Advanced Machine Learning
La Sapienza - 2019-2020

Giovanni Ficarra

July 2, 2020

Abstract

Summary of the material proposed by prof. Fabio Galasso in the course Adavanced Machine Learning, held at Sapienza University of Rome in A.A. 2019/2020. Original sources and authors are indicated in the slides provided by the prof.

These nodes are shared without any guarantee of complete correctness, since I may have done typos or misunderstood something. Feel free to drop an email at giovannificarra95@gmail.com to report errors, or contribute to my GitHub repo.

Contents

1 Introduction

What is machine learning:

- *Arthur Samuel (1959)*: "Machine Learning: Field of study that gives computers the ability to learn without being explicitly programmed";
- *Tom Mitchell (1998)*: "Well posed Learning Problem: A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E";
- *Examples*: database mining, applications which cannot be programmed by hand, self customizing programs, understanding human learning.

What is Perception (computer vision):

- *Science*: understand how do we see (explore computational model of human vision);
- *Engineering*: build systems that perceive the world;
- *Applications*: medical imaging, surveillance, entertainment, graphics, car industry.

The three keys to successes in **Deep Learning** are algorithms, data, computation.

Basic concepts of Computer Vision:

- Goals: recognize object, perceive depth...;
- *Recognition* problems:
  - *Identification*: recognize a specific object (es: your pen),
  - *Classification*: recognize a class of objects (es: any pen), it's also called *generic object recognition* or *object categorization*;
- *Segmentation*: separate pixels belonging to the foreground (object) and the background;
- *Localization/Detection*: position of the object in the scene, pose estimate;
- Challenges: multi scale, multi view, multi class, varying illumination, occlusion, cluttered background, articulation, high intraclass variance, low interclass variance.

2 Basics of digital image filtering

2.1 Linear Filtering

Image filtering is the process of applying some function to local image patches. Its goals are: reduce noise, fill-in missing values, extract image features.

Linear filtering is the simplest case of image filtering: replace each pixel by a linear combination of its neighbors.

2D discrete convolution is an example of linear filtering and is computed as follows:

$$f[m,n] = I \otimes g = \sum_{k,l} I[m-k, n-l] \cdot g[k,l], \tag{1}$$

that is, obtain the value of the pixel in position (m,n) of the filtered image f as the cross product of:

- a portion of the original image I given by the corresponding pixel and a number of neighbors given by the size of the kernel, i.e. $k \times l$, and
- the kernel g itself.

The operation, repeated for all the pixels of the image, corresponds to this process:

- mirror the filter across both dimensions (because of the minus sign),
- swipe it across the image,
- multiply and sum.

Analogously, if we have a 1D-filter and a 2D image, the convolution operation is:

$$f[m,n] = I \otimes g = \sum_{k} I[m-k, n] \cdot g[k]. \tag{2}$$

Examples:

- with a filter $[0,0,0,0,1,0,0,0,0]$ the image stays the same, since the considered pixel is copied and the neighbors are ignored;
- with a filter $[0,0,0,0,0,0,0,1,0]$ the image is shifted on the left of 3 pixels, since the filter is mirrored and so the considered pixel is copied in the position of its third neighbor on the left;
- with a filter $[0,0,0,1,1,1,0,0,0]$ the image is blurred, since the new pixel is obtained by the average of the considered pixel and its immediate neighbors.

Note that it's usually preferable that the elements of the filter vector sum up to 1, especially in a neural network where there are many convolutional filters.

Let's recall some basic properties of linear systems:

- homogeneity: $T[a \cdot X] = a \cdot T[X]$,
- additivity: $T[X_1 + X_2] = T[X_1] + T[X_2]$,
- superposition: $T[a \cdot X_1 + b \cdot X_2] = a \cdot T[X_1] + b \cdot T[X_2]$,
- linear systems $\iff$ superposition.

Thus, since those filters are linear, they can be combined to obtain new filters:

$$f[m,n] = I \otimes g_1 - I \otimes g_2 = I \otimes (g_1 - g_2)\,. \tag{3}$$

Example: with a filter $[0,0,0,0,2,0,0,0,0] - [0,0,0,1,1,1,0,0,0]$ the image is sharpened, since we enhance the considered filter before applying blur.

Furthermore, it is also true that

$$f[m,n] = (g_1 \otimes g_2) \otimes I = g_1 \otimes (g_2 \otimes I)\,, \tag{4}$$

that is, the filtered image f can be obtained by first applying one filter, and then another, or by applying directly their composition. If a 2D filter can be written as the product of two 1D filter (one in x-direction and one in y-direction), it's called **separable**.

2.2 Filtering to Reduce Noise

Filtering can be applied to reduce **noise**, such as light fluctuations, sensor noise, quantization effects, finite precision (low-level noise) or shadows, extraneous objects (complex noise). To do this, we assume that the neighborhood of a pixel contains information about its intensity.

We use the additive noise model: image I = signal S + noise N, and the intensity of the i-th pixel is $I_i = S_i + N_i$ with $E(N_i) = 0$, where S_i is deterministic and N_i, N_j are IID for $i \neq j$. Therefore, averaging noise reduces its effect and smoothing can give an inference about the original signal.

The **average filter** replaces each pixel with an average of its neighborhood. If all weights are equal, it is called a **box** filter (and it is separable).

A smarter average filter is the **Gaussian filter**:

- It's rotationally symmetric;
- It weights nearby pixels more than distant ones (probabilistic inference);
- The "amount" of smoothing depends on the parameter σ of the Gaussian;
- It is separable (see 4), thus we have the Gaussian in x-direction

  $$g(x) = \frac{1}{\sqrt{2\pi}\cdot\sigma}\cdot e^{-\frac{x^2}{2\sigma^2}}, \tag{5}$$

  the Gaussian in y-direction

  $$g(y) = \frac{1}{\sqrt{2\pi}\cdot\sigma}\cdot e^{-\frac{y^2}{2\sigma^2}}, \tag{6}$$

  and the Gaussian in both directions

  $$g(x,y) = \frac{1}{2\pi\sigma^2}\cdot e^{-\frac{x^2+y^2}{2\sigma^2}}; \tag{7}$$

- Let's prove the Gaussian's separability:

$$\begin{aligned}
h(i,j) &= f(i,j)\cdot g(i,j) \\
&= \sum_{k=1}^{m}\sum_{l=1}^{n} g(k,l)\cdot f(i-k,j-l) \\
&= \sum_{k=1}^{m}\sum_{l=1}^{n} e^{-\frac{k^2+l^2}{2\sigma^2}}\cdot f(i-k,j-l) \\
&= \sum_{k=1}^{m} e^{-\frac{k^2}{2\sigma^2}}\left[\underbrace{\sum_{l=1}^{n} e^{-\frac{l^2}{2\sigma^2}} f(i-k,j-l)}_{h'}\right] && \text{(apply 1-D Gaussian horizzontally)} \\
&= \sum_{k=1}^{m} e^{-\frac{k^2}{2\sigma^2}}\cdot h'(i-k,j-l) && \text{(apply 1-D Gaussian vertically)}
\end{aligned}$$

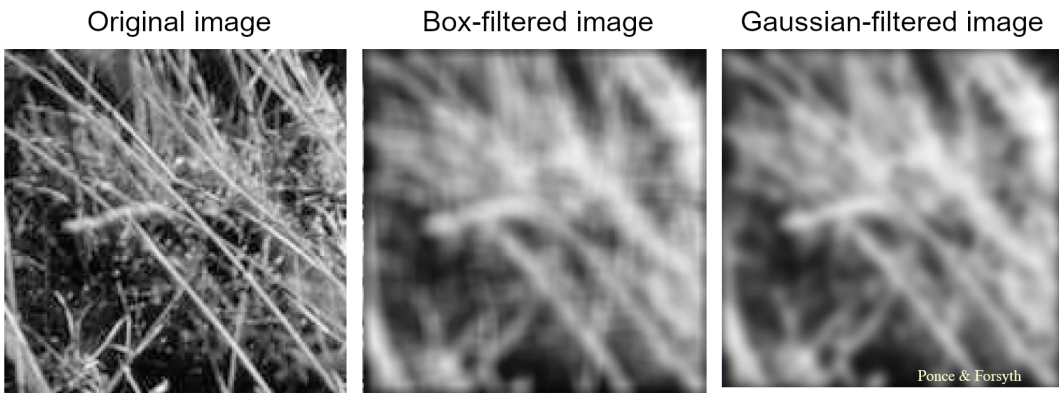

Figure 1: Gaussian and Box filters

2.3 Multi Scale Image Representation

To classify objects, a possibility is to compare the image with a set of objects' templates, used as kernels. Since it is very computationally expensive to execute a convolution with a big kernel, it is preferable to store small kernels, but in this manner you can't find bigger objects.

Thus, you can build a **Gaussian Pyramid**: repeatedly blur and down-sample the image, so that you obtain smaller and smaller images, until you find an object or the image reaches the size of the template/kernel. The blurring is quite fast since you can use a small linear filter, and the down-sampling is even faster since it is sufficient to remove a certain percentage of the pixels of the image.

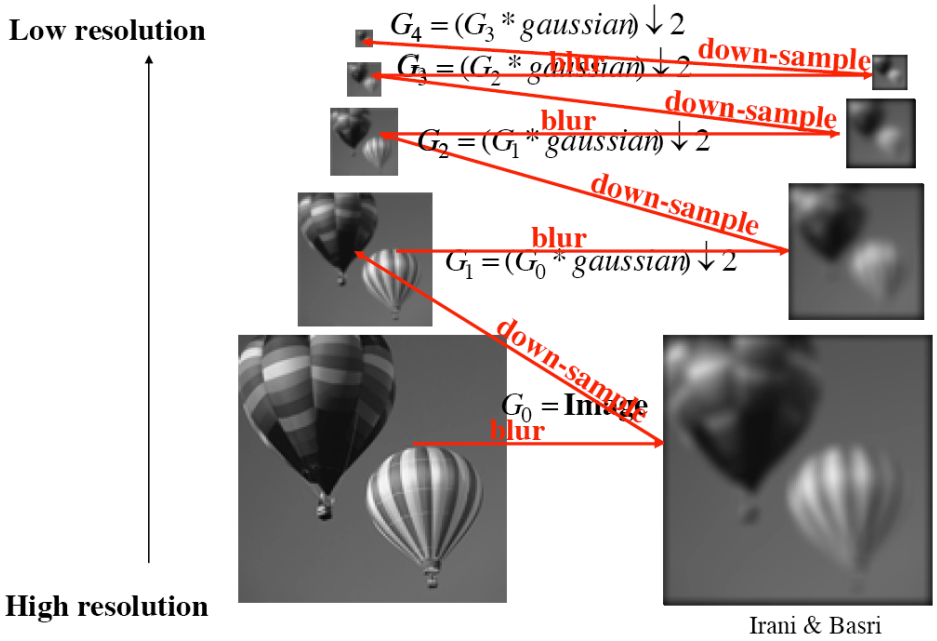

Figure 2: Gaussian Pyramid

In this way, you loose the details but you maintain average colors and shapes. That is, if you consider the image as made of light waves, you can apply the *Fourier Transform* to decompose those waves into a sum of sinusoidal curves (the original wave is is better approximated if you continue adding waves with higher frequency). Then, to blur and down-sample the image means to remove the waves with higher frequency.

2.4 Edge Detection

David Lowe's approach to recognition and localization:

1. Filter image to find brightness changes,
2. Fit lines to the raw measurements,
3. Project model into the image and match to lines (solving for 3D pose).

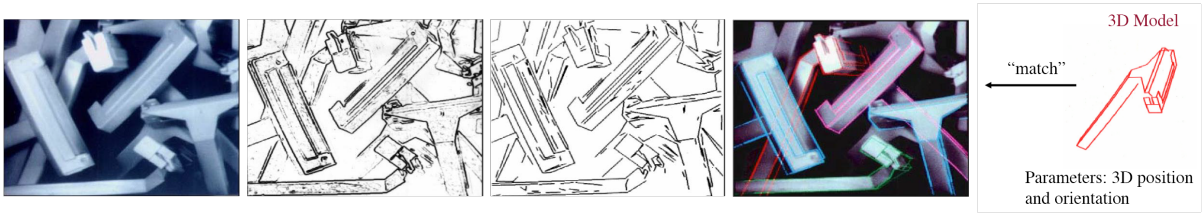

Figure 3: Lowe's method

The common approach in the 80s was to match models to edges and lines, so it was needed to reliably extract lines and edges.

Goals of edge detection:

- good detection: filter responds to edge, not to noise,
- good localization: detected edge near true edge,
- single response: one per edge.

First, we can note that it's useful to reduce noise in an image by applying a Gaussian blurring filter. Then, we can observe that edges correspond to fast changes, that is, where the magnitude of the first derivative is large, in other words, maximum and minimum points of the first derivative, or zero crossings of the second derivative (see figure 4).

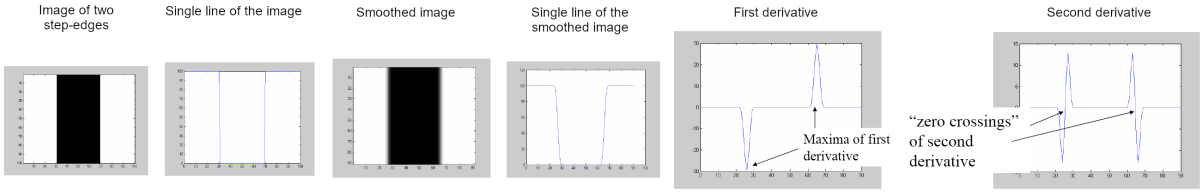

Figure 4: Edge detection example

The first derivative of $f(x)$ is

$$\frac{d}{dx}f(x) = \lim_{h\to 0}\frac{f(x+h)-f(x)}{h} \approx f(x+1)-f(x) \tag{8}$$

and can be implemented as a linear filter: direct $[-1,1]$ or symmetric $[-1,0,1]$.

Since derivative and convolution are linear operations, we can save one operation as follows:

$$\frac{d}{dx}(g\otimes f) = \left(\frac{d}{dx}g\right)\otimes f. \tag{9}$$

The second derivative of $f(x)$ is

$$\frac{d^2}{dx^2}f(x) = \lim_{h\to 0}\frac{\frac{d}{dx}f(x+h)-\frac{d}{dx}f(x)}{h} \approx \frac{d}{dx}f(x+1)-\frac{d}{dx}f(x) \approx f(x+2)-2f(x+1)+f(x) \tag{10}$$

and can be implemented as a linear filter $[1, -2, 1]$.

To implement these concepts with 2D images, we have to:

1. Compute the partial derivatives along each row and column:

$$\frac{d}{dx}I(x,y) = I_x \approx I \otimes D_x \qquad \text{(in } x \text{ direction)}$$
$$\frac{d}{dy}I(x,y) = I_y \approx I \otimes D_y \qquad \text{(in } y \text{ direction)}$$

   where D_x and D_y are often approximated with simple filters (finite differences):

$$D_x = \frac{1}{3}\begin{array}{|c|c|c|}\hline -1 & 0 & 1 \\ \hline -1 & 0 & 1 \\ \hline -1 & 0 & 1 \\ \hline \end{array},\ D_y = \frac{1}{3}\begin{array}{|c|c|c|}\hline -1 & -1 & -1 \\ \hline 0 & 0 & 0 \\ \hline 1 & 1 & 1 \\ \hline \end{array};$$

2. Observe that by using the derivative in x direction we can only find vertical edges, while by using the derivative in y direction we can only find horizontal edges (as shown in figure 5), thus we need to combine the two derivatives;

Figure 5: Edge detection with Gaussian filters

3. The *gradient* ∇f of a function f is the vector whose components are the partial derivatives of f:

$$\nabla I = (I_x, I_y) = \left(\frac{\partial I}{\partial x}, \frac{\partial I}{\partial y}\right), \tag{11}$$

   and it has the direction $\theta = \arctan(I_y, I_x)$ which is perpendicular to the edge and the magnitude (or norm) $\|\nabla I\| = \sqrt{I_x^2 + I_y^2}$ which is proportional to the strength of the edge;

4. Find sufficiently large local maxima in the first derivative, thus we can use two thresholds (*hysteresis*): a high threshold to start edge curve (maximum value of gradient should be sufficiently large) and a low threshold to continue them (in order to bridge gaps with lower magnitude).

Observation 2.1. The scale of the smoothing filter affects derivative estimates, and also the semantics of the edges recovered: strong edges persist across scales, but smallest edges are lost with heavier blurring.

There are three major issues:

- The gradient magnitude at different scales is different; which to choose?
- The gradient magnitude is large along a thick trail; how to identify the significant points?
- How to link the relevant points up into curves?

Observation 2.2. Based on the assumptions of linear filtering and additive IID Gaussian noise (see 2.2) and on the goal we fixed for edge detection in 2.4, the optimal detector is approximately derivative of Gaussian.

Observation 2.3 (Detection/localization tradeoff)**.** More smoothing improves detection and hurts localization.

The **Canny edge detector** uses thinning (or *non-maximum suppression*) to solve the issues we stated above and thus can be considered "optimal":

- Check if pixel is local maximum along gradient direction;
- Choose the largest gradient magnitude along the gradient direction;
- Requires checking interpolated pixels.

Another possibility is to use the **Laplacian filter**:

$$\nabla^2 f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}, \tag{12}$$

that is another linear filter:

$$\nabla^2 (G \otimes f) = \nabla^2 G \otimes f. \tag{13}$$

Thus, we can apply this filter to a smoothed image to find 1D edges:

$$\frac{d^2}{dx^2}(g \otimes f) = \left(\frac{d^2}{dx^2} g\right) \otimes f. \tag{14}$$

Furthermore, the Laplacian can be approximated with a difference of Gaussians at different scales, which allows us to build a **Laplacian Pyramid** starting from a Gaussian Pyramid (see 2.3), where the i-th element of the Laplacian Pyramid is $L_i = G_i - \text{expand}(G_{i+1})$ and the i-th element of the Gaussian Pyramid is $G_i = L_i + \text{expand}(G_{i+1})$ (see figure 6).

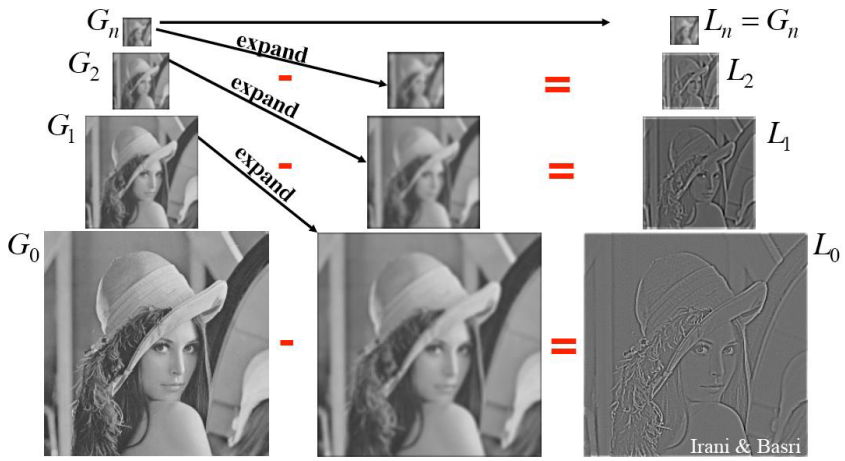

Figure 6: Laplacian Pyramid

2.5 Hough Transform

Edges can result from: object-background boundaries, object-object boundaries, shadows, discontinuities of object texture, discontinuities of surface normals.

We are interested in finding objects, thus we should discriminate among those types of edges. An approach is to assemble detected edges to extract object contours, even if they not necessarily correspond to object boundaries. **Hough Transformation** uses the knowledge that many contours belong to straight lines.

A line is represented as $y = ax + b$:

- the parameters a and b determine all the points of a line, which corresponds to a transformation: $(a, b) \longrightarrow (x, y)$, that is $y = ax + b$,
- inverse interpretation: transformation of $(x, y) \longrightarrow (a, b)$, that is $b = (-x)a + y$.

We can use this to conclude that the points for which the magnitude of the first derivative is large lie potentially on a line.

Idea: for a particular point (x, y), determine all lines which go through this point, with parameters given by $b = (-x)a + y$; i.e., those lines are given by a line in the parameter space (a, b).

Implementation:

- the parameter space (a, b) has to be discretized,
- for each candidate (x, y) for a line, store the line $b = (-a)x + y$,
- in principle, each candidate (x, y) votes for the discretized parameters,
- the maxima in the parameter space (a, b) correspond to lines in the image.

But this parameterization has two issues: the parameter a can become infinite for vertical lines, the discretization is problematic. Thus, it is used another parameterization with limited domain: $x\cos(\theta) + y\sin(\theta) = \rho$, where ρ is limited by the size of the image and $\theta \in [0, 2\pi]$.

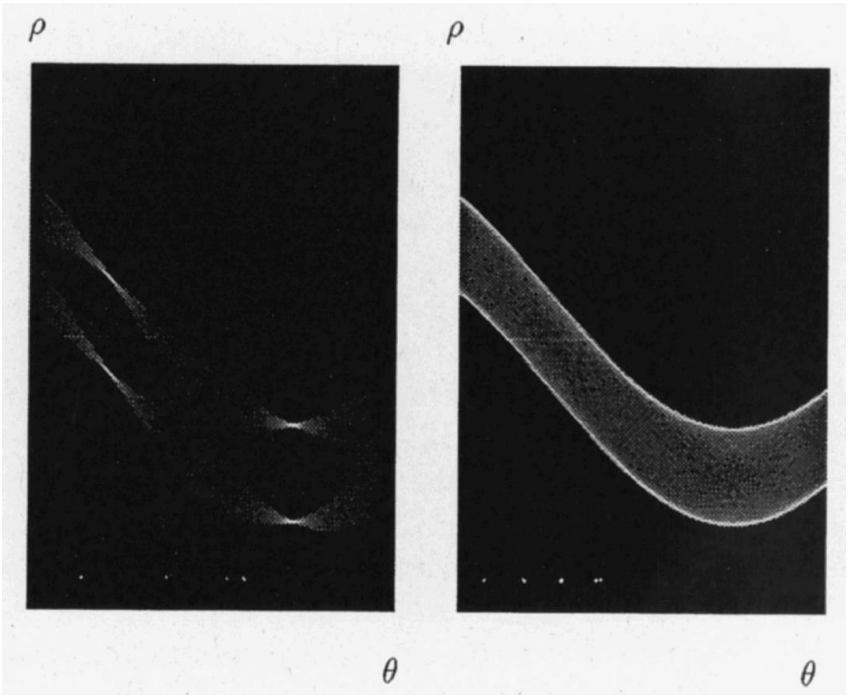

Figure 7: Hough transform for a square (left) and a circle (right).

The same idea can be used for other parameterized contours, such as the circle $(x - a)^2 + (y - b)^2 = r^2$, with three parameters: center point (a, b) and radius r; but there are some limitations: the parameter space should not become too large and not all contours can be parameterized.

Generalization for an arbitrary contour:

- choose reference point for the contour,
- for each point on the contour, remember where it is located with respect to the reference point,
- recognition: whenever you find a contour point, calculate the tangent angle and *vote* for all possible reference points.

3 Object Instance Identification using Color Histograms

The main challenges of object identification are the different modes of variations:

- Viewpoint changes (translation, image plane rotation, scale changes, out of plane rotation);
- Illumination;
- Clutter;
- Occlusion;
- Noise;

Until a paradigm shift in the 90s, the most used approach were **appearance-based identification**, which basic assumptions were:

- Objects can be represented by a collection of images, called *appearances*;
- For recognition, it is sufficient to compare the 2D appearances (no 3D model is needed).

Global representation:

- Represent each view of an object by a global descriptor;
- For recognizing objects, just match the global descriptors.

With this approach, we can take care of the modes of variations in three different ways:

- built into the descriptor (the descriptor can be invariant to image-plan rotations and translations),
- incorporated in the training data or the recognition process (the training set can be built with different viewpoints, scales, out-of-plane rotations),
- robustness of descriptor or recognition process (the descriptor matching strategy can be robust to illumination, noise, clutter, partial occlusion).

Such an identifier might be the **color histogram**:

- Color stays constant under geometric transformation and is robust to partial occlusion (since it is a *local feature*, i.e., it's defined for each pixel);
- We can use color statistics: given (R, G, B) for each pixel, compute a 1D histogram for each the three values and one for the luminance, or a single 3D histogram with $H(R, G, B) = \#\text{pixel with color}(R, G, B)$ (more meaningful and with a notion of closeness);

Observation 3.1. One component of the 3D color space is **intensity**, given by $I = R + G + B$, thus, if a color vector is multiplied by a scalar, the intensity changes but not the color itself, this means colors can be normalized by the intensity, to obtain their *chromatic representation*:

$$r = \frac{R}{R+G+B} \quad (15) \qquad g = \frac{G}{R+G+B} \quad (16) \qquad b = \frac{B}{R+G+B} \quad (17)$$

Observation 3.2. Since $r + g + b = 1$, only two parameters are necessary.

