Deep learning

Francesco Farinola

June 2021

Contents

1 Neural networks **4**
1.1 Expressiveness . . . 5
1.2 Training . . . 6
1.3 Gradient Descent technique . . . 6
1.4 Optimizers . . . 7
1.4.1 Stochastic Gradient Descent . . . 7
1.4.2 Momentum . . . 7
1.4.3 Nesterov momentum . . . 8
1.5 Backpropagation algorithm . . . 9
1.6 Learning issues . . . 11
1.6.1 Vanishing gradient problem . . . 12
1.7 Overfitting . . . 12
1.8 Activation and loss functions for classification . . . 13
1.8.1 Sigmoid . . . 13
1.8.2 Softmax . . . 13
1.8.3 Loss functions . . . 14

2 Convolutional Neural Networks **15**
2.1 Convolution in images . . . 15
2.2 Gaussian Smoothing . . . 16
2.3 Convolutional layer . . . 17
2.4 Pooling layer . . . 18
2.5 Real CNNs . . . 19
2.6 Residual Learning . . . 21
2.7 Backpropagation for CNNs . . . 22
2.8 Transposed convolutions . . . 22
2.9 Dilated (aka Atrous) Convolutions . . . 22
2.10 Transfer Learning . . . 23
2.11 How CNNs see the world . . . 24
2.11.1 Deconvolution . . . 25
2.11.2 Unpooling . . . 25
2.11.3 Another approach . . . 26

2.12 Fooling Neural Networks . . . 26
2.12.1 Manifold . . . 27
2.13 Inceptionism . . . 28
2.14 Style Transfer . . . 28

3 Segmentation **30**
3.1 Semantic segmentation . . . 30
3.2 Convolutionalization . . . 30
3.2.1 Fully Convolutional Netowrk . . . 31
3.3 U-net . . . 32

4 Object Detection **33**
4.1 Jaccard Index (aka Intersection over Union) . . . 33
4.2 YOLO . . . 34
4.3 Multi Scale processing . . . 37
4.3.1 Image Pyramids . . . 37
4.3.2 Feature Pyramids . . . 38
4.3.3 Featurized Image Pyramid . . . 38
4.4 Non Maximum Suppression . . . 39
4.5 Ablation . . . 39

5 AutoEncoders **41**
5.1 Principal Component Analysis PCA . . . 41
5.2 Deep Belief Networks . . . 43
5.3 Variational Autoencoder . . . 43
5.3.1 Kullback-Leibler regularization . . . 45

6 Generative Models **46**
6.1 Generative Adversarial Network GAN . . . 46
6.2 Combining GAN and VAE . . . 48
6.3 Cycle GANs . . . 51
6.4 Conditional generation . . . 52
6.4.1 Conditional VAE - CVAE . . . 53
6.4.2 Conditional GANs . . . 53
6.4.3 InfoGAN . . . 54
6.5 Concrete handling of the condition . . . 54
6.5.1 Vectorization . . . 54
6.5.2 Feature-wise Linear Modulation - FILM . . . 55

7 Recurrent Neural Networks - RNN **56**
7.1 Why training RNNs is difficult . . . 57
7.2 Long-Short Term Memory - LSTM . . . 58

8 Attention **61**
8.1 Squeeze and Excitation . . . 61
8.2 Key-value paradigm . . . 62
8.3 Alignment . . . 63

9 Transformers **64**
9.1 Structure . . . 64
9.2 Self-Attention . . . 65
9.3 Multi-head attention . . . 65
9.4 Residual connection . . . 65
9.5 Positional encoding . . . 66
9.6 Sinusoids at work . . . 66

10 Geometric Deep Learning **67**
10.1 Graph convolutions . . . 67
10.2 Graph Convolutional Networks . . . 68
10.3 Spectral-based approach . . . 68
10.4 Spatial-based approach . . . 70
10.4.1 GraphSage . . . 70
10.4.2 Mixture Model Network - MoNet . . . 71

11 Reinforcement Learning **73**
11.0.1 Future cumulative reward . . . 73
11.0.2 Markov Decision process . . . 73
11.1 Value-based approaches - Qlearning . . . 74
11.1.1 Epsilon greedy strategy . . . 76
11.2 Deep Q-learning . . . 79
11.2.1 Atari Q-learning . . . 80
11.3 Improvements . . . 80
11.3.1 Fixed Q-targets . . . 80
11.3.2 Double Q-learning . . . 81
11.3.3 Prioritized Experience Replay . . . 82
11.3.4 Dueling . . . 83
11.3.5 Noisy Networks . . . 84
11.3.6 Distributional RL . . . 84
11.3.7 Rainbow . . . 86
11.4 Policy Gradient techniques . . . 86
11.4.1 SARSA . . . 87
11.5 Policy gradients . . . 88
11.5.1 The Actor-critic architecture . . . 88
11.5.2 Trust Region Policy Optimization & Proximal PO . . . 90

1 Neural networks

Each neuron takes multiple inputs and produces a single output. Each neuron implements a logistic regressor $\sigma(wx+b)$. The activation function is responsible for threshold triggering.

Motivations behind neural computation: to investigate a different paradigm of computation, very far from traditional programming languages; to solve practical problems difficult to address with algorithmic techniques, useful even if the brain works in a different way.

If the network is acyclic, it is called **feedforward**. If it has cycles it is called **recurrent**.

If there is more than one hidden layer the network is **deep**. Otherwise it is called **shallow** network. Weights W_k are the **parameters** of the model and they are learned in training phase.

Hyperparameters: learning rate, number of neurons and the way they are connected, batch-size, number of epochs.

Dense layer: each neuron at layer k-1 is connected to each neuron at layer k. A single neuron *vectorized* to produce m outputs in parallel: $I^n * W^{nxm} + B^m = O^m$. Dense layers usually work on **flat unstructured** inputs. The order of the elements is irrelevant.

Convolutional layer: each neuron at layer $l-1$ is connected via a parametric kernel to a fixed subset of neurons at layer k. The kernel is convolved over the whole previous layer.

1. move the kernel K over a portion M of the input of equal size
2. compute the dot product $M * K$ and possibly add bias
3. shift the kernel and repeat

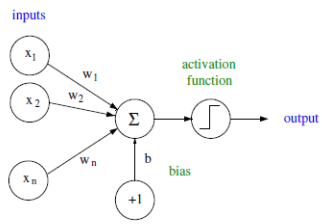
The dimension of the output only depends on the number of times the kernel is applied. Input is **structured**.

- **Signals**: raw input collected from sensors
- **Data**: meaningful but not focused
- **Features**: meaningful and focuses

Discovering good features is a complex task.

Deep learning exploits a hierarchical organization of the learning model, allowing complex features to be computed in terms of simpler ones, through non-linear transformations.

People is aiming to a more **Green AI**. Emphasis on efficiency as well as performance.



$$output = \begin{cases} 1 & if \sum_i w_i x_i + b \geq 0 \\ 0 & otherwise \end{cases} \qquad output = \begin{cases} 1 & if \sum_i w_i x_i \geq -b \\ 0 & otherwise \end{cases}$$

1.1 Expressiveness

He talks about perceptron only when there is an activation function of binary threshold (output only 0 or 1). The bias set the position of threshold.

The **hyperplane** $\sum_i w_i x_i + b = 0$ divides the space in two parts: to one of them (above) the perceptron gives value 1, to the other value 0. Above and below can be inverted just by inverting parameters.

With a perceptron we can implement the NAND function (Negation of AND) which is logically complete. If we connect gates able to compute NANDs, we are able to compute any boolean function.

Single layer perceptrons are not complete. We can't draw a straight line in XOR case.

We can't recognize patterns with a perceptron since each pixel should individually contribute to the classification which is not this case. The first pixel does not give us any additional information on the classification of the image because the network need to consider the second pixel. In linear model we are able to give importance to more feature than others but we cannot explore the relation between them.

In DL preprocessing is still important but not as linear techniques because DL are able to derive additional features by themselves.

Perceptrons are not complete because we need to compose them and consider Multi-Layer perceptrons (which are logically complete).

Important points: shallow nets are already complete; with **deep networks**, the same function may be computed with less neural units - by means of 2 layers, we already achieve completeness - we know that by means of normalization we can compute whatever formula and they are 'flat'.

Activation functions play an essential role, since they are the only source of nonlinearity, and hence the **expressiveness** of NNs - because if we stack linear functions we get a linear function, so we do not improve the expressiveness of the net.

We know that a step is located at the inflection point $x = -b/w$ and the steepness varies with w. For very high w we get bumps which can be approximated to a step function so we need to know only where is the step (bias). But generally weights are small to avoid gradients to be zero everywhere and infinite

step (satured network).

We can approximate any function by sums of bumps and choose very narrow bumps, then use bias to choose whether output is 0 or 1.

Every continuous function $R \to [0, 1]$ can be approximated by NNs.

1.2 Training

Training is a technique that allow the network to find by itself the best params configuration to solve a particular problem.

We have a **loss function** $L = L(x, \theta)$ depending on data x and parameters θ. The goal is to minimize it and loss tells us how much we are far from the goal.

We cannot change data (**fixed data set**) but we can **change the parameters** of the model.

Initially: randomly perturb weights and see if we get better results, if so save the change, else discharge. This is very inefficient and has high probability to make things worse.

Instead of making random adjustment we can predict it by using **derivatives** (tangent of the angle α.

The sign of the derivative provides **orientation.** It is positive if $\alpha < 90°$ and negative if $90° < \alpha < 180°$. If the derivative is positive we must decrease the parameter, if it is negative we must increase it. The **magnitude** is related to the steepness of the tangent: if close to 0 the angle is flat, if high the angle is almost right.

Binary threshold is not good for learning since the derivative is 0 everywhere, and infinite in correspondence of the jump.

The vector of all partial derivative (for each parameter) is called **gradient** of the function. The gradient points in the direction of steepest descent. Each partial derivative tells us how the surface changes if we mode in that direction - we move in the opposite direction.

1.3 Gradient Descent technique

1. Start with a completely random configuration of the parameters
2. Compute the gradient of the loss function
3. Make a small step in the direction opposite to the gradient
4. Iterate from step 2 until the loss is sufficiently small

It may *end up in local minima* or get *lost in plateau.* Only guaranteed if the surface is concave. In general the function is not convex but it practice it is not a major problem (dunno why).

Problems are: what is a small step? When should we stop iterating?

Learning rate is the dimension of the step in the direction of the gradient $w = w - \mu \nabla L(w)$ - it can be configured by the user and its evolution is governed

by **optimizers**. In general, we start with a high learning rate and decrease it as we reach the solution.

When fitting a line, constant values in a derivative do not influence at all, multiplicative factor have a small impact since they just scale the gradient (same impact of changing the learning rate).

The analytic solution only works in the linear case, and for fixed error functions. The backpropagation method can be generalized to multi-layer non-linear networks.

1.4 Optimizers

1.4.1 Stochastic Gradient Descent

In general, gradient should be computed on the whole training set, only in this case we have an improvement.

How often to update weights:

- **Online:** for each training sample - forward/backward for every sample, gradient zig-zags around the direction of the steepest descent
- **Full-batch:** full sweep through training data - in general a waste of time - once per step
- **Mini-batch:** for a small random set of training case - small but representative, the direction could be less precise but in the limit case it converges to the same value of the full-batch case - once for mini-batch

If data is very large, it is difficult to work with mini-batch, so we use online. The backward pass is more expensive than the forward pass so it is better to avoid making it very often.

1.4.2 Momentum

If, during consecutive training steps, the gradient seems to follow a stable direction, we could improve its magnitude, simulating the fact that it is acquiring a **momentum** along that direction, similarly to a ball rolling down a surface.

The hope is to reduce the risk to get stuck in a local mini- mum, or a plateau. No theoretical justification.

Momentum corrects the update at time t with the fraction of the update at time $t-1$. Calling v^t the vector of updates at time t, we have:

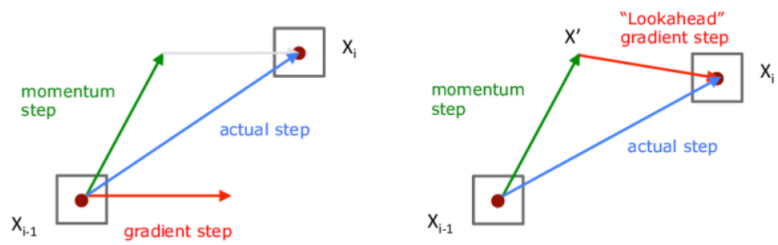
$$v^t = \mu * \nabla L(w) + \alpha * vt - 1 \tag{1}$$

where first term is the gradient step and second one is momentum.

1.4.3 Nesterov momentum

Nesterov momentum is a variant of the previous technique. The difference is just the position at which the gradient is computed: before or **after the momentum step**.

We first make a big jump in the direction of the previous accumulated gradient and then measure the gradient where you end up and make a correction.



Both methods are distinct only when the learning rate μ is reasonably large. When the learning rate μ is relatively large, Nesterov Accelerated Gradients allows larger decay rate α than Momentum method, while preventing oscillations.

1.5 Backpropagation algorithm

The **backpropagation algorithm** is simply the instantiation of the gradient descent technique tot he case of neural networks.

It provides iterative rules to compute partial derivatives of the loss function with respect to each parameter of the network.

We need the **chain rule**:

- **Univariate case**

  Given two derivable functions f, g with derivatives f' and g', the derivative of the composite function $h(x) = f(g(x))$ is $h'(x) = f'(g(x)) * g'(x))$ Equivalently, letting $y = g(x)$,

$$h'(x) = f'(g(x)) * g'(x) = f'(y) * g'(x) = df/dy * dg/dx \tag{2}$$

- **Multivariate case**

  Given a multivariable function $f(x, y)$ and two single variable functions $x(t)$ and $y(t)$ we get:

$$\frac{d}{dt} f(x(t), y(t)) = \frac{\partial f dx}{\partial x dt} + \frac{\partial f dy}{\partial x dt} = \nabla f * v'(x) \tag{3}$$

  where ∇f is the gradient of f, the vector of partial derivatives.

Logistic units at layer l compute the function

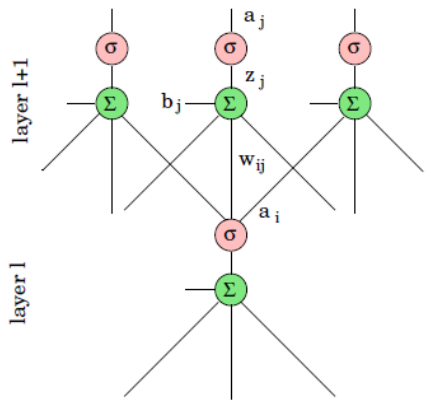
$$a^l = \sigma(b^l + w^l * x^l) \tag{4}$$

where a^l is the **activation vector** at layer l, $z^l = b^l + w^l * x^l$ is the **weighted input** at layer l and $x^{l+1} = a^l$ (the activation is the input at the next level.

The dimension of w^l and b^l depend on the number of neurons at layer l and all of them are parameters of the model.

The overall structure is:

- Compute the gradient of the error
- Backpropagate error derivatives to activations and weighted input of hidden layers, using chain rule
- Derive error derivatives at layer l w.r.t. each parameter of the layer.



$$\frac{\partial E}{\partial z_j} = \frac{\partial E}{\partial a_j}\frac{da_j}{dz_j} = \frac{\partial E}{\partial a_j}\sigma'(z_j)$$

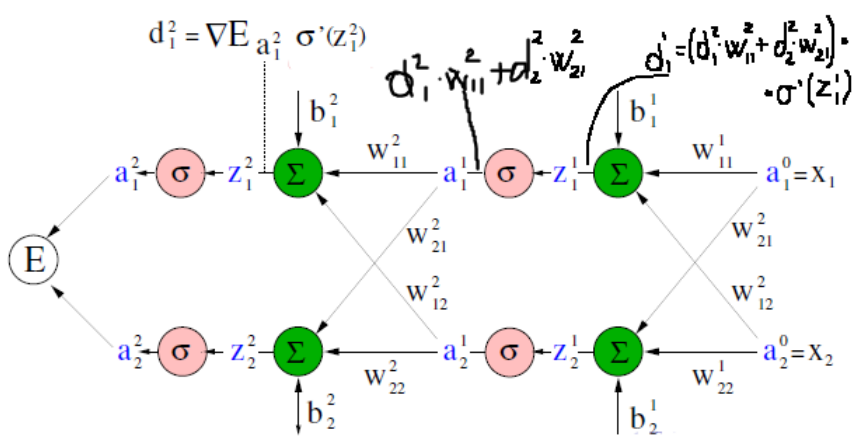
perché zj =wijai+bj

$$\frac{\partial E}{\partial a_i} = \sum_j \frac{\partial E}{\partial z_j}\frac{dz_j}{da_i} = \sum_j \frac{\partial E}{\partial z_j} w_{ij}$$

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial z_j}\frac{dz_j}{dw_{ij}} = \frac{\partial E}{\partial z_j} a_i$$

$$\frac{\partial E}{\partial b_j} = \frac{\partial E}{\partial z_j}\frac{dz_j}{db_j} = \frac{\partial E}{\partial z_j} \quad 1$$

- **Forward pass:** Take a sample $< x, y >$ and compute the vectors z^l, a^l at each layer l
- **Backward pass:**
  1. First compute the partial derivative of the error w.r.t. the last activations, then according to chain rule, backpropagate through a function at a time multiplying by the partial derivative of the function we trasversed.
  2. When an activation is shared by multiple units, need to sum together the contribution of the partial derivatives along all directions.
  3. We reached the z of the previous layer and repeat the same computation through all layers.



Given some error function E (euclidean distance) let us define the error derivative at l as the following vector of partial derivatives:

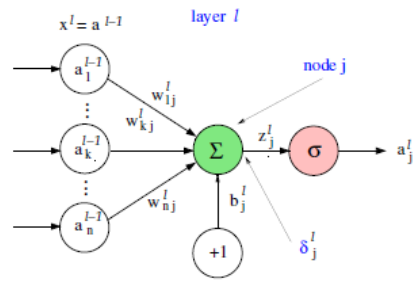
$$\delta' = \frac{\partial E}{\partial z'} \tag{5}$$

$$(BP1) \quad \delta^L = \nabla_{a^L} E \odot \sigma'(z^L)$$

$$(BP2) \quad \delta^l = (W^{l+1})^T \delta^{l+1} \odot \sigma'(z^l)$$

$$(BP3) \quad \frac{\partial E}{\partial b_j^l} = \delta_j^l$$

$$(BP4) \quad \frac{\partial E}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l$$



where the operator in BP1 and BP2 is the Hadarmard (point-wise) product. Backpropagation algorithm:

- **Input:** $< x, y >$: $a^0 = x$
- **Feedforward:** for $l = 1, 2, ..., L$ compute $z^l = w^l a^{l-1} + b^l$ e $a^l = \sigma(z^l)$
- **Output error:** compute the vector $\delta^L = \nabla_a E \odot \sigma'(z^L)$
- **Backpropagation:** for $l = L-1, L-2, ..., 1$ compute $\delta^l = ((w^l)^T \delta^{l+1} \odot \sigma'(z^l)$
- **Updating:** for $l = 1, 2, ..., L$ update the parameters as:
  - $w_{jk}^l = w_{jk}^l + \mu a_k^{l-1} \delta_j^l$
  - $b_j^l = b_j^l + \mu \delta_j^l$

1.6 Learning issues

The gradient doesn't necessarily points to the direction of the local minimum.

Issues with backprop: according to (BP4) equation, when activations are low (close to 0), weights change/learn slowly.

According to (BP2) equation, for sigmoid activations if $\sigma(z^l)\tilde{0}$ or $\sigma(z^l)\tilde{1}$, then $\sigma'(z^l)\tilde{0}$ and the neuron is saturated.

Summing up, a neuron learns slowly if either its input is low, or the output is saturated (either close to 1 or 0) - so we use small weights to avoid saturation.

1.6.1 Vanishing gradient problem

After a few layers the gradients disappears, becomes close to 0 and the net learns slowly.

For the first layers, the gradient is the product of many factors of the form $\sigma(z^l) \leq 1/4$ for small values of z^l.

The first layers learn much more slowly than the last layers. On the other side, if weights are small, the first layer loose most of the input information.

Hence, last layers learn fast but on a highly deteriorate information. *This problem is connected to the use of sigmoid function (not ReLU).*

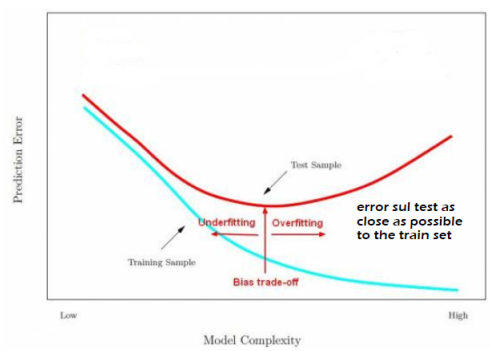
The vanishing gradient problem blocked the progress of neural networks for almost 15 years. It was first bypassed by network pre-training and later by the introduction of **Rectified Linear Units (RELU)** activation function.

