Cours 2 : Réseaux de neurones « simples » NN, CNN, RNN

université de BORDEAUX

Before starting

- **Contact:** victoria.bourgeais@u-bordeaux.fr
- **Schedule:** 12 weeks (37-43,45-49), 48h CI
  - 11/09 (4h) : Outillage
  - **18/09 (4h) : Réseaux de neurones « simples » : NN, CNN, RNN**
  - 25/09 (4h) + 02/10 (2h) : ResNet, VGG, U-Net
  - 02/10 (2h) + 09/10 (4h) : Modèles génératifs : Autogénération, GAN, Auto-encodeurs, VAE, (Akka)
  - 16/10 (4h) : Modèles de graphes : GNN, GCNN
  - 23/10 (4h) : Architectures pour le NLP : LSTM, Embeddings, NPLM, Mécanisme d'attention, tranformers à GPT
  - 06/10 (4h) : Stable diffusion
  - 13/11 (4h) : Réseaux pour les données multimodales : CLIP, …
  - 20/11, 27/11 et 04/12 (12h) : Projet

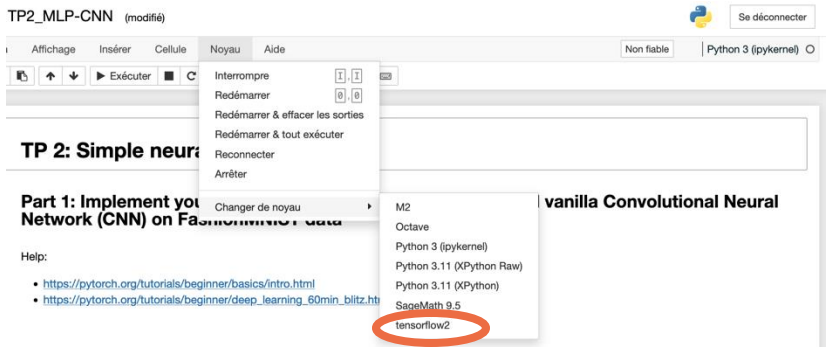
Before starting

- Working remotely: https://services.emi.u-bordeaux.fr/intranet/spip.php?article175
- Set the virtual environnement as a Jupyter kernel:
  1. Open a terminal

  2. Activate the virtual environment:
```
source espaces/ens/DeepLearning/python3/tensorflow2/bin/activate
```

  3. Run the following command (supposed ipykernel is installed)
```
python -m ipykernel install --user --name=tensorflow2
```

  4. When launching a jupyter notebook, you should see it:



Overview

1. Neural network: ingredients
   Multi-Layer Perceptron (MLP)

2. Convolutional Neural Network (CNN)
   A simple vanilla CNN: LeNet

3. Recurrent Neural Network (RNN)

Overview

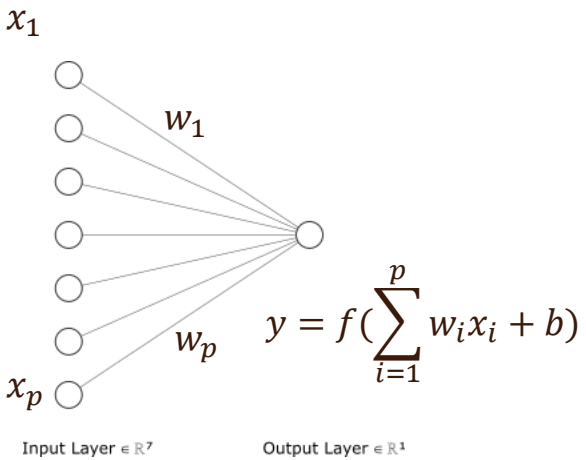
1. **Neural network: ingredients**
   Multi-Layer Perceptron (MLP)

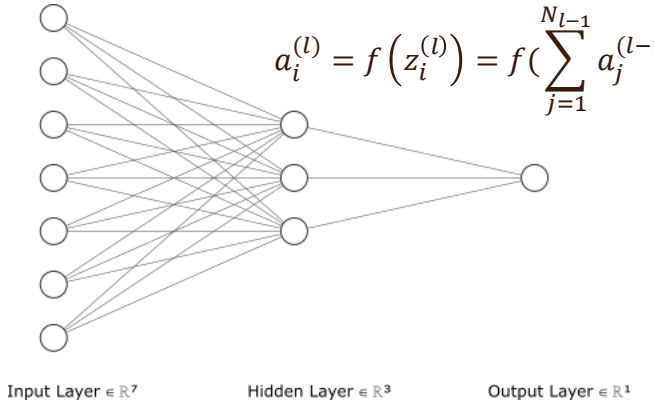
2. Convolutional Neural Network (CNN)
   A simple vanilla CNN: LeNet

3. Recurrent Neural Network (RNN)

From perceptron to multi-perceptron

(Bengio et al., 2017)





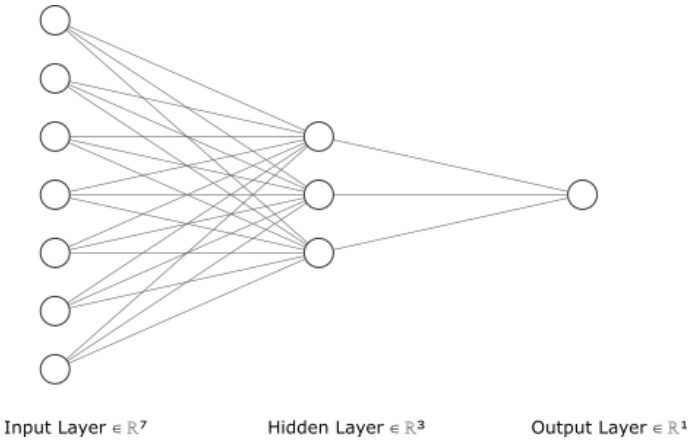
The set of parameters of a MLP with L layers is defined as:

$$\theta = \{(W^{(1)}, b^{(1)}), (W^{(2)}, b^{(2)}), \ldots, (W^{(L)}, b^{(L)})\}$$