Recognition using histograms (*nearest-neighbor* strategy):

1. Build a database with multiple training views per known objects;
2. Build a set of histograms $H = M_1, M_2, M_3, \ldots$ for each view image in the dataset;
3. Build a histogram T for the test image (of an unknown object);
4. Compare T to each $M_k \in H$;
5. Select the object with the best matching score, or reject the test image if no object is similar enough (distance above a threshold t).

3.1 Comparison measures

- **Intersection** measures the common part of both histograms (more robust): $\cap(Q,V) = \sum_i \min(q_i, v_i)$,
- Intersection for unnormalized histograms: $\cap(Q,V) = \frac{1}{2}\left(\frac{\sum_i \min(q_i,v_i)}{\sum_i q_i} + \frac{\sum_i \min(q_i,v_i)}{\sum_i v_i}\right)$,
- **Euclidean distance** focuses on the differences between the histograms, all cells are weighted equally (not very discriminant): $d(Q,V) = \sum_i (q_i - v_i)^2$,
- **Chi-square** tests if two distributions are different, it is more significant since cells are not weighted equally, but may have problems with outliers: $\chi^2(Q,V) = \sum_i \frac{(q_i - v_i)^2}{q_i + v_i}$,
- Statistical tests,
- Information theoretic measures.

3.2 Pro and cons

In the first paper in 1991 this approach worked surprisingly well, with 66 objects recognized almost without errors.

Advantages:

- invariant to object translations and rotations,
- slowly changing for out-of-plane rotations or with partial occlusion,
- no perfect segmentation necessary,
- possible to recognize deformable objects.

Disadvantages:

- the pixel colors change with the illumination (*color constancy problem*): intensity and spectral composition,
- not all objects can be identified by their color distribution.

4 Performance evaluation

To establish which method is better we can:

- compare a single number,
- compare curves.

The recognition algorithm identifies (*classifies*) the gallery image as matching the query object (test image) if their distance is below a threshold t. We can compare actual outcomes to predicted outcomes using a **confusion matrix** (*classification matrix*):

	Predicted = 0	Predicted = 1
Actual = 0	True Positives (TN)	False Positives (FP)
Not Retrieved	False Negatives (FN)	True Negatives (TN)

Table 1: Confusion matrix

The higher the t, the more gallery images are classified as matching (more TP but also more FP); the smaller the t the less gallery images are classified as matching (more TN but also more FN).

4.1 Single number evaluation metrics

N = number of observations

- **Overall accuracy**: How often the correct outcome is being predicted. How well a binary classification test correctly identifies or excludes a condition.

$$ACC = \frac{TP + TN}{N} \tag{18}$$

- **Overall error rate**:

$$PPV = \frac{FP + FN}{N} \tag{19}$$

- **Precision** (aka **Positive Predictive Value**): The fraction of relevant instances among the retrieved instances. (see figure 8)

$$PPV = \frac{TP}{TP + FP} \tag{20}$$

- **Recall** (aka **Sensitivity** or **True Positive Rate**): The fraction of relevant instances that have been retrieved over the total amount of relevant instances. The proportion of actual positives that are correctly identified as such. (see figure 8 and figure 9)

$$TPR = \frac{TP}{TP + FN} \tag{21}$$

- **Specificity** (aka **True Negative Rate**): The proportion of actual negatives that are correctly identified as such. (see figure 9)

$$TNR = \frac{TN}{FP + TN} \tag{22}$$

- **Negative Predictive Value**: The proportions of negative results that are true negative results.

$$NPV = \frac{TN}{TN + FN} \tag{23}$$

- **Fall-out** (**False Positive Rate**):

$$FPR = 1 - TNR = \frac{FP}{FP + TN} \tag{24}$$

- **Miss rate** (**False Negative Rate**):

$$FNR = 1 - TPR \tag{25}$$

- **False Discovery Rate**:

$$FDR = 1 - PPV \tag{26}$$

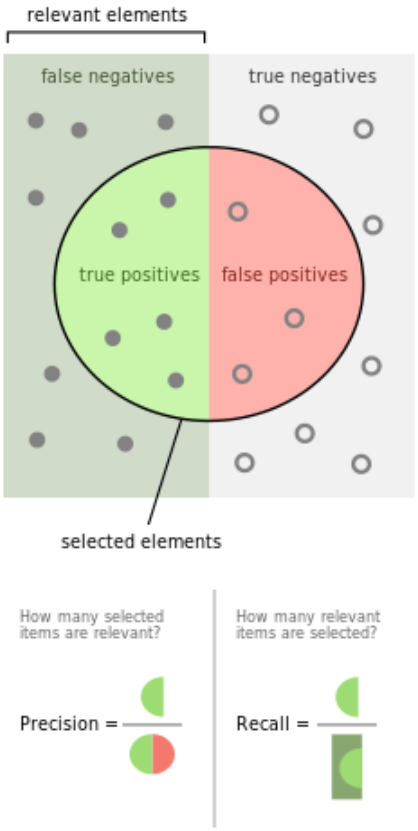

Figure 8: Precision-Recall

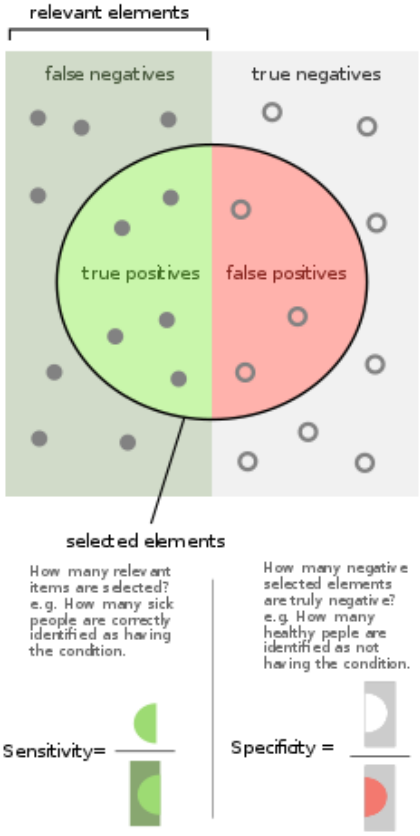

Figure 9: Sensitivity-Specificity

4.2 Curve-based metrics

We can use curve-based metrics to find the best trade-off for the threshold value, balancing the cost of failing to identify an object and the cost of raising false alarms, depending on the task :

- **Receiver Operating Curve**: It's a probability curve, the plot of the TPR against the FPR for a binary classification problem as you change the threshold. (see figure 10)
- **Area under the Receiver Operating Curve (AUC)**: It represents degree or measure of separability, it tells us how much the model is capable of distinguishing between classes, i.e., the closer the AUC is to 1, better the model is at predicting 0s as 0s and 1s as 1s, the closer it is to 0.5, the more the model is similar to a random guess. AUC is less affected by sample balance than accuracy. (see figure 10)
- **Precision-recall curve**: Preferred for detection, where TN's are undefined. (see figure 11)

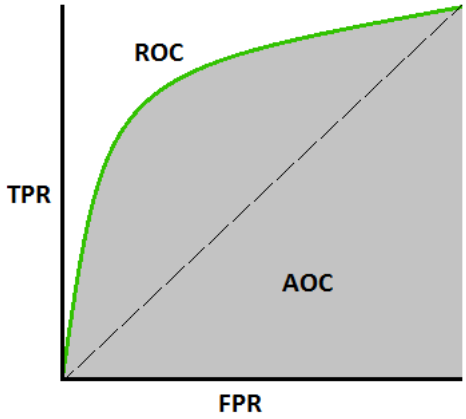

Figure 10: ROC-AUC curves

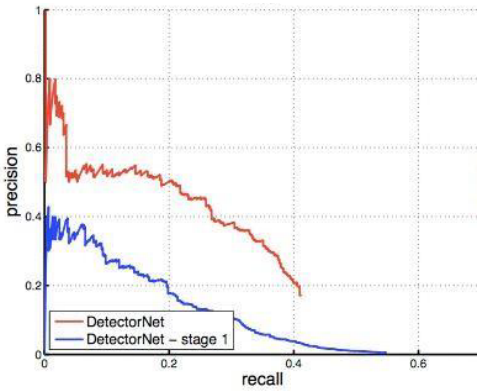

Figure 11: Precision recall curve

5 Image Classification

Image classification is a core task in Computer Vision, based on the problem that a computer "sees" an image just like a grid of numbers (*semantic gap*).
Furthermore, there are many challenges to face in image classification, not dissimilar to those we discussed for object identification in section 3:

- viewpoint variation,
- background clutter,
- illumination
- deformation
- occlusion
- intraclass variation

5.1 Data driven approaches (machine learning)

There is no obvious way to hard code an algorithm for recognizing images, even if attempts based on edge detection and corners have been made.
Thus, the preferred approach is based on Machine learning:

1. Collect a dataset of images and labels,
2. Use Machine Learning to train a classifier,
3. Evaluate the classifier on new images.

5.2 K Nearest Neighbors classifier

A first classifier we can consider is **Nearest Neighbor**. The approach is the following:

1. Memorize all data and labels;
2. For each test image:
   2.1. find closest (most similar) train image;
   2.2. predict label of the nearest image.

With a basic implementation, the training requires $O(1)$ steps, while the prediction $O(N)$, but we want classifiers that are fast at prediction. By the way, many methods exist for fast/approximate nearest neighbor.

To compare images, the **L1/Manhattan distance metric** is used:

$$d_1(I_1, I_2) = \sum_p |I_1^p - I_2^p| \tag{27}$$

that is, the sum of the pixel-wise absolute value differences between two images.

The idea of **KNN** is to take the majority vote from K closest points, instead of copying the label of the nearest one (see figure 12 for an example).

The distance metrics used in KNN are Manhattan like in Nearest Neighbor (see equation 27) or **L2/Euclidean distance**:

$$d_2(I_1, I_2) = \sqrt{\sum_p \left(I_1^p - I_2^p\right)^2} \tag{28}$$

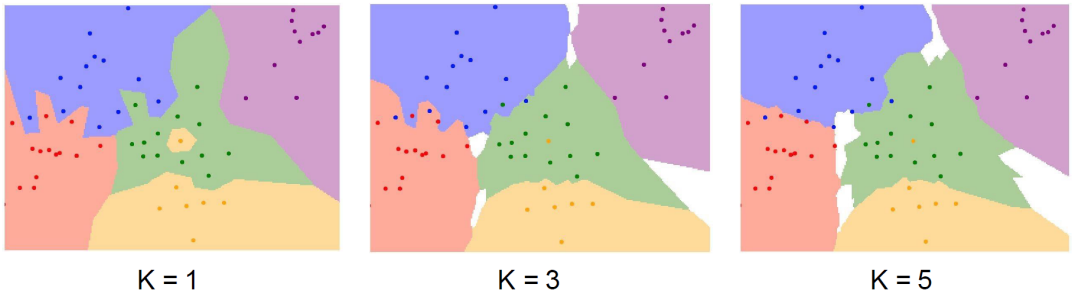

Figure 12: KNN example

The **hyperparameters** of a classifier are choices about the algorithm that we set rather than learn, such as the value of K or the distance to use. Since they are problem-dependent, we must try and see what works best.
A possibility to find the best hyperparameters is to split the data into *train*, *validation* and *test*, so that we can use the validation set to fine tune the classifier trained on the training set, and then test it on the images of the test set, that it has never seen before.
Another possibility is **k-fold cross validation**: part of the dataset is used as test set, the remaining is divided into k folds, each of which is used as validation set in turn, then the results are averaged (note that this k has nothing to do with the hyperparameter K of KNN). This approach is particularly useful for small dataset.

That said, in practice KNN is never used on images: it is too slow at test time, distance metrics on pixels are not informative, curse of dimensionality (details on this issue on Wikipedia).

5.3 Linear classification (parametric approach)

First of all, let's say that Linear classifiers are the building blocks of the Neural Networks.

Given an image represented as a 1D vector x (a flattened matrix of n elements) and m classes, a linear classifier is a function

$$f(x, W) = Wx + b \tag{29}$$

where W is a matrix of parameters (or *weights*) with m rows and n columns, and b is called *bias*.
We can give different interpretation of the linear classifier:

- Algebraic viewpoint: we compute a score for each class, based on matrix multiplication;
- Geometric viewpoint: we compute hyperplanes cutting up a space of images, to split it according to classes;
- Visual viewpoint: we apply to x a template for each class (a row of W can be seen as a template image with the same size of x).

5.4 Loss functions and regularization

At this point, we need a *loss function* that quantifies our unhappiness with the scores across the training data, that is, that tells how good our current classifier is.

Given a dataset of examples $\{(x_i, y_i)\} = N_{i=1}$, where x_i are images and y_i are integer labels, the **loss over the dataset** is the average of the losses over the examples:

$$L = \frac{1}{N} \sum_i L_i \left(f\left(x_i, W\right), y_i \right) \tag{30}$$

Given an example (x_i, y_i), and a scores vector $s = f(x_i, W)$, the **multiclass SVM loss** (or *Hinge loss*) is:

$$\begin{aligned} L_i &= \sum_{j \neq y_i} \begin{cases} 0 & \text{if } s_{y_i} \geq s_j + 1 \\ s_j - s_{y_i} + 1 & \text{otherwise} \end{cases} \\ &= \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1) \end{aligned} \tag{31}$$

that is, it gives a penalty to each score that is bigger than the score of the correct label + 1.

At this point it is worth noting that this loss isn't unique for a certain W, so we need a way to choose the "best" W among those with the same loss, thus we introduce **regularization**:

$$L(W) = \underbrace{\frac{1}{N} \sum_{i=1}^{N} L_i\left(f\left(x_i, W\right), y_i\right)}_{\text{data loss}} + \underbrace{\lambda R(W)}_{\text{regularization}} \tag{32}$$

where λ is a hyperparameter that determines the regularization's strength.
The data loss forces the model predictions to match training data, while the regularization prevents the model from doing too well on training data (i.e., avoids overfitting), expresses preferences over weights, makes the model simple so it works on test data, improves optimization by adding curvature.

Examples of regularization are:

- L2 regularization
$$R(W) = \sum_k \sum_l W_{k,l}^2; \tag{33}$$
- L1 regularization:
$$R(W) = \sum_k \sum_l |W_{k,l}|; \tag{34}$$
- Elastic net (L1 + L2):
$$R(W) = \sum_k \sum_l \left(\beta W_{k,l}^2 + |W_{k,l}|\right); \tag{35}$$
- Dropout;
- Batch normalization;
- Stochastic depth;
- Fractional pooling.

5.5 Softmax classifier (Multinomial Logistic regression)

The aim is to interpret raw classifier scores as probabilities.

Given a scores vector s, the Softmax function is

$$P(y|x_i) = P(Y = k|X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}, \tag{36}$$

that is, we start with un-normalized log-probabilities (*logits*), we compute the *exp* to obtain non negative values, and finally we normalize, so that the sum of the values is 1.

In this case, weights are updated with **maximum likelihood estimation**: choose weights to maximize the likelihood of the observed data. This is done by comparing the computed probabilities with a one-hot vector that gives probability 1 to the true label and 0 to the others. For the comparison three methods can be used:

1. Entropy:
$$H(P^t) = -\sum_y P^t(y) \log P^t(y) \tag{37}$$

2. Kullback–Leibler divergence:
$$D_{KL}\left(P^t||Q\right) = \sum_y P^t(y) \log \frac{P^t(y)}{Q(y)} \tag{38}$$

3. Cross Entropy (the sum of 1 and 2):
$$\begin{aligned} H(P^t, Q) &= H(P^t) + D_{KL}(P^t||Q) \\ &= \sum_y P^t(y) \left(\log \frac{P^t(y)}{Q(y)} - \log P^t(y) \right) \\ &= -\sum_y P^t(y) \log Q(y) \end{aligned} \tag{39}$$

Since the target distribution is
$$P^t(y) = \begin{cases} 1 & y = y_i \\ 0 & y \neq y_i \end{cases}$$
and the output of the network is $Q(y|x_i)$ (as given by softmax function 36), the *Cross Entropy Loss* for an image x_i, used to maximize the probability of the correct class, is:
$$\begin{aligned} L_i = L(x_i) &= -\sum_y P^t(y) \cdot \log Q(y|x_i) \\ &= -1 \cdot \log Q(y|x_i) \\ &= -\log P(Y = y_i|X = x_i) \\ &= -\log \frac{e^{s_{y_i}}}{\sum_j e^{s_j}} \end{aligned} \tag{40}$$

Observation 5.1. $L_i \in [0, \infty) \ \forall \ i$.

Observation 5.2. Since at initialization all s are approximately equal, the loss is $\log(C)$.

Observation 5.3. Softmax loss reflects more slight changes in the input, thus it is more useful to choose among different values of W.

5.6 Optimization via gradient descent

The most common strategy used to optimize weights is *gradient descent*, often represented as a walk from the top of a mountain towards the valley, following the slope (see figure 13).

As we already mentioned in section 2.4, in one dimension the derivative of a function $f(W)$ is
$$\frac{f(W)}{dW} = \lim_{h \to 0} \frac{f(W + h) - f(W)}{h}, \tag{41}$$
while in multiple dimensions the gradient is the vector of partial derivatives along each dimension.
The slope in any direction is the dot product of the direction with the gradient, thus the direction of the steepest descent is the negative gradient (see figure 14).

Figure 13: Follow the slope

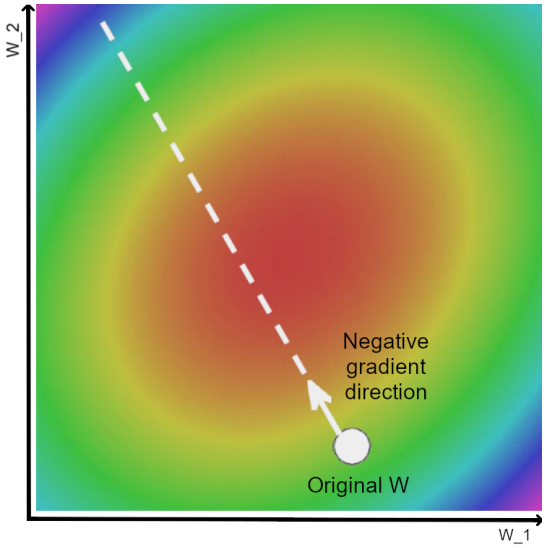

Figure 14: Gradient descent

To compute the **numeric gradient** of the loss $\nabla_W L$ requires to loop over all the dimensions of W and apply the formula 41 using a fixed small h, so it is too slow in practice, and approximate.
Since the loss is in fact just a function of W, we can use calculus to compute the **analytic gradient**.
In practice, analytic gradient is used for its speed, but its implementation is checked with numerical gradient, since it is more error-prone, although more accurate (*gradient check*).

Let's define the loss as in equation 30: $L(W) = \frac{1}{N}\sum_i^n L_i(W)$, the computation of the full sum is too expensive when N is large, thus it can be approximated in different ways:

- **Stochastic Gradient Descent**: randomly choose one training sample x_i and update weights based on loss $L_i(W)$;
- **Mini batch training**: process a subset of training samples $M \subset \{1, \dots, n\}$ and update weights based on $L_M(W) = \frac{1}{|M|}\sum_{i \in M} L_i(W)$;
- **Batch training** (no approximation): process all training samples update weights based on $L(W)$ (as defined above).

5.7 Image Features

To compute class scores for a set of images, it is necessary to adopt an opportune representation of those images. Feature transform can allow to separate points with a linear classifier, even if it wouldn't be possible with the original representation.

Examples of image features are:

- *Color histogram* (see section 3);
- *Histogram of oriented gradients* (divide the image in small regions and quantize edge direction into 9 bins for each region);
- *Bag of words*: extract random patches from the images and cluster them to build the *codebook* of visual words, then encode the images based on the occurrences of those patches in them.

With Convolutional Neural Networks, the feature extraction phase can be performed within the network itself.

5.8 Neural Networks and Backpropagation

As we introduced in section 5.6 there are many problems with manual computation of $\nabla_W L$:

- It requires lots of matrix calculus;
- If we change loss we have to re-derive from scratch;
- Not feasible for very complex models.

Thus we use computational graphs and backpropagation. A **computational graph** is a graph whose edges connect input with functions and functions to outputs. By walking forward on the graph we can compute the loss, and going back we can use **backpropagation** to compute the gradients. This approach is based on the recursive application of the *Chain rule*, which allows to compute the gradient of a node based on the next two:

$$\underbrace{\frac{\partial f}{\partial x}}_{\substack{\text{downstream}\\ \text{gradient}}} = \underbrace{\frac{\partial f}{\partial q}}_{\substack{\text{upstream}\\ \text{gradient}}} \cdot \underbrace{\frac{\partial q}{\partial x}}_{\substack{\text{local}\\ \text{gradient}}} \tag{42}$$

where q is the value of the current node, x is the value of the previous node (the one immediately on the left, with the input of the current node), f is the value of the next node (the one immediately on the right, with the output of the current node).

Observation 5.4. If a node takes more than one input, the gradient propagates on both branches using the Chain rule.

Observation 5.5. Computational graph representation may not be unique, thus it is important to choose one where local gradients at each node can be easily expressed.

Patterns in gradient flow (see figure 15):

- Add gate: gradient distributor,
- Multiplication gate: swap multiplier,
- Copy gate: gradient adder,
- Max gate: gradient router.

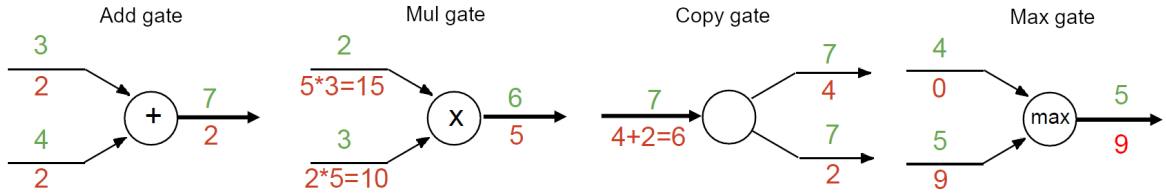

Figure 15: Gradient flow patterns

Vector derivatives:

- Scalar to scalar: regular derivative $\frac{\partial y}{\partial x} \in \mathbb{R}$;
- Vector to scalar: derivative is gradient $\frac{\partial y}{\partial x} \in \mathbb{R}^N$, $\left(\frac{\partial y}{\partial x}\right)_n = \frac{\partial y}{\partial x_n}$;
- Vector to vector: derivative is the *Jacobian matrix* $\frac{\partial y}{\partial x} \in \mathbb{R}^{N \times M}$, $\left(\frac{\partial y}{\partial x}\right)_{n,m} = \frac{\partial y_m}{\partial x_n}$.

Observation 5.6. Observations about vector-vector derivatives:

- $\frac{\partial L}{\partial x}$ always has the same shape as x.
- Since Jacobian is sparse, in fact off-diagonal entries are always zero, it isn't formed explicitly, but only multiplied implicitly.
- With an explicit representation Jacobians could take hundreds of GBs of memory.

- Let's recall that $y = Wx$, where x is a matrix of size $N \times D$, W is a matrix of size $D \times M$ and y is a matriz of size $N \times M$; then, an element $x_{n,d} \in x$ affects the whole row y_n of y, and $x_{n,d}$ affects $y_{n,m}$ by a factor $w_{d,m}$.
- Similarly to the previous point, for gradients we have:

$$\underbrace{\frac{\partial L}{\partial x}}_{N \times D} = \underbrace{\left(\frac{\partial L}{\partial y}\right)}_{N \times M} \cdot \underbrace{W^T}_{M \times D} \tag{43}$$

$$\underbrace{\frac{\partial L}{\partial W}}_{D \times M} = \underbrace{x^T}_{D \times N} \cdot \underbrace{\left(\frac{\partial L}{\partial y}\right)}_{N \times M} \tag{44}$$

6 Introduction to Deep Learning and PyTorch

6.1 Neural Networks and Deep Learning

Fully connected Neural Networks are stacks of linear functions and nonlinear activation functions; they have much more representational power than linear classifiers. This kind of neural network is also called *multi-layer perceptrons*.

While in linear classifier (such as the SVM) we had the linear score function $f = Wx$ (see 29), with a 2-layers NN we use the function

$$f = W_2 \max(0, W_1 x) \tag{45}$$

with $x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H \times D}, W_2 \in \mathbb{R}^{C \times H}$, similarly, with a 3-layers NN we use

$$f = W_3 \max(0, W_2 \max(0, W_1 x)) \tag{46}$$

with $x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H \times D}, W_2 \in \mathbb{R}^{C \times H}$ (see figure 16).
Note that in practice we usually add a learnable *bias* at each layer as well.

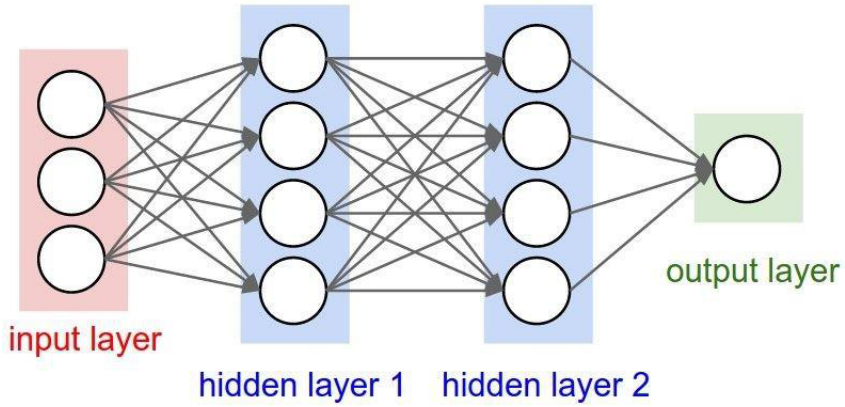

Figure 16: 3-layer Neural Network

The non linear function $\max(0, z)$ is an **activation function**. Note that without using it we end up with a new linear classifies, that isn't useful at all.

