Deep Learning

Last update: 27 April 2024

Academic Year 2023 – 2024

Alma Mater Studiorum · University of Bologna

Contents

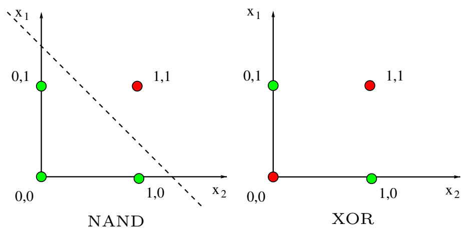
1 Neural networks expressivity

1.1 Perceptron

Single neuron that defines a binary threshold through a hyperplane:

$$\begin{cases} 1 & \sum_i w_i x_i + b \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

Expressivity A perceptron can represent a NAND gate but not a XOR gate. Perceptron expressivity



Remark. Even if NAND is logically complete, the strict definition of a perceptron is not a composition of them.

1.2 Multi-layer perceptron

Composition of perceptrons.

Shallow neural network Neural network with one hidden layer. Shallow NN

Deep neural network Neural network with more than one hidden layer. Deep NN

Expressivity Shallow neural networks allow to approximate any continuous function Multi-layer perceptron expressivity

$$f : \mathbb{R} \rightarrow [0, 1]$$

Remark. Still, deep neural networks allow to use less neural units.

1.2.1 Parameters

The number of parameters of a layer is given by:

$$S_{\text{in}} \cdot S_{\text{out}} + S_{\text{out}}$$

where:

- S_{in} is the dimension of the input of the layer.
- S_{out} is the dimension of the output of the layer.

Therefore, the number of FLOPS is of order:

$$S_{\text{in}} \cdot S_{\text{out}}$$

2 Training

2.1 Gradient descent

1. Start from a random set of weights w. Gradient descent
2. Compute the gradient $\nabla\mathcal{L}$ of the loss function.
3. Make a small step of size $-\nabla\mathcal{L}(w)$.
4. Go to 2., until convergence.

Learning rate Size of the step. Usually denoted with μ. Learning rate

$$w = w + \mu\nabla\mathcal{L}(w)$$

Optimizer Algorithm that tunes the learning rate during training. Optimizer

Stochastic gradient descent Use a subset of the training data to compute the gradient. Stochastic gradient descent

Full-batch Use the entire dataset.

Mini-batch Use a subset of the training data.

Online Use a single sample.

Remark. SGD with mini-batch converges to the same result obtained using a full-batch approach.

Momentum Correct the update v_t at time t considering the update v_{t-1} of time $t-1$. Momentum

$$\begin{aligned} w_{t+1} &= w_t + v_t \\ v_t &= \mu\nabla\mathcal{L}(w_t) + \alpha v_{t-1} \end{aligned}$$

Nesterov momentum Apply the momentum before computing the gradient. Nesterov momentum

Overfitting Model too specialized on the training data. Overfitting

Methods to reduce overfitting are:

- Increasing the dataset size.
- Simplifying the model.
- Early stopping.
- Regularization.
- Model averaging.
- Neurons dropout.

Underfitting Model too simple and unable to capture features of the training data. Underfitting

2.2 Backpropagation

Chain rule Refer to `SMM for AI (Section 5.1.1)`. Chain rule

Backpropagation Algorithm to compute the gradient at each layer of a neural network. Backpropagation

The output of the i-th neuron in the layer l of a neural network can be defined as:

$$a_{l,i} = \sigma_{l,i}(\mathbf{w}_{l,i}^T \mathbf{a}_{l-1} + b_{l,i}) = \sigma_{l,i}(z_{l,i})$$

where:

- $a_{l,i} \in \mathbb{R}$ is the output of the neuron.
- $\mathbf{w}_{l,i} \in \mathbb{R}^{n_{l-1}}$ is the vector of weights.
- $\mathbf{a}_{l-1} \in \mathbb{R}^{n_{l-1}}$ is the vector of the outputs of the previous layer.
- $b_{l,i} \in \mathbb{R}$ is the bias.
- $\sigma_{l,i} : \mathbb{R} \to \mathbb{R}$ is the activation function[1].
- $z_{l,i}(\mathbf{w}_{l,i}, b_{l,i} | \mathbf{a}_{l-1}) = \mathbf{w}_{l,i}^T \mathbf{a}_{l-1} + b_{l,i}$ is the argument of the activation function and is parametrized on $\mathbf{w}_{l,i}$ and $b_{l,i}$.

Hence, the outputs of the l-th layer can be defined as:

$$\mathbf{a}_l = \sigma_l(\boldsymbol{W}_l^T \mathbf{a}_{l-1} + \mathbf{b}_l) = \sigma_l(\mathbf{z}_l(\boldsymbol{W}_l, \mathbf{b}_l | \mathbf{a}_{l-1}))$$

where:

- $\sigma_l : \mathbb{R}^{n_l} \to \mathbb{R}^{n_l}$ is the element-wise activation function.
- $\boldsymbol{W}_l \in \mathbb{R}^{n_l \times n_{l-1}}$, $\mathbf{a}_{l-1} \in \mathbb{R}^{n_{l-1}}$, $\mathbf{b}_l \in \mathbb{R}^{n_l}$, $\mathbf{a}_l \in \mathbb{R}^{n_l}$.

Finally, a neural network with input $\mathbf{x}$ can be expressed as:

$$\begin{aligned} \mathbf{a}_0 &= \mathbf{x} \\ \mathbf{a}_i &= \sigma_i(\mathbf{z}_i(\boldsymbol{W}_i, \mathbf{b}_i | \mathbf{a}_{i-1})) \end{aligned}$$

Given a neural network with K layers and a loss function $\mathcal{L}$, we want to compute the derivative of $\mathcal{L}$ w.r.t. the weights of each layer to tune the parameters.

First, we highlight the parameters of each of the functions involved:

Loss $\mathcal{L}(a_K) = \mathcal{L}(\sigma_K)$ takes as input the output of the network (i.e. the output of the last activation function).

Activation function $\sigma_i(\mathbf{z}_i)$ takes as input the value of the neurons at the i-th layer.

Neurons $\mathbf{z}_i(\boldsymbol{W}_i, \mathbf{b}_i)$ takes as input the weights and biases at the i-th layer.

Let $\odot$ be the Hadamard product. By exploiting the chain rule, we can compute the derivatives w.r.t. the weights going backward:

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}_K} = \frac{\partial \mathcal{L}}{\partial \sigma_K} \frac{\partial \sigma_K}{\partial \mathbf{z}_K} \frac{\partial \mathbf{z}_K}{\partial \boldsymbol{W}_K} = \underbrace{\nabla \mathcal{L}(\mathbf{a}_K)}_{\mathbb{R}^{n_K \times 1}} \odot \underbrace{\nabla \sigma_K(\mathbf{z}_K)}_{\mathbb{R}^{n_K \times 1}} \cdot \underbrace{\mathbf{a}_{K-1}^T}_{1 \times \mathbb{R}^{n_{K-1}}} \in \mathbb{R}^{n_K \times n_{K-1}}$$

[1] Even if it is possible to have a different activation function in each neuron, in practice, each layer has the same activation function.

