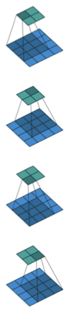
Lesson Notes (28/03 - 26/04)

Lesson 09: Backpropagation for CNNs, Transposed Convolutions, Dilated Convolutions, Normalization Layers

@March 28, 2023

Backpropagation for CNNs

Try to understand convolution as a single dense linear layer. We can have a 4x4 input, a 3x3 kernel with stride 1 and an output 2x2. Now imagine to linearize the input (flatten it into a single vector of size 16) and four outputs.



$$\begin{pmatrix} w_{0,0} & 0 & 0 & 0 \\ w_{0,1} & w_{0,0} & 0 & 0 \\ w_{0,2} & w_{0,1} & 0 & 0 \\ 0 & w_{0,2} & 0 & 0 \\ w_{1,0} & 0 & w_{0,0} & 0 \\ w_{1,1} & w_{1,0} & w_{0,1} & w_{0,0} \\ w_{1,2} & w_{1,1} & w_{0,2} & w_{0,1} \\ 0 & w_{1,2} & 0 & w_{0,2} \\ w_{2,0} & 0 & w_{1,0} & 0 \\ w_{2,1} & w_{2,0} & w_{1,1} & w_{1,0} \\ w_{2,2} & w_{2,1} & w_{1,2} & w_{1,1} \\ 0 & w_{2,2} & 0 & w_{1,2} \\ 0 & 0 & w_{2,0} & 0 \\ 0 & 0 & w_{2,1} & w_{2,0} \\ 0 & 0 & w_{2,2} & w_{2,1} \\ 0 & 0 & 0 & w_{2,2} \end{pmatrix}^T$$

Each column will correspond to a different application of the kernel, with $w_{i,j}$ being the kernel weight applied by the i^{th} row and j^{th} column of the kernel respectively.

Sparse matrix (we have a lot of 0s)

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

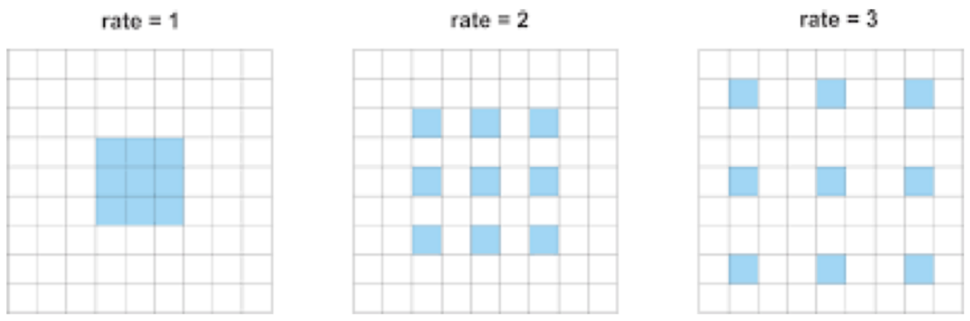
Transposed Convolutions

In some situations we might want to upsample the input (increase the dimension), for example when doing image to image processing, to obtain an image of the same dimension of the input (or higher) after some compression to an internal encoding; or when projecting feature maps to a higher-dimensional space. One could use upsampling layers (mathematical upsampling, still need to ensure derivability). Essentially you can think it as a normal convolution with sub-unitarian stride ($0 < x < 1$). To mimic this, we need to enlarge artificially the input (insert padding between the pixels - usually zero-padding) and then apply a single-strided convolution.

In another way, we could simply transpose the convolution matrix. This means that the kernel defines a convolution matrix, but which one (direct or transposed) is determined by how it is applied.

Dilated (a.k.a Atrous) Convolutions

Dilated Convolutions are “convolutions with holes”.



It enlarges the receptive fields, keeping a low number of parameters. Useful in first layers, when working on high resolution images. Used in TCNs (Temporal Convolutional Networks) to process long input sequences.

Normalization Layers

The benefits behind the introduction of normalization layers are: more stable and possibly faster training and an increased independence between layers.

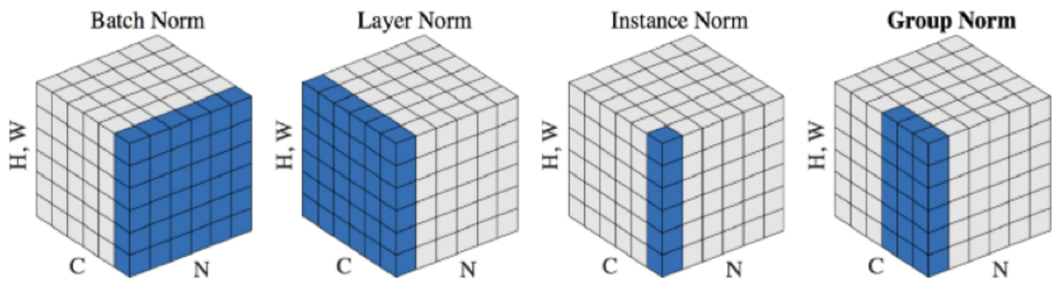
Batch Normalization

Batch Normalization operates on single layers, per channel base. At each training iteration, the input is normalized according to batch (moving) statistics, subtracting the mean μ^B and dividing by the standard deviation σ^B. Then, an opposite transformation is applied based on learned parameters γ (scale) and β (center)

$$BN(x) = \gamma \cdot \frac{x - \mu^B}{\sigma^B} + \beta$$

Batch statistics are moving (we keep running values and slowly update them based on the current mini-batch). Batch normalization behaves differently in training and prediction mode. Typically, after training, we use the entire dataset to compute a stable estimates of the statistics and use them at prediction time (statistics over a given training set don't change anymore once training is concluded)

Other Forms



N is the Batch Axis, C is the Channel Axis, H and W are the Spatial Axes. Pixels in blue are normalized by the same mean and variance, computed by aggregating the values of these pixels.

Lesson 10: Gradient Ascent on Input

@March 29, 2023

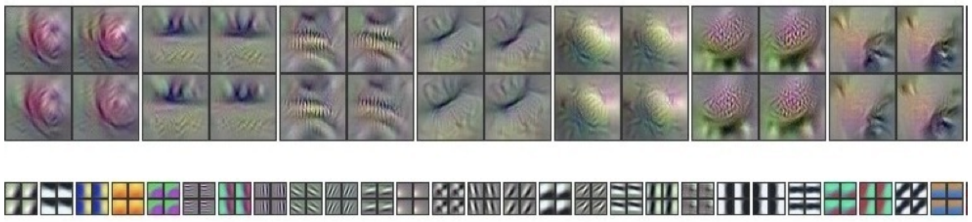
Each neuron in a neural network gets activated by specific patterns in the input image, defined by the weights in its receptive field. The intuition is that neurons at higher layers should recognize increasingly complex patterns, obtained as a combination of previous patterns over a larger receptive fields. So, in the highest layers, neurons may start recognizing patterns similar to features of objects in the dataset. In the final layers, neurons get activated by "patterns".

When we compute the gradient, we consider the loss function; the function depends on the parameters (configuration of the model) and the input $\mathcal{L}(\theta, x)$. When we train the Neural Network, the input is fixed (mini-batch $\Rightarrow$ approximation that works). We could also have a given network and we try to explore the behavior of the network. In this case the set of parameters is fixed and we can use the same computation as before, but compute the gradient of the loss function w.r.t. input (how should I modify my input to obtain a determined value from my network = maximize the activation of a neuron).

Slight modification of the input to ensure strong activation. Called "Gradient Ascent" → we move in the opposite direction of the gradient. Here we want to optimize log likelihood.

So, we start from a random image x and:

- do a forward pass using the image x as input to the network to compute the activation $a_i(x)$ caused by x at some neuron (or the whole layer)
- do a backward pass to compute the gradient $\delta a_i(x)/\delta x$ of $a_i(x)$ with respect to each pixel of the input image
- we modify the image adding a small percentage of the gradient and repeat the process until we get a sufficiently high activation of the neuron


Patterns generated from the first two layers of AlexNet. Note that they could change when generated in different instances because of randomness.

First features are very simple, and get increasingly more complex at higher level, as the receptive field increases due to nested convolutions. The patterns show what we're gonna capture, what we're reacting to (can find interesting patterns or not).

The same technique can be applied to a layer e/o complete network. But also, one could try to understand. We have an input image: what of this input image is recognized by the network, aka what information from the input image is considered relevant by the neural network? Therefore we aim to synthesize an image that is undistinguishable from the original. (DeConv, 2014)

The goal is, given an input image x_0 with an internal representation $\Theta_0 = \Theta(x_0)$, generate a different image x s.t. its internal representation $\Theta(x) = \Theta_0$. The approach is minimizing the distance from Θ_0 starting from a noise image.

$$\arg\min l(\Theta(x), \Theta_0) + \lambda \mathcal{R}(x)$$

It is simpler to invert from the first layer, as we've lost less information. The deeper we go, the more information we lose (internal representation becomes progressively complicated). Deeper layers invert back to multiple copies of input's parts at different positions and scales (more abstract).

