Deep Learning

Matteo Donati

November 23, 2021

Original work: https://github.com/matteodonati/unibo_artificial_intelligence_notes/

Contents

Chapter 1

Introduction

Deep learning is the branch of machine learning which uses neural networks in order to make inferences. Deep learning techniques can be applied to all the problems suitable for machine learning.

1.1 Neural Networks

Neural networks are networks of artificial neurons. Each neuron takes multiple inputs and produces a single output. Moreover, a neural network is usually divided into layers, where each layer contains a specific number of neurons. This type of networks has been introduced in order to understand, via simulation, how the brain works, and to solve practical problems difficult to address with algorithmic techniques (e.g. object recognition). In particular:

- Each neuron implements a logistic regressor: typically a non-linear function

$$y = \sigma(\boldsymbol{w}\boldsymbol{x} + b) \tag{1.1}$$

  where $\boldsymbol{x}$ is the vector containing all the inputs received from the previous layer, $\boldsymbol{w}$ is the vector containing the weights associated with each input, b is a scalar called *bias*, and σ is an *activation function*.

- If a given network is acyclic, then it is called a **feed-forward network**, while if it has cycles it is called a **recurrent network**.

- If a given network is composed of more than one hidden layer, then it is called a **deep network**, otherwise it is called a **shallow network**.

- A **dense layer** is a layer in which each neuron at layer $k-1$ is connected to each neuron at layer k.

- A **convolutional layer** is a layer in which each neuron at layer $k-1$ is connected via a parametric **kernel** to a fixed subset of neurons at layer k. In convolutional neural networks, the generic kernel is convolved over the whole $k-1$ layer.

- The weights $\boldsymbol{w}$ are the parameters of the model and they are learned during the training phase. Moreover, the number of layers and the number of neurons in each layer are **hyper-parameters** chosen by the user before the training phase.

1.2 Features and Deep Features

Any individual measurable property of data is called a **feature**. Discovering good features from data is a complex task. In particular, deep learning exploits a hierarchical organization of the learning model, allowing complex features to be computed in terms of simpler ones (i.e. each layer synthesize new features in terms of the previous ones), through non-linear transformations.

1.3 Successful Applications

Some examples of successful applications are the following:

- Image processing and computer vision:
  - Image classification and object detection.
  - Image segmentation, scene understanding.
  - Style transfer.
  - Deep dreams and inceptionism.
- Natural language processing:
  - Speech recognition.
  - Text processing.
- Generative modelling:
- Deep reinforcement learning:
  - Robot navigation and autonomous driving.
  - Planning and simulation.

Chapter 2

Expressiveness

2.1 Single Perceptron

The single perceptron implements a binary threshold. This simple network is able to produce the following output:

$$y = \begin{cases} 1 & \text{if } \sum_i w_i x_i + b \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{2.1}$$

In particular, the bias set the position of the threshold. More generally, the set of points such that $\sum_i w_i x_i + b = 0$ defines a hyperplane in the space of the variables x_i. For example, when $\boldsymbol{x} \in \mathbb{R}^2$, this hyperplane is a line which divides the Cartesian plane into two parts.

2.2 Multi-Layer Perceptron

Multi-layer perceptrons (MLPs) are composed by multiple single perceptrons. In particular:

- Multi-layer perceptrons can be either shallow networks or deep networks, however with a multiple hidden layers (i.e. using a deep network) one is able to approximate a given function with less neural units.
- The chosen activation functions play an essential role, since these functions are the only source of non-linearity.
- Every continuous function $f : \mathbb{R} \to [0, 1]$ can be approximated by a shallow neural network.

two layers (hidden, output)

Chapter 3

Training

The goal of training a neural network is to minimize a given loss function. In particular, a loss function L is a function of the parameters of the network, $\boldsymbol{\theta}$, which, for every input the network receives, computes the error between the predicted output and the true output. In particular, the **gradient** of a multi-variate function is the vector containing all the partial derivatives of the given function:

$$\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}) = \left[\frac{\partial L(\boldsymbol{\theta})}{\partial \theta_1}, \ldots, \frac{\partial L(\boldsymbol{\theta})}{\partial \theta_n} \right] \tag{3.1}$$

In particular, the gradient of a multi-variate function points in the direction of steepest ascent. This makes $-\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta})$ a **descent direction**.

3.1 Gradient Descent

Gradient descent is an iterative descent method which, at every iteration, computes an approximated solution to the minimization problem at hand, $w_{k+1} = g(w_k)$, which, in the case of gradient descent, is translated into the following update rule:

$$\boldsymbol{\theta} \overset{\leftarrow}{=} \boldsymbol{\theta} - \mu \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}) \tag{3.2}$$

where μ is the **learning rate** or **step-size** of the algorithm, and $-\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta})$ is the selected descent direction.

3.1.1 Stochastic Gradient Descent

Given N data points (i.e. samples), stochastic gradient descent approximates the true gradient of the loss function by considering a number $M < N$ of data points. In particular, in deep learning one often considers objective functions which are evaluated as the sum of the errors, L_n, computed at each training sample n:

$$L(\boldsymbol{\theta}) = \sum_{n=1}^{N} L_n(\boldsymbol{\theta}) \tag{3.3}$$

The gradient of such loss function is then computed as follows:

$$\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}) = \sum_{n=1}^{N} \nabla(L_n(\boldsymbol{\theta})) \tag{3.4}$$

Since computing the gradient of such loss function is quite expensive (in terms of memory consumption), by using stochastic gradient descent it is possible to select a **mini-batch** composed of $M < N$ samples such that:

small mini-batch allows one to frequently update weights, but in so doing one approximate more the real gradient

$$\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}) = \sum_{n=1}^{M} \nabla(L_n(\boldsymbol{\theta})) \tag{3.5}$$

In particular, by using stochastic gradient descent the direction of the gradient could be less precise but, by using appropriate values for the learning rate, one can obtain the results as the ones obtained by applying gradient descent.

3.1.2 Gradient Descent with Momentum

if one is going well, then one train faster (e.g. the ball acquires momentum), this allows one to cross local minima or plateau

If, during consecutive training steps, the gradient seems to follow a stable direction, one could improve its magnitude, simulating the fact that it is acquiring a **momentum** along the descent direction, similarly to a ball rolling down a surface. The objective of gradient descent with momentum is to reduce the risk to get stuck in a local minimum or in a plateau. In this case, the update rule is the following:

update vector: $v^t = \mu \nabla L(w) + \alpha v^{t-1}$

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \mu \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}) + \alpha \Delta \boldsymbol{\theta}_k \tag{3.6}$$

momentum

where $\alpha \in [0, 1]$ and $\Delta \boldsymbol{\theta}_k = \boldsymbol{\theta}_k - \boldsymbol{\theta}_{k-1}$ is the update obtained at iteration k.

3.2 Backpropagation

The backpropagation algorithm is simply the instantiation of the gradient descent technique to the case of neural networks. In particular, this algorithm provides iterative rules to compute partial derivatives of the loss function with respect to each parameter of the network and then it applies gradient descent to minimize the given objective function.

- The backpropagation algorithm utilizes the **chain rule** of derivation. Given two derivable functions, f and g, the derivative of the composite function $h(x) = f(g(x))$ is:

  $x \xrightarrow{g} g(x) \xrightarrow{f} f(g(x))$

  $$h'(x) = f'(g(x))g'(x) = \frac{\partial f}{\partial g}\frac{\partial g}{\partial x} \tag{3.7}$$

  how fast f changes when x is changed

  This can be extended to consider the multi-variate case. Given a multi-variate function $f(x, y)$ and two single variable functions $x(t)$ and $y(t)$, the derivative of the composition is given by:

  $$\frac{\partial f(x(t), y(t))}{\partial t} = \frac{\partial f}{\partial x}\frac{\partial x}{\partial t} + \frac{\partial f}{\partial y}\frac{\partial y}{\partial t} \tag{3.8}$$

If $\boldsymbol{v}(t) = [x(t), y(t)]^T$, then:

$$\frac{\partial f(x(t), y(t))}{\partial t} = \nabla f \cdot \boldsymbol{v}'(t) \tag{3.9}$$

- Let $E(\boldsymbol{\theta})$ the given loss function, a_i^l the output (i.e. $\sigma(\boldsymbol{w}\boldsymbol{x} + b)$) of the i-th neuron in the l-th layer, z_i^l the value of the weighted input (i.e. $(\boldsymbol{w}\boldsymbol{x} + b)$) of the i-th neuron in the l-th layer, w_{kj}^l the weight connecting the neuron k in the $(l-1)$-th layer and neuron j in the l-th layer, and b_j^l the bias of the j-th neuron in the l-th layer. The backpropagation algorithm for a generic MLP works by applying the following steps:

  - **Input**: set

    $$\boldsymbol{a}^0 = \boldsymbol{x} \tag{3.10}$$

    where $\boldsymbol{x}$ is a generic training instance given to the network.
  - **Feed-forward**: for $l = 1, 2, \ldots, L$ compute

    $$\boldsymbol{z}^l = \boldsymbol{w}^l \boldsymbol{a}^{l-1} + \boldsymbol{b}^l \tag{3.11}$$

    $$\boldsymbol{a}^l = \sigma(\boldsymbol{z}^l) \tag{3.12}$$

  - **Output error**: compute the vector

    $$\boldsymbol{\delta}^L = \frac{\partial E}{\partial \boldsymbol{z}^L} = \nabla_{\boldsymbol{a}^L} E \ \odot \ \sigma'(\boldsymbol{z}^L) \tag{3.13}$$

    where $\nabla_{\boldsymbol{a}^L} E$ is the gradient of E with respect to $\boldsymbol{a}^L$, $\sigma'(\boldsymbol{z}^L)$ is the vector computed evaluating the first derivative of σ in $\boldsymbol{z}^L$, $\odot$ is the Hadamard product (i.e. component-wise) product.
  - **Backpropagation**: for $l = L-1, L-2, \ldots, 1$ compute

    $$\boldsymbol{\delta}^l = \frac{\partial E}{\partial \boldsymbol{z}^l} = (\boldsymbol{w}^{l+1})^T \boldsymbol{\delta}^{l+1} \ \odot \ \sigma'(\boldsymbol{z}^l) \tag{3.14}$$

  - **Updating**: for $l = 1, 2, \ldots, L$ update the parameters of the network:

    $$w_{kj}^l = w_{kj}^l - \mu \frac{\partial E}{\partial w_{kj}^l} = w_{kj}^l - \mu a_k^{l-1} \delta_j^l \tag{3.15}$$

    $$b_j^l = b_j^l - \mu \frac{\partial E}{\partial b_j^l} = b_j^l - \mu \delta_j^l \tag{3.16}$$

- Backpropagation has the following issues:

– From equation 3.15 it follows that when activations are low, weights change slowly. From equations 3.13 and 3.14 it follows that if $\sigma(z^l) \approx 0$ or $\sigma(z^l) \approx 1$, then $\sigma'(z^l) \approx 0$. In this case the specific neuron is said to be **saturated**. In both of these cases, the considered neuron learns slowly.

– From equation 3.14 it follows that for the first layers in the network, the gradient is the product of many factors of the form $\sigma'(\boldsymbol{z}^l) \leq \frac{1}{4}$ for small values of $\boldsymbol{z}^l$. This means that the first layers learn much more slowly than the last layers. Moreover, if weights are small, the first layer loses most of the input information, hence the last layers learn fast but on a highly deteriorate information. This is called **vanishing gradient problem**.

3.3 Overfitting and Underfitting

Whenever the model at hand is too complex and specialized over the peculiarities of the samples in the training set one has **overfitting**. In particular, whenever the model overfits the training set, it becomes unable to generalize well on unseen data. The opposite problem is called **underfitting**. In order to avoid overfitting one could apply one of the following techniques:

- Collect more data.
- Reduce the model capacity. — Reduce model complexity (this can be achieved by means of ablation techniques, i.e. try to remove parts of the network)
- Early stopping.
- Regularization (e.g. weight-decay)
  - L1: error linear wrt size of parameters
  - L2 error squared ~
  - values added to the loss
- Model averaging.
- Data augmentation.
- Dropout. In particular, during the training process, hidden neurons are randomly disabled with probability p. — similar to train many networks and averaging between them

3.4 Activation and Loss Functions for Classification

- When one is trying to solve a binary classification problem, it is customary to use the **sigmoid function** as the activation function for the output layer of the network:

$$\sigma(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{1+e^x} \tag{3.17}$$

  This function returns a real value in $[0, 1]$. probability

- When one is trying to solve a multi-class classification problem, it is customary to use the **softmax function** as the activation function for the output layer of the network:

$$\text{softmax}(j, x_1, \ldots, x_k) = \frac{e^{x_j}}{\sum_{j=1}^{k} e^{x_j}} \tag{3.18}$$

This function returns a probability distribution. In the binary case, the softmax function is equivalent to the sigmoid function.

