An introduction to deep learning

Claire Boyer

SCIENCES SORBONNE UNIVERSITÉ

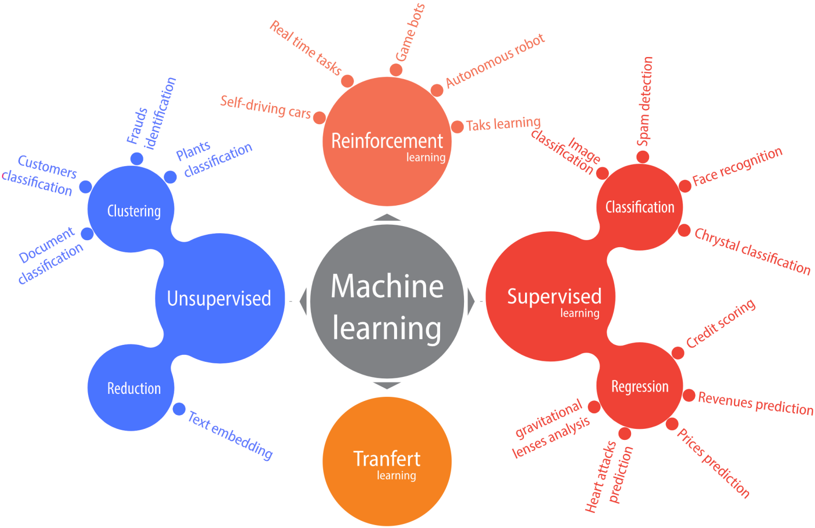
Summary

Learning scenarios

ML develops generic methods for solving different types of problems:

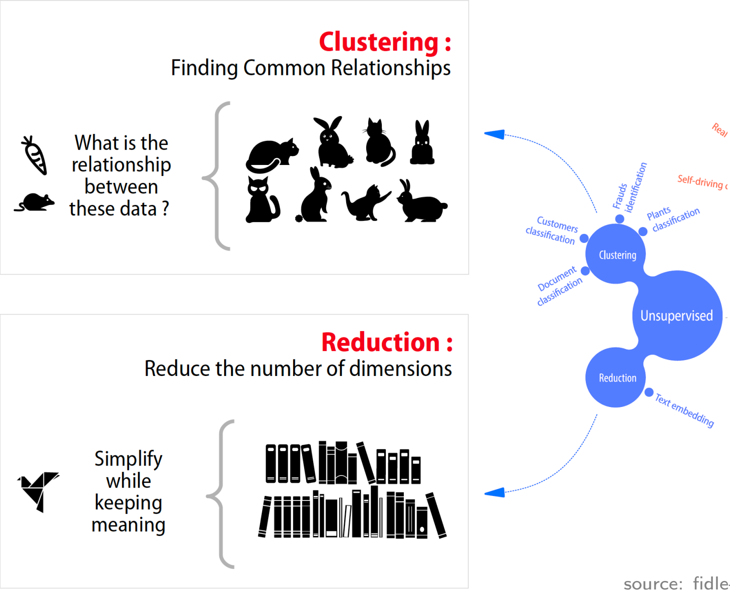
- Supervised learning
  Goal: learn from examples
- Unsupervised learning
  Goal: learn from data alone, extract structure in the data
- Reinforcement learning
  Goal: learn by exploring the environment (e.g. games or autonomous vehicle)