Still fine-tuning starting from food network weights (VGG) is a viable approach for many problems (Transfer learning)

1.7 Overfitting

- **Overfitting:** the model is too complex and specialized over the peculiarities of the samples in the training set
- **Underfitting:** the model is too simple and does not allow to express the complexity of observations

Deep models are good at fitting, but the real goal is generalization.



Dense layers are more prove to overfitting, CNNs less prone because they average different situations.

To reduce overfitting:

- **Collect more data:** using a generator (but creating a supervided dataset is expensive bacause we need annotated data)
- **Reduce model capacity:** smaller models are less expensive
- **Early stopping:** stop when loss improve on training but no more on test

- **Regularization:** we apply weight decay, a component proportional to the current value of the weight to penalize large weights and keep them as small as possible
- **Model averaging:** compute different models and average them (forest of trees)
- **Data augmentation:** mimic the data by applying some changes (rotate, add noise)
- **Dropout:** only in Deep Learning which is peculiar - the idea is to cripple the network stochastically removing hidden units (reduce model capacity.

  During training, at each iteration hidden units are disable with probability $1-p$ (0.5). Hidden units cannot co-adapts to other units - similar to train many networks and averaging between them.

  At each stage of training, only the crippled network is trained, then the omitted units are reinserted and the process repeated. At test time, we weight each unit with its expectation p. For a single layer this is equivalent to take a **geometric average** among all different crippled networks.

  With dropout, we are **randomly sampling** from an exponential number of different architectures *(all architectures share weights)*.

  Sharing weights means that **every** model is very strongly **regularized** by all the other models - a good alternative to L2 or L1 penalties that pull the weights towards zero.

1.8 Activation and loss functions for classification

1.8.1 Sigmoid

When the result of the network is a value between 0 and 1, e.g. a probability for a binary classification problem, it is customary to use the sigmoid function as activation function:

$$\sigma(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{1+e^x} \tag{6}$$

1.8.2 Softmax

When the result of the network is a probability distribution, e.g. over K different categories, the softmax function is used as activation:

$$softmax(j, x_1, ..., x_k) = \frac{e^{x_j}}{\sum_{j=1}^{k} e^{x_j}} \tag{7}$$

It is easy to see that $0 < softmax(j, x_1, ..., x_k) < 1$ and that the sum is up to 1.

It is easy to prove than for any c constant $softmax(j, x_1 + c, ..., x_k + c) = softmax(j, x_1, ..., x_k)$. We can always assume one argument is null. In the

binary case, we would be left with a single argument, and in particular: $\sigma(x) = softmax(x, 0) = \frac{e^x}{e^x+e^0} = \frac{e^x}{e^x+1}$.

e^{x+c} becomes ce^x, so a constant which is ignored and can be simplified. Generalizing, computing sigmoid for 1 output is the same as computing softmax for 2 outputs.

1.8.3 Loss functions

Comparing distributions: probability distributions can be compared according to many different metrics:

1. **Wasserstein distance:** their difference $P - Q$

2. **Kullback Leibler divergence:** it is not a distance in a mathematical sense because it is not symmetric. $DKL(P||Q)$ between two distributions is a measure of the information loss due to approximating P with Q:

$$DKL(P||Q) = -H(P) - \sum_i P(i) logQ(i) \tag{8}$$

where i is the category, $P(i)$ is the true probability of belonging to i and $Q(i)$ is the predicted probability of belonging to i

We call **cross-entropy** between P and Q the quantity:

$$H(P, Q) = -\sum_i P(i) logQ(i) = H(P) + DKL(P||Q) \tag{9}$$

both CE and DKL are measure of distance between P and Q. Minimizing CE or DKL is the same because they just differ for the entropy of P.

We can take as **learning objective** the minimization of the Kullback-Leibler divergence. Since, given training data, their entropy $H(P)$ is constant, minimizing DKL is equivalent to minimizing the cross-entropy $H(P, Q)$ between P and Q.

Binary case (Binary Cross-Entropy): let's consider $Q(y = 1|x)$ the probability that x is classified as 1 and $Q(y = 0|x) = 1 - Q(y = 1|x)$ we use **sigmoid** as activation function and the **negative log-likelihood/binary cross-entropy** as loss:

$$H(P, Q) = -ylog(Q(y = 1|x)) - (1 - y)log(1 - Q(y = 1|x)) \tag{10}$$

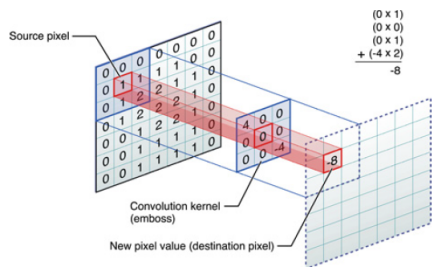
Multi-label case (Categorical Cross-Entropy: we want to split it according to the possible labels l of Y: $logQ(l_1|X) + ... + logQ(l_n|X)$. According to the actual probability that X had label l, we use **softmax** as activation function and **categorical cross-entropy as loss**:

$$\sum_l P(l|X) log(Q(l|X)) \tag{11}$$

2 Convolutional Neural Networks

Use CNNs only when the input is a **signal** that has a relation between adjacent values. If the order of the date is irrelevant then don't use them.

Filters and convolutions: the center element of a kernel is placed over the source pixel. The source pixel is then replaced with a weighted sum of itself and nearby pixels.



The activation of a neuron is not influenced from all neurons of the previous layer, but only from a small subset of adjacent neurons: his **receptive field**. As a consequence we can change values outside RF and the valued of the neuron does not change. *A neuron cannot see anything outside its receptive field.*

Every neuron works as a **convolutional filter**. Weights are **shared**: every neuron perform the same trasformation on different areas of its input.

With a cascade of convolutional filters intermixed with activation functions we get complex non-linear filters **assembling local features** of the image into a **global structure.**

2.1 Convolution in images

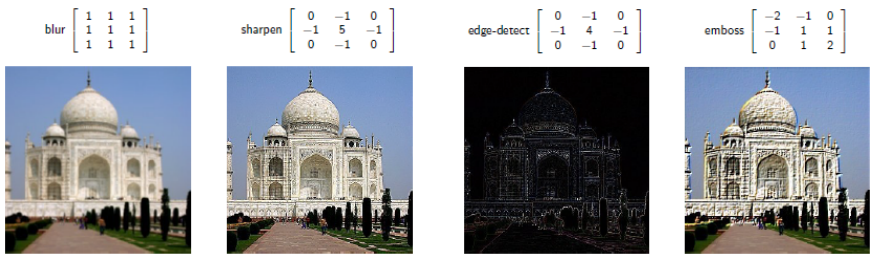
An **image** is coded as a numerical matrix - grayscale (0-255) or RGB (triple 0-255). They can be interpreted as surfaces with height proportional to intensity.

Some interesting points are points where there is a **discontinuity**. a fast variation of the intensity which can be calculated by means of **derivatives**. We can compute **discrete approximations of derivatives** convolving simple **linear filters** - look for regions where the derivative is high.

Linear filters: the derivative is an example of linear filter.

Idea: create new images where each pixel is a linear combination (defined by a **kernel**) of the adjacent pixels- The same trasformation is repeatedly applied centering the kernel on every pixel (**convolution**).

Relevant properties: the output is a linear transformation of the input, a shift of the input results in a shift of the output, linear filter can be combined.



Convolution is a mathematical operation transforming an input matrix by means of another matrix. The operation can be generalized tot he continuous case. In the binary case, given a function $f(x, y)$ and a kernel $k(x, y)$ the convolution $f * k$ if f and k are defined as:

$$f(x,y) * k(x,y) = \begin{cases} \int_u \int_v f(x-u, y-v) * k(u,v) & \text{continuous} \\ \sum_u \sum_v f(x-u, y-v) * k(u,v) & \text{discrete} \end{cases} \tag{12}$$

Convolution is symmetric, associative and distributive.
Having a kernel in the interval $[-M, M]$,

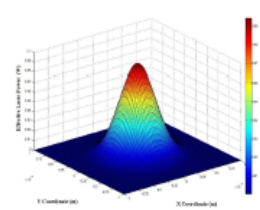
$$(f * k)(x) = \sum_{m=-M}^{M} f(x-m) * k(m) \tag{13}$$

Observe that k(-M) is the multiplicative factor for f(x+M), that is the kernel must be **flipped** before taking products. If you do not flip the kernel, you get a **cross-correlation**. Not relevant if the kernel is symmetric and not so relevant for neural nets since weights are generated by the machine.

2.2 Gaussian Smoothing

It is important to remove noise via smoothing. In averaging, we weight pixels according to their distance from the center. It is sharper than the box filter since the latter introduces artifacts inside the image while gaussian not.

$$K = \begin{bmatrix} .02 & .08 & .14 & .08 & .02 \\ .08 & .37 & .61 & .37 & .08 \\ .14 & .61 & 1 & .61 & .14 \\ .08 & .37 & .61 & .37 & .08 \\ .02 & .08 & .14 & .08 & .02 \end{bmatrix}$$



Frequence: how likely the intensity of a pixel is going to change inside an image. High frequency $\rightarrow$ a lot of change.

In general we expect to have continuative image (low frequency). The noise is instead high frequency because is related to perturbation of single pixels so filter out high frequency, using low pass filter to remove noise.

$$\underbrace{\begin{bmatrix} -1 & 0 & 1 \end{bmatrix}}_{\text{finite diff}} \overset{\text{composition}}{*} \underbrace{\begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix}}_{\text{simple gaussian}} = \underbrace{\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}}_{\text{Sobel vert. edge detect.}}$$

$$\underbrace{\begin{bmatrix} 1 & 2 & 1 \end{bmatrix}}_{\text{simple gaussian}} * \underbrace{\begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix}}_{\text{finite diff}} = \underbrace{\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}}_{\text{Sobel horiz. edge detect.}}$$

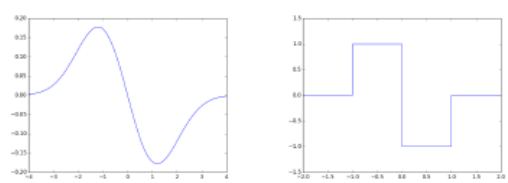
Figure 2: Sobel operator

We can directly convolve with the derivative of the gaussian instead of applying first the filter and then computing derivative:

$$\frac{\partial(G_\sigma * f)}{\partial x} = G'_\sigma * f \tag{14}$$

$$G_\sigma = e^{-\frac{x^2}{2\sigma^2}} \qquad G'_\sigma = -\frac{x}{\sigma^2} e^{-\frac{x^2}{2\sigma^2}} \tag{15}$$

As for the finite difference, the derivative of the Gaussian will be null in flat regions and will produce peaks in correspondence of steps. Far less sensible to noise.



gaussian derivative finite difference

Each filter provides a different information. We do not want to use predefined filters, but we want the network to **learn the filters**.

2.3 Convolutional layer

A **convolutional layer** is defined by the following parameters:

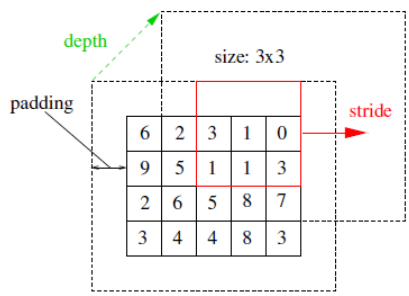
- **kernel size:** the dimension of the linear filter

- **stride:** movement of the linear filter. With a low stride receptive fields largely overlap. With higher stride, we have less overlap and the dimension of the output get smaller

- **padding:** artificial enlargement of the input to allow the application of filters on borders. It is used to enlarge the input to preserve the dimension of the output. In general we use 'zero padding' to fill with 0s.

  1. **valid:** no padding is applied
  2. **same:** you add a minimal padding enabling the kernel to be applied an integer number of times → preserve the dimension of the image

- **depth:** number of different filters that we wish to syntesize. Each neuron will be connected to the same portion of the input. Number of channels in the output.



Along each axes the dimension of the output is given by the formula

$$\frac{W + P - K}{S} + 1 \tag{16}$$

where W is the dimension of the input, P the padding, K the kernel size and S the stride. W+P is the actual dimension of the image because we enlarged it with padding. Once we apply the kernel, the image is W+P-K. The number of movements that we can apply is the numerator divided by the stride S, and +1 because is the first possible location of the kernel.

Adding an activation layer, the output dimension does not change.

IMPORTANT: Unless stated different, a filter operates on **all input channels in parallel.** So, if the input layer has depth D, and the kernel size is NxM, the actual dimension of the filter will be NxMxD which is a cube.

The convolution kernel is tasked with simultaneously mapping **cross-channels** correlations and **spatial correlations**.

2.4 Pooling layer

In deep NNs, it is common practice to alternate layers with **pooling** layers, where each neuron simply takes the **mean or maximal** value in its receptive

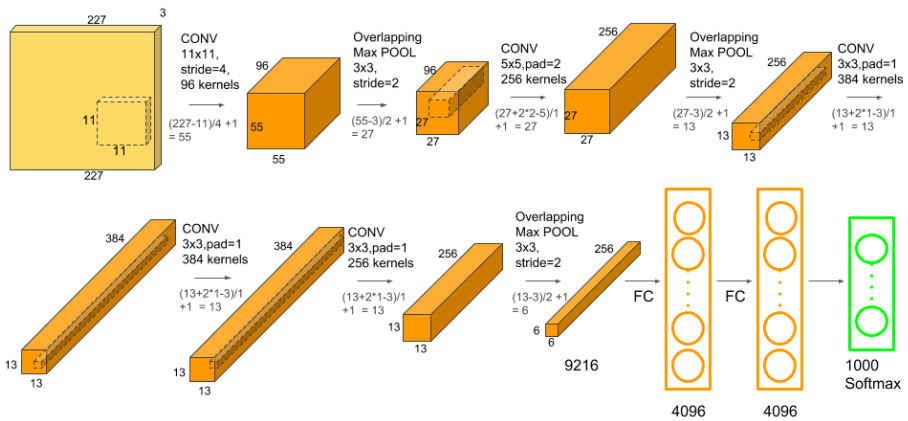
field. This has a double advantage: reduces the dimension of the input and gives some tolerance to translations.

If we only want to decrease dimension we just increase the stride. Before pooling layers were more famous, now less. Mean is a linear filter while max is non-linear. Working with small layers is less computational expensive.

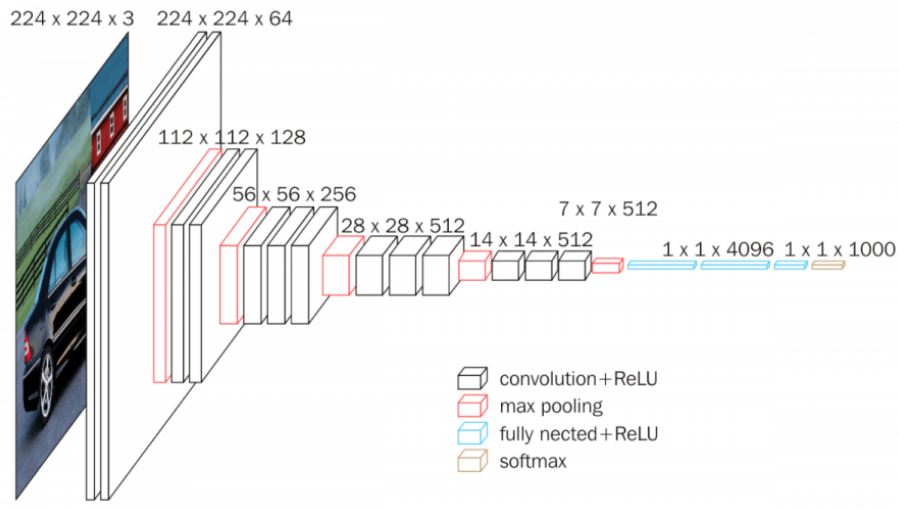
With Max Pooling we **lose localization** but gain tolerance to translations.

2.5 Real CNNs

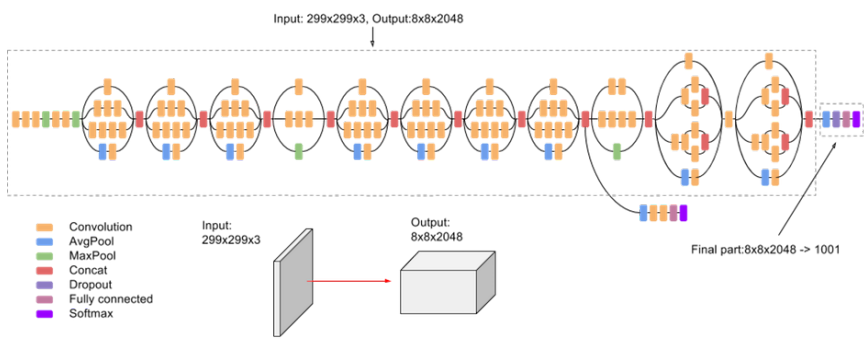
- **AlexNet** winner of the NIPS contest in 2012: Kernel=(11x11) is unusual (in general we use 3x3). It is good to have a lot of channels because they contain interesting info. When we reduce the spatial dimension, we enlarge the channel dimension. We use 2 dense layers because we learned that 2 are enough. Use Max Pool because we are no more interested in localization of features, so we extract them to perform classification. Use features for classification (2 layers + 1 output).



- **VGG:** there are two versions 16 and 19 which differ for the number of layers. It has a simple, linear and clean architecture. It is an alternation of conv to reduce size and pool to double the dimension. It has a heavy number of parameters w.r.t. AlexNet. We use MaxPool before Dense to allow the use of an input of arbitrary dimension.



- **Inception V3:** The convolutional part is a long composition of inception modules (tower of nets). Input is processed along different towers, then is collected and passed to the next layer.

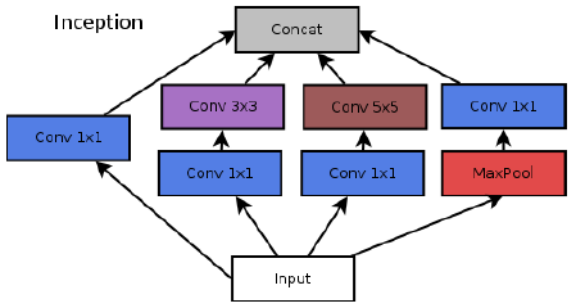


  Conv 1x1: 1x1xC, we get a simple weight for each channel, in a sense we are processing together all the channels, we reduce the dim of the channels by keeping only the most informative dim. We perform a linear transformation essentially equivalent to PCA: "give me the C most informative components of the input".

  It can be better to **decouple kernels**, independently looking for cross-channels correlation (via 1x1 Convs), and spatial 2D Convs.

  Inception modules can be understood as an intermediate step between a regular convolution and a depthwise separable convolution (a depthwise convolution followed by a pointwise convolution).

- **Xceptions**



- **MobileNet:** a class of light models conceived to be deployed on mobile devices.

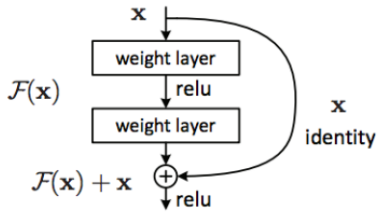
Depth-wise separable convolutions kernel applies C 2D kernels, each of a single channel on input so we have C results (depth-wise convolution). Then applies a 1x1xC kernel (point-wise convolution) on the intermediate result obtaining a result with one channel. If we apply C_out kernel 1x1xC we obtain the desired result.

Suppose we have a convolutional layer with a 3x3 kernel, 16 input channels and 32 output channels. The input is convolved 32 times with different kernels of dimension 3x3x16 = 144: we have a total of 32x144 = 4608 parameters.

In a depthwise separable convolution on the same example, we first traverse the 16 channels with a different 3x3 kernel, and then we apply 32 different kernels with dimension 1x1x16. The total number of parameters is 16x3x3 + 32x1x1x16 = 656. We have fewer parameters so it is simpler to learn.

2.6 Residual Learning

Instead of learning a function $F(x)$, we try to learn $F(x) + x$.



We add a residual shortcut connection every 2-3 layers. Inception ResNet is an example of a such an architecture.

Not well understood why it works. Usual explanation is that during back propagation, **the gradient at higher layers can easily pass to lower layers**, without being mediated by the weights layers, **which may cause vanishing gradient or exploding gradient problem.**

The derivative of the identity is 1, so the error is backpropagated 'as it is'.

ConvNets grow essentially in three directions: **Layers, Channels**, **Resolution** (spatial width of layers).

How to scale a ConvNet?

EfficientNet increase first the channels because they bring a sensible improvement and then resolution.

2.7 Backpropagation for CNNs

Input and output are unrolled into vectors from left to right, top to bottom.

The operation performed by the convolution can be seen a single dense network.

Each column corresponds to a different application of the kernel

$w_{i,j}$ is a kernel weight, with i and j being the row and column of the kernel respectively. Matrix is trasponsed for convenience.