- When one is trying to solve a classification problem, it is customary to use the **cross-entropy loss function** in order to compare two probability distributions. This function is based on the **Kullback-Leibler divergence** ($\text{DKL}(P\|Q)$) between two distributions P and Q, which is a measure of the information loss due to approximating P with Q:

not symmetric (thus, it is not a distance)

ground truth

output of the model

$$\begin{aligned}
\text{DKL}(P\|Q) &= \sum_i P(i) \log \frac{P(i)}{Q(i)} \\
&= \sum_i P(i)(\log P(i) - \log Q(i)) \\
&= \sum_i P(i) \log P(i) - \sum_i P(i) \log Q(i)) \\
&= - \underbrace{\mathcal{H}(P)}_{\text{Entropy}} \underbrace{- \sum_i P(i) \log Q(i)}_{H(P,Q)}
\end{aligned} \tag{3.19}$$

In particular, the **cross-entropy** between P and Q is given by:

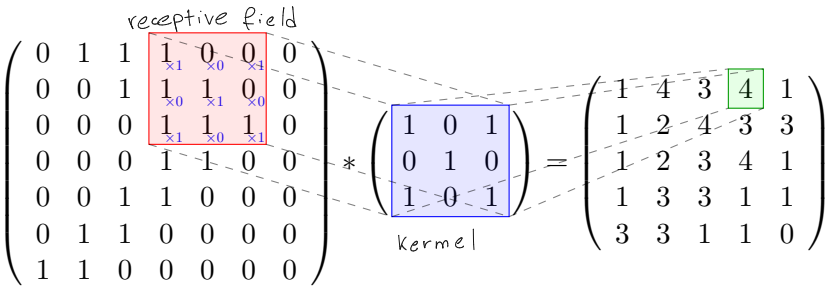
$$\mathcal{H}(P, Q) = - \sum_i P(i) \log Q(i) = \mathcal{H}(P) + \text{DKL}(P\|Q) \tag{3.20}$$

Chapter 4

Convolutional Neural Networks

In convolutional neural networks the activation of a neuron is not influenced from the entire input signal, but only from a small region of such input. The dimension of the input which influences the activation of such neuron is called **receptive field**. In particlaur, every neuron works as a **convolutional filter** which is convolved over the whole input data. For this reason, in a convolutional neural network, the weights between layers are shared.

composition of fielters allows ave to synthesize more cimeor functions of the input



4.1 Convolution and Correlation

- **Convolution** is a mathematical operation which transforms an input matrix by means of another matrix (kernel). This operation can be seen as the transformation of a given function via another function. Considering the 2D case:

$$f(x,y) * k(x,y) = \begin{cases} \int_u \int_v f(x-u, y-v)k(u,v) & \text{(continuous case)} \\ \sum_u \sum_v f(x-u, y-v)k(u,v) & \text{(discrete case)} \end{cases} \tag{4.1}$$

  In particular, convolution is symmetric, associative and distributive.

- **Correlation** is a mathematical operation similar to convolution which does not flip the given kernel. In particular, given a uni-variate function $f(x)$ and a uni-variate kernel $k(x)$ in the interval $[-M, M]$, the 1D convolution between these two is given by:

$$(f * k)(x) = \sum_{m=-M}^{M} f(x-m)k(m)$$

In this case it is possible to notice that $k(-m)$ is the multiplicative factor for $f(x+m)$, that is, the kernel must be flipped before taking the products. This means that convolution, in general, flips the given kernel and then computes the products. If one does not flip the kernel, then one computes the correlation $(f \circ k)$ between the two functions.

- There is no difference between convolution and correlation if the given kernel is symmetric (e.g. a Gaussian kernel) and the difference between the two operations is not so relevant when considering neural networks (the weights are learned by the machine).

4.2 Convolutional Layers

Convolutional layers are defined by the following parameters:

- **Kernel size**, which is the dimension of the linear filter.
- **Stride**, which is the number of input elements to be crossed in consecutive applications of the kernel.
- **Padding**, which is an artificial enlargement of the input to allow the application of filters on borders of the input. The two possible ways of padding are the valid padding, in which no padding is applied, and the same padding, in which one adds the minimal padding enabling the kernel to be applied an integer number of times.
- **Depth**, which is the number of different filters that one wish to synthesize. corresponds to the depth of the next layer

In particular, along each axes the dimension of the output is given by the formula:

$$\frac{W+P-K}{S}+1 \tag{4.2}$$

where W is the dimension of the input, P is the padding, K is the kernel size and S is the stride.

- Given a 1D input of size $W = 7$, a kernel of size $K = 3$, $P = 0$ and $S = 1$, one obtains $(W-K)/S + 1 = (7-3)/1 + 1 = 5$ output values.
- Given a 2D colour image of size $32 \times 32 \times 3$, twelve kernels of size 6×6, $P = 0$ and $S = 2$, one obtains an output whose size is $14 \times 14 \times 12$.

*

4.3 Pooling Layers

In deep convolutions networks, it is common practice to alternate convolutional layers with **pooling layers**, where each neuron simply takes the mean or the maximum value in its receptive field. This practice reduces the dimension of the output. Moreover it allows one to achieve invariance wrt small translations. Examples are max-pooling (non-linear), average-pooling (linear)

* A filter operates on all channels in parallel If the input layer has depth D, and the kernel size is N x M, the actual dimension of the filter will be N x M x D.

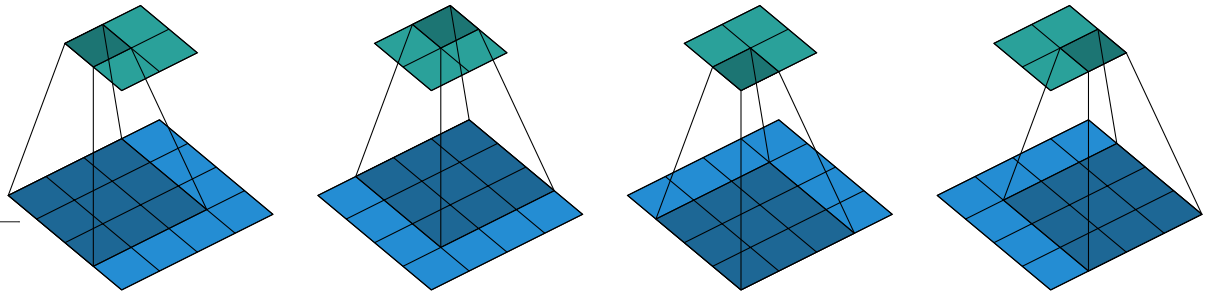
* Examples of CNNs:
- AlexNet
- VGG 16 (very modular and clean structure)
- Inception V3
- ResNet

*

4.4 Backpropagation in CNNs

In order to apply backpropagation in convolutional neural networks, one can proceed as follows:

1. Unroll the input and the output into vectors, from left to right and from top to bottom. Considering the following example



kernel is 3x3

the input has dimension $4 \times 4 = 16$, and the output has dimension $2 \times 2 = 4$. In this case, the operation performed by the convolutional layer can be seen as a single dense network, with sixteen inputs and four outputs.

2. The weights to be updated are the kernel weights. In particular, it is possible to define the weights matrix, $W \in \mathbb{R}^{|input| \times |output|}$, as follows:

$$W = \begin{bmatrix} w_{0,0} & 0 & 0 & 0 \\ w_{0,1} & w_{0,0} & 0 & 0 \\ w_{0,2} & w_{0,1} & 0 & 0 \\ 0 & w_{0,2} & 0 & 0 \\ w_{1,0} & 0 & w_{0,0} & 0 \\ w_{1,1} & w_{1,0} & w_{0,1} & w_{0,0} \\ w_{1,2} & w_{1,1} & w_{0,2} & w_{0,1} \\ 0 & w_{1,2} & 0 & w_{0,2} \\ w_{2,0} & 0 & w_{1,0} & 0 \\ w_{2,1} & w_{2,0} & w_{1,1} & w_{1,0} \\ w_{2,2} & w_{2,1} & w_{1,2} & w_{1,1} \\ 0 & w_{2,2} & 0 & w_{1,2} \\ 0 & 0 & w_{2,0} & 0 \\ 0 & 0 & w_{2,1} & w_{2,0} \\ 0 & 0 & w_{2,2} & w_{2,1} \\ 0 & 0 & 0 & w_{2,2} \end{bmatrix}^\top$$

- Each column corresponds to a different application of the kernel (the specific kernel can be applied only four times in the aforementioned example).
- $w_{i,j}$ is a kernel weight, with i and j being the row and column of the kernel respectively.

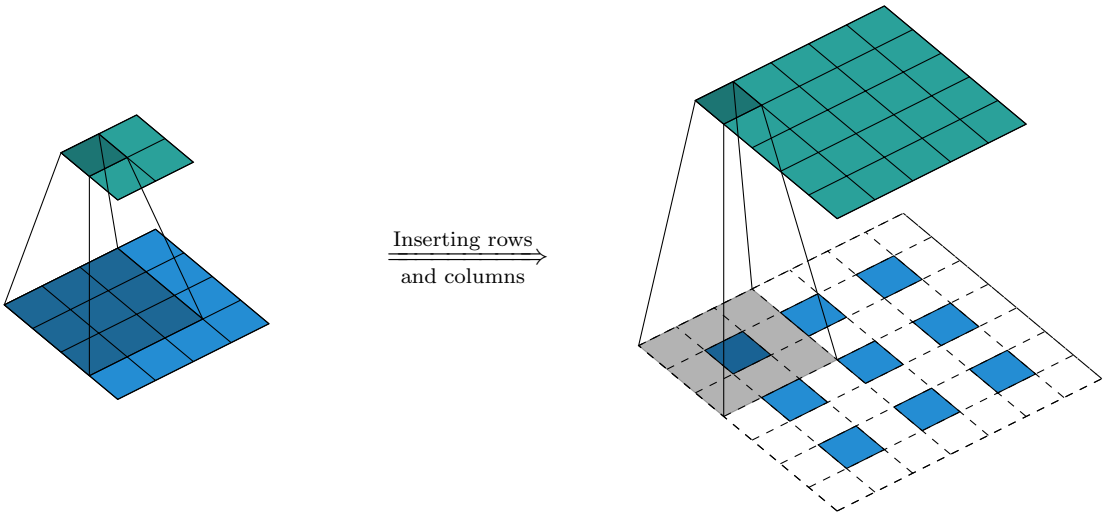
3. The weights are updated using the usual update rule for dense networks.

$$W = \begin{bmatrix} \boxed{w_{0,0}} & 0 & 0 & 0 \\ w_{0,1} & \boxed{w_{0,0}} & 0 & 0 \\ w_{0,2} & w_{0,1} & 0 & 0 \\ 0 & w_{0,2} & 0 & 0 \\ w_{1,0} & 0 & \boxed{w_{0,0}} & 0 \\ w_{1,1} & w_{1,0} & w_{0,1} & \boxed{w_{0,0}} \\ w_{1,2} & w_{1,1} & w_{0,2} & w_{0,1} \\ 0 & w_{1,2} & 0 & w_{0,2} \\ w_{2,0} & 0 & w_{1,0} & 0 \\ w_{2,1} & w_{2,0} & w_{1,1} & w_{1,0} \\ w_{2,2} & w_{2,1} & w_{1,2} & w_{1,1} \\ 0 & w_{2,2} & 0 & w_{1,2} \\ 0 & 0 & w_{2,0} & 0 \\ 0 & 0 & w_{2,1} & w_{2,0} \\ 0 & 0 & w_{2,2} & w_{2,1} \\ 0 & 0 & 0 & w_{2,2} \end{bmatrix}^\top$$

- Updates relative to a same kernel weight must be shared, e.g. taking a mean among all updates.

4.5 Transposed Convolution

In general, normal convolutions with a stride $S \neq 1$ downsample the input dimension. In some cases, one may be interested in upsampling the input (e.g. image-to-image processing, to obtain an image of the same dimension of the input after some compression, or to project feature maps to a higher-dimensional space). A **transposed convolution** (sometimes called **deconvolution**) can be thought as a normal convolution with a sub-unitarian stride. To mimic such stride, one must first properly upsample the input (e.g. by inserting empty rows and columns) and then apply a single strided convolution.