Learning scenarios



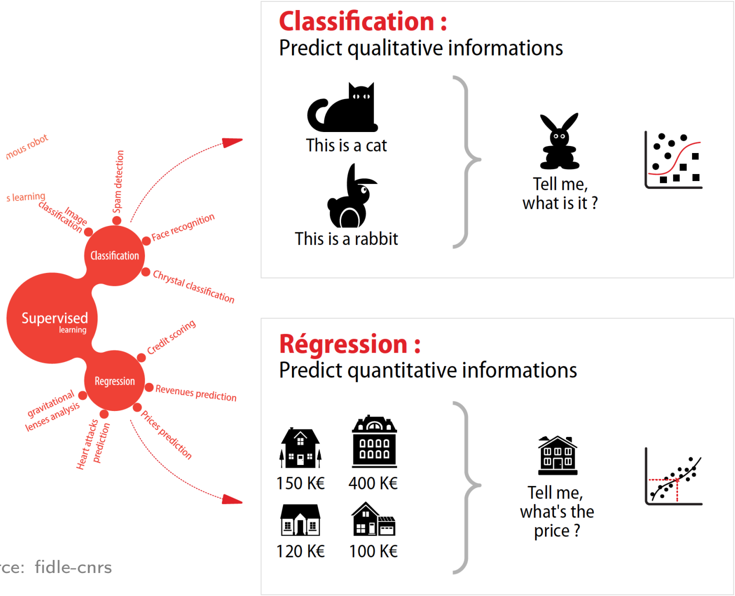
source: fidle-cnrs

Unsupervised learning



source: fidle-cnrs

Supervised learning



source: fidle-cnrs

Supervised learning, more formally

- Supervised learning: given a training sample $(X_i, Y_i)_{1 \leqslant i \leqslant n}$, the goal is to "learn" a predictor f_n such that

$$\underbrace{f_n(X_i) \simeq Y_i}_{\text{prediction on training data}} \qquad \text{and above all} \qquad \underbrace{f_n(X_{\text{new}}) \simeq Y_{\text{new}}}_{\text{prediction on test (unseen) data}}$$

Often

- (classification) $X \in \mathbb{R}^d$ and $Y \in \{-1, 1\}$
- (regression) $X \in \mathbb{R}^d$ and $Y \in \mathbb{R}$

How to measure the performance of a predictor?

- Loss function in general: $\ell(Y, f(\mathrm{X}))$ measures the goodness of the prediction of Y by $f(\mathrm{X})$
- Examples:
  - (classification) Prediction loss: $\ell(Y, f(\mathrm{X})) = 1_{Y \neq f(\mathrm{X})}$
  - (regression) Quadratic loss: $\ell(Y, f(\mathrm{X})) = |Y - f(\mathrm{X})|^2$
- The performance of a predictor f in regression is usually measured through the risk

$$\mathrm{Risk}(f) = \mathbb{E}\Big[\ell\big(Y_{\mathrm{new}}, f(X_{\mathrm{new}})\big)\Big]$$

- A minimizer $f^\star$ of the risk is called a Bayes predictor

Learning by minimizing the empirical risk

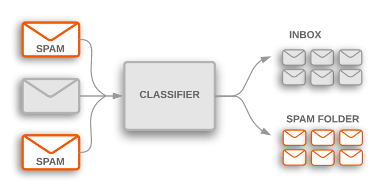
- We want to construct a predictor with a small risk
- The distribution of the data is in general unknown, so is the risk
- Instead, given some training samples $(X_1, Y_1), \ldots (X_n, Y_n)$, find the best predictor f that minimizes the empirical risk

$$\hat{\mathcal{R}}_n(f) := \frac{1}{n} \sum_{i=1}^{n} \ell(Y_i, f(X_i)).$$

- Learning means retrieving information from training data by constructing a predictor that should have good performance on new data

Examples I

Spam detection (Text classification)

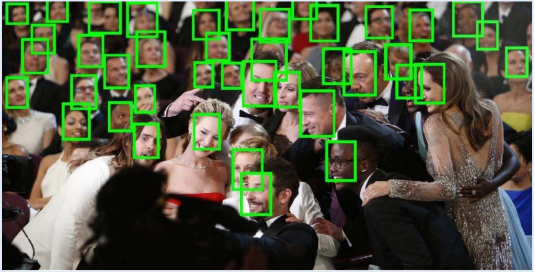


from here

- Data: email collection
- Input: email
- Output : Spam or No Spam

Examples II

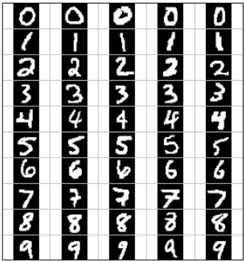
Face Detection



- Data: Annotated database of images
- Input : Sub window in the image
- Output : Presence or no of a face...