Some (non linear) activation functions that may be used are:

- ReLU: $\max(0, x)$,
- Sigmoid: $\sigma(x) = \frac{1}{1+e^{-x}}$,
- Tanh: $\tanh(x)$,
- Leaky ReLu: $\max(0.1x, x)$,
- Maxout: $\max(W_1^T x + b_1, W_2^T x + b_2)$,
- ELU: $\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$.

It's important to note that increasing the number of layers and their size (the number of neurons per layer) leads to an improvement to the capacity of the NN, but increases overfitting too: the network adapts too well to the input data, so it isn't able to generalize (see figure 17). Thus, we use regularization to solve this problem (see figure 18).

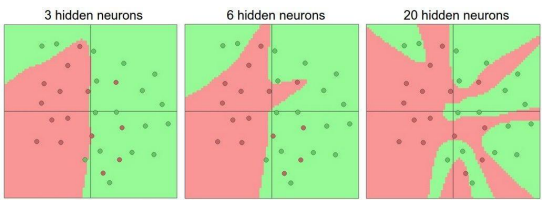

Figure 17: The effect of increasing the number of neurons

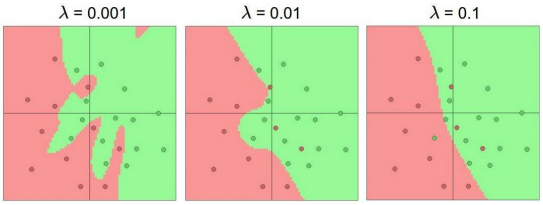

Figure 18: The effect of increasing the regularization's strength

While it is possible to compare the artificial neurons with the biological ones, since they receive electrical stimuli from other synapses through dendrites, and propagates the impulse according to a sort of activation function of the received stimuli, like the one computed in NNs, there are some important differences between them:

- Biological neurons have complex connectivity patterns, while artificial ones are organized into regular layers for computational efficiency (even if NN with random connections can work too);
- There exists many different types of biological neurons;
- Dendrites can perform complex non linear computations;
- Synapses are not a single weight but a complex non linear dynamical system;
- Rate code may not be adequate.

Since, as we saw, the number of layers of the network (*depth*) increases its capacity, we are particularly interest in **Deep Neural Networks**, that is, those networks with many hidden layers.

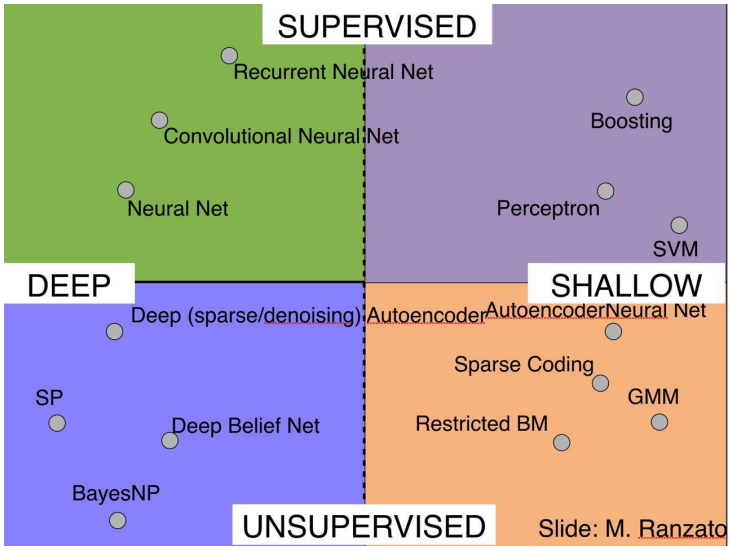

Figure 19: Overview of Neural Network Architectures

Deep Learning is giving so good result and performances thanks to different factors:

- Availability of large datasets;
- Massive parallel compute power;
- Advances in machine learning over the years;
- Internet (availability of large scale data);
- GPUs (availability of parallel compute power);
- Deep / hierarchical models with end to end learning.

Thanks to the progress of deep networks, the process of feature extraction we sketched in section 5.7 has changed dramatically. The traditional approach is:

- features are hand crafted and fixed,
- extracted features might be too general or too specific,
- to achieve best classification performance more complex classifier are built.

The research on hand-crafted features lead to considerable improvements over the years, and many hand designed features are currently in use.

With deep learning, though, we can perform parameterized feature extraction, and we can obtain features that are efficient to compute and to train (differentiable), by joining the training of feature extraction and of classification into one pipeline, a *unique end-to-end system*. In this way, all parts are adaptive, and we can apply a nonlinear transformation directly from input to desired output. Furthermore, we can built complex networks just by composition of simple building blocks: each block has its trainable parameters and produces an intermediate representation of the input image, more abstract from layer to layer. Like in MLP, during the forward pass the output is generated, than the error is propagated backwards to update weights in the backpropagation step.

6.2 Deep Learning Hardware

CPUs and GPUs are the most important hardware for Deep Learning. Let's see the differences between them:

- **CPU**:
  - Fewer cores, but each core is much faster and much more capable,
  - Great at sequential tasks;
- **GPU**:
  - More cores, but each core is much slower and "dumber",
  - Great for parallel tasks;
- **TPU**: New hardware specialized for deep learning, by Google.

Note that NVIDIA GPUs are more supported than AMD ones in machine learning frameworks such as TensorFlow or PyTorch, thanks to the *CUDA Deep Neural Network* library.

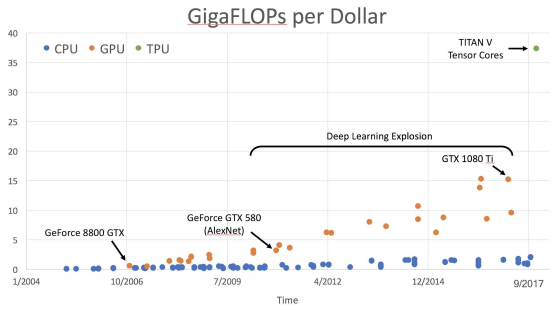

Figure 20: The evolution of CPUs adn GPUs

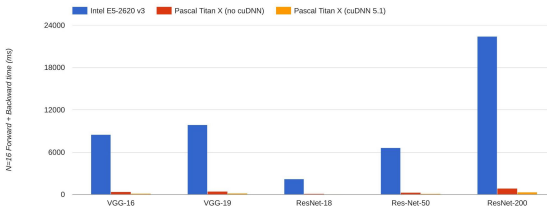

Figure 21: CPU vs GPU in practice (note that CPU performance are not well optimized, so the plot might be a little unfair)

Since data typically resides on disk, training can bottleneck on reading data and transferring to GPU. Possible solutions are:

- Read all data into RAM;
- Use SSD instead of HDD;
- Use multiple CPU threads to prefetch data.

6.3 PyTorch

The main advantages of deep learning frameworks are:

- Quick to develop and test new ideas;
- Automatically compute gradients;
- Run it all efficiently on GPU (wrap cuDNN, etc).

PyThorch, in particular, allows to build **dynamic computation graphs**:

- Operations on *Tensors* with `requires_grad=True` cause PyTorch to build a computational graph and to perform the actual computations;
- `loss.backward()` causes search for path between loss and weights (for backpropagation) and perform computation;
- On every iteration the graph and the backpropagation path are thrown away, and are rebuild from scratch;
- Its's inefficient if we are building the same graph over and over again, but allows to generate different graphs on each iteration under different conditions;
- Particularly useful for *recurrent networks*, *recursive networks*, *modular networks*.

To overcome the inefficiency of dynamic graphs, if there is no need to change the graph at runtime, one can use **static computation graphs**:

- Build computational graph describing the computation and finding paths for backpropagation;
- Reuse the same graph on every iteration;
- The framework can optimize the graph before it runs;
- Once graph is built, it can be serialized, exported to ONXX, and executed without the code that built it.

7 Convolutional Neural Networks

Convolutional Neural Networks, and deep CNNs in particular, proved to be very effective in image classification tasks, as shown in the *ImageNet Large Scale Visual Recognition Challenge* (see figure 22). By the way, this model is used also for semantic segmentation, detection, interpretation of pose and actions, even within videos (spatial and temporal streams).

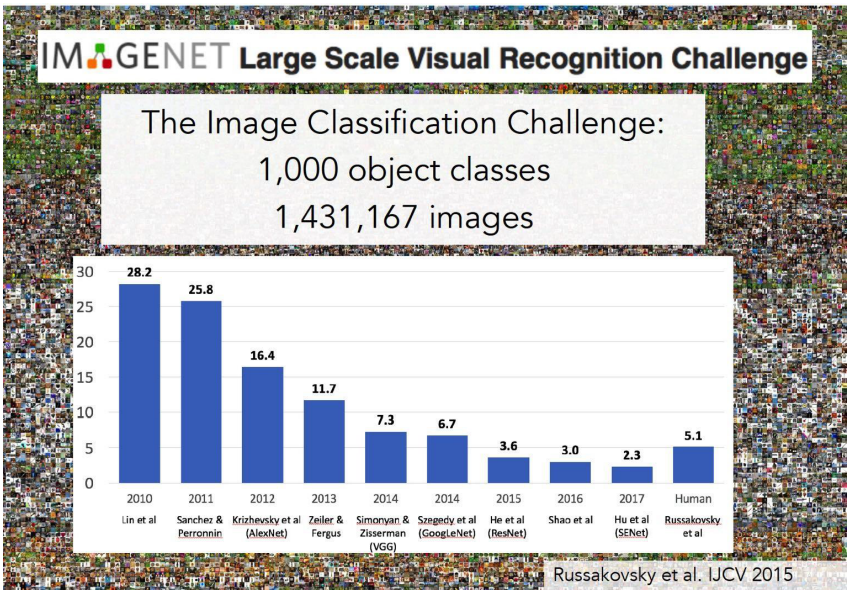

Figure 22: Successes of Deep CNNs

As we saw in section 6.1, in a **traditional NN** with an input of size N and a hidden layer of size M

- the weight matrix W has size $N \times M$,
- each node of the hidden layer is connected to each node of the input layer (that's why it's called *fully connected NN*),
- it can be seen as with M "kernels" which have dimension N each,
- there are many parameters, thus it suffers severe overfitting.

Since spatial correlation is local, and it is what mostly interests in image-related tasks, we can reduce the number of parameters per layer, to put resources elsewhere. By applying this principle, we obtain a **locally connected NN**:

- the output is based only on the *receptive field* of size $P < N$, so W is $P \times M$,
- now we have only M kernels each with dimension P,
- there are less parameters to train, so less overfitting,
- this approach is meaningful since image data are stationary: statistics is similar at different locations.

The next step is to share the same parameters across different locations:

- apply convolution with learned kernels,
- W has size $P \times 1$,
- even fewer parameters to train,
- can simultaneously train many maps to extract more features (learns multiple filters).

This approach is particularly useful for images: while standard NNs scale quadratically with the size of the input and don't leverage stationarity, **Convolutional NN** connect each hidden unit to a small

patch of the input and share the weight across hidden units.
By *pooling* filter responses at different locations, we gain robustness to the exact spatial location of features.

7.1 The architecture of CNNs

Characteristics of CNNs:

- Feed-forward:
  - convolve input,
  - non-linearity (rectified linear),
  - pooling (subsampling - see later);
- Supervised;
- Train convolutional filters by back-propagating classification error.

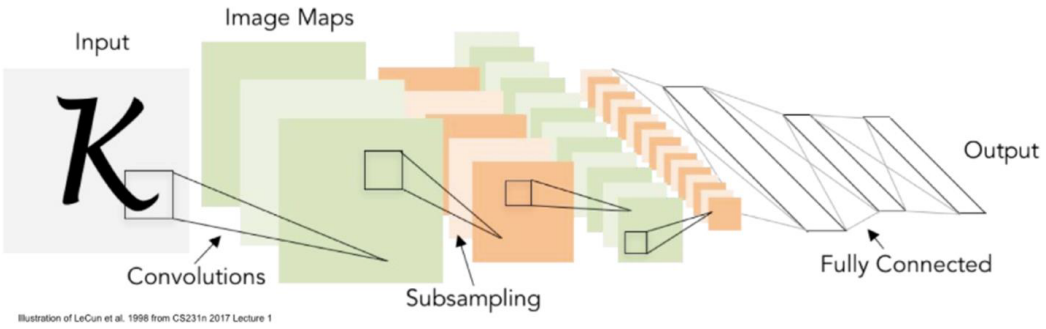

Figure 23: Convolutional Neural Network

Differently from a fully connected layer, a *convolution layer* preserve the original spatial structure, by sliding over the image spatially, computing dot products. The matrix resulting from the application of each filter is called *activation map*. Each filter produces higher level features, which at the last level compose a linearly separable classifier.
Note that filters always extend the full depth of the input volume.

The size of the produced activation map with respect to the input size depends on the filter size, and on the stride (how much to slide the filter over the input image). If the filter doesn't fit the image with the fixed stride, a zero-padding is applied.
Thus, a convolutional layer of size $W_1 \times H_1 \times D_1$ with K filters of size F, stride S, amount of zero-padding P, produces a volume of size $W_2 \times H_2 \times K$, where

$$W_2 = \frac{W_1 - F + 2P}{S + 1}, \tag{47}$$

$$H_2 = \frac{H_1 - F + 2P}{S + 1}. \tag{48}$$

The total number of parameters that have to be learned is $(F \cdot F \cdot D_1) \cdot K$ weights plus K biases.

To go on with the brain-NN analogy, we can say that an activation map is a sheet of neuron outputs: each is connected to a small region in the input and all of them share parameters.

The *pooling layer* makes the representations smaller and more manageable, and operates over each activation map independently. Possible pooling functions are max pooling or average pooling.
With an input of size $W_1 \times H_1 \times D_1$, size F and stride S, it produces a volume of size $W_2 \times H_2 \times D_1$,

where

$$W_2 = \frac{W_1 - F}{S + 1}, \tag{49}$$

$$H_2 = \frac{H_1 - F}{S + 1}. \tag{50}$$

Usually, many layers with small filters are used, since smaller filters are easier to learn. Moreover, the depth of a network is very important, since reducing it the performance drops.

Another important aspect that needs to be underlined, is the invariance property: CNN are invariant with respect to translation, scale, rotation.

7.2 Visualizing CNNs

Raw coefficients of learned filters in higher layers are difficult to interpret.

Several approaches look to optimize input to maximize activity in a high level feature.

A common approach to map activations at high layers back to the input is to use **Deconvolutional Networks**:

- They apply the same operations as CNNs, but in reverse: unpool feature maps, then convolve unpooled maps;
- In this case they are used as a probe, without inference or learning, as originally purposed;
- The result is a set of portions of an input image that give strong activation of the given feature map, which gives a non-parametric view on invariances learned by the model;
- From this, you can retrieve patches of validation images that give maximal activation of a given feature map.

8 Training Neural Networks

8.1 Activation Functions

In section 6.1 we introduced some activation functions. Now we are going to dive into deeper details to point out which is better to use.

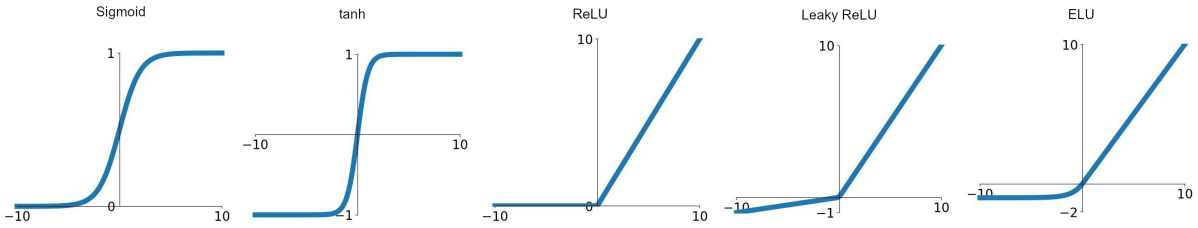

Figure 24: Some activation functions

- **Sigmoid** $\sigma(x) = \frac{1}{1+e^{-x}}$:
  - squashes numbers to range $[0, 1]$;
  - historically popular since they have nice interpretation as a saturating "firing rate" of a neuron;
  - problems: saturated neurons kill the gradients, outputs are not zero-centered, `exp()` is a bit computationally expensive;
  - if the input of a neuron is always positive, the gradients of the relative weights are all positive or all negative.
- **Tanh** $\tanh(x)$:
  - squashes numbers to range $[-1, 1]$;
  - outputs *are* zero-centered;
  - problem: saturated neurons kill the gradients.
- **ReLU** $\max(0, x)$:
  - does not saturate in positive region;
  - very computationally efficient;
  - converges much faster than *sigmoid* or *tanh*;
  - problems: outputs are not zero-centered, if the input is negative the dead *ReLU* will never activate and never update weights;
  - people like to initialize *ReLU* neurons with slightly positive biases.
- **Leaky ReLu** $\max(0.1x, x)$:
  - has all the advantages of *ReLU*;
  - does not "die".
- **Parametric Rectifier - PReLU** $\max(\alpha x, x)$:
  - similar to *Leaky ReLU*;
  - backpropagation into α parameter.
- **Exponential Linear Units - ELU** $\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$:
  - all benefits of *ReLU*;
  - closer to zero mean outputs;
  - negative saturation regime compared with *Leaky ReLU* adds some robustness to noise;
  - problem: `exp()` is a bit computationally expensive.

- **Maxout "neuron"**: $\max(W_1^T x + b_1, W_2^T x + b_2)$,
  - it is a neuron without the form of dot product (non-linearity);
  - generalizes *ReLU* and *Leaky ReLU*;
  - advantages: linear regime, does not saturate, does not die;
  - problem: doubles the number of parameters.

To sum up:

- Use *ReLU*. Be careful with your learning rates.
- Try out *Leaky ReLU* / *Maxout* / *ELU*.
- Try out *tanh* but don't expect much.
- Don't use *sigmoid*.

8.2 Data Preprocessing

Possible preprocessing approaches are zero-centering, normalization, PCA (Principle Component Analysis), whitening.

After normalization, classification loss is less sensitive to small changes in weights and easier to optimize.

When dealing with images, it is better to apply only centering, while PCA or whitening aren't common:

- subtract the mean image;
- subtract per-channel mean;
- subtract per-channel mean and divide by per-channel std.

8.3 Weight Initialization

In some cases, it is important to carefully initialize weights to avoid saturating the activation:

- Small random numbers (**activation statistics**):
  - uses a gaussian with zero mean and 1e-2 standard deviation,
  - works quite well for small networks,
  - all activations tend to zero for deeper network layers, so there is no learning, even with a larger std;
- **Xavier initialization**:
  - uses std $= \frac{1}{\sqrt{Din}}$, where for convolutional layers $Din = \text{kernel\_size}^2 \cdot \text{input\_channels}$,
  - assumes zero-centered activation function, thus it doesn't work with ReLU;
- **Kaiming/MSRA initialization**:
  - uses std $= \sqrt{\frac{2}{Din}}$,
  - activations are nicely scaled for all layers.

It's worth noting that proper initialization is an active area of research.

8.4 Batch Normalization

Consider a batch of activations at some layer. To make each dimension zero-mean unit-variance, apply the following formula:

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}} \tag{51}$$

where the input x has shape $N \times D$, $\mu_j = \frac{1}{N}\sum_{i=1}^{N} x_{i,j}$ (per-channel mean with shape D) and $\sigma_j^2 = \frac{1}{N}\sum_{i=1}^{N}(x_{i,j} - \mu_j)^2$ (per-channel variance, with shape D). The normalized x has shape $N \times D$ too.

If zero-mean and unit-variance is a too hard constraint, we can use a slight different approach:

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j \tag{52}$$

where β and γ are learnable scale and shift parameters with shape D.

Since estimates depend on minibatch, this method can't be used at test time. During testing batch normalization becomes a linear operator, and can be fused with the previous fully connected or convolutional layer.

The Batch Normalization Layer is usually inserted after Fully Connected or Convolutional layers, and before nonlinearity.

Advantages of batch normalization:

- Makes deep networks much easier to train;
- Improves gradient flow;
- Allows higher learning rates, faster convergence;
- Networks become more robust to initialization;
- Acts as regularization during training;
- Zero overhead at test time (since it can be fused with the previous layer).

Problem: behaves differently during training and testing, and this is a very common source of bugs.
Layer Normalization gives same behavior to at train and test to fully-connected networks, while **Instance Normalization** does the same job for convolutional networks.

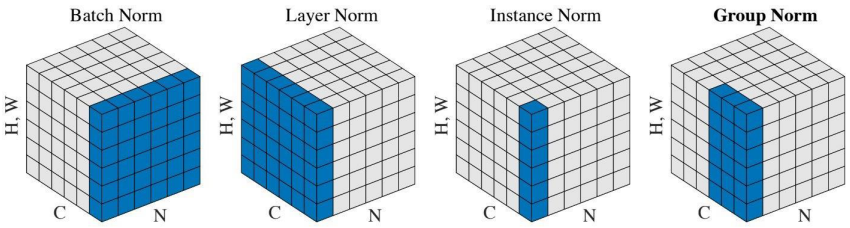

Figure 25: Normalization types

8.5 Optimizers

Stochastic Gradient Descent (introduced in section 5.6) may present some problems:

- If loss changes quickly in one direction and slowly in another, gradient descent progresses slowly along the shallow dimension and jitters along the steep direction. Loss function has high condition number: ratio of largest to smallest singular value of the Hessian matrix is large.
- If the loss function has local minima or saddle points, the gradient is zero, thus the gradient descent gets stuck.

- Gradients come from mini-batches, so they can be noisy.

A possible solution is to add **Momentum** to the computed gradient: at each iteration the gradient is added to the *velocity* (computed as a running mean of gradients) multiplied by a *friction* factor (typically $\rho = 0.9$ or 0.99). That is, weights are updated as follows:

$$v_{t+1} = \rho v_t + \nabla f(x_t) \tag{53}$$
$$x_{t+1} = x_t - \alpha v_{t+1} \tag{54}$$

Another possible and equivalent formulation is the following:

$$v_{t+1} = \rho v_t - \alpha \nabla f(x_t) \tag{55}$$
$$x_{t+1} = x_t + v_{t+1} \tag{56}$$

Another option is to use **Nesterov Momentum**: "Look ahead" to the point where updating using velocity would take us, then compute gradient there and mix it with velocity to get actual update direction:

$$v_{t+1} = \rho v_t - \alpha \nabla f(x_t + \rho v_t) \tag{57}$$
$$x_{t+1} = x_t + v_{t+1} \tag{58}$$

or, to obtain the update in terms of f_x, let be $\tilde{x}_t = x_t + \rho v_t$:

$$v_{t+1} = \rho v_t - \alpha \nabla f(\tilde{x}_t) \tag{59}$$
$$\begin{aligned} \tilde{x}_{t+1} &= \tilde{x}_t - \rho v_t + (1+\rho) v_{t+1} \\ &= \tilde{x}_t + v_{t+1} + \rho(v_{t+1} - v_t) \end{aligned} \tag{60}$$

AdaGrad adds element wise scaling of the gradient based on the historical sum of squares in each dimension (*Per parameter learning rates* or *adaptive learning rates*):

$$gs_{t+1} = gs_t + (\nabla f(x_t))^2 \tag{61}$$
$$x_{t+1} = x_t - \alpha \cdot \frac{\nabla f(x_t)}{\sqrt{gs_{t+1}} + \text{1e-7}} \tag{62}$$

In this way, progress along steep directions is damped, while progress along flat directions is accelerated. Over long time, the step size decays to zero.

With **RMSProp**, a sort of "Leaky AdaGrad", grad squared becomes

$$gs_{t+1} = \text{decay\_rate} \cdot gs_t + (1 - \text{decay\_rate}) \cdot (\nabla f(x_t))^2 \tag{63}$$
$$x_{t+1} = x_t - \alpha \cdot \frac{\nabla f(x_t)}{\sqrt{gs_{t+1}} + \text{1e-7}} \tag{64}$$

This solves the problem of radically diminishing learning rates.

Adam optimizer uses first and second moments, plus bias correction for the fact that first and second moment estimates start at zero:

$$\text{first\_moment}_{t+1} = \beta_1 \cdot \text{first\_moment}_t + (1 - \beta_1) \cdot \nabla f(x_t) \tag{65}$$
$$\text{second\_moment}_{t+1} = \beta_2 \cdot \text{second\_moment}_t + (1 - \beta_2) \cdot (\nabla f(x_t))^2 \tag{66}$$
$$\text{first\_unbias}_{t+1} = \frac{\text{first\_moment}_{t+1}}{1 - \beta_1^t} \tag{67}$$
$$\text{second\_unbias}_{t+1} = \frac{\text{second\_moment}_{t+1}}{1 - \beta_2^t} \tag{68}$$
$$x_{t+1} = x_t - \alpha \cdot \frac{\text{first\_unbias}_{t+1}}{\sqrt{\text{second\_unbias}_{t+1}} + \text{1e-7}} \tag{69}$$

Adam with $\beta_1 = 0.9, \beta_2 = 0.999$, and learning_rate = 1e-3 or 5e-4 is a great starting point for many models.