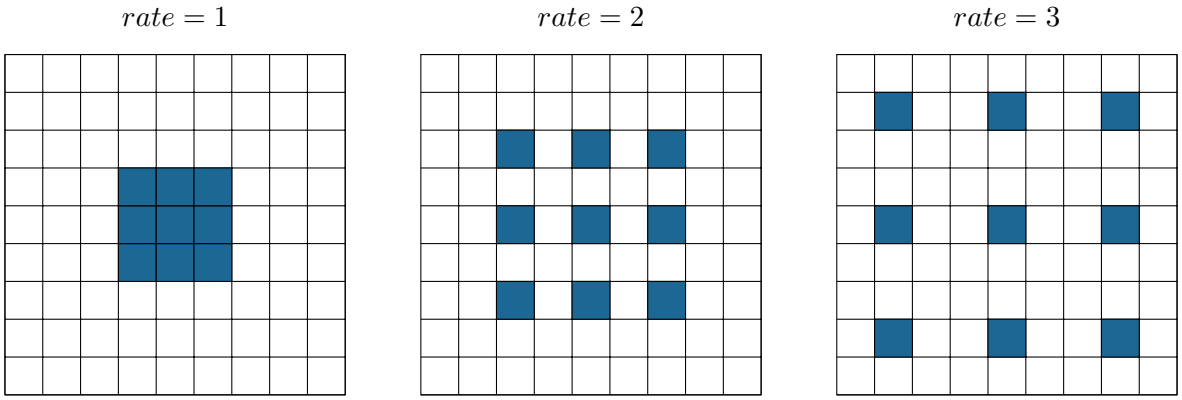
One could also upsample the image by using other tools, instead of adding empty rows and columns.

* Normalization layers:
Automatic normalization for layers of the network (e.g. normalizing the output of layers)
- Batch normalization: the input is normalized based on the batch (moving) statistic.

$BN(x) = \gamma \cdot \frac{x - \mu^B}{\sigma^B} + \beta$ (scale, center), γ and β are learned, μ^B and σ^B are computed.

4.6 Dilated Convolution

Dilated convolutions are obtained by expanding the convolution kernel with holes between its consecutive elements. Applying a dilated convolution is the same as applying a normal convolution but, by dilating a kernel, one is able to enlarge the receptive field of the filter (i.e. to cover a larger ~~are~~ area of the input).



*

4.7 Transfer Learning with CNNs

Usually, when dealing with images and convolutional neural networks, it is possible to transfer knowledge from one network to another. In particular, the filters that have been learned (the most primitive ones) are likely to be independent from the particular kind of images they have been trained on. This means that the filters applied to the first convolutional layers are able to extract low-level features that are applicable across image (e.g. edges, patters, gradients), while the last layers usually identify specific features within an image (i.e. they are not useful when knowledge has to be transferred). In order to transfer knowledge one could:

1. Train a convolutional neural network using a first given dataset X_1.
2. Freeze the early convolutional layers of the network.
3. Train only the last few layers using a second dataset X_2.

This is particularly useful whenever the two problems have similar inputs or whenever $|X_1| \gg |X_2|$.

4.8 How CNNs See the World (gradient ascent on the input)

Each neuron in a convolutional neural network gets activated by specific patterns in the input image. The general intuition is that neurons in higher layers should recognize increasingly complex patterns, obtained as a combination of previous patterns, over a larger receptive field. In the highest layers (i.e. last layers of the network) neurons may start to recognize patterns similar to features of objects in the given images. In particular, it is possible to visualize which patterns are recognized by a specific given neuron by following different procedures:

- The loss function $\mathcal{L}(\theta, x)$ of a network depends on the parameters θ and on the input x. During training, one fixes x and compute the partial derivative of $\mathcal{L}$ with respect to the parameters θ in order to adjust the such parameters. In the same way, one could fix θ and use partial derivatives with respect to x in order to synthesize images which minimize the loss (i.e. to find which images activate a given neuron). This is achieved by using the gradient ascent technique:

  1. Generation of a random image, x. — for example a noise image
  2. Computation of a forward pass using x as input to the network to compute the activation $a_i(x)$ caused by x at some neuron.
  3. Computation of a backward pass in order to compute the gradient of $\partial a_i(x)/\partial x$ of $a_i(x)$ with respect to each pixel of the input image.
  4. Modification of x by adding a small percentage of the gradient $\partial a_i(x)/\partial x$ so to reduce the error.
  5. Repetition of the process until a sufficiently high activation value is obtained.

- Instead of synthesizing an image which maximizes the activation of a neuron, one could emphasize what caused the activation. This process uses the deconvolution operation and the unpooling operation (i.e. inverse of pooling operation). In particular, a general **DeConv Network** works in the following way:

  1. Selection of an image with a strong activation for a neuron.
  2. Zero-ing out all the activations for different neurons.
  3. Computation of the corresponding image, starting from the given feature map, using the given DeConv network (this network uses deconvolution and unpooling in order to go back to image space). By doing so, convolutional kernels must be transposed in order to emphasize the portion of the image that caused the activation.

- One could also try to understand the inner representation at some layer by generating an image indistinguishable from the original one. In this case, the objective is similar to autoencoding. In particular, one progressively adjust the source image by gradient ascent, but instead of optimizing towards a given category, the goal is to minimize the distance from an internal encoding $\Theta_0 = \Theta(x_0)$ of a given image x_0:

$$\text{argmin}_{x \in \mathbb{R}^{H \times W \times C}} \underbrace{\mathcal{L}(\Theta(x), \Theta_0)}_{\text{Loss}} + \underbrace{\lambda \mathcal{R}(x)}_{Regularizer} \tag{4.3}$$

4.9 Fooling CNNs

Gradient ascent can be used to modify an input image so to emphasize the activation of a given neuron. The same technique can be use to increase the classification score of whatever class one wants by synthesizing images of the selected class (data augmentation):

- When synthesizing new images, a tiny consistent perturbation of all the pixels is enough to fool the classifier.
- The same effect can also be achieved by applying an evolutionary technique (i.e. consisting of mutations, perturbations) and forgetting about gradient ascent.
- All classification techniques are vulnerable, since in high dimensional space it is easy to move away from the manifold (i.e. a collection of points forming a certain kind of set) of a given category. This is due to the low-dimensionality manifolds associated with natural images.

4.10 Inceptionism

Inceptionism is achieved as follows:

1. Train the image classification network.
2. Starting from some specific layer, revert the network to slightly adjust (via backpropagation) the original image to improve activation of a specific neuron.
3. After enough iterations, the image will be incepted by the desired features.

4.11 Style Transfer

The gradient ascent technique can also be used to mimic artistic style:

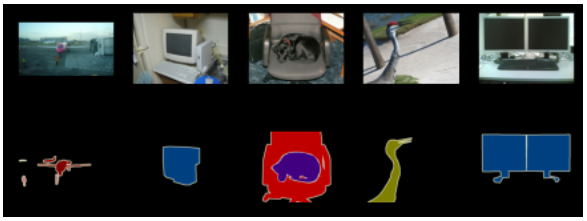
This can be achieved by adding a feature space on top of the original CNN representations, which computed correlations between the different feature maps (channels) at each given layer. In particular:

- At each layer l, an image is encoded with D^l distinct feature maps, F^l_d, each of size M^l ($M = (W \times H)$). The feature map $F^l_{d,x}$ if thus the activation of the filter d at position x in the l-th layer.
- Feature correlations for the given image are given by the Gram matrix, $G^l \in \mathbb{R}^{D^l \times D^l}$, where $G^l_{d_1,d_2}$ is the result of the dot product between the feature maps $F^l_{d_1}$ and $F^l_{d_2}$ in the l-th layer.

dot product computes correlation

4.12 Segmentation

Segmentation is the task of classifying each pixel in an image according to the object category it belongs to. Building supervised training sets is quite expensive, since it requires a complex human operation. Given a dataset, each sample of such dataset, (x, y), is a pair of images. In particular, x is the original image to be segmented, and y is the corresponding segmented image:

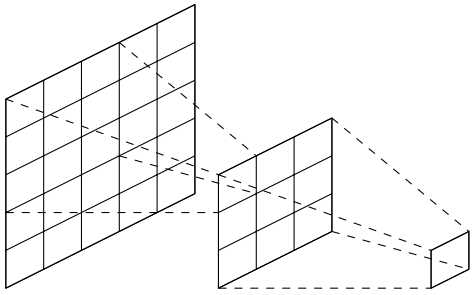


For this reason, semantic segmentation can be regarded as a special case of image-to-image transformation.

4.12.1 Convolutionalization

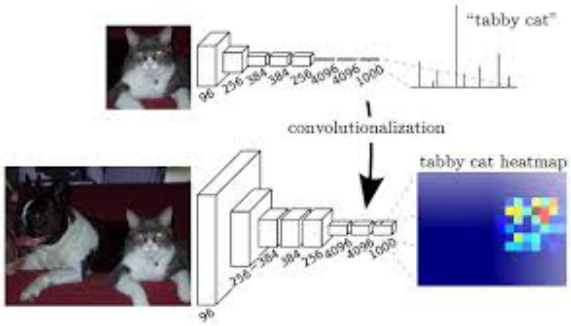
In convolutionalization one is interested in using CNNs in order to segment a given input. In particular:

- The composition of convolutional layers behaves as a convolutional layer. In particular, the stride of the composite convolution is the product of the strides of the the single components.
- For example, supposing one wants to compose two kernels with dimension three and stride one:



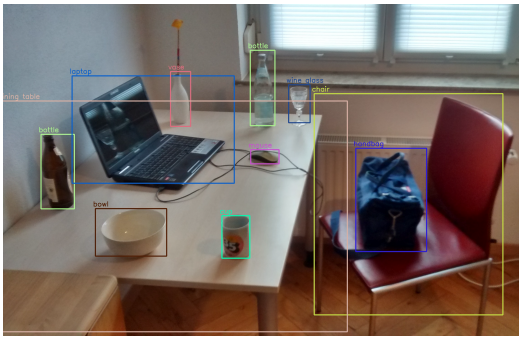
then, the intermediate dimension is $(1-1)*1+3 = 3$, and the initial dimension is $(3-1)*1+3 = 5$. In general, $D_{\text{in}} = S * (D_{\text{out}-1}) + K$.

- Dense layers, usually present as final layers in a convolutional neural network, can be seen as a convolution with a filter whose size if equal to the given input.
- If one starts from an image classification network which works on images of a given, fixed size, one can get a network which takes in input images of arbitrary dimension and produces in output a heatmap of activations of the different objects categories, relative to different locations of the input image.



4.13 Object Detection

Object detection is similar to segmentations, but in this case one must return a boundary box containing the considered object.



Typically, the quality of each individual bounding box is evaluated with respect to the corresponding ground truth using the IoU operator (Intersection over Union):

$$IoU(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{4.4}$$

The more the predicted and true bounding boxes are overlapped, the higher is the obtained IoU value. In particular, there exist two main approaches one could follow:

- **Region Proposal** methods, in which region proposals are usually extracted via selective search algorithms, aimed at identifying possible locations of interest. These algorithms typically exploit the texture and structure of the image, and they are object independent.
- **Single Shot** methods which rely on really fast techniques. An example of such methods is You Only Look Once (YOLO).

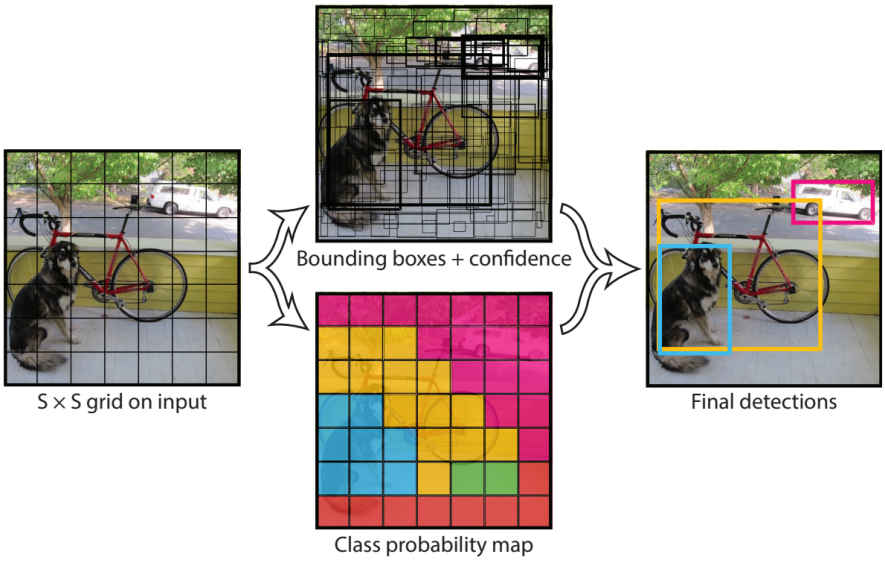
4.13.1 You Only Look Once (YOLO)

YOLO is an object detector which uses features learned by a fully convolutional neural network to detect an object. In particular:

- The input image is divided into an $S \times S$ grid. If the centre of an object falls into a grid cell, that particular grid cell is responsible for detecting the particular object.
- Each grid cell predicts B bounding boxes and confidence scores for the specific boxes. These confidence scores reflect how confident the model is that the box contains an object an also how accurate it thinks the box is that it predicts. Formally, the confidence scores are computed as $P(object) \cdot IoU(truth, pred)$. If no object exists in that cell, the confidence scores should be zero. Otherwise, one wants the confidence score to be equal to $IoU(truth, pred)$.
- Each bounding box consists of five predictions, x, y, w, h and confidence. In particular, the (x, y) coordinates represent the centre of the box relative to the bounds of the grid cell. The width, w, and the height, h, are predicted relative to the whole image. Finally, the confidence prediction represents the IoU between the predicted box and any ground truth box.
- Each grid cell also predicts C conditional class probabilities, $P(Class_i|Object)$. These probabilities are conditioned on the grid cell containing an object. One only predicts one set of class probabilities per grid cell, regardless of the number of boxes B.
- At test time, one multiplies the conditional class probabilities and the individual box confidence predictions:

$$P(Class_i|Object) \cdot P(Object) \cdot IoU(truth, pred) = P(Class_i) \cdot IoU(truth, pred) \tag{4.5}$$

  which gives the class-specific confidence scores for each box. These scores encode both the probability of that class appearing in the box and how well the predicted box fits the object.