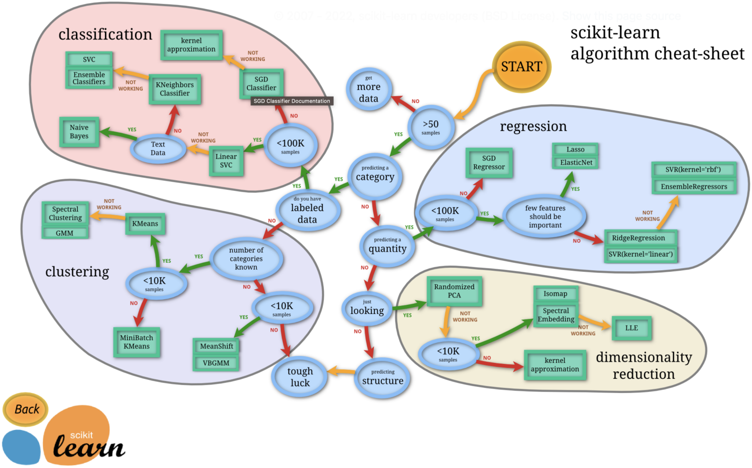
Examples III

Number Recognition



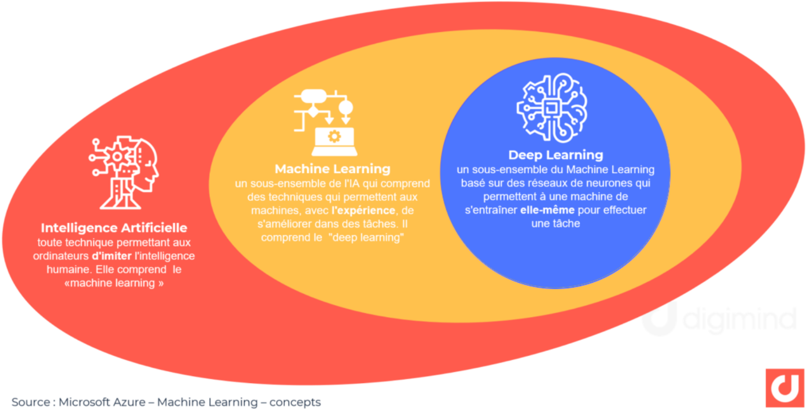
- Data: Annotated database of images (each image is represented by a vector of $28 \times 28 = 784$ pixel intensities)
- Input: Image
- Output: Corresponding number

There exist plenty of learners



see https://scikit-learn.org/stable/tutorial/machine_learning_map/index.html

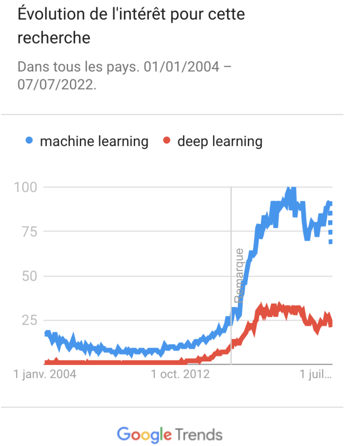
Deep learning is a way to answer your question



Source : Microsoft Azure – Machine Learning – concepts

What is Deep Learning?

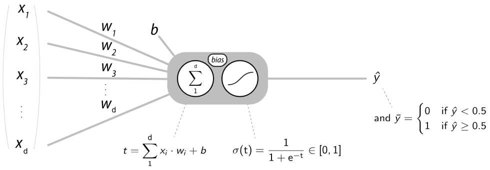
- In the past 10 years, machine learning and artificial intelligence have shown tremendous progress
- Much of the current excitement concerns a subfield of it called "deep learning".
- This recent success can be attributed to:
  - Explosion of data
  - Cheap computing cost – CPUs and GPUs
  - Improvements of machine learning models



Summary

A unit / an artificial neuron

Goal: estimate the function f that links the input X to the output Y, i.e. $Y = f(X)$.
How? Use a single neuron.



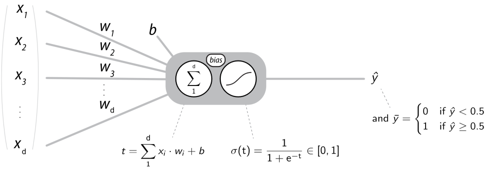
source: fidle.cnrs

- Inputs: x_j
- Weights: w_j
- Bias: b
- Non-linear activation function: σ

$$\sigma(x) = \begin{cases} \mathbb{1}_{x>0} & \text{(perceptron)} \\ \frac{1}{1-\exp(-x)} & \text{(logistic regression)} \end{cases}$$

A unit / an artificial neuron

Goal: estimate the function f that links the input X to the output Y, i.e. $Y = f(X)$.
How? Use a single neuron.



source: fidle.cnrs

- Inputs: x_j
- Weights: w_j
- Bias: b
- Non-linear activation function: σ

$$\sigma(x) = \begin{cases} \mathbb{1}_{x>0} & \text{(perceptron)} \\ \frac{1}{1-\exp(-x)} & \text{(logistic regression)} \end{cases}$$

