Machine Learning for biology

V. Monbet

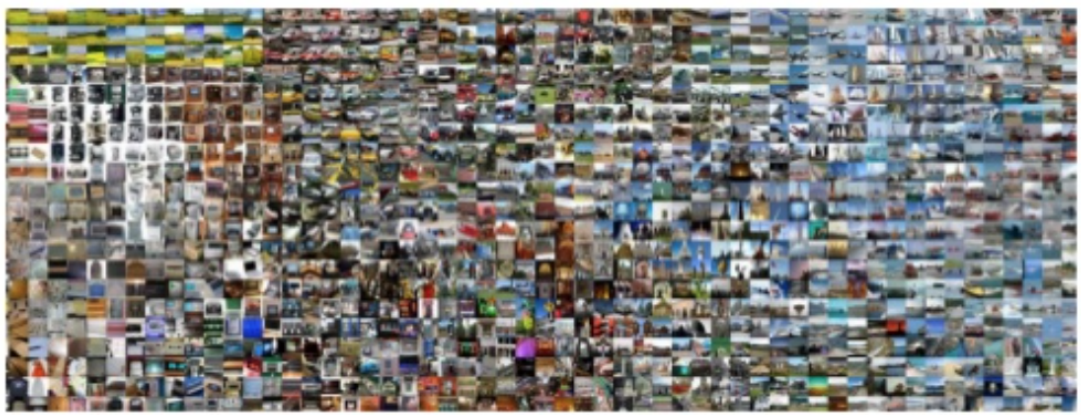
UFR de Mathématiques
Université de Rennes

Outline

Outline

- Deep Learning has experienced significant growth over the past decade. It was initially applied to the classification of very large image datasets.
- In 2012, AlexNet (University of Toronto) won the ImageNet competition (1.2 million images, 1000 labels) with a convolutional neural network-based algorithm. The classification error of the winners was around 15
- In 2013, Clarify ConvNet achieved an error rate of 11

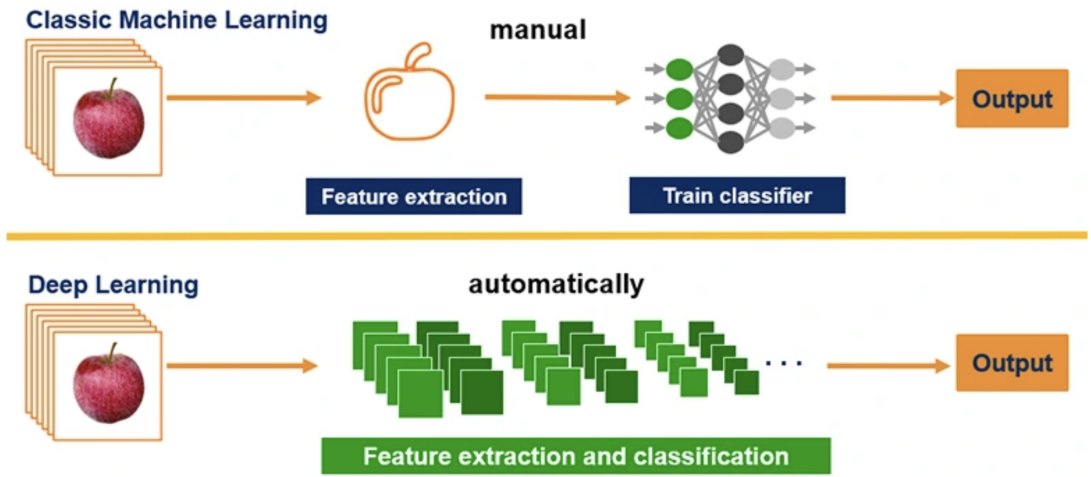
IMAGENET



Source: t-SNE visualization of CNN codes. Andrej Karpathy

- Since then, Deep Learning (or deep neural networks) has been applied to a wide range of problems.

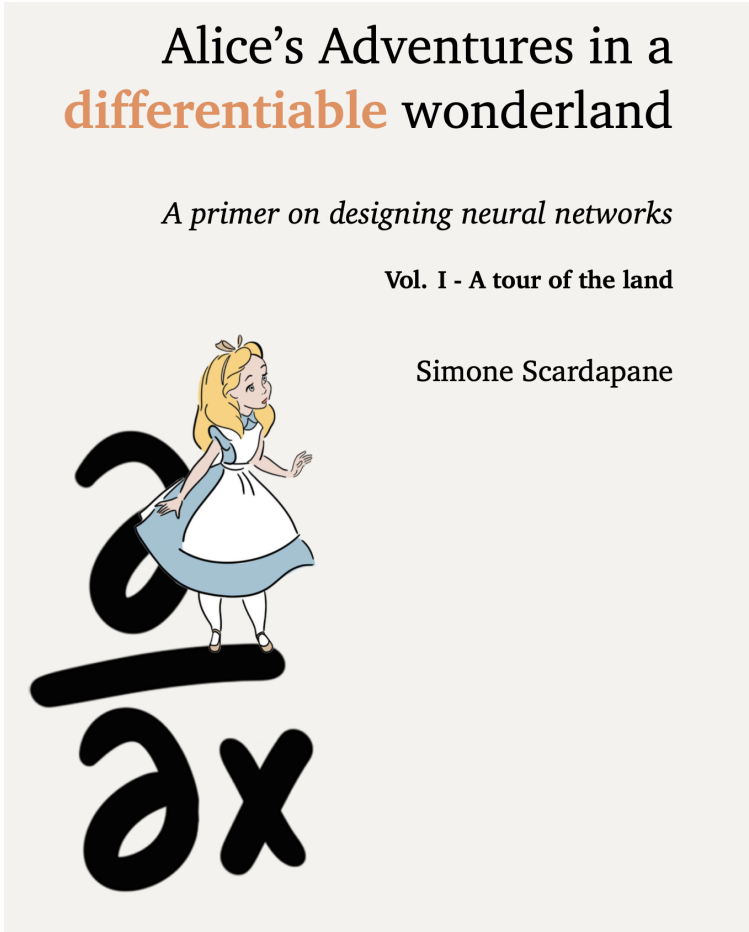
Machine learning vx Deep learning



- In a conventional Machine Learning procedure, when dealing with high-dimensional data :
  1. We extract features from the data (e.g., through Principal Component Analysis or traditional image analysis techniques). For images, we extract information about texture, colors, brightness, etc.
  2. We fit a model that takes these new variables as input.

- In deep learning, the feature extraction phase is included within the model, typically a neural network. However, the network is then (very) deep and contains a (very) large number of parameters. This requires :
  - A very large dataset to ensure that the number of observations remains significantly larger than the number of parameters.
  - A significant amount of computation time for training.

One of my favorite ressources



Available here : `https://arxiv.org/abs/2404.17625`

In the sequel, most of the figures and codes are reproduced from this book.