- YOLO's loss function consists of two parts, the **localization loss** for bounding box offset prediction and the **classification loss** for conditional class probabilities. In particular, former one is computed as:

training only the neuron in the middle of the box

$$\mathcal{L}_{loc} = \sum_{i=0}^{S^2} \sum_{j=0}^{B} 1_{ij}^{obj} [(x_i - \hat{x}_i)^2 + (y_i - \hat{y_i}^2) + (\sqrt{w_i} - \sqrt{\hat{w}_i})^2 + (\sqrt{h_i} - \sqrt{\hat{h}_i})^2] \tag{4.6}$$

where v is the ground truth value, while $\hat{v}$ is the predicted value, i ranges over cells, j ranges over bounding boxes, 1_{ij}^{obj} is a delta function indicating whether the j-th bounding box of the i-th cell is responsible for the object detection. The classification loss, instead, is computed as:

$$\begin{aligned} \mathcal{L}_{cls} = &\sum_{i=0}^{S^2} \sum_{j=0}^{B} (1_{ij}^{obj} + \lambda_{noobj}(1 - 1_{ij}^{obj}))(C_{ij} - \hat{C}_{ij})^2 \\ &+ \sum_{i=0}^{S^2} \sum_{c \in C} 1_i^{obj} (p_i(c) - \hat{p}_i(c))^2 \end{aligned} \tag{4.7}$$

where λ_{noobj} is a configurable parameter meant to down-weight the loss contributed by background cells containing no objects. Finally, the whole loss is computed as:

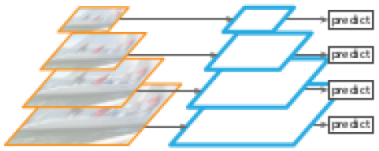
$$\mathcal{L} = \lambda_{coord} \mathcal{L}_{loc} + \mathcal{L}_{cls} \tag{4.8}$$

where λ_{coord} is an additional parameter, balancing the contribution of $\mathcal{L}_{loc}$ and $\mathcal{L}_{cls}$. In YOLO, $\lambda_{coord} = 5$ and $\lambda_{noobj} = 0.5$.

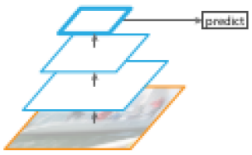
4.13.2 Multi-Scale Processing

Multi-scale processing (predicting something given a feature map) can be achieved in multiple ways.

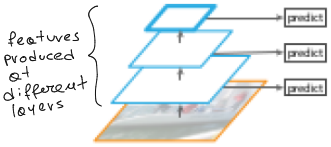
1. Using an image pyramid to build a feature pyramid. Features are computed on each of the image scales independently, which is slow.



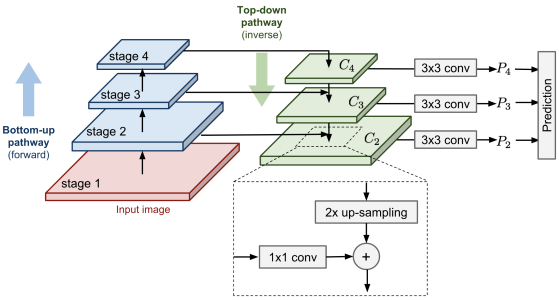
2. The first systems for fast object detection (such as YOLO) opted to use only higher level features at the smallest scale. This usually compromises detection of small objects.



3. An alternative is to reuse the pyramidal feature hierarchy computed by a convolutional network as if it were a featured image pyramid.



4. Modern systems recombine features along a backward pathway. This is as fast as point 2. and 3. but more accurate.



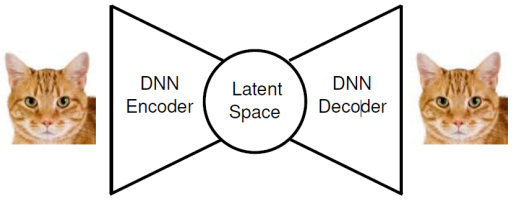
Chapter 5

Autoencoders

it is not possible to synthesize images of high level categories starting from a classifier (the space occupied by images that make sense is negligible wrt the dimension of the entire data space. This allows one to fool the classifier)

indeed, autoencoders are able to lower the dimensionality

An autoencoder is a network trained to reconstruct input data out of a learned internal representation.

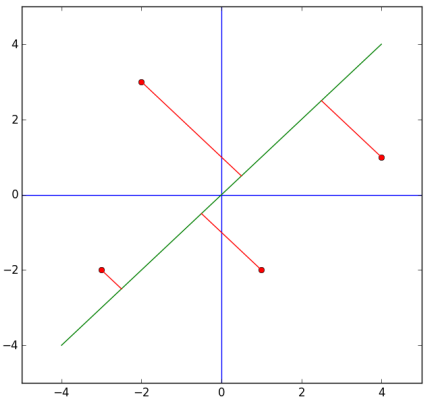


Usually, the internal representation has lower dimensionality with respect to the input. In general, data compression is possible because one exploits regularities (i.e. correlations) in the features describing input data. If the input has a random structure no compression is possible. In particular:

- If the internal layer has fewer units of the input, autoencoders can be seen as a form of data compression. This compression is:
  - **Data-specific**, i.e. it only works well on data with strong correlations (e.g. digits, faces).
  - **Lossy**, i.e. the output id degraded with respect to the input.
  - **Directly trained** on unlabelled data samples (self-supervised training).

× 5.1 Principal Component Analysis

To understand how autoencoding works it is convenient to look at Principal Component Analysis (PCA), a traditional statistical technique for dimensionality reduction. In particular, given points in a d-dimensional space, the objective of PCA is to minimize the quadratic error of their reconstruction (e.g. find the best projection on a line of some bi-dimensional data, find the best planar approximation of a three-dimensional image). In general, minimizing the projection error is equivalent to maximize the variance.



In particular, in order to maximize the variance, the following concepts are needed:

- An eigenvector of a matrix A is the vector v such that for some scalar λ:

$$Av = \lambda v \tag{5.1}$$

- Given a dataset $X \in \mathbb{R}^{N \times D}$, the covariance matrix of X is the matrix in $\mathbb{R}^{D \times D}$ given by:

$$var(X) = \mathbb{E}[(X - \mathbb{E})^T (X - \mathbb{E})] \tag{5.2}$$

  If the dataset is centred, i.e. $\mathbb{E} = 0$, then:

$$var(X) = \mathbb{E}[X^T X] = \frac{1}{n} X^T X \tag{5.3}$$

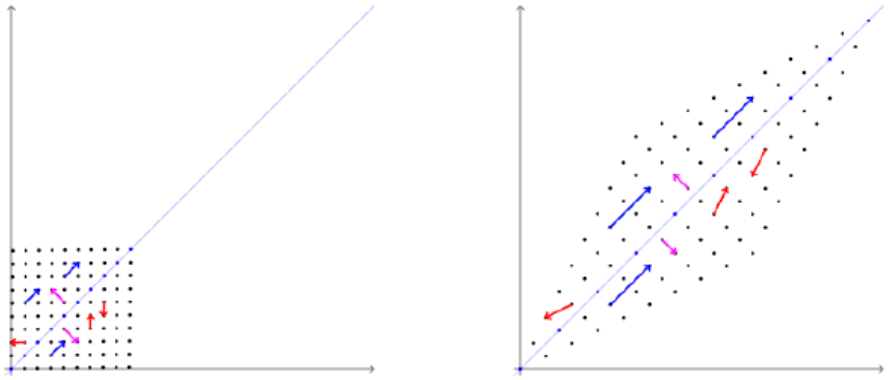
  For example:

$$X = \begin{bmatrix} -3 & -2 \\ -2 & 3 \\ 1 & -2 \\ 4 & 1 \end{bmatrix} \quad \Rightarrow \quad X^T X = \begin{bmatrix} 30 & 2 \\ 2 & 18 \end{bmatrix}$$

  Such matrix is always symmetric and expresses the deformation of the given data, i.e. the way data are distributed in space.

- Every square matrix A defines a linear transformation. In particular, every symmetric matrix defines a scaling along particular directions orthogonal to each other. In general, the direction of some vectors is preserved, while some other directions are not preserved. The preserved directions are precisely the ones defined by the eigenvectors of A. For example:

$A = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$



The eigenvectors of A are $v_1 = [1, 1]^T, v_2 = [1, -1]^T$, which corresponds to the orthogonal scaling directions, and the corresponding eigenvalues are 3 and 1, which corresponds to the intensities of the scaling along the considered directions. In this case, v_1 is the **principal component** (its corresponding eigenvalue is the greatest one) and v_2 is the **secondary one**.

- In order to maximize the variance, one can fit an ellipsoid over the training data, and project over the main axes. These axes are just the eigenvectors of the covariance matrix of the given dataset. As a matter of fact, the eigenvectors are the axes of the ellipsoid conceptually fitting the given data.

The PCA algorithm is the following:

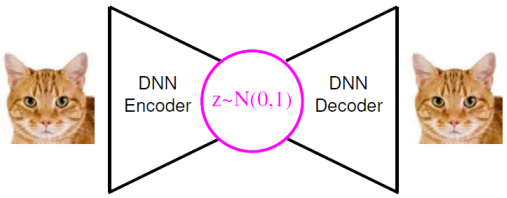
1. Put data in a matrix X.
2. Normalize X so that the mean along each dimension is null (in order to obtain $\mathbb{E} = 0$).
3. Compute the covariance matrix $\Sigma = X^T X$.
4. Find the eigenvectors of Σ and the corresponding eigenvalues.
5. Keep the m eigenvectors with largest eigenvalues.

By taking a fully connected linear network with k hidden units, no activation functions, and quadratic reconstruction error as objective function, one may compute the k principal components (weights) and their respective projections (hidden values). Moreover, **deep belief networks** are deep, non-linear, networks which try to reduce the dimensionality of a given image, using the best possible representation (this can be achieved by applying principal components analysis at each layer).

5.2 Variational Autoencoders

Variational autoencoders are generative models which allow one to generate data by sampling in the latent space (i.e. to generate data which is similar to the data at hand), even though usually one does

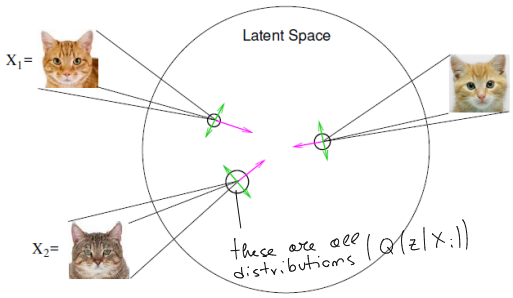
not know the distribution of the latent variables. By using variational autoencoders one tries to force latent variables to have a known distribution:



In particular:

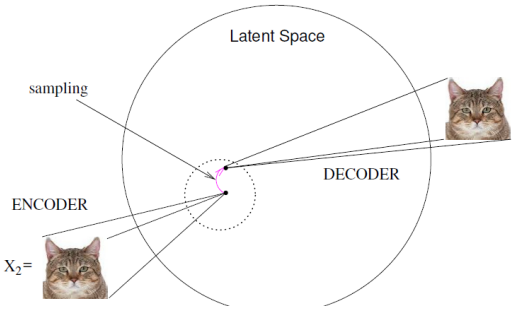
- Given an image, X_i, and the encoder part of an autoencoder, Q, one is able to project X_1 onto the latent space through Q, obtaining $Q(z|X_i)$, which can be set equal to a Gaussian distribution $G(\mu(X_i), \sigma(X_i))$. Usually, each data-point whose projection onto the latent space is near the projection of X_i will be similar to X_i. Different X_i gets modelled using different μ and σ.
- The overall idea of variational autoencoders is to first compute $\mu_z(X)$ and $\sigma_z(X)$ (using the encoder) for each data-point X and each latent variable z. Then, a regularization term is added to the loss function in order to:
  - Push $\mu_z(X)$ towards zero (loss is $\mu_z(X)^2$), i.e. move the projections of each X to the centre of the latent space.
  - Push $\sigma_z(X)$ towards one (loss is $\sigma_z(X)^2 - \log(\sigma_z^2(X)) - 1$), i.e. enlarge the projections of each X.