- Prediction:

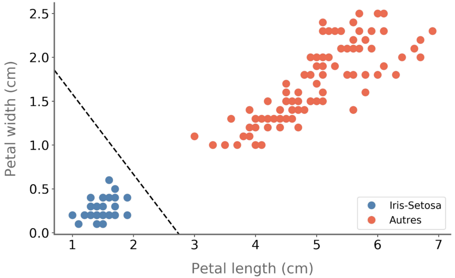
$$\hat{y} = \sigma\left(w_1 x_1 + \ldots + w_d x_d + b\right) = \sigma\left(\sum_{j=1}^{d} w_j x_j + b\right) = \sigma\left(w^\top x + b\right)$$

- A neuron $\equiv$ a nonlinear function applied on a linear function
- Training a neuron $\equiv$ finding the best w, b that fit the training data

What can you do with a single neuron?

Iris plants dataset

Dataset from : Fisher, R.A. "The use of multiple measurements in taxonomic problems" Annual Eugenics, 7, Part II, 179-188 (1936)



Length	Width	Iris Setosa (0/1)
x_1	x_2	y
1.4	1.4	1
1.6	1.6	1
1.4	1.4	1
1.5	1.5	1
1.4	1.4	1
4.7	4.7	0
4.5	4.5	0
4.9	4.9	0
4.0	4.0	0
4.6	4.6	0
(...)		

source: fidle-cnrs

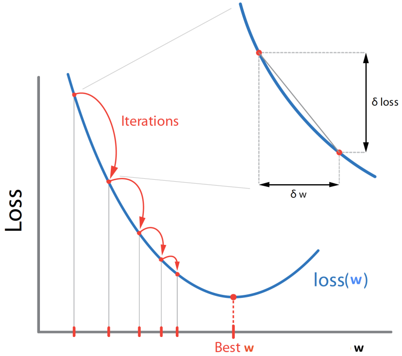
How to find the best parameters?

- Goal: minimize the loss function evaluated on training data

$$\frac{1}{n}\sum_{i=1}^{n} \mathrm{loss}_{w,b}\left(y_i, \hat{y}_i\right)$$

- Problem: there is no explicit minimizer
- Need to reach a minimizer by an iterative procedure

Minimization by gradient descent



- By changing w from ∂w we improve the loss by ∂loss
- The gradient $\nabla \text{loss}(w)$ is the direction of greatest growth of the loss function locally in w
- We follow the "slope" of $-\nabla \text{loss}(w)$ to make the function decrease

$$w \leftarrow w - \eta \nabla \text{loss}(w)$$

equivalently

$$w_j \leftarrow w_j - \eta \frac{\partial \text{loss}}{\partial w_j}(w)$$

- Iterate to minimize the loss function $\rightsquigarrow$ gradient descent

Necessary to compute the whole gradient?

- Note that to update the weights w, one needs to compute n gradients for all the training points: too expensive!

$$\frac{1}{n}\sum_{i=1}^{n}\frac{\partial \text{loss}}{\partial w}(y_i, \hat{y}_i(w))$$

- Instead pick a single training point and backpropagate only the gradient associated to its error

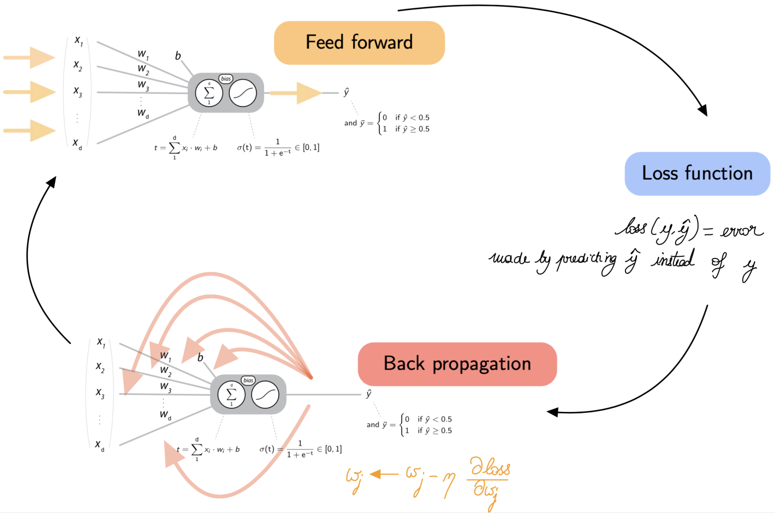
$$w \to w - \eta \cancel{\frac{1}{n}\sum_{i=1}^{n}}\frac{\partial \text{loss}}{\partial w}(y_i, \hat{y}_i(w))$$