Inceptionism

Creative form of image manipulation that starts from an input image and injects information (activates different layers through gradient ascent). The approach is:

- Train a network for image classification
- revert the network to slightly adjust via backpropagation the original image in order to improve activation of a specific neuron

After enough reiterations, even imagery initially devoid of the sought features will be incepted by them. The generated images take advantage by strong regularizers privileging inputs that have statistics similar to natural images (e.g. correlations between neighboring pixels - texture).

We could also fix a layer and enhance whatever it detected. Each layer of the network deals with features at a different level of abstraction. Lower layers will produce strokes or simple ornament-like patterns because those layers are sensitive to basic features such as edges and their orientations.

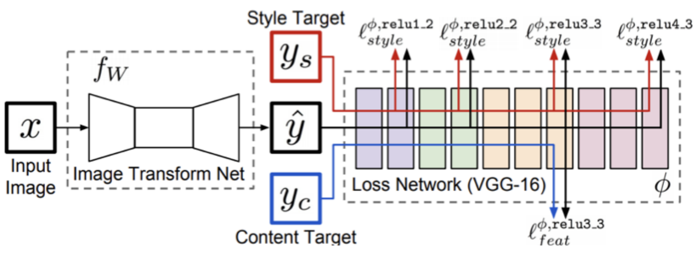
Style Transfer

The attempt of manipulating/re-creating an image mimicking the style of a particular work of art (painting, picture… e.g. Van Gogh's Starry Night). We're using meta-knowledge from Image Processing and Pattern Recognition. In order to capture the style of an image, we can use the so-called Gram Matrix (all possible combinations of the feature maps).

We know that at layer l an image is encoded with D^l distinct feature maps F^l_d each of size M^l (width$\times$height). $F^l_{d,x}$ is thus the activation of the filter d at position x at layer l. Feature correlations for the given image are given by the Gram Matrix $G^l \in R^{D^l \times D^l}$ where $G^l_{d_1,d_2}$ is the dot product between the feature maps $F^l_{d_1}$ and $F^l_{d_2}$ at layer l.

$$G^l_{d_1,d_2} = F^l_{d_1} \cdot F^l_{d_2} = \sum_k F^l_{d_1,k} \cdot F^l_{d_2,k}$$

Famous Variant not based on a Gradient Ascent approach → more or less the idea is the same. Function that computes a content target, a style target and a third input.


Style Target
$\ell_{style}^{\phi,relu1\_2}$
$\ell_{style}^{\phi,relu2\_2}$
$\ell_{style}^{\phi,relu3\_3}$
$\ell_{style}^{\phi,relu4\_3}$
y_s
f_W
x
Input Image
Image Transform Net
$\hat{y}$
y_c
Loss Network (VGG-16)
ϕ
$\ell_{feat}^{\phi,relu3\_3}$
Content Target

The network computes the loss (use of a complex loss). At each layer they compute the map of activations and the Gram Matrix. the loss will be the difference between the gram matrix from the third and the gram from style target plus some information from the content.

Recap: Possible Applications of Gradient Ascent

- If the loss corresponds to the activation of a specific neuron (or a specific layer) we may try to generate images that cause a strong activation of it, hence explaining the role of the neuron inside the network (what neurons see)
- If the loss is the distance, in the latent space, from the internal representation of a given image, we may try to generate other images with the same internal representations (hence explaining what features have been captured in the internal representation)
- If the loss is the similarity to a given texture, we may try to inject stylistic information in the input image

Distance from a target category

We can use the gradient ascent technique in an image classification framework to increase, starting from noise or any given picture, the score of whatever class we want.

Since we have many pixels, a tiny, imperceptible to humans, yet consistent perturbation of all of them is able to fool the classifier. The technique based on gradient ascent requires the knowledge of the neural network in order to fool it. We can do something similar using the network as a black box by means (i.e.) of evolutionary techniques. They were able to produce not only “noisy” adversarial images, but also geometrical examples with high regularities, which are meaningful for humans. In the evolutionary approach, we start with a random population of images and alternately apply selection (keep the best) and mutation (random perturbation/crossover)

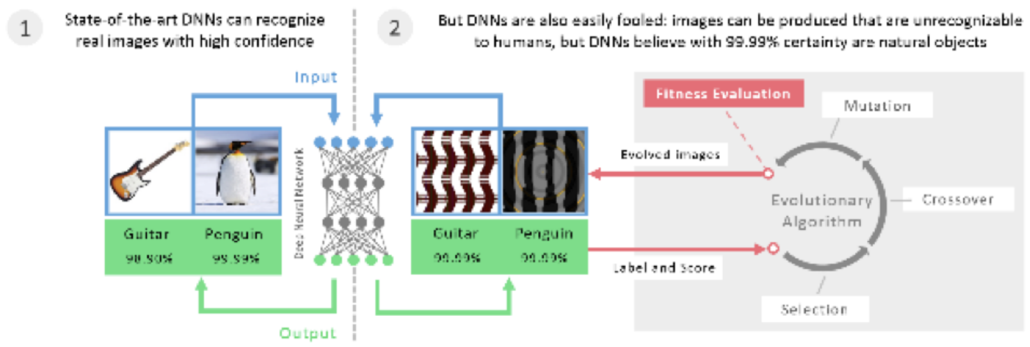

1
State-of-the-art DNNs can recognize real images with high confidence
2
But DNNs are also easily fooled: images can be produced that are unrecognizable to humans, but DNNs believe with 99.99% certainty are natural objects
Input
Deep Neural Network
Guitar
98.90%
Penguin
99.99%
Guitar
99.99%
Penguin
99.99%
Output
Fitness Evaluation
Mutation
Evolved images
Evolutionary Algorithm
Crossover
Label and Score
Selection

Figure 2. Although state-of-the-art deep neural networks can increasingly recognize natural images (*left panel*), they also are easily fooled into declaring with near-certainty that unrecognizable images are familiar objects (*center*). Images that fool DNNs are produced by evolutionary algorithms (*right panel*) that optimize images to generate high-confidence DNN predictions for each class in the dataset the DNN is trained on (here, ImageNet).

Lesson 11: Why classification techniques are vulnerable. The data manifold. Compression via Autoencoders

@April 4, 2023

Why Classification Techniques are Vulnerable

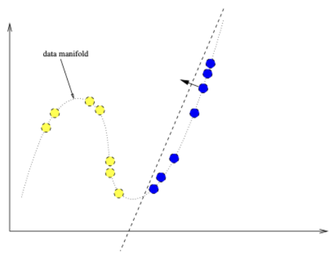
It's not related to Neural Networks, it's a general property. A classification network is a discriminative technique: the features used by the network allow you to distinguish between different categories, but are not necessarily informative about the subject's category. When you take a generative approach, you try first to understand the data, the distribution, etc. and after, you discriminate. When you take a discriminative approach, you're not interested in the distribution of the samples, but only in the features to discriminate the classes. The objects we're interested in occupy a low-dimensional portion in the feature space (negligible portion); so it's easy to make modifications.

The structure of a network for image processing is a sequence of convolutional layers where you extract information (try to lose as little as possible) and a short, dense final sequence of layers. While you're in the first section, it can make sense to invert, while on the last, you're losing too much information (you only have what you need to discriminate, not to recognize enough of the category to synthesize a new image)

The Data Manifold

If you generate a random image, it looks like noise; you will never get something that makes sense (the probability of randomly generating something that makes sense is null). This means that the so-called "natural images", i.e. the ones that make sense, occupy a portion of the feature space that has almost no dimension at all. Plus, we expect a continuity (small modifications still make sense, smooth linear interpolation still makes sense, in the sense that is different from noise). We expect the Manifold to be rather continuous.

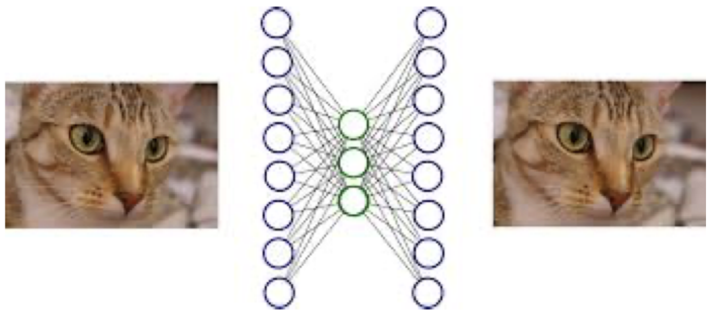
Manifold ⇒ underlying the idea of a structure within a set, which is not explicit. Suppose to have a two-features space and a manifold of dimension 1 and want to perform classification on a two-category dataset. In this example this means drawing a line. We want to influence the result = moving the object, change its features so that it crosses the border (escape the data manifold = create an outlier).