Outline

Toward Convolutional Neural Networks (CNN)

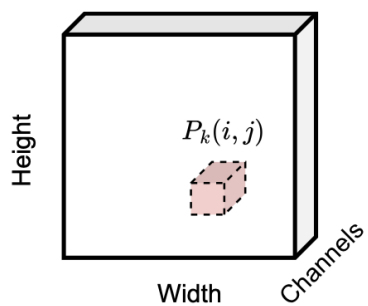
✗ Fully connected networks are not enough

- An image is decribed by a tensor $X \sim (h, w, c)$ where h denoted the height, w the width and c the number of chanels
  One variable = one pixel.
- For using dense neural networks, the image would need to be flatten

$$\mathbf{h} = \Phi(\mathbf{W}.\text{vect}(X))$$

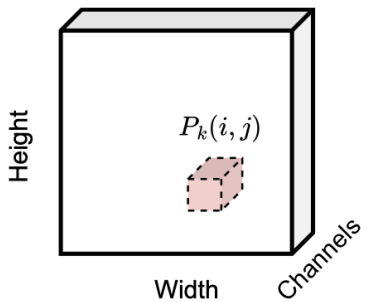
  - ✗ Some proximity properties are losen
  - ✗ It leads to a huge vector of variables, and neaural networks with a huge number of weights...
- CNN helps to preserves the patches

Figure F.7.1: *Given a tensor (h, w, c) and a maximum distance k, the* ***patch*** *$P_k(i, j)$ (shown in red) is a $(2k + 1, 2k + 1, c)$ tensor collecting all pixels at distance at most k from the pixel in position (i, j).*



Tensor/patches

Figure F.7.1: *Given a tensor (h, w, c) and a maximum distance k, the* ***patch*** *$P_k(i, j)$ (shown in red) is a $(2k + 1, 2k + 1, c)$ tensor collecting all pixels at distance at most k from the pixel in position (i, j).*

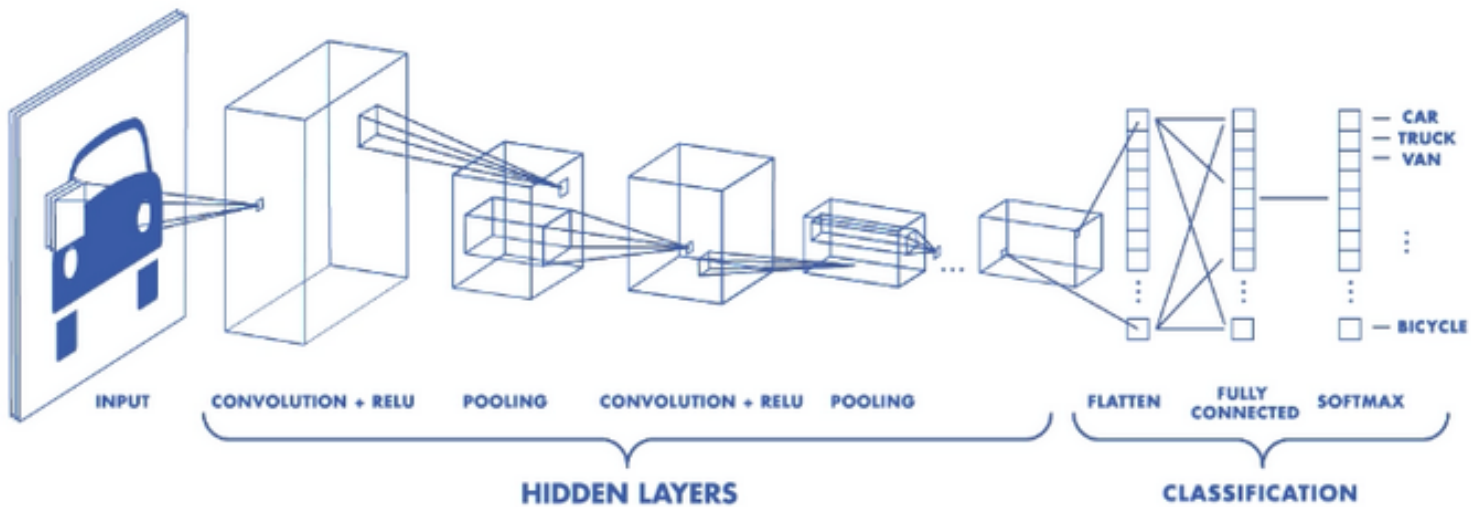


$f = 2k + 1$ is called the filter size or kernel size.

In the sequel, the patch centerd on pixel (i, j) will be denoted $P_k(i, j)$

Convolutional Neural Networks (CNN)

- One of the primary models used to extract essential features from structured input data (such as curves, images, or videos) is the Convolutional Neural Network (CNN).
- A convolutional network automatically constructs new variables to focus on the features that allow for object recognition.
- This preprocessing step is based on a combination of filtering (convolution) and downsampling (pooling) operations.
- The final part of the neural network is usually a multi-layer perceptron.

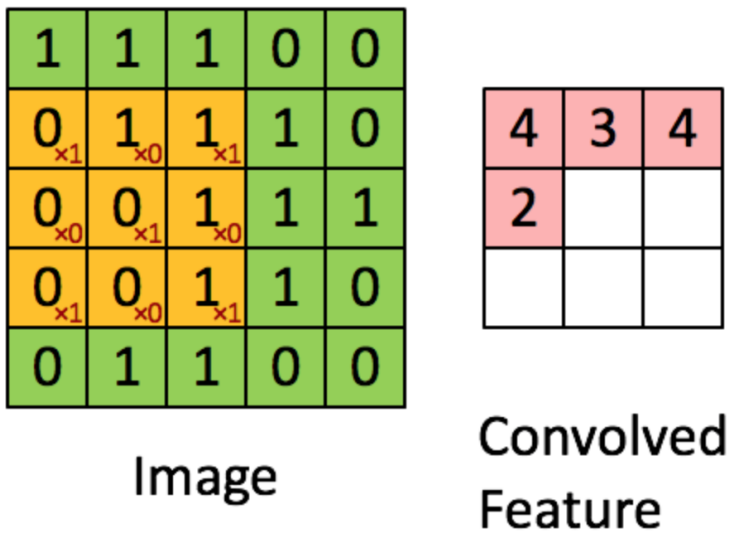


A convolutional neural network architecture. Source: Mathworks

Convolution, locally-connected layers

A local layer act on a patch $P_k(i,j)$:

$$H_{ij} = \sigma\left(\mathbf{W}_{ij}.vect(P_k(i,j))\right)$$



▸ Link

- W are the weights to be estimated.
- Different values of W lead to different filters.
- The most common activation function for σ is the ReLu function.

Example

4 pixels image $\mathbf{x} = [x_1, x_2, x_3, x_4]$

$$\begin{bmatrix} h_1 \\ h_2 \\ h_3 \\ h_4 \end{bmatrix} = \begin{bmatrix} W_{12} & W_{13} & 0 & 0 \\ W_{21} & W_{22} & W_{23} & 0 \\ 0 & W_{32} & W_{33} & W_{34} \\ 0 & 0 & W_{43} & W_{44} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}$$

We remark that the filter is not defined for the borders (x_1 and x_4) : **zeros padding** is used

$$\begin{bmatrix} h_1 \\ h_2 \\ h_3 \\ h_4 \end{bmatrix} = \begin{bmatrix} W_{11} & W_{12} & W_{13} & 0 & 0 & 0 \\ 0 & W_{21} & W_{22} & W_{23} & 0 & 0 \\ 0 & 0 & W_{31} & W_{32} & W_{33} & 0 \\ 0 & 0 & 0 & W_{41} & W_{42} & W_{43} \end{bmatrix} \begin{bmatrix} 0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ 0 \end{bmatrix}.$$

It consists in adding zeros on the borders.