- or pick several training points (batch) and backpropagate the gradient associated to their error:

$$w \to w - \eta \frac{1}{\text{batch size}}\sum_{i\in \text{batch}}\frac{\partial \text{loss}}{\partial w}(y_i, \hat{y}_i(w))$$

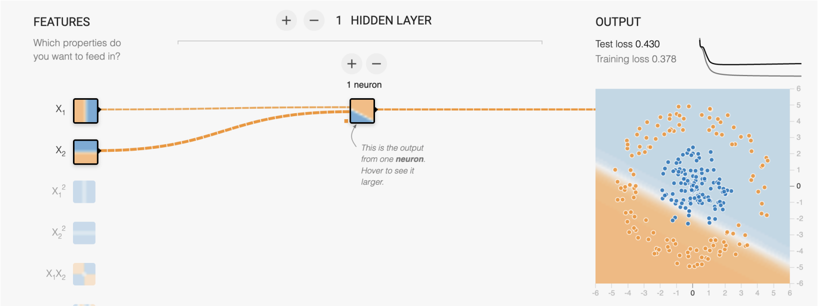
- Convergence to a minimizer is preserved

Training a single neuron: the big picture



- Training of a (slightly more complicated) logistic regression
  https://www.youtube.com/watch?v=kWvwR4ER_UE&ab_channel=TLDRu

Towards more complex decision function

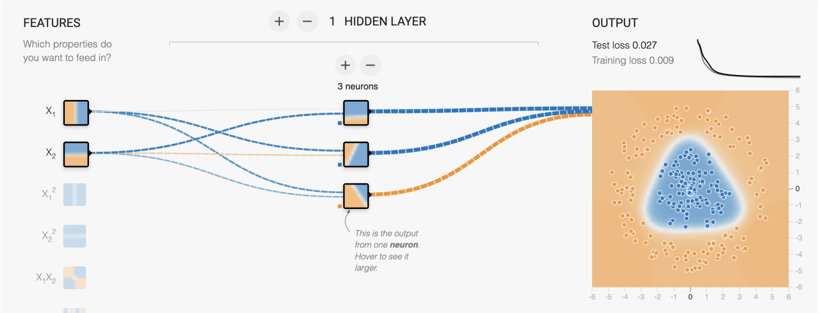


from `https://playground.tensorflow.org/`

A single neuron is not sufficient to classify this data

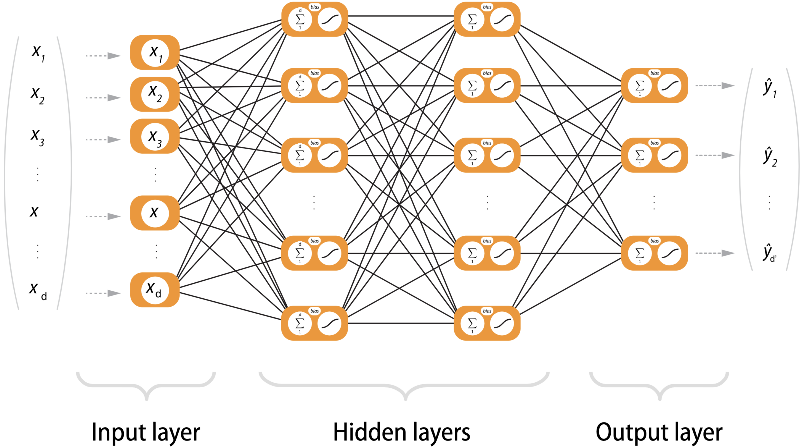
Towards more complex decision function

Idea: stack several neurons and combine their outputs



from `https://playground.tensorflow.org/`

Multi-Layer Perceptron (MLP)