(effects of the KL loss)

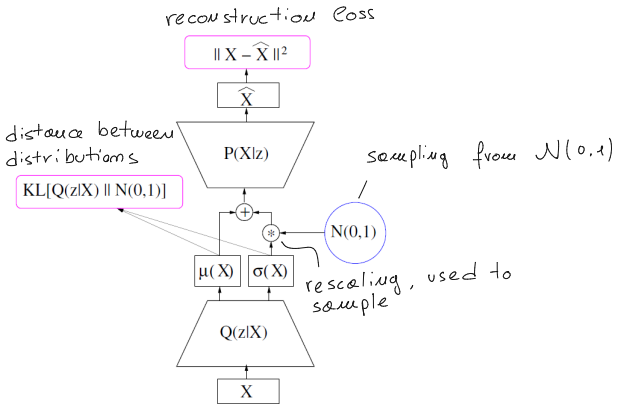


The effect of the regularization term is to induce a Gaussian distribution of points in the latent space.

- Given a sample X, the encoder part of the autoencoder should predict $\mu(X)$ and $\sigma(X)$. Then, it is possible to sample around $\mu(X)$ using the computed variance $\sigma(X)$ and compute the reconstruction of the point X. The reconstruction error is finally used to tune both $\mu(X)$ and $\sigma(X)$.

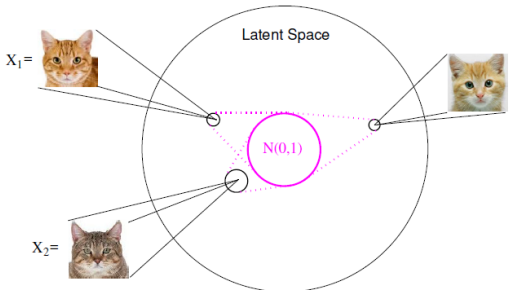


- The full picture is the following:



where $KL(Q(z|X)\|\mathcal{N}(0,1))$ is the distance between the $Q(z|X)$ distribution and the normal distribution $\mathcal{N}(0,1)$. In particular, the actual distribution of latent variables is the marginal distribution $Q(z)$, and hopefully:

$$Q(z) = \sum_X Q(z|X) \approx \mathcal{N}(0,1) \tag{5.4}$$



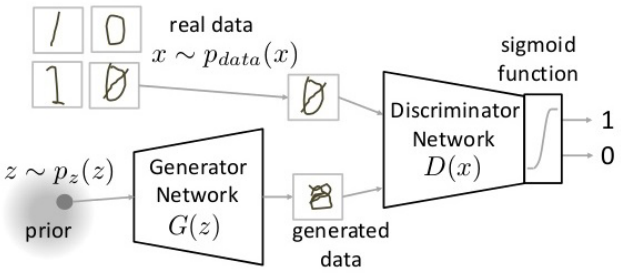
- Once the network has been trained using backpropagation (i.e. adjusting $\mu_z(X)$ and $\sigma_z(X)$) one is able to sample $z \sim \mathcal{N}(0,1)$ (i.e. to generate a point in the latent space) and use the decoder in order to produce a new data-point which is similar to the ones at hand.

Chapter 6

Generative Adversarial Networks

A generative adversarial network (GAN) is a generative model such as variational autoencoders. In general, a **generative model** is a model which tries to learn the actual distribution of the data from the available samples. The goal of generative model is, thus, to build a probability distribution, p_{model}, close to actual distribution of the data, p_{data}.
The output of generative models is intrinsically multimodal (e.g. generating a car, there are a lot of possible outputs). If one bases learning on minimizing an average distance between all the elements in the training set and the set of expected features, one could end up with exceedingly blurred images. In particular:

- The generative adversarial network approach is based on game theory. The players of the game are the two parts of the network: the **generator** and the **discriminator**.



- The generator, G, and the discriminator, D, play a minimax game, $Min_G Max_D V(D,G)$, where:

$$\text{loss} \quad -V(D,G) = \mathbb{E}_{x\sim p_{\text{data}}(x)}[\log D(x)] + \mathbb{E}_{z\sim p_z(z)}[\log(1 - D(G(z)))] \tag{6.1}$$

  In particular, $\mathbb{E}_{x\sim p_{\text{data}}(x)}[\log D(x)]$ is the negative cross-entropy of the discriminator with respect to the true data distribution, while $\mathbb{E}_{z\sim p_z(z)}[\log(1 - D(G(z)))]$ is the negative cross entropy of the false discriminator with respect to the fake generator. From a high-level point of view, the

generator network generates an image and tries to fool the discriminator, which should be able to tell if the received data is a generated image or a real data image.

- The training process consists in training the discriminator, by freezing the generator, and training the generator, by freezing the discriminator. The algorithm is the following:

---

for number of training iterations **do**

for k steps **do**

- Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
- Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
- Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right].$$

end for

- Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
- Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right).$$

end for

The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

- The fact that the discriminator get fooled does not mean that the fake generated by the generator is good. As a matter of fact, neural networks can be easily fooled.

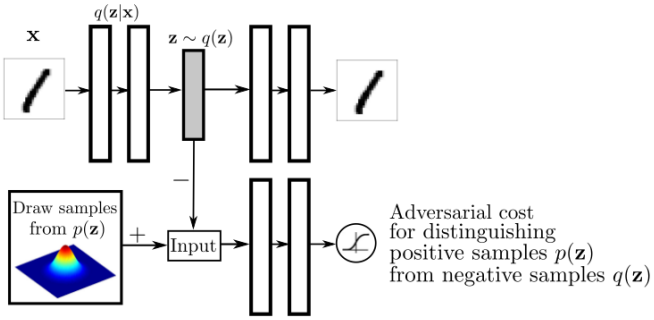
6.1 Combinations of VAEs and GANs

Both VAEs and GANs have problems as generative models:

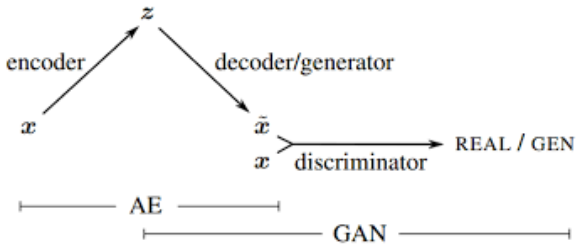
- When using VAEs, the similarity metric is crucial. Pixel-wise metrics, like the squared error, are too sensible to local perturbations (e.g. small translations or rotations).
- When using GANs, the generative approach has problems to capture the real data distribution (learning is difficult and unstable).

To avoid these problems, two main approaches are possible:

- Acting in the latent space. In particular, this approach works by replacing the KL-divergence with a discriminator network. The whole network should match the aggregated posterior distribution $Q(z)$ with the expected (arbitrary) prior distribution p_z ($P(z)$), in order to fool the discriminator. These networks are called **adversarial autoencoders**.

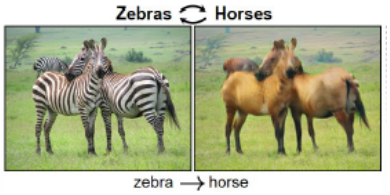


- Acting in the visible space. In particular, this approach works by replacing the reconstruction loss with a discriminator network. This network tries to distinguish the original image from the reconstructed image. These networks are called **VAE-GAN**s.



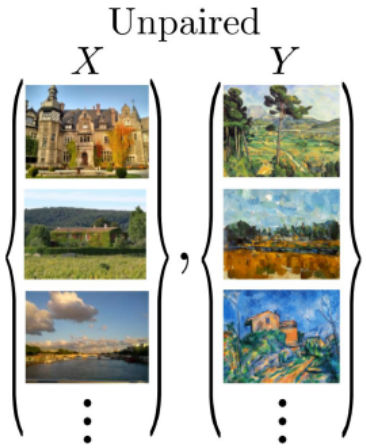
6.2 Cycle GANs

Cycle GANs allow unpaired image-to-image translation (this can be seen as a stylistic transformation). For example, given an image of two zebras, one can obtain the same image but with two horses, preserving the content of the image:



In order to do so:

- Training is done using images divided into two classes. These images should not be paired. For example:



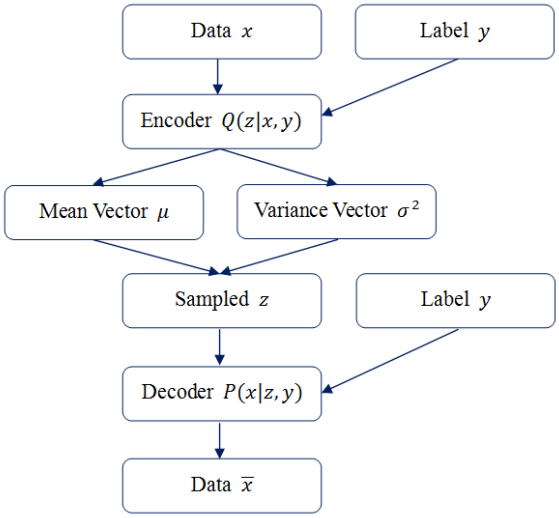
- A generator and a discriminator network play the minimax game. The generator tries to produce samples from the desired distribution, based on an input of the other distribution, and the discriminator tries to predict if the sample belongs to the actual distribution or it was produced by the generator. However, this cannot guarantee that the generated image is related to the original image.
- To this aim, the **cycle-consistency** constraint has been added: if one transforms from the source distribution to the target distribution and then back again to the source distribution, one would expect to obtain the original image.
- Given the mappings $G : X \to Y$ and $F : Y \to X$, and discriminators D_Y and D_X, the loss function of such networks is composed of two parts:

$$\begin{aligned} \text{Adversarial loss:} \quad & \mathbb{E}_{y \sim p_{\text{data}}(y)}[\log(D_Y(y))] + \mathbb{E}_{x \sim p_{\text{data}}(x)}[\log(1 - D_Y(G(x)))] \\ \text{Cycle consistency loss:} \quad & \mathbb{E}_{x \sim p_{\text{data}}(x)}(\|F(G(x)) - x\|_1) + \mathbb{E}_{y \sim p_{\text{data}}(y)}(\|G(F(y)) - y\|_1) \end{aligned}$$

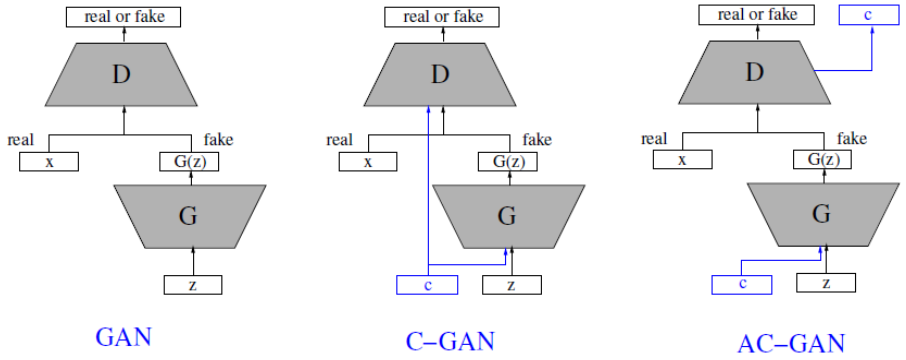
6.3 Conditional Generation

When using a generative model one would like to parametrize the generation according to specific attributes (e.g. generate a given digit or the face of an old man wearing glasses).

- **Conditionl VAE**s (CVAE) are networks in which both the decoder $Q(z|X)$ and the decoder $P(X|z)$ are parametrized with respect to a given condition c: $Q(z|X, c)$ and $P(X|z, c)$.



- **Conditional GAN**s are networks in which both the generator and the discriminator take in input the condition. **Auxiliary Classifier GAN**s, instead, try to classify with respect to different conditions in addition to compute a true/fake discrimination.