Translation equivarience : "a horse is a horse"

- ✗ In locally-connected layers, two identical patches could lead to two different outputs if the weights $\mathbf{W}_{ij}$ are not the same as $\mathbf{W}_{i'j'}$
- Translation equivariance is imposed

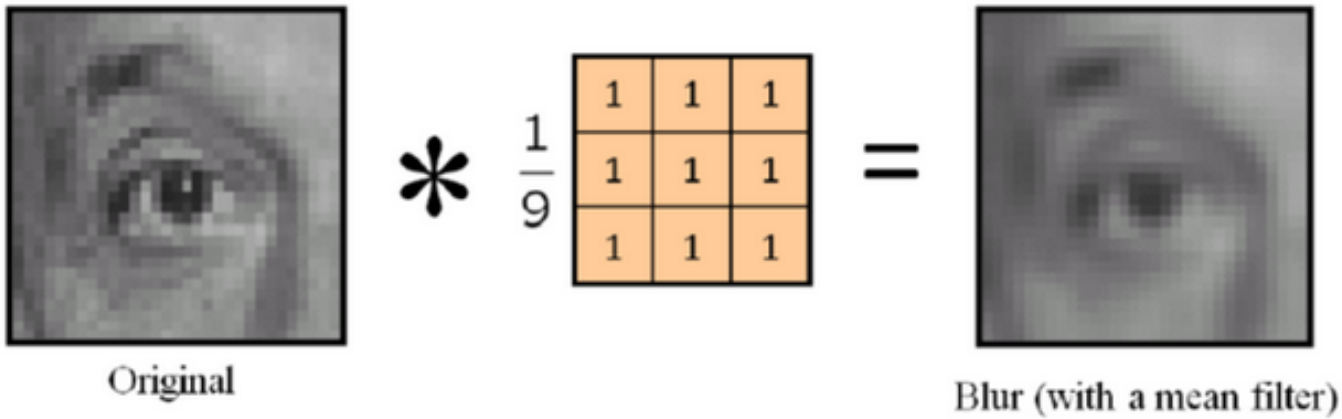
$$P_k(i,j) = P_k(i',j') \quad \text{implies } \Phi\left(\mathbf{W}_{ij}.vect(P_k(i,j))\right) = \Phi\left(\mathbf{W}_{i'j'}.vect(P_k(i',j'))\right)$$

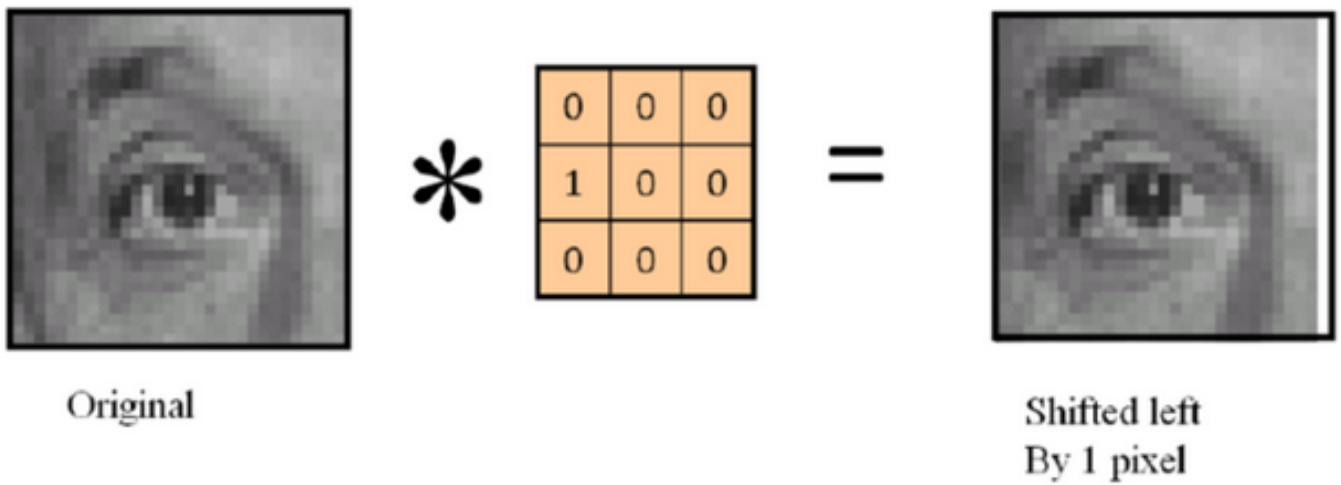
- It is obtained via **weight sharing**.
- **Convolution layer**

$$H_{ij} = \mathbf{W}.vect(P_k(i,j)) + \mathbf{b}$$

where $\mathbf{W}$ and $\mathbf{b}$ are trainable parameters.

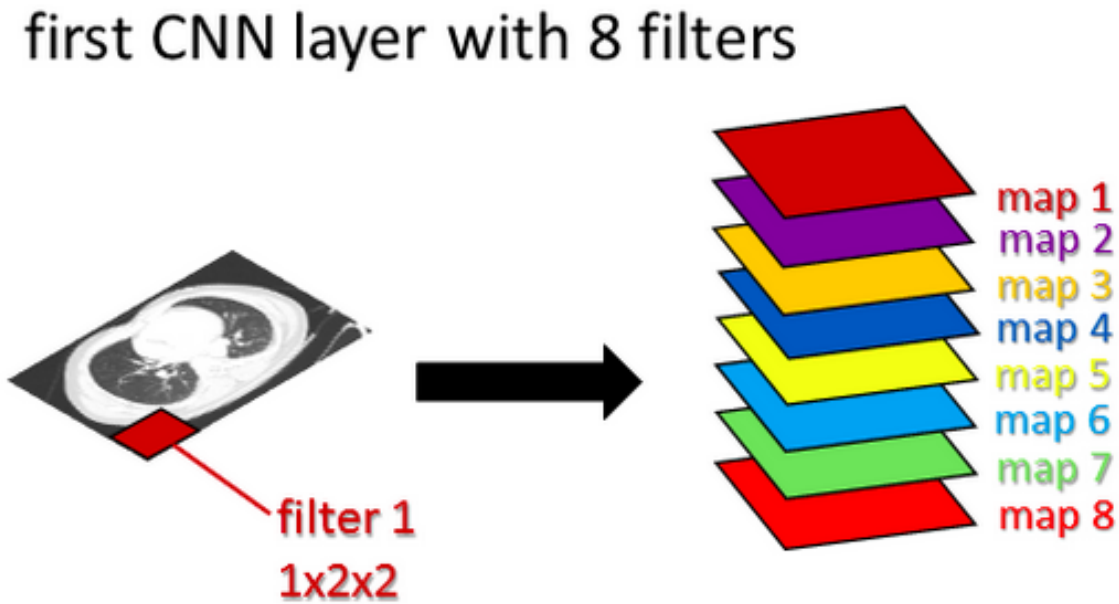
Examples of 3x3 filters





https://towardsdatascience.com/

In practice, at each layer of the CNN, multiple filters of the same size are applied in parallel. The weights of each of the filters are different. This way, different features of the image are explored.



Convolution, size of layers

To summarize, the Conv Layer

- Accepts a volume of size $h_1 \times w_1 \times c_1$
- Requires four hyperparameters :
  - Number of filters N,
  - their spatial extent f (default 3 or 5),
  - the stride s, (default $s = 1$)
  - the amount of zero padding p (default chosen to keep the dimension of the input).
- Produces a volume of size $h_2 \times w_2 \times d_2$ where :

$$W_2 = (w_1 - f + 2p)/s + 1, \quad h_2 = (h_1 - f + 2p)/s + 1, \quad D_2 = K$$

- With parameter sharing, it introduces $f^2 d_1$ weights per filter, for a total of $f^2 d_1 K$ weights and K biases.
- In the output volume, the d-th depth slice (of size $w_2 \times h_2$) is the result of performing a valid convolution of the d-th filter over the input volume with a stride of s, and then offset by d-th bias.