$W^{(l)}$ has a shape of (n_l, n_{l-1}) and $b^{(l)}$ $(n_l, 1)$

From perceptron to multi-perceptron

Implementation in PyTorch

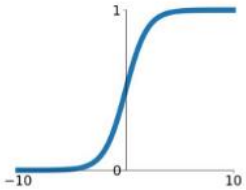



```
import torch.nn as nn

class VanillaMLP(nn.Module):
     def __init__(self,n_classes):
          super(VanillaMLP,
          self).__init__()
          self.fc1 = nn.Linear(7, 3)
          self.fc2 = nn.Linear(3, 1)
          self.relu = nn.ReLU()

     def forward(self, x):
          h1 = self.relu(self.fc1(x))
          y = self.fc2(h1)
     return y
```

Activation functions

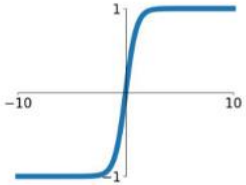
Sigmoid

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



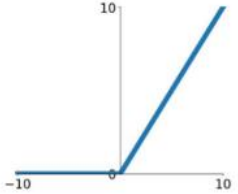
tanh

$$\tanh(x)$$



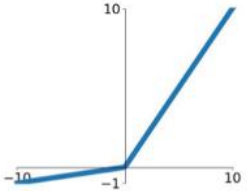
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

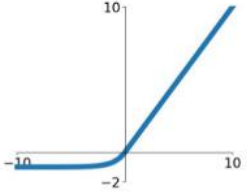


Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Output layer: activation function and loss

- It depends on the type of learning: supervised or unsupervised learning
- In case of supervised learning: two cases

➢ Classification: compute a probability

- Binary:

Use of sigmoid function such as $y = a^{(L)} = \frac{1}{1-\exp(z^{(L)})}$

Use of Binary Cross-Entropy (BCE) Loss: $\mathcal{L} = -\frac{1}{N}\sum_{i=1}^{N}[y_i \log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)]$

- Multiclass:

Use of softmax function such as $y_k = a_k^{(L)} = \frac{\exp(z_k^{(L)})}{\sum_{j=1}^{K}\exp(z_j^{(L)})}$ with K the number of classes

→ one output neuron by class k

Use of Cross-Entropy (CE) Loss: $\mathcal{L} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{k=1}^{K} y_{i,k}\log(\hat{y}_{i,k})$

Output layer: activation function and loss

- It depends on the type of learning: supervised or unsupervised learning
- In case of supervised learning: two cases

  - Regression :

    Use of identity function such as $y = a^{(L)} = z^{(L)}$

    Use of Mean-square error (MSE) Loss: $\mathcal{L} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$

Gradient descent

- To minimize the loss and optimize the parameters of your NN: you should use a gradient descent algorithm.
- There exist different ones: SGD, SGD with mini-batch, Adam… Most of them are implemented in PyTorch in the module torch.optim

```
optimizer = optim.SGD(net.parameters(), lr=0.001)
for epoch in np.arange(N_EPOCHS):
     for idx, batch in enumerate(trainloader): #training loop
     # get the inputs; batch is a list of [inputs, labels]
          inputs,labels=batch
          inputs=inputs.to(device) #train on GPU
          labels=labels.to(device)
          # zero the parameter gradients
          optimizer.zero_grad()
          # forward + backward + optimize
          out = net(x=inputs)
          loss = criterion(out, labels)
          loss.backward()
          optimizer.step()
```

Gradient descent: focus on Adam optimizer

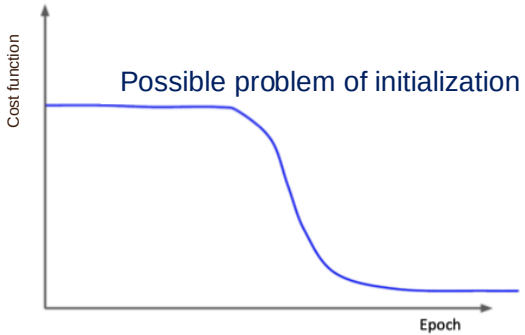
- Adam is quite popular and effective for most tasks. It is a combination of RMSprop and Momentum.

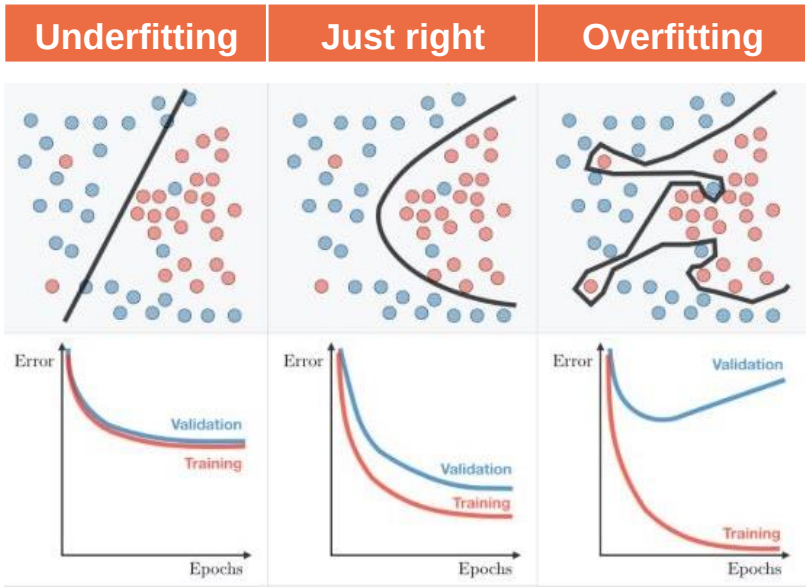
Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
  $m_0 \leftarrow 0$ (Initialize 1st moment vector)
  $v_0 \leftarrow 0$ (Initialize 2nd moment vector)
  $t \leftarrow 0$ (Initialize timestep)
  while θ_t not converged **do**
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
    $\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
    $\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
  end while
  return θ_t (Resulting parameters)

Diagnose learning problems (most common ones)