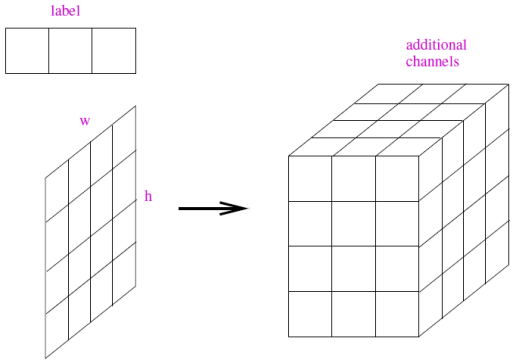


In general, in conditional networks, one passes the label / condition as an additional input:

- If one needs to add such label to a dense layer, one just needs to concatenate the label to the input vector.
- If one needs to add such label to a convolutional layer, one can perform vectorization or Feature-wise Linear Modulation (FILM).

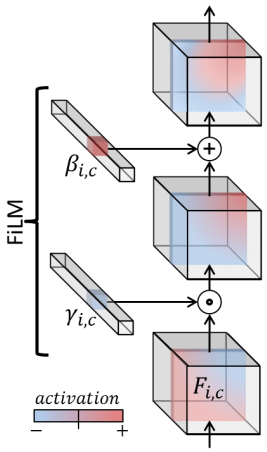
6.3.1 Vectorization

This method consists in repeating the label (typically in categorical form) for every input neuron, and stack them as new channels.



6.3.2 Feature-wise Linear Modulation

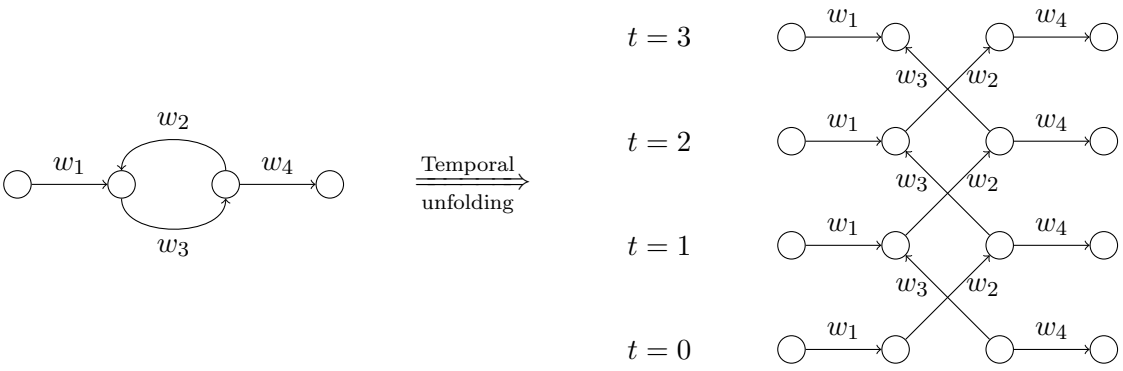
This method consists in using the condition to give a different weight to each feature (i.e. to each channel). In particular, the vectors γ and β, whose size is equal to the channels of the layer, are generated from the condition. Then, the layers are rescaled by these two vectors.



Chapter 7

Recurrent Neural Networks

A recurrent neural network (RNN) is an artificial neural network where connections between neurons form a directed graph along a temporal sequence. These networks allow sequence-to-sequence processing. In order to define the semantics of a recurrent neural network one can temporally unfold the network as shown below:

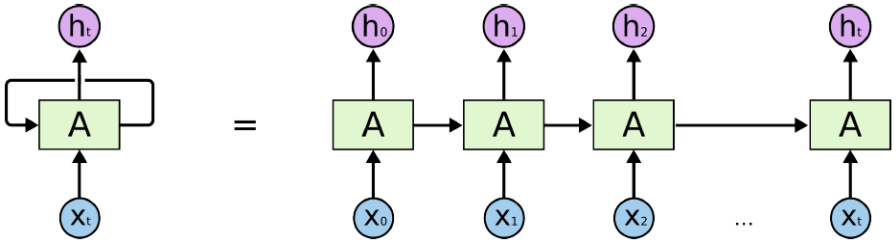


In particular, the recurrent network is just a layered network that keeps re-using the same weights through time. Moreover:

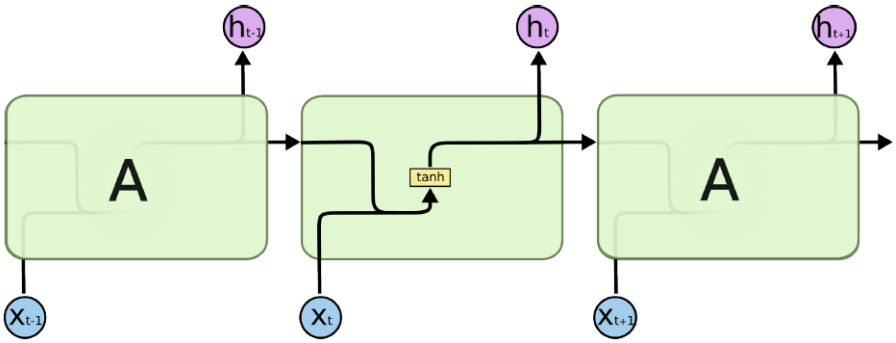
- The backpropagation algorithm is modified in order to incorporate **equality constraints** between weights: one computes the gradients as usual, and then average gradients so that they induce a same update. For example, to constraint $w_i = w_j$, one can compute $\partial E/\partial w_i$ and $\partial E/\partial w_j$ and use $\partial E/\partial w_i + \partial E/\partial w_j$ to update both w_i and w_j.
- The hidden neurons activations at time $t = 0$ need to be initialized. These can be either initialized manually or they can be treated as parameters and learned in the same way as the weights.

7.1 Long-Short Term Memory (LSTM)

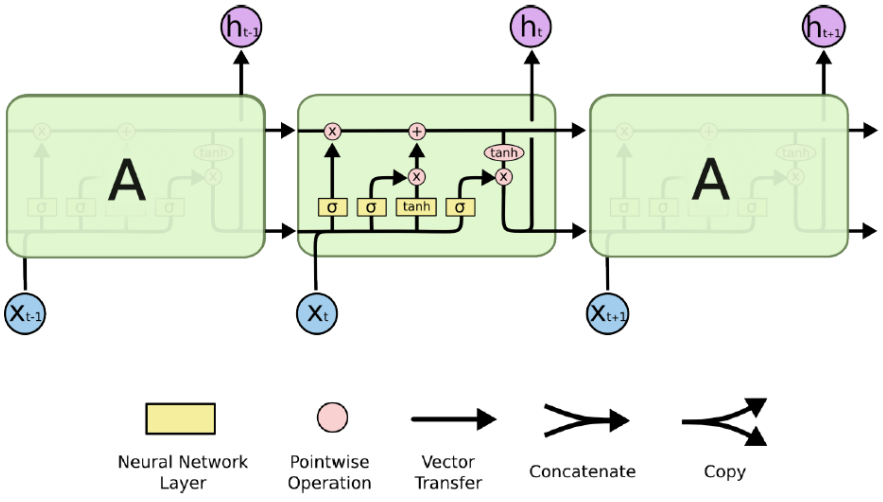
In this section, each RNN is depict using the temporally unfolded representation, i.e. a forward link between two units must be seen as a looping connection:



In a traditional RNN, the content of the memory cell C_t and the input x_t are combined in order to produce the output h_t, which coincides with the new content of the cell C_{t+1}.

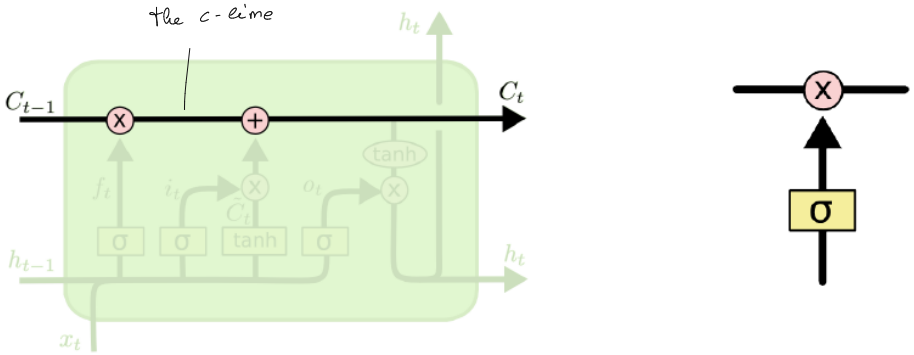


The overall structure of a LSTM is different:

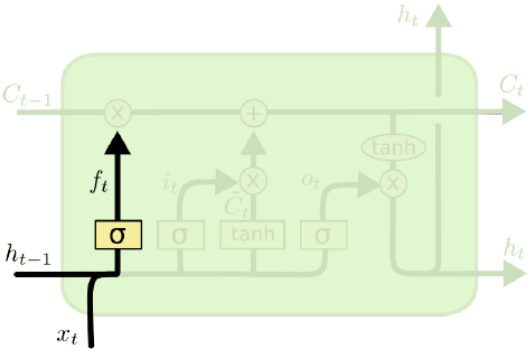


In particular:

- The LSTM has the ability of removing or adding information to the cell state, in a way regulated by suitable **gates**. Gates are a way to optionally let information through. In particular, the product with a sigmoid neural network layer simulates a boolean mask.



- The **forget gate** decides what part of the memory cell to preserve. In particular, this gate computes the following operation:

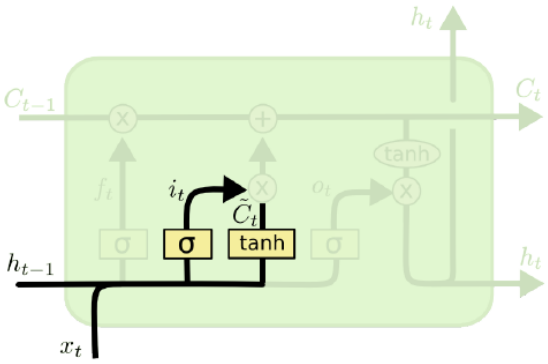


concatenation

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \qquad (7.1)$$

current weights

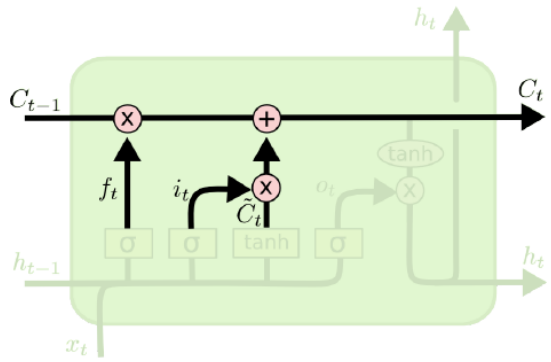
- The **input gate** decides what part of the input to preserve. The tanh layer creates a vector of new candidate values $\tilde{C}_t$ to be added to the state:



$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \qquad (7.2)$$

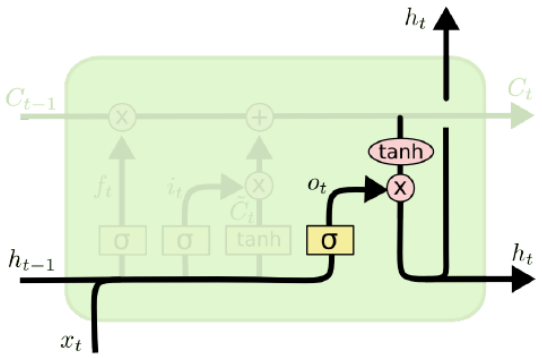
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C) \qquad (7.3)$$

- Cell updating consists in multiplying the old state by the boolean mask f_t, and then adding $i_t * \tilde{C}_t$:



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t \tag{7.4}$$

- The **output gate** computes h_t by filtering the content of the cell. In particular, such gate decides what parts of the cell state to output. The tanh is used to re-normalize values in the interval $[-1, 1]$:



$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \tag{7.5}$$
$$h_t = o_t * \tanh(C_t) \tag{7.6}$$

7.2 Attention

In general, attention is the ability to focus on different parts of the input, according to the requirements of the problem being solved. From the point of view of neural networks, one would expect the attention mechanism to be differentiable, so that one can learn where to focus by standard backpropagation techniques. In particular:

- Attention mechanisms can be implemented as gating functions. The gating maps are dynamically generated by some neural network, allowing to focus on different part of the input at different times. An example of attention mechanisms are the forget map, the input map and the output map in LSTM.
- The most typical attention layer is based on a **key-value** paradigm, implementing a sort of associative memory. This memory is then accessed with **queries** to be matched with keys. The resulting **scores** generate a boolean map that is used to weight values. In particular:

1. For each key k_i a score a_i is generated as follows:

$$a_i = \alpha(\boldsymbol{q}, k_i) \tag{7.7}$$

how similar q and k_i are

2. The **attention weights** are obtained using the softmax function:

$$\boldsymbol{b} = \text{softmax}(\boldsymbol{a}) \tag{7.8}$$

generating a distribution

3. The weighted sum of the values is then returned:

$$o = \sum_{i=1}^{n} b_i v_i \tag{7.9}$$

weighting probabilities

In many applications, values are also used as keys (self-attention).