Code

```
from torch.nn import functional as F
x = torch.randn(16, 3, 32, 32)
w = torch.randn(64, 3, 5, 5)
F.conv2d(x, w, padding='same').shape
# [Out]: torch.Size([16, 64, 32, 32])
```

Beyond 2D convulution

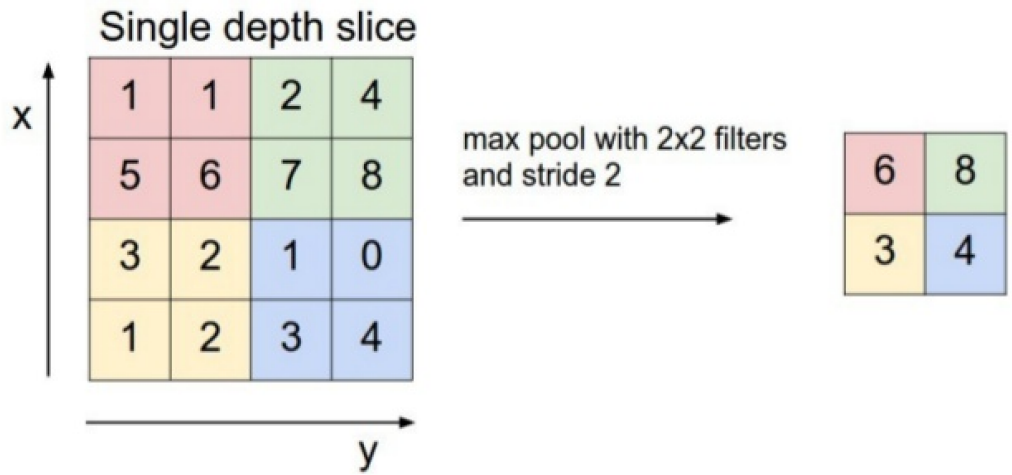
Convolution can also be applied to

- Time series or curve $\rightarrow$ 1D convolution
- Videos, IRM, $\cdots$ $\rightarrow$ 4D convolution

Pooling

After the convolution step (or filtering), there is typically a pooling step that reduces the dimension of each feature map.

- The convolved images are divided into patches (e.g., of size 2 by 2), and each patch is replaced by its average (or maximum).
- Example of *max-pooling*



- This transformation does not require any new parameters.

Pooling, formal definition

Given a tensor $X \sim (h, w, c)$, a max-pooling layer, denoted a $MaxPool(X) \sim (\frac{h}{2}, \frac{w}{2}, c)$, is defined element-wise as

$$[MaxPool(X)]_{ijc} = \max\left([X]_{2i-1:2i,2j-1:2j,c}\right)$$

Hence, we take 2×2 windows of the input, and we compute the maximum value independently for each channel

Pooling, layers size

The pooling layer

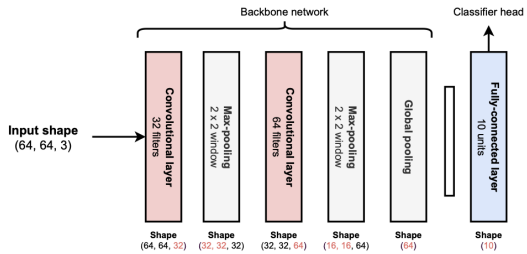
- Accepts a volume of size $W_1 \times H_1 \times D_1$
- Requires two hyperparameters :
  - their spatial extent F,
  - the stride S.
- Produces a volume of size $W_2 \times H_2 \times D_2$ where :

$$W_2 = (W_1 - F)/S + 1, \quad H_2 = (H_1 - F)/S + 1, \quad D_2 = D_1$$

- Introduces zero parameters since it computes a fixed function of the input
- For Pooling layers, it is not common to pad the input using zero-padding.

Designing the complete model

The complete model, then, can be decomposed as three major components : a series of convolutional blocks, a global average pooling, and a final block for classification.



$$H = (ConvBlocko \cdots oConvBlock)(X) \tag{1}$$

$$\mathbf{h} = \frac{1}{h'w'} \sum_{i,j} H_{ij} \tag{2}$$

$$y = MLP(\mathbf{h}) \tag{3}$$

(1) : Convolution blocks, (2) : global pooling [1], (3) : Multilayers Perceptron for classification.

Global pooling allow to remove the spatial dependency (is it really usefull ?)

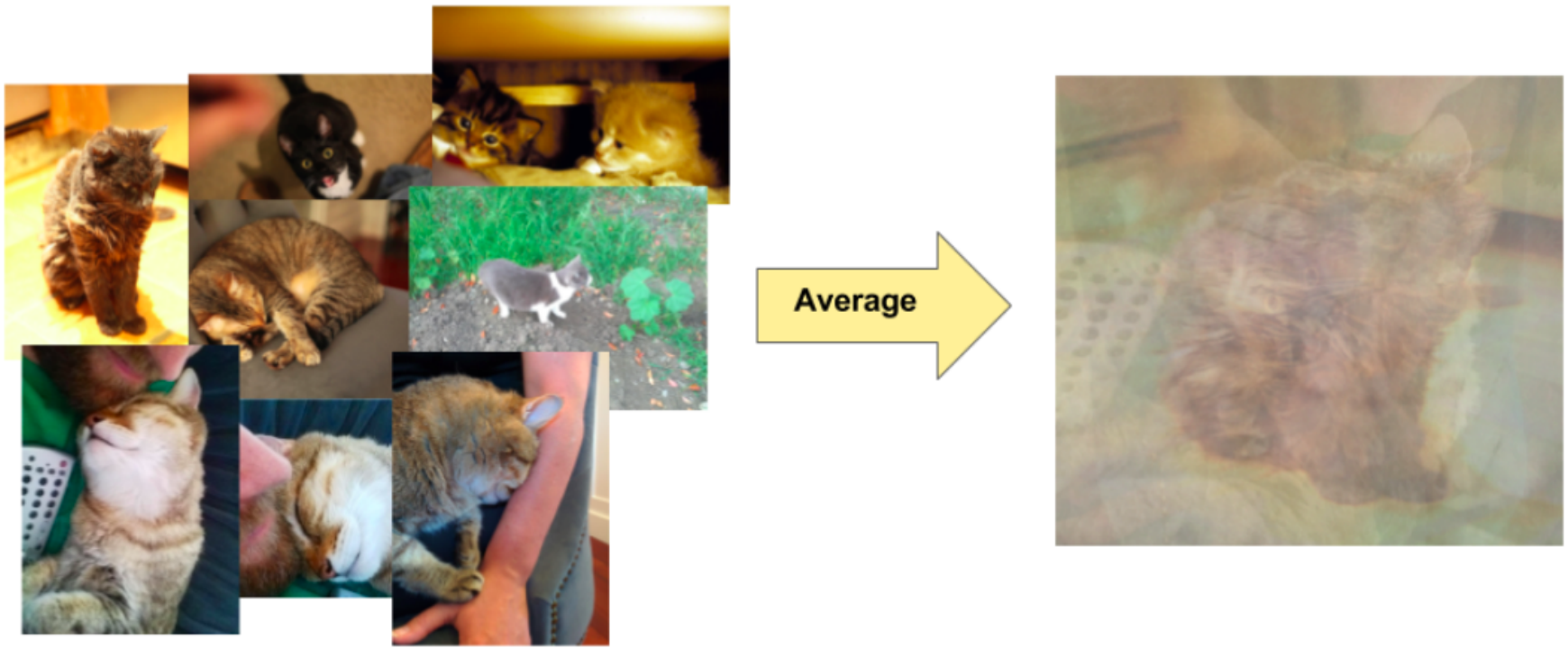
1. precisions ?

Outline

Images classification

Several years ago, to classify images, the goal was to identify image features such as background color, shapes, and so on.

However, as seen in the images below, images of the same 'object' can be very diverse. For example, if we calculate their average, no clear trend emerges.