- Overfitting/underfitting
- Problem of initialization



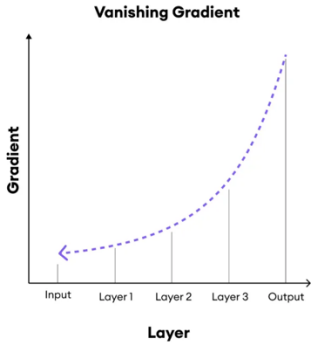


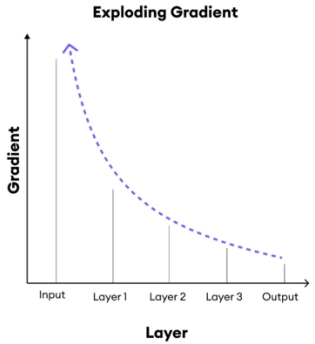
➔Complexify the model
➔Add more features

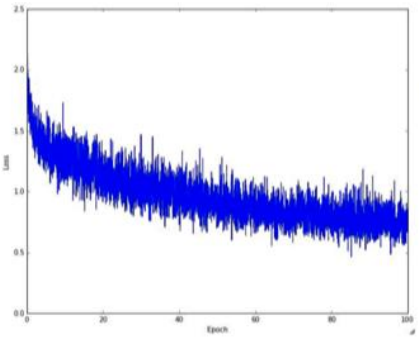
➔Perform regularization

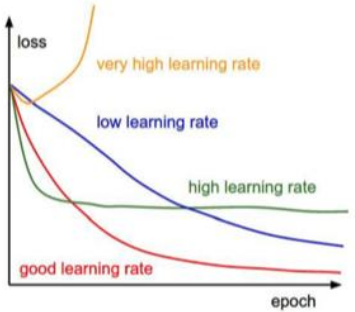
Diagnose learning problems (most common ones)

- Vanishing Gradient: gradients tend to nullify (in particular, fully-connected and large neural networks are sensitive).
- Exploding Gradient: gradients become very large.









Which solutions?

- Appropriate initialization of the model parameters
- Regularization techniques
  - Early stopping
  - L1,L2 Regularization
  - Batch Normalization
  - Dropout

Initialization of model parameters

- Initialization will affect the speed of convergence of the gradient descent algorithm.
- Initialization with zero weights doesn't work most of the time.
- In Pytorch, the parameters are initiated with a uniform distribution: $U(-\sqrt{k}, \sqrt{k})$ where $k = \frac{1}{fan\_in}$ (check the documentation of `nn.Linear`).
- The choice of initialization is linked to the activation function (have a look at `nn.init`) with `fan_in`=number of input neurons and `fan_out`=number of input neurons.

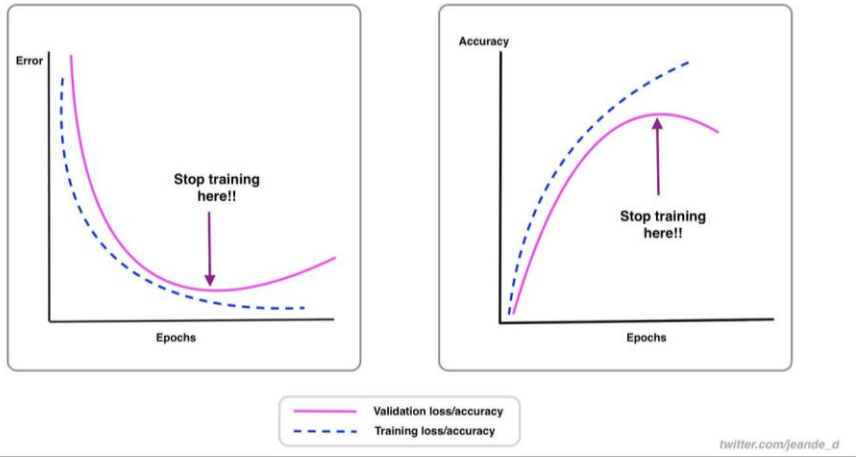
Initialization	Activation functions	σ (normal distribution)
Glorot	Identity, tanh, logistic, softmax	$\frac{1}{\sqrt{fan_{in} + fan_{out}}}$
He	ReLU and variants	$\sqrt{\frac{2}{fan_{in}}}$

Regularization techniques

PyTorch Ignite

Pytorch `EarlyStopping` class

- **Early stopping:**
  - Check the learning curves over time.
  - Stop the training when the model starts to overfit within a specific window, supposing using a validation set.



Regularization techniques

- **L1,L2 Regularization:** facilitate a parcimonious solution.

Lasso regularization (L1-loss): $\mathcal{L}_{total} = \mathcal{L}_{CE} + \frac{\lambda}{2N}\sum_{l=1}^{L}\left\|W^{(l)}\right\|_1$

where $\left\|W^{(l)}\right\|_1 = \sum_{i,j} w_{i,j}$

Ridge regularization (L2-loss): $\mathcal{L}_{total} = \mathcal{L}_{CE} + \frac{\lambda}{2N}\sum_{l=1}^{L}\left\|W^{(l)}\right\|_2^2$

where $\left\|W^{(l)}\right\|_2^2 = \sum_{i,j} w_{i,j}^2$

Regularization techniques

`nn.BatchNorm1d`

Some versions exist to work on top of convolutional layers

- **Batch normalization (BN):** reduce the problem of covariate shift, i.e. changes in the distribution of internal activations during training. By normalizing the pre-activations ($\{Z_l\}$) in each layer l, BN makes training more stable and enables higher learning rates to be used.

