is deemed unnecessary.

In scenarios where early stopping is applied, **continued optimization** strategies can be implemented to further refine the model:

- **Save Weights in a Model:** Before stopping the training process, the weights of the model can be saved. This allows for later reinitialization of the model with the saved weights.
- **Continue Fitting:** After saving the weights, the training process can be resumed. However, to avoid potential issues, certain precautions should be taken:
  - **Reset decay/momentum:** Resetting decay or momentum parameters helps the model adapt to the new training phase.
  - **Learning rate scheduling:** Implementing learning rate scheduling, such as cosine-based scheduling, can be beneficial. It adjusts the learning rate during training, potentially improving convergence and preventing overshooting.
  - **Different optimizer:** Experimenting with a different optimizer during continued optimization can provide the model with new opportunities for improvement.

4.6 Test time augmentation

Test time augmentation involves augmenting images during the prediction phase for test data. While this might initially seem counterintuitive—why introduce difficulty during testing?—the rationale behind it is to provide the model with several opportunities to make correct predictions and then average the results.

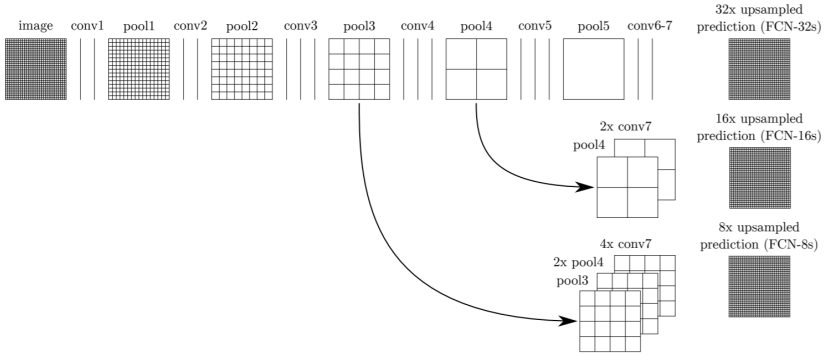
A straightforward way to implement test time augmentation is to reuse the `ImageDataGenerator` from the training data on the test data. This process does not require re-fitting the model. By applying augmentation techniques during testing, the model is exposed to variations it might not have encountered during training, potentially improving its robustness.

4.7 Semantic Segmentation models

Semantic Segmentation is also refered to as dense rediction, and labels pixelwise, assigning a class to every pixel in an image. Semantic segmentation models are typically constructed using the foundational concepts covered in this course. A significant advancement in this field was achieved with the introduction of fully convolutional networks for semantic segmentation. These models generally consist of a series of convolutional and pooling blocks, followed by deconvolution/strided convolution or bilinear upsampling at the end. This final step may involve the combination of information from two or more levels to upscale the output to the full image size. This process requires a careful consideration of the tradeoff between spatially fine details and semantic precision.

A practical example of such a model is utilizing Inception V3 as a basis. In this case, dense layers are replaced with Conv2d, and the model includes upscaling at the end to achieve the desired segmentation results.

The following image is an example of a fully convolutional network.



We use the feature map in the earlier layers an concatinate them, with the extracted features and upsample from that point. The closer to the begininng the more spatial information you have. The deeper we are to the flattened area, the less spatial information we have.

4.7.1 U-Net

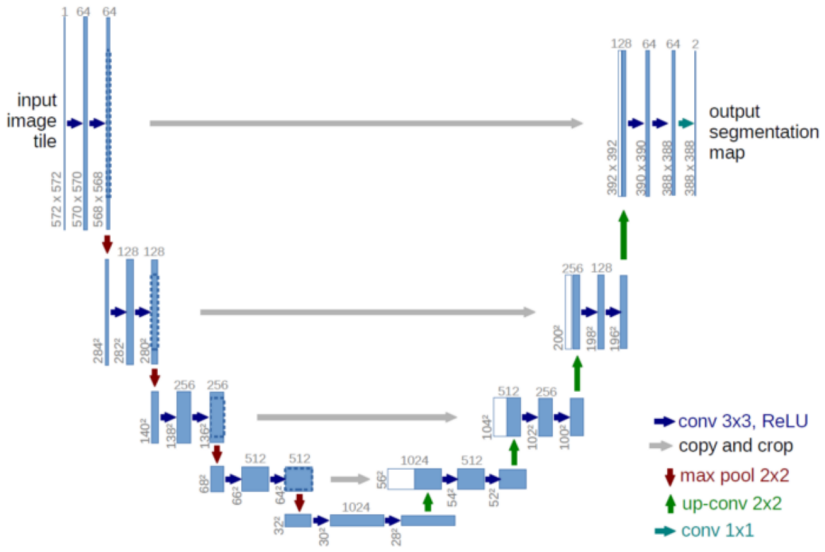
U-Net is a convolutional neural network architecture designed for semantic segmentation tasks in image processing. The U-Net architecture consists of a contracting path to capture context, and a symmetric expanding path to achieve precise localization, with skip-connections.