- Different score functions lead to different attention layers. In particular, the two main approaches are based on:
  - The dot product, in which the query and the key must have the same dimension d:

computes correlation

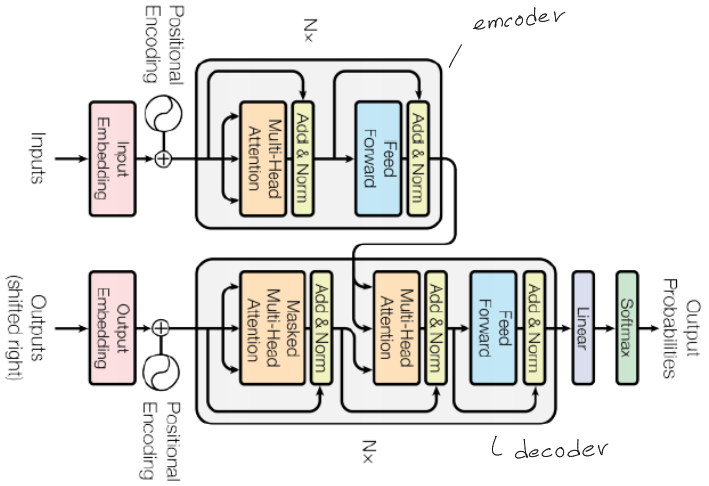
$$\alpha(\boldsymbol{q}, \boldsymbol{k}) = \boldsymbol{q} \cdot \frac{\boldsymbol{k}}{\sqrt{d}} \tag{7.10}$$

  - Multi-layer perceptrons, where α is computed by a neural network, usually composed of a single layer:

$$\alpha(\boldsymbol{q}, \boldsymbol{k}) = \tanh(W_q \boldsymbol{q} + W_k \boldsymbol{k}) \tag{7.11}$$

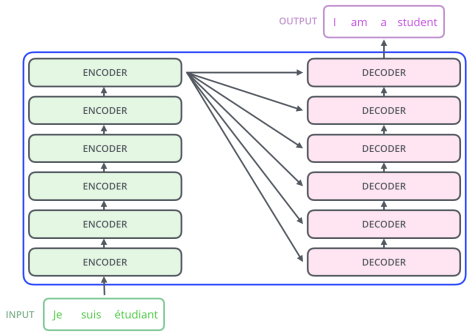
7.3 Transformers

The general structure of a transformer is the following:

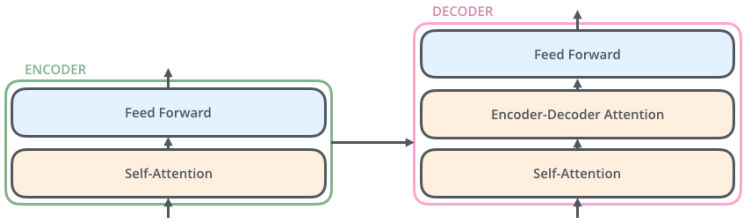


In particular:

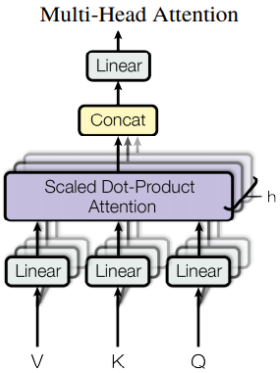
- A transformer has a traditional encoder-decoder structure, with connections between them. Moreover, the encoder and the decoder are both composed of multiple components:



- Each encoder element is organized as a self-attention layer, followed by a feed-forward component. Each decoder element is similar, with an additional attention layer that helps the decoder focusing on relevant parts of the input sentence:



- Multi-head attention expands the model's ability to focus on different positions, for different purposes. As a result, multiple representation subspaces are created, focusing on potentially different aspects of the input sequence.



- Positional encoding is added to word embeddings to give the model some information about the relative position of the words in the sentence. The positional information is a vector of the same dimensions, d_{model}, of the word embedding. As positional encoding, the authors proposed the sine and cosine functions of different frequencies:

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right) \tag{7.12}$$

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{(2i+1)/d_{\text{model}}}}\right) \tag{7.13}$$

  According to the this encoding, each dimension i of the positional encoding vector, PE, corresponds to a sinusoid, where the wavelengths form a geometric progression from 2π to $10000 \cdot 2\pi$.

Chapter 9

Reinforcement Learning

Reinforcement learning problems are problems involving an **agent** interacting with an **environment** which provides numeric **reward**. At each time step t:

1. The agent is in state s_t and selects the **action** a_t according to some **policy** $\pi(a_t|s_t)$. In particular, a policy $\pi(a|s)$ is a probability distribution of actions given states.
2. The environment answer with a local reward r_t.
3. The agent enters into a new state s_{t+1}.

In order to solve such problems, one wants to learn the best way to act, i.e. the best policy. This is achieved by maximizing the **future cumulative reward**:

$$R = \sum_{i \geq 1} r_i \tag{9.1}$$

Moreover, one could also take into account the fact that distant rewards are less likely than close ones (these are more predictable). To this aim, one can express R by weighting each local reward r_i with a discount factor γ:

$$R = \sum_{i \geq 1} \gamma^i r_i \tag{9.2}$$

The reinforcement learning paradigm can be mathematically modelled using the concept of **Markov Decision Process**. In particular:

- A Markov Decision Process is characterized by the **Markov property**, which states that the current state completely characterizes the state of the world (i.e. future actions only depend on the current state).
- A Markov Decision Process is defined by a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{P}, \gamma)$, where:
  - $\mathcal{S}$ is the set of possible states. At time step $t = 0$, the environment is is state s_0.

- $\mathcal{A}$ is the set of possible actions.
- $\mathcal{R}$ is the reward probability given a (s, a) pair. After each action, the environment samples $r_t \sim \mathcal{R}(r_t|s_t, a_t)$.
- $\mathcal{P}$ is the transition probability to the next state given a (s, a) pair. After each action, the environment samples $s_{t+1} \sim \mathcal{P}(s_{t+1|s_t,a_t})$. In particular, if the learning model needs to learn the probability $\mathcal{P}(s_{t+1|s_t,a_t})$, then the model is called **model-based**, otherwise it is called **model-free**.
- γ is the discount factor.

Moreover, a policy produces trajectories of the form $s_0, a_0, r_1, s_1, a_1, r_2, s_2, \ldots$ The optimal policy, π^* is given by:

$$\pi^* = \text{argmax}_\pi \, \mathbb{E} \sum_{t \geq 0} \gamma^t r_t \tag{9.3}$$

where the average is taken over all the possible trajectories. In order to be able to find the optimal policy, all the techniques have to deal with exploration / exploitation trade-off. In particular, there exist two basic techniques:

- The **value-based** approach, in which one tries to evaluate each state s with a value function $V(s)$. In this case, the policy is implicit: at each step, one should choose the action taking to the state whose evaluation is the best.
- The **policy-based** approach, in which one tries to directly optimize the policy (remember that $a = \pi(s)$ is the probability of performing a in state s).

9.1 Value-Based Approaches

These approaches are based on two evaluation functions:

- An evaluation function for states:

  this would require model-based knowledge of $P(s'|s,a)$

  $$V(s) = \mathbb{E}_{s_0=s} \sum_{t \geq 0} \gamma^t r_t \tag{9.4}$$

  where V measures how good a state is.

- An evaluation function for actions:

  $$Q(s, a) = \mathbb{E}_{s_0=s, a_0=a} \sum_{t \geq 0} \gamma^t r_t \tag{9.5}$$

  where Q measures how good is action a for state s. It is possible to compute V from Q ($V(s) = \sum_a \pi(a|s)Q(s, a)$) but not vice-versa. Moreover, the optimal Q-value function, $Q^*(s, a)$, is the maximum expected cumulative reward achievable from state s performing action a:

$$Q^*(s,a) = \max_\pi \mathbb{E}_{s_0=s, a_0=a} \sum_{t \geq 0} \gamma^t r_t \tag{9.6}$$

The optimal policy π^* consists in taking the best action in any state as specified by Q^*.

Another important concept is introduced by the **Bellman equation**:

- The Bellman equation expresses a relation between the solution for a given problem in terms of the solutions for its sub-problems. According to this concept:

$$Q^*(s,a) = \mathbb{E}_{s'}[r_0 + \gamma \max_{a'} Q^*(s',a')] \tag{9.7}$$

Indeed, $R_{s'} = \max_{a'} Q^*(s',a') = V^*(s')$ is the optimal future cumulative reward from s', and the optimal cumulative reward from s when taking action a is given by $r_0 + \gamma R_{s'}$.

- Since Q^* satisfies the Bellman equation, the main idea is to use the above-mentioned formulation in order to perform an iterative update on progressive approximations, Q_i, of Q^*:

$$\underbrace{Q^{i+1}(s,a)}_{\text{next estimation}} = \underbrace{Q^i(s,a)}_{\text{current estimation}} + \alpha \underbrace{(r_0 + \gamma \max_{a'} Q^i(s',a') - Q^i(s,a))}_{\text{recursive update}} \tag{9.8}$$

where α is the learning rate. The recursive update is the derivative of the quadratic distance between $Q^i(s,a)$ and $r_0 + \gamma \max_{a'} Q^i(s',a')$ (these two quantities should become equal).

9.1.1 Q-Learning

The **Q-learning** algorithm works in the following way:

> Initialize the Q-table.
> Repat until termination of the episode:
> Choose action a in current state s according to the current Q-table.
> Perform action a and observe reward r and new state s'.
> Update the table: $Q(s,a) = Q(s,a) + \alpha(r_0 + \gamma \max_{a'} Q(s',a') - Q(s,a))$.

- In order to perform an update, all the information is contained in a transition tuple (s,a,r,T,s'), where s is the current state, a is the computed action, r is the obtained reward, T is a boolean stating the termination of the episode, s' is the new state. In particular, such tuples are collected by exploring the environment and can be saved into an **experience replay buffer**. Since such algorithm only needs these tuples, it is an **off-policy** technique (i.e. does not rely on any policy).

- At start, the Q-table is not informative. Taking actions according to it could introduce biases and prevent exploration. In particular, in early stages one may be interested in privileging random exploration, and start relying more on the Q-table when more experience is acquired.
- In order to achieve this, one can specify an exploration rate ϵ, initially equal to one. This variable represents the rate of steps that are chosen randomly. This variable is progressively reduced during training.

The above-mentioned points allows one to introduce the **ϵ-greedy Q-learning** algorithm:

Initialize the Q-table, the replay buffer D and ϵ ($\epsilon = 1$).
Repeat for the desired number of episodes:
  Initialize state s.
  Repat until termination of the episode:
    Choose a random move a with probability ϵ, otherwise $a = \max_a Q(s, a)$.
    Perform action a and observe reward r and new state s'.
    Store transition (s, a, r, T, s') into D.
    Sample a random mini-batch of transitions (s, a, r, T, s') from D.
    For each transition in the extracted mini-batch:

$$R = \begin{cases} r & \text{if } T \\ r + \gamma \max_{a'} Q(s', a') & \text{if not } T \end{cases}$$

    $Q(s, a) = Q(s, a) + \alpha(R - Q(s, a))$
    Decrement ϵ.

9.1.2 Deep Q-Learning (DQN)

The major drawback of standard Q-learning algorithm is that $Q^i = Q^*$ when $i \to \infty$. This means that $Q(s, a)$ must be computed for every state-action pair. Computing such Q-table is impossible. To this aim, **deep Q-learning** use a function approximator (a neural network) to estimate the optimal action-value function:

$$Q(s, a, \theta) \approx Q^*(s, a) \tag{9.9}$$

where θ are the function parameters to be learned. Instead of taking action a and state s as input, in deep Q-learning is customary to take only the state s and return a value for each possible action a.

- Given a state, action pair (s, a), the current Q-value estimate of the network is $Q(s, a, \theta)$.
- The expected value, given by the Bellman equation is $\mathbb{E}_{s'}[r_0 + \gamma \max_{a'} Q(s', a', \theta)]$.
- The loss function one tries to minimize is the following:

$$\mathcal{L}(\theta) = (\mathbb{E}_{s'}[r_0 + \gamma \max_{a'} Q(s', a')] - Q(s, a, \theta))^2 \tag{9.10}$$

- In order to train the network, one need to rely on the experience buffer in order to get transition samples.