Adaptive learning rates are preferable for faster optimizations, especially as gradients become sparser. *Adam* is the most used.

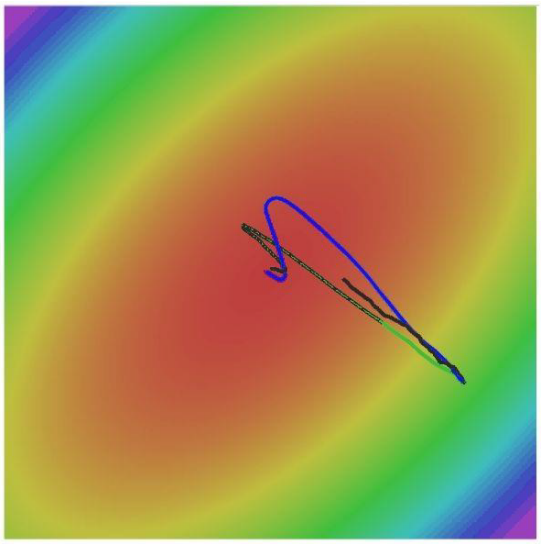

Figure 26: Different kinds of SGD and momentum. SGD in black, SGD + Momentum in blue, Nesterov Momentum in green.

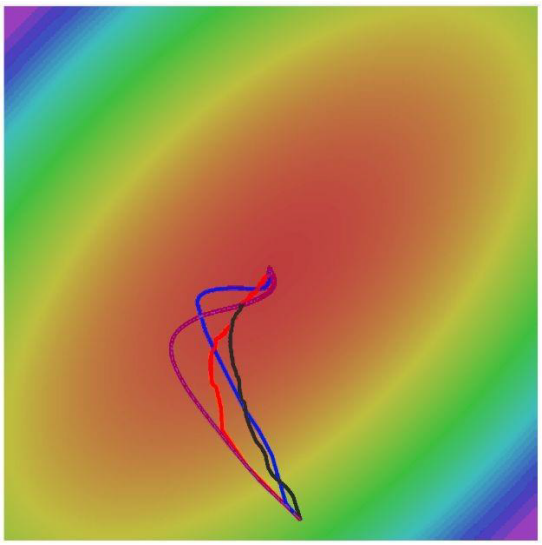

Figure 27: Different kinds of SGD and optimizers. SGD in black, SGD + Momentum in blue, RMSProp in red, Adam in purple.

8.6 Learning Rate Decay

SGD, *SGD+Momentum*, *Adagrad*, *RMSProp*, *Adam* all have learning rate as a hyperparameter, so we have to choose it. It is a good practice to begin with a large learning rate, and decay over time. There are several approaches to learning rate decay; let be α_0 the initial learning rate, α_t the learning rate at epoch t, T the total number of epochs, then:

- **Step**: reduce learning rate at a few fixed points;
- **Cosine**: $\alpha_t = \frac{1}{2}\alpha_0(1 + \cos(\frac{t\pi}{T}))$;
- **Linear**: $\alpha_t = \alpha_0(1 - \frac{t}{T})$;
- **Inverse sqrt**: $\alpha_t = \frac{\alpha_0}{\sqrt{t}}$.

Linear Warmup: High initial learning rates can make loss explode, linearly increasing learning rate from 0 over the first 5000 iterations can prevent this.

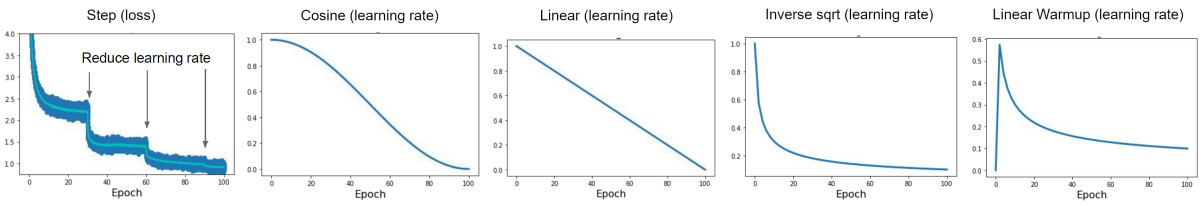

Figure 28: Learning rate decay methods

Empirical rule of thumb: If you increase the batch size by N, also scale the initial learning rate by N.

In practice, *Adam* is a good default choice in many cases, even with constant learning rate. *SGD+Momentum*

can outperform *Adam* but may require more tuning of LR and schedule (try cosine schedule).

Better optimization algorithms help reduce training loss, but we are interested in error on new data. A good way to reduce the gap between the two curves, is to stop training the model when accuracy on the validation set decreases (**early stopping**).

Model Ensembles:

- Train multiple independent models and, at test time, average their results.
- Or use multiple snapshots of a single model during training.
- Instead of using actual parameter vector, keep a moving average of the parameter vector and use that at test time (*Polyak averaging*).

8.7 Regularization

A common approach to improve single-model performance is to add a term to the loss, such as *L1* or *L2 regularization* or *Elastic net*, as we saw in section 5.4.

Another possibility is **Dropout**: in each forward pass, randomly set some neurons to zero, with probability p, a hyperparameter. This is useful because it forces the network to have a redundant representation, and prevents co-adaptation of features. Another interpretation: dropout is training a large ensemble of models (that share parameters), where each binary mask is one model. Because of how it works, dropout makes the output random. At test time we want to average out the randomness:

$$y = f(x) = E_z[f(x,z)] = \int p(z)f(x,z)dz \tag{70}$$

where y is the label, x is the input image, and z is the random mask.
The integral can be approximated by multiplying by dropout probability. Consider a single neuron: at test time we have

$$E[a] = w_1 x + w_2 y, \tag{71}$$

during training we have

$$\begin{aligned} E[a] &= \frac{1}{4}(w_1 x + w_2 y) + \frac{1}{4}(w_1 x + 0y) + \frac{1}{4}(0x + 0y) + \frac{1}{4}(0x + w_2 y) \\ &= \frac{1}{2}(w_1 x + w_2 y). \end{aligned} \tag{72}$$

In other words, at test time all neurons are active always, thus we must scale the activations so that for each neuron `output at test time = expected output at training time`.

A common pattern for regularization is to add some kind of randomness at training time, and average out that randomness (eventually with some approximation) at testing time. For example, in batch normalization, during training we normalize using statistics from random minibatches, but during testing we use fixed stats to normalize.

Another mechanism which can be used for regularization purposes, is **Data Augmentation**, obtained through image transformations such as:

- *Horizontal flips*;
- During training sample *random crops and scales*, during test average a fixed set of crops;
- *Color jitter*:
  - Randomize contrast and brightness, or
  - apply PCA to all pixels in training set, sample a "color offset" along principal component directions, add offset to all pixels of a training image;
- Combination of *translation, rotation, stretching, shearing, lens distortions...*

Other useful regularization techniques are:

- **DropConnect** drops connections between neurons (setting weights to 0) during training, and uses all the connections in testing;
- **Fractional Pooling** uses randomized pooling regions in training, and average predictions from several regions in testing;
- **Stochastic Depth** skips some layers in the network during training, and uses all the layers in testing;
- **Cutout** sets random image regions to zero during training, and uses full image in testing;
- **Mixup** trains on random blends of images, and tests with original images.

In conclusion:

- Consider dropout for large fully connected layers;
- Batch normalization and data augmentation almost always a good idea;
- Try cutout and mixup especially for small classification datasets.

8.8 Choosing Hyperparameters

To choose hyperparameters could require a lot of time or computational resources, so it is important to follow a set of guidelines:

1. **Check initial loss**: Turn off weight decay, sanity check loss at initialization.
2. **Overfit a small sample**: Try to train to 100% training accuracy on a small sample of training data; fiddle with architecture, learning rate, weight initialization.
3. **Find LR that makes loss go down**: Use the architecture from the previous step, use all training data, turn on small weight decay, find a learning rate that makes the loss drop significantly within 100 iterations.
4. **Coarse grid, train for 1 5 epochs**: Choose a few values of learning rate and weight decay around what worked from Step 3, train a few models for 1-5 epochs.
5. **Refine grid, train longer**: Pick best models from Step 4, train them for longer without learning rate decay.
6. **Look at learning curves**:
    - Losses may be noisy, use a scatter plot and also plot moving average to see trends better, while plotting *training loss*;
    - In presence of a loss drop, the problem may be bad initialization (see figure 29);
    - In case of loss plateaus, try learning rate decay (see figure 30);
    - With learning rate step decay, loss was still going down when learning rate dropped, so the decay was too early (see figure 31);
    - If *validation accuracy* is still going up, you need to train longer (see figure 32);
    - If there is a huge gap between train and validation accuracy, there is overfitting, so increase regularization, or get more data (see figure 33);
    - No gap between train/val means underfitting, so train longer, or use a bigger model (see figure 34).
7. **GOTO step 5**.

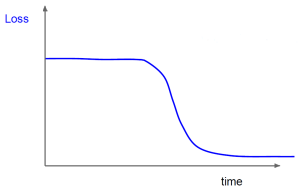

Figure 29: Loss drop

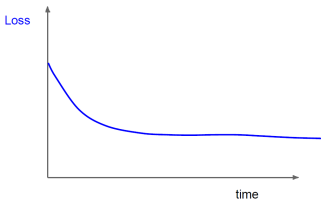

Figure 30: Loss plateaus

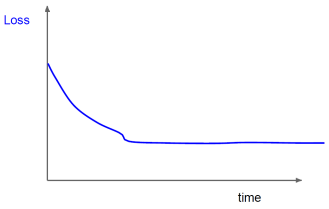

Figure 31: Learning rate step decay

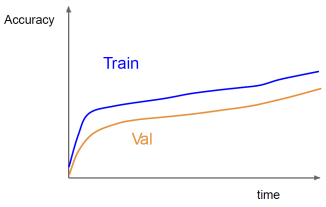

Figure 32: Accuracy still going up

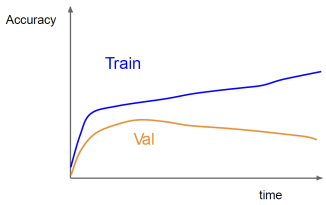

Figure 33: Huge train/val gap

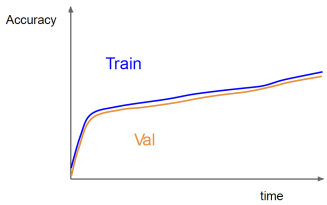

Figure 34: No train/val gap

Possible hyperparameters to tune are network architecture, learning rate and relative decay schedule, update type, regularization.

It is useful to track the ratio of weight updates/weight magnitudes: we want this to be somewhere around 0.001 or so.

8.9 Transfer Learning

The basic idea of *transfer learning* is to train a big CNN model on a huge generic dataset, then freeze most of its layers and reuse the previous learned weights to train just the last layer(s) on the smaller task-specific dataset.

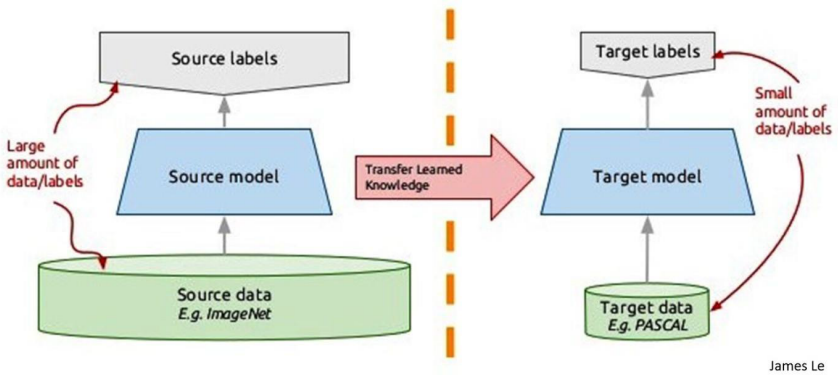

Figure 35: Transfer-learning

Depending on the situation, we need to use different approaches:

- very little data and very similar datasets: use linear classifier on top layer;
- very little data but very different datasets: try linear classifier from different stages (but there is little to do in this case);
- quite a lot of data and very similar datasets: finetune a few layers;

- quite a lot of data and very different datasets: finetune a larger number of layers.

Always keep in mind that first layers are more generic, while last layers are more specific.

Transfer learning with CNNs is pervasive, but recent results show it might not always be necessary.

- ImageNet pre training speeds up convergence early in training, but does not necessarily provide regularization or improve final target task accuracy;
- Sufficiently long training time can compensate for the lack of pre-training;
- Training from scratch may be superior to pre-training when the task is different;
- *BatchNorm* is important for training from scratch but it is generally removed after pre-training.

9 Detection and Segmentation

Computer Vision tasks:

- **Classification**: no spatial extent (see section 5);
- **Semantic Segmentation**: assign a class to each pixel in the image (no objects);
- **Object Detection**: determine position, number and class of the objects in the image;
- **Instance Segmentation**: Semantic Segmentation + Object Detection, that is, associate objects to pixel, to distinguish them.

Moreover, in *Semantic Segmentation* you can find "stuff" (like sky, grass or a forest), while in *Object detection* you will find "things" (such as cats, docs, or trees). Notice that a thing may be part of a stuff (many trees compose a forest).

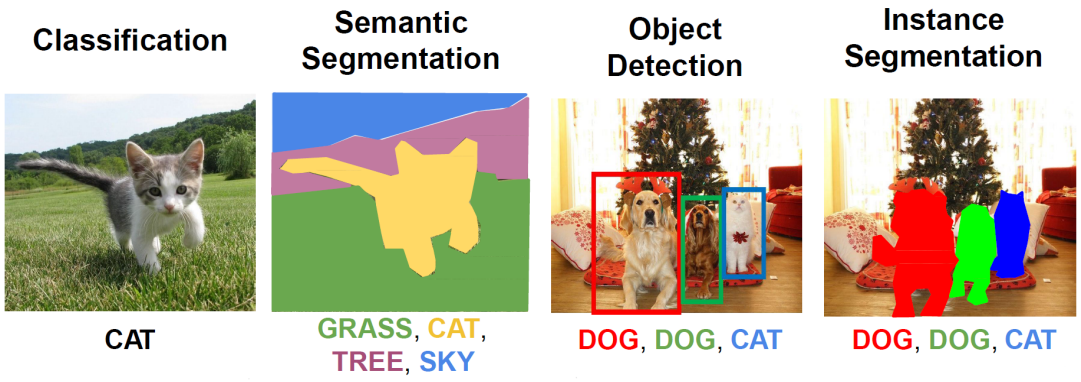

Figure 36: From Classification to Instance Segmentation

9.1 Semantic Segmentation

Label each pixel in the image with a category label. Don't differentiate instances, only care acout pixels.

The first, very basic idea, is a *sliding window*: execute a CNN on each portion of an image. This is very inefficient, since shared features between overlapping patches are not reused.

An alternative is a *fully convolutional* approach: design a network as a bunch of convolutional layers to make predictions for pixels all at once. But there is a problem: convolutions at original image resolution will be very expensive.
The solution is to introduce downsampling and upsampling into the network. **Downsampling**, by pooling and strided convolution, reduces the size of the original image, so that the following operations are cheaper. **Upsampling** allows to output a prediction for each pixel of the original image (at full resolution).

There are different approaches to upsampling:

- *Nearest Neighbor unpooling*: copy the value of a pixel in k neighbor positions, to obtain an image that is k times bigger;
- *Bed of Nails*: copy each pixel in a fixed position of the new image, and add $k-1$ zeros around it, to obtain an image that is k times bigger;
- *Max unpooling*: when doing max pooling, remember which element was max, then, in the corresponding upsampling layer, copy the pixel in that position, and surround it with zeros;

- *Transpose convolution*:
  - The filter moves k pixel in the output for every one pixel in the input, that is, stride k gives the ratio between movements in output and input;
  - In this way, output contains copies of the filter weighted by the input, summing where output overlaps;
  - It may be possible that you need to crop one pixel from the output to make it exactly $k\times$ input;
  - convolution can be expressed in terms of a matrix multiplication in this way: $\vec{x} \cdot \vec{a} = X\vec{a}$, transpose convolution can be expressed as $\vec{x}^T \cdot \vec{a} = X^T\vec{a}$.

Evaluation metrics for Semantic Segmentation:

- *Accuracy* can be very unbalanced (for example, if there is a small object on a large background, classifying all the pixels as background may give high accuracy);
- *Intersection over union*, then average across classes and images;
- *Per-class accuracy*, then average across classes and images.

It is important to notice that it is quite difficult to collect labeled data for Semantic Segmentation: it is hard to annotate precise localization, and annotating every pixel leads to heavy tails (many classes with few pixels that belong to them). A common solution is to annotate few classes and mark the rest as "other". Other possibilities are label propagation or semi-automatic labeling annotation tools.

9.2 Object Detection

9.2.1 Single Object

A possible solution to detect a single object is to treat localization as a *regression problem*, and compute a **Multitask Loss**. That is, we add two fully connected layers to a (often pretrained) CNN: one with Softmax loss for class scores, and one with L2 loss for box coordinates; and, finally, we sum up the two losses.

9.2.2 Multiple Objects

The main problem with multiple objects detection is that each image needs a different number of outputs, depending on how many objects are in the image.

A possible solution is to apply a CNN to many different crops of the image. It will classify each crop as object or background. But, in this way, we'd need to apply the CNN to a huge number of locations, scales and aspect ratios, which is very computationally expensive.

9.2.3 Selective Search

Thus, instead of applying the CNN to each possible crops, we need some method which performs a *selective search*, to which provide *region proposals*, that is, portions of the image that are likely to contain objects. These methods are relatively fast too.

Segmentation as Selective search is one such method. Its main goal is to obtain *high recall*, since it is important not lo lose any object. Since an accurate delineation is not necessary for recognition, and nearby context might be useful, is uses *bounding boxes*. Another goal is to take less than 10 seconds per image.

Since segmentation at a single scale is not enough, the method proposed to obtain high recall is based on the hierarchical grouping intrinsic into the images:

1. Start by over-segmenting the input image;
2. Compute similarity measure between all adjacent region pairs a and b, for example as:

$$S(a,b) = \alpha S_{\text{size}}(a,b) + \beta S_{\text{color}}(a,b) \tag{73}$$

   where

$$S_{\text{size}}(a,b) = 1 - \frac{\text{size(a)} + \text{size(b)}}{\text{size(image)}}$$

   (which encourages small regions to merge early) and

$$S_{\text{color}}(a,b) = \sum_{k=1}^{n} \min(a^k, b^k),$$

   with a^k and b^k color histograms (which encourages similar regions to merge);
3. Merge two most similar regions based on S;
4. Update similarities between the new region and its neighbors;
5. Go back to step 3. until the whole image is a single region;
6. Take bounding boxes of all generated regions and treat them as possible object locations.

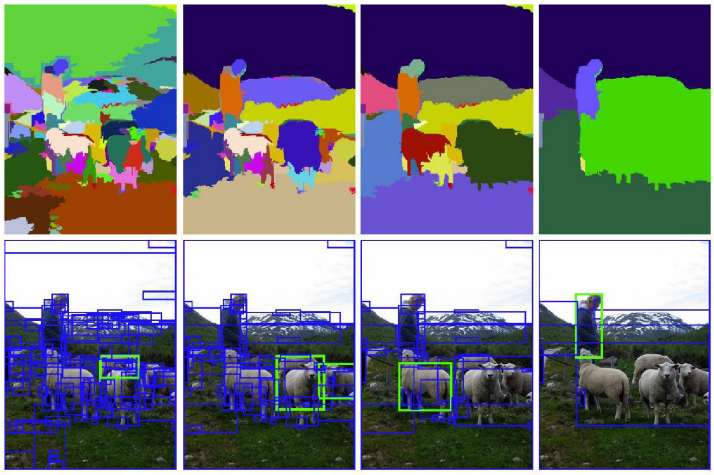

Figure 37: Selective search

To obtain higher recall, we may diversify the set of segmentations, by using different color spaces of different similarity thresholds.

To evaluate object hypotheses, we compute recall as a proportion of objects that are covered by some box with overlap greater than 50%.

9.2.4 "Slow" R-CNN

An R-CNN is a neural network structured as follows:

1. From the input image, obtain *Regions of Interest* (ROIs) from a proposal method;
2. From the ROIs, compute warped image regions of the needed size (e.g. 244×244);
3. Forward each region through a *CNN*;

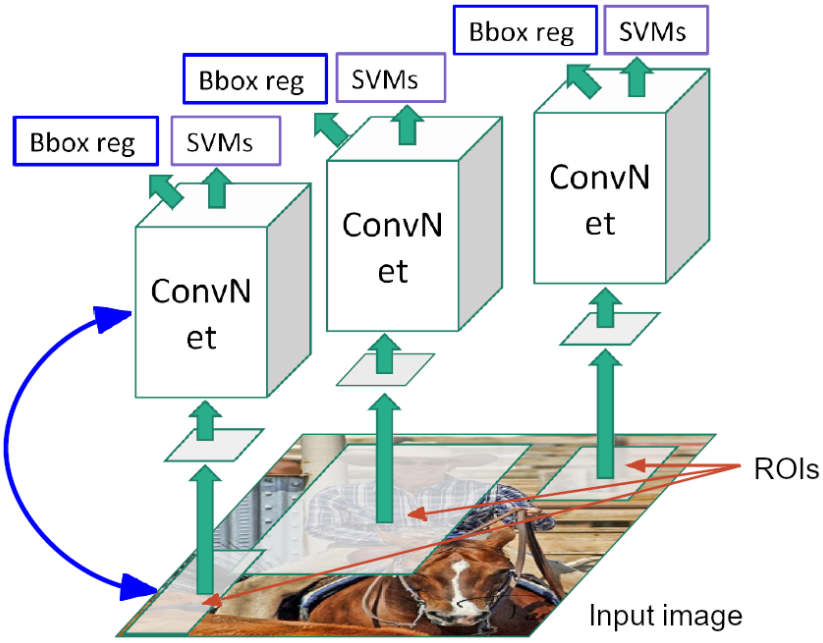

Figure 38: R-CNN

4. Classify each region with an SVM;
5. Predict corrections to the ROI.

The problem with this method is it slowness: it has to run about 2000 independent CNN for each image. A possible solution is to process image before cropping. That is, swapping convolution and cropping.

9.2.5 Fast R-CNN

The idea is to use the sliding windows of the conv-net filters to perform features extraction for all the ROIs at once.

The resulting structure is the following:

1. Process the whole input image with a *backbone network*, such as AlexNet, VGG, ResNet...;
2. Use the CNN's result to compute the "conv5 features" relative to the ROI given from a proposal method;
3. Crop and resize features;
4. Pass the output to a *per-region network*;
5. Compute Object category with a linear layer and softmax;
6. Compute the bounding box offset with a linear layer.

How to crop features? A possible approach is **RoI Pool**:

1. Project proposals onto features;
2. Snap to grid cells;
3. Divide into 2×2 grid of roughly equal subregions;
4. Max-pool within each subregion;
5. Obtain region features that always have the same size even if input region have different sizes.

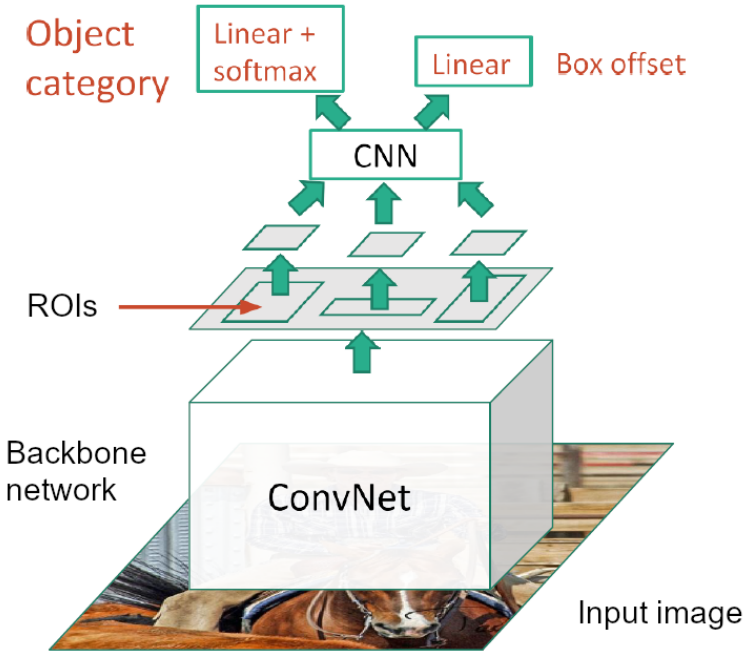

Figure 39: Fast R-CNN

This approach has the flaw that region features may be slightly misaligned. Thus, we can use another approach, i.e., **RoI Align**:

1. Project proposals onto features;
2. Divide into 2×2 grid of equal subregions (without snapping);
3. Sample at regular points in each sub-region using bilinear interpolation;
4. Feature f_{xy} for point (x, y) is a linear combination of features at its four neighboring grid cells: $f_{x,y} = \sum_{i,j=1}^{2} f_{i,j} \max(0, 1 - |x - x_i|) \max(0, 1 - |y - y_i|)$;
5. Max-pool within each subregion.