$$
\begin{aligned}
\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}_{K-1}} &= \frac{\partial \mathcal{L}}{\partial \sigma_K} \frac{\partial \sigma_K}{\partial \mathbf{z}_K} \frac{\partial \mathbf{z}_K}{\partial \sigma_{K-1}} \frac{\partial \sigma_{K-1}}{\partial \mathbf{z}_{K-1}} \frac{\partial \mathbf{z}_{K-1}}{\partial \boldsymbol{W}_{K-1}} \\
&= (\underset{\mathbb{R}^{n_K \times 1}}{\nabla \mathcal{L}(\mathbf{a}_K)} \odot \underset{\mathbb{R}^{n_K \times 1}}{\nabla \sigma_K(\mathbf{z}_K)})^T \cdot \underset{\mathbb{R}^{n_K} \times \mathbb{R}^{n_{K-1}}}{\boldsymbol{W}_K} \odot \underset{\mathbb{R}^{n_{K-1}} \times 1}{\nabla \sigma_{K-1}(\mathbf{z}_{K-1})} \cdot \underset{1 \times \mathbb{R}^{n_{K-2}}}{\mathbf{a}_{K-2}^T} \in \mathbb{R}^{n_{K-1} \times n_{K-2}}
\end{aligned}
$$

$$\vdots$$

In the same way, we can compute the derivatives w.r.t. the biases:

$$
\frac{\partial \mathcal{L}}{\partial \mathbf{b}_K} = \frac{\partial \mathcal{L}}{\partial \sigma_K} \frac{\partial \sigma_K}{\partial \mathbf{z}_K} \frac{\partial \mathbf{z}_K}{\partial \mathbf{b}_K} = \underset{\mathbb{R}^{n_K \times 1}}{\nabla \mathcal{L}(\mathbf{a}_K)} \odot \underset{\mathbb{R}^{n_K \times 1}}{\nabla \sigma_K(\mathbf{z}_K)} \cdot 1 \in \mathbb{R}^{n_K}
$$

$$
\begin{aligned}
\frac{\partial \mathcal{L}}{\partial \mathbf{b}_{K-1}} &= \frac{\partial \mathcal{L}}{\partial \sigma_K} \frac{\partial \sigma_K}{\partial \mathbf{z}_K} \frac{\partial \mathbf{z}_K}{\partial \sigma_{K-1}} \frac{\partial \sigma_{K-1}}{\partial \mathbf{z}_{K-1}} \frac{\partial \mathbf{z}_{K-1}}{\partial \mathbf{b}_{K-1}} \\
&= (\underset{\mathbb{R}^{n_K \times 1}}{\nabla \mathcal{L}(\mathbf{a}_K)} \odot \underset{\mathbb{R}^{n_K \times 1}}{\nabla \sigma_K(\mathbf{z}_K)})^T \cdot \underset{\mathbb{R}^{n_K} \times \mathbb{R}^{n_{K-1}}}{\boldsymbol{W}_K} \odot \underset{\mathbb{R}^{n_{K-1}} \times 1}{\nabla \sigma_{K-1}(\mathbf{z}_{K-1})} \cdot 1 \in \mathbb{R}^{n_{K-1}}
\end{aligned}
$$

$$\vdots$$

It can be noticed that many terms are repeated from one layer to another. By exploiting this, we can store the following intermediate values:

$$
\begin{aligned}
\delta_K &= \frac{\partial \mathcal{L}}{\partial \mathbf{z}_K} = \frac{\partial \mathcal{L}}{\partial \sigma_K} \frac{\partial \sigma_K}{\partial \mathbf{z}_K} = \nabla \mathcal{L}(\mathbf{a}_K) \odot \nabla \sigma_K(\mathbf{z}_K) \\
\delta_l &= \frac{\partial \mathcal{L}}{\partial \mathbf{z}_l} = \delta_{l+1}^T \cdot \boldsymbol{W}_{l+1} \odot \nabla \sigma_l(\mathbf{z}_l)
\end{aligned}
$$

and reused them to compute the derivatives as follows:

$$
\begin{aligned}
\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}_l} &= \frac{\partial \mathcal{L}}{\partial \mathbf{z}_l} \frac{\partial \mathbf{z}_l}{\partial \boldsymbol{W}_l} = \delta_l \cdot \mathbf{a}_{l-1}^T \\
\frac{\partial \mathcal{L}}{\partial \mathbf{b}_l} &= \frac{\partial \mathcal{L}}{\partial \mathbf{z}_l} \frac{\partial \mathbf{z}_l}{\partial \mathbf{b}_l} = \delta_l \cdot 1
\end{aligned}
$$

Vanishing gradient As backpropagation consists of a chain of products, when a component is small (i.e. < 1), it will gradually cancel out the gradient when backtracking, causing the first layers to learn much slower than the last layers. Vanishing gradient

Remark. This is an issue of the sigmoid function. ReLU was designed to solve this problem.

3 Computer vision

3.1 Convolutions

Convolution neuron Neuron influenced by only a subset of neurons in the previous layer. Convolution neuron

Receptive field Dimension of the input image influencing a neuron. Receptive field

Convolutional layer Layer composed of convolutional neurons. Neurons in the same convolutional layer share the same weights and work as a convolutional filter. Convolutional layer

Remark. The weights of the filters are learned.

A convolutional layer has the following parameters:

- **Kernel size** Dimension (i.e. width and height) of the filter. Kernel size
- **Stride** Offset between each filter application (i.e. stride > 1 reduces the size of the output image). Stride
- **Padding** Artificial enlargement of the image. Padding

  In practice, there are two modes of padding:

  - **Valid** No padding applied.
  - **Same** Apply the minimum padding needed.
- **Depth** Number of different kernels to apply (i.e. augment the number of channels in the output image). Depth

The dimension along each axis of the output image is given by:

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

where:

- W is the size of the image (width or height).
- P is the padding.
- K is the kernel size.
- S is the stride.

Remark. If not specified, a kernel is applied to all the channels of the input image in parallel (but the weights of the kernel change at each channel).

3.1.1 Parameters

The number of parameters of a convolutional layer is given by:

$$(K_\text{w} \cdot K_\text{h}) \cdot D_\text{in} \cdot D_\text{out} + D_\text{out}$$

where:

- K_w is the width of the kernel.

- K_h is the height of the kernel.
- D_in is the input depth.
- D_out is the output depth.

Therefore, the number of FLOPS is of order:

$$(K_\text{w} \cdot K_\text{h}) \cdot D_\text{in} \cdot D_\text{out} \cdot (O_\text{w} \cdot O_\text{h})$$

where:

- O_w is the width of the output image.
- O_h is the height of the output image.

3.2 Backpropagation

A convolution can be expressed as a dense layer by representing it through a sparse matrix. Therefore, backpropagation can be executed in the standard way, with the only exception that the positions of the convolution matrix corresponding to the same cell of the kernel should be updated with the same value (e.g. the mean of all the corresponding updates).

Example. Given a 4×4 image I and a 3×3 kernel K with stride 1 and no padding:

$$I = \begin{pmatrix} i_{0,0} & i_{0,1} & i_{0,2} & i_{0,3} \\ i_{1,0} & i_{1,1} & i_{1,2} & i_{1,3} \\ i_{2,0} & i_{2,1} & i_{2,2} & i_{2,3} \\ i_{3,0} & i_{3,1} & i_{3,2} & i_{3,3} \end{pmatrix} \qquad K = \begin{pmatrix} w_{0,0} & w_{0,1} & w_{0,2} \\ w_{1,0} & w_{1,1} & w_{1,2} \\ w_{2,0} & w_{2,1} & w_{2,2} \end{pmatrix}$$