Can we influence the effect to stay close to the manifold? Inceptionism technique = exploits regularization techniques

Autoencoders

If the actual dimensionality of the manifold is rather low compared to the space, can we compress the representation? You can exploit the structure to derive more compact features.



An autoencoder is a net trained to reconstruct input data out of a learned internal representation. Usually, the internal representation has lower dimensionality w.r.t. the input.

We can perform compression because we exploit regularities (correlations) in the features describing input data. If the input has a random structure (high entropy) no compression is possible. If the internal layer has fewer units of the input, autoencoders can be seen as a form of data compression, which in this case is:

- data-specific → only works well on data with strong correlations
- lossy → the output is degraded with respect to the input
- directly trained on unlabeled data samples (self-supervised learning)

This means that they are not so good for data compression, but have applications in data denoising, anomaly detection, feature extraction (generalization of PCA) and generative models (VAE)

Lesson 12: Segmentation

We try to split the image into regions with uniform information, in which the information is semantic. We want to identify the region belonging to a particular category. Main applications in the medical field. Building a supervised training set is expensive, as it requires a complex human operation. Semantic Segmentation can be seen as Classification performed at pixel level: we classify each pixel in an image according to the object category it belongs to (semantic segmentation).

The label is itself an image, with a different color for each category. We could talk about image-to-image transformation.

Famous Datasets:

- Pascal VOC 2012 → Pascal Visual Object Classes: 20 classes, divided in 4 macro-areas: Person, Animal, Vehicle, Indoor
- COCO (Microsoft Common Objects in Context) → has a much wider application (300K images of which >200K are labeled)
- Cityscapes → large-scale dataset containing stereo video sequences recorded in street scenes from 50 different cities. 5K frames high-quality pixel-level annotated and 20K weakly annotated frames. Semantic understanding of urban street scenarios.
- Syntia → collection of photorealistic frames rendered from a virtual city and precise pixel-level semantic annotations for 13 classes.
- ScanNet → richly annotated 3D reconstructions of indoor scenes. RGB-D video dataset containing 2.5M views in more than 1500 scans annotated with 3D camera poses, surface reconstructions and instance-level semantic segmentations.
- Sun RGB-D → 10K RGB-D images of densely annotated indoor scenes

Convolutionalization

If you compose convolutions you get a convolution. Essentially the composition of convolutional layers still behaves as a convolutional layer. What is the stride of the composed convolution (compound kernel) that we get? Product of the strides of the composed convolutions.

As for the kernel dimension, we know that, neglecting padding:

$$D_{in} = S * (D_{out} - 1) + K$$

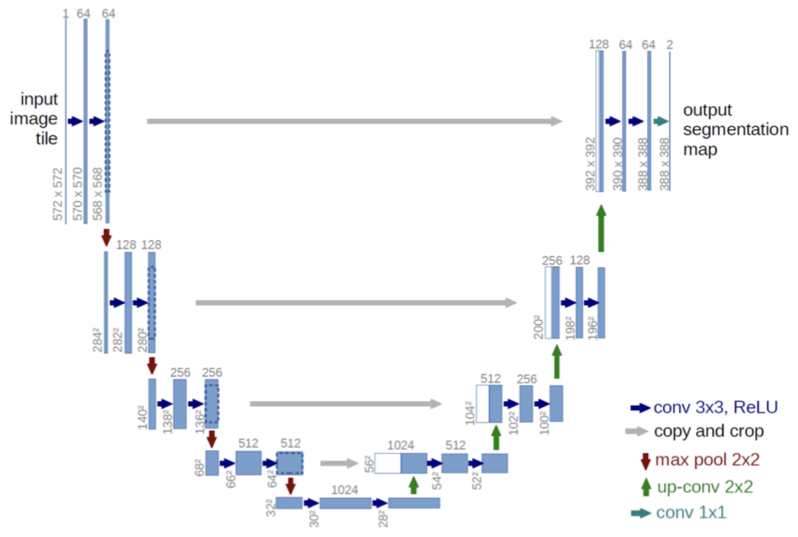
From this, we can apply the rule and obtain the compound kernel.

What breaks convolutionality are the dense layers at the end of the networks (if max pooling has a fixed pooling dimension).

We know that neurons in dense or convolutional layers share the same functionalities: they simply compute weighted sums of their inputs (dot products). So we can look at a dense layer as a convolution with a particularly large filter, equal to its input size. This way, each dense layer can be turned into a convolutional layer, reusing the same weights, making the resulting networks fully convolutional.

Inception V3 as a single convolution → input with dimension 75 (smallest flat input to pass to Inception V3) and stride 32. The canonical would be something like 299. (output of dimension 8x8 when feature map pooling operations). The result of such a coarse network would though be far from precise. Some have addressed the issue by adding skips that combine the final prediction layer with lower layers with finer strides.

U-Net



Relatively simple architecture that does not rely on a classification network, but learns segmentation in an end-to-end setting. Works in two directions:

- Downscale+Upscale → convolutional autoencoder
- Horizontal

It can work well with relatively few training images, due to a large exploiting of data augmentation, and yields accurate precise segmentations.

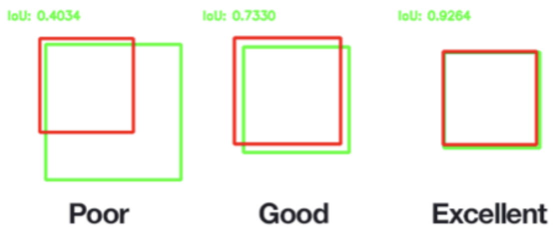
Lesson 13: Object Detection

@April 12, 2023

In a sense, it is quite similar to segmentation. The difference is in the way we expect to get the results: we are supposed to return a boundary box containing the object. So, you don't need to strive about borders. Issue: you don't know how many outputs you're supposed to obtain. In the case of YOLO, the output of the network is post-processed. Segmentation can be understood as an image-to-image task, with a clear loss function, while in this case it is not evident.

We have a ground truth, the network makes a guess (another box). How good is the box predicted? We use a notion of similarity between the boxes called Intersection-over-Union:

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



Results of IoU and quality of the approximation. In case of disjoint boxes, we have 0. The Green box is the ground truth.

The two most important datasets for Object Detection are PASCAL VOC and Coco (which are also used for Segmentation).

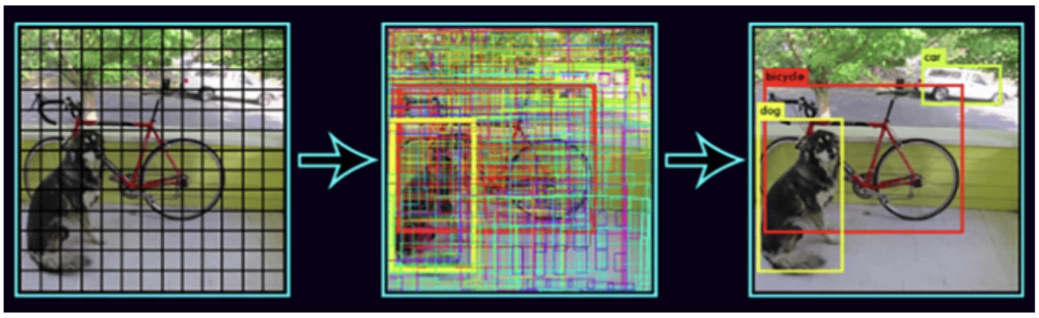
Two main approaches:

- Region Proposal methods → steps. You have a first phase in which you try and identify the regions of interest independently of the objects you're looking for (Selective Search algorithms). They exploit the discontinuities in texture and structure of the image and are object-independent. Examples: R-CNN
- Single Shot methods → e.g. YOLO. You try to do everything in a single pass (region of interest - box - content recognition). They are slightly faster than region proposal and therefore suited for real-time applications.