However, updates relative to a same kernel weight must be shared (taking the mean). Learning parameters in a CNN are faster than in a Dense NN since weights are shared. Also, CNNs are less prone to overfitting because we compute the average of a lot of positions, so it is difficult that they overfit in all of them.

2.8 Transposed convolutions

Normal convolutions with stride > 1 downsample the input dimension. We might downsample to gain efficiency or to reduce the input dimension so to enlarge the dimension of receptive field.

In some cases, we might want to upsample the input. i.e. to obtain an image of the same dimension of the input after some compression to an internal encoding or to project feature maps to a higher-dimensional space.

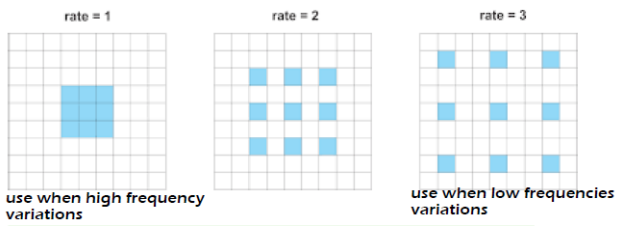
A **transposed convolution** can be seen as a normal convolution with subunitarian stride. To mimic it, we must upsample the input and then apply a single strided convolution. When enlarging we can use 0 or similar values.

By simply transposing the matrix, we convert an input of dimension o into an output of dimension i. So the kernel defines a convolution matrix, but whether it is a direct or transposed convolution is determined by how the matrix is applied. In a transposed convolution we invert the forward and backward pass.

2.9 Dilated (aka Atrous) Convolutions

Dilated convolutions are just normal convolutions with holes. In upsample, we have a dialtion of the input while here we have a **dilation of the kernel**.

The rate is the distance between two consecutive points of the kernel. It enlarges the receptive fiels, keeping a low number of parameters. Might be useful in first layers, when working on high resolution images because nearby pixels look similar.



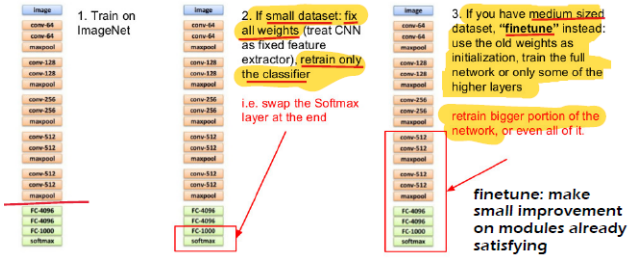
Temporal Convolutional Networks TCNs use dilated convolutions to process long input sequences. Here, dilated convs cover all the long sequence with a few convolutions.

2.10 Transfer Learning

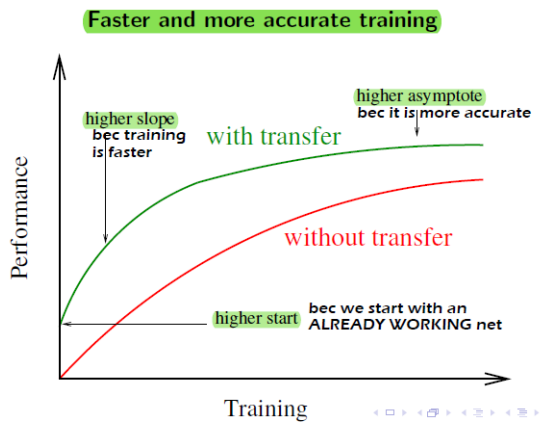
The first layers of convolutional networks for computer vision compute feature maps of the original image of growing complexity.

The filters that have been learned (in particular, the most primitive ones) are likely to be **independent** from the particular kind of images they have been trained on.

They have been trained on a **huge amount of data** and are probably very good. It is a good idea to try to reuse them for other classification tasks and trasfer the knowledge acquired on a dataset to another.



Transferring knowledge from problem A to problem B makes sense if the two problems have **similar inputs**, and we have **much more training data** for A than for B. Both process images and these images do not belong to a very specific domain.



2.11 How CNNs see the world

Each neuron gets activated by specific patterns in the input images. The intuition is that neurons at higher levels should recognize **increasingly complex patterns** obtained as a combination of previous patterns, over a larger receptive field.

In the highest layers, neurons may start recognizing patterns similar to features of objects in the dataset.

In the final layers, neurons gets activated by 'patterns' identifying objects in the category.

The gradient ascent technique

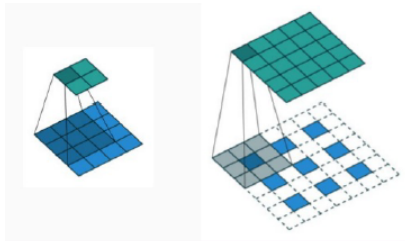
- Start with a random image.
- Do a forward pass using this image x as input to the network to compute the activation $a_i(x)$ caused by x at some neuron (or at a whole layer).
- Do a backward pass to compute the gradient of $\partial a_i(x)/\partial x$ of $a_i(x)$ with respect to **each pixel** of the input image.
- Modify the image adding a small percentage of the gradient $\partial a_i(x)/\partial x$ and repeat the process until we get a sufficiently high activation of the neuron. It's the same as traditional backward pass, but it arrives at the input and then modifies it.

In first layers of AlexNet, patterns look similar the derivatives of gaussian. In VGG, we can identify in first layers just colors or very simple patterns. Each channel then implements a different pattern that we are trying to synthetize.

Instead of syntesizing an image **maximizing the activation** of a neuron, we can **emphasize** in real images what caused the activation through deconvolutions (convolution with sub-unitarian stride) or unpooling via switches (boolean maps).

2.11.1 Deconvolution

Also called transposed convolutions, are like normal convolutions with sub-unitarian stride (adds zero values to upscale the image.



We go forward up to the layer we are interest in, then we go backward.

1. Select an image with a strong activation for a given neuron
2. Zero out all activations for different neurons
3. Go back to image space, but through the deconv net

Warning: convolutional kernel must be transposed.

Applying the transpose of a kernel we emphasize the portion of the image that caused the activation.

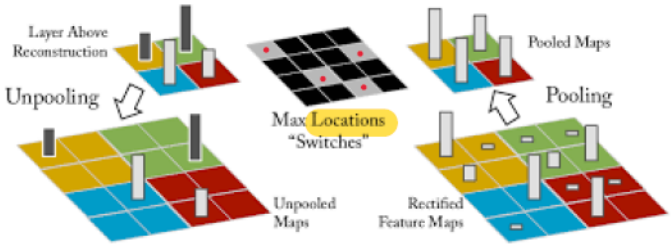
From a signal processing perspective, the transpose of a kernel is the so called **matched filter**.

The matched filter can be used to detect the presence of a *template signal* in some arbitrary, noisy signal.

2.11.2 Unpooling

Remember that when pooling we lose information. **Goal** is to approximate the inversion of a max pooling operation.

We memorize the positions of the maximum via a boolean map.



Moving towards higher levels we observe: growing structural complexity (oriented lines, colors - angles, arcs - textures), more semantical grouping, greater invariance to scale and rotation.

2.11.3 Another approach

Try to understand the inner representation at some layer by generating an image indistinguishable from the original one.

Synthetize other images that have the same internal representation to maximize the fact to have an output equal to the interpretation of the image.

DeConv nets look at how certain network outputs are obtained, whereas we look for *what information is preserved* by the network output.

Similar to autoencoding but:

- it works with an arbitrary network, not an autoencoder
- we are not training the network, backpropagation is used to reconstruct the image
- loss is measured on the **internal representation**

we 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.**

$$\arg \min_{x \in R^{HxWxC}} l(\Theta(x)), \Theta_0) + \lambda R(x) \tag{17}$$

where the first term is the loss and second one is a regularizer.

First layers are an invertible code, progressively becoming **fuzzier**.

Last layers invert back to multiple copies of object's parts at different positions and scales.

2.12 Fooling Neural Networks

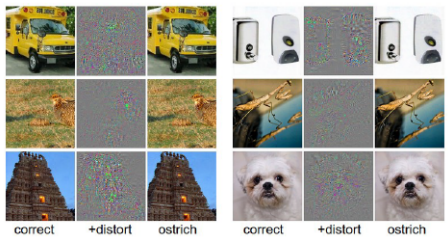
Application of gradient ascent technique: modify images to fool the net.

We already know that we may use gradient ascent to to modify an input image to emphasize the activation of a given neuron.

In an image classification framework, we can use this technique to increase the score of whatever class we want.

So, in principle, we should be able to automatically syntesize images of any of the categories under consideration.

Since we have many pixels, a tiny (imperceptible to humans!), consistent perturbation of all of them is enough to fool the classifier.



In "Deep Neural Networks are easily fooled", Nguyen et al. proved that it was easy to produce adversarial examples using an **evolutionary technique**.

In particular, they were able to produce not only noisy adversarial images, but also geometrical examples with high regularities (images meaningful for humans).

Evolutionary approach:

- Start with a random population of images
- Alternatively apply **selection** (keep best) and **mutation** (random perturbation/crossover).

Image encoding:

- **Direct encoding:** explicit representation of the image as an array of pixels.
- **Indirect encoding:** implicit representation of the image as a composition of regular functions: sine, gaussian, sigmoid, linear, etc.

A general problem is that all classification techniques are vulnerable. Generative approaches do not sensibly improve the situation.

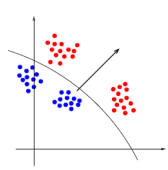
2.12.1 Manifold

Manifold: a collection of points forming a certain kind of set, such as those of a topologically closed surface.

Random images are completely different, but for us may look similar. The probability to generate images with regularities in a random way is zero.

Natural images have *manifolds with low dimensionality* (that is why we can compress them).

In high dimensional space, easy to move away from the manifold of a given category - with modifications imperceptible to the human eye (because high dimension = lots of features, we do not know how they are combined in order to classify the image).



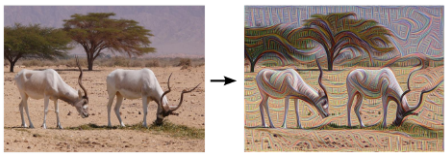
Due to correlations between features, datapoints occupy small and specific ares of the space. Moving away from a class the probability to end up in a meaningful point can be very low.

2.13 Inceptionism

Initially intended to visualize what a deep neural network is seeing when it is looking in a given image. It evolved into a new form of psychedelic and abstract art.

Deep dreams approach:

- train a network for image classification
- revert the network to slightly adjust (via backpropagation) the original image to improve activation of a specific neuron
- after enough reiterations, images will be incepted by the desired features, creating psychedelic and surreal effects;
- generated images take advantage by strong regularizers privileging inputs that have statistics similar to natural images, like e.g. correlations between neighboring pixels (texture).



Instead of prescribing which feature we want to amplify, we can also fix a layer and enhance whatever it detected.

Each layer of the network deals with **features at a different level of abstraction**. We enrich the image with the activation of specific neurons. Lower layers will produce *strokes or simple ornament-like patterns*, because those layers are *sensitive to basic features such as edges and their orientations*.

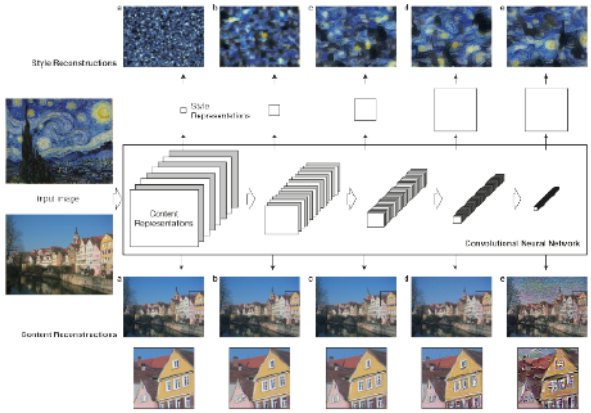
2.14 Style Transfer

Represent the same content in a different style.

Mimicking artistic style: the gradient ascent technique can also be used to mimic artistic style.

We add a feature space on top of the original CNN representations computing correlations between the different features maps (channels) at each given layer.

Capture the textures of the image considering the correlation between the channels. In first layer, we consider the correlation between high frequencies.



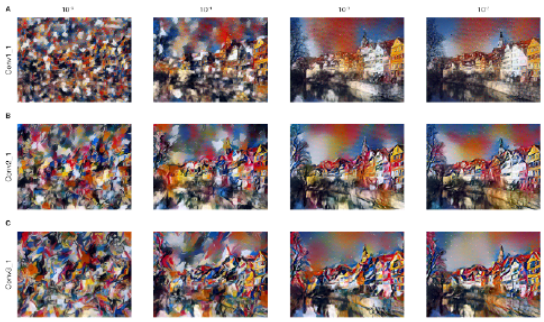
We know that at layer l an image is encoded with D^l distinct feature maps F^l_d each of size M^l (width times height). $F^l_{d,x}$ is thus the activation of the filter d at position x at layer l.

Feature correlations for the given image are given by the Gram matrix $G^l \in R^{D^l x D^l}$, where $G^l_{d_1,d_2}$ is the dot product between the feature maps $F^l_{d_1}$ and $F^l_{d_2}$ at layer l

$$G^l_{d_1,d_2} = F^l_{d_1} * F^l_{d1_2} = \sum_k F^l_{d_1,k} * F^l_{d_2,k} \tag{18}$$

This is just yet another internal representation of the image, useful to encode style. The gram matrix is the measure of the texture and the objective is to change the original image in such a way to obtain a gram matrix similar to the other gram matrix.

Content and style are separable:



Other approached: **Cycle GANs**, as many other approaches, require per-style training, and are difficult to generalize.

More recent techniques, as the one described in **Universal Style Transfer via Feature Transforms**, overcome this limitation, taking in input a content image, a style image to mimic, and adapting content to the given style.

3 Segmentation

3.1 Semantic segmentation

It is pixel-level segmentation. Classify each pixel in an image according to the object category it belongs to.

Building supervised training set is expensive, since it requires a complex human operation.

The label is itself an image with a different color for each category. For this reason, semantic segmentation can be regarded as a special case of **image-to-image transformation**.

Datasets:

- **PASCAL VOC 2012:** has 20 classes including person, animal (bird,cat,cow, dog,horse, sheep), vehicle (aeroplane, bicycle, boat, bus, car, motorbike, train), indoor (bottle, chair, dining table, potted plant, sofa, tv).

  The train/val data has 11530 images containing 27450 ROI annotated objects and 6929 segmentations.

- **Cityscapes**: semantic understanding of urban street scenarios. A large dataset containing stereo video sequences recorded in street scenes from 50 different cities, with high quality pixel-level annotations of 5k frames.

- **Syntia:** a collection of photo-realistic frames rendered from a virtual city with annotations for 13 classes.

- **ScanNet:** 3D reconstructions of indoor scenes.

- **Sun RGB-D:** 10000 RGB-D images densely annotated indoor scenes including 146k 2D polygons and 58k 3D bounding boxes.

3.2 Convolutionalization

Composing convolutions we still get a convolution. Specifically, the composition of convolutional layers essentially behaves as a convolutional layer.

The stride of the compound convolution is the **product** of the strides of the components.

Dimension of kernel

Remember that, neglecting padding $\frac{D_{in}-K}{S}+1 = D_{out}$ or equivalently $D_{in} = S*(D_{out}-1)+K$.

The input kernel dimension increases linearly w.r.t. the dimension of the output kernel, more rapidly if the stride is greater than 1.

Example: suppose a kernel of dimension $K_1 = 5$ and stride $S_1 = 3$ with another kernel of dimension $K_2 = 3$ and stride $S_2 = 2$.

By applying the rule we get for $D_2 = 1$, $D_1 = 2*(1-1)+3 = 3$ and $D_0 = 3*(3-1)+5 = 11$.

What breaks convolutionality are the dense layers at the end of networks. If maxpooling has a fixed pooling dimension, because dense layers want input **fixed** dimensions. Instead convolutions work with input of **arbitrary** dimensions. That's why we use **global** pool layer before the dense.

Neurons in dense or convolutional layers share the **same functionalities:** they simply compute weighted sums of their inputs (dot products).

We can look at dense layer as a convolution with a particularly large filter (equal to its input size).

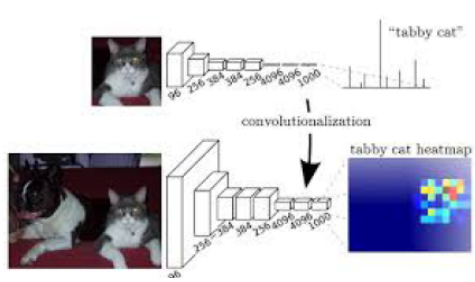
This way each dense layer can be turned into a convolutional layer (reusing the same weights), making the resulting networks **fully convolutional**.

3.2.1 Fully Convolutional Netowrk

If we start from an image classification network working on images of given, fixed size, we get DCNs that:

- takes in input images of **arbitrary** dimension
- produces in output a **heatmap** of activations of the different categories, relative to different locations of the input image upon which the DCN was convolved.

Basically, we train a Conv+Dense NN, then drop dense and insert other CNN with unitary kernel and get a *full CNN* to get as output an heatmap. So as, we find the category (what) and the location (where).



Dense layers are now unary convs directly working on the channels output spatial dimension (x,y) and a number of channels for each category. We are doing segmentation.

InceptionV3 as a single convolution

We can look at the whole convnet as a single convolution. The stride is the product of all strides which is $2^5 = 32$ while kernel size is $D_{in} = 32D_{out} + 43$.

For $D_{out} = 1$ we get 75 as kernel size which is also the **minimal** input dimension for inceptionV3.

For $D_{out} = 8$ which is the dimension of the global pooling layer, we get 299 that is the **canonical** input dimension for inceptionV3.

Padding adds artifacts, so better avoid them.

Given an image with input dimension D_{in}, the dimension of the resulting heatmap will be: $\frac{D_{in}-75}{32} + 1$ that is very coarse (nor precise) because the dimension of the output is a lot smaller than the dimension of the input.

Localization of objects is far from precise because the dimension of the heatmap is small.

From one side we want to reduce the dimension of the output, but at the same time this is a problem. The 32 pixel stride at the final prediction layer limits the **scale of detail** in the upsampled output.

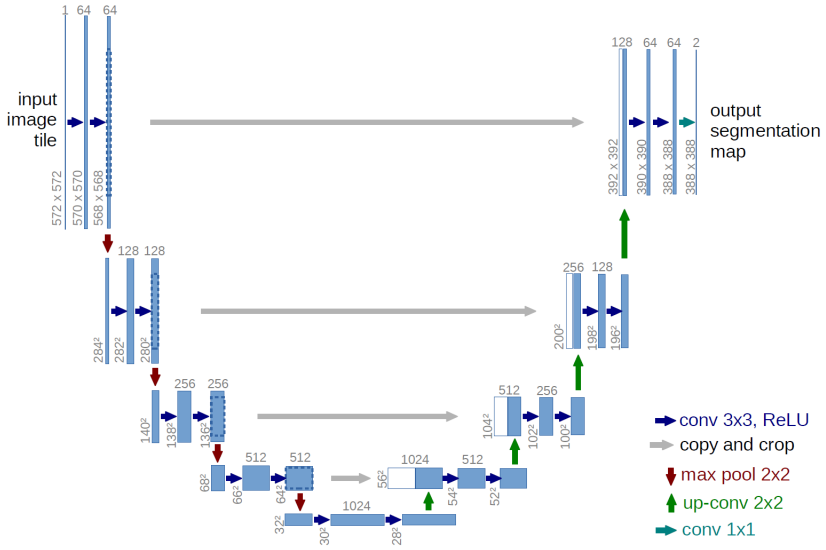
Shelhamer address this issue by adding **skips** that combine the final prediction layer with lower layers with finer strides.

Classification form previous layers is *less accurate* from semantic point of view but *more accurate from localization point of view*.

We combine classification form last layer with classification from previous layers. But not too early layers otherwise we lose semantic classification.

3.3 U-net

The U-NET was developed by Olaf Ronneberger et al. for Bio Medical Image Segmentation.



The architecture contains two paths.

- First path is the **contraction path** (also called as the encoder) which is used to capture the context in the image. The encoder is just a traditional stack of convolutional and max pooling layers.

  When using Max Pooling layers we downsample the image → reduce spatial dimension, and increase channel dimensions - so as the feature information is increased and we have more meaningful features in the lower part of the network.

- The second path is the **symmetric expanding path** (also called as the decoder) which is used to enable precise localization using transposed convolutions which up-sample the image.

Thus it is an end-to-end fully convolutional network (FCN), i.e. it only contains Convolutional layers and does not contain any Dense layer because of which it can accept image of any size.

It does not rely on a classification network because it learns segmentation in an end-to-end setting.

It can work with relatively **few training images** (data augmentation was largely exploited) and yields accurate precise segmentations.

4 Object Detection

Similar to segmentation, but we are supposed to return a **boundary box** containing the object and *not a boolean map*.

Pro: no need to strive about borders.

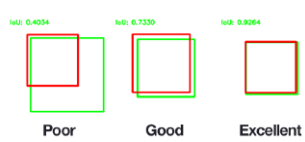
Cons: multiple outputs of unknown numbers. Difficult to train end-to-end, no evident loss function.