The convolutional layer can be represented through a convolutional matrix and by flattening the image as follows:

$$\begin{pmatrix} w_{0,0} & 0 & 0 & 0 \\ w_{0,1} & w_{0,0} & 0 & 0 \\ w_{0,2} & w_{0,1} & 0 & 0 \\ 0 & w_{0,2} & 0 & 0 \\ w_{1,0} & 0 & w_{0,0} & 0 \\ w_{1,1} & w_{1,0} & w_{0,1} & w_{0,0} \\ w_{1,2} & w_{1,1} & w_{0,2} & w_{0,1} \\ 0 & w_{1,2} & 0 & w_{0,2} \\ w_{2,0} & 0 & w_{1,0} & 0 \\ w_{2,1} & w_{2,0} & w_{1,1} & w_{1,0} \\ w_{2,2} & w_{2,1} & w_{1,2} & w_{1,1} \\ 0 & w_{2,2} & 0 & w_{1,2} \\ 0 & 0 & w_{2,0} & 0 \\ 0 & 0 & w_{2,1} & w_{2,0} \\ 0 & 0 & w_{2,2} & w_{2,1} \\ 0 & 0 & 0 & w_{2,2} \end{pmatrix}^T \cdot \begin{pmatrix} i_{0,0} \\ i_{0,1} \\ i_{0,2} \\ i_{0,3} \\ i_{1,0} \\ i_{1,1} \\ i_{1,2} \\ i_{1,3} \\ i_{2,0} \\ i_{2,1} \\ i_{2,2} \\ i_{2,3} \\ i_{3,0} \\ i_{3,1} \\ i_{3,2} \\ i_{3,3} \end{pmatrix} = \begin{pmatrix} o_{0,0} \\ o_{0,1} \\ o_{1,0} \\ o_{1,1} \end{pmatrix} \mapsto \begin{pmatrix} o_{0,0} & o_{0,1} \\ o_{1,0} & o_{1,1} \end{pmatrix}$$

3.3 Pooling layer

Pooling Layer that applies a function as a filter.

Max-pooling Filter that computes the maximum of the pixels within the kernel. Max-pooling

Mean-pooling Filter that computes the average of the pixels within the kernel. Mean-pooling

3.4 Inception hypothesis

Depth-wise separable convolution Decompose a 3D kernel into a 2D kernel followed by a 1D kernel.

Depth-wise separable convolution

Given an input image with C_{in} channels, a single pass of a traditional 3D convolution uses a kernel of shape $k \times k \times C_{\text{in}}$ to obtain an output of 1 channel. This is repeated for a desired C_{out} number of times (with different kernels).

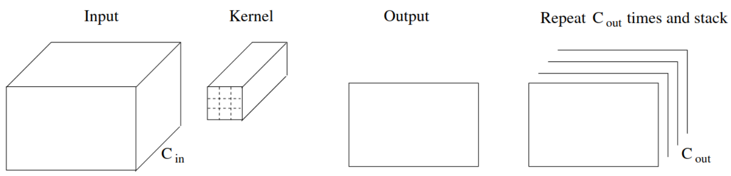


Figure 3.1: Example of traditional convolution

A single pass of a depth-wise separable convolution uses C_{in} different $k \times k \times 1$ kernels first to obtain C_{in} images. Then, a $1 \times 1 \times C_{\text{in}}$ kernel is used to obtain an output image of 1 channel. The last 1D kernel is repeated for a C_{out} number of times (with different kernels).

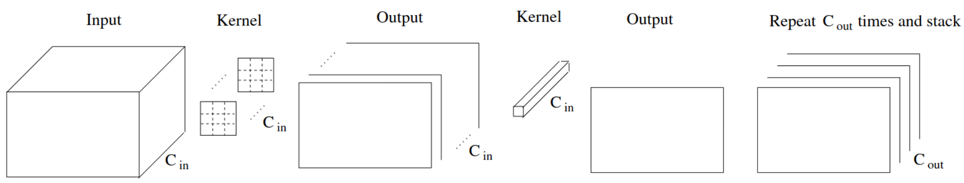


Figure 3.2: Example of depth-wise separable convolution

3.4.1 Parameters

The number of parameters of a depth-wise separable convolutional layer is given by:

$$(K_{\text{w}} \cdot K_{\text{h}}) \cdot D_{\text{in}} + (1 \cdot 1 \cdot D_{\text{in}}) \cdot D_{\text{out}}$$

where:

- K_{w} is the width of the kernel.
- K_{h} is the height of the kernel.
- D_{in} is the input depth.
- D_{out} is the output depth.

3.5 Residual learning

Residual connection Sum the input of a layer to its output.

Residual connection

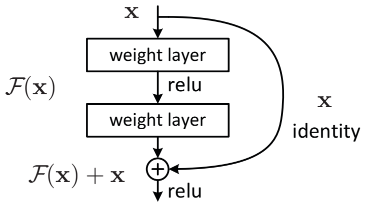


Figure 3.3: Residual connection

Remark. The sum operation can be substituted with the concatenation.

Remark. The effectiveness of residual connections is only shown empirically.

Remark. By adding the input, without passing through the activation function, might help to propagate the gradient from higher layers to lower layers and avoid the risk of vanishing gradient.

Another interpretation is that, by learning the function $F(x) + x$, it is easier for the model to represent, if it needs to, the identity function as the problem is reduced to learn $F(x) = 0$. On the other hand, without a residual connection, learning $F(x) = x$ from scratch might be harder.

3.6 Transfer learning and fine-tuning

Transfer learning Reuse an existing model by appending some new layers to it. Only the new layers are trained. Transfer learning

Fine-tuning Reuse an existing model by appending some new layers to it. The existing model (or part of it) is trained alongside the new layers. Fine-tuning

Remark. In computer vision, reusing an existing model makes sense as the first convolutional layers tend to learn primitive concepts that are independent of the downstream task.

3.7 Other types of convolution

Transposed convolution / Deconvolution Convolution to upsample the input (i.e. each pixel is upsampled into a $k \times k$ patch). Transposed convolution / Deconvolution

Remark. A transposed convolution can be interpreted as a normal convolution with stride < 1.

Dilated convolution Convolution computed using a kernel that does not consider contiguous pixels. Dilated convolution

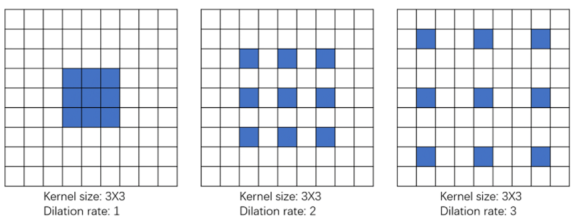


Figure 3.4: Examples of dilated convolutions

Remark. Dilated convolutions allow the enlargement of the receptive field without an excessive number of parameters.

Remark. Dilated convolutions are useful in the first layers when processing high-resolution images (e.g. temporal convolutional networks).

3.8 Normalization layer

A normalization layer has the empirical effects of:

- Stabilizing and possibly speeding up the training phase.
- Increasing the independence of each layer (i.e. maintain a similar magnitude of the weights at each layer).

Batch normalization Given an input batch X, a batch normalization layer outputs the following:

Batch normalization

$$\gamma \frac{X - \mu}{\sqrt{\sigma^2 + \varepsilon}} + \beta$$

where:

- γ and β are learned parameters.
- ε is a small constant.
- μ is the mean and σ^2 is the variance. Depending on when the layer is applied, these values change:

  Training μ and σ^2 are computed from the input batch X.

  Inference μ and σ^2 are computed from the training data. Usually, it is obtained as the moving average of the values computed from the batches during training.

3.9 Gradient ascent

3.9.1 Hidden layer visualization

Visualize what type of input features activate a neuron.

Hidden layer visualization

Image ascent approach During training, the loss function of a neural network $\mathcal{L}(\mathbf{x}; \boldsymbol{\theta})$ is parametrized on the weights $\boldsymbol{\theta}$ while the input $\mathbf{x}$ is fixed.