The contracting path (the encoding) progressively reduces spatial dimensions and captures abstract feature.It involves convolutional and pooling layers, until we reach the bottleneck. This will extract the most important features without considering the spatial information.

The expanding path (the decoding)restores spatial dimensions and refines the segmentation. It uses upsampling and convolutional layers for pixel-wise classification.

Skip-connections help in preserving the spatial information lost during downsampling by providing a shortcut from encoder to decoder.

U-Net is widely used in medical image analysis and other applications where precise segmentation is essential.



4.7.2 Other semantic segmentation networks

Several other networks are employed for semantic segmentation beyond U-Net. If you understand U-net, the rest follow the same structure. For example, we have V-Net for 3D imaging data, which is specifically crafted for three-dimensional imaging data, V-Net stands out as a notable architecture in this field.

There various architectures, but som are less intuitive. Some take the scene into account, or even more advanced stuff. Many use ROIs (region of interest) as intermediate steps, while some use sets of atrous convolutions

5 Mathematics in convolutional neural networks

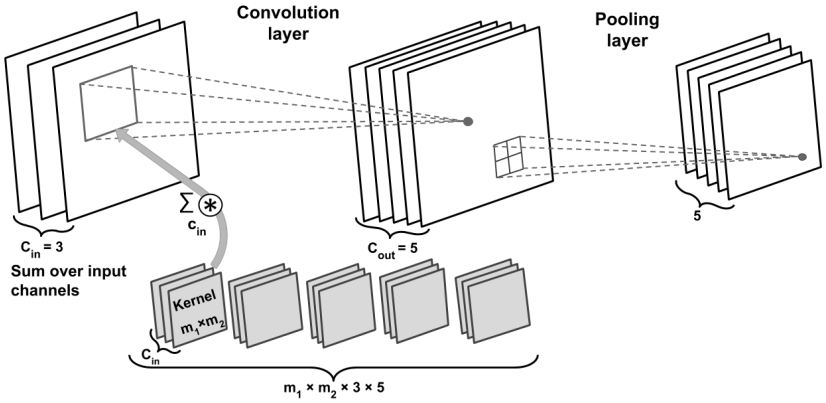
5.0.1 Determining the size of the convolution output

The output size of a convolution is determined by the total number of times that we shift the filter, $\mathbf{w}$, along the input vector. If we have input vector, $\mathbf{x}$, of size n and filter, $\mathbf{w}$, of size m, then the output size, o, resulting from $\mathbf{y} = \mathbf{x} * \mathbf{w}$ with padding, p, and stride, s, will be:

$$o = \left[\frac{n+2p-m}{s}\right] + 1 \tag{16}$$

This was also covered in a previous section

Number of trainable parameters in one layer:
m_1 x m_2 x number of color channels x number of kernels + number of kernels



Suppose your input is a 300 by 300 color image, and you use convolutional layer with 100 filters that are each 5x5. How many paramters does the hidden layer have (without bias)?
5 x 5 x 3 x 100 = 7 500

You have an input volume that is 63x63x16, and convolve it with 32 filters that are each 7x7, using a stride of 2 and no padding. What is the output volume?
O = (63-7)/2 + 1 = 29
29 x 29 x 32 = 26 912

You have an input volume that is 32x32x16, and aply max pooling with a stride of 2 and a filter size of 2. what is the output volume?
O = (32+0-2)/2+1=16
16x16x16

If your input is 64x64x16, how many paramters are there in a single 1x1 convolution filter, including bias?
1x1x16+1=17

6 Useful pages and videos

But what is a convolution? - 3Blue1Brown

Optimization in Deep Learning, All Major Optimizers Explained in Detail - Coding Lane

Optimization for Deep Learning, Momentum, RMSprop, AdaGrad, Adam - DeepBean

ResNet (actually) explained in under 10 minutes - rupert ai

Inception Explained: Understanding the Architecture and Module of Inception Networks - Nicolai Nielsen

The U-Net (actually) explained in 10 minutes - rupert ai

Image Classification with Convolutional Neural Networks (CNNs) - StatQuest with Josh Starmer

Dive into deep learning - online book