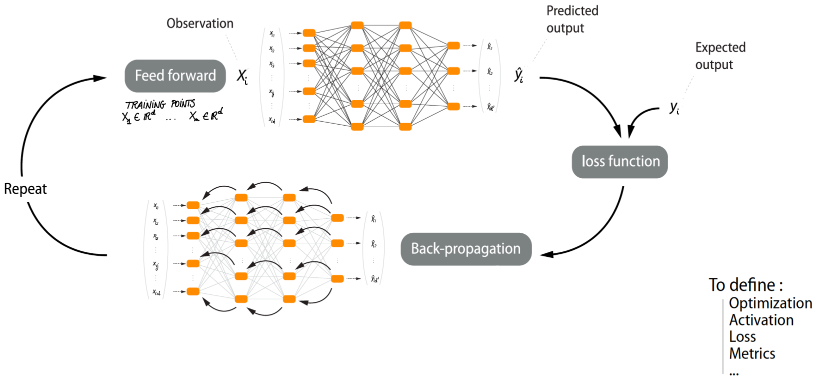


source: fidle-cnrs

- Cascade of linear and nonlinear functions
- Formally

$$\hat{y} = \sigma\left(W_L \sigma\left(W_{L-1} \sigma\left(\ldots \sigma\left(W_1 x\right)\right)\right)\right)$$

Training MLPs



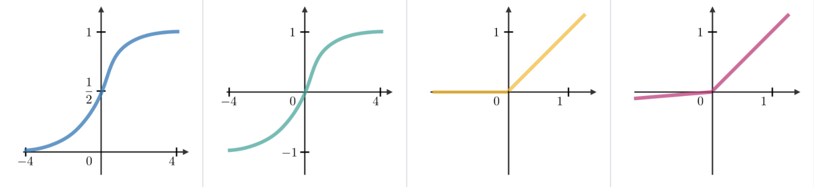
source: fidle-cnrs

Remarks:

1. with a single neuron, backpropagation allows in general to find a global minimizer (if it exists)
2. with MLPs, backpropagation finds only local minimizers (but this is fine in practice)

Zoology of activation functions

Sigmoid	Tanh	ReLU	Leaky ReLU
$g(z) = \dfrac{1}{1+e^{-z}}$	$g(z) = \dfrac{e^z - e^{-z}}{e^z + e^{-z}}$	$g(z) = \max(0, z)$	$g(z) = \max(\epsilon z, z)$ with $\epsilon \ll 1$



from https://stanford.edu/~shervine/teaching/cs-229/cheatsheet-deep-learning

- Most of the time people use the ReLU function (to prevent costly computation, saturation, vanishing gradient, etc.) in MLP
- Sigmoid and tanh are mostly used for recurrent neural networks (see later)

Zoology of output/loss functions

- The output layer depends on the learning task
- The loss metrics depends on the learning task

Task	Nb of neurons in the output layer	Output function	Loss
Regression	1 (or d')	Linear	RMSE $\ell(y, y') = \sqrt{(y - y'))^2}$ MAE $\ell(y, y') = \lvert y - y' \rvert$
Binary classification	1	Sigmoid	Cross-entropy
Multiclass classification	nb K of classes	Softmax $\text{softmax}(z) = \begin{pmatrix} p_1 \\ \vdots \\ p_K \end{pmatrix} = \frac{1}{\sum_{k=1}^K e_k^z} \begin{pmatrix} e^{z_1} \\ \vdots \\ e^{z_K} \end{pmatrix}$	Cross-entropy $\sum_{k=1}^K \mathbb{1}_{y=k} \log(p_k)$

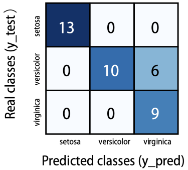
For an exhaustive list, see `https://scikit-learn.org/stable/modules/model_evaluation.html#regression-metics`

Zoology of output/loss functions

- The output layer depends on the learning task
- The loss metrics depends on the learning task

Task	Nb of neurons in the output layer	Output function	Loss
Regression	1 (or d')	Linear	RMSE $\ell(y, y') = \sqrt{(y - y'))^2}$ MAE $\ell(y, y') = \lvert y - y' \rvert$
Binary classification	1	Sigmoid	Cross-entropy
Multiclass classification	nb K of classes	Softmax $\text{softmax}(z) = \begin{pmatrix} p_1 \\ \vdots \\ p_K \end{pmatrix} = \frac{1}{\sum_{k=1}^K e_k^z} \begin{pmatrix} e^{z_1} \\ \vdots \\ e^{z_K} \end{pmatrix}$	Cross-entropy $\sum_{k=1}^K \mathbb{1}_{y=k} \log(p_k)$

For an exhaustive list, see `https://scikit-learn.org/stable/modules/model_evaluation.html#regression-metics`



For visualization in classification, use confusion matrices!