4.1 Jaccard Index (aka Intersection over Union)

Tipically, the quality of each individual bounding box is evaluated versus the corresponding ground truth using **Intersection over Union:**

$$IoU(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{19}$$

It is good symmetric metric. If i use a very large box the intersection is 1, but if union is very large we get a small IoU. If i use a small box, union is low but also intersection.



Then, these must be summed up for all detections, and suitably combined with classification errors.

Datasets for object detection:

- **PASCAL Visual Object Classes**
- **Coco**: a large-scale object detection, segmentation and captioning dataset composed of over 200k labeled images, spanning 80 categories.

Deep Object Detection

1. **Region proposal methods** (R-CNN, Fast R-CNN, Faster R-CNN). Region proposals are usually extracted via **Selective Search** algorithms, aimed to identify possible locations of interest. These algorithms tipically exploit the **texture and structure** of the image, and are **object independent**. They are ROI supposed to contain the object, independently from the category, then identify the object (2 phases).

   We have oversegmentation of the image, then combine similar adjacent segments to obtain objects.

2. **Single shots methods** (YOLO, SSD, Retina-net, FPN) which are really fast techniques and a more direct approach to NN and DL. Here there are no sliding windows, we divide the grid and detect object in each cell.

4.2 YOLO

YOLO is an object detector that uses features learned by a deep convolutional neural network to detect an object.

YOLO is an acronym for You Only Look Once, and refers to the fact that (in yolo-1) the input image is processed a single time. It is a Fully Convolutional Networks. The input is progressively downsampled by a factor $2^5 = 32$ (in "same" modality).

For instance, an input image of dimension 416x416 would be reduced to a grid of neurons of dimension 13x13: **the feature map.**

If you divide the input image in a grid of dimension 13x13, you get the **positions** where convolutions would be applied. The dimension of the convolutional kernel (i.e. the receptive field of each neuron in the final grid) depends from the network.

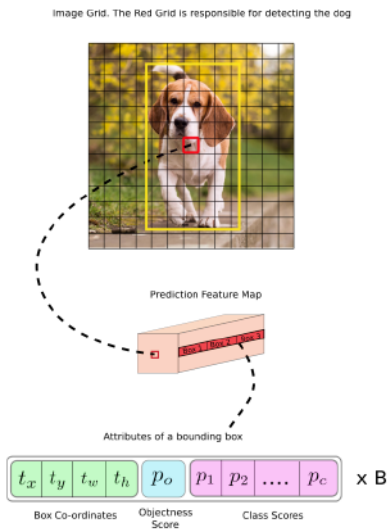
Detection of an object may concern all neurons inside the bounding box. In YOLO a single neuron is responsible for detection: *the one whose grid-cell contains the center of the bounding box.*

This neuron makes a finite number of predictions.

The **objective category** is the probability distribution over all the different classes.

We have a 13x13 neurons in the feature map.

Depth-wise, we have (B x (5 + C)) entries, where B represents the number of bounding boxes each cell can predict (say, 3), and C is the number of different object categories.



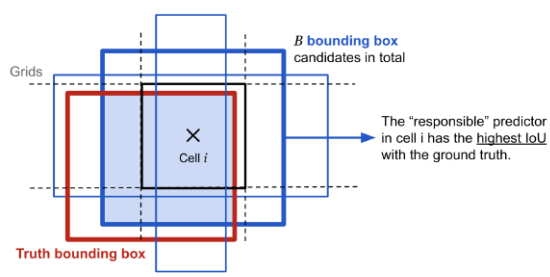
Each bounding box has 5 + C attributes, which describe the center coordinates (2), the **dimensions** (2), the **objectness score**(which is the confidence that i am really looking at an object) (1) and **C class confidences**.

In the image tx and ty usually is 0,1 so sigmoid function. th and tw sometimes too.

- **Anchor boxes:** Trying to directly predict width and the height of the bounding box leads to **unstable gradients** during training. Modern object detectors predict log-space affine transforms for pre-defined default bounding boxes called **anchors**.

  Then, these transforms are applied to the anchor boxes to obtain the prediction. YOLO v3 has three anchors = 3 BB per cell. The BB which detects the object is one whose anchor has the highest IoU with the ground truth box.

  Anchors are choosed by K-means clustering on the training set reflecting the most likely shapes of bounding boxes.



- **Making predictions:** we get BB predictions from the network outputs t_x, t_y, t_w, t_h relative to the cell at coordinated c_x, c_y for an anchor of dimensions p_w, p_h according to the following formula:

  $b_x = c_x + \sigma(t_x)$

  $b_y = c_y + \sigma(t_y)$

  $b_w = p_w * e^{t_w}$

  $b_h = p_h * e^{t_h}$

- **Center coordinates:** YOLO **does not predict** the absolute coordinated of the bounding box's center, but **offsets** which are:

  - relative to the **top left corner of the grid cell** which is predicting the object
  - **normalised by the dimensions of the cell from the feature map** which is 1 (which motivates the use of the sigmoid)

- **Dimensions of the bounding box:** are predicted by applying a log-space transform to the output and then multiplying with the dimensions of the anchor.

  The resultant predictions, b_w and b_h, are *normalised by the height and width of the image*: training labels are chosen this way.

- **Objectness Score:** represents the *probability that an object is contained inside a bounding box.* The objectness score is also passed through a sigmoid, as it is to be interpreted as a probability.

- **Class confidences:** represents the *probabilities of the detected object belonging to a particular class.*

  Before v3, YOLO class scores were computed with softmax. Now, **multiple sigmoid** are used, considering that objects may belong to **multiple categories** and labels are not guaranteed to be mutually exclusive.

- **Loss function:** consists of two parts: the **localization loss** for *bounding box offset prediction* and **classification loss** for *conditional class probabilities.*

  Localization loss: (to minimize quadratic distance)

$$\mathcal{L}_{loc} = \sum_{i=0}^{S^2} \sum_{j=0}^{B} 1_{ij}^{obj} [(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 + (\sqrt{w_i} - \sqrt{\hat{w}_i})^2 + (\sqrt{h_i} - \sqrt{\hat{h}_i})^2] \tag{20}$$

  where i ranges over cells and j over bounding boxes. 1_{ij}^{obj} is a delta function indicating whether the j-th bounding box of the cell i is responsible for the object detection. We care only of the neurons in charge of detecting.

Square root to reduce relevance of large boxes and normalization over 'size' of the boxes not to penalize large objects since mistake of the same amount on large object is less important than those on small objects.

Classification loss in the sum of two components relative to the **objectness confidence** (second term) and the **actual classification** (first term = classification error):

$$\mathcal{L}_cls = \sum_{i=0}^{S^2}\sum_{j=0}^{B}(1_{ij}^{obj}+\lambda_{noobj}(1-1_{ij}^{obj}))(C_{ij}-\hat{C}_{ij})^2+\sum_{i=0}^{S^2}\sum_{c\in C}1_i^{obj}(p_i(c)-\hat{p}_i(c)-\hat{p}_i(c))^2 \tag{21}$$

λ_{noobj} is a configurable parameter meant to down-weight the loss contributed by 'background' cells containing no objects. This is important because they are a large majority.

noobj does not means 'there's no object here' but 'i'm not in charge for objects'.

Whole loss is:

$$\mathcal{L} = \lambda_{coord}\mathcal{L}_{loc} + \mathcal{L}_{cls} \tag{22}$$

where λ_{coord} is an additional parameter, balancing the contribution between the two losses.

In YOLO, $\lambda coord = 5$ and $\lambda_{noobj} = 0.5$. Remember that when multiplying by a constant, we multiply also the gradient.

4.3 Multi Scale processing

We process an image at different scale because we cannot recognize objects at a single given scale since it is very large or vert small, so we need more scales.

4.3.1 Image Pyramids

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

b) First systems for fast object detection (like YOLO v1) opted to use only higher level features at the smallest scale. This usually **compromises detection of small objects**.

(a) Featurized image pyramid (b) Single feature map

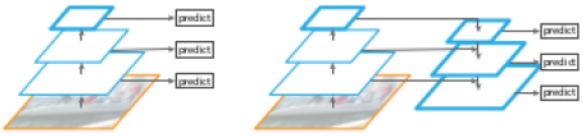
4.3.2 Feature Pyramids

These use the intermediate feature maps to compute additional predictions.

c) An alternative (**Single Shot Detector SSD**) is to reuse the pyramidal feature hierarchy computed by a ConvNet as if it were a **featurized image pyramid**.

d) Modern systems (**FPN, RetinaNet, YOLOv3**) recombine features along a **backward pathway**. This is as fast as b) and c) but more accurate. This is similar to U-net, we recombine the upsample of the small feature maps with the information of the forward pass. Slightly more complex and more slow, but we gain info.

   In the figures, feature maps are indicated by blue outlines and thicker outlines denote **semantically stronger**.

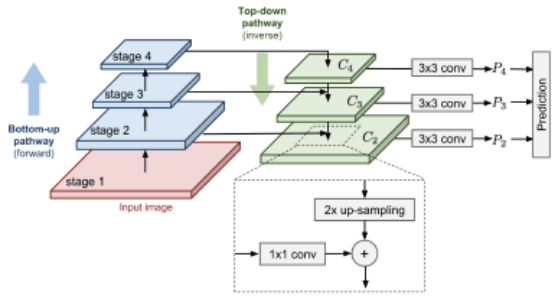


(c) Pyramidal feature hierarchy (d) Feature Pyramid Network

4.3.3 Featurized Image Pyramid

- **Bottom-up pathway** is the normal feedforward computation.
- **Top-down pathway** goes in the inverse direction, adding **coarse but semantically stronger** feature maps back into the previous pyramid levels of a larger size **via lateral connections**.
  1. First, the higher-level features are spatially upsampled.
  2. The feature map coming from the Bottom-up pathway undergoes channels reduction via a 1x1 conv layer
  3. Finally, these two feature maps are merged (by element-wise addition, or concatenation)

  1x1 Conv to reduce the dimension of the channel. Addition if same dimension otherwise concatenation over channel dimensions.



4.4 Non Maximum Suppression

YOLO v3 predicts features maps at scales 13, 26 and 52. At the end we have ((13x13) + (26x26) + 52x52)) x3 = 10647 bounding boxes.

These operations are done algorithmically and consist in two phases:

1. **Threshold by Object Confidence:** first, we filter boxes based on their objectness score. Generally, boxes having scores below a threshold are ignored.

2. **Non Maximum Suppression:** NMS addresses the problem of multiple detections of the same image, corresponding to different anchors, adjacent cells in maps.

It is an NP problem, we use greedy heuristic to get a small list of clearly separated objects.

NMS outline:

```
Divide bounding boxes BB according to the predicted class c
Each list BB_c is processed separately.
Order BB_c according to the object confidence.
Initialize TruePredictions to an empty list for a given category.
while BB_c is not empty:
    pop the first element p from BB_c (the element with highes object confidence)
    add p to TruePredictions
    remove from BB_c all elements with an IoU with p > th (remove boxes that overlap too much with the already selected box)
return TruePredictions
```

4.5 Ablation

Reading articles about Object Detection it is easy to meet discussions relative to ablative studies.

In general, ablation is **removal or destruction** of material from an object.

In the context of deep learning, it consists in a **progressive removal of components of the network**, aimed to asses their contribution to the overall behaviour.

Understanding causality in your system is the most straightforward way to generate reliable knowledge. Get rid of the noise in the research process: do ablation studies.

5 AutoEncoders

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

Usually, the internal representation has lower dimensionality w.r.t. the input. It is interesting to understand it because it is a different representation given by the NN.

Compression is possible because we exploit **regularities** (correlations) in the features describing input data.

If the input has a random structure (**high entropy**) no compression is possible: random, lawless, uncompressible, high entropy.

If the internal layer has fewer units of the input, autoencoders can be seen as a form of **data compression:**

- **Data-specific:** it only works well on data with strong correlations (digits, faces). Different from traditional data compression algorithms.
- **Lossy:** the output is degraded with respect to the input. This is different from textual compression algorithms, such as gzip.
- **Directly trained** on unlabeled data samples → **self-supervised** training because the label/ground truth is the same as input

They are not good for data compression but can be applied to **data denoising** (compression removes noise) and **dimensionality reduction** (explore internal representation - what are the main features)

Image denoising: add noise and train to reconstruct the original image. Denoising autoencoders inspired **dropout**.

Anomaly detection: Autoencoding is data specific: the autoencoder works well on data similar to those it was trained on. If applied on different data (anomaly), it will perform poorly.

5.1 Principal Component Analysis PCA

It is a statistical technique for dimensionality reduction.

Given points in d-dimensional space, linearly project them in a lower dimensional space, minimizing the quadratic error of their reconstruction.

In an image, i choose the pixels with the **highest variance** because they are more informative and so they are more likely to be interesting.

With PCA, we can choose other axis different from the original to represent data and then project them in a lower dimensional space.

Instead of minimizing the error, it is equivalent to maximize the variance since the distance of each point is from the baricenter is constant.

Higher variance preserve more original information. PCA uses the weights to asses which feature is more important.

Maximizing the variance: we fit an ellipsoid over the training data, and project over the main axes which are the **eigenvectors** of the **covariance matrix**.

An eigenvector of a matrix A is vector v such that for some scalar λ: $Av = \lambda v$.

- **Covariance**: With bi-dimensional data, the variance of the individual components x e y does not faithfully capture the variance of data. In addition to variances $\sum_{xx} = E[(X - \mu_x)^2]$ and $\sum_{yy} = E[(Y - \mu_y)^2]$ we need to compute covariances:

$$\sum_{xy} = E[(X - \mu_x)(Y - \mu_y)] = \sum_{yx} \tag{23}$$

- **Covariance Matrix** (symmetric) of X is the matrix dxd: $var(x) = E[(X - E(X))^T(X - E(X))]$. If the dataset is centered, that is $E(X) = 0$, then

$$var(X) = E[X^T X] = \frac{1}{n} X^T X \tag{24}$$

- **Eigenvectors**: Every square matrix A defines a linear transformation X $\rightarrow$ AX In particular, every **symmetric** matrix defines a "scaling" along particular directions orthogonal to each other.

  The direction of some vectors is preserved, others no. Those preserved are precisely the eigenvectors. $Av = \lambda v$ where λ is the eigenvalue of v.

  Eigenvecotors provide the **orthogonal scaling directions** and eigenvalues give the respective **intensities**

Summing up, the **covariance matrix** *expresses the deformation of data*, that is the way they are distributed or "scaled" in space. The **eigevectors** are the **directions** of such scaling, and correspond to the axes of the ellipsoid conceptually fitting data.

PCA algorithm:

1. Put data in a matrix X
2. Normalize X so that the mean along each dimension is null
3. Compute the covariance matrix $\sum = X^T X$
4. Find the eigenvectors of $\sum$ and the corresponding eigenvalues
5. Keep the m eigenvectors with largest eigenvalues

Application to face recognition

Idea is $face = meanface + \sum_k c_k eigenface_k$ which is the linear combination of the eigenfaces to get a real image.

Everyface is described by the K parameters c_k, where c_k is just the **projection** (dot product) of the face along the eigenface k.

Combining a limited number of eigenfaces suffices to reconstruct the original face with good accuracy.

When visualizing weights, white pixels denote high weights: portions of the input image relevant to discriminate objects in the given class.

Autoencoders and PCA combined: Taking a fully connected linear network with k hidden units, no activation functions, and quadratic reconstruction error as objective function, we may compute the k principal components (weights) and their respective projections (hidden value).

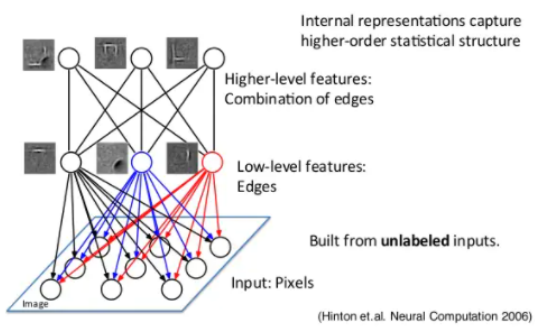
5.2 Deep Belief Networks

Goal: use deep networks to improve dimensionality reduction and compute better internal representations for data reconstruction - with non-linear combination we can get better components.

Deep Belief Networks solved the issue via local pretraining of each layer.

- **Pre-training:** unsupervised pretraining of a stack of Restricted Boltzmann Machines (RBM), each one relative to a specific hidden layer
- **Unrolling:** of the stack to build a deep encoder
- **Fine-tuning:** of the model by means of traditional backpropagation techniques.

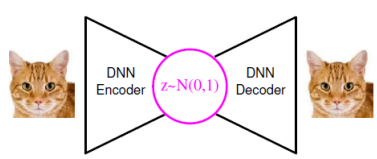
Training of the RBMs for the different layers are independent from each other.



5.3 Variational Autoencoder

An autoencoder is a net trained to reconstruct input data out of a learned internal representation (e.g. minimizing quadratic distance).

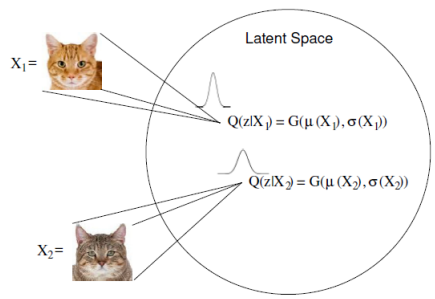
We can't use a decoder to generate data by sampling in the latent space since we don't know the distribution of latent variables.



In Variational Autoencoder we try to force latent variable to have a Normal Distribution. By adding some regulatization techniques.

We can do it with a Gaussian. It is not an arbitrary choice. If our NN is sufficiently powerful, you can always transform gaussian distribution in any other distribution.

It is suitable to think that all the points in a neighborhood of the point z encode something similar to the input X1.. because it is a gaussian. Why gaussian? Because it has the **Maximum Entropy** given the mean and the variance.



Overall ideas:

- Compute $\mu_z(X)$ and $\sigma_z(X)$ for each data point X and each latent variable z.
- Add a regulatization term to the loss function that pushes $\mu_z(X)$ towards 0 and $\sigma_z(X)$ towards 1.

  For $\mu_z(X)$ loss is $\mu_z(X)^2$ - add quadratic loss.

  For $\sigma_z(X)$ loss is $\sigma_z(X)^2 - log(\sigma_z^2(X)) - 1$.

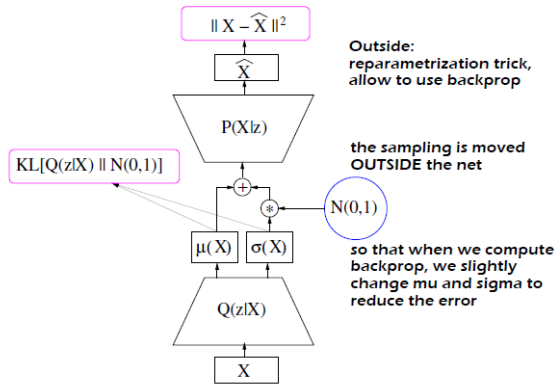
  We enlarge the variance to cover all the latent space so that we are sure that wherever we sample we get a point from which we can reconstruct the input image. Push the variance as large as possible to prevent it to become 0.

Learning the latent distribution

The encoder is supposed to compute both $\sigma(X)$ and $\mu(X)$. To do so, we sample around $\mu(X)$ using the current variance $\sigma(X)$. Use the resulting point to reconstruct X and use the **reconstruction error** to rune both parameters.

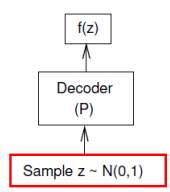
We start from the mean and then we move away but the 'step' is **smaller** than the variance, so that the new point still belongs to that distribution.

In this way we train the DNN to reconstruct the point (mean) not only given the mean, but also all the points in its neighborhood (in the circle, where r $<$ variance). Among other things, sampling add noise to the encoding improving its robustness.



The 2 terms (in regular loss) adjust the close form of KL distance between Q and N in the case Q has a normal distribution center in $\mu(X)$ and with variance $\sigma(X)$.

$\mu(X)$ and $\sum(X)$ are not used to generate new samples from the input domain.



5.3.1 Kullback-Leibler regularization

The loss function of the VAE considered is the closed form of the Kullback-Leibler distance:

$$KL(Q(z|X)||N(0,1)) \tag{25}$$

Remember that KL as objective loss function equals the negative Entropy of Q(z|X) plus the cross-entropy of Q(X) and P(z).

The actual distribution of latent variables is the marginal distribution Q(z), hopefully resembling the prior P(z) = N(0,1). It's the marginal distribution that is close to N(0,1), not the single distribution.

$$Q(z) = \sum_X Q(z|X) \sim N(0,1) \tag{26}$$

Summing up, we compute mean and variance because we want to push Q(z|X) close to a Normal distribution N(0,1) because when averaging on all X, we will get $Q(z) \sim N(0,1)$

6 Generative Models

Discriminative models (logistic regression) find a separation between classe, we are not interested in their distribution.

Generative models (Naive Bayes) understand the distribution of the classes and then possibly use it to derive discrimination.