The problem with Fast R-CNN is that now the runtime is dominated by region proposals.

9.2.6 Faster R-CNN

The solution given by Faster R-CNN is to make CNN do proposals: insert a Region Proposal Network (RPN) to predict proposals from features. Otherwise same as Fast R-CNN: crop features for each proposal, classify each one.

Region Proposal Network:

1. Imagine an *anchor box* of fixed size at each point in the feature map;
2. At each point, predict whether the corresponding anchor contains an object (per-pixel logistic regression);
3. For positive boxes, also predict a transformation from the anchor to the ground-truth box (regress 4 numbers per pixel);
4. Better, use K different anchor boxes of different size/scale at each point;
5. Ignore overlapping proposals with *non-max suppression*;
6. Sort the boxes by their *object socre* and take top 300 as our proposals.

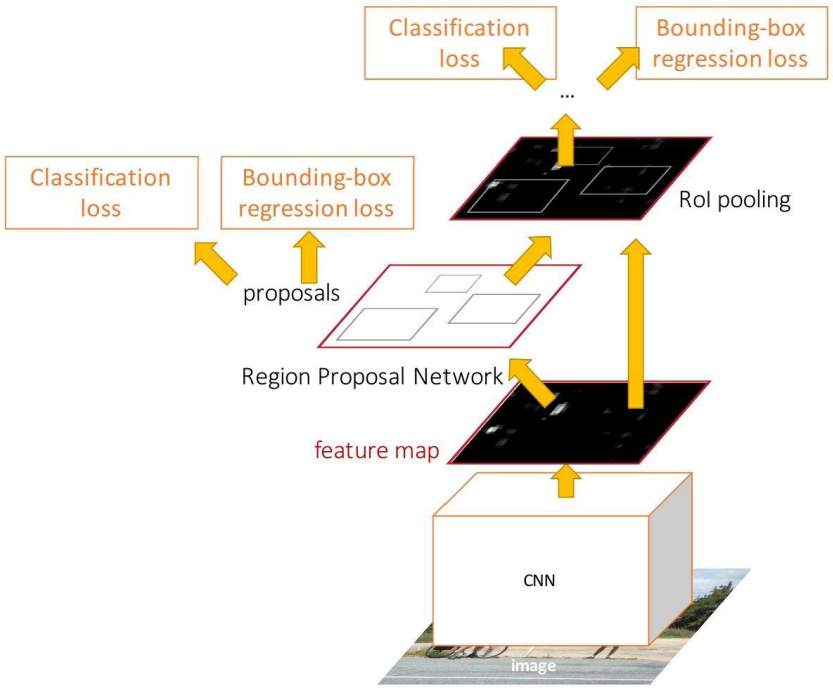

Figure 40: Faster R-CNN

So, we jointly train four losses:

1. RPN classify object/non object,
2. RPN regress box coordinates,
3. Final classification score (object classes),
4. Final box coordinates.

Faster R-CNN can be seen as a **two-stage object detector**:

- First stage: run once per image:
  - backbone network,
  - region proposal network;
- Second stage: run once per region:
  - crop features (RoI pool or align),
  - predict object class,
  - predict bounding box offset.

9.2.7 Single-Stage Object Detectors

Single-Stage Object Detectors are based on the idea that the second stage is not really necessary: it is sufficient to specialize the *object/non object classifier* in the RPN to predict each possible class, plus "background". Going into more detail:

1. Divide the input image into a 7×7 grid;
2. Create a set of B base boxes centered at each grid cell;
3. Within each grid cell:
   3.1. Regress from each of the B base boxes to a final box with 5 numbers: `(dx, dy, dh, dw, confidence)`;

3.2. Predict scores for each of C classes (including *background*).

Examples of such detectors are **YOLO, SSD, RetinaNet**.

9.3 Instance Segmentation

To perform Instance Segmentation task, we can use a network slightly different from a Faster R-CNN, which is called **Mask R-CNN**. It consists in a Faster R-CNN where we add a small mask network that operates on each RoI and predicts a 28×28 *binary mask* for each of C classes.

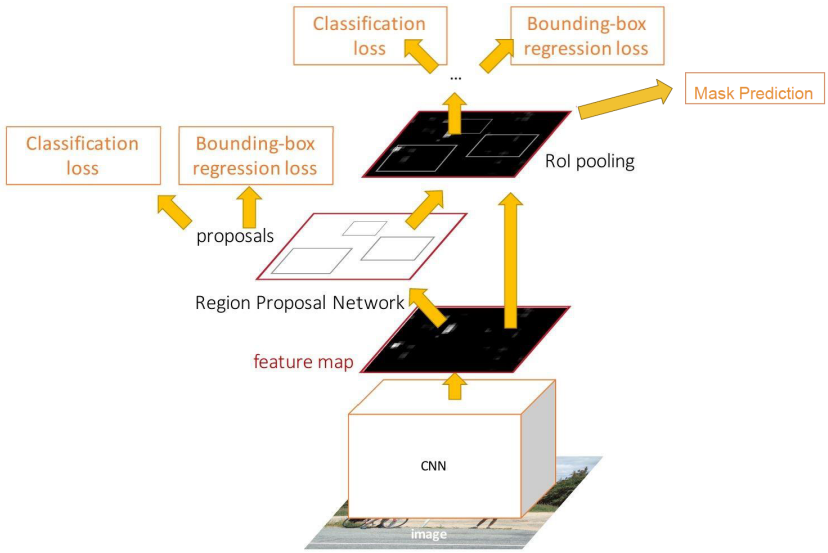

Figure 41: Mask R-CNN

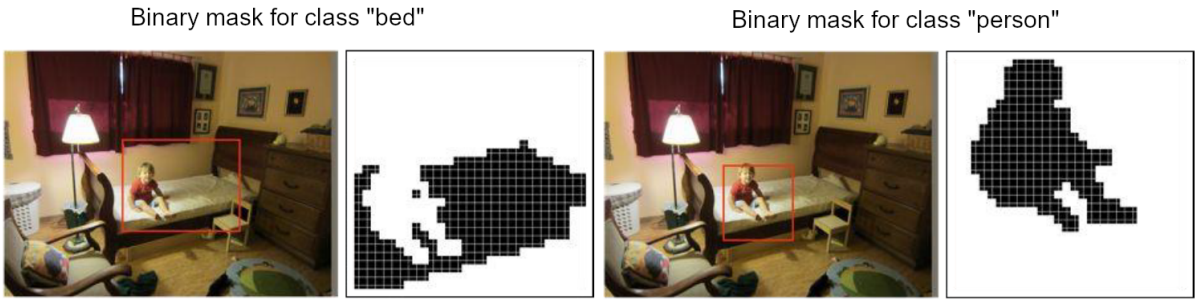

Figure 42: Binary Masks

In addition to the impressive results when applied to instance segmentation, it's worth notice that Mask R-CNN is useful for *pose detection* too.

To evaluate how an instance identification task is performed, we clearly need labeled datasets, where each image is tagged with a class for each pixel or region. One such dataset is **COCO**, Common Objects in Context.

9.4 Beyond 2D Object Detection

Other more advanced and very interesting tasks are:

- **Dense Captioning**: Object detection + Captioning, that is, detect each object in an image and give a textual description for them (e.g. "Man wearing black shirt");

- **Scene Graphs**: Objects + Relationships, i.e. recognize, predict and describe relationships between objects in one or more images;
- **3D Object Detection**: predict 3D oriented bounding boxes, for this task a model of the observing camera is needes:
  - **Simple Camera Model**: a point on the image plane corresponds to a *ray* in the 3D space, a 2D box on an image is a *frustrum* in the 3D space, the object can be anywhere in the camera viewing frustrum;
  - **Monocular Camera**: use the same idea as Faster R-CNN, but make proposals sampling in 3D space and projecting in 2D space;
- **3D Shape Prediction**: use cubes (*voxel*), spheres (*pointcloud*) or triangles (*mesh*) to approximate the predicted 3D shape of an object in an image.

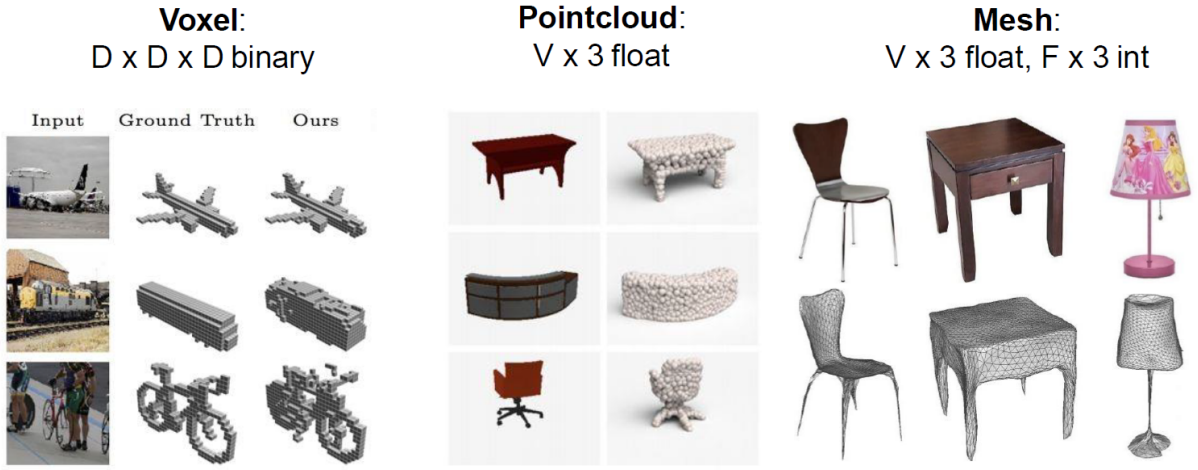

Figure 43: 3D shape prediction

10 Visual search, object retrieval and person re-identification

Visual search principle: given a new image of a specific object or person, find other images relative to the same instance represented in the given one.

Visual search applications:

- reverse image search,
- web search engine,
- personal photo collection,
- geolocalization,
- query for more information,
- shopping interfaces,
- ambient intelligence.

Main challenges in this task are illumination, object pose, clutter, occlusion, intra-class appearance, viewpoint. Furthermore, the problem is inherently ambiguous: it is very hard to understand what the user means with a single query image, and the user goal is application dependent, thus it is very important to leverage prior information given during training.

Key considerations:

- What is the measure of similarity?
- How do we represent the image/video?
- What is the search procedure?
- What background knowledge is available, or what can be learned?

The goal of *person search* is to find a queried person within a gallery of images (detection and re-identification). Motivations can range from find missing person to surveillance, to access granting.

Notes:

- Detection algorithms find and localize all object instances of a class and all people;
- Inference is on unseen object instances and people identities (compare with classification).

10.1 Object Search (or Instance-Level Retrieval)

Goal: find specific object instances.

The first problem, when it comes to Object Search, is how to represent objects.
Early methods used low-level clues, such as color histograms, texture and shape descriptors. However, they couldn't deal with appearance variation. So, different approaches are necessary.

10.1.1 Local representations

Local representations leverages on **local descriptors**, which guarantee invariance by capturing the local appearance, and thus are discriminative; furthermore, they assure repeatability by carefully choosing locations. This approach requires more sophisticated matching, though.
The *pipeline* is the following:

1. Extract local descriptors from the query image,
2. Compare local descriptors of the query image with local descriptors of each gallery image,

3. Apply geometric verification to the resulting list of matches,
4. Return a shorter list of retrieved images.

Geometric verification:

- From a set of local pairwise matches, find the geometric transformation that fits the highest number of matches,
- Retain only images with enough matches for the estimated transformation,
- Provides coarse object localization,
- Is a costly process, thus we need to verify only a short list of images.

There are different approaches to scale the first selection process. On the one hand, efficient approximate nearest neighbor search on local descriptors, using Randomized K-DTrees or Locality-Sensitive-Hashing. On the other hand, **quantization** of local descriptor space into a visual codebook. That is, discretize the local descriptor space and let two descriptors match if and only if they fall in the same bin. This concept applies to **Visual Words** (see figure 44):

1. Extract some local features from a number of images;
2. Quantize the feature space, where each point is a descriptor vector;
3. Map high dimensional descriptors to tokens/words by quantizing the feature space:
   3.1. Quantize via clustering, let cluster centers be the prototype *words*;
   3.2. Determine which word to assign to each new image region by finding the closest cluster center.

To leverage quantization for efficient retrieval, first create an inverted file with all descriptors of the gallery set, then use this for efficiently matching a query image.

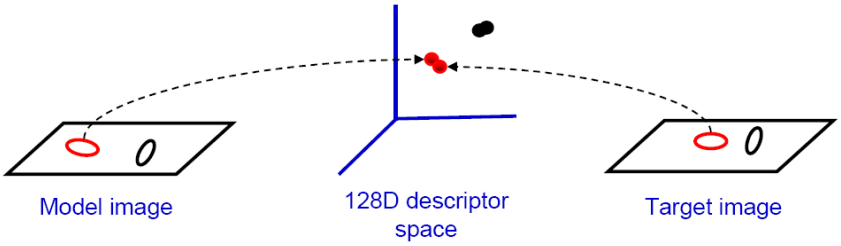

Figure 44: Visual Words

10.1.2 Global representations

This full pipeline is a costly approach, because of the memory cost of storing all the local descriptors, and the computational cost of matching and verification. Thus, an alternative cheaper approach is to use global representations: **global descriptors** aggregate individual local descriptors per image, then simple similarity measures can be applied between global descriptors.

An example of this approach is **Bag of Words**:

- Extract local descriptors;
- Convert them into visual words, using a visual codebook;
- Represent images as histograms of occurrences.

In this way, the obtained representation is relatively coarse, thus we can increment the number of entries in the codebook, but with a significant computational cost, or use other statistics, such as mean and variance. **Mean: VLAD** (Vector of Locally Aggregated Descriptors)

- Aggregate all descriptors assigned to the same visual word;
- Concatenate vectors for individual words.

Mean + Variance: Fisher Vector

- Probabilistic codebook as a mixture of Gaussians;
- Descriptors soft assigned to words;
- Compute the gradient of the log likelihood with respect to the parameters of the model;
- Aggregate the contribution of all local descriptors;
- Concatenate statistics for all the visual words.

We can use **mean Average Precision** (mAP) to evaluate those different approaches. The result is that *matching-based methods* (i.e., local representations) give highest accuracy, but with the high cost of matching and geometric verification, while *aggregation methods* (i.e., global representations) are faster and more efficient, but give lower accuracy.

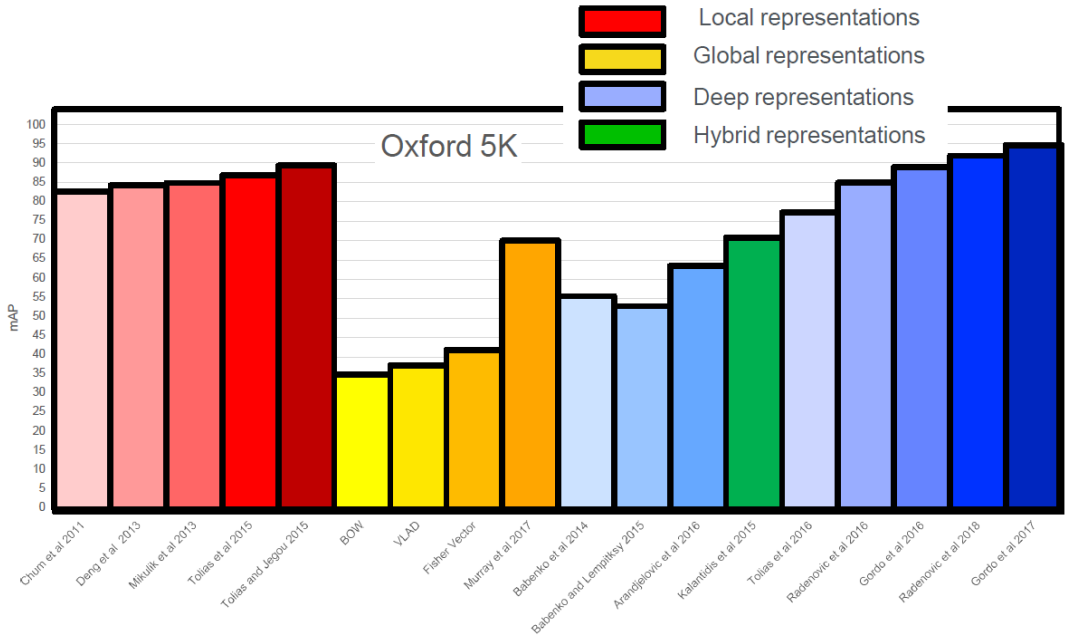

Figure 45: Evaluation of different kinds of representation on Oxford 5K dataset

10.1.3 Deep representations

A naive deep learning approach consists in pretraining a network for classification and using it as a feature extractor. The resulting representations are compact and fast at test time. The problem is the network is trained for generic classes (intra-class generalization) and the produced representation has low resolution and distorted aspect ratio, thus leading to underwhelming results.

The first solution is to combine with more standard approaches, that is, *hybrid methods*:

- **CNN + VLAD = NetVlad**: add a VLAD block at the end of a CNN;
- **Fisher Vectors + Fully Connected Layers**: combine some unsupervised for computing low level descriptors with supervised fully connected layers;

The second solution is to improve the pipeline to tailor it to the retrieval task, using *recent deep approaches*. A first approach of this kind, may be to collect and leverage an appropriate training set, and

then fine tune the network. But there are some limitations: images are distorted, the network still uses a classification loss, public landmark datasets are noisy. Various aspects can be improved:

- *Training data:* automatic cleaning of the landmark images, resulting in less images, but with bounding box annotations;
- *Architecture*:
  - Small details are important for instance level retrieval: need to accommodate high resolution, undistorted images during training;
  - Use **R-MAC descriptor** to obtain local features from input images;
  - Compute local features using a CNN, then aggregate them into region feature vectors, and combine them into a final global representation;
  - Advantages: no aspect ratio distortion, can encode high resolution images, fast comparison with the dot product;
  - Observations: The aggregation steps can be integrated inside the network, since every step is differentiable, the model can be trained end-to-end;
- *Training objective*:
  - Train explicitly for retrieval;
  - To learn to rank, use a network that takes in input three images: the query image, a relevant image and a non relevant image (see figure 46);
  - Use a *triplet loss* which combine information from all the input images:

$$L_v(q, d^+, d^-) = \frac{1}{2}\max(0, m + \left\|q - d^+\right\|^2 - \left\|q - d^-\right\|^2), \tag{74}$$

where q is related to the query image, d^+ to the relevant image and d^- and m to the non relevant image.

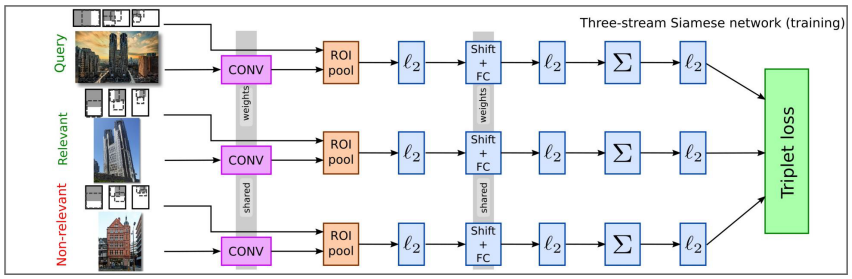

Figure 46: Training for retrieval

10.2 Person Search

Goal: find specific person identities.

10.2.1 Person Re-Identification

Sub-goal: find the queried person in a gallery of cropped samples. Retrieve the same person from different cameras, assuming that image crops are produced by a person detector.

The datasets for image re-identification are composed of boxes (hand written or obtained from a detector). The boxes in the training set are associated with an identity; the images in the test set represent people not seen at training. Thus, the algorithm should generalize to recognize unseen people from a query.

The re-identification task poses many challenges:

- Inaccurate detections;
- Large variations of viewpoint and poses;
- Large inter id variation, similar appearance of different identities;
- Occlusions;
- Varying lighting/weather conditions;
- Different camera setups and resolutions.

There are two main evaluation metrics we can use for re-identification:

- **Cumulative Matching Characteristic**:
  - As from object retrieval literature;
  - Probability that a match will be found in top-k results for a query.
- **Mean Average Precision**:
  - Similar to evaluating classification;
  - Specific emphasis on recall;
  - A perfect re-identification model should recall all true matches;
  - Sort the gallery by the score similarity function with respect to a certain query, compute the average precision over the obtained vector, average over all the queries;
  - Weighted mean of precisions achieved with increasing thresholds.

There are many different approaches to person re-identification, but the fundamental points are:

- Find representations invariant of pose, viewpoint, illumination...
- Fix the alignment issue, using spatial transformer networks, pose estimation/part or joint localization, or attention mechanism.

The approaches we discuss are CNN-based:

- Perform feature extraction by learning features embeddings, apply identification models and Triplet Loss (see 74), then compare identities by their embeddings (obtained from the last fully connected layer of the network).
- Use verification models with binary classification loss, that is, a network which takes in input a pair of images, extracts local features separately for each image, then measures similarity with another sub-network, and compute a loss that predicts if the images represent the same person or different persons.

A simple **Siamese framework**, with pre-trained *ResNet* nets and a ranking triplet loss provides the best embeddings. The architecture is shown in picture 47, but there ae some details which need further discussion:

1. input size (do not distort images);
2. data augmentation;
3. backbone architecture (use best NN architectures);
4. pooling (max pooling better than average pooling);
5. triplet sampling mechanism;
6. model pretraining.

The base mechanism used by this net is called **Curriculum Learning**:

- Pre-training for classification (ImageNet first, then classification - see step #6);
- Increasingly difficult data transformation (image cut-out of increasing sizes - see step #2);
- Hard triplet mining (see step #5).

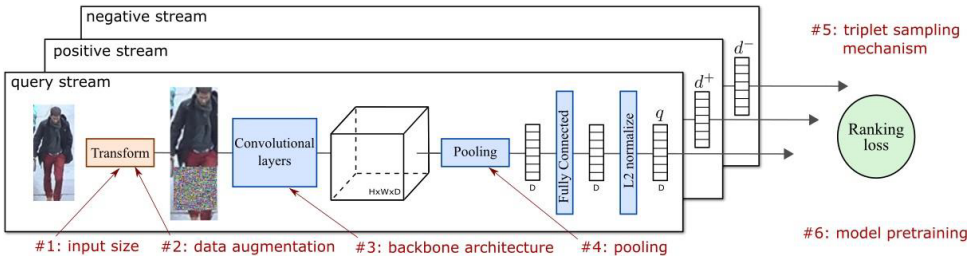

Figure 47: Siamese network for person re-identification

Grad-CAM highlights regions that activate dimensions in the resulting embeddings, so we can see which regions matter for matching image pairs, referring to the dimensions that contribute most to the similarity.

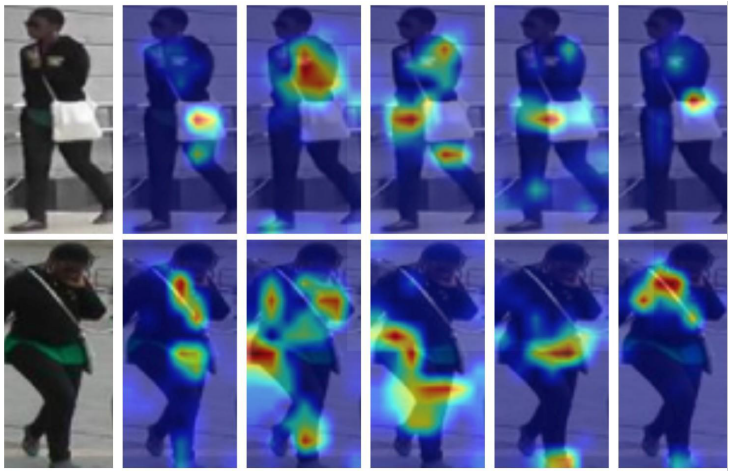

Figure 48: Grad-CAM

10.2.2 Person Search

Sub-goal: find a queried person within a gallery of images (detection + re-identification, without cropping gallery images).

This task is closer to actual applications such as missing persons or surveillance.

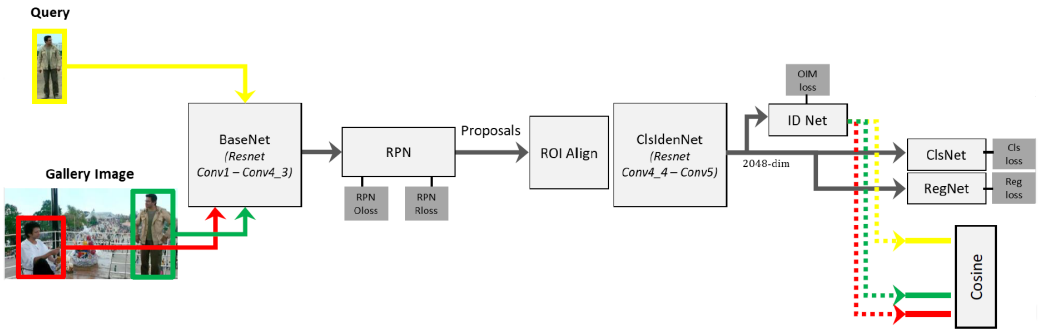

Figure 49: OIM network

The idea behind the **OIM** network (in figure 49) is to combine a *Faster R-CNN* detection network to *ID Net*, a network for re-identification vector embedding.