Python Library `Detectron2` where you can find implementations and evaluations of novel computer vision research (e.g. Mask R-CNN, RetinaNet…)

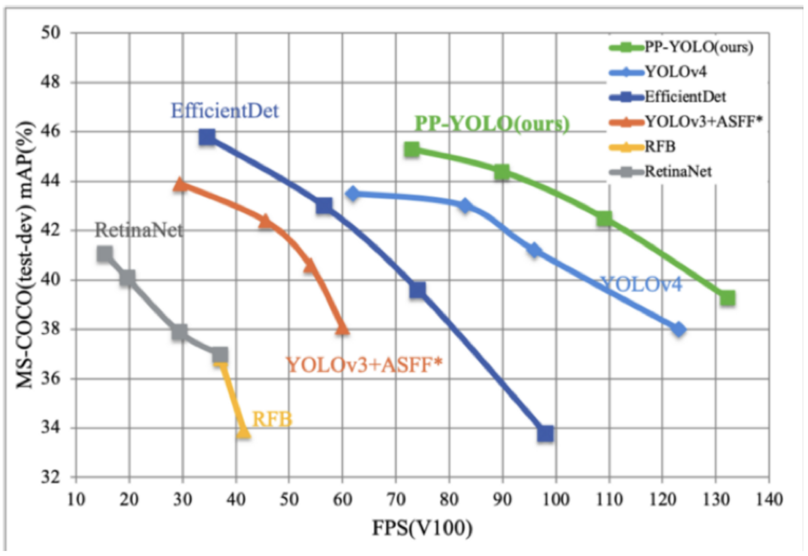
YOLO: Real-Time Object Detection

Acronym: You Only Look Once → a single pass of the input image is enough to process the image. The network outputs many predictions, each labelled with a confidence value, which is then leveraged in post-processing to select the most promising boxes (it is done algorithmically). NP-Complete problem that makes use of heuristics.



Here the thickness of the box is used to express the confidence

The author (Joseph Redmon) quit because he was concerned about the applications of his research for military purposes.



A prediction is a bounding box, with a label expressing the kind of objects we expect to find inside the box. The label is expressed as a probability.

Who's doing the prediction? As usual, we're processing images by means of CNNs, which are composed by sequences of convolution-downsampling-… we are interested to increase the receptive fields of the neurons. Grid of 13x13 neurons, each portion of the grid does the process and will be asked to output a prediction (actually, three predictions).

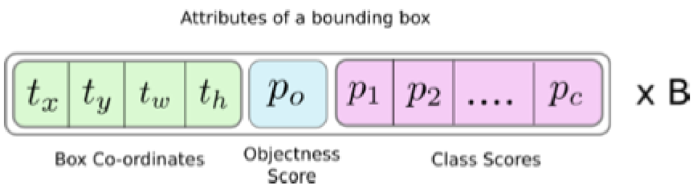
How can I train the network to recognize the boxes? Suppose I have the ground truths, now I want to train the network to produce. It is not trivial. The neuron that is supposed to recognize the object is the one in the middle.

In the loss function, you will have a boolean mask that will put to 0 all the neurons that not correspond to the middle of any objects and multiply for the confidence.

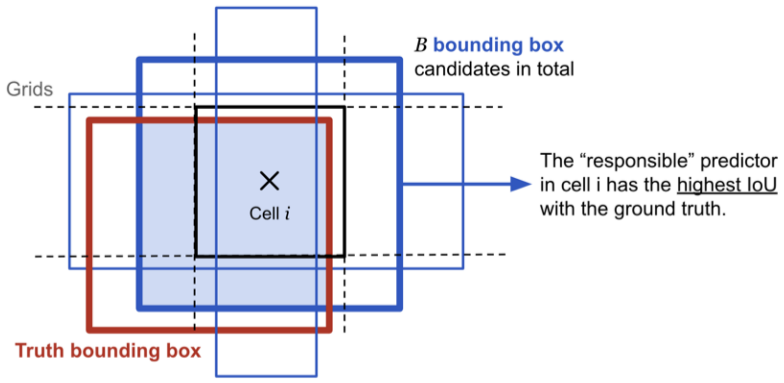
Mean average precision matrix used for performance comparisons.

YOLO's Architecture

YOLO is a fully convolutional neural network, with the input progressively downsampled by a factor $2^5 = 32$. So, an input image of dimension 416x416 is reduced to a grid of neurons of dimension 13x13 called feature map. You expect (and so it is) that the neuron whose grid-cell contains the center of the bounding box is responsible for detection; this neuron makes a finite number of predictions. Depth-wise, we have $B \times (5 + C)$ entries, where B is the number of bounding boxes each cell can predict and C the number of different object categories. Each bounding box has $5 + C$ attributes, which describe the center coordinates, the dimensions, the objectness score and C class confidences.



First improvement: trying to directly guess the coordinates leads to unstable gradients during training. Most of the modern implementations (log-space affine transforms for pre-defined default boxes) use anchor boxes: fixed number of predefined shapes, with dimensions. You define the boxes by inspection of the dataset. The prediction is allowed to perform small deformations of the anchors (e.g. translation, size). The bounding box responsible for detecting the object is one whose anchor has the highest IoU with the ground truth box.


B bounding box candidates in total
Grids
The "responsible" predictor in cell i has the highest IoU with the ground truth.
Cell i
Truth bounding box

Anchors are chosen by K-Means clustering on the training set, reflecting the most likely shapes of bounding boxes.

Predictions

$$
\begin{aligned}
b_x &= c_x + \sigma(t_x) \\
b_y &= c_y + \sigma(t_y) \\
b_w &= p_w * e^{t_w} \\
b_h &= p_h * e^{t_h}
\end{aligned}
$$

where t_x, t_y, t_h, t_w are the network outputs relative to the cell at coordinates c_x, c_y for an anchor of dimensions p_w, p_h.

YOLO does not predict the absolute coordinates of the bounding box's center, but offsets which are relative to the top left corner of the grid cell which is predicting the object and normalized by the dimensions of the cell from the feature map (which is 1 and motivates the use of the sigmoid).

The dimensions of the bounding box are predicted by applying a log-space transform to the output and then multiplying with dimensions of the anchor. The resulting predictions b_w and b_h are normalized by the height and width of the image. Training labels are chosen this way.

Objectness Score

Probability that an object is contained inside a bounding box. It is also passed through a sigmoid, as it is to be interpreted as a probability.

Class Confidences

Probabilities of the detected objects belonging to a particular class. Before v3, YOLO class scores were computed via Softmax function. Since YOLOv3, multiple sigmoid functions are used instead, considering that objects might belong to multiple hierarchical categories and hence labels are not guaranteed to be mutually exclusive.

YOLO's Loss Function

Two parts: localization loss and classification loss (it is a composition):

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

where i ranges over cells and j over bounding boxes. 1_{ij}^{obj} is a delta function indicating whether the j-th bounding box of the cell i is responsible for the object prediction.

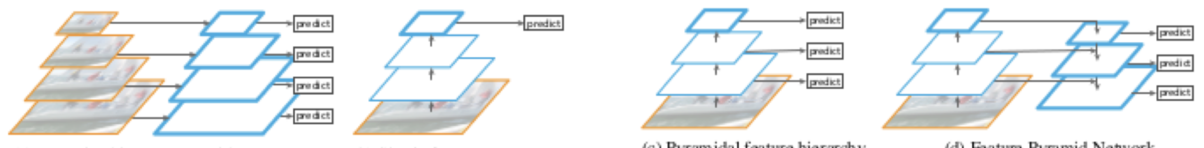
$$\mathcal{L}_{cls} = \sum_{i=0}^{S^2} \sum_{j=0}^{B} (1_{ij}^{obj} + \lambda_{noobj}(1 - 1_{ij}^{obj}))(C_{ij} - \hat{C}_{ij})^2 + \sum_{i=0}^{S^2} \sum_{c \in C} 1_{i}^{obj} (p_i(c) - \hat{p}_i(c))^2$$

λ_{noobj} is a configurable parameter meant to down-weight the loss contributed by background cells containing no objects (which are a large majority).

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

The balance must be human-calibrated through the use of λ_{coord}. You can't learn it. It is a hyperparameter. In YOLO: $\lambda_{coord} = 5, \lambda_{noobj} = 0.5$

Multi-Scale Processing



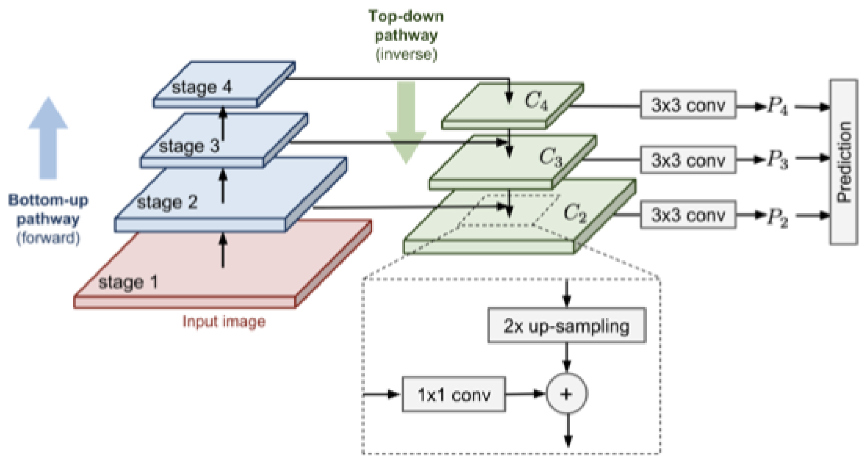
(a) Featurized image pyramid (b) Single feature map (c) Pyramidal feature hierarchy (d) Feature Pyramid Network

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

(b) First systems for fast object detection (YOLOv1) opted to use only higher level features at the smallest scale. This compromises detection of small objects.