Generative model: a model that tries to learnt he actual distribution p_{data} of real data from available samples. **Goal:** build probability distribution p_{model} close to p_{data}.

We can explicitly estimate the distribution or build a generator able to sample according to p_{model}. Mostly focus on sample generation.

We study those to improve our knowledge on the latent representation of data and the encoding of complex high-dimensional distributions; generative models, and GANs in particular, allow us to work with multi-modal outputs, forcing a choice instead of averaging; find a way to produce realistic samples from a given probability distribution; generative models can be incorporated into reinforcement learning; generate new synthetic data similar to training data (to fake human with turing test?).

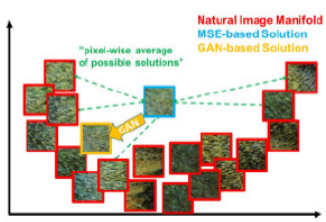
All the relevant information to generate is contained in the RDM seed.

Multi-modal output. The output of generative models is intrinsically multimodal (generate a car). If we base learning on minimizing an average distance with all elements in the training set with the "expected features", we could end up with **exceedingly blurred images** (likelihood based models).

6.1 Generative Adversarial Network GAN

Pixel-wise average of possible solutions could produce images outside the actual data manifold.

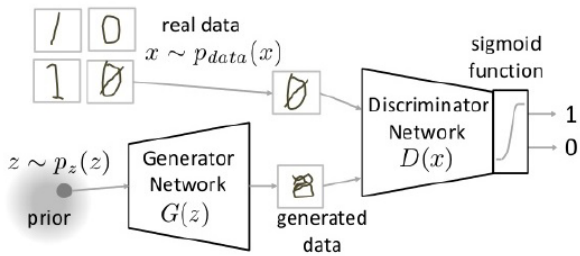
GAN drives the reconstruction towards the natural image manifold producing perceptually more convincing solutions.



Forcing to operate a choice - instead of mediating, could result in **sharper images**. We are not able to reconstruct the original image, but produce a realistic image.

A two player game - Min Max game

The winner of the game is the generator. At the end we expect the discriminator to make a random guess. The goal of discriminator is to maximize



V(D,G) while goal of generator is to minimize the previous. G cannot change real data but can generate better fake data.

$$Min_G Max_D V(D,G) \tag{27}$$

$$V(D,G) = E_{X \sim p_{data}(x)}[logD(x)] + E_{z \sim p_z(z)}[log(1 - D(G(z)))] \tag{28}$$

First term is the negative cross entropy of the discrimination w.r.t. the true data distribution - max probability that D answers 1 given x from real distribution.

Second term is the negative cross entropy of the 'false' discriminator w.r.t. the fake generator - max the probability that D answers 0 given z from sampled distribution.

Alternatively train the discriminator, freezing the generator, and the generator freezing the discriminator:

Problems with GANs:

- the fact that the discriminator get fooled does not mean the fake is good (neural networks are easily fooled)
- problems with counting, perspective, global structure, ... Those a problems related to the discriminator: how it understands what kind of features are contained inside the images.
- **mode collapse**: generative specialization on a good, fixed example. The generator always produces the same example, which is the one that has the higher probability to belong to real data (the peak).

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

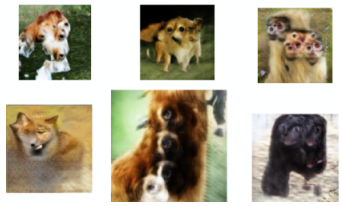
- Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
- Update the generator by descending its stochastic gradient:

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

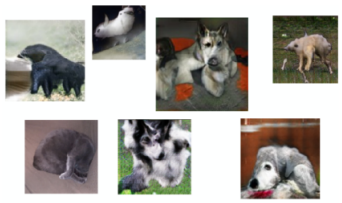
end for

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

- Problems with **counting**: GANs fail to differentiate how many of a particular object should occur at a location.



- Problem with **perspective**: to adapr to 3D objects. It doesn't understand perspective (frontview vs backview).



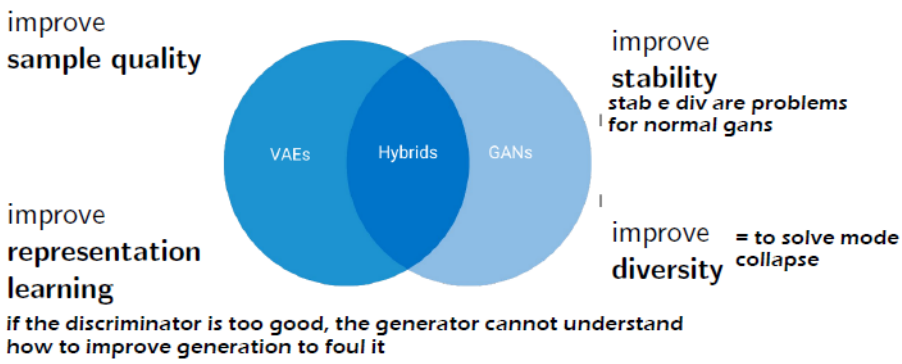
- Problem with **global structure**: to understand the holistic structure

6.2 Combining GAN and VAE

- VAE problems: the similarity metric is crucial. Pixel-wise metrics like squared error are too sensible to local perturbations like small translations or rotations.

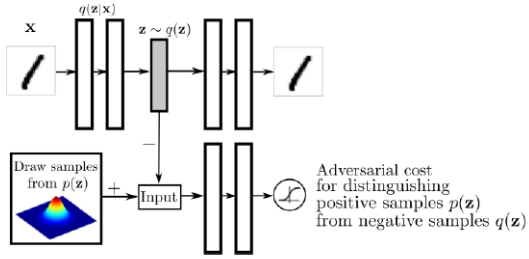
- GAN problems: the generative approach has problems to capture the real data distribution; learning is difficult and unstable

Approach on log likelihood (VAE) results in 'avg image', but no problem with mode collapse because encoder encodes all samples in latent space and decoder is trained to generate all of them.



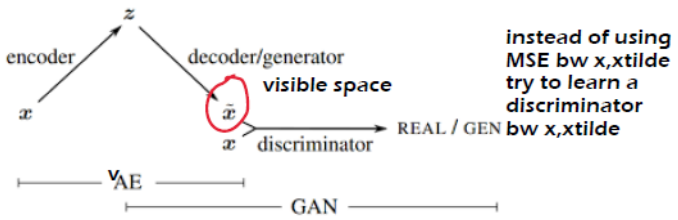
Two approaches:

- **Adversarial autoencoders:** Acting in the **latent space** replace KL-divergence with a Discriminator, matching the aggregated posterior Q(z) with the expected (arbitrary) prior distribution P(z). In the 2nd net, discriminatively predict whether a sample arises from the hidden code of the autoencoder or from a sampled distribution specified by the user. We push Q close to P and discriminate the distribution. Also we try to choose a different prior (not gaussian).



- **VAE-GAN**: Acting in the **visible space** (discriminate the images) replacing the reconstruction loss with a Discriminator trying to distinguish original from reconstructed images.

  Idea is to replace the element-wise reconstruction metric with the **feature-wise metric** learned **by the discriminator** and to improve the generative part of GAN with a variational autoencoder.



CelebA is a dataset of 202k face images annotated with 40 binary attributes such as eye-glasses, bangs, pale skin, etc.

VAE produces blurred images specially on backgrounds. GAN more sharper images.

We can exploit the rich labeling of CelebA to find **directions** in the latent space corresponding to **specific visual features** in the image space.

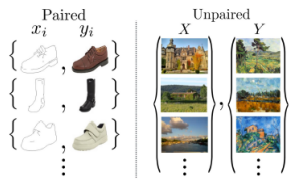
Visual attribute vector: for each attribute, compute the **mean vector** (encoded) for images with the attribute and the **mean vector** for images without the attribute, and take the difference.

Use attribute vectors to modify the latent space before reconstruction, in order to add a specific feature. Encoded image having an attribute have an offset w.r.t. the origin, indicating the presence/direction of the attribute. It's not the correct way to do conditional generation, but it is useful to understand the latent space.

6.3 Cycle GANs

We preserve the content (shape, angle) but change the texture (zebra to horse, colors).

Unpaired image-to-image because we have an image as input and an image as output. For training, we need images in two classes but **not** paired images. Unpaired means that i do not expect to have the same image of zebra or horses in the same position.



Cycle is essential to ensure to preserve the content. We have a cycle tot train the net from zebras to horses but also from horses to zebras.

A Generator and a Discriminator network playing against each other. 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.

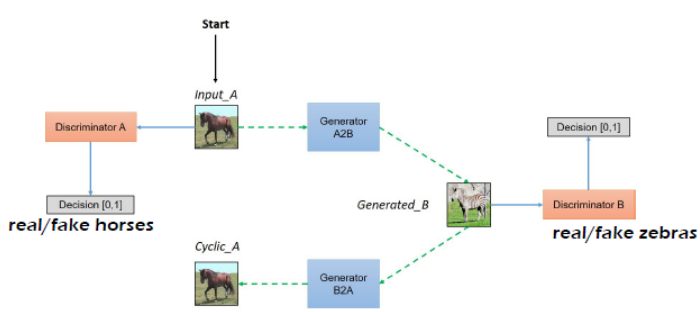
This cannot guarantee to generate an image in the target distribution related to the original image.

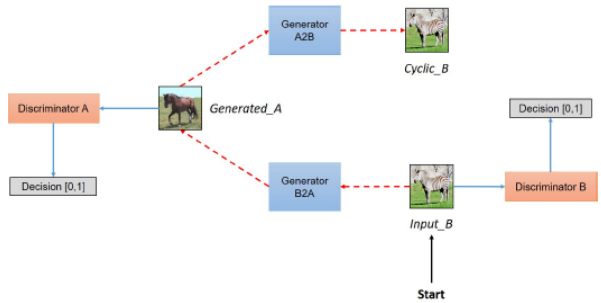
To this aim, the authors introduce the **cycle-consistency constraint**: if we transform from source distribution to target and then back again to source distribution, we would expect to obtain the original image.

Standard GANs problem: we may specialize in producing a single image for all the training images.

Cycle GAN: No problem, because we enforce to create a zebra as close as possible to the original image, such that b2a generates a horse similar to the original image.

We create as many different zebras as many different horses in training -> no mode collapse.





Loss function: Given mapping G: X $\rightarrow$ Y, F:Y $\rightarrow$ X and discriminators D_x, D_y we have:

- **Adversarial loss:**

$$L_{GAN}(G; D_Y) = E_{y \sim p_{data}(y)}[log(D_Y(Y)] + E_{x \sim p_{data}(x)}[1 - log(D_Y(G(x))] \tag{29}$$

- **Cycle consistency loss:** composition of F and G must return the identity (get the original image back.

$$L_{Cyc}(G; F) = E_{x \sim p_{data}(x)}[||F(G(x)) - x||_1] + E_{y \sim p_{data}(y)}[||G(F(y)) - y||_1] \tag{30}$$

Network structure: each mapping is defined as a sequence of convolutions followed by transposed convolutions.

To improve similarity between input and output, the authors also exploit **residuality**.

Generator with input and output of the SAME size: can we downsample??

1) fully cnn WITHOUT any downsample: if no downsample at all we do not allow the sufficient number of RFs to understand the texture of the image.

2) sth similar to Unet: downsample and upsample, but not too much downsample otherwise we do not preserve the content.

6.4 Conditional generation

Parametrize the behaviour according to the input labels.

General issue: A neural network compute a single function. Can we compute a **family** of functions instead? (a function parametric w.r.t. given attributes)

For instance, in generative model, we would like to parametrize generation according to specific attributes: generate a given digit, generate the face of an old man wearing glasses, generate a red, sportive car.

Two issues: *integrate the condition* inside the generative model, *concrete handling of the condition* because conditional 1d has different shape w.r.t. the image 2/3D

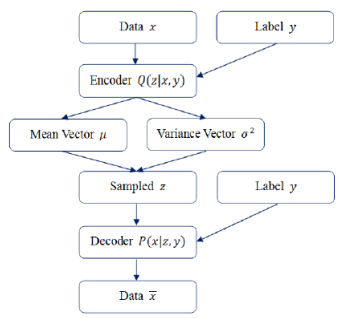
6.4.1 Conditional VAE - CVAE

Both the encoder Q(z|X) and the decoder P(X|x) are now parametrized w.r.t. a given condition c: Q(z|X, c) and P(X|z, c). What about the prior?

- We can still work with a single, **condition independent prior** (e.g. a normal gaussian) → simpler, a little more burden on the decoder side.

- We can also use a **different - possibly learned - prior** (e.g. a different Gaussian) for each condition (condition-dependent) → slightly more complex, not clearly beneficial.

In this case we have a different gaussian for each condition, with mean and variance that may depend on the condition itself.

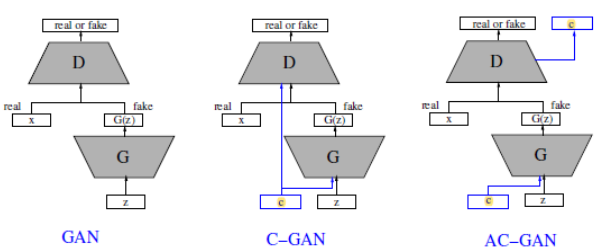


6.4.2 Conditional GANs

The generator always takes in input the condition, in addition to the noise.

Two approaches for the discriminator:

- use the condition to discriminate fakes for real of the given class (**CGAN**) → condition as input.
- try to classify w.r.t. different conditions in addition to true/fake discrimination (**Auxiliary Classifier GAN**). → condition as output.



AC-GAN loss function: the generator tries to learn the first, so first and second as close as possible:

- $p^*(x, c)$ is **true** image-condition joint distribution

- $p_\theta(x, c)$ is the joint distribution of generated data
- $q_\theta(c|x)$ is the classifier

In addition to the usual GAN objective, we also try to minimize the following quantities:

$$- E_{p^*(c,x)} ln(q_\theta(c|x)) - E_{p_\theta(x,c)} ln(q_\theta(c|x)) \tag{31}$$

Term 1: the classifier should be **consistent** with the **real** distribution - correct classify real images w.r.t. the condition.

Term 2: the generator must create images easy to classify - correct classify fake images w.r.t. the condition. A consequence of term 2 are images far from the borders.

6.4.3 InfoGAN

AC-GANs are closely related to InfoGan. Here we have only the first term of (31). The second term help to generate images **far from boundaries** between classes, and so more sharp (i.e. young and old people, not mid-aged people).

6.5 Concrete handling of the condition

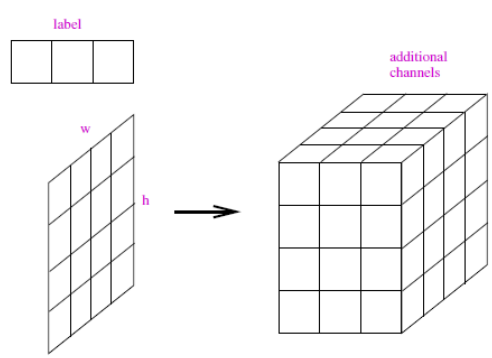
In conditional networks, we pass the label/condition as an additional input.

How is this input going to be processed?

If we need to add it to a **dense layer**, we just concatenate the label to the input. If we need to add it to a **convolutional layer**, we have two basic ways: Vectorization and Feature-wise Linear Modulation (FILM).

6.5.1 Vectorization

Correspondent of *repeatVector* in keras. Repeat the label (typically in categorical form) for every input neuron, and stack them as new channels.

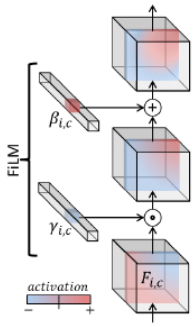


6.5.2 Feature-wise Linear Modulation - FILM

Idea: use the condition to give a different weight to each feature (each channel).

Use the condition to generate two vectors γ and β with size equal to the channels of the layer.

Rescale layers by γ and add β. Less invasive than parametrizing the weights.



Gamma and beta have a number of units equal to the number of channels (also the number of condition).

7 Recurrent Neural Networks - RNN

RNNs are good at modelling sequences. Typical problems where they are applied are:

- **turn an input sequence into an output sequence** (possibly in a different domain): translation between different languages, speech/sound recognition.
- **predict the next term in a sequence**.

  The target output sequence is the input sequence with an advance of 1 step. Blurs the distinction between supervised and unsupervised learning.
- **predict a result from a temporal sequence of states**.

  Typical of Reinforcement learning, and robotics.

Memoryless approach: compute the output as a result of a **fixed number** of elements in the input sequence. Used in Bengio's first predictive natural language model and Qlearning for Atari Games.

Difficult to deal with very long-term dependencies.

There are however *simple problems that require memory* such as the **arithmetical sum** (the carry is propagated but we do not know up to when.. so we cannot decide a priori a window typical problem: we cannot work always with a fixed dimension) and the **T-maze** (we need to remember the initial position because it is needed to make a decision at the end).

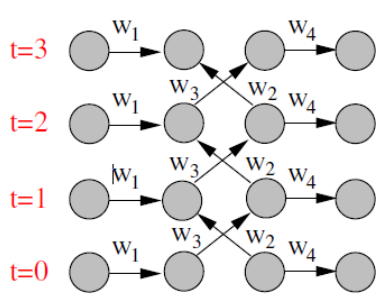
RNNs: in presence of backward connections, hidden states depend on the past history of the net. Hidden state may be updated in complex ways, according to the usual non-linear dynamics of neural nets.

Temporal unfolding: Weights are updated at precise times steps.

The recurrent net is just a layered net that keeps reusing the same weights. With temporal unfolding we get a traditional feed forward net.

It is a solution to instability: freeze the time, discretize the time.

Weights are shares: it is the content that is propagated, but when we back-propagate even the same weights get different updates.



We have cycles because in our brain we have them. Still exploring this even if our brain doesn't work exactly this way.

Sharing weights through time: It is easy to modify the backprop algorithm to **incorporate equality constraints** between weights.ù

We compute the gradients as usual, and then **average gradients** so that they induce a same update. If the initial weights started satisfied the constraints, they will continue to do.

According to bp, we get different updates for the same weight at different layers.. but we want to constraint that the weights are shared.. so we use the avg?

In feedforward NN they are shared spatially, here shared full time.

To constrain $w_1 = w_2$ we need $\Delta w_1 = \Delta w_2$ so we compute $\frac{\partial E}{\partial w_1}$ and $\frac{\partial E}{\partial w_2}$ and use $\frac{\partial E}{\partial w_1} + \frac{\partial E}{\partial w_2}$ to update both w_1 and w_2.

Backpropagation through time - BPTT: think of the recurrent net as a layered, feed-forward net with shared weights and train the feed-forward net with weight constraints. Reasoning in the time domain:

- the forward pass builds up a stack of the activities of all the units at each time step
- the backward pass peels activities off the stack to compute the error derivatives at each time step
- finally we add together the derivatives at all the different times for each weight.

Hidden state initialization

We need to specify the initial activity state of all the hidden and output units. The alternive is to use random weights but it's not good.

The best approach is to treat them as **parameters**, learning them in the same way as we learn the weights:

- start off with an initial random guess for the initial states
- at the end of each training sequence, backpropagate through time all the way to the initial states to get the gradient of the error function with respect to each initial state
- adjust the initial states by following the negative gradient

7.1 Why training RNNs is difficult

- **The backward pass is linear**.

  There is a big difference between the forward and backward passes. In the forward pass we use squashing functions (like the logistic) to prevent the activity vectors from exploding. So we get values that we can control.

  The *backward pass* is linear. If you double the error derivatives at the final layer, all the error derivatives will double.

The *forward pass* determines the slope of the linear function used for backpropagating through each neuron.

Long backprop equals to a long composition of functions and so many products. They are constant because depend on the value of the activation.

- **Vanishing and exploding gradients**.

  What happens to the magnitude of the gradients as we backpropagate through many layers? If the weights are small, the gradients shrink exponentially while if the weights are big the gradients grow exponentially.

  Typical feed-forward neural nets can cope with these exponential effects because they only have a limited number of layers.

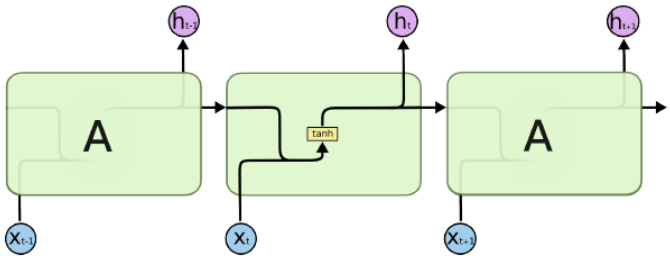
  In an RNN trained on long sequences (e.g. 100 time steps) the gradients can easily explode or vanish.