Source `https://developers.google.com/machine-learning/practica/image-classification`

CNN for images classification

- Example of code, with keras, for classifying handwritten digits
  In code illustrate
  - how to build a CNN, fit the model
  - how predict the class of a test dataset and validate the model
  - the output of the layers of the CNN

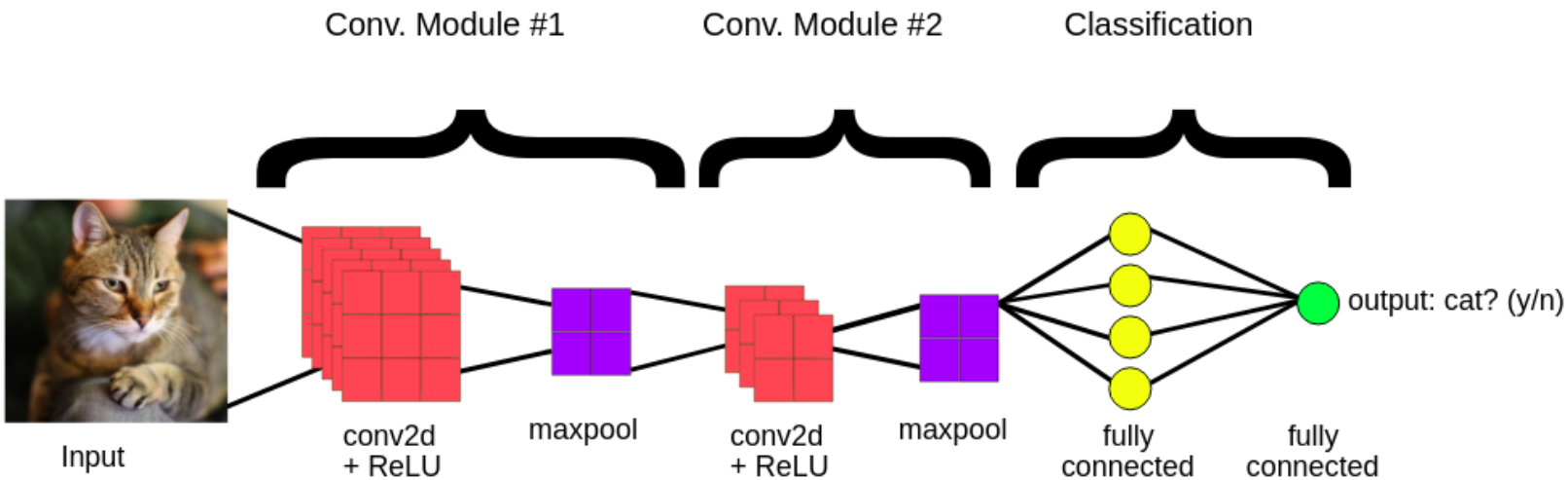
Example based on keras :
`https://drive.google.com/open?id=1A7xgkLqVlWwi40FTqmDNmntsjD2R2SUb`

Example based on pytorch : `https://colab.research.google.com/drive/1JesK8ok8lv5vDUxWTcwexWPoiakycB0c?usp=sharing`

Example based on equinox and jax :
`https://docs.kidger.site/equinox/examples/mnist/`

CNN example

Exercice : Run the code for identifying cat versus dog. Try to understand each step.

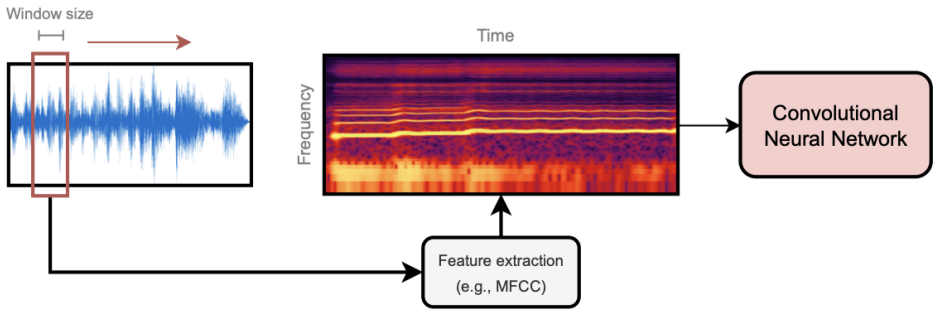


https://colab.research.google.com/drive/1PcGgXrZ7OcTV_P9drKFbYvhSyVirBIox?usp=sharing

Outline

CNN for audio

Audio can be represented as either a 1D sequence (left), or a 2D image in a time-frequency domain (middle). In the second case, we can apply the same techniques described in the previous chapter.



If it is considered as a 1D signal,

$$X\,(t, c)$$

t : length of the sequence, c : number of features.

Convolution for 1D sequence

$$[Conv1D(X)]_i = \Phi(\mathbf{W}\text{vect}(P_k(i)) + \mathbf{b})$$

with trainable parameters $\mathbf{W} \sim (c', kc)$ and $\mathbf{b} \sim c'$

Dealing with variable-length inputs

Consider two audio files (or two time series, or two texts), described by their corresponding input matrices $X_1 \sim (t_1, c)$ and $X_2 \sim (t_2, c)$

✗If g is a generic composition of 1D convolutions and max-pooling operations, the outputs of the block are matrices

$$\mathbf{H}_1 = g(\mathbf{X}_1), \quad mathbfH_2 = g(\mathbf{X}_2)$$

having the same number of columns but a different number of rows.

✓After global average pooling, the dependence on the length disappears

$$\mathbf{h}_1 = \sum_i \mathbf{H}_{1i}, \quad \mathbf{h}_2 = \sum_i \mathbf{H}_{2i}$$

and we can proceed with a final classification on the vectors $\mathbf{h}_1$ and $\mathbf{h}_2$.

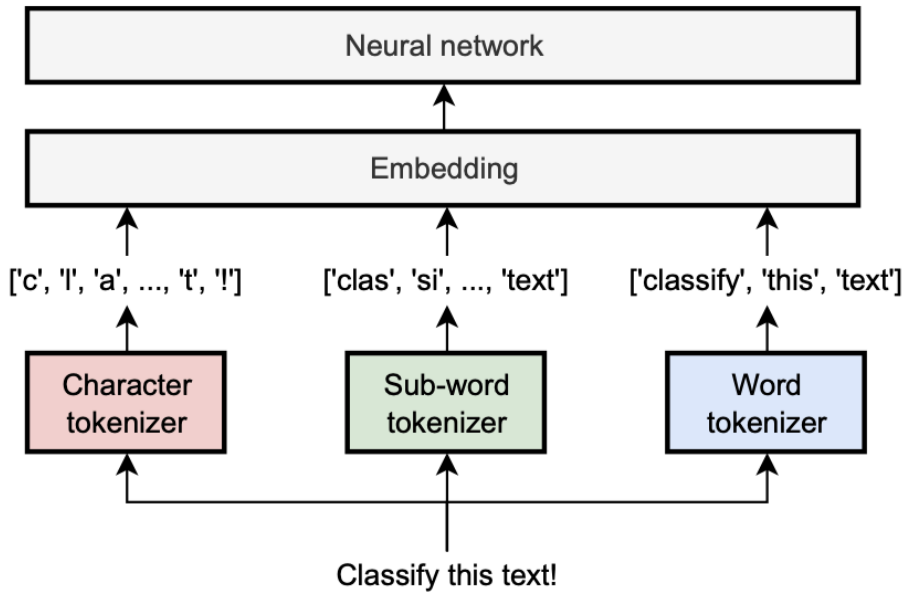
✗mini-batches cannot be built from matrices of different dimensions
✓this can be handle by zero-padding

code

```
# Sequences with variable length
# (3, 5, 2, respectively)
X1, X2, X3 = torch.randn(3, 8),
torch.randn(5, 8),
torch.randn(2, 8)
# Pad into a single mini-batch
X = torch.nn.utils.rnn.pad_sequence(
[X1, X2, X3],
batch_first=True)
print(X.shape)
# [Out]: torch.Size([3, 5, 8])
```

CNN for text

The first step in dealing with text is **tokenization**.



The user has to define a **dictionary** (**vocabulary**) of allowed tokens. Pre-trained subword tokenizers are a standard choice nowadays [2]

The sequence can be equivalently represented by a sequence of integers.
Example : *"This is perplexing !"* wcan be represented by

$$[2028, 374, 74252, 287, 0],$$

each number representing a token.

2. OpenAI has released an open-source version of its own tokenizer ; it contains about 100k subwords

After the tokenization step, the tokens must be **embedded** into vectors to be used as inputs for a CNN.

Simple one- hot encoding strategy here works poorly.