To visualize the patterns that activate a (convolutional) neuron, it is possible to invert the optimization process by fixing the parameters $\boldsymbol{\theta}$ and optimizing an image $\mathbf{x}$ so that the loss function becomes $\mathcal{L}(\boldsymbol{\theta}; \mathbf{x})$. The process works as follows:

1. Start with a random image $\mathbf{x}$.
2. Do a forward pass with $\mathbf{x}$ as input and keep track of the activation function $a_i(\mathbf{x})$ of the neuron(s) of interest.
3. Do a backward pass to compute the gradient $\frac{\partial a_i(\mathbf{x})}{\partial \mathbf{x}_{i,j}}$ (i.e. chain rule) for each pixel (i, j) of the image.
4. Update the image as $\mathbf{x} = \mathbf{x} + \eta \frac{\partial a_i(\mathbf{x})}{\partial \mathbf{x}}$.
5. Repeat until the activation function $a_i(\mathbf{x})$ is high enough.

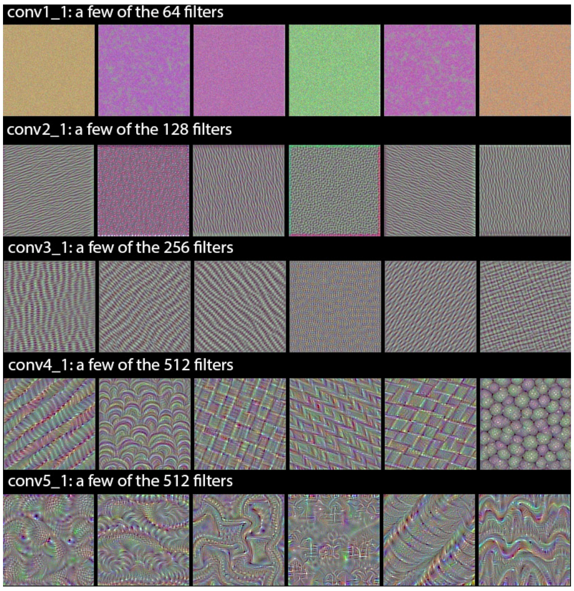


Figure 3.5: Example of generative image ascent visualization approach

Generative approach Starting from an image $\hat{\mathbf{x}}$ that makes a specific layer l output $\Theta_l(\hat{\mathbf{x}})$, generate another image $\mathbf{x}$ that makes the same layer l output a similar value $\Theta_l(\mathbf{x}) \approx \Theta_l(\hat{\mathbf{x}})$ (i.e. it cannot distinguish between $\mathbf{x}$ and $\hat{\mathbf{x}}$).

Fixed $\hat{\mathbf{x}}$, the problem can be solved as an optimization problem:

$$\arg\min_{\mathbf{x}} \left\{ l\big(\Theta_l(\mathbf{x}), \Theta_l(\hat{\mathbf{x}})\big) + \lambda \mathcal{R}(\mathbf{x}) \right\}$$

where l is a loss function to measure the distance between the two representations and $\mathcal{R}$ is a regularizer.

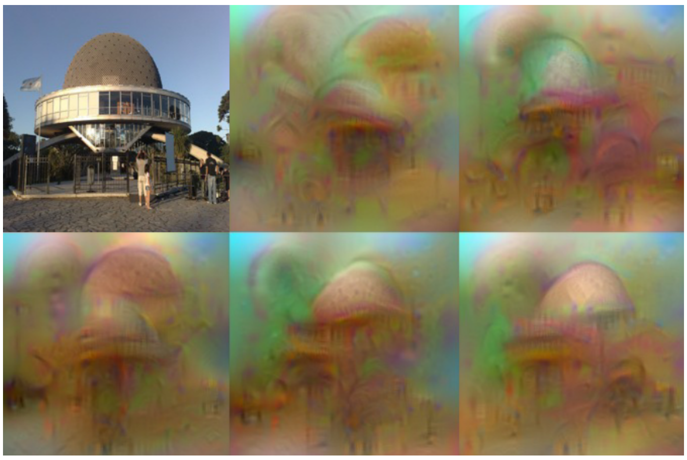


Figure 3.6: Example of generative visualization approach

3.9.2 Inceptionism

Employ the same techniques for hidden layer visualization to create psychedelic and abstract images. Inceptionism

Deep dream Iteratively apply gradient ascent on an image: Deep dream

1. Train a neural network for image classification.
2. Repeatedly modify an input image using gradient ascent to improve the activation of a specific neuron.

After enough iterations, the features that the target neuron learned to recognize during training are injected into the input image, even if that image does not have that specific feature.

Remark. Strong regularizers are used to prioritize features that statistically resemble real images.

Content enhancing Same as above, but instead of selecting a neuron, an entire layer is fixed and the input image is injected with whatever that layer detects. Content enhancing

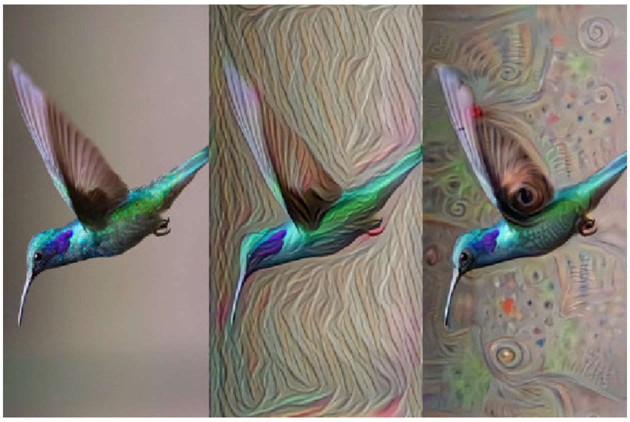


Figure 3.7: Example of deep dream images

3.9.3 Style transfer

Mimic the style of an image and transfer it to the content of another one. Style transfer

Internal representation approach Given a convolutional neural network pretrained for classification, the method can be divided into two parts:

Content reconstruction Given an image $\hat{\mathbf{x}}$, consider the output of the l-th layer of the network. Its internal representation of the image has C^l distinct channels (depending on the number of kernels) each with $M^l = W^l \cdot H^l$ elements (when flattened).

The representation (feature map) of the l-th layer can therefore be denoted as $F^l \in \mathbb{R}^{C^l \times M^l}$ and $F^l_{c,k}$ is used to denote the activation of the c-th filter applied at position k of the l-th layer.

As higher layers of a CNN capture high-level features, one of the high layers is selected and its feature map is used as the content representation.

Given a content representation $\mathcal{C} = \hat{F}^l$ of $\hat{\mathbf{x}}$, chosen as the feature map at the l-th layer, it is possible to reconstruct the original image $\hat{\mathbf{x}}$ starting from a random one $\mathbf{x}$ by minimizing the loss:

$$\mathcal{L}_{\text{content}}(\hat{\mathbf{x}}, \mathbf{x}, l) = \sum_{c,i} (F^l_{c,i} - \mathcal{C}_{c,i})^2$$

where F^l is the feature representation of the random image $\mathbf{x}$.

Style reconstruction Given an image $\hat{\mathbf{y}}$ and its feature maps F^l for $l \in \{1, \dots L\}$, at each layer l, the Gram matrix $G^l \in \mathbb{R}^{C^l \times C^l}$ obtained as the dot product between pairs of channels (i.e. correlation between features extracted by different kernels):

$$G^l_{c_1,c_2} = F^l_{c_1} \odot F^l_{c_2} = \sum_k (F^l_{c_1,k} \cdot F^l_{c_2,k})$$

allows to capture the concept of style.

The Gram matrices at each layer are considered as the style representation.