7.2 Long-Short Term Memory - LSTM

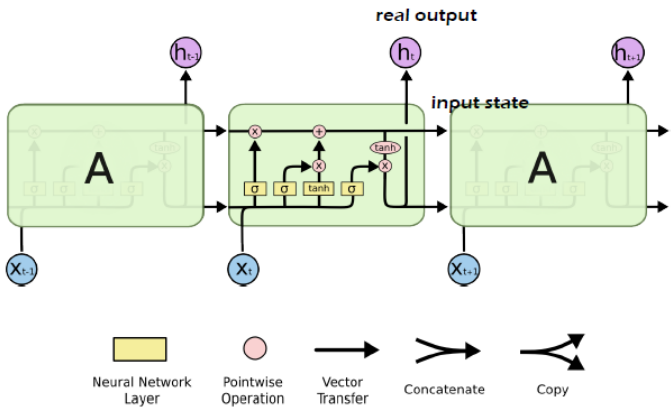
A forward link between two units must be understood as a **looping connection**.

Traditional RNN

The content of the memory cell C_t , and the input x_t are combined through a simple neural net to produce the output h_t that coincides with the new content of the cell C_{t+1}. This because it is better trying to preserve the memory cell, letting the neural net learn how and when to update it.



LSTM

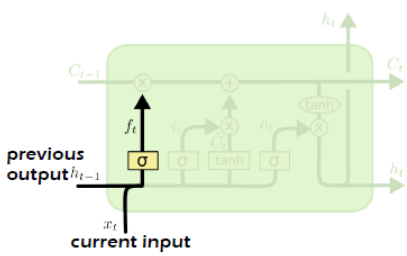


The LSTM has the ability to remove or add information to the cell state, in a way regulated by suitable **gates**. Gates are a way to optionally let information through: the product with a sigmoid neural net layer simulates a **boolean mask** (because it is a sigmoid).

What memory to preserve (close to 1) and what to delete (close to 0). Then we sum.

Attention mechanism: multiply a boolean mask to some information.

- **Forget gate:** decides what port of the memory cell to preserve/forget based on the current and previous input.

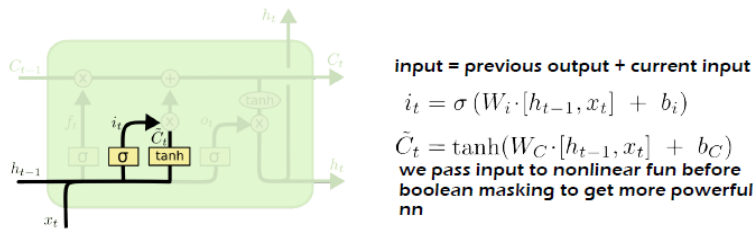


$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \ + \ b_f\right)$$

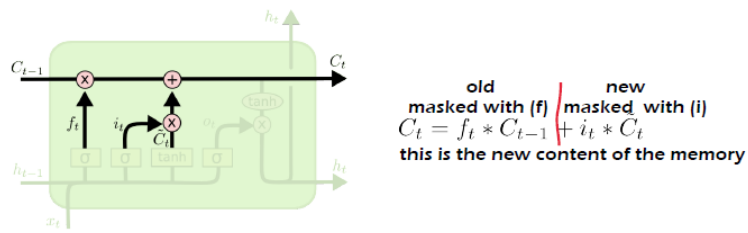
- **Update gate:**

  Input gate decides what part of the input to preserve. Tanh layer creates a vector of new candidate values C_t to be added to the state.

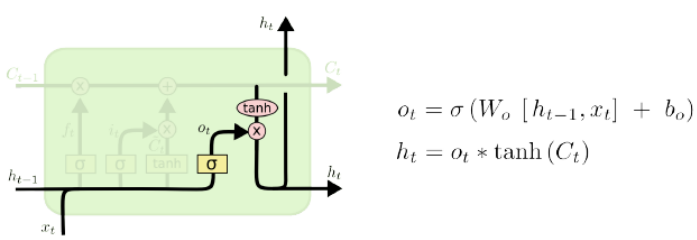
  C_t is the update but we do not directly add the update we compute a boolean mask i_t and apply to C_t before summing to C_{t-1} this to add more **parametricity** to have more powerful net.



We multiply the old state by the boolean mask f_t. Then we add $i_t * C_t$



- **Output gate:** the output h_t is a **filtered** version of the content of the cell. Output gate decides what parts of the cell state to output. The tanh function is used to **renormalize** values in the interval [-1,1].



In LSTM we set *return_ sequences* to output the same lengths as the input to stack more LSTM layers and we set *return_ state* to not return a single value, but also the value of the memory cell and the state.

8 Attention

Attention is the ability to focus on different parts of the input, according to the requirements of the problem being solved.

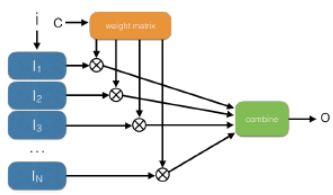
It is an essential component of any intelligent behaviour, with potential application to a wide range of domains.

From the point of view of Neural Networks, we would expect the attention mechanism to be differentiable, so that we can learn where to focus by standard backpropagation techniques.

The current approach (not necessarily the best one) is to **focus** everywhere, just **to different extents**.

Attention mechanisms can be implemented as gating functions (i.e. Boolean mask). The gating maps are dynamically generated by some neural net, allowing to focus on different part on the input at (e.g.) different times.

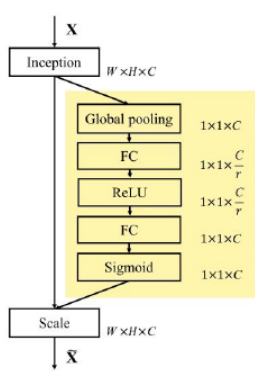
The forget map, input map and output map in LSTMs are examples of attention mechanisms because we use them to focus on some part of the input and discard the other.



8.1 Squeeze and Excitation

SE layers are a building component of **Squeeze and Excitation Networks**.

SE layers implement a form of self attention, allowing to focus on particular channels in a dynamical way, according to the input under consideration.



SE-Inception Module

Adaptively recalibrates channel-wise feature responses by explicitly modelling interdependencies between channels.

Improving the quality of representations by explicitly modelling interdependencies bw channels. Perform feature recalibration,through which it can learn to use global information to selectively emphasise informative features and suppress less useful ones.

Squeeze to extract the main components and sigmoid to get a boolean mask. Attention because we decide which channels are more important.

8.2 Key-value paradigm

The most typical attention layer is based on the key-value paradigm, implementing a sort of **associative memory.**

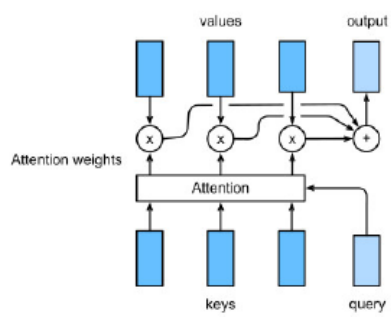
We access this memory with **queries** to be matched with keys.

The resulting **scores** generate a boolean map that is used to weight values.

For each key k_i compute the scores a_i for each pair (q, k_i) as: $a_i = \alpha(q, k_i)$.

Obtain **attention weights** via softmax: $\overrightarrow{b} = softmax(\overrightarrow{a})$.

Return a weighted sum of the values $o = \sum_{i=1}^{n} b_i v_i$. In many applucations, values are also used as keys **(self-attention)**.



Different **score functions** to lead to different attention layers.

Two approaches:

- **Dot product:** more frequent, dot product between query and key. This is an index of correlation (find the key more correlated to the q). Here query and key must have the same dimension d.

$$\alpha(q, k) = q * k/\sqrt{d} \tag{32}$$

  where $\sqrt{d}$ is a normalization.

- **MLP:** α is computed by a neural net composed by a single layer usually. We apply a dense layer to k, one to q to make them more interesting.

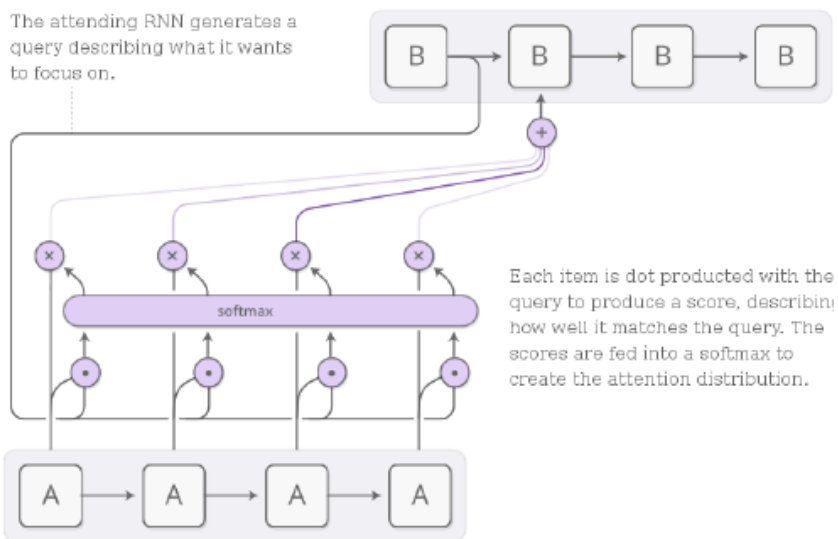
$$\alpha(k, q) = tanh(W_k \overrightarrow{k} + W_q \overrightarrow{q}) \tag{33}$$

8.3 Alignment

Alignment is a form of attention. Tio which part of the input sequence should i focus on?

Alignment: identify which parts of the input sequence are relevant to each word in the output or to other parts of the input sequence.

Translation: is the process of using the relevant information to select the appropriate output.



By letting the decoder have an attention mechanism, we relieve the encoder from the burden of having to encode all information in the source sentence into a fixed-length vector.

With this new approach the information can be spread throughout the sequence of annotations, which can be selectively retrieved by the decoder accordingly.

9 Transformers

Transformers have been introduced in **Attention is All You Need**, one of the most influential works of recent years. Transformers have rapidly become the model of choice for NLP.

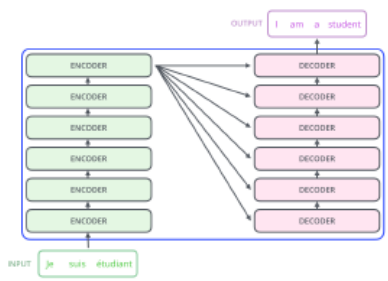
Applications like BERT and GPT, (with all relative families) are based on Transformers.

9.1 Structure

A **Transformer** has a traditional encoder-decoder structure with connections between them.

Encoding part is a stack of encoders and similarly decoding component is a stack of decoders.

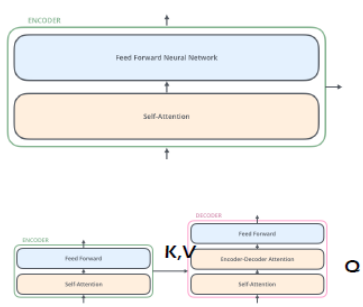
Each decoder can be influenced by the output of the final encoder, that has the best understanding of the input sequence.



They run in parallel on the whole sequence.

Encoder is organized as a self-attention layer (query, key and value are shared), followed by feedfoward component (a couple of dense layers).

Decoder is similar, with an additional attention layer that helps the decoder to focus on relevant parts of the input sentence.



9.2 Self-Attention

Self-attention: same input value as query, value, key in the decoder 2 attention layers:

a) one for previous decoder output,

b) one for encoder output, in which: query from decoder and key and values from encoder.

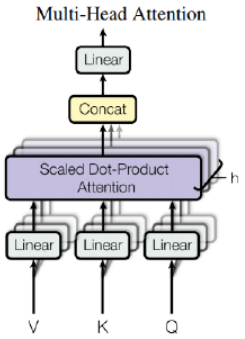
9.3 Multi-head attention

Multi-head attention: replicate operation along additional dimension. There are several attention layers running in parallel.

Using multiple heads for 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.

Split v,k,q to reduce dimensions to preserve the whole dimensionality of the input.

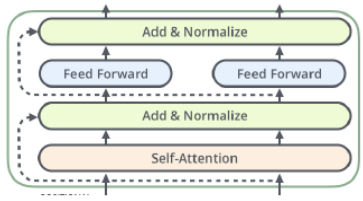


Typically, in a Transformer module, we can apply a boolean mask to the input, to hide part of its content.

This is frequently used in the decoder to prevent it to attend at future positions during generation because we want that things are generated once per time, and that the token is not influenced by the next one.

9.4 Residual connection

Each sub-layer (self-attention, ffnn) in each encoder has a residual connection around it, and it is followed by a layer- normalization step.



9.5 Positional encoding

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.

The authors use sine and cosine functions of different frequencies:

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{model}} \tag{34}$$

$$PE_{(pos,2i+1)} = cos(pos/10000^{2i+1/d_{model}} \tag{35}$$

where pos is the position of the token in the input sequence and i is the position of the token inside the PE so that in range $[0, d_{model}]$.

9.6 Sinusoids at work

According to the previous encoding, each dimension i of the PE vector corresponds to a sinusoid.

We chose this function because we hypothesized it would allow the model to easily learn to attend by relative positions, since for any fixed offset k, PE_{pos+k} can be represented as a linear function of PE_{pos}.

Property of this encoding: if i shift the encoding by k, i can compute the encoding of the shifted pos via a **fixed linear transformation** which are easy to learn.

10 Geometric Deep Learning

We can explore high dimensional data with non euclidean structure through graphs (molecules, trees, networks, manifolds).

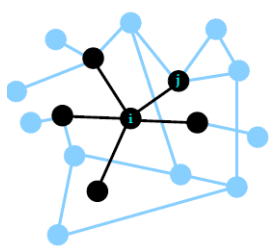
CNN: explore the correlation between each point and its correlates by using the kernel. Neighbors have fixed size and are ordered.

Graphs do not consider each node as standalone: number of neighbors is not fixed and order is irrelevant.

10.1 Graph convolutions

- local neighborhood structure modeled as a graph
- edfe feature function $h_\theta(*,*)$ parametrized by learnable θ
- permutation-invariant aggregation operator $\square$ on the neighborhood of i
- edge convolution $x'_i = \square_j h_\theta(x_i, y_i)$

$\square$ means **computation-invariant** since order is irrelevant.



Inference on graphs tasks:

- Predict labels for some nodes, from values of other nodes. Think for example of a road graph, where we want to predict global traffic knowing the traffic at particular crossroads.
- Divide the nodes into clusters not known a priori (unsupervised learning on graphs);
- Classify the entire graph, for example to predict a chemical characteristic of a given protein.

What is shared by all these problems is the need for an architecture that processes the signal taking into account the structure of the graph.

10.2 Graph Convolutional Networks

It is an approach, not the entire domain.

Given a vector of features X, a traditional layer of a Neural Network process X by means of a matrix of weights W (and an activation σ), to get a new vector of features $\sigma(XW)$.

This scheme processes each node independently, completely ignoring the connections between them (think of it as a unary convolution, independently applied to all digits).

Unfortunately, in the case of graphs, enlarging the kernel to take into account neighbours is not trivial, due to the irregularity of the graph itself, as each node can have a variable and unordered number of neighbors.

GCN consider the following simple variant:

$$GCN(X) = \sigma((A + I)XW) \tag{36}$$

where A is the **adjacency** matrix of the graph and I is the identity. A+I is fixed/given and domain specific and W is the matrix of weights learned.

We can conceive the operation as of two parts:

1. we processes the signal on each node independently by multiplying it with W;

2. the product with A + I returns a linear combination between the signals of each node and the respective neighbors.

Several layers of GCN can be combined **stacked together as normal convolutions**, to obtain architectures that "spread" information on the graph. + we just learn W as in normal convolutions.

A is typically not normalized and therefore the multiplication with A will completely change the scale of the feature vectors (the number of neighbours for each node is different).

Normalizing A such that all rows sum to one, i.e. $D^{-1}A$, where D is the diagonal node-degree matrix, gets rid of this problem.

In practice, dynamics get more interesting when using a **symmetric normalization**, i.e. $D^{-1/2}AD^{-1/2}$, that give us the GCN layer:

$$GCN(X) = \sigma(D^{-1/2}(A + I)D^{-1/2}XW) \tag{37}$$

Node-degree matrix: for each node associate the number of neighbors in the graph.

Two main typer of GCNs: spectral based and spatial based. GCN is at the intersection between them, but it is usually enrolled as spectral based.

10.3 Spectral-based approach

Rooted in signal processing; convolutions are expressed in the "Fourier" domain, exploiting the convolution theorem: $f \hat{*} g = \hat{f} * \hat{g}$

It decomposes signal in its spectral components.

GCNs are linked to the theory of spectral analysis of graphs, and in particular to **graph signal processing.**

This gives a strong theoretical justification to the methods, as well as inspiring various types of techniques to simplify modeling.

Much of this theory starts from the analysis of the so-called **Laplacian matrix.**

Let G be an undirected graph with adjacency n x n matrix A. The **(unnormalized) Laplacian** of G is $\Delta = D - A$ where $D = diag(\sum_{i \neq j} a_{ij}$ is the **degree matrix.** Δ is a n x n **symmetric positive-semidefinite matrix,** admitting an eigendecomposition $\Delta = \Theta \wedge \Theta^T$.

We use these eigenvectors as a base for a spectral transformation of the signal very similar to the classic Fourier transform , called graph Fourier transform: $\hat{x} = \Theta^T X$.

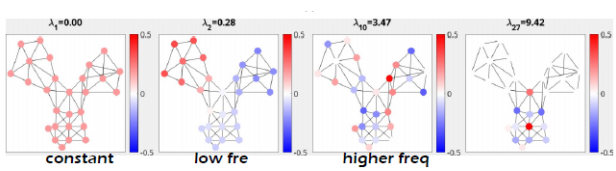
It compares [the value of the nodes weighted by the number of neighbors] with [the value of the neighbors].

How the info is stored inside the graph: information flow.

In case of a time signal, the Fourier transform allows it to be represented by a **weighted sum of sinusoids of increasing frequency**, whose coefficients are called Fourier coefficients.

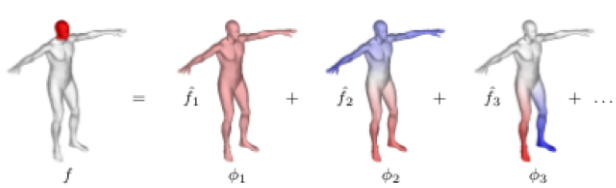
In the case of graphs, the equivalent of the Fourier coefficients is the vector $\hat{x}$, while the equivalent of the "sinusoids" are the **eigenvectors**, *sorted according to their eigenvalues.*

Each node of the graph is uniquely identified by a sequence of coefficients for the eigenvectors.



The eigevectors, essentially express the main 'vibration modes' of the graph and encode information on graphs of **increasing frequencies**.

Any surface can be uniquely expressed as a combination.



Fourier basis = Laplacian eigenfunctions: $\Delta\phi_k(x) = \lambda_k \phi_k(x)$

Learning a convolution in the spectral domain amounts to learn a diagonal matrix of parameters W for the following transformation:

$$g = \Theta W \Theta^T \tag{38}$$

Problems

- filters are **basis-dependent** $\rightarrow$ hard to generalize across domains
- $O(n)$ parameters per layer
- $O(n^2$ operations to compute forward and inverse Fourier transforms Θ^T and Θ
- No guarantee of spatial localization of filters.
- We cannon transfer knowledge.

In very rough terms, GCN corresponds to substituting multiplication by $\wedge$ with a simpler linear filter on the eigenvalues of the Laplacian matrix.

Spatial based approach works slightly better.

10.4 Spatial-based approach

Exploit the idea of **information propagation** between adjacent nodes.

A traditional convolution computes the output value for a node as a (learned) linear composition of its neighbours. Preserving the spatial structure - in GCN structure fixed in matrix A.

Similarly, we look for an efficient way to compute the output value for a node, as a **learned aggregation of its adjacent nodes**:

$$\hat{x}_i = \square_j h_\theta(x_i, y_i) \tag{39}$$

where $\square$ is the aggregation operator and h is a learnable **edge feature function**.

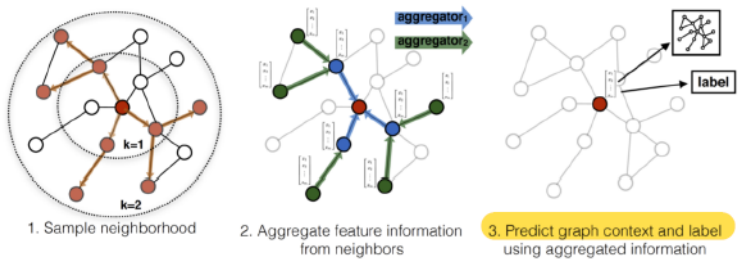
10.4.1 GraphSage

The approach of GraphSage is simple, intuitive and effective. Processing in GraphSage is essentially composed of three steps: Neighborhood sampling, Aggregation and Prediction.

1. **Neighborhood Smpling:** depth is the maximum distance of nodes to consider, V is the number of features.

   Each aggregator function aggregates information from a different number of hops - or search depth - away from a given node.