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

(d) Modern Systems (FPN, RetinaNet, YOLOv3) recombine features along a backward pathway. This is as fast as (b) and (c), but more accurate.

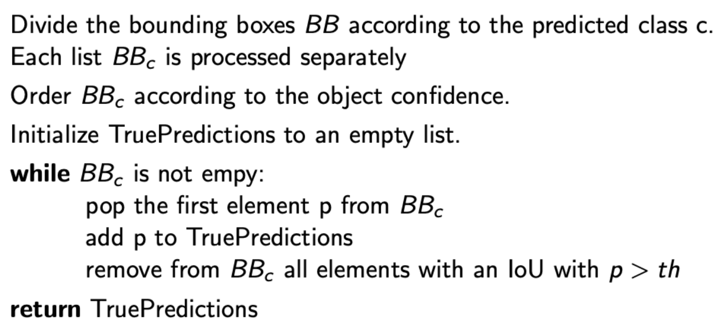


The Bottom-Up pathway is the normal feedforward computation, while the Top-Down pathway goes in the inverse direction, adding coarse but semantically stronger feature maps back into the previous pyramid levels of a larger size via lateral connections. The higher-level features are spatially upsampled; the feature map coming from the bottom-up pathway undergoes channel reduction via 1x1 convolutional layer. Finally, the two feature maps are merged.

Non-Maximum-Suppression

YOLOv3 predicts feature maps at scales 13, 26 and 52. At the end, we have a total of 10647 bounding boxes, each one of dimensions 85 (4 coordinates, 1 confidence and 80 class probabilities). To reduce the number, we operate algorithmically through:

- Thresholding by Object Confidence → we filter boxes based on their objectness score
- NMS → address the problem detections of the same image, corresponding to different anchors, adjacent cells in maps.



```
Divide the bounding boxes $BB$ according to the predicted class c.
Each list $BB_c$ is processed separately
Order $BB_c$ according to the object confidence.
Initialize TruePredictions to an empty list.
while $BB_c$ is not empy:
    pop the first element p from $BB_c$
    add p to TruePredictions
    remove from $BB_c$ all elements with an IoU with $p > th$
return TruePredictions
```

About Ablation

In general, ablation is the removal or destruction of material from an object. In the context of Deep Learning, it consists in a progressive removal of components of the network, aimed to assess their contribution to the overall behavior.

Lesson 14: Generative Models

@April 18, 2023

When you use Discriminative Models you're interested mainly in Classification. In Generative Models you try to learn the actual probability distribution of the different classes from the training set, and then eventually go for discrimination. They can also be applied to unsupervised settings as the goal is to build the probability distribution p_{model} close to the probability distribution of the data p_{data}. We can either define its mathematical formulation or try to learn a generator according to p_{model}, possibly providing estimations of the likelihood.

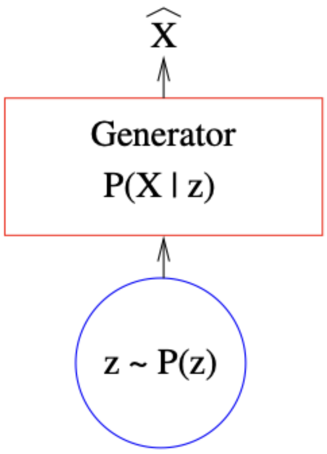
Why do we study Generative Models (Adversative Studies)? To improve our knowledge on data and their distribution in the visible feature space. To improve our knowledge on the latent representation of data and the encoding of complex high-dimensional distributions (an alternative name for most of the models we'll see is Latent Variable Models). It is a typical approach in many problems involving multi-modal outputs. More practically, we would want to find a way to produce realistic samples from a given probability distribution. Generative models can be incorporated into reinforcement learning, like for example to predict possible futures.

We talk about multi-modal output when there is no unique intended solution to a given problem (e.g. add colors to a gray-scale image). When the output is intrinsically multi-modal and we do not want to give up the possibility to produce multiple outputs, we need to rely on generative modeling.

Latent Variable Models → we express the probability of a data point through marginalization over a vector of latent variables.

$$P(X) = \int P(X|z)P(z)dz \approx \mathbb{E}_{z\sim P(z)}P(X|z)$$

$P(X)$ represents the probability that X belongs to the actual population we are considering. We try to learn a way to sample X starting from a vector of values z ($P(X|z)$ - Generator: computes the probability on the vector of latent variables), where z is distributed with a known prior distribution $P(z)$, otherwise we wouldn't know how to sample. z is called latent encoding of X and represents an internal representation of X.

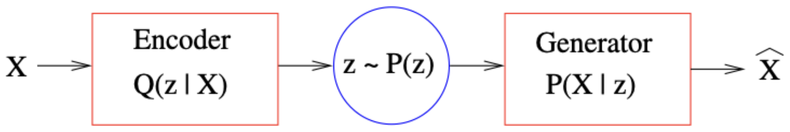


The latent and visible space are different. The Visible space has a usually higher dimension, as it commonly represents images, while the latent space can be much smaller. So, to sum up, Dimension of the latent space ≤ dimension of the visible space

There are four main classes of generative models, that differ in the way the generator is trained

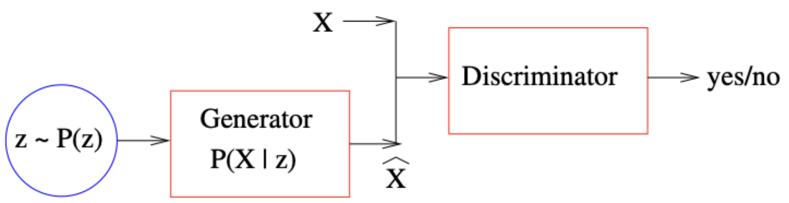
- Compressive models → latent space smaller
    - Variational AutoEncoders (VAEs)

      You couple the generator (that can be sees as a sort of decoder) with an encoder producing a latent encoding z given X. This will be distributed according to an inference distribution $Q(z|X)$ - Marginal Distribution. You jointly train the generator and the encoder. The coupling helps understand in which way to train the model to reconstruct the original input image. We don't know the distribution of the z inside the latent space in a classical AutoEncoder. In the VAE, we try to force the z to assume a known distribution (usually Gaussian), by adding Kullback-Leibler Divergence component. The loss function aims to minimize the reconstruction error between X and $\hat{X}$ and bring the marginal inference distribution $Q(z)$ close to the prior $P(z)$.

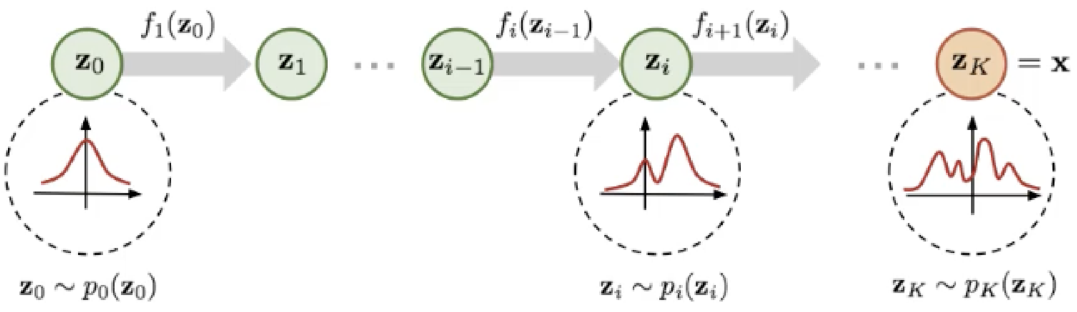


    - Generative Adversarial Networks (GANs)

      Couple the generator with a discriminator. The purpose of the discriminator is to distinguish between real data from the real distribution and fake data created by the generator . The discriminator and generator must be trained alternatively (you freeze one while you train the other). The point of the generator is: I have no knowledge of what's a good point, I just wanna fool the discriminator by doing something sufficiently good. Tricky and slow process. The loss function aims to instruct the detector to spot the generator and instruct the generator to fool the detector