At a given layer l:

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1\ldots m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

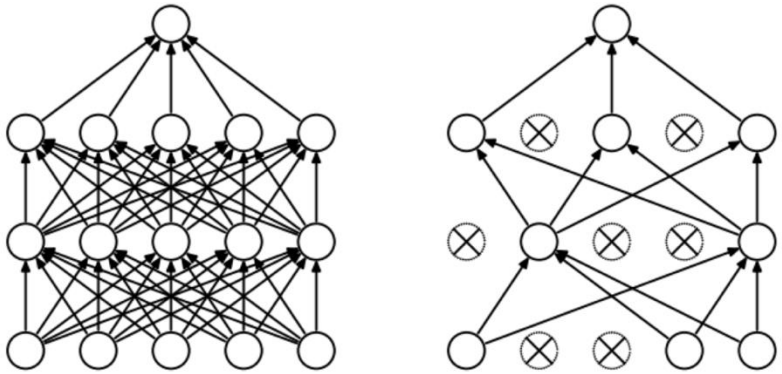
$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

where x_i represents here the pre-activation $z_{i,l}$ for sample i at layer l

During inference or test time, the mean and variance are fixed. They are estimated using the previoulsy calculated means and variances of each training batch.

Regularization techniques

- **Dropout:** `nn.Dropout`
  - Set randomly the activation of some neurons to 0 with a p probability.
  - It forces the network to learn several solutions (improve the robustess).
  - Helpful to limit the problem of vanishing gradients.



(a) Standard Neural Net (b) After applying dropout.

Full pipeline:

```
nn.Linear
nn.BatchNorm1d
nn.ReLU
nn.Dropout
```

Good practice

- Don’t forget your data:
  - Preprocessing is important.
  - Problem of data imbalance and data size.
  - Some models work pretty well with some data: CNN for image data, RNN/Transformers for text data…
  - Assuming you have two datasets A and B of same type (i.e., image), it’s not because your model works well on A that it will work well on A.
  - Split your data in at least three sets: train, test, val. If you don’t have enough data, you can use cross-validation.

Good practice

- Track your training using TensorBoard, WeightsAndBiases, MLFlow…
- Hyper finetune your model using Optuna, Ray Tune…
- Comment your code useful either for yourself or your collaborators
- Start with a simple model and then custom the model

Assignment 1

Overview

1. Neural network: ingredients
   Multi-Layer Perceptron (MLP)

2. **Convolutional Neural Network (CNN)**
   A simple vanilla CNN: LeNet

3. Recurrent Neural Network (RNN)

Convolutional Neural Network (CNN)

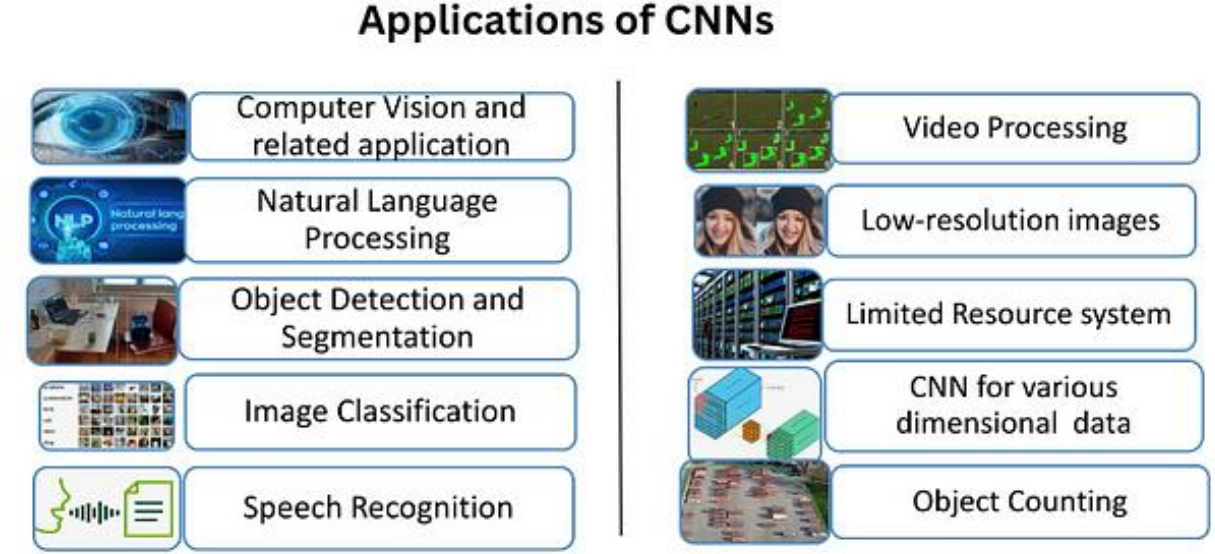
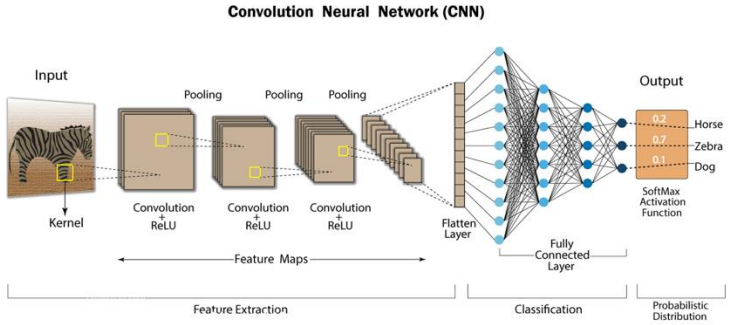
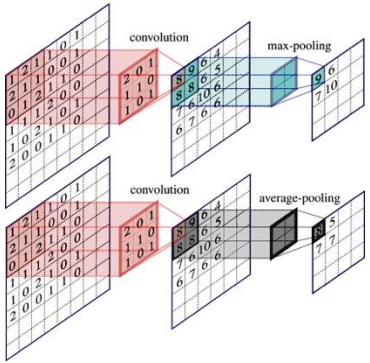


Image source: https://medium.com/@aiblogtech/convolutional-neural-network-a-simple-and-detailed-guide-bd0fbdc8cedb

CNN: main ingredients



- Convolution layer: `nn.Conv2d`
- Pooling layer: `nn.AvgPool2d`, `nn.MaxPool2d`
- Dense layer = MLP



- Input: $(N, C_{in}, H_{in}, W_{in})$ or (C_{in}, H_{in}, W_{in})
- Output: $(N, C_{out}, H_{out}, W_{out})$ or $(C_{out}, H_{out}, W_{out})$, where

$$H_{out} = \left\lfloor \frac{H_{in} + 2 \times \text{padding}[0] - \text{dilation}[0] \times (\text{kernel\_size}[0] - 1) - 1}{\text{stride}[0]} + 1 \right\rfloor$$