2. **Aggregation:** three functions for aggregation:



Algorithm 1: GraphSAGE embedding generation (i.e., forward propagation) algorithm

Input : Graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$; input features $\{\mathbf{x}_v, \forall v \in \mathcal{V}\}$; depth K; weight matrices $\mathbf{W}^k, \forall k \in \{1, ..., K\}$; non-linearity σ; differentiable aggregator functions $\text{AGGREGATE}_k, \forall k \in \{1, ..., K\}$; neighborhood function $\mathcal{N} : v \rightarrow 2^{\mathcal{V}}$

Output : Vector representations $\mathbf{z}_v$ for all $v \in \mathcal{V}$

$\mathbf{h}_v^0 \leftarrow \mathbf{x}_v, \forall v \in \mathcal{V}$;
for $k = 1...K$ **do**
  for $v \in \mathcal{V}$ **do**
    $\mathbf{h}_{\mathcal{N}(v)}^k \leftarrow \text{AGGREGATE}_k(\{\mathbf{h}_u^{k-1}, \forall u \in \mathcal{N}(v)\})$;
    $\mathbf{h}_v^k \leftarrow \sigma\left(\mathbf{W}^k \cdot \text{CONCAT}(\mathbf{h}_v^{k-1}, \mathbf{h}_{\mathcal{N}(v)}^k)\right)$
  end
  $\mathbf{h}_v^k \leftarrow \mathbf{h}_v^k / \|\mathbf{h}_v^k\|_2, \forall v \in \mathcal{V}$ normalization
end
$\mathbf{z}_v \leftarrow \mathbf{h}_v^K, \forall v \in \mathcal{V}$

- **Mean aggregation**: averaging all the neighborhood node features (can be weighted average)
- **LSTM aggregation**: using an LSTM cell to sequentially aggregate neighborhood node features (ordered randomly)
- **Pooling aggregation**: Max pooling only takes the "highest" feature into consideration (performed the best in experiments) - last is the most effective.

3. **Prediction and training:** What we obtain after the aggregation phase should be compared to the feature map extracted from an image after the convolutional layers (without top dense layers).

   The aggregated neighborhood node features collected by each node are used to compute a resulting value, according to the logic of the problem (node classification, structure/context determination,...).

   This is where the learning happens, according to problem-specific loss functions.

   We get a value according to the logic, we decide the lostt and we do the backward pass.

10.4.2 Mixture Model Network - MoNet

The **MoNet** model brought 3 important contributions to the field of geometric deep learning:

- a **generalization** of various Graph Learning approaches, trying to unify spatial and spectral approaches
- a new approach using **parametric kernels** and **pseudo-coordinate**s, integrating well with existing models (Anistropic CNN, Geodesic CNN,...)
- a wide series of **experiments** performed on different benchmark manifolds, graphs, and networks

For each node i and neighbour $j \in N(i)$ it is defined a d-dimensional vector of **pseudo-coordinates** $u(i,j)$.

These are fed to P learnable kernel functions $w_1(u(i,j)), ..., w_P(u(i,j))$

The **patch operator** for a signal x: $D_p(i)x = \sum_{j \in N(i)} w_P(u(i,j))x(j)$

where $x(j)$ is the signal value at the node j , and $p = 1, ..., P$ are the dimensions of the extracted patch.

Extract the relative coordinates, pass them to the kernel w, apply the result to the input value x, sum all together.

By carefully selecting pseudo coordinates $u(i,j)$ and kernel functions $w_P(u)$ many known graph convolutional network models can be viewed as a specific case of MoNet.

Table 1. Several CNN-type geometric deep learning methods on graphs and manifolds can be obtained as a particular setting of the proposed framework with an appropriate choice of the pseudo-coordinates and weight functions in the definition of the patch operator. x denotes the reference point (center of the patch) and y a point within the patch. $\mathbf{x}$ denotes the Euclidean coordinates on a regular grid. $\bar{\alpha}, \bar{\sigma}_\rho, \bar{\sigma}_\theta$ and $\bar{\mathbf{u}}_j, \bar{\theta}_j, j = 1, \ldots, J$ denote fixed parameters of the weight functions.

Method	Pseudo-coordinates	$\mathbf{u}(x,y)$	Weight function $w_j(\mathbf{u}), j = 1, \ldots, J$
CNN [28]	Local Euclidean	$\mathbf{x}(x,y) = \mathbf{x}(y) - \mathbf{x}(x)$	$\delta(\mathbf{u} - \bar{\mathbf{u}}_j)$
GCNN [32]	Local polar geodesic	$\rho(x,y), \theta(x,y)$	$\exp(-\frac{1}{2}(\mathbf{u} - \bar{\mathbf{u}}_j)^\top \begin{pmatrix} \bar{\sigma}_\rho^2 & \\ & \bar{\sigma}_\theta^2 \end{pmatrix}^{-1} (\mathbf{u} - \bar{\mathbf{u}}_j))$
ACNN [7]	Local polar geodesic	$\rho(x,y), \theta(x,y)$	$\exp(-\frac{1}{2}\mathbf{u}^\top \mathbf{R}_{\bar{\theta}_j} \begin{pmatrix} \bar{\alpha} & \\ & 1 \end{pmatrix} \mathbf{R}_{\bar{\theta}_j}^\top \mathbf{u})$
GCN [26]	Vertex degree	$\deg(x), \deg(y)$	$\left(1 - \lvert 1 - \frac{1}{\sqrt{u_1}} \rvert\right) \left(1 - \lvert 1 - \frac{1}{\sqrt{u_2}} \rvert\right)$
DCNN [2]	Transition probability in r hops	$p^0(x,y), \ldots, p^{r-1}(x,y)$	$\mathrm{id}(u_j)$

Learning weights brings usual benefits w.r.t. handcrafted ones.

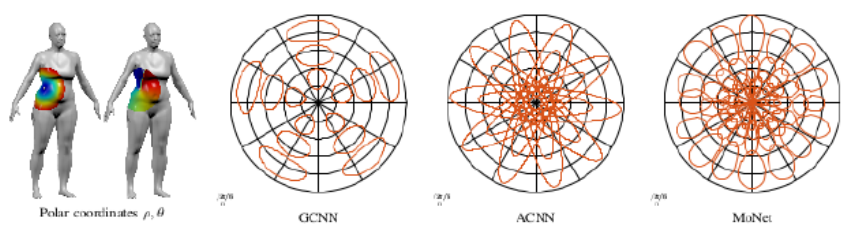


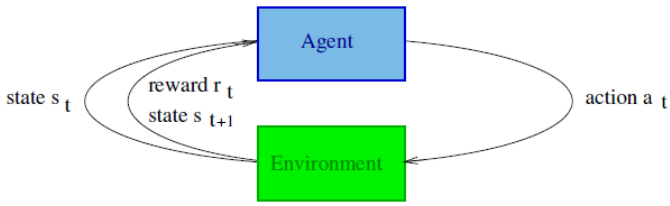
Figure 1. Left: intrinsic local polar coordinates ρ, θ on manifold around a point marked in white. Right: patch operator weighting functions $w_i(\rho, \theta)$ used in different generalizations of convolution on the manifold (hand-crafted in GCNN and ACNN and learned in MoNet). All kernels are L_∞-normalized; red curves represent the 0.5 level set.

On the 3D models are local polar coordinates centered at a point on their manifolds.

The circle diagrams are patch operator weighting functions used in different convolutional operations on the manifold. Observe the "smoother" weighting functions synthesized by MoNet.

11 Reinforcement Learning

Problems involving an **agent** interacting with an **environment**, which provides **numeric rewards**.



At time step t, the agent is in state s_t (the state contains all the info to perform the best action and do not consider the past):

- agent selects action a_t according to some policy $\pi(a_t|s_t)$
- environment answer with a local reward r_t
- and then enters into a new state s_{t+1}

A policy $\pi(a_t|s_t)$ is a **probability distribution** of actions given states.

11.0.1 Future cumulative reward

We want to learn the **best way to act** that is the **best policy.** We want to maximize the **future cumulative reward**: $R = \sum_{1 \leq i} r_i$.

We use big R for *cumulative reward* and small r for *local rewards.*

We could also take into account the fact that **distant** rewards are less likely than close ones, that are more predictable.

To this aim we multiply the reward by a discount rate $0 < \gamma \leq 1$ *exponentially decreasing with time.*

Future discounted cumulative reward: $R = \sum_{1 \leq i} \gamma^i r_i$.

11.0.2 Markov Decision process

Typical formalization of the problem is through the **Markov Decision process.** We cannot use this technique if there's no compliance with this property.

Markov property: current state completely characterises the state of the world: future actions only depend on the current state.

Defined by a tuple (S, A, R, P, γ) where S is the set of possible states, A possible actions, R the reward probability given (state, action) pair, P the transition probability to next state given (state, action) pair and γ the discount factor.

At time step $t = 0$, the environment is in state s_0. Then for $t = 0$ until done:

- agent selects action a_t according to some policy $\pi(a_t|s_t)$

- environment samples reward $r_t \sim R(r_t|s_t, a_t)$ so reward is not always deterministic.
- environment samples next state $s_{t+1} \sim P(s_{t+1}|s_t, a_t)$

A policy produces **trajectories** or paths $s_0, a_0, r_1...$ and we want to find the **optimal policy** that is:

$$\pi^* = \arg\max_{\pi} E \sum_{t \geq 0} \gamma^t r_t \tag{40}$$

which is the policy that maximize the expectation of the future rewards and where the average is taken over all possible trajectories.

The transition from state st to state s_{t+1} is not always deterministic, but governed by some probability $P(s_{t+1}|s_t, a_t)$.

If the learning model needs to learn this probability $P(s_{t+1}|s_t, a_t)$, then it is called **model-based.**

In **model-free approaches**, this information is left implicit: you learn to take actions from past experience relying on trial-and-error. We shall mostly investigate model free approaches.

If we know in which state(s) we will end up, we can explore the future (as a tree).

All techniques have to deal with the exploration/exploitation trade-off - to acquire experience we need to act randomly, but if we act too randomly we exploit too little.

- **Exploration** is finding more information about the environment.
- **Exploitation** is taking advantage of the available information to maximize the reward.

Better to explore at start and then exploit.

Two basic techniques:

- **Value-based**: We try to evaluate each state s with a value function V(s). The *policy is implicit*: we shall choose the action taking us to the next state with the best evaluation. It evaluates how good a state is.
- **Policy-Based**: we directly try to **improve the current policy**, hopefully optimizing it. Remember that the policy defines the agent behavior at a given state: $a = \pi(s)$.

  where, $\pi(s)$ is the probability to perform a in state s.

11.1 Value-based approaches - Qlearning

How good is a state? **Value function:** is the expected reward i can get starting from state s. It's not useful if i don't know in which state i will end up if i perform a specific action.

$$V(s) = E_{s_0=s} \sum_{t \geq 0} \gamma^t r_t \tag{41}$$

How good is an action a for state s? **Q-function**: is the expected reward i can get starting from s and performing action a:

$$Q(s,a) = E_{s_0=s, a_0=a} \sum_{t \geq 0} \gamma^t r_t \tag{42}$$

Expectations are on all trajectories defined by the given strategy.

We can easily compute V from Q: $V(s) = \sum_a \pi(a|s) * Q(s,a)$ i.e. we sum every action-value weighted by the probability $\pi(a|s)$ to take that action.

However, to compute Q from V we would need **(model-based!!)** knowledge of the state s' we are likely to end up by taking action a:

$$Q(s,a) = \sum_{s'} P(s'|s,a) * V(s') \tag{43}$$

The **optimal Q-value function** $Q*(s,a)$ is the maximum expected cumulative reward achievable from state s performing action a:

$$Q(s,a) = \max_{\pi} E_{s_0=s, a_0=a} \sum_{t \geq 0} \gamma^t r_t \tag{44}$$

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

The **Bellman equation** expresses a relation between the solution for a given problem in terms of the solutions for subproblems.

Q^* satisfies the following Bellman equation: $Q^*(s,a) = E_{s'}[r_0 + \gamma max_{a'} Q^*(s',a')]$.

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

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

We know that Q^* satisfies the Bellman equation.

The idea is to use it to perform iterative update on progressive approximations Q^i of Q^*:

$$Q^{i+1}(s,a) = Q^i(s,a) + \alpha(r_0 + \gamma max_{a'} Q'(s',a') - Q^i(s,a)) \tag{45}$$

where α is the learning rate, first term is the current estimation and second term is the **recursive update**.

The recursive update is the derivative of the quadratic distance between $Q^i(s,a)$ and $r_0 + max_{a'} Q^i(s',a')$ that should be equal, according to the Bellman equation.

Pseudo code:

```
initialize the Q-table
repeat 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 with (45)
```

In order to perform an **update**, all the information we need is contained in a tuple (transition): $(s, a.r.T, s')$ where: s is the current state, a is the action done, r is the reward obtained, T is a boolean stating the termination of the episode, s' is the new state after doing the action.

Q-learning transitions.

Transitions (s, a, r, T, s') are collected by exploring the environment. Each tuple is **independent** from the others.

They can be saved into an **experience replay buffer** and re-executed at leisure. Better than learning from batches of consecutive samples because:

- consecutive samples are correlated) inefficient learning
- great risk of introducing biases during learning by exploiting unbalanced sets of transitions.

Q-learning is an **off-policy technique**: it does not rely on any policy, and only needs local transitions (tuples).

Exploration vs Exploitation

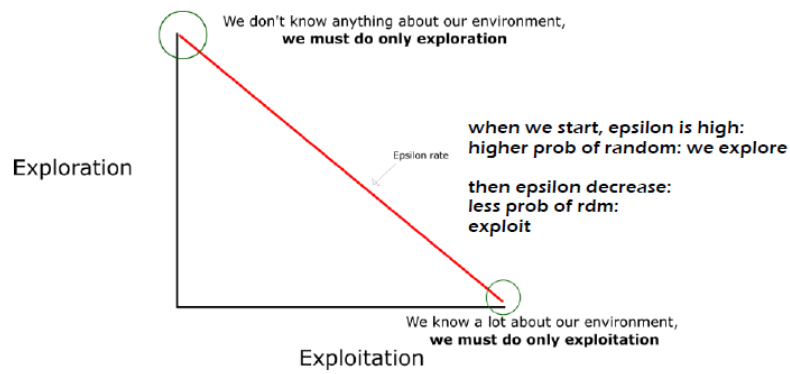
Using the Q-table at start is not informative since it introduces bias and prevent exploration, so we don't use it. We first move randomly for exploration and start relying more on the table when more experience is acquired.

Issue: too random will end up in an area which is close to the point where we started, so we don't really explore the whole space but only a neigh of the start.

11.1.1 Epsilon greedy strategy

We specify an exploration rate ϵ, initially equal to 1. This is the rate of steps that done **randomly**.

We generate a random number. If this number is larger then ϵ, then we choose the action according to the information collected in the Q-table (exploitation); otherwise we choose the action at random (exploration). We progressively reduce ϵ along training but at start it is the max (1).



Pseudo code revisited

initialize the Q-table, Replay Buffer D, $\epsilon = 1$
repeat for the desired number of episodes:
  initialize state s
  repeat until termination of the episode:
  - with probability ϵ choose a random move a
    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') in D
  - sample random minibatch of transitions (s, a, r, T, s') from D
  - **for each** transition in the minibatch:
    - compute reward

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

questa sostituita da nn più avanti

- iterative update

$$Q(s, a) \leftarrow Q(s, a) + \alpha(R - Q(s, a))$$

choose max: since Q is an appox, we can overestimate it

decrement ϵ

The grid world is an **abstraction**. Each cell is a different state and we can pass from a state to another taking actions.

The visibility of the agent is **confined to its current cell**. He can only choose an action; it is the action that determines the new state we end up into.

We will see that little by little, by **trial and error**, the agent will discover the good way of action that will allow him to reach a winning position starting from any state.

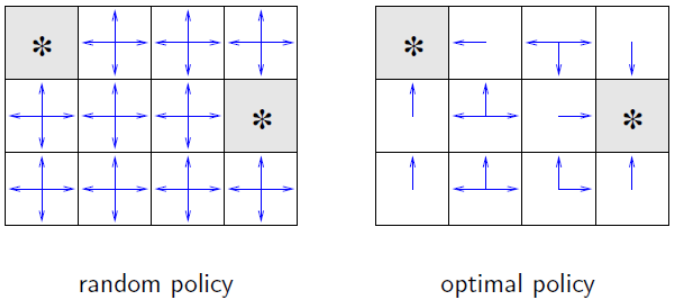


Figure 3: Policies for Grid World

If $s \rightarrow_a s'$ then Bellman's equation:

$$Q^*(s,a) = r + \max_{a'} Q^*(s',a') = -1 + V^*(s') \tag{46}$$

so the current state is equal to the local reward plus the state where i finish which is equal to -1 (cost of the action - negative reward for each transition) plus the state in which i finish.

V-value instead is just the **max of Q-values** over all possible actions: $V(s) = \max_a Q(s,a)$

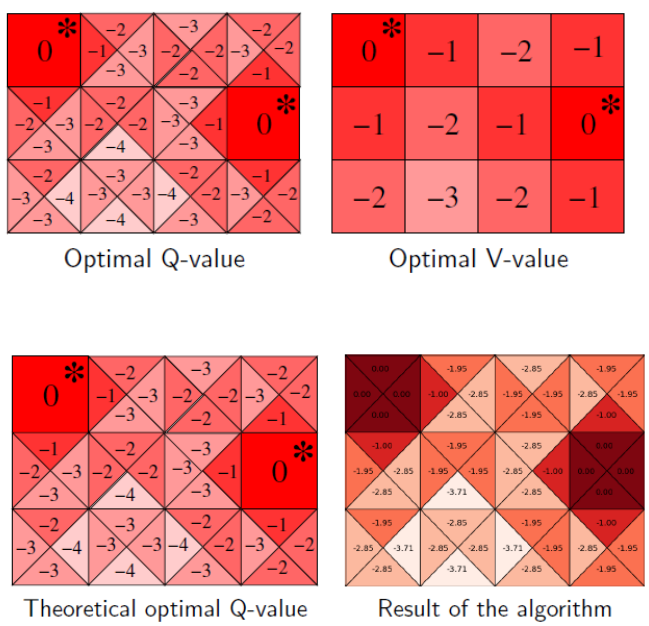


Figure 4: Result after 10000 iterations

When we start we only know the tight values for terminal states, while other states will get a random value.

Most of the actions produce meaningless updates, since the current estimation of the Q- value function is erroneous.

The relevant actions are those leading to states whose Q-value is accurate; at the beginning these are just terminal states.

11.2 Deep Q-learning

Q-learning exploits the Bellman equation. We compute Q^* via iterative updates, but it is not scalable. We must compute Q(s,a) for every state-action pair.

Deep Q-learning: use a **function approximator** (a Neural Network) to estimate the optimal action-value function.

$Q(s, a, \theta) \sim Q^*(s, a)$ where θ are the function parameters to be learned.

Instead of taking a as input, it is customary to return a value for each possible action a (the two functions are isomorphic).

In Q-learning we get in input a pair (state, action) and output the Q-value for that pair. While in DeepQ we have in input a state and compute the Q-value for each pair of that state with any action.

Loss function: try to minimize the distance between the current Q-value estimate of the network and the expected value given by the Bellman equation:

$$L(\theta) = (E_{s'}[r_0 + \gamma \max a' Q(s', a')] - Q(s, a, \theta))^2 \tag{47}$$

To compute it, we need transitions $(s_i, a_i, r_i, T_i, s_{i+1})$.

We can store these transitions in an **experience memory**, and **replay** them at leisure during training (**experience replay**).

Better than learning from batches of consecutive samples because: samples tend to be correlated) $\rightarrow$ inefficient learning plus a great risk of introducing biases during learning.

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

11.2.1 Atari Q-learning

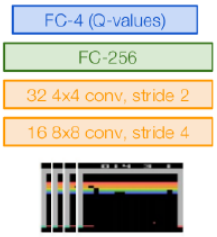


Figure 5: Architecture Atari Q-learning

State: 84x84x4 stack of last 4 frames (RGB → grayscale conversion, rescaling)

A single feed forward pass to compute Q-values for all actions from the current state.

Last layer has an output for each action a, returning Q(s,a). Number of actions between 4-18 depending on Atari game.

We need a sequence of frames to *capture movement*. As an alternative you can use an LSTM layer, after processing the state and before computing Q-values.

The same network architecture was applied to all games but retrained for each one of them with as input the screen frames. Stated that it works well for **reactive games**, not for planning.

Recently extended to Augmentation Imagination 2017.

Rewards are very different from game to game.

How can we use a same architecture? The solution adopted in **Playing Atari with DRL** was to crop all positive rewards to 1, and all negative rewards to -1.

The lesson to be learned seems to be that the entity of the reward does not really matter: what matters is to distinguish between positive, negative, and neutral moves.

An alternative was to use RAM instead of screenshots. The RAM for the Atari 2600 console consists of 128 bytes and exploit it as state observations.

11.3 Improvements

First 4 techniques are the most used and important while last three (Noisy nets, distributional RL and rainbow) are new techniques.

11.3.1 Fixed Q-targets

With Q-learning, we try to approximate the optimal target Q-function Q^*, through progressive updates: $Q^*(s,a) - Q(s,a)$.