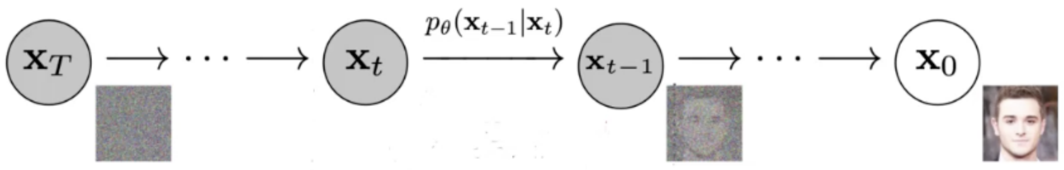


- Dimension preserving → latent space equal
  - Normalizing Flows


$f_1(\mathbf{z}_0)$
$\mathbf{z}_0$
$\mathbf{z}_1$
$f_i(\mathbf{z}_{i-1})$
$\mathbf{z}_{i-1}$
$f_{i+1}(\mathbf{z}_i)$
$\mathbf{z}_i$
$\mathbf{z}_K = \mathbf{x}$
$\mathbf{z}_0 \sim p_0(\mathbf{z}_0)$
$\mathbf{z}_i \sim p_i(\mathbf{z}_i)$
$\mathbf{z}_K \sim p_K(\mathbf{z}_K)$

    The idea is to split the generator in a long chain of transformations, assuming that all the transformations are invertible. We know that the two spaces have the same dimensions, so the idea is to deform the latent space to make it look the most similar to the real data. The network is trained by maximizing the log-likelihood. We have a precise computation of the likelihood (high confidence). Negative: you need to restrict to invertible transformations, which limit the expressiveness.

  - Denoising Diffusion Models



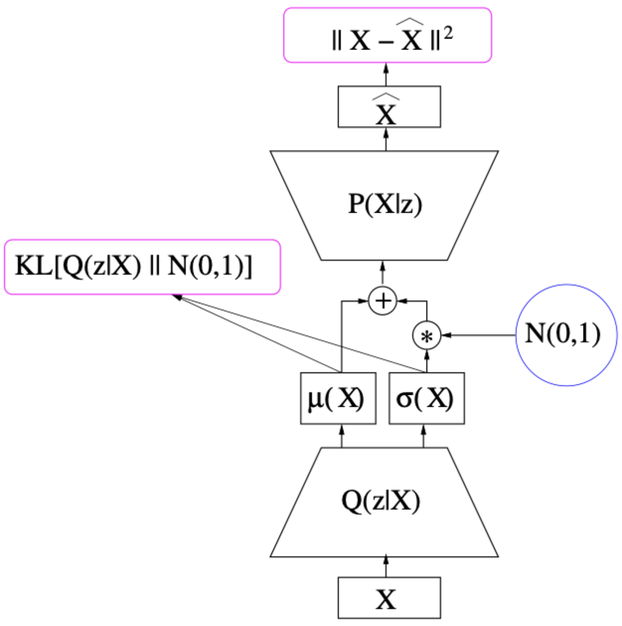
    It is the technique of the moment. In Diffusion Models you understand the latent space as a strongly noised version of the image to be generated. And so the generator is a long chain of reverse diffusion steps (You try to extract some data from the noise). Diffusion = add noise to the data. Relatively slow operation (1000 iterations of steps).

Variational Autoencoders

An autoencoder is a network trained to reconstruct input data out of a learned internal representation. We can’t use the decoder to generate data by sampling in the latent space, as we do not know the distribution of latent variables.

In a Variational Autoencoder we try to force latent variables to have a known prior distribution $P(z)$ (e.g. a Normal Distribution). If the distribution computed by the generator is $Q(z|X)$ we try to force the marginal distribution $Q(z) = \mathbb{E}_{X \sim P_{data}} Q(z|X)$ to look like a normal distribution. We assume $Q(z|X)$ has a Gaussian distribution $G(\mu(X), \sigma(X))$ - distribution of the latent variables associated with our input X - with different moments for each different input X. The values $\mu(X), \sigma(X)$ are both computed by the generator, which is hence returning an encoding $z = \mu(X)$ and a variance $\sigma(X)$ around it, expressing the portion of the latent space essentially encoding an information similar to X.

During training, we sample around $\mu(X)$ with the computed $\sigma(X)$ before passing the value to the decoder. Among other things, sampling adds noise to the encoding, improving its robustness.



Pipeline of the VAE

$\mu(X)$ and $\Sigma(X)$ are not used to generate new samples from the input domain, as we have no X.

The KL component is the distance between two gaussian components (probability distributions). The effect of KL-Divergence on latent variables consist in:

- one component pushing $\mu_z(X)$ towards zero, so as to center the latent space around the origin
- push $\sigma_z(X)$ towards 1, augmenting the “coverage” of the latent space, essential for generative purposes

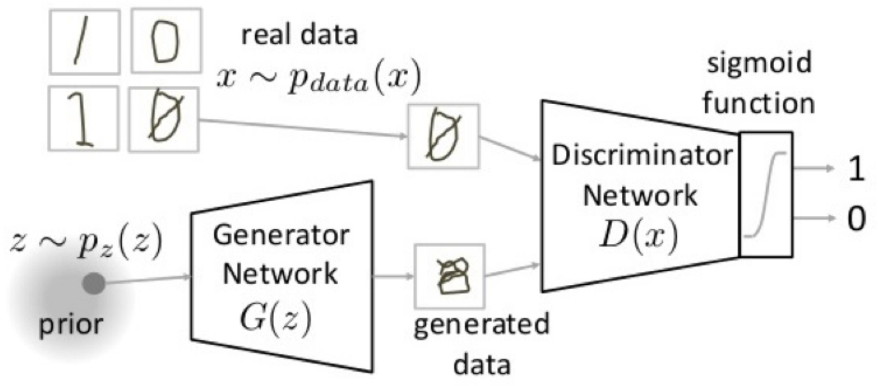
Problems with VAE:

- balancing log-likelihood and KL regularizer in the loss function
- variable collapse phenomenon → can also have positives, as we are identifying an implicit dimension of the latent space
- marginal inference vs prior mismatch

- blurriness (aka variance loss)

Generative Adversarial Networks

@April 19, 2023



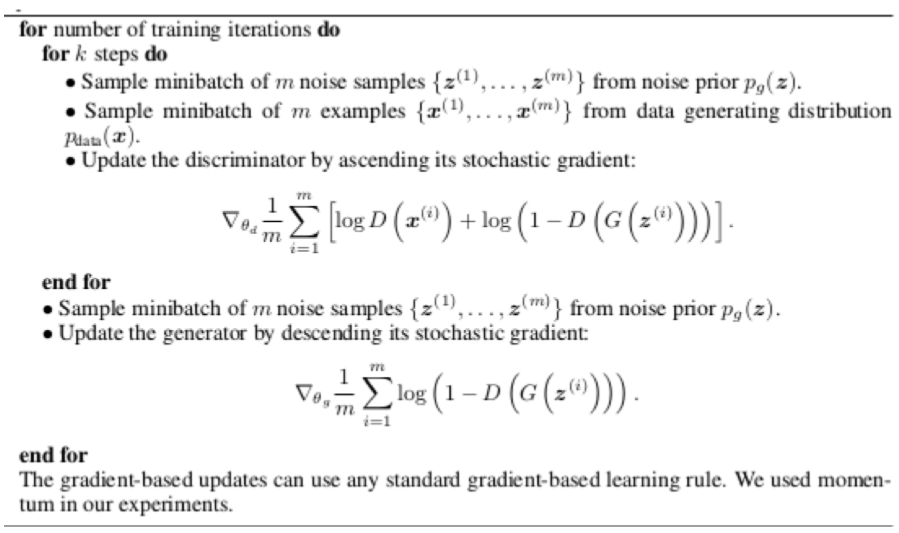
Purpose of the Discriminator is to distinguish good data from “fake” data generated by the generator starting from a seed (the discriminator network is basically a classifier). From a mathematical point of view it is a MinMax Game.

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

$\mathbb{E}_{X\sim p_{data}(x)}[\log D(x)]$ is the negative cross entropy of the discriminator w.r.t. the true data distribution

$\mathbb{E}_{z\sim p_z(x)}[\log(1 - D(G(z)))]$ is the negative cross entropy of the “false” discrimination w.r.t. the fake generator.