The embeddings can be *trained* together with the rest of the network. In practice, a matrix of embeddings $\mathbf{E} \sim (n, e)$ is intialized.
Then a look-up operation replaces each integer with the corresponding row in E.

$$\text{LookUp}(x) = \mathbf{X} = \begin{bmatrix} \mathbf{E}_{x_1} \\ \mathbf{E}_{x_2} \\ \vdots \\ \mathbf{E}_{x_m} \end{bmatrix} \quad \leftarrow \text{ Row } x_1 \text{ in } \mathbf{E}$$

where m is the length of the input sequence.

$$\hat{y} = CNN(\mathbf{X})$$

An alternative is to use pretrained networks.

- ✓ It is a powerfull idea because the embeddings can be used as a "translation"/"representant" for other tasks !
- ✓ It like if all the words/token are projected into a lerge dimension space. Two words/token with the same meaning are close to each other in this space.

code

```
class TextCNN(nn.Module):
    def __init__(self, n, e):
        super().__init__()
        self.emb = nn.Embedding(n, e)
        self.conv1 = nn.Conv1d(e, 32, 5,padding='same')
        self.conv2 = nn.Conv1d(32, 64, 5,padding='same')
        self.head = nn.Linear(64, 10)
    def forward(self, x): # (*, m)
        x = self.emb(x) # (*, m, e)
        x = x.transpose(1, 2) # (*, e, m)
        x = relu(self.conv1(x)) # (*, 32, m)
        x = max_pool1d(x, 2) # (*, 32, m/2)
        x = relu(self.conv2(x)) # (*, 64, m/2)
        x = x.mean(2) # (*, 64)
        return self.head(x) # (*, 10)
```

Outline

Transformers

Transformers [VSP+17] are based on an architecture designed to handle efficiently long-range dependencies in natural language processing.

- ✓ decoupling between the data type (through the use of appropriate tokenizers) and the architecture, which for the most part remains data-agnostic.

Core component of transformers : **multi-head attention**.

`https://www.youtube.com/watch?v=wjZofJX0v4M&list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi&index=6`

Consider these two sentences :

The cat is on the table

and a longer one :

The cat, who belongs to my mother, is on the table

- Both must be tokenized.
- Tokens belonging to *cat* and *table* have the same dependencies in both sentences.

We need weighting the token in such a way that learn and use long range dependencies.

Let $\mathbf{x}$ be a sequence of n tokens in a space E of dimension e.

Capturing dependencies with CNN : kernel of size k applied to each token i

$$h_i = \sum_{j=1}^{2k+1} \mathbf{w}_j \mathbf{x}_{i+k+1-j}$$

$\rightarrow$ long dependencies require large k and many waigth matrices $\mathbf{w}_j$

Idea : use a function of the "distance" between i and j

$$h_i = \sum_{j=1}^{n} g(i-j)\mathbf{x}_j$$

$\rightarrow$ The sum is now on *all* tokens.
Link with CNN : $g(i,j) = \mathbf{W}_{ij} \quad \text{if} \quad |i-j| \leq k, \quad 0 \quad \text{elsewhere}$

More generally, the function can depend on the content of the tokens

$$h_i = \sum_{j=1}^{n} g(\mathbf{x}_i, \mathbf{x}_j)\mathbf{x}_j$$

Attention

The MHA layer is a simple version

$$g(\mathbf{x}_i, \mathbf{x}_j) = \frac{\mathbf{x}_i^T \mathbf{x}_j}{\sqrt{e}}$$

where e is the number of tokens in the embedding space.

$$h_i = \sum_{j=1}^{n} \text{softmax}_j(g(\mathbf{x}_i, \mathbf{x}_j))\mathbf{x}_j$$

where softmax_j means we are applying the softmax normalization to the set $\{g(\mathbf{x}_i, \mathbf{x}_j)\}_{j=1}^n$ independently for each i.

Remarks

- ✓ each token has a given amount of attention it can allocate to the other tokens
- ✗ this layer has no trainable parameters

Attention layer (single head)

The idea of an attention layer is to recover trainable paramters by adding trainable projections to the input before computing the MHA.

- Key tokens : $k_i = \mathbf{w}_k^T \mathbf{x}_i$
- Value tokens : $v_i = \mathbf{w}_\nu^T \mathbf{x}_i$
- Query tokens : $q_i = \mathbf{w}_q^T \mathbf{x}_i$

Self-attention (SA) layer

$$h_i = \sum_{j=1}^{n} \text{softmax}_j(g(q_i, k_j)) v_j$$

where $\mathbf{w}_k \in \mathbb{R}^{(e,k)}$, $\mathbf{w}_v \in \mathbb{R}^{(e,\nu)}$, $\mathbf{w}_q \in \mathbb{R}^{(k)}$ and k, ν are hyper parameters.

Attention layer (multi head)

- Single-head attention allows to model pairwise dependencies across tokens with high flexibility.
  However, we may wish to model several dependencies in parallel for instance *cat* and *table* or *cat* and *mother*.

- A multi-head layer achieves this by running multiple attention operations in parallel, each with its own set of trainable parameters.

- It is followed by a pooling operation.

Why "Key" and "Query" ?

A dictionary in Python : a value is returned only if a perfect key-query match is found. Otherwise, we get an error.

```
d = dict()
d["Alice"] = 2
d["Alice"]     # Returns 2
d["Alce"]      # Returns an error
```

Variants exist where the closest key is returned.

We can interpret the self-attention layer as a soft approximation in which each token is updated with a weighted combination of all values based on the corresponding key/query similarities.

One more step towards the complete transformer

- ✗ the MHA is invariant by permuting the position of the words...
- We need to add some information about the relative positons of the tokens : $g(\mathbf{x_i}, \mathbf{x}_j) \rightarrow g(\mathbf{x_i}, \mathbf{x}_j, i-j)$
- In practice, we add a trainable matrix

$$g(\mathbf{x}_i, \mathbf{x}_j) = \frac{\mathbf{x}_i^T \mathbf{x}_j}{\sqrt{e}} + \mathbf{B}_{ij}$$

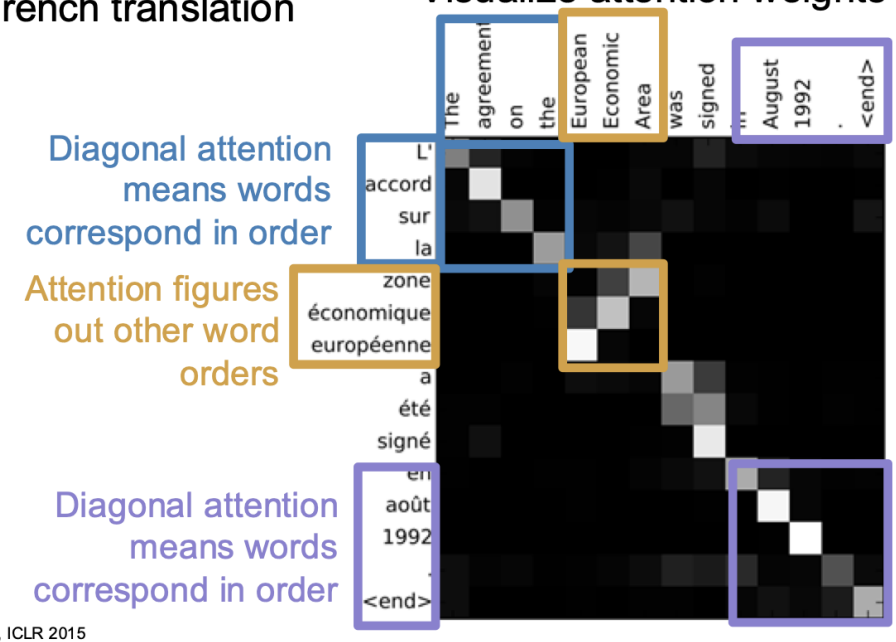
where $\mathbf{B} \in \mathbb{R}^{(m,m)}$, m the number of words in the sentence.

Example: English to French translation

Visualize attention weights $a_{t,i}$

Input: "The agreement on the European Economic Area was signed in August 1992."