Zoology of optimizers

- Training a neural network is very challenging
- A key ingredient is the optimizer you use to find a local optimum
- They are all based on stochastic gradient descent (SGD)
  - with adaptive learning rates: try to retrieve second-order information (Hessian) only based on first-order information (gradient)
  - with momentum: gradient memory

`https://distill.pub/2017/momentum/`

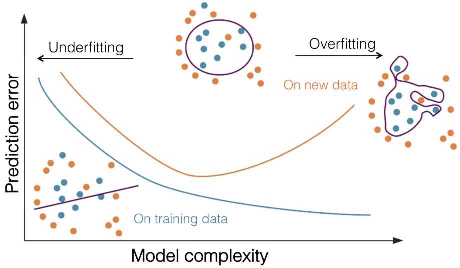
$$\text{SGD} \leqslant \underbrace{\text{Adagrad/RMSProp}}_{\text{adaptive learning rates}} \leqslant \underbrace{\text{ADAM}}_{\text{with momentum}}$$

Bias-variance tradeoff

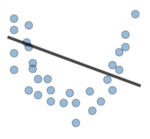
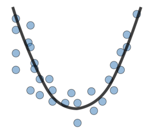
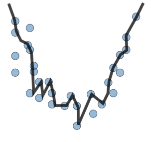
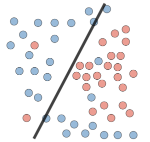
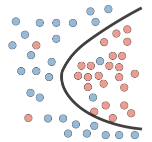
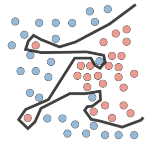
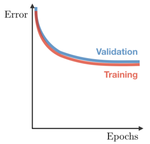
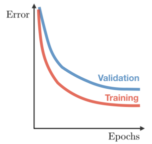
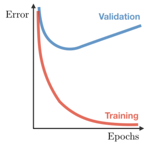
- Usually the constructed predictor f_n is constrained to live in a class $\mathcal{F}$ of functions
- Complexity of the model $\equiv$ Size of $\mathcal{F}$
- Learning always implies to tune hyper-parameters (NN architecture, etc.)
- How to tune them?
  Statistical wisdom: take care of the so-called bias-variance tradeoff

Bias: systematic error, the predictor model is too simple to grasp data complexity

Variance: how much the predictions for a given point vary between different realizations of the model



Bias-variance tradeoff

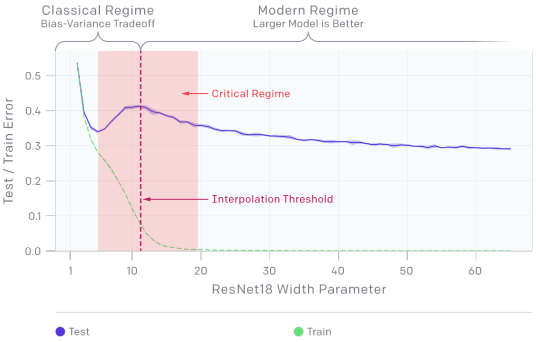
	Underfitting	Just right	Overfitting
Symptoms	• High training error • Training error close to test error • High bias	• Training error slightly lower than test error	• Very low training error • Training error much lower than test error • High variance
Regression illustration			
Classification illustration			
Deep learning illustration			
Possible remedies	• Complexify model • Add more features • Train longer		• Perform regularization • Get more data

from here

Going beyond the traditional bias-variance tradeoff

New insights in the parametric world: adding another billion parameters to a neural network improves the predictive performances

[OpenAI, Deep Double Descent, Nakkiran et al. 2021]



Double descent phenomenon at least well-understood in linear models

[Hastie et al. 2019]