The discriminator and generator are trained alternatively, while freezing the other.



```
for number of training iterations do
    for $k$ steps do
        • Sample minibatch of $m$ noise samples $\{z^{(1)}, \dots, z^{(m)}\}$ from noise prior $p_g(z)$.
        • Sample minibatch of $m$ examples $\{x^{(1)}, \dots, x^{(m)}\}$ from data generating distribution $p_{data}(x)$.
        • Update the discriminator by ascending its stochastic gradient:
            $\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(x^{(i)}\right) + \log\left(1 - D\left(G\left(z^{(i)}\right)\right)\right)\right].$
    end for
    • Sample minibatch of $m$ noise samples $\{z^{(1)}, \dots, z^{(m)}\}$ from noise prior $p_g(z)$.
    • Update the generator by descending its stochastic gradient:
        $\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(z^{(i)}\right)\right)\right).$
end for
The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.
```

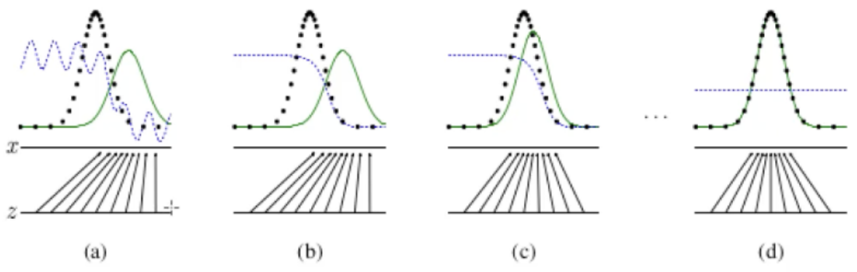


Figure 1: Generative adversarial nets are trained by simultaneously updating the discriminative distribution (D, blue, dashed line) so that it discriminates between samples from the data generating distribution (black, dotted line) $p_{\boldsymbol{x}}$ from those of the generative distribution p_g (G) (green, solid line). The lower horizontal line is the domain from which $\boldsymbol{z}$ is sampled, in this case uniformly. The horizontal line above is part of the domain of $\boldsymbol{x}$. The upward arrows show how the mapping $\boldsymbol{x} = G(\boldsymbol{z})$ imposes the non-uniform distribution p_g on transformed samples. G contracts in regions of high density and expands in regions of low density of p_g. (a) Consider an adversarial pair near convergence: p_g is similar to p_{data} and D is a partially accurate classifier. (b) In the inner loop of the algorithm D is trained to discriminate samples from data, converging to $D^*(\boldsymbol{x}) = \frac{p_{\text{data}}(\boldsymbol{x})}{p_{\text{data}}(\boldsymbol{x})+p_g(\boldsymbol{x})}$. (c) After an update to G, gradient of D has guided $G(\boldsymbol{z})$ to flow to regions that are more likely to be classified as data. (d) After several steps of training, if G and D have enough capacity, they will reach a point at which both cannot improve because $p_g = p_{\text{data}}$. The discriminator is unable to differentiate between the two distributions, i.e. $D(\boldsymbol{x}) = \frac{1}{2}$.

There's a typical unstable behavior - see demo on slides. One of the typical problems of GAN is that if you have a distribution with two peaks, it can focus on creating only points for one of the two.

Why the quality is good? When we trained the VAE, our objective was to create, reconstruct something similar to the input. Difficult because a tiny modification in the latent space must generate a tiny modification of the data. The GAN is different, it has nothing to reconstruct, its purpose is just to fool the discriminator and choose the best patch as possible. Therefore the results are perceived as better.

Problems:

- The fact that the discriminator gets fooled does not mean that the fake is good (neural networks get easily fooled)
- Problems with the global understanding of data $\rightarrow$ due to the latent encoding $\Rightarrow$ the dimension of the latent space influences the network.
- Mode collapse phenomenon: generative specialization on a good, fixed sample $\rightarrow$ the generator just learns to generate well a single point of the distribution (e.g. one face) and is happy just like that, as the discriminator is not able to recognize it. Induce diversity = modifications on the loss function, like imposing some regularization techniques. It can be proved that although GANs have better performance, they lack in diversity.

Latent Space Exploration (not specific of GANs)

A generator is a continuous process, so small movements inside the latent space, result in small movements/modifications in the visible space.

Suppose attribute editing $\rightarrow$ e.g. transform the picture of a cat into one of a dog, working in the Latent Space. Compute all the latent representations of cats and dogs and draw a frontier inside the

Latent Space (which is an hyperplane). The perpendicular to the hyperplane tells us the direction in which to move the latent embedding of the cat to obtain a dog.

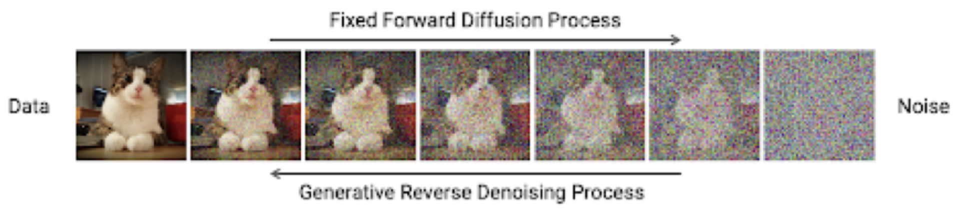
The generative process is continuous: a small de-placement in the latent space produces a small modification in the visible space. Real-world data depends on a relatively small number of explanatory factors of variations (latent features) providing compressed internal representations. Understanding these features, we may define trajectories producing desired alterations of data in the visible space.

When there is more than one attribute, editing one may affect another, since some semantics can be coupled with each other (entanglement). To achieve more precise control (disentanglement), we could use projections to force the different directions of variation to be orthogonal to each other.

Relation between the latent space of different generative techniques (even trainings of the same networks). Able to transform the latent space of a model into the latent space of another through the use of a simple linear map preserving most of the content.

The organization of the latent space seems to be independent from the training process, the network architecture and the learning objective (GAN and VAE share the same space). The map can be defined by a small set of point common to the two spaces (support set). Locating these points in the two spaces is enough to define the map.

Diffusion Models


Fixed Forward Diffusion Process
Data
Noise
Generative Reverse Denoising Process

Fairly recent technique, based on a solid mathematical foundation, with a lot of different applications.

The Generative Process as a sort of reversed Forward Diffusion Process (take an image and little by little you add gaussian noise). Train a single network on the move from an image, add a fixed amount of noise. Then train the network to de-noise the image and generate back the original image. Take in input the noised image and a rate (amount of the original signal still in the image).

The denoising network implements the inverse of the operation of adding a given amount of noise to an image (direct diffusion). The denoising network takes in input:

1. a noisy image x_t
2. a signal rate α_t expressing the amount of the original signal remaining in the noisy image

and try to predict the noise in it: $\epsilon_\theta(x_t, \alpha_t)$. The predicted image would then be:

$$\hat{x}_0 = \frac{x_t - \sqrt{1-\alpha_t} \cdot \epsilon_\theta(x_t, \alpha_t)}{\sqrt{\alpha_t}}$$

Training Step

- Take an input image x_0 in the training set and normalize it
- Consider a signal ratio α_t
- Generate a random noise $\epsilon \sim N(0,1)$
- Generate a noisy version x_t of x_0 defined as $x_t = \sqrt{\alpha}_t \cdot x_0 + \sqrt{1-\alpha_t} \cdot \epsilon$
- Let the network predict the noise $\epsilon_\theta(x_t, \alpha_t)$ from the noisy image x_t and the signal ratio α_t
- Train the network to minimize the prediction error: $\min \|\epsilon - \epsilon_\theta(x_t, \alpha_t)\|$

Sampling Procedure

- Fix a scheduling $\alpha_T > \alpha_{T-1} > ... > \alpha_1$
- Start with a random noisy image $x_T \sim N(0,1)$
- For t in $T...1$ do
    - Compute the predicted error $\epsilon_\theta(x_t, \alpha_t)$
    - Compute $\hat{x}_0 = \frac{x_t - \sqrt{1-\alpha_t} \cdot \epsilon_\theta(x_t, \alpha_t)}{\sqrt{\alpha_t}}$
    - Obtain x_{t-1} reinjecting noise at rate α_{t-1}, namely $x_{t-1} = \sqrt{\alpha_{t-1}} \cdot \hat{x}_0 + \sqrt{1-\alpha_{t-1}} \cdot \epsilon$

As architecture, one could use a conditional UNet.

Lesson 16: Conditional Generation

@April 26, 2023

Up until now, we wanted to sample data according to a distribution, but we’re interested in generating data according to some attributes. Neural Networks usually compute a single function (monolithic in some sense), but we’re interested in some parametrization - compute a family of functions - according to some additional features (e.g. generation according to some specific sets of attributes).

We have two issues:

- How do we take into account - integrate - the condition inside the generative model (how do we change the design of the model)
- How can we concretely pass and process all the conditions inside the neural network

We will deal with CVAE and GANs, but the issues stay also for Diffusion Models.