The objective of the network is to minimize the distance among features from the same identities and maximize those from different identities. The acronym OIM stands for the *Online Instance Matching* lookup table, containing features of labeled and unlabeled identities, to compare against, and used for learning ID Net. Softmax converges slowly and needs more parameters with more identities.

Since the next model we will study uses this module, let's now introduce the **Squeeze-and-Excitation Network** (SE-Net): it performs feature recalibration based on channel interdependencies, giving better results then ResNet on the ImageNet dataset. (See figures 50 and 51)

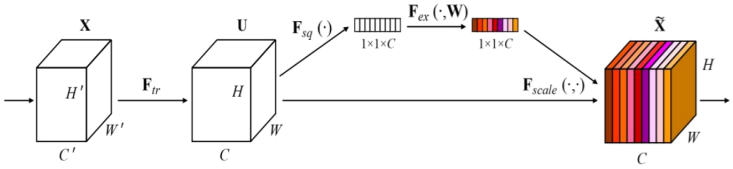

Figure 50: SE-Net

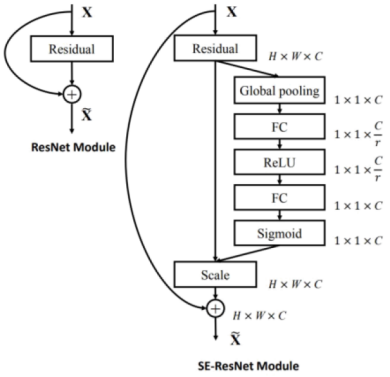

Figure 51: SE-Net vs ResNet

The principle on which **QEEPS** (Query-guided End-to-End Person Search) is based, is to use the query image extensively and learn end to end. That is, the query image is complete, not cropped, so that the network can use the context (e.g. illumination, colors, ...) to better distinguish the person we are looking for.
The architecture (see figure 52) is composed as follows:

- *Siamese Faster RCNN*: processes the query and gallery image in the same way and learns from the joint feedback;
- *Re-ID network*: performs the re-identification part of the task;
- *QSimNet* (see figure 53): metric learning of re-identification vector distances, i.e., the similarity between the query image and the gallery image is determined using a NN instead of the cosine similarity (as in OIM);
- *QRPN* (Query-guided RPN - see figure 54): query-specific person proposals;
- *QSSE* (Query-guided Siamese SE-Net - see figure 55): global-context guidance to the base net.

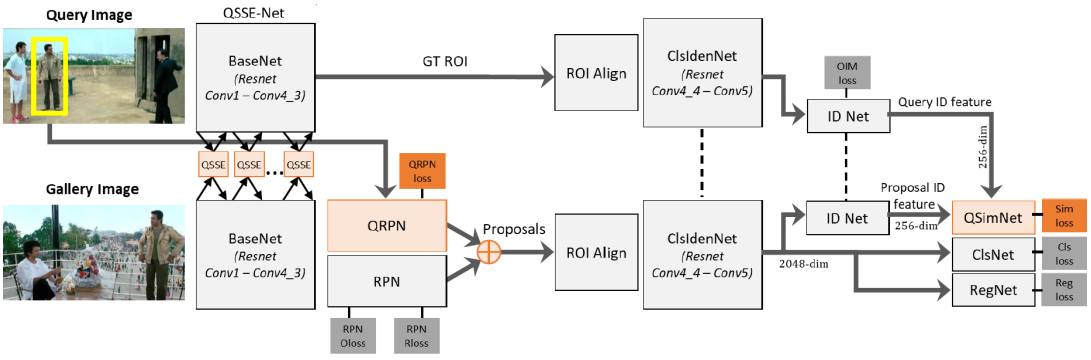

Figure 52: QEEPS

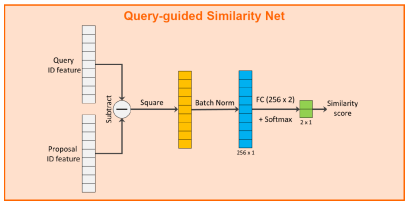

Figure 53: QSimNet

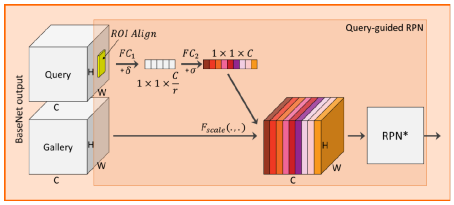

Figure 54: QRPN

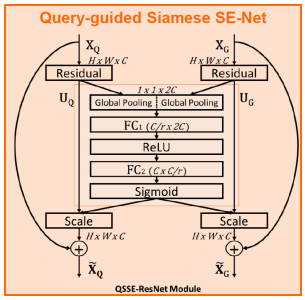

Figure 55: QSSE

The corresponding overall loss function is computed as follows, as a combination of different losses:

$$L = \underbrace{\lambda_1 L_{cls} + \lambda_2 L_{reg} + \lambda_3 L_{rpn_o} + \lambda_4 L_{rpn_r}}_{\text{Faster R-CNN losses}} + \underbrace{\lambda_5 L_{oim}}_{\text{OIM loss}} + \underbrace{\lambda_6 L_{qrpn}}_{\text{QRPN loss}} + \underbrace{\lambda_7 L_{sim}}_{\text{QSimNet loss}} \tag{75}$$

where L_{qrpn} and L_{sim} are cross-entropy losses, i.e. $-\frac{1}{N}\sum_N \log(p_n^u)$, and QSSE Net gets the implicit same supervision as the BaseNet.

Evaluation:

- *Ablating the model parts*: incremental tests are performed by using only OIM first, then adding the other modules; the result is that QSimNet is the module that gives the most relevant performance improvement, but it is still useful to combine all the modules.
- *Comparison to state-of-the art*: using larger image resolution gives state-of-the art performance, and in general this model gives consistent improvement across datasets and image resolutions.
- *Qualitative results*: plotting the test images with the results given by QEEPS shows that it can distinguish even very similar persons, also in challenging conditions (e.g. from the back).

10.3 Semantic Search

Goal: mine patterns in large collections.

Queries can be complex and involve the full scene, instead of a single object as in Object Search (see section 10.1).

The intrinsic complexity of the task causes that even human annotations are difficult, since different annotators can give different descriptions of an image and its meaning. Thus, the evaluation si based on a leave-one-user-out agreement score: 89.1 ± 4.6.

An attempt to capture semantic information in images has been done with the *Visual Genome dataset*, composed of thousands of images with millions of descriptions and object instances.

Practically speaking, the objective is to learn a compact visual representation for the semantic search task. Training data is composed of images with human captions, privileged information which can be leveraged as a proxy for semantic similarity. That, is: two image are semantically similar if their captions are similar. This idea can be applied to visual and textual representations: *visual representations* of images with similar captions are close in the semantic embedding space, *textual representations* of corresponding captions are close in the semantic embedding space.

To learn such a **semantic embedding**, we can use a Three-stream Siamese Network like the one used for deep representations in visual search (see figure 46, section 10.1.3) and multiple losses similar to the

previously cited triplet loss (see 74):

$$L_v(q, d^+, d^-) = \frac{1}{2} \max(0, m - \phi_q^T \phi_+ + \phi_q^T \phi_-) \quad \text{(Visual loss)}$$

$$L_{t1}(q, d^+, d^-) = \frac{1}{2} \max(0, m - \phi_q^T \theta_+ + \phi_q^T \theta_-) \quad \text{(Textual loss 1)}$$

$$L_{t2}(q, d^+, d^-) = \frac{1}{2} \max(0, m - \theta_q^T \phi_+ + \theta_q^T \phi_-) \quad \text{(Textual loss 2)}$$

Possible evaluation metrics are:

- *Evaluation on the triplets*: User-based score (agreement score between visual predictions and users' ground truth);
- Evaluation on the full test set (agreement score between visual predictions and proxy ground truth, i.e. human-generated captions):
  - NDCG: normalized discounted cumulative gain,
  - PCC: Pearson's correlation coefficient.

The joint embedding allows for multimodal queries.

To tackle more complex queries, it is useful to implicitly understand or explicitly model scenes.

11 Pose estimation

Pose estimation consists in detecting joint locations for persons, such ase nose, shoulders, elbows, hips... This task poses several challenges: strong articulations, small and barely visible joints, occlusions, the need to capture context.

11.1 Human pose estimation given detection

Assuming people have been already detected, and a rough bounding box is given, we can leverage information about scale and number of persons in the image. We have the advantage of disentangling detection and pose estimation, but the disadvantage of an unrealistic hypothesis.

A common practice for evaluating this task is the following:

- Evaluate every key-point separately;
- For each person, check if key-point is correct;
- Compute fraction of people for which key-point is correct (*Probability of Correct Key-point*).

Early methods comprise:

- Graphical models with handcrafted unaries, pre-defined pairwise constraints;
- Pictorial structures:
  - tree structured graphical models to represent spatial correlations,
  - kinematic priors between connected limbs,
  - occlusions, errors due to correlations outside the model;
- Non-tree models:
  - additional edges for symmetry, occlusion, long range interactions,
  - can only use simpler approximate inference.

More recent approaches use R-CNN: given an input image, extract region proposals, compute CNN features, and finally classify regions. Then there are many ways to refine proposed bounding boxes. The first one is **Regression**:

- Find and refine targets relative to bounding box;
- It is necessary to combine local and global appearance;
- **Deep networks** take whole images as input and use global information for joint regression, then regress joint locations, so there is no need to design detectors or explicitly model joint relations and spatial constraints;
- The objective of the network is to minimize L2 distance between the prediction and the true pose vector, and the the output is the pose vector normalized with respect to bounding box: $y^* = N^{-1}(\psi(N(x); \theta))$;
- On the one hand, DNs are simple and holistic, don't need to define losses to capture interactions, have hidden layers shared by joint regressors;
- On the other hand DNs have limited ability to consider details;
- With the same network architecture but different parameters (i.e. θ_s) we can build a **Cascade of regressors**, where the CNN regressor at stage $s > 1$ is trained with predicted joint from stage $s - 1$;
- W.r.t. DNs, this method obtains increasingly detailed predictions along cascade stages, but can produce only one prediction per image, no candidates, and the final result depends on the quality of the initial prediction;

- It is also possible to consider multimodal distributions.

The second one is **Heatmaps**:

- In training, each key-point is a separate binary heatmap, where the key-point is positive and the rest is negative;
- Possible approaches are:
  - *Softmax* over all locations: $p(x,y) = \frac{e^{s(x,y)}}{\sum_{x',y'} e^{s(x,y)}}$,
  - *Sigmoid* at each location: $p(x,y) = \frac{1}{1+e^{-s(x,y)}}$;
- The resolution issue may have different solutions, such as dilation, multiple layers, multiple image scales (multi-resolution feature extraction, re-utilization of convolutional features).

These methods can be combined by using heatmap to predict coarse location and then predicting an offset $(\Delta x, \Delta y)$ at each location.

It is important to consider that key-points are not all independent, given the image. For example, a person can't have right elbow and left wrist on the same arm. Thus, we have to capture *key-point dependence*, with **structured prediction**: let l be a candidate location for each key-point, then we have

$$E(l) = \underbrace{\sum_i s_i(l_i)}_{\substack{\text{score from} \\ \text{heatmap}}} + \underbrace{\phi(l)}_{\substack{\text{consistency} \\ \text{between joints}}} \tag{76}$$

$$= \sum_i s_i(l_i) + \sum_{ij} \phi_{ij}(l_i, l_j) \tag{77}$$

The optimal l^* is given by

$$l^* = \arg\min E(l) \tag{78}$$

which is a conditional random field, where ϕ is unknown and needs to be learned.
Dependence between key-points can be represented with pictorial structures, such as in figure 56.

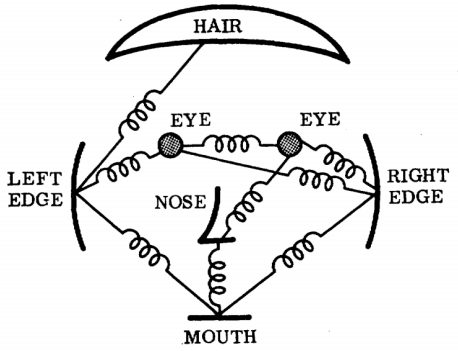

Figure 56: Pictorial structures

Another possible approach is to learn a *flexible mixture of parts*:

$$S(I,L) = \sum_{i \in V} \alpha_i \cdot \phi(I, l_i) + \sum_{ij \in E} \beta_{ij} \cdot \psi(l_i, l_j) \tag{79}$$

where $\psi(l_i, l_j)$ represents spatial features between l_i and l_j, and $\beta_{i,j}$ represents pairwise springs between part i and part j.
This can be done with **structured SVMs**:

- Very large output spaces,
- A scoring function that scores input-output pairs $h_w(x,y)$,
- Predicted output is arg max of the scoring function,

- Loss is margin rescaled loss.

So, inference is performed as follows:

$$E(l) = \sum_i s_i(l_i) + \sum_{i,j} \phi_{i,j}(l_i, l_j) \tag{80}$$

$$l^* = \arg\min E(l) \tag{81}$$

$$l_i^* = \arg\min_{l_i} \left(s_i(l_i) + \sum_j \phi_{i,j}(l_i, l_j^*) \right) \tag{82}$$

$$l_i^{(t+1)} \leftarrow \arg\min_{l_i} \left(s_i(l_i) + \sum_j \phi_{i,j}(l_i, l_j^{(t)}) \right) \tag{83}$$

That is, instead of learning scoring function, we apply **iterative models** to approximately minimize it. This approach is similar to Inference machines and Autocontext (the former share the parameters, the latter don't - see figures 57 and 58).

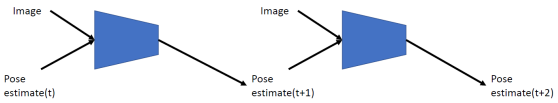

Figure 57: Inference machine

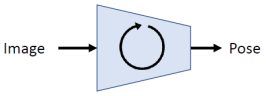

Figure 58: Autocontext

Going into more detail:

- In each iteration, beliefs of one variable are updated using current beliefs of the others, according to equation 83;
- Frame each iteration of inference as a differentiable function;
- Write inference as a convolutional network.

For example:

- $P(\text{eye at } p) = \sum_q P(\text{eye at } p | \text{nose at } q)$,
- $P(\text{eye at } p | \text{nose at } q)$ only depends on relative location of p and q,
- $f(p) = \sum_q w(p-q)g(q)$: convolution,
- $f = w \cdot g$.

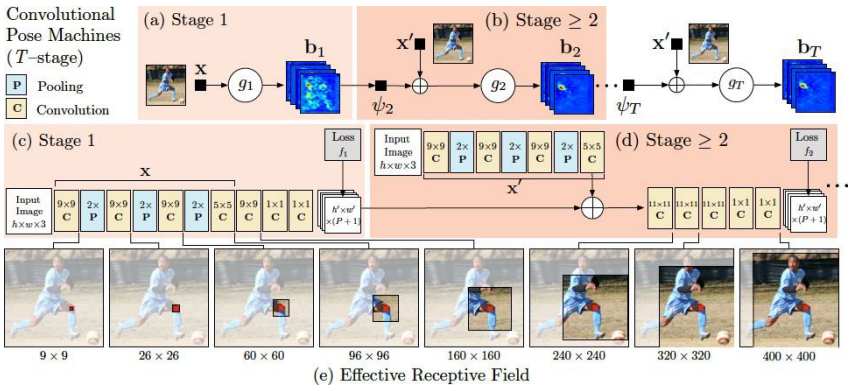

Figure 59: Convolutional Pose Machine

Differently from other models such as **FLIC** or **LSP**, in **Convolutional Pose Machines** each iteration can involve multiple convolution/subsampling layers over beliefs from previous iteration. In this network (see figure 59), each stage sees two inputs: image features and context features (based on output from

previous stage). Its goal is to achieve large receptive fields to learn complex and long range interactions. First stage:

- Predict part belief based on local image values;
- Output $P + 1$ heat maps (P for parts and 1 for background);
- Small receptive field, to capture relation between close parts (e.g. between head and shoulders, but not head and knees).

Following stages:

- Image features x' from the previous stage;
- Context function ψ encodes landscape of belief maps around part locations (ψ is the receptive field, actually);
- In this way, the network decides how to combine features and learn higher relations, and hand-defined graphical models are no more needed;
- Receptive field size can be increased by more pooling (with loss of local details), larger filters (but the number of parameters increases) or more layers (with vanishing gradient problem).

Since magnitude of backpropagated gradients decreases rapidly in initial layers, intermediate supervision are used to ensure greater variance in gradients.

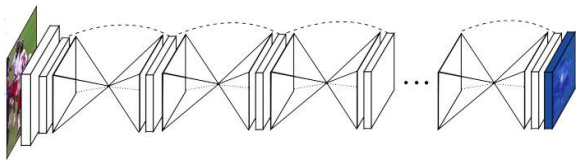

Figure 60: Staked Hourglass Network (a)

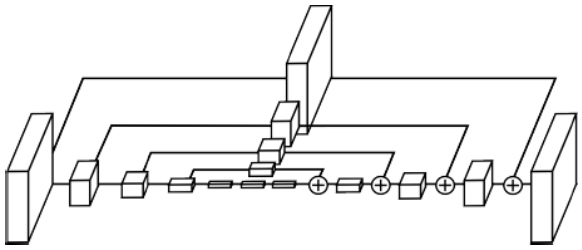

Figure 61: Staked Hourglass Network (b)

An alternative model, which can be used for this task, is the **Stacked Hourglass Network**. As the name suggests, and as figure 60 shows, it has a "hourglass structure". Each refinement round has to combine global information about pose, and use this information to produce new precise pose estimate. Moreover, rounds don't need to share parameters.
This network combine local appearance, needed for accurate part detection, with global reasoning, useful for orientation of body, limb arrangement and part relationships. It is designed to process multiple scales and achieve pixel-wise predictions, by applying convolution and max-pooling to very low 4×4 resolution and then upsampling and combining with skip connection (see figure 61). The refinement is obtained through iterative stages, by reducing and augmenting resolution. It is also possible to apply intermediate supervision for each stage:

- Network has had a chance to reason both locally and globally,
- Subsequent hourglass modules can reassess high order spatial relations,
- Ask network to repeatedly reason across scales,
- 1x1 convolution to add intermediate heatmaps to feature channels.

Stacked (cascading) hourglass is better than long hourglass, which, in turn, is better than short hourglass. Furthermore, intermediate supervision helps both single an stacked models.

11.2 Human pose estimation without detection

Without any assumption about detection, we have no idea of the scale or the number of persons in the image. This can be also an opportunity: we can leverage key-point estimates to improve detections. We

have the advantage of a more realistic scenario, but the disadvantage of conflating detection and pose estimation.

There are two different approaches to human pose estimation without detection: top-down and bottom-up.

11.2.1 Top-Down pose estimation

The key idea of this approach is *detect people, then estimate their parts.*
In this case, it is possible to use any of the technique described in section 11.1, the problem is similar to instance segmentation, it is easy to get object level information, but hard to recover from bad detections.

An example of network to perform this task is **Mask R-CNN**, shown in figure 62 (see section 9.3).

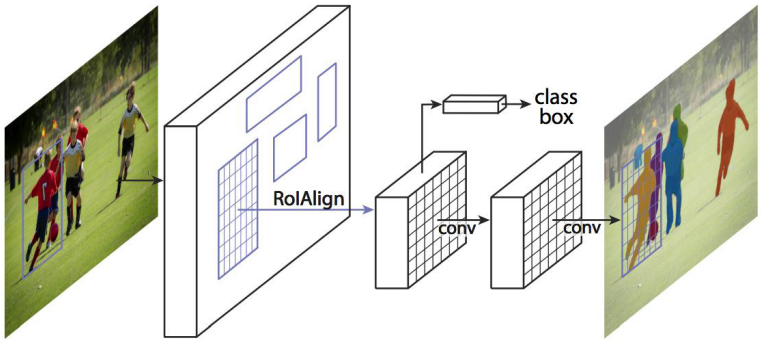

Figure 62: Mask R-CNN

11.2.2 Bottom-Up pose estimation

The key idea of this approach is *detect parts, then associate those to people.*
In this case, we need a way to group joints, which is a hard problem, due to its inherent ambiguity, and requires heuristics; moreover, there is no simple way to have object level information.

Realtime Multi Person 2D Pose Estimation using Part Affinity Fields is a technique that uses detection of limbs and limb orientation to guide the key-point grouping. The novelty introduced by this method is to jointly learn parts detection and parts association, through a single CNN.
Parts detection is based on *sequential prediction with learned spatial context*: stage by stage, each key-point is found, and finding the previous one helps finding the next one. For this reason, it is useful to find the easier parts first, then the harder ones.
Part-to-part and Part-to-person association is guided by the *affinity score*. Since the part affinity score is dependent on visual appearance, it can be encoded on the image plane: *part affinity fields* encode direction and position of body parts, which are used to determine the affinity score, which in turn helps to define the midpoint score map for part-to-part association and the part-to-part association itself, and to reduce spatial ambiguities. Part affinity fields are learned in parallel with parts detection by the second branch of the network, as shown in figure 63.

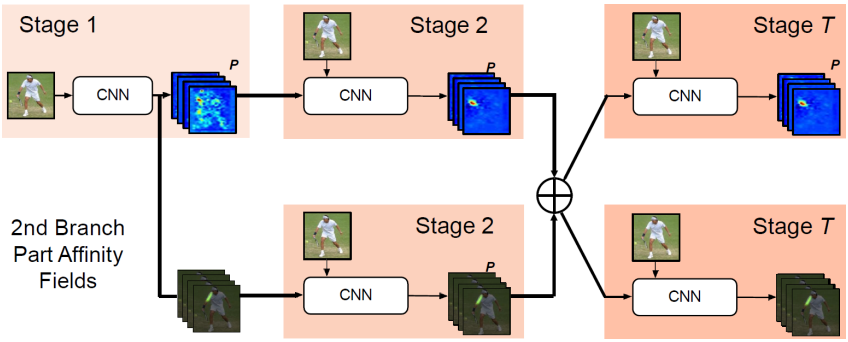

Figure 63: Pose Estimation using Part Affinity Fields (architecture

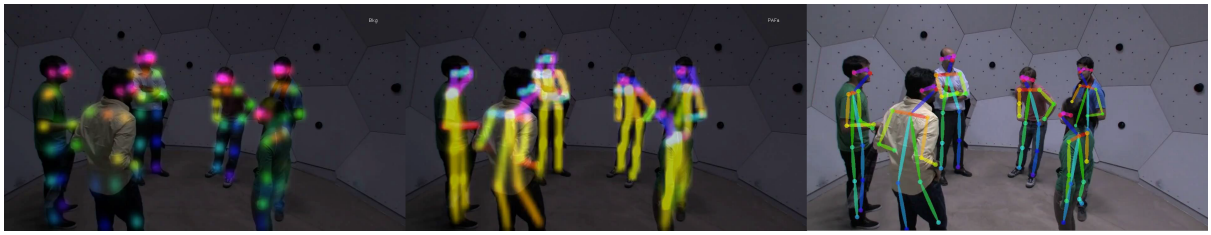

Figure 64: Pose Estimation using Part Affinity Fields (application example)

11.3 3D pose estimation of people and objects

To do pose estimation in 3D, we need to know relative lengths of each limb. The basic assumption is *scaled orthographic projection*, that is, we don't need to know the exact depth of each limb, we can approximate it *if* variation in depth is very smaller than depth, i.e., if the whole person is almost at the same distance from the camera. In math terms:

$$x = \frac{X}{Z} \approx \frac{X}{Z_0} \tag{84}$$

$$y = \frac{Y}{Z} \approx \frac{Y}{Z_0} \tag{85}$$

with Z_0 constant.
The relative depth between two points (X_1, Y_1, Z_1) and (X_2, Y_2, Z_2), called d_Z, is computed as follows:

$$l^2 = (X_1 - X_2)^2 + (Y_1 - Y_2)^2 + (Z_1 - Z_2)^2 \tag{86}$$

$$(u_1 - u_2) = s(X_1 - X_2) \tag{87}$$

$$(v_1 - v_2) = s(Y_1 - Y_2) \tag{88}$$

$$d_Z = (Z_1 - Z_2) = \sqrt{l^2 - \frac{(u_1 - u_2)^2 + (v_1 - v_2)^2}{s^2}} \tag{89}$$

Pose estimation for rigid objects is easier than for people, since they have fixed shape. The goal, in this case, is to understand position and orientation of the key-points of the objects (e.g. the wheels of a car). In other words, we're interested in capturing the "six degrees of freedom" of the object, that is, its X, Y, Z position, plus its roll (cyclo-rotation), its pitch (elevation) and its yaw (azimuth). To do that, we try to minimize the *reprojection error*, i.e., the distance between the ground truth and the key-points detected and reprojected on the plane. The best way to fit viewpoint to key-points would be to have canonical frames for each object (e.g. CAD models) which provide 3D location of each key-point, but too many models would be needed. Thus, we just use simplified models for classes of objects, or represent objects as combination of basic shapes (analogously to what we do with Fourier transforms for waves).