Given the style representation $\mathcal{S}^1, \dots, \mathcal{S}^L$ of $\hat{\mathbf{y}}$, it is possible to reconstruct the same style of the original image $\hat{\mathbf{y}}$ starting from a random image $\mathbf{y}$ by minimizing the loss:

$$\mathcal{L}_{\text{style}}(\hat{\mathbf{y}}, \mathbf{y}) = \sum_{l=1}^{L} \gamma_l \left(\sum_{i,j} (G^l_{i,j} - \mathcal{S}^l_{i,j})^2 \right)$$

where γ_l is a weight assigned to each layer and G^l is the l-th Gram matrix of the random image $\mathbf{y}$.

Put together, given:

- An image $\hat{\mathbf{x}}$ from which the content has to be copied.
- An image $\hat{\mathbf{y}}$ from which the style has to be copied.
- The content representation $\mathcal{C}$ of $\hat{\mathbf{x}}$.
- The style representation $\mathcal{S}^1, \dots, \mathcal{S}^L$ of $\hat{\mathbf{y}}$.

A new random image $\mathbf{o}$ is fitted by minimizing the loss:

$$\mathcal{L}_{\text{total}} = \alpha \mathcal{L}_{\text{content}}(\hat{\mathbf{x}}, \mathbf{o}, l) + \beta \mathcal{L}_{\text{style}}(\hat{\mathbf{y}}, \mathbf{o})$$

where α and β are hyperparameters.

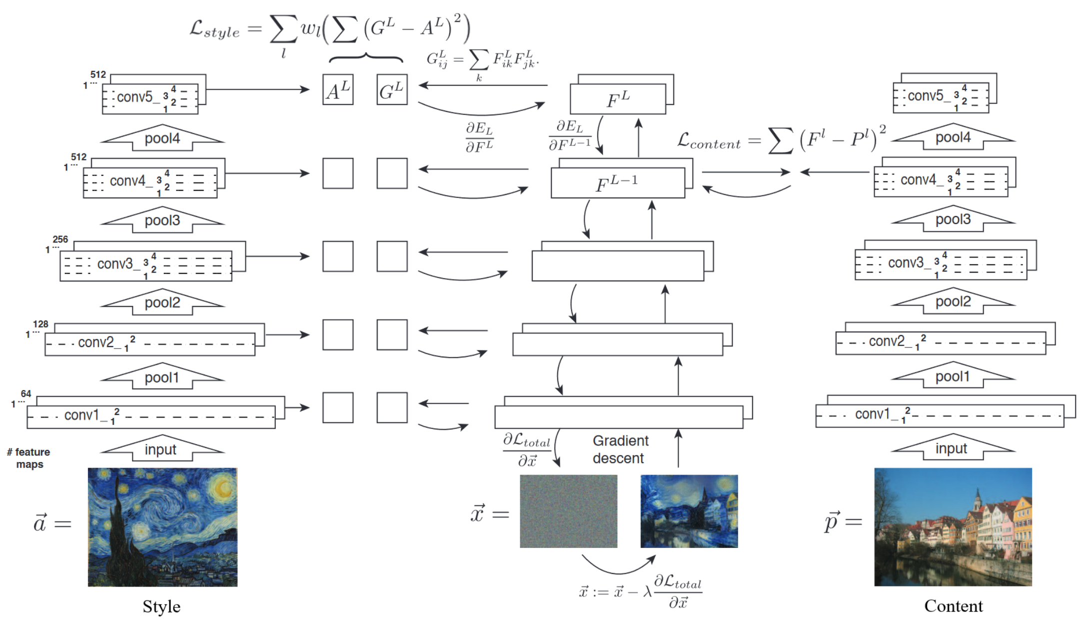


Figure 3.8: Internal representation style transfer workflow

Perceptual loss approach A CNN pretrained for classification is used as a loss network to compute perceptual loss functions to measure the difference in style and content between images. The representation for style and content is extracted in a similar way as above.

The loss network is then kept fixed and an image transformation network is trained to transform its input $\mathbf{x}$ into an image $\mathbf{y}$ compliant (i.e. minimizes the perceptual losses) with a given style image $\mathbf{y}_s$ and a content image $\mathbf{y}_c$ (if the goal is to keep the content of the input, then $\mathbf{y}_c = \mathbf{x}$).

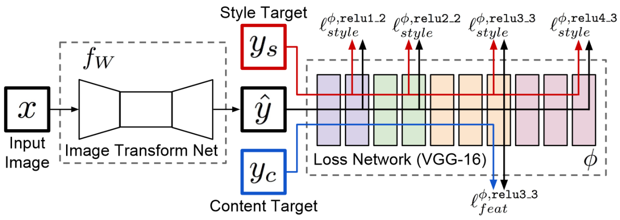


Figure 3.9: Perceptual loss style transfer workflow

3.10 Data manifold

3.10.1 Adversarial attacks

Hijack a neural network classifier to forcefully predict a given class. Adversarial attacks

Gradient ascent approach White-box technique that uses gradient ascent to compute an image that the network classifies with the wanted class.

Let:

- $\mathbf{x}$ be the input image.
- $f(\mathbf{x})$ the probability distribution that the network outputs.
- c the wanted class.
- p the wanted probability distribution (i.e. $p_c = 1$ and $p_i = 0$ elsewhere).
- $\mathcal{L}$ the loss function.

By iteratively updating the input image with the gradient of the loss function $\frac{\partial \mathcal{L}(f(\mathbf{x}),p)}{\partial \mathbf{x}}$ computed wrt to $\mathbf{x}$, after enough iterations, the classifier will classify the updated $\mathbf{x}$ as c.

Remark. The updates computed from the gradient of the loss function are usually imperceptible.

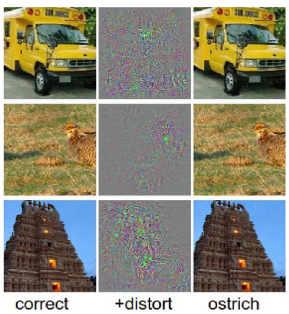


Figure 3.10: Examples of hijacked classifications

Evolutionary approach Black-box technique based on an evolutionary approach.

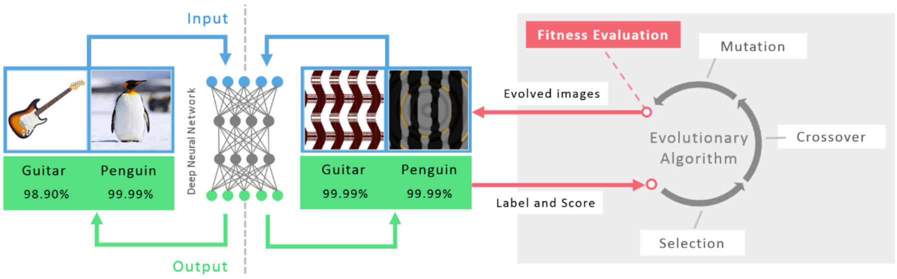


Figure 3.11: Workflow for evolutionary-based attacks

3.10.2 Manifold

Manifold Area of the feature space that represents "natural" images (i.e. images with a meaning an without artificial noise). Manifold

This area is usually organized along a smooth surface which is a minimal portion of the entire space of all the possible images.

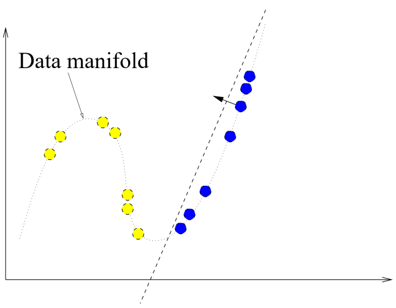


Figure 3.12: Example of manifold in two dimensions

Remark. As one cannot know where the classifier draws the boundaries, a tiny change in the data might cause a misclassification.
Adversarial attacks also exploit this to cause misclassifications.