Output: "L'accord sur la zone économique européenne a été signé en août 1992."



Bahdanau et al, "Neural machine translation by jointly learning to align and translate", ICLR 2015

source `https://cs231n.stanford.edu/slides/2025/lecture_8.pdf`

Transformer block

The Transformer

Transformer Block

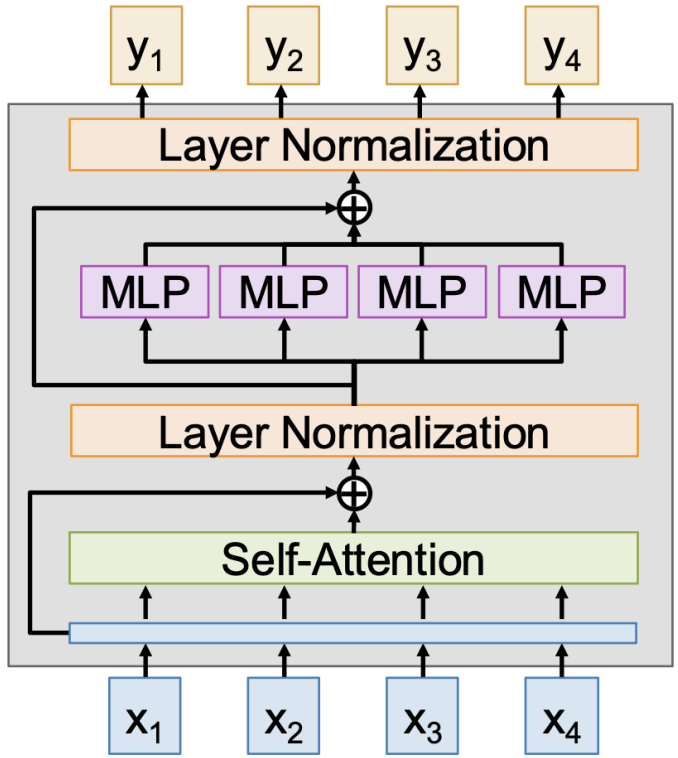
Input: Set of vectors x
Output: Set of vectors y

Self-Attention is the only interaction between vectors

LayerNorm and MLP work on each vector independently

Highly scalable and parallelizable, most of the compute is just 6 matmuls:

4 from Self-Attention
2 from MLP



Vaswani et al, "Attention is all you need," NeurIPS 2017

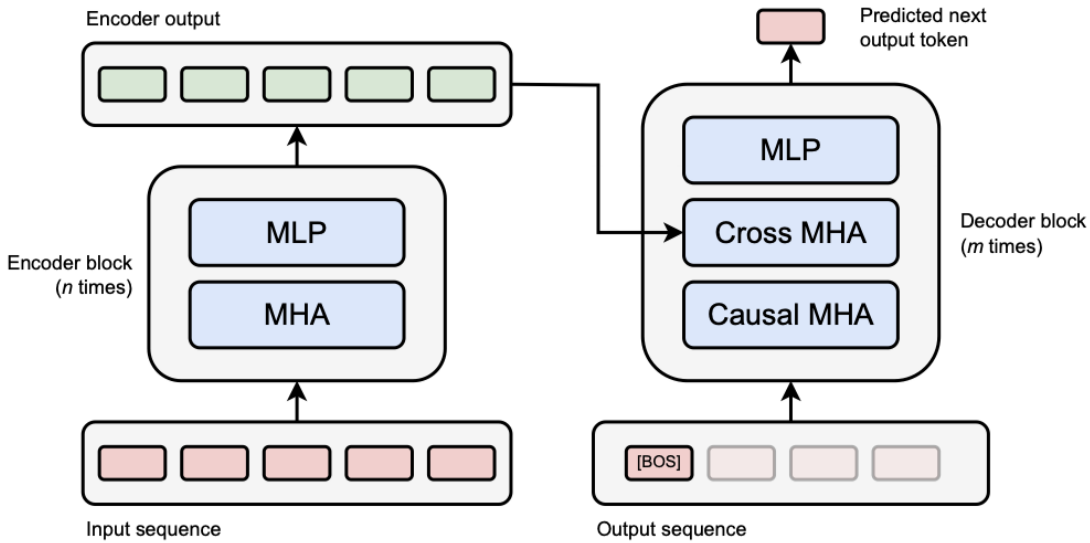
source `https://cs231n.stanford.edu/slides/2025/lecture_8.pdf`

- This transformer can be used for regression or classification
- Normalization layer is a standardization ; it helps to stabilizes training.

Transformer encoder-decoder

For sequence to sequence task, it has to be called in a **encoder-decoder** model.

- an encoder that processes the input sequence to a transformed representation
- a decoder that autoregressively generates the output sequence conditioned on the output of the encoder.



- Cross attention helps to capture the dependencies between input sequence and output sequence.
- Causal attention ensures autoregressive generation. It works with a mask which desactivate tokens that appear after the current position.

A commented example

For the MNIST dataset

https://medium.com/correll-lab/building-a-vision-transformer-model-from-scratch-a3054f707cc6

Outline

Various Stochastic Algorithms

In deep learning, several variants of the stochastic gradient descent (SGD) algorithm have been proposed.

- Momentum
  This method involves replacing the gradient with a weighted average of all "historical" gradients. An exponentially weighted forgetting factor is used, resulting in the well-known form of exponential moving average.

$$m_r = \alpha m_{t-1} + (1-\alpha)\nabla_\omega L(\omega; \mathbf{x}_{i:i+k}, y_{i:i+k})$$
$$\omega_{r+1} = \omega_r - \gamma m_r$$

  Typically, α is chosen to be close to 0.9. This technique is widely used in practice.
- RMSprop
  In this variant, adjustments are made to the learning rate.

$$v_r = \alpha v_{r-1} + (1-\alpha)\left(\nabla_\omega L(\omega; \mathbf{x}_{i:i+k}, y_{i:i+k})\right)^2$$
$$\omega_{r+1} = \omega_r - \frac{\gamma}{\sqrt{v_r + \epsilon}} \nabla_\omega L(\omega; \mathbf{x}_{i:i+k}, y_{i:i+k})$$

  The intuition here is that larger steps can be taken when the gradient is small (indicating a flat region of the objective function) and vice versa.
- Adam
  The Adam algorithm combines ideas from both Momentum and RMSprop.

Some Vocabulary

- **Batch** : A small sample of observations (randomly drawn) used to approximate the gradient during a step of the stochastic gradient descent algorithm. Typically, batches have a size of $b = 128, 216, 512$, etc. In practice, the training dataset is randomly shuffled, and the first batch consists of the first b observations, and so on.

- **Epoch** : A cycle of gradient descents during which as many iterations as necessary are performed to evaluate all the batches.

Outline

Regularization to Prevent Overfitting

In deep neural networks (including very deep ones), the number of parameters is high : adding layers and compute power for training is proportionally linked to the accuracy of the model up to a saturation point given by the dataset.
✗it runs into a number of problems ranging from slow optimization to gradient issues and numerical instabilities

✓Various methods exist for regularization, i.e., reducing the model's complexity during training.

- Weight regularization
  Similar to Ridge or Lasso regression, a L1 or L2 norm penalty is added to the loss function.
- Data augmentation
  Add transformed data to the learning set in order to increase the data set size.
- Early stopping
  Stop the gradient descent before convergence.
- Dropout
- Batch Normalization
- Residual connections

Weight regularization

Weight regularization is also known as weight decay.

$$L_{\text{reg}} = L(\mathbf{w}) + \lambda R(\mathbf{w})$$

If $R(\mathbf{w}) = ||w||^2 = \sum_i \mathbf{w}_i^2$, then

$$\nabla L_{\text{reg}} = \nabla L(\mathbf{w}) + 2\lambda \mathbf{w}$$

And gradient descent step is simplified as follows

$$\mathbf{w}_r = \mathbf{w}_{r-1} - g\big(\nabla L(\mathbf{w})\big) - \lambda \mathbf{w}_{r-1}$$

where g stands for instance for the update of Adam's algorithm.