Moreover, we approximate the computation of $Q^*(s,a)$ as $r_0 + \max a' Q(s',a')$, giving us the q-learning update:

$r_0 + \gamma \max a' Q(s',a') - Q(s,a)$ where the first term is the approximated target Q^* and the second one the current estimation.

The Q function is computed by a neural network Q(s,a,w).

In the loss function, the same network is used to provide two different estimations of the Q-function **(shared weights)**:

$$r_0 + \gamma \max a' Q(s',a',w) - Q(s,a,w) \tag{48}$$

At every step of training, the Q value shifts but also the "target value" shifts. We are getting closer to our target but the target is also moving, that may lead to big oscillations in training.

A solution is to **use a separate network** $\overline{Q}$ with **fixed parameters** for estimating the TD target. Periodically copy the parameters from Q to $\overline{Q}$ to update the target network.

The line over Q means 'fixed'/more stable.

Implementation of Fixed Q-targets:

- create two (identical) networks: DQNetwork, TargetNetwork
- define a function to transfer parameters from DQNetwork to TargetNetwork
- during training, compute the TD target using our target network. Update the target network with the DQNetwork every tau steps (tau is a user-defined hyper-parameter).

11.3.2 Double Q-learning

The approximation of target action value is computed using a **maximum over actions:**

$$r_0 + \gamma * \max_{a'} Q(s',a') \tag{49}$$

where the first term is the local reward for taking action a and second one is the max Q-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 **overestimation** of the correct value.

Overestimation is an issue because we want to be as more precise as possible.

The Double Q-learning approach consists in decoupling the choice of the action from its estimation, using two networks Q^A and Q^B.

If we use Q^A to select best action $a^* = \arg\max_a Q^A(s',a)$, the value of $Q^A(s',a^*)$ could be biased by the choice (we know it will be large: it was the maximum!).

```
Initialize Q^A, Q^B, s
repeat
    choose a using ϵ, Q^A, Q^B; observe r, s'
    choose (e.g. random) between UPDATE-A and UPDATE-B
    if UPDATE-A: # use Q^A to choose action, Q^B to estimate it
        a* = arg max_a Q^A(s', a)
        Q^A(s, a) ← Q^A(s, a) + α(r + Q^B(s', a*) − Q^A(s, a))
    else if UPDATE-B: # use Q^B to choose action, Q^A to estimate it
        a* = arg max_a Q^B(s', a)
        Q^B(s, a) ← Q^B(s, a) + α(r + Q^A(s', a*) − Q^B(s, a))
    s ← s'
until end
```

Since Q^B was updated on the same problem, but with a **different set of experience samples**, $Q^B(s', a^*)$ provides a better, unbiased estimate for the value of action a^*.

Use Q^A to find a^*, then use Q^B to estimate (s', a^*).

11.3.3 Prioritized Experience Replay

Prioritized Experience Replay (PER) exploits the idea is that some experiences may be more important than others, and thus should be replayed more frequently.

We want to give *higher priority* to transitions for which there is a *large difference between our prediction and the expected target.*

For a transition $t = (s, a, r, F, s')$, its update is

$$\delta_t = r + \gamma \max_{a'} Q(s', a') - Q(s, a) \tag{50}$$

and we set its priority to $p_t = |\delta_t|$.

Use delta to decide the relevance of a transition: higher prob to be selected from the replay buffer to Qs having higher delta.

Stochastic prioritization: The probability of being chosen for a replay is computed according to the following rule:

$$P_t = \frac{p_t^\alpha}{\sum_t p_t^\alpha} \tag{51}$$

normalization over the sum of all values to get a probability value.

If $\alpha = 0$, all transitions have same probability; if α is large, it privileges transitions with high priority p_t.

However, we are introducing a bias toward high-priority samples (more chances to be selected), with a risk of over-fitting over the small portion of experiences that we presume to be interesting.

We can **correct the bias with importance sampling weights**. If N is the dimension of the replay buffer, then $w_t = (N * P_t)^{-\beta}$ that compensate the non-uniform probabilities P_t when $\beta = 1$ (if some transition has a high probability, we reduce its weight).

These weights are folded into the Q-learning update by using $w_t \delta_t$ instead of δ_t .

For stability reasons, weights are normalized by $1/(\max_t w_t)$ so that they only scale the update δ_t downwards. β is initialized to 1 and goes to 0 during training. When it it equal to 1 means all selected with the same probability.

11.3.4 Dueling

Each Q-value Q(s,a) estimates how good it is to take action a in state s: it depends both on the action a and the value of the given state s.

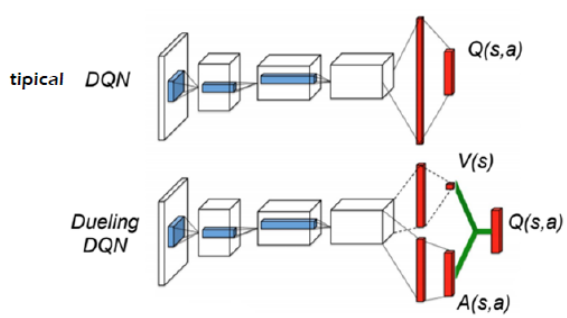
We can decompose Q(s,a) as the sum of:

- V(s): the value of being at state s
- A(s,a): the **advantage** of taking action a in state s, measuring how much better is to take action a versus all other possible actions at that state.

$$Q(s, a) = V(s) + A(s, a) \tag{52}$$

where V is the value and A the advantage which can be also negative.

Dueling Network Architectures (DDQN) split the computation of V(s) and A(s,a) in two different streams:



There are 2 stacks of dense layer, one to predict V and one to predict A and then combine them by sum.

Intuitively, the dueling architecture can learn **which states are (or are not) valuable**, without having to learn the effect of each action for each state.

This is particularly useful in states where actions *do not affect the environment in any relevant way.*

Conversely, in states where the action is relevant, it can focus on the advantage without caring for the current evaluation of the state.

In saliency maps on Enduro the assumption is that you can't go out of road.

For estimating Value, the net focus on the horizon and score. If no car is close by the action is not relevant and when you pay attention to the car in front an action is crucial.

Naif Aggregation:$Q(s,a) = V(s) + A(s,a)$

Problem: given Q(s,a), it is impossible to recover V(s) and A(s,a) and hence it is difficult to distribute the error during backpropagation (identifiability).

Previous formula is wrong as we force A to have **zero advantage** for the best action a^* so we subtract the best action:

$$Q(s,a) = V(s) + A(s,a) - max_a A(s,a) \tag{53}$$

where second and third term are zero if $a = a^*$.

Experimentally, replacing max with **mean seem to work better**:

$$Q(s,a) = V(s) + A(s,a) - mean_a A(s,a) \tag{54}$$

Subtracting the mean value helps to improve stability during training.

11.3.5 Noisy Networks

Is an alternative to epsilon policy. We ad noise to NN because we want a choice a bit more random than simply greedy. Because it is difficult to transform the evaluation of Q in a probability distribution.

Noisy Networks are characterized by **noisy dense layers**, combining a deterministic and a noisy stream (double the layer):

$$y = b + Wx + b_{noisy} \odot \epsilon^b + (W_{noisy} \odot \epsilon^w)x) \tag{55}$$

- $W, b, W_{noisy}, b_{noisy}$ are learned parameters
- ϵ^b, ϵ^w are randomly generated.

This layer is used in substitution of any standard dense layer (doubling the number of parameters). We add noise that depend on the states - in some cases we want to be noisy, other times not, so we learn b_n and W_n.

The purpose of the noise is to augment the randomicity in the choice of actions and we want the net to learn the amount of noise needed.

Since noisy-weights are learned, and the resulting noise is state dependent, this allows the network to randomly explore the environment at different rates in different parts of the state space.

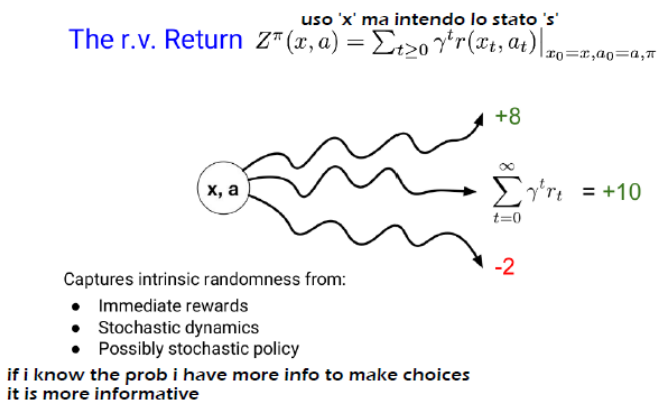
Noisy networks replace and seem to work better than the ϵ-greedy strategy.

11.3.6 Distributional RL

Distributional RL try to learn the probability **distribution** of the future cumulative reward, instead of the traditional approach of modeling the **expectation** of this return.

Specifically, DRL addresses the random return Z whose expectation is the value Q.

Expectation is learn the mean of a random variable but here is the learning of the whole distribution of a random variable, not only the mean.



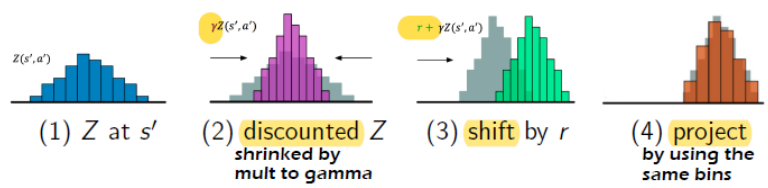
Distributional Bellman

Let a^* be the best possible action in s'. Bellman equation is $Q(x,a) = R(x,a) + \gamma Q(x', a^*)$ and similarly:

$$Z(s,a) = R(s,a) + \gamma Z(s', a^*) \tag{56}$$

Q are to be interpreted as the mean of a distribution and Z as the distributions.

The distribution is discretized over a support in a given range between V_{min} and V_{max} using a fixed number of bins (e.g. 51).



The loss between the two distribution can be computed with KL-divergence or Cross-Entropy.

11.3.7 Rainbow

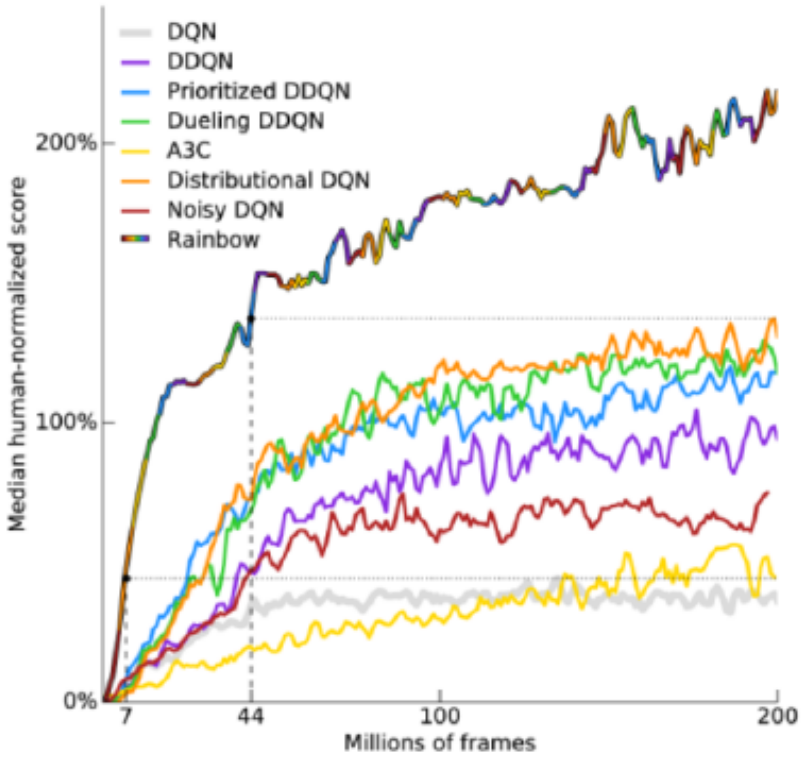


Figure 6: Combined techniques vs Single techniques

11.4 Policy Gradient techniques

Q-learning is a 1-step method since it updates the action value Q(s,a) toward the one-step return $r + \gamma max'_a Q^i(s', a')$. (slow and if we fix s' then it does not update s).

This only directly affects the value of the state action pair (s, a) that led to the reward.

The values of other 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 relevant preceding states and action.

Can we learn a policy directly, e.g. finding the best policy from a collection of policies?

Q-learning is an **off-policy technique**: it does not rely on any policy, and only needs local transitions - can take advantage of experience replay and there's

clear separation between exploration and exploitation.

On-policy techniques try to improve the current policy: it requires sampling long trajectories according to the current strategy - need many diversified trajectories (e.g. parallel agents).

11.4.1 SARSA

It is not a policy gradient technique because it does not compute directly a policy. It is based on **mini-trajectories**: $s_i, a_i, r_i, s_{i+1}, a_{i+1}$).

For this reason, SARSA is traditionally considered on-policy because we acquire experience to some policy at some point.

State-Action-Reward-State-Action (SARSA) is a learning algorithm very similar to Q-learning.

- update for Q-learning:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha[r_t + \gamma max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \tag{57}$$

- update for SARSA:

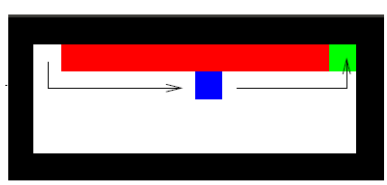
$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha[r_t + \gamma max_a Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)] \tag{58}$$

Instead of considering the best action at time t+1 (greedy choice) we consider the actual action a_{t+1} under the current policy.

Mouse and cliff scenario

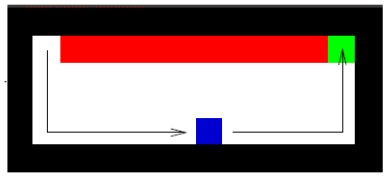
A mouse (blue) is trying to get to a piece of cheese (green). Additionally, there is a cliff in the map (red) that must be avoided, or the mouse falls and dies.

- via Q-learning: With Q-learning, the mouse ends up running along the edge of the cliff, but occasionally jumping off and plummeting to its death.



- via SARSA: With SARSA, the mouse learns that from time to time it commits errors. The best path is not to run straight to the cheese along the edge of the cliff but taking a safer root, far away from it. Even if a random action is chosen there is little chance of it resulting in death.

  It is able to understand the danger and to privilege the 'safer' path thanks to $(a_(i + 1))$ because it understand that here's high probability to die in the next moves.



11.5 Policy gradients

Directly learn the policy.

How can we improve a strategy? Suppose to have a class of parametrized policies $\Pi = \{\pi_\theta\}$.

For each policy π_θ, define its value:

$$J(\theta) = E \sum_{t \geq 0} \gamma^t r_t \tag{59}$$

We want to find the optimal policy $\theta^* = argmax_\theta J(\theta)$. As usual, we address the problem by **gradient ascent** on policy parameters.

REINFORCE approach is one of the most basic technique and updates the policy parameters θ in the direction

$$\nabla_\theta log\pi(a_t|s_t, \theta)R_t \tag{60}$$

(nice theory but no time to investigate) **Intuition**: if R_t is high push up the probability of actions, if R_t is low push down the probability of actions. Privilege action according to the future cumulative reward and weighted by the current probability of the action.

Problem: the raw value of a trajectory is not so meaningful. E.g. if rewards are all positive, we keep pushing up probabilities of all actions.

What matters is whether a reward is better or worse than expected.

Idea: Introduces a **baseline** dependent on the state. The new estimator becomes:

$$\nabla_\theta log\pi(a_t|s_t, \theta)(R_t - b(s_t)) \tag{61}$$

This difference is called **advantage**: measure of quality of an action.

The baseline decreases the variance in the gradient. Introducing a baseline is always useful because it simplifies a task for a NN, e.g. in A we subtracted mean/max.

11.5.1 The Actor-critic architecture

An excellent choice for the baseline is the value function of the state: $b(s_t) = V^\pi(s_t)$

This approach can be viewed as an **actor-critic** architecture where the policy π is the actor and the **value function** (baseline) is the **critic**.

Most parameter of actor and critic are share. In Atari $t_{max} = 5$ and $\gamma = 0.99$

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 $T > T_{max}$

perform a full episode until:
- reach the end of the episode
- reach the max #steps fixed
we collect info to compute R?

go backward

Figure 7: A3C Pseudocode

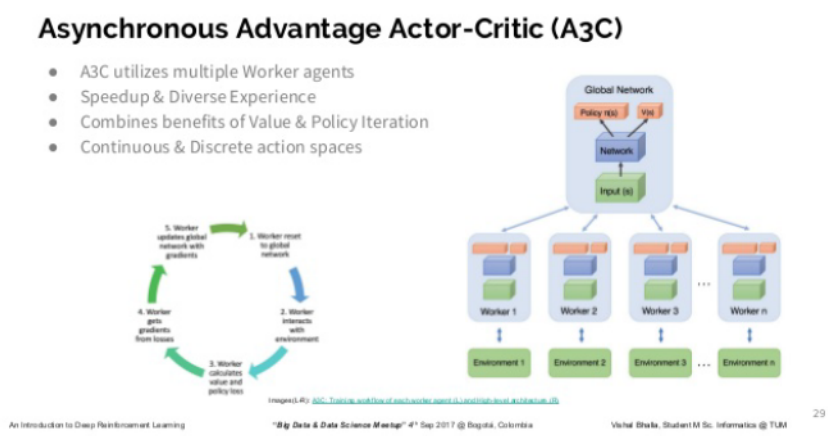


Figure 8: A3C structure

A2C is similar to A3C but without asynchronous agents; essentially, it is a single-worker variant of A3C. Empirically, A2C produces comparable performance with A3C while being more efficient.

Our synchronous A2C implementation performs better than our asynchronous implementations - we have not seen any evidence that the noise introduced by asynchrony provides any performance benefit. This A2C implementation is more cost-effective than A3C when using single-GPU machines, and is faster than a

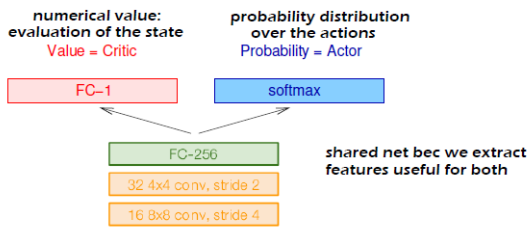


Figure 9: A3C architecture for Atari

CPU-only A3C implementation when using larger policies.

11.5.2 Trust Region Policy Optimization & Proximal PO

When passing from a given policy π_k (e.g. a randomized version of the current policy) to a new policy π, small modifications can easily results in large fluctuations in behaviours and performances.

How can we take the biggest possible improvement step on a policy, without stepping so far that we accidentally cause performance collapse?

TRPO

Let π_θ denote a policy with parameters θ.
In theory, the TRPO update is:

conditioned on a constrained:
the diff bw the params is not too big

$$\theta_{k+1} = \arg\max_\theta \mathcal{L}(\theta_k, \theta) \text{ s.t. } \bar{KL}(\theta||\theta_k) \leq \delta$$

avoid too large update of params

where $\mathcal{L}(\theta_k, \theta)$ is the surrogate advantage, a measure of how new policy π_θ performs relative to the old policy π_{θ_k} using data from the old policy:

A is the advantage
(R(s)-V(s))

ratio bw new prob of taking action a
and old prob..

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

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

$$\bar{KL}(\theta||\theta_k) = \mathbb{E}_{s\sim\pi_{\theta_k}} KL\left(\pi_\theta(\cdot|s)||\pi_{\theta_k}(\cdot|s)\right)$$

compare the distrib according to KLdist

The previous theoretical update is hard to implement. TRPO makes approximations based on Taylor expansions, to improve effciency.

PPO is a simplification of TRPO and achieves a similar objective of TRPO by updating policies:

$$\theta_{k+1} = argmax_\theta E_{s,a\sim\pi_{\theta_k}}[L(s,a,\theta_k,\theta)] \tag{62}$$

where

$$L(s,a,\theta_k,\theta) = min(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s,a), g(\epsilon, A^{\pi_{\theta_k}}(s,a))) \tag{63}$$

and $g(\epsilon, A)$ is equal to $(1+\epsilon)A$ if A is greater or equal to 0 or $(1-\epsilon)A$ if A is lesser than 0. Like this we can distinguish two cases:

- **Positive advantage:** When the advantage is positive, the objective will increase if the action becomes more likely, that is, if $\pi_\theta(a|s)$ increases.

  The min in this term puts a *limit to how much the objective can increase*, with a ceiling at $(1+\epsilon)A^{\pi_{\theta_k}}(s,a)$.

  If ratio is too large, clip modification, not higher than 1+epsilon.

- **Negative advantage**: When the advantage is negative, the objective will increase if the action becomes less likely, that is, if $\pi_\theta(a|s)$ decreases.

  The max in this term puts a *limit to how much the objective can decrease*, with a bottom at $(1-\epsilon)A^{\pi_{\theta_k}}(s,a)$.