Deep Q-learning can also be improved using the following techniques:

- **Fixed Q-targets**. In order to compute the loss in DQN, the same neural network is used to provide two different estimations of the Q function ($\underbrace{r_0 + \gamma \max_{a'} Q(s', a', \theta)}_{\text{approximated target } Q^*} - \underbrace{Q(s, a, \theta)}_{\text{current estimation}}$). At every step of training, the Q value shifts but also the target value shifts. The solution to this problem is to use a separate network, $\overline{Q}$, with **fixed parameters** for estimating the target:

$$\underbrace{r_0 + \gamma \max_{a'} \overline{Q}(s', a', \theta)}_{\text{approximated target } Q^*} - \underbrace{Q(s, a, \theta)}_{\text{current estimation}} \tag{9.11}$$

  Then, one can periodically copy the parameters of Q into $\overline{Q}$, in order to update the target network.

- **Double Q-learning**. In order to approximate the target action value in DQN, the maximum over actions is considered ($\underbrace{r_0}_{\text{local reward for taking action } a} + \gamma \underbrace{\max_{a'} Q(s', a', \theta)}_{\text{max } Q \text{ value over all possible actions}}$). Since the approximation is noisy, it is possible to prove that this will eventually result in a positive bias, finally resulting in an over-estimation of the correct value. The solution to this is to decouple the choice of the action from its estimation, using two different networks, Q^A and Q^B:

  Initialize Q^A, Q^B and s.
  Repeat:
    Choose a using ϵ, Q^A, Q^B. Observe r and s'.
    Choose between procedures $UPDATE\text{-}A$ and $UPDATE\text{-}B$.
    If $UPDATE\text{-}A$: (use Q^A to choose the action and Q^B to estimate the action)
      $a^* = \text{argmax}_a\, Q^A(s', a)$
      $Q^A(s, a) = Q^A(s, a) + \alpha(r + Q^B(s, a^*) - Q^A(s, a))$
    Else if $UPDATE\text{-}B$: (use Q^B to choose the action and Q^A to estimate the action)
      $a^* = \text{argmax}_a\, Q^B(s', a)$
      $Q^B(s, a) = Q^B(s, a) + \alpha(r + Q^A(s, a^*) - Q^B(s, a))$
    s = s'
  Until end

If one uses Q^A to select the best action a^*, the value of $Q^A(s', a^*)$ could be biased by this choice. Since Q^B is updated on the same problem, but with different set of experience samples, $Q^B(s', a^*)$ provides a better, unbiased estimate for the value of action a^*.

- **Prioritized experience replay**. This technique exploits the idea that some experience samples may be more important than others, and thus should be replayed more frequently. Thus, a higher priority is given to transitions for which there is a large difference between prediction and expected target. In particular, considering a transition $t = (s, a, r, F, s')$, the relative update is:

$$\delta_t = r + \gamma \max_{a'} Q(s', a') - Q(s, a) \tag{9.12}$$

and the relative priority is:

$$p_t = |\delta_t| \tag{9.13}$$

Stochastic prioritization can be applied. In particular, the probability of being chosen for a replay is computed as:

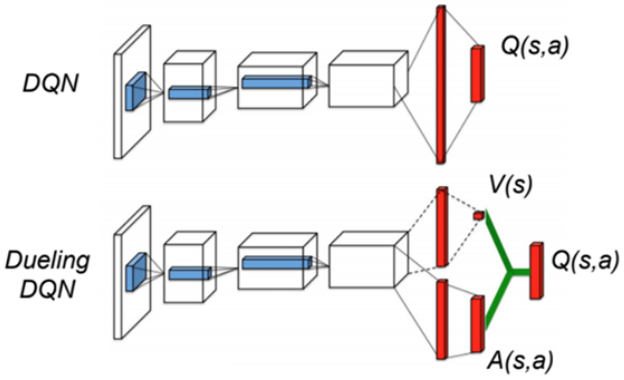
$$p_t = \frac{p_t^\alpha}{\sum_t p_t^\alpha} \tag{9.14}$$

If $\alpha = 0$ all the transitions have the same probability. If α is large, then transitions with high priority are privileged. However, by introducing priorities, one may over-fit over the small portion of experience that one presumes to be interesting. In order to avoid this, one can weight every p_t. In particular, is some transition has a high probability, the related weight will be reduced.

- **Dueling**. Each Q value $Q(s, a)$ estimates how good it is to take action a in state s. One can decompose $Q(s, a)$ into the sum of two terms:
  - $V(s)$, the value of being in state s.
  - $A(s, a)$, the **advantage** of taking action a in state s. This function measures how much better is to take action a with respect to all other possible actions.

$$Q(s, a) = \underbrace{V(s)}_{\text{value}} + \underbrace{A(s, a)}_{\text{advantage}} \tag{9.15}$$

In particular, dueling network architectures (DDQN) split the computation of $V(s)$ and $A(s, a)$ into two different streams:



Intuitively, the dueling architecture can learn which states are valuable, without having to learn the effect of each action for each state. This is useful in states where actions do not affect the environment in any relevant way or, conversely, it allows focusing on the advantage without caring for the current evaluation of the state.

- **Noisy networks**. These networks are characterized by noisy dense layers, combining a deterministic and a noisy stream:

$$y = \underbrace{b + Wx}_{\text{usual layer}} + \underbrace{(b_{noisy} \odot \epsilon^b + (W_{noisy} \odot \epsilon^W)x)}_{\text{noisy stream}} \tag{9.16}$$

where W, b, W_{noisy} and b_{noisy} are learned parameters, while ϵ^b and ϵ^W are randomly generated. In particular, this type of layer is used instead of any standard dense layer. The purpose of this technique it to augment the randomicity in the choice of actions. This allows the network to randomly explore the environment at different rates in different parts of the state space.

- **Distributional reinforcement learning** (DRL). This technique tries to learn the probability distribution of the future cumulative reward, instead of the modelling the expectation of the reward. Specifically, DRL addresses the random reward Z (random variable) whose expectation is the value Q. In particular:

$$Z^\pi(s,a) = \sum_{t \geq 0} \gamma^t r_t \big|_{s_0=s, a_0=a, \pi} \tag{9.17}$$

Similarly to the Bellman equation ($Q(s,a) = R(s,a) + \gamma Q(s', a^*)$):

$$Z(s,a) = R(s,a) + \gamma Z(s', a^*) \tag{9.18}$$

9.1.3 SARSA

The Q-learning is an off-policy technique algorithm is a one-step method since it updates the action value $Q(s, a)$ toward the one-step return $r + \gamma \max_{a'} Q^i(s', a')$. This only directly affacts the value of the state action pair (s, a) that led to the reward. The values of the state action pairs are affected only indirectly through the updated value $Q(s, a)$. This can make the learning process slow since many updates are required to propagate a reward to all the relevant preceding states an actions. To this aim, on-policy techniques try to improve the current policy by sampling long trajectories according to the current strategy. In particular, **State-Action-Reward-State-Action** (SARSA) is a learning algorithm very similar to Q-learning. The SARSA update is the following:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha[r_t + Q(s_{t+1}, a_{t+1} - Q(s_t, a_t))] \tag{9.19}$$

Instead of considering the best action at time $t + 1$ (greedy choice) one considers the actual action a_{t+1} under the current policy. Moreover, Q-learning is based on single step transitions (s_t, a_t, r_t, s_{t+1}), while SARSA is based on mini-trajectories, composed of two steps $(s_t, a_t, r_t, s_{t+1}, a_{t+1})$. For this reason, SARSA is considered on-policy.

9.2 Policy Gradients

Given a class of parametrized policies $\Pi = \{\pi_\theta\}$, for each policy π_θ it is possible to define the value of it:

$$J(\theta) = \mathbb{E} \sum_{t \geq 0} \gamma^t r_t \tag{9.20}$$

The optimal policy is the one associated with θ^*:

$$\theta^* = \text{argmax}_\theta \, J(\theta) \tag{9.21}$$

θ^* can be learned via gradient ascent.

9.2.1 REINFORCE Approach

There exist several different approaches to policy gradient ascent. In particular, for a given and sampled trajectory, standard REINFORCE updates the policy parameters in the direction $\nabla_\theta \log \pi(a_t|s_t, \theta) R_t$. The problem of this is that the raw value of a trajectory is not that meaningful (e.g. if rewards are all positive, one keeps pushing probabilities of all actions). What matters the most is whether a reward is better or worse than expected. To solve this, a **baseline**, dependent on the state, is introduced. Thus, the new estimator becomes:

$$\nabla_\theta \log \pi(a_t|s_t, \theta)(R_t - b(s_t)) \tag{9.22}$$

An good choice for such baseline is the value function of the state, $b(s_t) = V^\pi(s_t)$. In particular, this approach can be seen as an **actor-critic** architecture where the policy π is the actor and the value function (i.e. the baseline) is the critic.

9.2.2 A3C and A2C

The **Asynchronous Advantage Actor Critic** (A3C) pseudo-code is the following:

Algorithm S3 Asynchronous advantage actor-critic - pseudocode for each actor-learner thread.

// Assume global shared parameter vectors θ *and* θ_v *and global shared counter* $T = 0$
// Assume thread-specific parameter vectors θ' *and* θ'_v
Initialize thread step counter $t \leftarrow 1$
repeat
  Reset gradients: $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$.
  Synchronize thread-specific parameters $\theta' = \theta$ and $\theta'_v = \theta_v$
  $t_{start} = t$
  Get state s_t
  repeat
    Perform a_t according to policy $\pi(a_t|s_t;\theta')$
    Receive reward r_t and new state s_{t+1}
    $t \leftarrow t + 1$
    $T \leftarrow T + 1$
  until terminal s_t **or** $t - t_{start} == t_{max}$
  $R = \begin{cases} 0 & \text{for terminal } s_t \\ V(s_t, \theta'_v) & \text{for non-terminal } s_t \text{// Bootstrap from last state} \end{cases}$
  for $i \in \{t-1, \ldots, t_{start}\}$ **do**
    $R \leftarrow r_i + \gamma R$
    Accumulate gradients wrt θ': $d\theta \leftarrow d\theta + \nabla_{\theta'} \log \pi(a_i|s_i;\theta')(R - V(s_i;\theta'_v))$
    Accumulate gradients wrt θ'_v: $d\theta_v \leftarrow d\theta_v + \partial (R - V(s_i;\theta'_v))^2 / \partial \theta'_v$
  end for
  Perform asynchronous update of θ using $d\theta$ and of θ_v using $d\theta_v$.
until $T > T_{max}$

The **Advantage Actor Critic** (A2C) is similar to A3C but without asynchronous agents. Essentially, it is a single-worker variat of A3C. Empirically, A2C produces comparable performance with A3C while being more efficient.

9.2.3 TRPO and PPO

When passing from a given policy π_k (e.g. a randomized version of the current policy) to a new policty π, small modifications can easily result in large fluctuations in behaviour and performance. The goal here is to take the biggest possible improvement step on a policy, without stepping so far that one accidentally cause performance collapse. In particular, let π_θ denote a policy with parameters θ:

- The TPRO update consists in:

$$\theta_{k+1} = \text{argmax}_\theta \, \mathcal{L}(\theta_k, \theta) \text{ such that } \overline{KL}(\theta \| \theta_k) \leq \delta \tag{9.23}$$

where $\mathcal{L}(\theta_k, \theta)$ is the **surrogate advantage**, which represents a measure of how policty π_θ performs with respect to the old policy π_{θ_k} using data from the old policy:

$$\mathcal{L}(\theta_k, \theta) = \mathbb{E}_{s,a\sim\pi_{\theta_k}} \frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s,a) \tag{9.24}$$

and $\overline{KL}(\theta\|\theta_k)$ is an average KL-divergence between policies across states visited by the old policy:

$$\overline{KL}(\theta\|\theta_k) = \mathbb{E}_{s\sim\pi_{\theta_k}} KL(\pi_\theta(\cdot|s)\|\pi_{\theta_k}(\cdot|s)) \tag{9.25}$$

However, such theoretical update is hard to implement.

- PPO achieves a similar objective of TRPO by updating policies via:

$$\theta_{k+1} = \text{argmax}_\theta \, \mathbb{E}_{s,a\sim\pi_{\theta_k}}[L(s,a,\theta_k,\theta)] \tag{9.26}$$

where:

$$L(s,a,\theta_k,\theta) = \min\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s,a), g(\epsilon, A^{\pi_{\theta_k}}(s,a))\right) \tag{9.27}$$

and:

$$g(\epsilon, A) = \begin{cases} (1+\epsilon)A & \text{if } A \geq 0, \text{ i.e. positive advantage} \\ (1-\epsilon)A & \text{if } A < 0, \text{ i.e. negative advantage} \end{cases} \tag{9.28}$$