Remark. Inceptionism aims to modify the data while remaining in the manifold.

3.10.3 Autoencoders

Network composed of two components: Autoencoder

Encoder Projects the input into an internal representation of lower dimensionality.

Decoder Reconstructs the input from its internal representation.

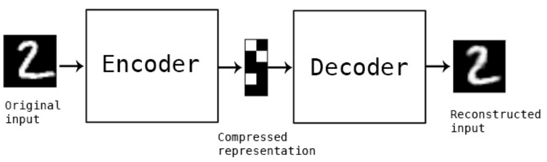


Figure 3.13: Autoencoder structure

An autoencoder has the following properties:

Data-specific It only works on data with a strong correlation (i.e. with regularities in the feature space).

Lossy By passing through the internal representation, the reconstruction of the input is nearly always degraded.

Self-supervised Training happens directly on unlabelled data.

Applications of autoencoders are:

Denoising Train the autoencoder to reconstruct noiseless data. Given an image, the input is a noisy version of it, while the output is expected to be similar to the original image.

Anomaly detection As autoencoders are data-specific, they will perform poorly on data different from those used for training.

This allows to detect anomalies by comparing the quality of the reconstruction. If the input is substantially different from the training data (or has been attacked with an artificial manipulation), the reconstructed output is expected to have poor quality.

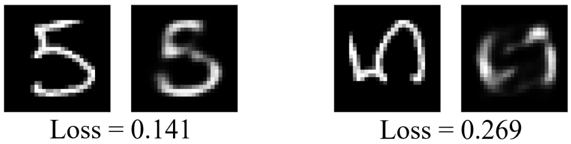


Figure 3.14: Example of anomaly detection

3.11 Segmentation

Semantic segmentation Classify the pixels of an image depending on the category it belongs to. Semantic segmentation

Remark. Creating a dataset for segmentation is expensive.

Figure 3.15: Example of semantic segmentation

3.11.1 Convolutionalization

Given a pre-trained image classification network, it can be adapted into a segmentation network by converting its final dense layers into convolutions with kernel size 1×1 and depth equal to the number of neurons in that layer. Convolutionalization

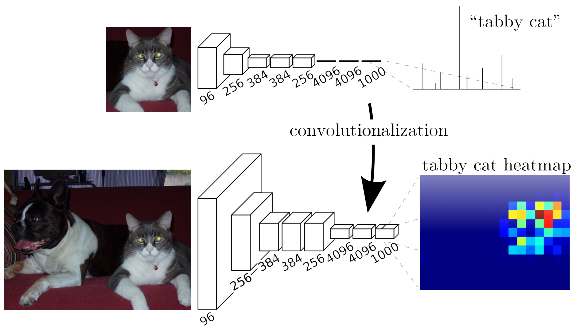


Figure 3.16: Example of convolutionalization

The resulting model has the following behavior:

- It takes as input an image of arbitrary shape. This is possible as the network is composed of only convolutions (i.e. it can be seen as a single big convolution).
- It outputs a heatmap of activations of the different object classes (i.e. the categories of the pre-trained classification network).

As the output is obtained through a series of convolutions, its shape does not match the input image. Therefore, the initial output heatmap needs to be upsampled by using transposed convolutions.
To avoid losing information from previous layers, the original work proposes to use skip connections before upsampling.

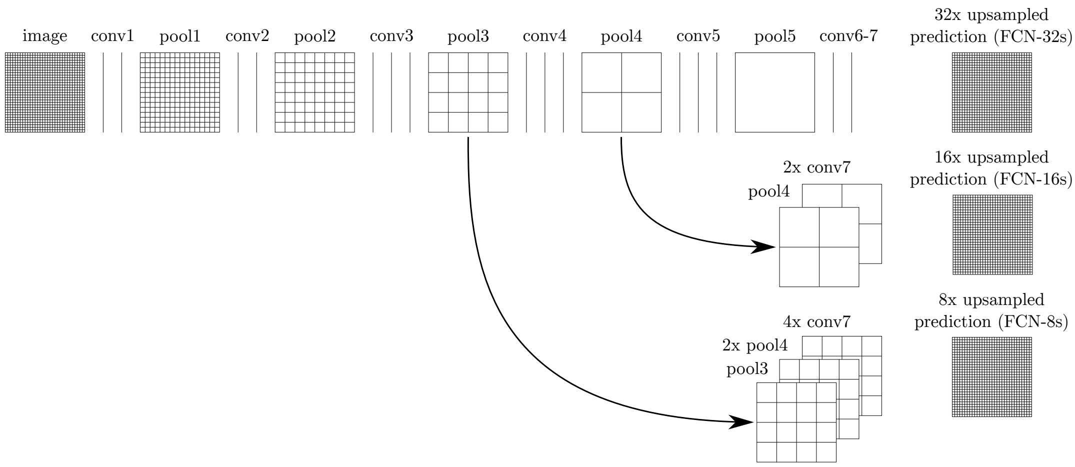


Figure 3.17: Examples of upsampling. The first row shows the upsampling process of the output (`conv7`) without skip connections. The second row shows the upsampling process with a skip connection from the second last pooling layer (`pool4`): the output (`conv7`) is partially upsampled to match the shape of the skip connectionm, then upsampling is done on their concatenation. The third row shows the upsampling process with skip connections up to the third last pooling layer (`pool3`).

3.11.2 U-net

Segmentation architecture that does not rely on a pre-trained classification network. The architecture is composed of two steps:

U-net

Downsampling Using convolutions and max-pooling.

Upsampling Using transposed convolutions and skip connections.

Remark. An interpretation of the two operations is the following:

Downsampling Aims to find what the image contains.

Upsampling Aims to find where the found objects are.

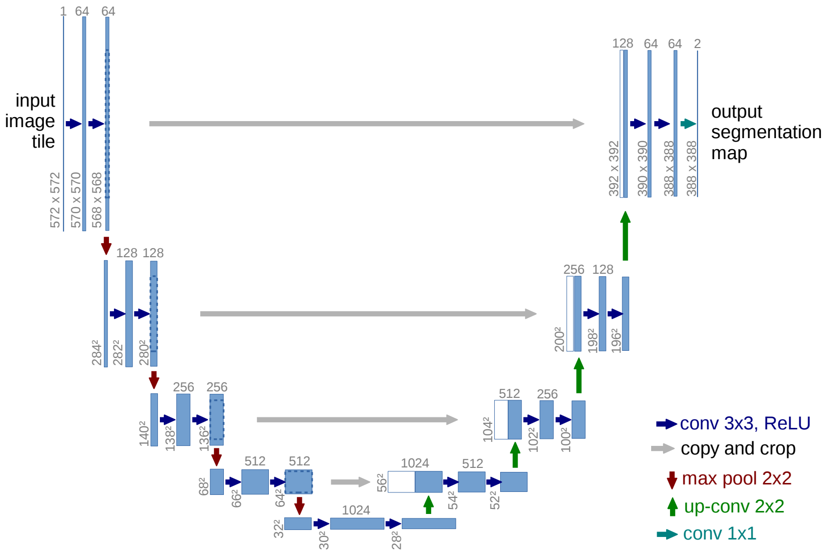


Figure 3.18: Example of U-net architecture without padding

Remark. In the original work, the architecture is defined using cropping and without padding, making the output shape smaller than the input. Segmentation was therefore done on a cropped portion of the input image.
Another approach is to use padding to maintain the same shape of the input in the output.

3.12 Object detection

Intersection over union Metric used to determine the quality of a bounding box w.r.t. a ground truth:

Intersection over union

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

Object detection Find bounding boxes containing a specific object or category.