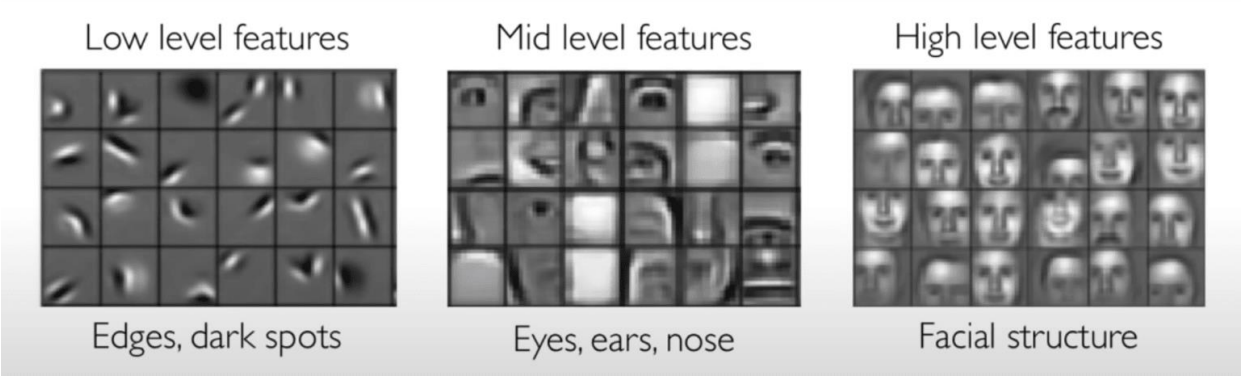
$$W_{out} = \left\lfloor \frac{W_{in} + 2 \times \text{padding}[1] - \text{dilation}[1] \times (\text{kernel\_size}[1] - 1) - 1}{\text{stride}[1]} + 1 \right\rfloor$$

Variables

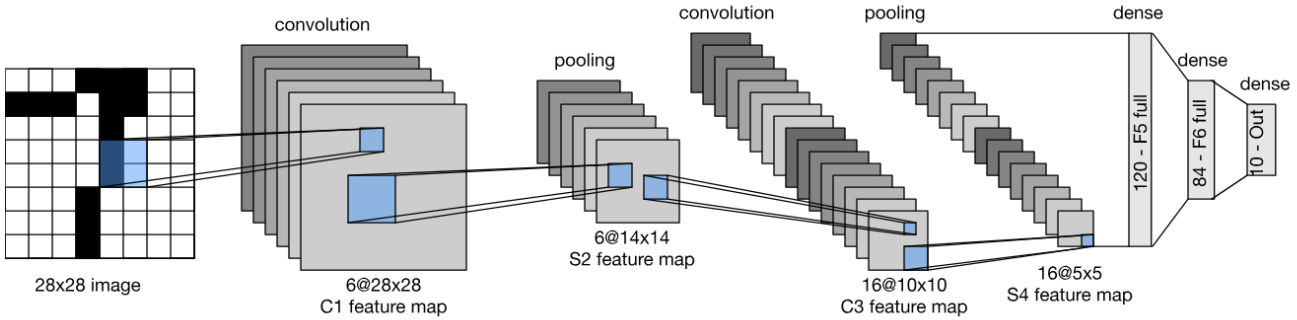
- **weight** (*Tensor*) – the learnable weights of the module of shape $(\text{out\_channels}, \frac{\text{in\_channels}}{\text{groups}}, \text{kernel\_size}[0], \text{kernel\_size}[1])$. The values of these weights are sampled from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$ where $k = \frac{groups}{C_{in} * \prod_{i=0}^{1} \text{kernel\_size}[i]}$
- **bias** (*Tensor*) – the learnable bias of the module of shape (out_channels). If `bias` is `True`, then the values of these weights are sampled from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$ where $k = \frac{groups}{C_{in} * \prod_{i=0}^{1} \text{kernel\_size}[i]}$