12 Sequence Modeling and Forecasting

So far, we have seen *one to one* ("vanilla") Neural Networks, but there exist different kinds of nets (see figure 65):

- *One to many*, e.g. Image Captioning, from image to sequence of words;
- *Many to one*: e.g. Sentiment Classification, from sequence of words to sentiment;
- *Sequence to sequence*: e.g. Machine Translation, from sequence of words to sequence of words;
- *Many to many*: e.g. Video Classification on frame level, from sequence of images to sequence of words

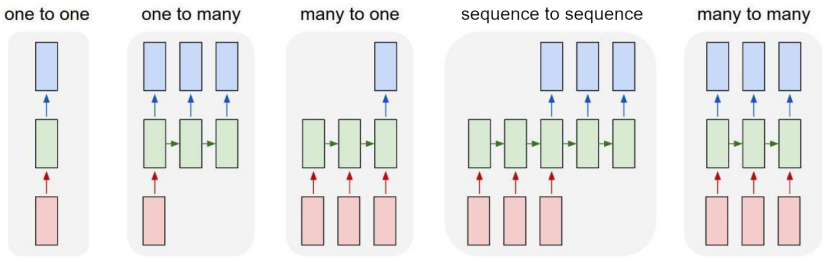

Figure 65: Different kinds of Neural Networks

It is also possible to apply sequential processing to non-sequence data for example to classify images by taking a series of "glimpses", or to generate images one piece at a time.

Moreover, sequences are particularly useful for forecasting. Trajectory forecasting is used in human robot interaction, long term tracking across occlusions, safety of autonomous vehicles. Other things to forecast with NNs are weather and natural disasters, market trends, critical pressure of a power plant, gas supply and demand.

There are many kinds of networks suitable for this task. We'll analyze them in the next sections.

12.1 Convolutional Neural Networks for Sequences

CNNs present some problems when applied to sequences:

- Fixed size and static input, but this issue can be resolved by concatenating more frames as input;
- The output is a single choice from a fixed list of options, however one may predict one word at a time or a fixed number of output words.

Temporal Convolutional Networks exploits CNNs' strengths to build a network with the following features:

- Convolutions are causal;
- Dilation: 12 layers may process a history of size 2^{12};
- Easy to train and exceeding expectations;
- Needs compromise on flexibility (e.g. variable number of input frames);
- Trivial to parallelize (per layer);
- Exploits local dependencies;
- "Interaction distance" between positions linear or logarithmic.

The main issue with this approach is that long distance dependencies require many layers.

12.2 Recurrent Neural Networks for Sequences

RNNs offer a lot of flexibility in all scenarios described above (see 12).
The key idea is that RNNs have an *internal state* that is updated as a sequence is processed. We can process a sequence of vectors x by applying a recurrence formula at every time step:

$$\underbrace{h_t}_{\text{new state}} = \underbrace{f_W}_{\substack{\text{some function} \\ \text{with parameters } W}} (\underbrace{h_{t-1}}_{\text{old state}}, \underbrace{x_t}_{\substack{\text{input vector} \\ \text{at time step } t}}) \tag{90}$$

Notice that the same function and the same set of parameters (weight matrix) are used at every time step.

12.2.1 RNN and LSTM models and their applications

The state of a simple RNN (also called *Vanilla RNN* or *Elman RNN*, see figure 66) consists of a single hidden vector h:

$$h_t = f_W(h_{t-1}, x_t) \tag{91}$$
$$\downarrow \tag{92}$$
$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t) \tag{93}$$
$$y_t = W_{hy}h_t \tag{94}$$

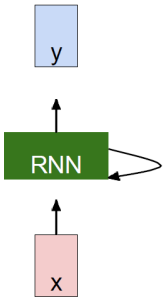

Figure 66: Simple RNN

In figures 67, 68, 69, 70 are shown the computational graphs for different kinds of RNNs. About the sequence-to-sequence RNN, it's worth to notice that it is the combination of a many-to-one RNN, which encodes the input sequence in a single output vector, and a one-to-many RNN, which produces an output sequence from the single input vector.

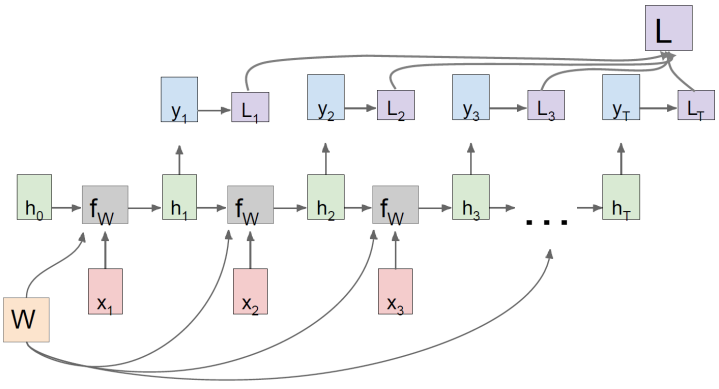

Figure 67: Many-to-many RNN

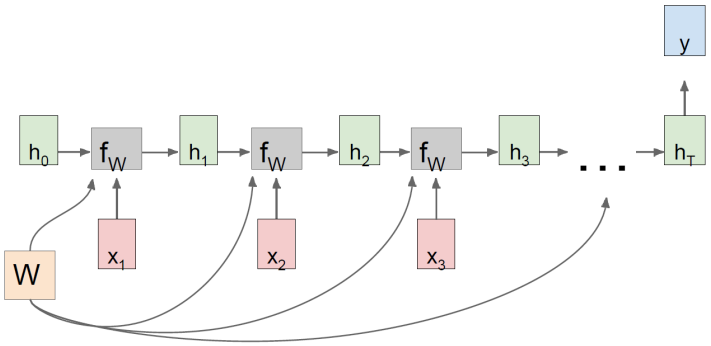

Figure 68: Many-to-one RNN

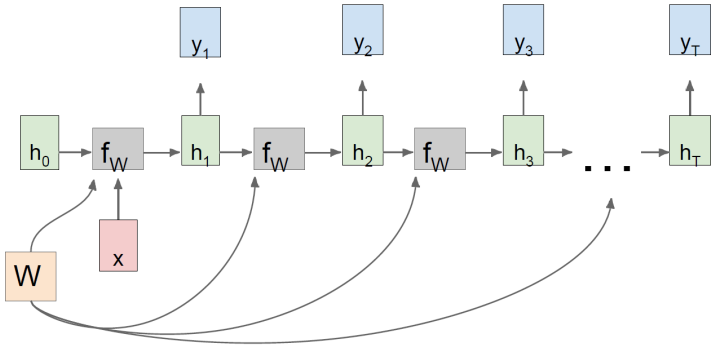

Figure 69: One-to-many RNN

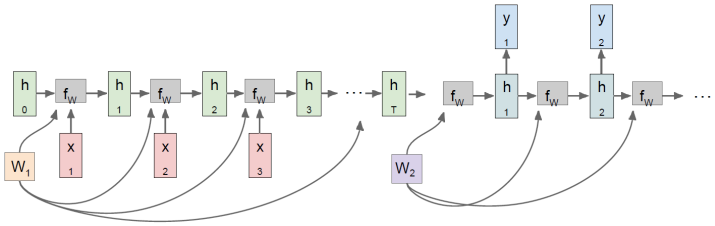

Figure 70: Sequence-to-sequence RNN

One interesting application of RNNs in NLP is to train **language models**, that is, to learn the probability for each work in the vocabulary to follow a given word (or set of words), based on a lot of sentences, for example from Wikipedia or from books:

$$\Pr(\text{next word}|\text{previous words}). \tag{95}$$

If we use a single word as *history* to predict next word, we can have a RNN like the one in figure 71, where we give in input 300 learnable numbers associated with each words in the dataset, then we have a hidden layer $h_4 = \tanh(0, W_{hx} \cdot x_4 + W_{hh} \cdot h_3)$, with 500-dimensional vectors, and finally the output vector $y_4 = W_{hy} \cdot h_4$ with 10,001-dimensional class scores produced by Softmax function over 10,000 words of the English vocabulary + 1 special `<END>` token.
We can use the trained language model to generate sentences, by sampling a word from each y_i in the output vector, until the `<END>` token.

As shown in figures 72 and 73, we can train and test a language model also at character level. At test time, it will be sufficient to sample characters one at a time, and feed them back to the model.

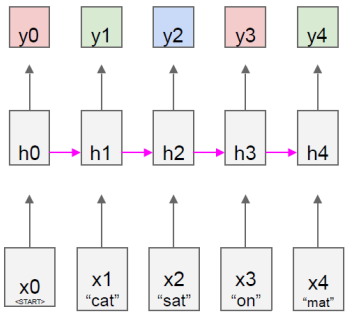

Figure 71: Example of RNN for training a language model.

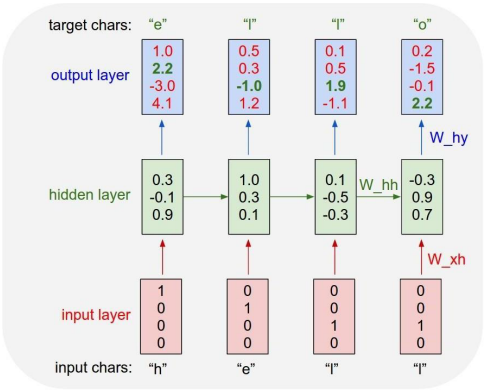

Figure 72: Training a char-level language model

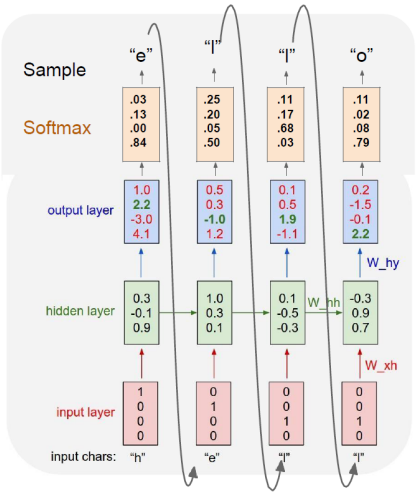

Figure 73: Sampling from a char-level language model

In RNNs, we have two different possibility to perform backpropagation (see figures 74 and 75):

1. *"Classic" backpropagation*: Forward through entire sequence to compute loss, then backward through entire sequence to compute gradient;
2. *Truncated backpropagation*: Run forward and backward through chunks of the sequence instead of whole sequence. In this way, we carry hidden states forward in time for the whole training, but only backpropagate for some smaller number of steps.

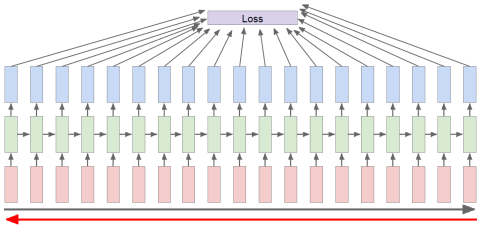

Figure 74: Classic Backpropagation in RNNs

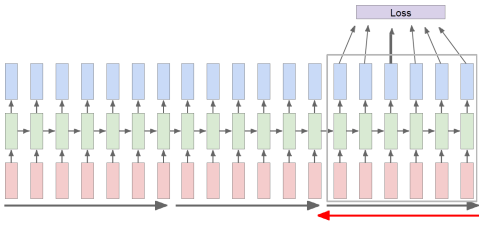

Figure 75: Truncated Backpropagation in RNNs

Possible applications of sentence generation are: generation of sonnets or code, image captioning, visual question answering. In some cases, it is possible to give an interpretation to cells to find, for example, quote detection cells, line length tracking cells, if statement cells, comment cells, code depth cells...

For **image captioning**, the input image is passed through a CNN, whose output layer (fully connected + softmax) is removed, and the result is given in input to a RNN, with hidden layer $h = \tanh(W_{xh} \cdot x + W_{hh} \cdot h + W_{ih} \cdot v)$, where $W_{ih} \cdot v$ comes from the CNN's output. For this task, as well as for **visual question answering**, better results can be obtained using **Attention mechanisms**. By weighting the features, the RNN focuses its attention at different spatial location when generating each word, or when answering to each question (see figures 76 and 77).

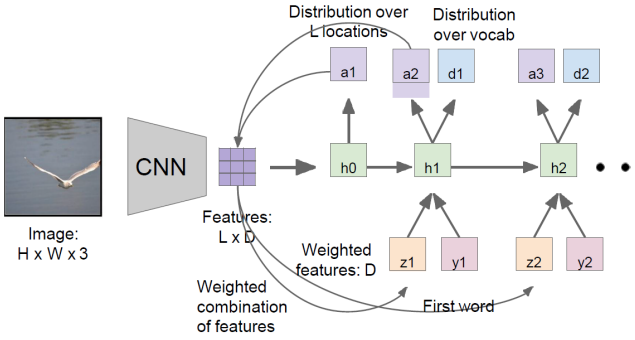

Figure 76: Image Captioning with Attention

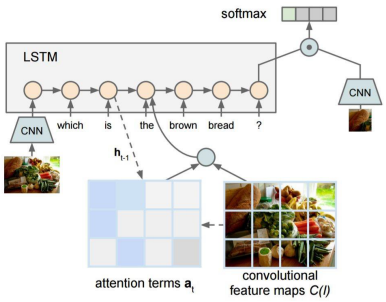

Figure 77: Question Answering with Attention

As we saw in equation 91, in a vanilla multilayer RNN, we have

$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t) \tag{96}$$

$$= \tanh\left((W_{hh}W_{hx})\begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}\right) \tag{97}$$

$$= \tanh\left(W\begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}\right) \tag{98}$$

So, the backpropagation from h_t to h_{t-1} multiplies by W. Thus, computing gradient of h_0 involves many factors of W and repeated tanh (see figure 78). This can lead to *exploding gradients* if the largest singular value is > 1, or to *vanishing gradients* if it is < 1. The first issue can be solved with **gradient clipping**, i.e., scaling gradient if its norm is too big; the second one require to change RNN architecture, and that's why LSTMs exist.

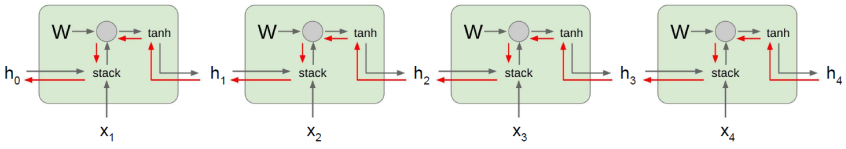

Figure 78: Backpropagation in vanilla RNNs

In **Long Short Term Memory** networks, the hidden layer becomes

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W\begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix} \tag{99}$$

$$c_t = f \cdot c_{t-1} + i \cdot g \tag{100}$$

$$h_t = o \cdot \tanh(c_t) \tag{101}$$

where i is the *input gate*, that decides whether to write to a cell, f is the *forget gate*, that decides whether to erase a cell, o is the *output gate*, that decided how much to reveal a cell, and g is the *gate gate*, that decides how much to write to a cell. Thus, the backpropagation from c_t to c_{t-1} is only an elementwise multiplication by f, without any matrix multiplication by W, so we can have an uninterrupted gradient flow from c_n to c_0, as shown in figure 79. Notice that this behavior is somehow similar to *ResNet*'s.

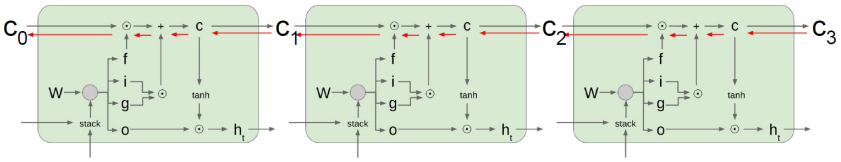

Figure 79: Backpropagation in LSTMs

Other RNN variants are **GRU** and **MUT1/2/3**. Let's see GRU's structure:

$$r_t = \sigma(W_{xr}x_t + W_{hr}h_{t-1} + b_r) \tag{102}$$
$$z_t = \sigma(W_{xz}x_t + W_{hz}h_{t-1} + b_z) \tag{103}$$
$$\tilde{h}_t = \tanh(W_{xh}x_t + W_{hh}(r_t \cdot h_{t-1}) + b_r) \tag{104}$$
$$h_t = z_t \cdot h_{t-1} + (1 - z_t) \cdot \tilde{h}_t \tag{105}$$

To conclude about RNN models, we have to say that better/simpler architectures are a hot topic of current research, as well as new paradigms for reasoning over sequences.

12.2.2 Modelling multiple agents and the scene

Now we'll spend some more words about **Trajectory Forecasting**, or *predicting unpredictable people*. The goal is to predict future positions, given previous positions.

We'll analyze this topic through its three key aspects:

- **Social interaction** (groups and collision avoidance): combine results from an LSTM for each person by using *social pooling* to exploit the knowledge that each individual is influenced by the surrounding people (see figure 80);
- **People's attention** (head pose):
  - *MX-LSTM* use people oriented velocity and head pose jointly to forecast people motion,
  - Based on sociological findings and data statistics,
  - The input of the LSTM is a combination of input position and head pose, optimized via log Choleski factorization,
  - Head pose can be applied to social pooling too;
- **The context** (scene): the scene, usually encoded by a CNN, constraints the plausible motion.

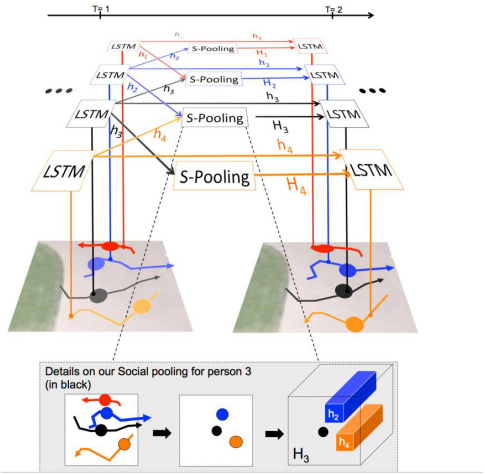

Figure 80: Social pooling in trajectory forecasting

12.3 Transformer Networks for Sequences

As we saw in previous sections, to learn representations of variable length data, essential for sequence-to-sequence learning, we can use LSTMs or GRUs, but they present some issues:

- They use *sequential computation* to transfer knowledge from the first elements of a sequence to the next, but this approach inhibits parallelization;
- They don't have any explicit modeling of long and short range dependencies;
- We want model hierarchy.

On the other hand, CNNs have many advantages (such as easy per layer parallelizability, exploitation of local dependencies, *interaction distance* between positions linear or logarithmic), but their long-distance dependencies require many layers.

Since we know that attention between encoder and decoder is crucial in *Neural Machine Translation*, we can apply the same idea for representations: **Self-Attention** introduces interesting improvements over RNNs and CNNs (see also figure 81):

- Constant path length between any two positions;
- Gating/multiplicative interactions;
- Trivial to parallelize (per layer).

Layer Type	Complexity per Layer	Sequential Operations	Maximum Path Length
Self-Attention	$O(n^2 \cdot d)$	$O(1)$	$O(1)$
Recurrent	$O(n \cdot d^2)$	$O(n)$	$O(n)$
Convolutional	$O(k \cdot n \cdot d^2)$	$O(1)$	$O(log_k(n))$
Self-Attention (restricted)	$O(r \cdot n \cdot d)$	$O(1)$	$O(n/r)$

Figure 81: Self Attention VS RNNs VS CNNs

The **Transformer** is a very successful encoder-decoder network that use this approach. In figures 82 and 83 is shown its architecture at higher and lower level: the last encoder sends the representations of the input words it computed to all the decoders, thus giving them a *memory*.

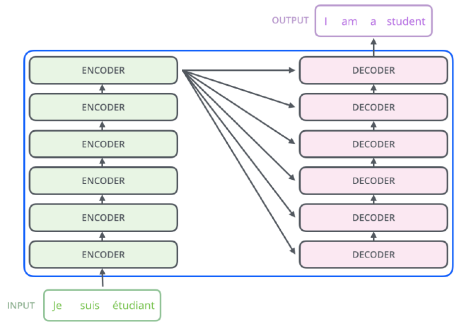

Figure 82: Trasnsformer architecture (a)

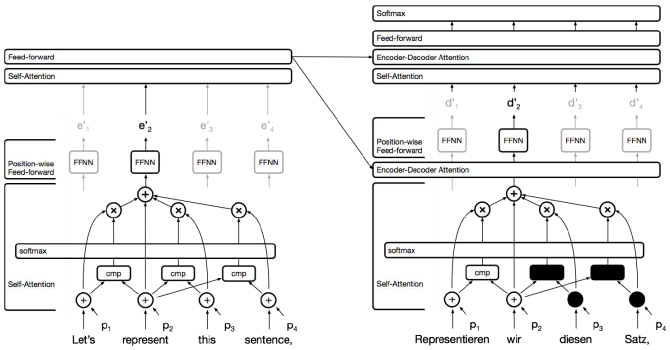

Figure 83: Transformer architecture (b)

Attention value is a function of Q (the query), K (the key) and V (the value):

$$A(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{106}$$

That is, A uses softmax to give a weighted average of probabilities, where weights are given by the inputs, so that the output of a certain word depends mostly on words to which it is more tightly related, rather than closer ones. So, this approach is also invariant to word order. With *multi-head attention*, parallel

attention layers with different linear transformations on input and output are applied to each word.
It is important to notice that the self-attention blocks are different in encoders and decoders, as shown in figures 84 and 85: in the encoders, the output originated from a word, depend on previous and next words, while in decoders, next words are masked, so that only previous words are considered; furthermore, in decoders, keys and values come from the last encoder, while queries from the previous decoder.

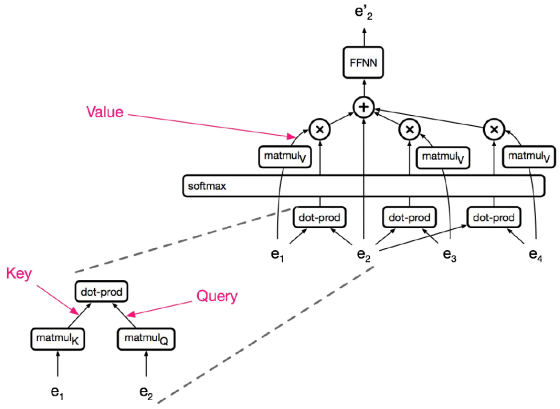

Figure 84: Encoder Self-Attention

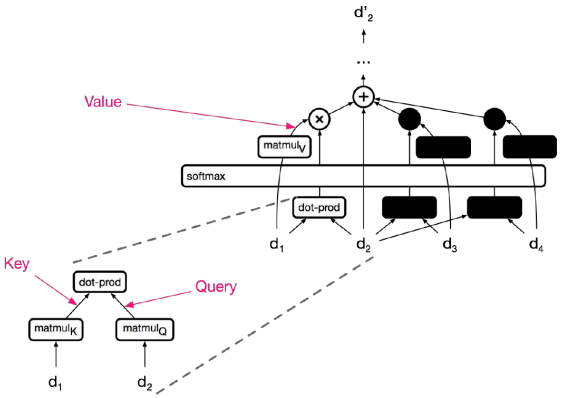

Figure 85: Decoder Self Attention

The Transformer works in steps, described in figures 86 and 87:

- *Step 1*: compute K and V from the input;
- *Step 2*: combine Q from the output with K and V, autoregressively.

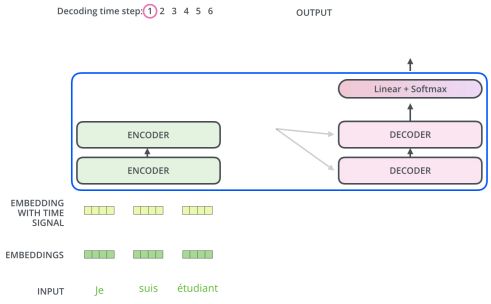

Figure 86: Transformer - step 1

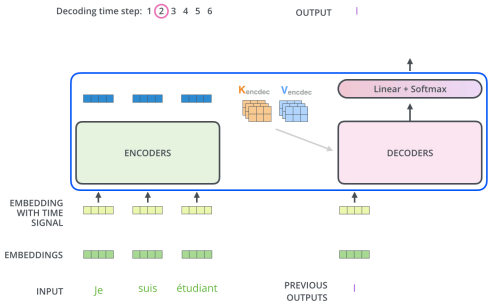

Figure 87: Trasnformer - step 2

As we said before, self-attention mechanism is permutation invariant to word order. But, if the ordering is relevant, the relative position of each word can be encoded in the input (see figure 88). It's important to note that Residuals (a structure similar to the one used in ResNet, shown in figure 89) are fundamental for position representation, since they carry positional information to higher layers, among other information.