Object detection

There are two main strategies:

Region proposal Object-independent method that uses selective search algorithms to exploit the texture and the structure of the image to find locations of interest.

Region proposal

Single-shot Fast method oriented towards real-time applications.

Single shot

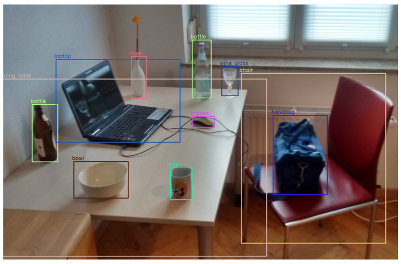


Figure 3.19: Example of bounding boxes

3.12.1 YOLOv3

YOLO is a fully convolutional neural network belonging to the family of single-shot methods.

Anchor box

Anchor box It has been shown that directly predicting the width and height of the bounding boxes leads to unstable gradients during training. A common solution to this problem is to use pre-defined bounding boxes (anchors).

Anchors are selected using k-means clustering on the bounding boxes of the training set using `IoU` as metric (i.e. the most common shapes are identified). Then, the network learns to draw bounding boxes by placing and scaling the anchors.

YOLO architecture

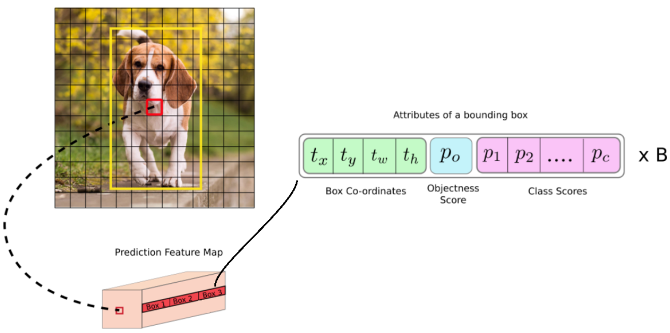
Architecture An input image is progressively downsampled through convolutions by a factor of 2^5 to obtain a feature map of $S \times S$ cells (e.g. a 416×416 image is downsampled into a 13×13 grid).

Each entry of the feature map has a depth of $(B \times (5 + C))$ where:

- B is the number of bounding boxes (one per anchor) the cell proposes.
- C is the number of object classes.

Therefore, each bounding box prediction has associated $5 + C$ attributes:

- t_x and t_y describe the center coordinates of the box (relative to the predicting cell).
- t_w and t_h describe the width and height of the box (relative to the anchor).
- p_o is an objectness score that indicates the probability that an object is contained in the predicted bounding box (useful for thresholding).
- $p_1, \dots, p_C$ are the probabilities associated to each class. Since YOLOv3, the probability of each class is given by a sigmoid instead of passing everything through a softmax. This allows to associate an object with multiple categories.



Inference YOLO inference

Remark. Each cell of the feature map is identified by a set of coordinates relative to the feature map itself (e.g. the first cell is at coordinate $(0,0)$, the one to its right is at $(0,1)$).

Given a cell of the feature map at coordinates (c_x, c_y), consider its i-th bounding box prediction. The bounding box is computed using the following parameters:

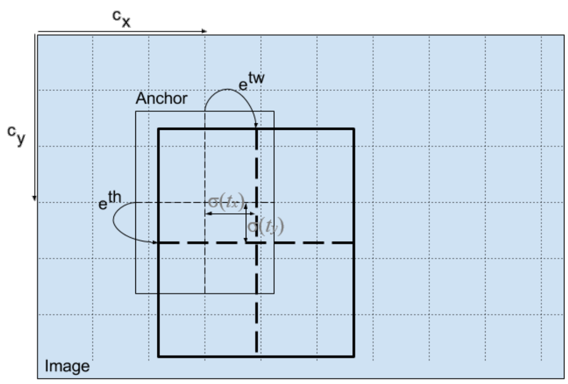
- The predicted relative position and dimension $\langle t_x, t_y, t_w, t_h \rangle$ of the box.
- The width p_w and height p_h of the anchor associated with the i-th prediction of the cell.

Then, the bounding box position and dimension (relative to the feature map) are computed as follows:

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

where:

- (b_x, b_y) are the coordinates of the center of the box.
- b_w and b_h are the width and height of the box.
- σ is the sigmoid function.



Training During training, for each ground truth bounding box, only the cell at its center and the anchor with the highest `IoU` are considered for its prediction. In other words, only that combination of cell and anchor influences the loss function. YOLO training

Given a $S \times S$ feature map and B anchors, for each prediction, YOLO uses two losses:

Localization loss Measures the positioning of the bounding boxes:

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

where:

- $\mathbb{1}_{ij}^\text{obj}$ is a delta function that is 1 if the j-th anchor of the i-th cell is responsible for detecting the object.

- (x_i, y_i) are the predicted coordinates of the box. $(\hat{x}_i, \hat{y}_i)$ are the ground truth coordinates.
- w_i and h_i are the predicted width and height of the box. $\hat{w}_i$ and $\hat{h}_i$ are the ground truth dimensions.
- λ_{coord} is a hyperparameter (the default is 5).

Classification loss Considers the objectness score and the predicted classes:

$$\begin{aligned}\mathcal{L}_{\text{cls}} = &\sum_{i=0}^{S \times S} \sum_{j=0}^{B} (\mathbb{1}_{ij}^{\text{obj}} + \lambda_{\text{no-obj}}(1 - \mathbb{1}_{ij}^{\text{obj}}))(C_{ij} - \hat{C}_{ij})^2 \\ &+ \sum_{i=0}^{S \times S} \sum_{c \in \mathcal{C}} \mathbb{1}_{i}^{\text{obj}} (p_i(c) - \hat{p}_i(c))^2\end{aligned}$$

where:

- $\mathbb{1}_{ij}^{\text{obj}}$ is defined as above.
- $\mathbb{1}_{i}^{\text{obj}}$ is 1 if the i-th cell is responsible for classifying the object.
- C_{ij} is the predicted objectness score. $\hat{C}_{ij}$ is the ground truth.
- $p_i(c)$ is the predicted probability of belonging to class c. $\hat{p}_i(c)$ is the ground truth.
- $\lambda_{\text{no-obj}}$ is a hyperparameter (the default is 0.5). It is useful to down-weight cells that are not responsible for detecting this specific instance.

The final loss is the sum of the two losses:

$$\mathcal{L} = \mathcal{L}_{\text{loc}} + \mathcal{L}_{\text{cls}}$$

3.12.2 Multi-scale processing

Feature pyramid Techniques to manipulate the input image to detect objects at different scales. Feature pyramid

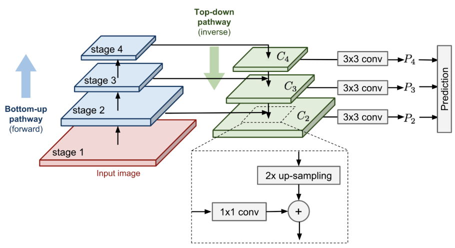
Possible approaches are:

Featurized image pyramid A pyramid of images at different scales is built. The features at each scale are computed independently (which makes this approach slow).

Single feature map Progressively extract features from a single image and only use features at the highest level.

Pyramidal feature hierarchy Reuse the hierarchical features extracted by a convolutional network and use them as in the featurized image pyramid approach.

Feature Pyramid Network Progressively extract higher-level features in a forward pass and then inject them back into the previous pyramid layers.



Remark. YOLOv3 predicts feature maps at scales 13, 26 and 52 using a feature pyramid network.