CNN: main ingredients

- Features extracted by cnn blocks along the architecture

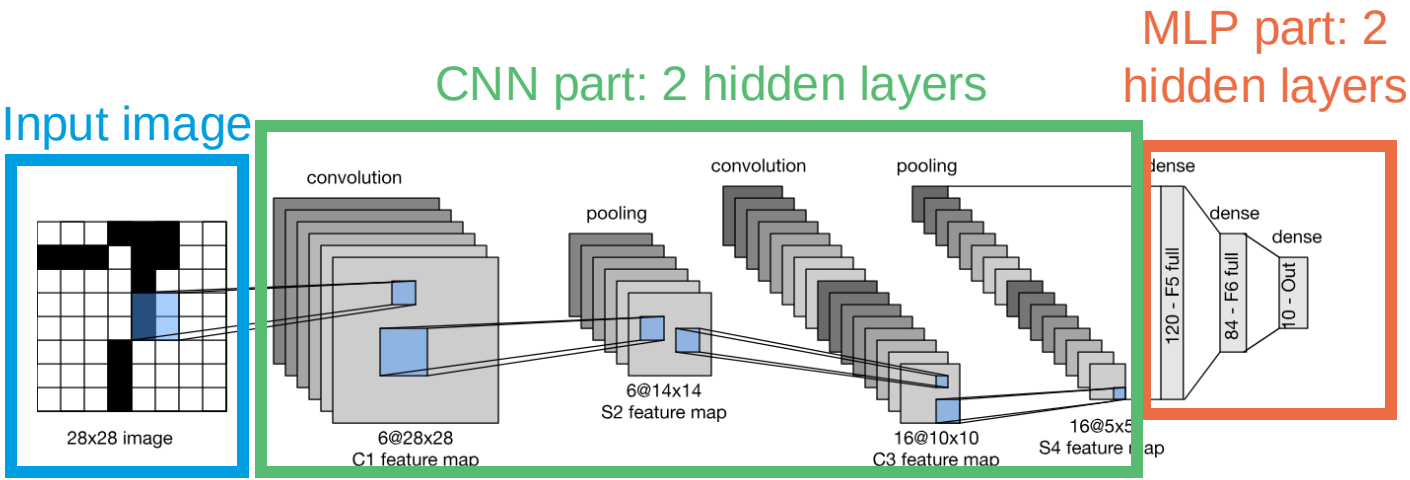


CNN: let’s get familiar with LeNet model



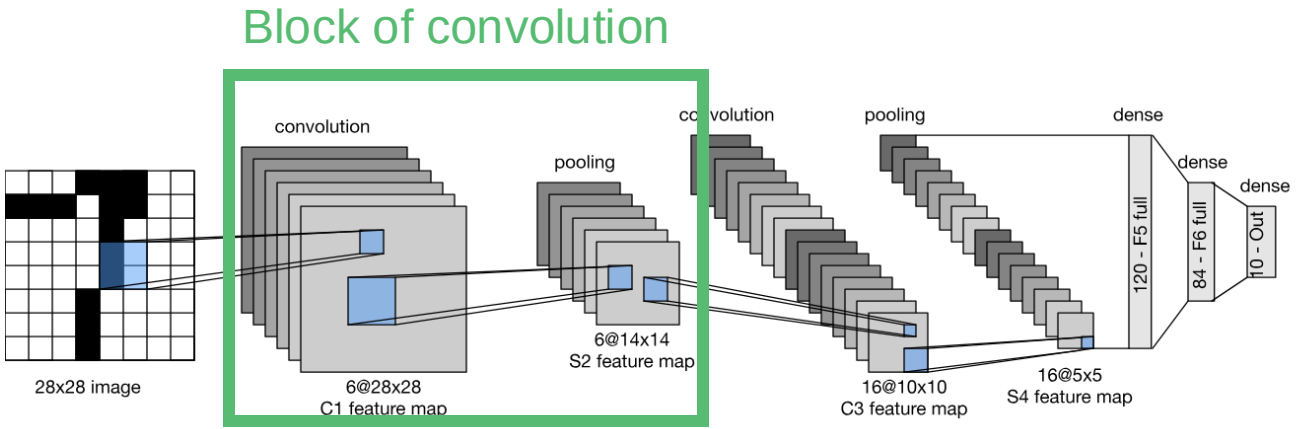
Introduced by Yann LeCun et al. in 1998, the first works date from 1989.

CNN: let’s get familiar with LeNet model

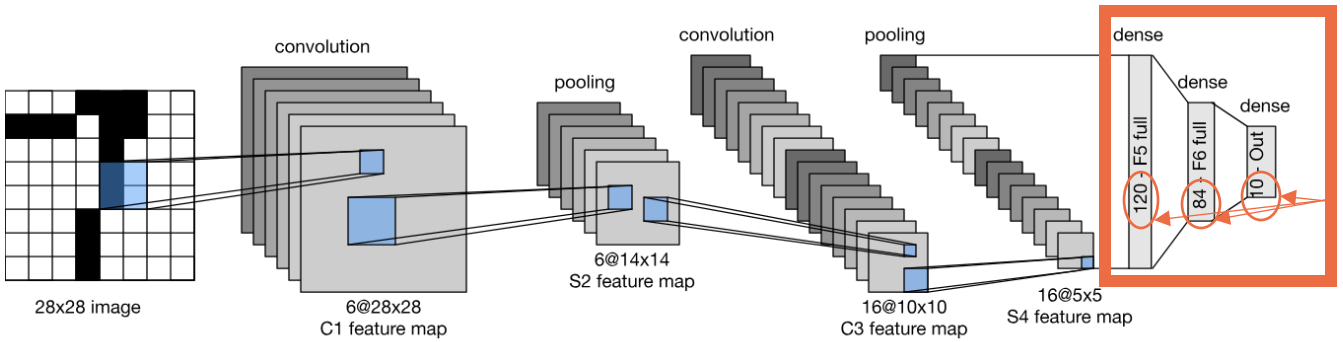


Introduced by Yann LeCun in 1998, the first works date from 1989.

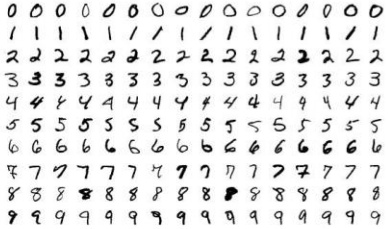
CNN: let’s get familiar with LeNet model



CNN: let’s get familiar with LeNet model



Flatten operation required



Assignment 2

Overview

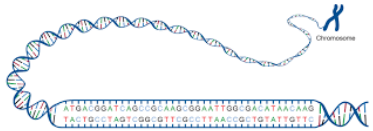
1. Neural network: ingredients
   Multi-Layer Perceptron (MLP)

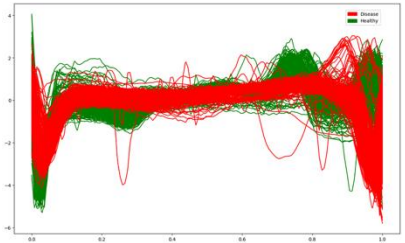
2. Convolutional Neural Network (CNN)
   A simple vanilla CNN: LeNet

3. Recurrent Neural Network (RNN)

Deep Learning for sequential data

What for? Deal with sequential data





- A **sequence** is a observation $\{x^{<t>}\}$ for $t \in \{1; T\}$. Generally, the observation at the time step t depends on the previous observations.

  Some examples: text, DNA, time series…

➔ We need to find a model that considers this temporality and that predicts $y^{<t>}$ for each $x^{<t>}$.