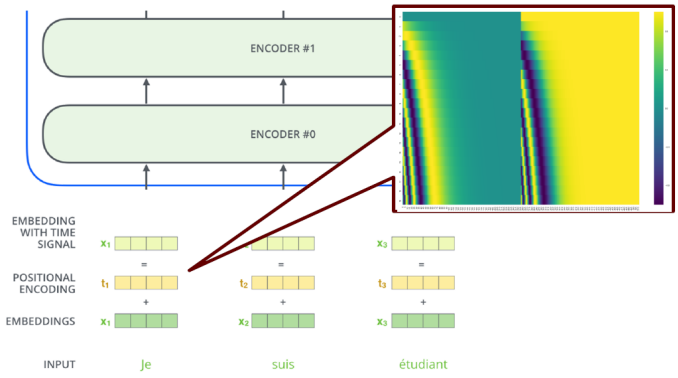

Figure 88: Transformer with positional encoding

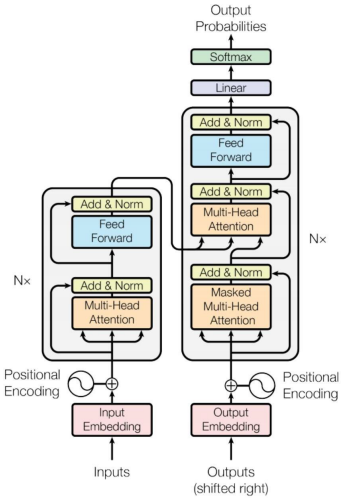

Figure 89: Transformer architecture (c)

Training details for Transformers:

- ADAM optimizer with a learning rate warmup (warmup + exponential decay);
- Dropout during training at every layer just before adding residual;
- Layer norm;
- Attention dropout;
- Checkpoint averaging;
- Label smoothing;
- Auto-regressive decoding with beam search and length biasing.

12.3.1 Self-Similarity, Image and Music Generation

Self-similarity is a property of images and music, for which portions (*motifs*) repeat immediately or at a distance. This property can be exploited for **non-local means**: the attention window is restricted to local neighborhoods (good assumption thanks to spatial locality), but more relevance is given to most similar pixels, via a weighted average. This leads to the creation of **Image Transformers**, where positional encoding is based on pixel coordinates. **Relative attention** provides expressive timing, equivariance, and extends naturally to graphs, by combining multi-head attention and convolution:

$$RA = softmax(QK^T + Qf(E_{rel})) \tag{107}$$

Examples of application of these principles are:

- *Probabilistic image generation*:
  - Model the joint distribution of pixels;
  - Turn it into a sequence modeling problem
  - Assigning probabilities allows measuring generalization;
  - CNNs with gating match RNNs in quality, but are much faster due to parallelization;
  - Modeling long range dependencies (important for big images) with CNNs requires either many layers (harder training) or large kernels (large parameter/computational cost).
- *Texture synthesis*: analyze similarities to reproduce a pattern;
- *Denoising*: exploit similarities among close pixels to remove noise from an image;

- *Super-Resolution*: enhance the resolution of an image;
- *Unconditional and Conditional Image generation*;
- *Text to speech*;
- *Pentagram to music*;
- *Music models* to continue a given initial motif.

12.3.2 Pre-trained Language Models

Language models only use left context or right context, for two reasons:

1. Directionality is needed to generate a well formed probability distribution;
2. Words could "see themselves" in a bidirectional encoder.

But language understanding is bidirectional. To solve this problem, we can use **Masked Language Models**: mask out $k\%$ of the input words, and then predict the masked words. As usual, we need to find a trade-off for the size of k: if it is too little, the model is too expensive to train, if it is too big, we don't have enough context.
This approach introduces a problem: the `mask` token is never seen at fine-tuning time. A possible solution is to select $k\%$ of the words to predict, but don't replace always with `mask` token, instead:

- 80% of the time, replace with `mask` token,
- 10% of the time, replace with a random word,
- 10% of the time, keep same.

Next Sentence Prediction: to learn relationships between sentences, predict whether Sentence B is actual sentence that proceeds Sentence A, or a random sentence. The input is represented by tokens that are sum of three embeddings: token embedding (which represents the word in the sentence), segment embedding (which says if a word belongs to A or B) and position embedding (which describe the position of the word in the sentences). Single sequence is much more efficient.

BERT is a *Bidirectional transformer* that can be pretrained on couples of sentences, and fine-tuned to perform different tasks, such as question answering, sentence classification or sentence tagging (see figure 90). SQuAD is the model for question-answering fine-tuning, it uses token 0 (`CLS`) to emit logit for "no answer", which directly compete with answer span.

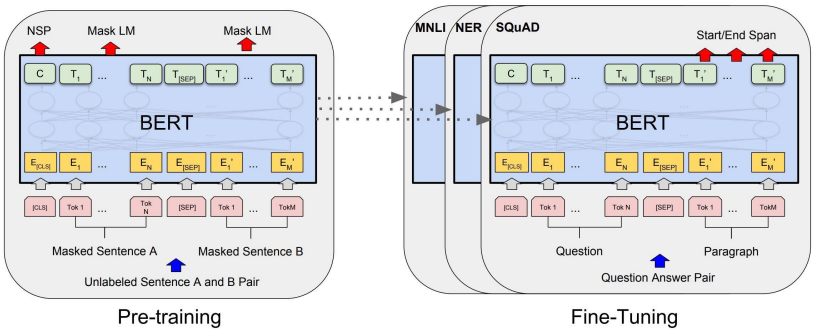

Figure 90: BERT

Observations:

- Masked LM (compared to left-to-right LM) is very important on some tasks, Next Sentence Prediction is important on other tasks;
- Left-to-right model does very poorly on word level task (SQuAD), although this is mitigated by BiLSTM;

- Masked LM takes slightly longer to converge because we only predict $k\%$ instead of 100%, but absolute results are much better almost immediately;
- Big models help a lot.

RoBERTa, is a *Robustly Optimized BERT Pretraining Approach*: trained BERT for more epochs and/or on more data, each combination is useful.

ELECTRA, *Pre-training Text Encoders as Discriminators Rather Than Generators*, trains model to discriminate locally plausible text from real text.

These models are very accurate, but too large and expensive. Thus, to apply them to low-latency production services, companies apply **distillation**, or *model compression*:

- Train *teacher*: use a big dataset and fine-tuning technique to train a model with maximum accuracy;
- Label a large amount of unlabeled input examples with Teacher;
- Train *Student*: much smaller model which is trained to mimic Teacher output;
- Student's objective is typically Mean Square Error or Cross Entropy.

Distillation works much better than pre training + fine tuning with smaller model.

13 Multi-Task and Meta Learning

To study how we can enable agents to learn skills in the real world, let's consider robots: they can teach us things about intelligence, because they faced with the real world , they must generalize across tasks, objects and environments, they need some common sense understanding, they can't rely on supervision. Nowadays, robots/agents can learn very well a single task in a single environment, starting from scratch, relying on detailed supervision and guidance, moreover, often the researcher has to do lot of work to train the robot.

Deep learning allows us to handle unstructured inputs (pixels, language...) without hand engineering features, with less domain knowledge, but we need huge diverse data to generalize. For some tasks, by the way, we don't have large datasets, in other cases it is impractical to learn from scratch for each specific task, in still other cases, the dataset is unbalanced (*long tail*: few classes with lots of data-points), moreover, sometimes we need to quickly learn something new. In all these cases, standard ML paradigms don't work.

Multi-task learning and meta-learning can overcome these problems. A *task* is identified by a dataset, a loss function, and a parameterized model f_θ. To apply these strategies, we need that the tasks we consider share some structure, but this is often the case.

Informal problem definitions:

- **The multi-task learning problem**: Learn all of the tasks more quickly or more proficiently than learning them independently.
- **The meta-learning problem**: Given data/experience on previous tasks, learn a new task more quickly and/or more proficiently.

Furthermore, multi-task and meta-learning are critical for the *democratization* of AI: research datasets are huge, while real-world datasets are much smaller.

13.1 Multi-Task Learning

Multi-task learning can reduce to single task learning, by aggregating the data across tasks and learning a single model, but we can do better if we exploit the fact that we know that data is coming from different tasks.

Formally, in single-task supervised learning a task is defined by the dataset $\mathcal{D} = \{(x, y)_k\}$ and the objective $\min_\theta \mathcal{L}(\theta, \mathcal{D})$, where usually the loss is negative log likelihood $\mathcal{L}(\theta, \mathcal{D}) = -\mathbb{E}_{(x,y)\sim\mathcal{D}}\left[\log_{f_\theta}(y|x)\right]$. On the other hand, a task in a multi-task learning context is defined as a data generating distribution $\mathcal{T}_i := \{p_i(x), p_i(y|x), \mathcal{L}_i\}$, with corresponding datasets $\mathcal{D}_i^{tr}$ and $\mathcal{D}_i^{test}$. Example of multi-task learning are:

- *multi-task classification*, such as per-language handwriting recognition, where $\mathcal{L}_i$ is the same across all tasks;
- *multi-label learning*, such as scene understanding, where $\mathcal{L}_i$ and $p_i(x)$ are the same across all tasks.

When you care more about one task than another, also $\mathcal{L}_i$ varies.

We can add a parameter z_i to represent the *task descriptor* (for example with the task index), so that the function learned by the model is $f_\theta(y|x, z_i)$ instead of $f_\theta(y|x)$. Now we must find a way to optimize the objective

$$\min_\theta \sum_{i=1}^{T} \mathcal{L}_i(\theta, \mathcal{D}_i). \tag{108}$$

In other words, we have to find a way to condition on z_i, in order to share as little as possible. A solution is independent training within a single network, with no shared parameters, and multiplicative gating

to compute the output: $y = \sum_j \mathbb{1}(z_i = j) y_j$ (see figure 91). The opposite solution is to concatenate z_i with input and/or activations, so that all parameters are shared, except those directly following z_i (see figure 92).

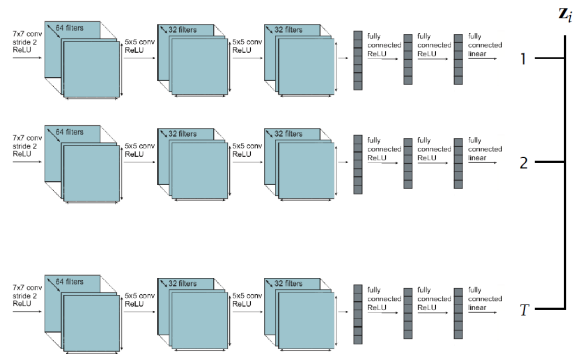

Figure 91: Solution 1

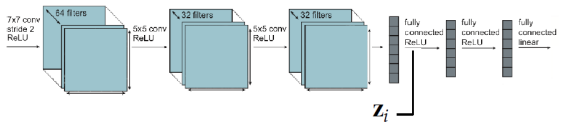

Figure 92: Solution 2

An alternative view on the multi-task objective is to split the parameter θ into shared parameters θ^{sh} and task specific parameters θ^i. Then, our objective becomes

$$\min_{\theta^{sh}, \theta^1, \dots, \theta^T} \sum_{i=1}^{T} \mathcal{L}_i \left(\left\{ \theta^{sh}, \theta^i \right\}, \mathcal{D}_i \right). \tag{109}$$

In this way, choosing how to condition on z_i is equivalent to choosing how and where to share parameters:

1. Concatenation-based conditioning (see figure 93);
2. Additive conditioning (see figure 94);
3. Multi-head architecture (see figure 96);
4. Multiplicative conditioning (see figure 97), good because more expressive, and able to generalize independent networks and independent heads.

Notice that solutions 1 and 2 are equivalent, as shown in figure 95.
Unfortunately, these design decisions are like neural network architecture tuning: problem dependent and largely guided by intuition or knowledge of the problem.

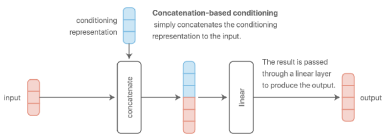

Figure 93: Concatenation-based conditioning

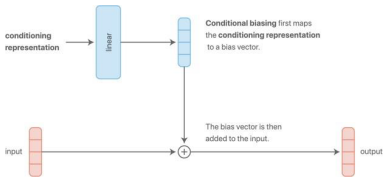

Figure 94: Additive conditioning

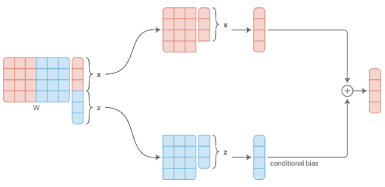

Figure 95: Concatenation ≡ Additive

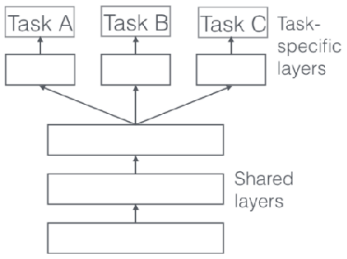

Figure 96: Multi-head architecture

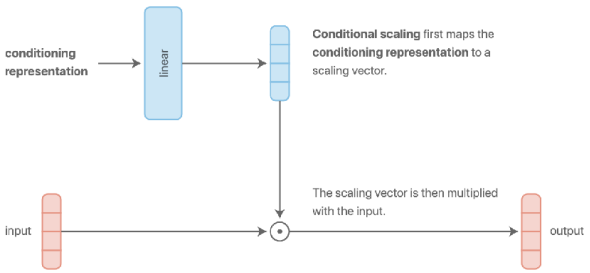

Figure 97: Multiplicative conditioning

To optimize the objective in equation 108, we have to follow the steps below:

1. Sample mini-batch of tasks $\mathcal{B} \sim \{\mathcal{T}_i\}$;
2. Sample mini-batch datapoints for each task $\mathcal{D}_i^b \sim \mathcal{D}_i$;
3. Compute loss on the mini-batch $\hat{\mathcal{L}}(\theta, \mathcal{B}) = \sum_{\mathcal{T}_k \in \mathcal{B}} \mathcal{L}_k(\theta, \mathcal{D}_k^b)$;
4. Backpropagate loss to compute gradient $\nabla_\theta \hat{\mathcal{L}}$;
5. Apply gradient with a neural network optimizer.

This approach ensures that tasks are sampled uniformly, regardless of data quantities. For regression problems, we must make sure that task labels are on the same scale.

Two main problems may arise in multi-task learning. The first one is *negative transfer*: sometimes independent networks work the best. This may happen for different reasons: optimization challenges caused by cross-task interference or tasks that learn at different rates, or by limited representational capacity (multi task networks often need to be much larger than their single task counterparts). The solution is to share less across tasks (i.e., *soft parameter sharing*):

$$\min_{\theta^{sh}, \theta^1, \ldots, \theta^T} \sum_{i=1}^{T} \mathcal{L}_i \left(\left\{ \theta^{sh}, \theta^i \right\}, \mathcal{D}_i \right) + \sum_{t'=1}^{T} \left\| \theta^t - \theta^{t'} \right\|. \tag{110}$$

This allows for more fluid degrees of parameter sharing, but adds more design decisions.
The second problem is overfitting. It may be due to too little sharing among tasks. In this sense, we can look at multi-task learning as a form of regularization. The solution is to share more.

13.2 Meta-Learning

We would like for the machines to be able to understand, generalize and learn from few similar - not identical - examples as human are, but they aren't, even though they are getting better at it, for example with *human-level concept learning through probabilistic program induction.*

The main problem we have to deal with in this case, is **few-shot learning**: we want to design a learning algorithm A that outputs good parameters θ of a model M, when fed a small dataset $\mathcal{D}^{tr} = \{x_i, y_i\}_{i=1}^{L}$. The idea is to learn A end-to-end. This approach is called *meta-learning*, or *learning to learn.*

Related works are:

- *Transfer learning*: large image datasets have been shown to allow training representations that transfer to other problems, while, in few shot learning, we aim at transferring the complete training of the model on new datasets (not just transferring the features or initialization), and ideally there should be no human involved in producing a model for new datasets;
- *One-shot learning*: old studies largely relied on hand-engineered features, but with recent progress in end-to-end deep learning, we hope to learn a representation better suited for few-shot learning;

- *Neural Architecture Search*: techniques based on Bayesian optimization and reinforcement learning to look for the best NN hyperparameters.

But, if you don't evaluate on never-seen problems/datasets, it's not meta-learning.

Let's define a learning algorithm A as:

- *input*: training set $\mathcal{D}^{\text{tr}} = (x_i, y_i)$,
- *output*: parameters θ for the model M (the *learner*),
- *objective*: good performance on the test set $\mathcal{D}^{\text{test}} = (x'_i, y'_i)$;

then, a meta-learning algorithm can be defined as:

- *input*: meta-training set $\mathcal{D}^{\text{meta-train}} = \left\{ \left(D^{(n)}_{\text{train}}, D^{(n)}_{\text{test}} \right) \right\}_{n=1}^{N}$ of episodes,
- *output*: parameters θ of an algorithm A (the *meta-learner*), to learn model M,
- *objective*: good performance on the meta-test set $\mathcal{D}^{\text{meta-test}} = \left\{ \left(D'^{(n)}_{\text{train}}, D'^{(n)}_{\text{test}} \right) \right\}_{n=1}^{N'}$.

Thus, a couple of (training set, test set), that are used to train a *learner*, is an *episode*, and a set of episodes composes a meta-training set, which is used to train a *meta-learner*. Someone calls the training set "support set" and the test set "query set", in this case, the meta-training set and the meta-test set can be called simply training test and test set, respectively. The approach is shown in figure 98.

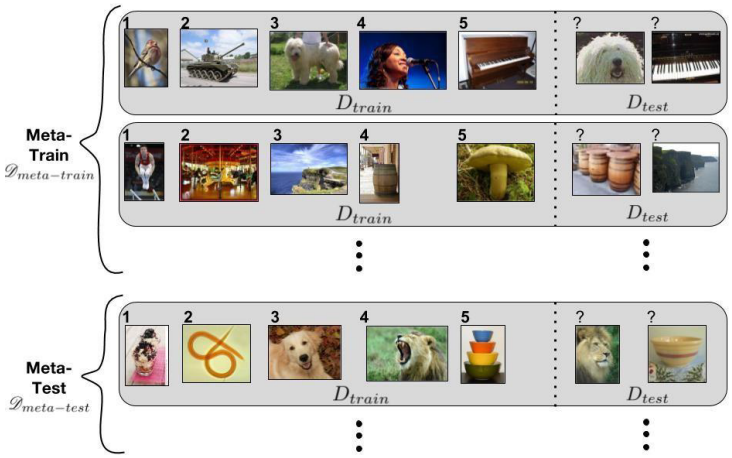

Figure 98: Meta-learning

To evaluate few-shot image recognition, a useful dataset is *Omniglot*, a transpose of MNIST for meta-learning tasks, with many classes and few examples, from 50 different alphabets. Other datasets used for this task are *MiniImagenet, CIFAR, CUB, CelebA*. A possible evaluation strategy is *5-way, 1-shot image classification*: given 1 example of 5 classes, classify new examples.

13.2.1 Optimization-based Techniques

The key idea is to acquire ϕ_i through optimization of

$$\max_{\phi_i} \log p(\mathcal{D}_i^{tr}|\phi_i) + \log p(\phi_i|\theta). \tag{111}$$

The meta-parameters θ serve as a prior, e.g.: initialization for *fine-tuning*. In this case, for many gradient steps, during training we fine-tune in this way:

$$\phi \leftarrow \theta - \alpha \nabla_\theta \mathcal{L}(\theta, \mathcal{D}^{tr}), \tag{112}$$

where θ are pre-trained parameters from a similar task with a big datasets and $\mathcal{D}^{tr}$ training data for the new task.

Some common practices:

- Fine tune with a smaller learning rate;
- Lower learning rate for lower layers;
- Freeze earlier layers, gradually unfreeze;
- Reinitialize last layer;
- Search over hyperparameters via cross validation;
- Architecture choices matter.

Notice that fine-tuning is less effective with very small datasets.

At test time, apply meta-learning by optimizing this objective:

$$\min_{\theta} \sum_{\text{task } i} \mathcal{L}\left(\theta - \alpha \nabla_{\theta} \mathcal{L}(\theta, \mathcal{D}_i^{tr}), \mathcal{D}_i^{test}\right) \tag{113}$$

where θ is the parameter vector being meta-learned. The key idea is to learn parameter vector θ that transfers via fine tuning, over many tasks (see figure 99, where ϕ_i^* is the optimal parameter vector for task i).

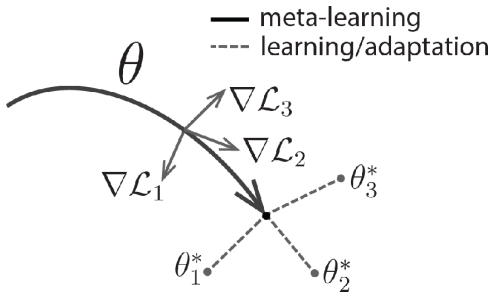

Figure 99: Optimization-Based Inference

Thus, the optimization-based approach works like this:

1. Sample task $\mathcal{T}_i$ (or mini batch of tasks);
2. Sample disjoint datasets $\mathcal{D}_i^{tr}, \mathcal{D}_i^{test}$ from $\mathcal{D}_i$;
3. Optimize equation 112;
4. Update θ using $\nabla_{\theta}\mathcal{L}\left(\phi_i, \mathcal{D}_i^{test}\right)$;
5. Restart from step 1.

This brings up second-order derivatives (the Hessian), which isn't explicitly computed. No higher order derivatives are needed for more gradient descent steps.

13.2.2 Non-parametric Techniques

The key idea is to use a *non-parametric learner*: given a training set $\mathcal{D}_i^{tr}$ and a test data-point x^{test}, compare test image with training images. Pixel space or L2 distance aren't appropriate, so we learn to compare using meta-training data: train a Siamese network to predict whether two images belong to the same class (meta-training with binary classification), then test comparing image x^{test} to each image in $\mathcal{D}_j^{tr}$ (meta-test with N-way classification).

It is possible to combine meta-train and meta-test in a unique network trained end-to-end in order to find nearest neighbors in the learned embedding space (see figure 100). The network output is $\hat{y} = \sum_{i=1}^{k} a(\hat{x}, x_i) y_i$, where $a(\hat{x}, x_i) = \frac{e^{c(f(\hat{x}), g(x_i))}}{\sum_{j=1}^{k} e^{c(f(\hat{x}), g(x_j))}}$.

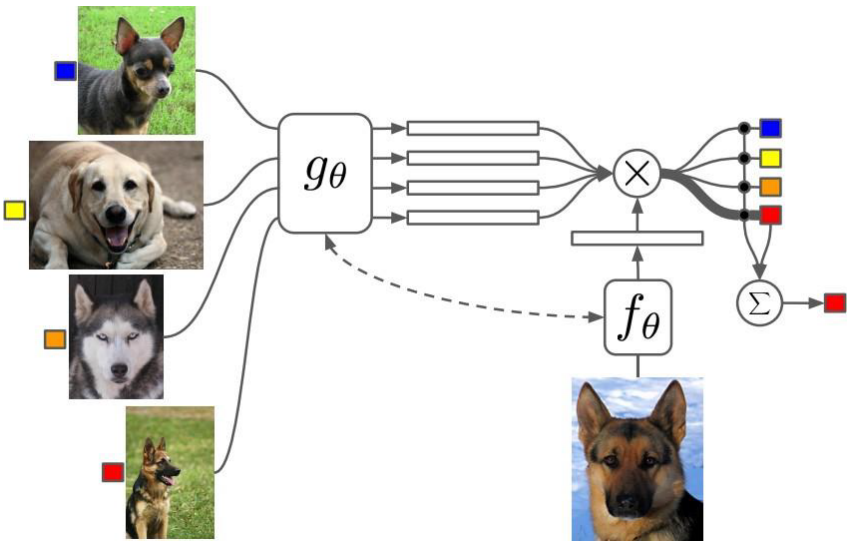

Figure 100: Matching Networks

An alternative is to train a **prototype extractor** (see figure 101):

$$p_\phi(y = k|x) = \frac{\exp(-d(f_\phi(x), c_k))}{\sum_{k'} \exp(-d(f_\phi(x), c_{k'})} \tag{114}$$

$$d = \text{Euclidean or Cosine distance} \tag{115}$$

$$c_k = \frac{1}{|S_k|} \sum_{(x_i, y_i) \in S_k} f_\phi(x_i) \tag{116}$$

$$S_k = \left\{(x_i, y_i) | y_i = k, (x_i, y_i) \in \mathcal{D}^{tr}\right\} \tag{117}$$

$$\phi \equiv \Theta \tag{118}$$

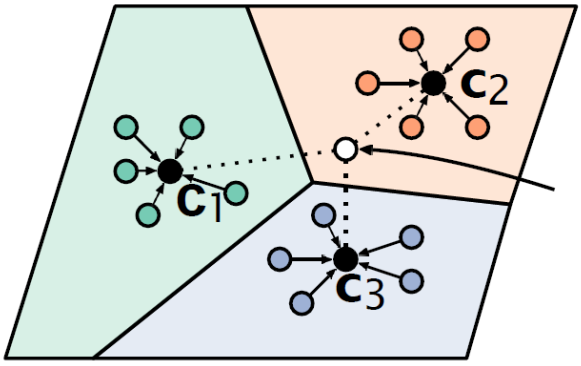

Figure 101: Prototypical Networks