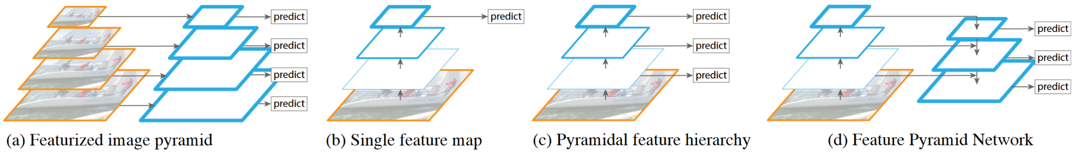


Figure 3.20: Feature pyramid recap

3.12.3 Non-maximum suppression

Non-maximum suppression Method to remove multiple detections of the same object. Non-maximum suppression

Given the bounding boxes BB_c of a class c and a threshold t, NMS does the following:

1. Sort BB_c according to the objectness score.
2. While BB_c is not empty:
    a) Pop the first box p from BB_c.
    b) p is considered as a true prediction.
    c) Remove from BB_c all the boxes s with $\texttt{IoU}(p, s) > t$.

4 Generative models

Generative model Model that tries to learn a probability distribution p_{model} close to that of the data p_{data}.

Generative model

This can be done either by:

- Explicitly estimating the distribution.
- Building a generator to sample data from the distribution p_{model} and possibly providing a likelihood.

Remark. Generative models are suited for problems with multi-modal outputs (i.e. with no unique solution).

Latent variable model Given a vector of values z with known prior distribution $P(z)$, a latent variable model expresses the probability of a data point X (of the visible space) through marginalization over z:

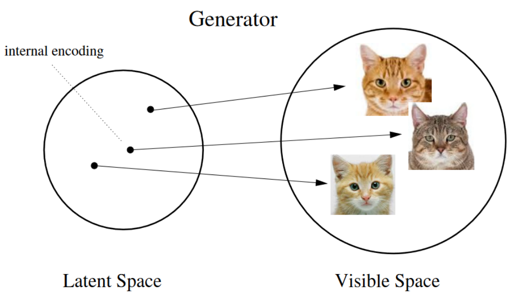
Latent variable model

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

z is considered the latent encoding of X (usually it is some sort of noise).

The network (generator) on input z can either learn:

- The probability $P(X|z)$.
- To generate data points $\hat{X}$ (most likely) belonging to the distribution $P(X|z)$.



Generative models are categorized into two families:

Compressive models Models where the latent space is smaller than the visible space.

Compressive models

Dimension-preserving models Models where the latent space has the same dimension as the visible space.

Dimension-preserving models

Remark. Training latent variable models requires a way to encode the visible training data X into their latent space z. Training relying only on the latent space does not make sense as, in this case, the output of the generator can be arbitrary.

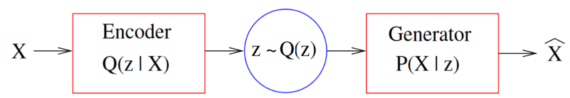
4.1 Variational autoencoder (VAE)

Model belonging to the family of compressive models.

4.1.1 Training

An autoencoder is modified in such a way that:

- The encoder takes as input a visible data point X and outputs its latent encoding z. The encoder is trained to force the marginal distribution of the latent space $Q(z) = \mathbb{E}_{X \sim P_{\text{data}} Q(z|X)}$ into a known distribution (usually a standard Gaussian).
- The decoder (generator) takes as input the latent encoding z of the encoder and outputs a reconstruction $\hat{X}$.



It is assumed that for each different input X, $Q(z|X)$ has a different Gaussian distribution $G(\mu(X), \sigma(X))$ where both $\mu(X)$ and $\sigma(X)$ are computed by the encoder.
$z = \mu(X)$ can be seen as the latent encoding of X, while $\sigma(X)$ represents an area of the latent space around z that encodes an information similar to X.
During training, the decoder receives in input a point sampled around $\mu(X)$ with variance $\sigma(X)$.
Two losses are used:

Reconstruction distance Aims to minimize the distance between the input X and its reconstruction $\hat{X}$:

$$\|X - \hat{X}\|^2$$

Kullback-Leibler divergence Aims to bring the marginal inference distribution $Q(z)$ close to a known distribution (e.g. a standard Gaussian):

$$\text{KL}[Q(z|X)||P(z)] = \text{KL}[Q(z|X)||\mathcal{N}(0, 1)]$$

Remark. The loss is applied to the single distributions $Q(z|X)$ but the effects are propagated to the marginal $Q(z)$.

Remark. An effect of this loss (with the standard Gaussian) is to:

- Push $\mu(X)$ towards 0 (i.e. occupying a small area of the latent space).
- Push $\sigma(X)$ towards 1 (i.e. making the latent variables have a larger coverage space to make it easier to generate new significant data).

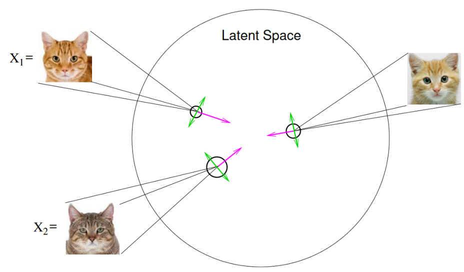


Figure 4.1: Effect of the KL-divergence on the latent space

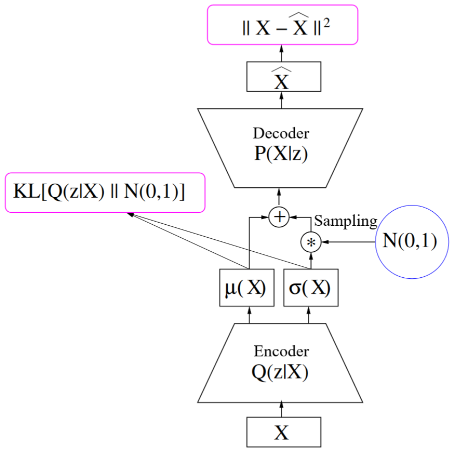


Figure 4.2: Recap of the VAE training process

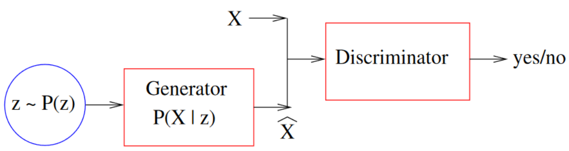
4.1.2 Inference

During inference, the encoder is not used as there are no visible input data X. The decoder generates new data by simply taking as input a latent variable z sampled from its prior distribution (e.g. a standard Gaussian).

4.2 Generative adversarial network (GAN)

Model belonging to the family of compressive models.
During training, the generator is paired with a discriminator that learns to distinguish between real data and generated data.
The loss function aims to:

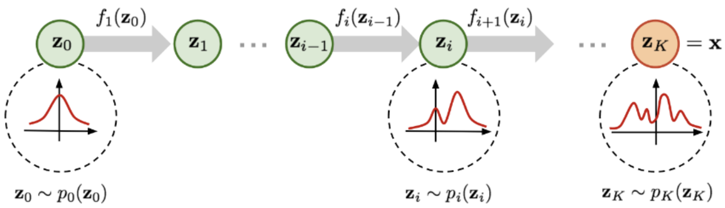
- Instruct the discriminator to spot the generator.
- Instruct the generator to fool the discriminator.



4.3 Normalizing flows

Model belonging to the family of dimension-preserving models.
The generator is split into a chain of invertible transformations. During training, the log-likelihood is maximized.

Remark. Using only invertible transformations limits the expressiveness of the model.



4.4 Diffusion models

Model belonging to the family of dimension-preserving models.
The generator is split into a chain of denoising steps that attempt to remove a Gaussian noise with varying σ. It is assumed that the latent space is a noisy version of the image.