Deep Learning for sequential data

➔ 1D-CNN can be used, but it cannot manage that:

- ➢ the sequences could be of different sizes (the window size is fixed)
- ➢ the predictions could be dependent (no memory effect)

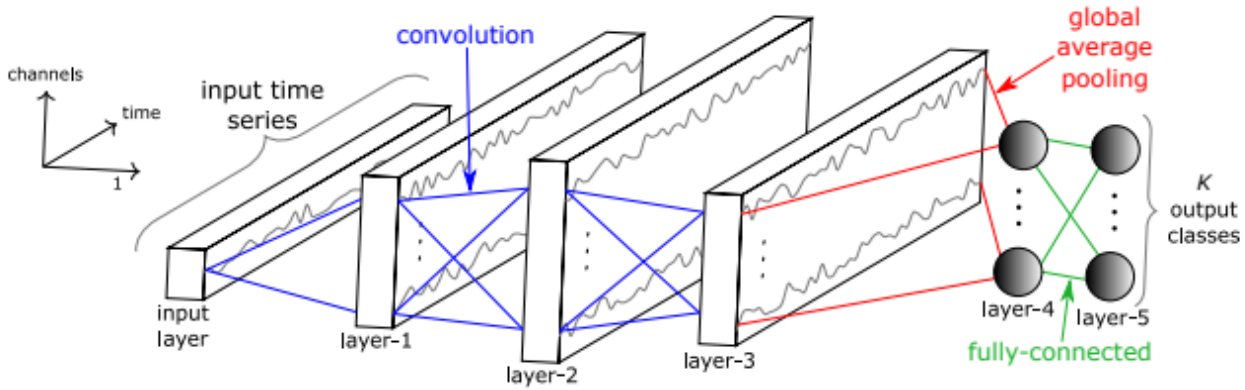
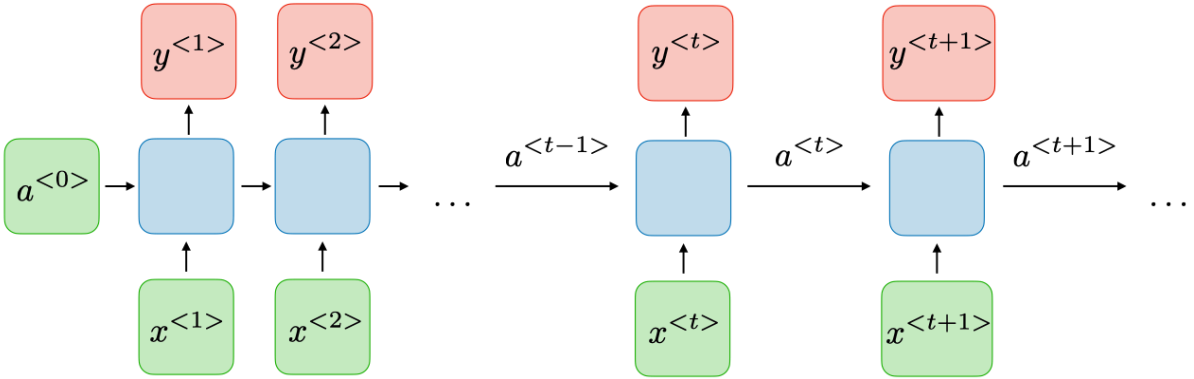


Figure from (Fawaz et al., 2019)

An introduction to RNN (Elman, 1990)



Figures extracted from:
Stanford lecture CS230

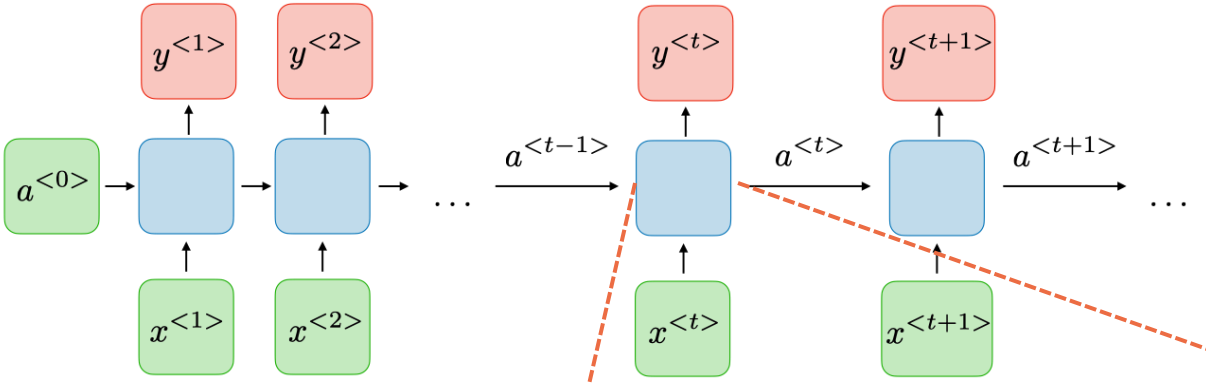
where:

$\{x^{<t>}\}_{t\in\{1;T\}}$ a sequence, $x^{<t>} \in \mathbb{R}^d$

$\{a^{<t>}\}_{t\in\{1;T\}}$ set of hidden states, $a^{<t>} \in \mathbb{R}^l$

$\{y^{<t>}\}_{t\in\{1;T\}}$ set of outputs, $y^{<t>} \in \mathbb{R}$

An introduction to RNN (Elman, 1990)



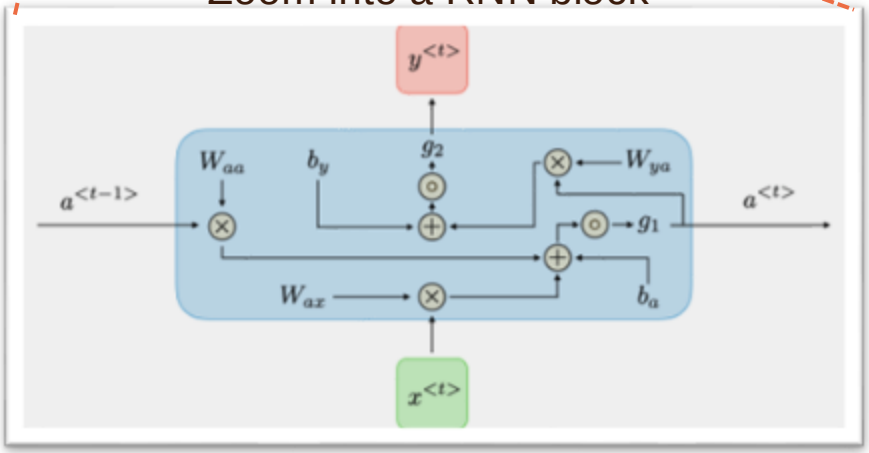
where:

$$a^{<t>} = g_1(a^{<t-1>}, x^{<t>})$$
$$= g_1(W_{aa}a^{<t-1>} + W_{ax}x^{<t>} + b_a)$$