It leads to AdamW algorithm which may work better than Adam's one.

Early stopping

From the point of view of optimization, minimizing a function $L(\mathbf{w})$ is the task of finding a stationary point as quickly as possible, i.e. a point $\mathbf{w}_r$ such that

$$||L(\mathbf{w}_r) - L(\mathbf{w}_{r-1})||^2 \leq \epsilon$$

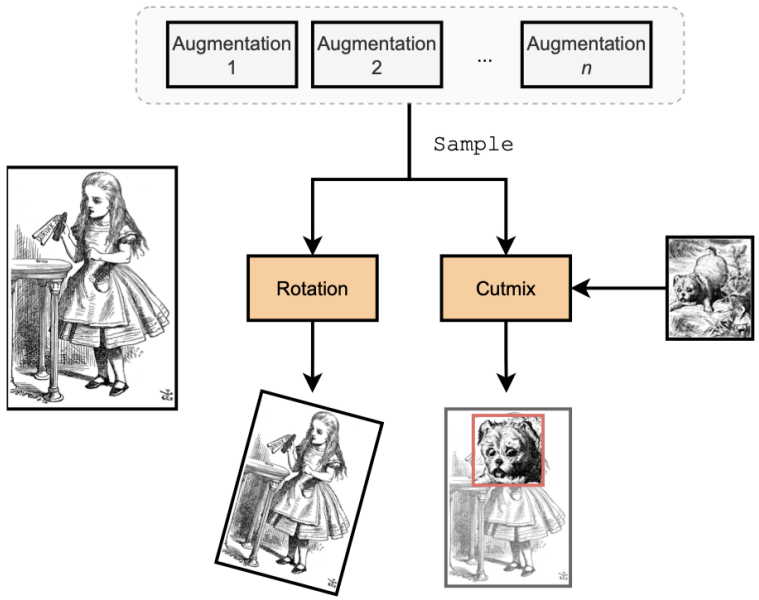
The idea of early stopping is to look at the loss function of the validation dataset. If it does not decreasing during k epoch (k is called the **patience**), the algorithm is stopped.

✓Early stopping can be seen as a simple form of **model selection**, where we select the optimal number of epochs based on a given metric. Any metric can be used.

Data augmentation

Data augmentation is the process of transforming images during training according to one or more of these transformations.

It can be applied to the batches during training so that the images are not stored.



Code

```
# Image Classification
import torch
from torchvision.transforms import v2

H, W = 32, 32
img = torch.randint(0, 256, size=(3, H, W), dtype=torch.uint8)

transforms = v2.Compose([
    v2.RandomResizedCrop(size=(224, 224), antialias=True),
    v2.RandomHorizontalFlip(p=0.5),
    v2.ToDtype(torch.float32, scale=True),
    v2.Normalize(mean=[0.485, 0.456, 0.406],
                 std=[0.229, 0.224, 0.225]),
])
img = transforms(img)
```

Dropout

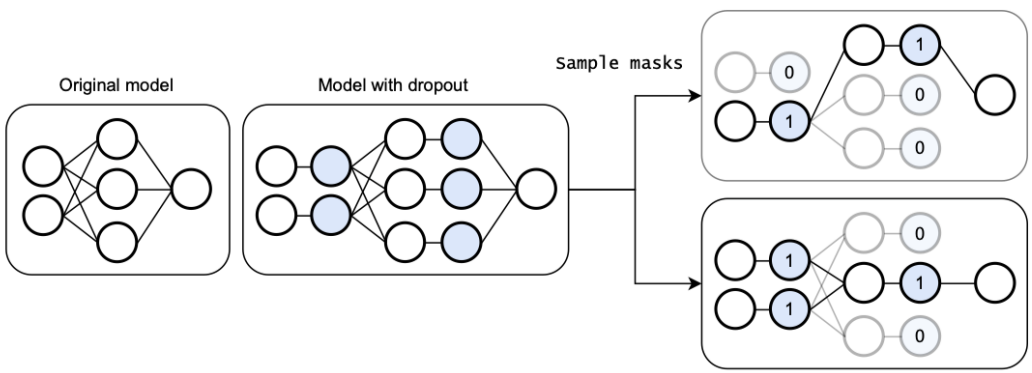
Previous methods imply modifications to the optimization algorithm or to the dataset itself, and they can be applied to a wide range of algorithms.

Dropout is more specific to neural networks.

In a dropout layer, we first sample a binary matrix $M \sim Binary(n, c)$, whose elements are drawn from a Bernoulli distribution with probability p (where $p \in [0, 1]$ is a user's hyper-parameter). The output of the layer is obtained by masking the input :

$$\text{Dropout}(X) = M \odot X$$

The layer has a single hyper-parameter, p, and no trainable parameters.



While dropout can improve the performance, the output y is now a random variable with respect to the sampling of the different masks inside the dropout layers, which is undesirable after training.
For example, two forward passes of the network can return two different outputs.
Hence, we require some strategy to replace the forward pass with a deterministic operation.

- **Monte Carlo Dropout** One choice is to run the forward pass many times (with random dropout) and replace the dropout effect with its expected value.

$$E_{p(M)}[F(x;M)] \simeq \frac{1}{k}\sum_{i=1}^{k} f(x;Z_i), \quad Z_i \sim p(M)$$

- **Inverted Dropout** A simpler (and more common) option is to replace the random variables layer-by-layer, which is a reasonable approximation.

$$E_{p(M)}[\text{Dropout}(X)] = pX$$

Remark : High p can be used for fully connected neural networks. For CNN, dropping single elements of the input tensor results in sparsity patterns which are too unstructured (see **spatial dropout**).

Code

```
x = torch.randn((16, 2))
# Training with dropout
model.train()
y = model(x)
# Inference with dropout
model.eval()
y = model(x)
# Monte Carlo dropout for inference
k = 10
model.train()
y = model(x[:, None, :].repeat(1, k, 1)).mean(1)
```

Building a CNN for Cat vs. Dog Image Classification

Exercise 1 : CNN from scratch
Click here
(or copy the link
`https://colab.research.google.com/github/google/eng-edu/blob/main/ml/pc/exercises/image_classification_part1.ipynb`)

Exercise 2 : We will use augmentation and dropout to reduce the risk of overfitting.
Click here

(or copy the link
`https://colab.research.google.com/github/google/eng-edu/blob/main/ml/pc/exercises/image_classification_part2.ipynb`)

Batch normalization

The idea of Batch normalization it to standardize the output of the layers. It is done for the mini batch.

Consider again the output of any fully-connected layer $\mathbf{X} \sim (n, c)$, where n is the mini-batch size. Each column of $\mathbf{X}$ is normalized

$$\mu_j = \frac{1}{n}\sum_{i=1}^{n} \mathbf{X}_{ij}, \quad \sigma_j^2 \frac{1}{n}\sum_{i=1}^{n}(\mathbf{X}_{ij} - \mu_j)^2$$

$$\mathbf{X}' = \frac{\mathbf{X} - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

The choice of zero mean and unitary variance is just a convention, not necessarily the best one. To generalize it, we can let the optimization algorithm select the best choice.

$$\mathbf{X}'' = \alpha \mathbf{X}' + \beta$$

with α and β trainable parameters.

✓ Batch normalization helps to stabilize the optimization algorithm and permits to use larger learning rates.

✗ The variance of μ during training will grow large for small mini-batches, and training can be unfeasible for very small mini-batch sizes

✗ Replacing μwith a different value after training creates an undesirable mismatch between training and inference (inference = prédiction)
In practice, a rolling set of statistics are updated after each forward pass of the model during training, and use these after training.

$$\hat{\mu} \leftarrow \lambda \hat{\mu} + (1 - \lambda)\mu$$