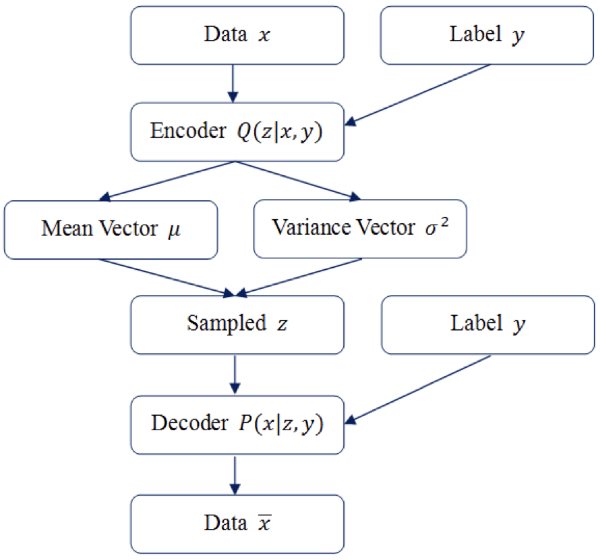
Conditional VAE (CVAE)

We have an encoder $Q(z|X)$ who's trying to compute the latent representation and a decoder $P(X|z)$ who's trying to compute the output. They are now parametrized with respect to a given condition/attribute we associate to $X \rightarrow Q(z|X, c)$ and $P(X|z, c)$.

Now, how do we deal with the prior?

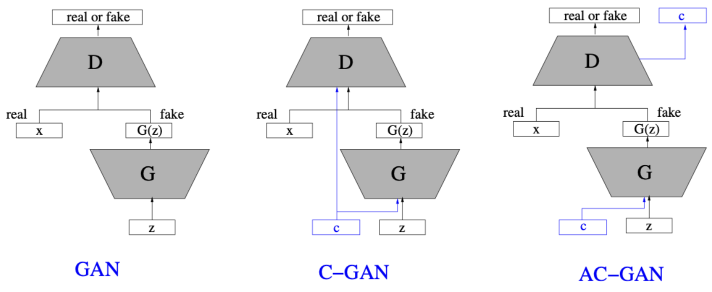
- We can still work with a single, condition-independent prior - -so we keep working with a Gaussian distribution - which is simpler, but heavier on the decoder side
- We can also use a different, possibly learned, prior for each condition, which is more complex and it is not clearly stated how beneficial it is

The structure of the CVAE is similar to the original VAE. We have our inputs: the data x and our labels y. The encoder produces still a mean and variance vectors, from which you sample z. To the decoder you pass the samples and the labels and try to obtain an output the most similar to the original data. In the mean vector we have the KL component that tries to force the distribution to be similar to a Gaussian distribution.



You simply treat the condition as an additional input

Conditional GANs



The generator takes in input the condition, in addition to the noise. The discriminator, instead:

- uses the condition to discriminate fakes for real of the given class (Conditional GAN) - it is an approach similar to the CVAE. The label guides the classification of the data by the discriminator. It works, but can be slightly improved
- tries to classify w.r.t. different conditions in addition to true/fake discrimination (Auxiliary Classifier GAN) - You add a classifier to the discriminator that learns to guess what is the category of the image it receives as input (you don't pass the labels to the discriminator anymore).

AC-GAN Loss Function

$p^*(x, c)$ is the true image-condition joint distribution, $p_\theta(x, c)$ is the joint distribution of generated data, $q_\theta(c|x)$ is the classifier.

In addition to the usual GAN objective (the MinMax Game), we also try to minimize further quantities:

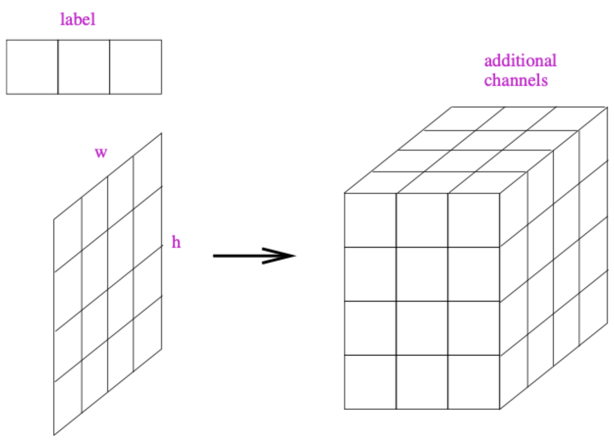
$$-\mathbb{E}_{p^*(x,c)} \ln(q_\theta(c|x)) - \mathbb{E}_{p_\theta(x,c)} \ln(q_\theta(c|x))$$

which means that the classifier should be consistent with the real distribution (which is a feature of InfoGANs) and it must create images easy to classify → images far from boundaries between classes, likely sharper.

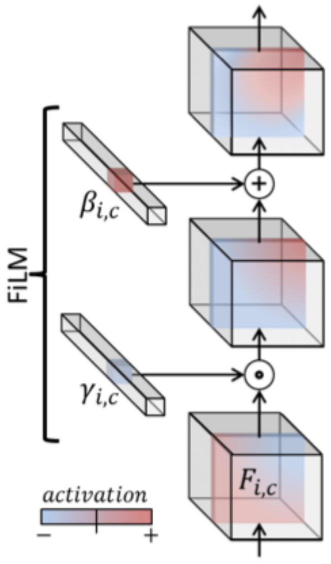
Concrete Handling of the Condition

In Conditional Networks, we pass the label/condition as an additional input. How do we process the input? If we need to add it to a dense layer, we just concatenate the label to the input. If we need to add it to a convolutional layer, we have two basic ways:

- Vectorization (more typical, mainly for first attempts)→ repeat the label (typically in categorical form) for every input neuron and stack them as new channels - idea that convolutions use just a portion of the data and you want to transmit the conditions to all the possible convolutions.



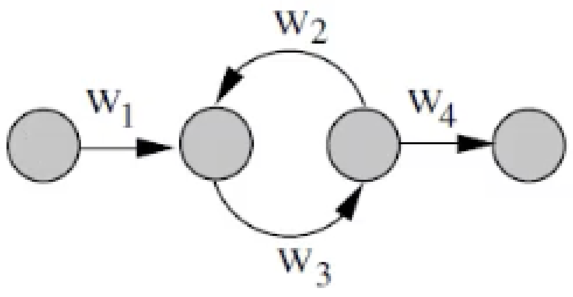
- Feature-wise Linear Modulation (FILM) → the idea is to use the condition to give a different weight to each feature (each channel). We use the kernels to create feature maps with different weights. We use the condition to generate two vectors γ and β with size equal to the channels of the layer; then, we rescale layers by γ and add β. It is a less invasive approach than parametrizing the weights.



This method is a form of Attention

Lesson 17: Recurrent Neural Networks

A Recurrent Neural Network is a network with cycles in it.

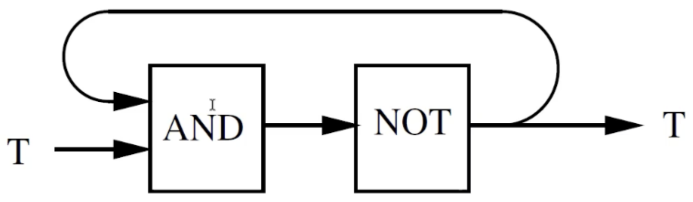


This is the alternative to Feed-Forward Neural Networks (which is what we've seen until now).

RNN are associated with two important notions:

- Good for processing temporal sequences data
- "Memory" in the sense that the output (hidden states) depends on the past history of the network and data inside the network

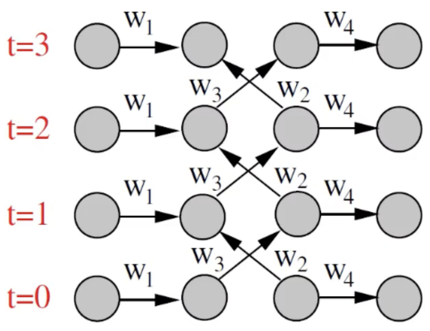
An issue that must be considered in cyclic network (that can be seen in Logical Circuits) is that they are unstable.



Suppose that the input is True and the last value the Network has seen is a True again; this value is fed back to the first AND by the cycle, to give a True, which is fed to the NOT and gives back a False. If the output is False, the final result will be True and cycles like this infinitely.

Solution: we need a clock and we observe the output of the system at specific time steps given by the clock, and this time step should allow the complete propagation of the signal through the whole network.

Idea of Temporal Unfolding of the network → state of the Neural Network at different time steps. The flow of information is from a state in a time step, to a state in the following time step.



We have a different input at each time step, similarly we have different outputs at each time step and the RNN transforms the sequence of input values into the sequence of output values.