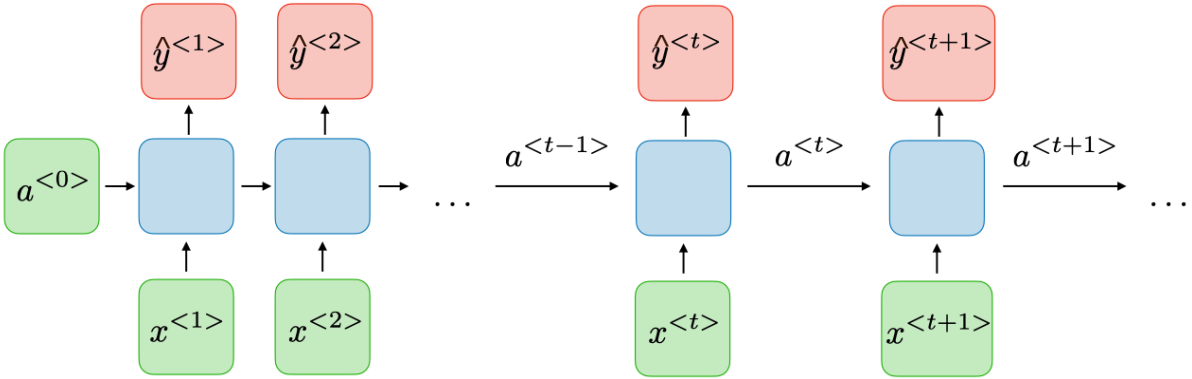
$$y^{<t>} = g_2(W_{ya}a^{<t-1>} + b_y)$$

*shared parameters

Zoom into a RNN block



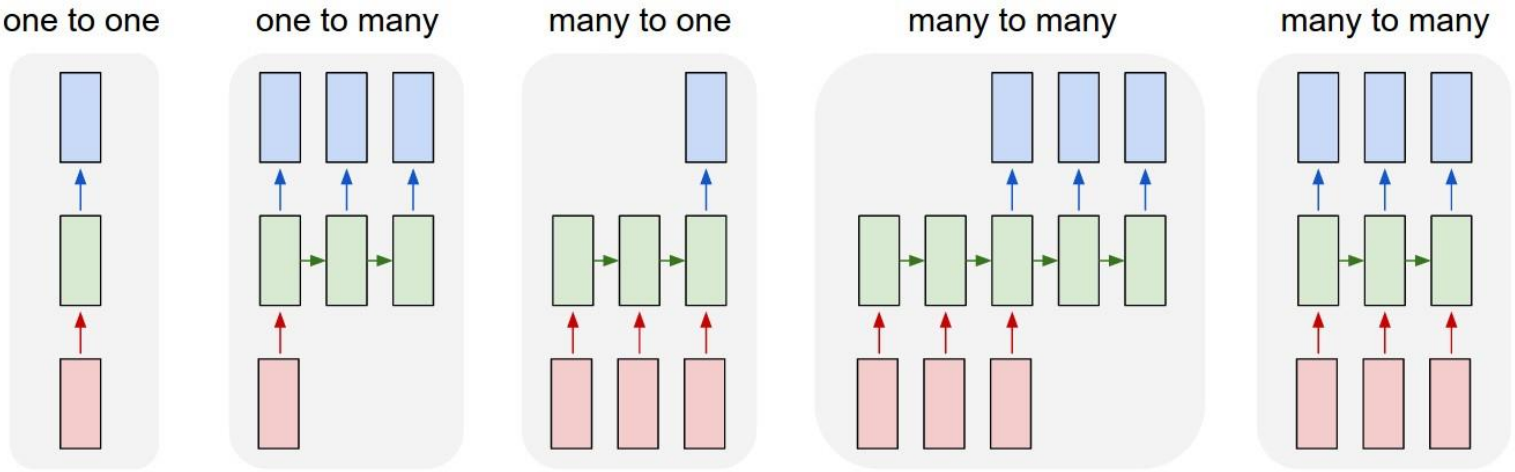
An introduction to RNN (Elman, 1990)



- Loss function in case of classification:
  - at time step t: $\mathcal{L}(\hat{y}^{<t>}, y^{<t>})$
  - total: $\mathcal{L}(\{\hat{y}^{<t>}\}, \{y^{<t>}\}) = \sum_{t=1}^{T} \mathcal{L}(\hat{y}^{<t>}, y^{<t>})$
- The gradient is applied considering the temporality: $\frac{\partial \mathcal{L}}{\partial W} = \sum_{t=1}^{T} \frac{d\mathcal{L}}{dW}\Big|_{(t)}$

The algorithm is called: *backpropagation through time (BPTT)*

Types of RNN


one to one
one to many
many to one
many to many
many to many
Traditional NN
Music generation
Sentiment classification
Machine translation
Name entity recognition

Problems with naive RNN

- Tends to forget old information (distant relationship of unknown length)
- Vanishing gradient problem

Assignment 3

References

- **[Bengio et al., 2017]** Bengio, Y., Goodfellow, I., & Courville, A. (2017). Deep learning (Vol. 1). Cambridge, MA, USA: MIT press.
- **[LeCun et al., 1998]** LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, *86*(11), 2278-2324.
- **[Ismail Fawaz, et al., 2019]** Ismail Fawaz, H., Forestier, G., Weber, J., Idoumghar, L., & Muller, P. A. (2019). Deep learning for time series classification: a review. *Data mining and knowledge discovery*, *33*(4), 917-963.
- **[Elman, 1990]** Elman, J. L. (1990). Finding structure in time. *Cognitive science*, *14*(2), 179-211.