Over-parametrization in neural networks

The risk can be always decomposed as follows

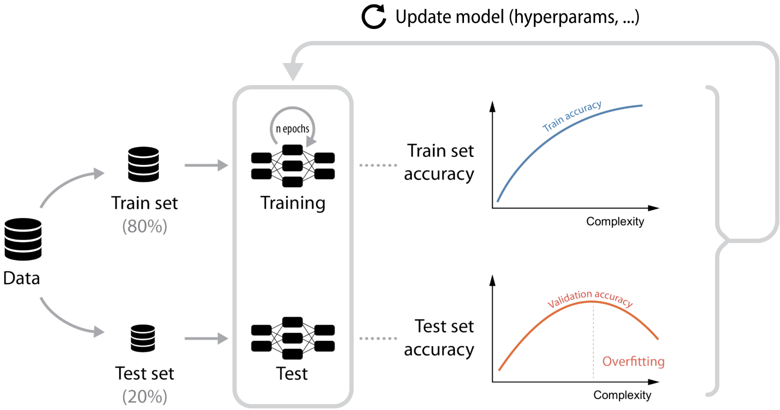
$$\text{Risk} = \text{approximation error} + \text{estimation error} + \text{optimisation error}$$

Why does not overparametrization hurt NN training ?

- approximation error: more parameters, better approx capacities
- optimisation error: more parameters, nicer optimisation space
  [NGuyen et al. 2019, Nguyen 2020]
- estimation error: more parameters, implicit regularisation
  [Deep learning: a statistical viewpoint, Bartlett, Montanari, Rakhlin, 21]

Evaluation strategies

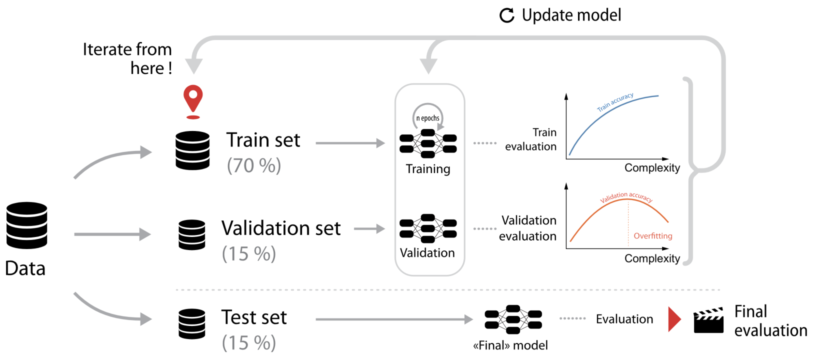
- Idea: always monitor the generalization of the trained algorithm
- Train/test splitting



source: fidle-cnrs

- Problem: we adapt our model according to the test data! $\rightsquigarrow$ Bias

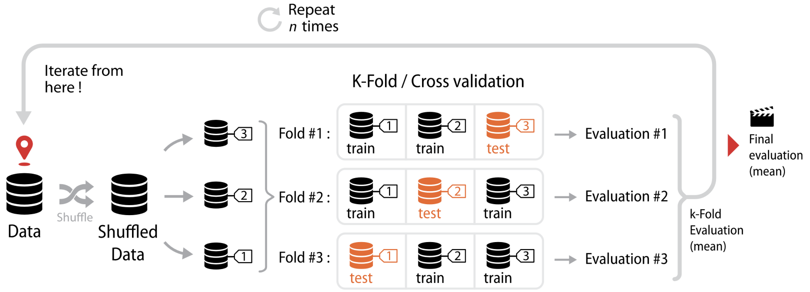
Hold-out validation



source: fidle-cnrs

- OK for large datasets
  otherwise val and test sets too small $\Longrightarrow$ unstable evaluation

K-fold cross-validation



source: fidle-cnrs

- Probably the best strategy for datasets of reasonable size
- Choose $K = 5$ or 10

Summary

Image representation

- Represented as 2D (grayscale) or 3D (color) arrays (tensors)
- Integers between 0 and 255

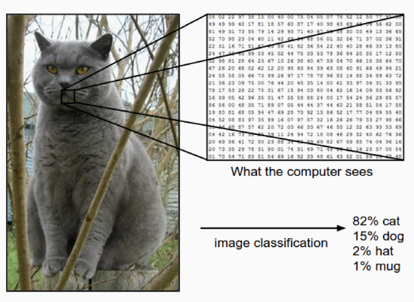
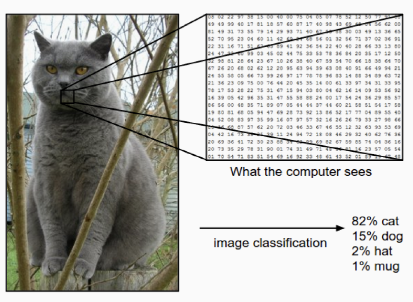
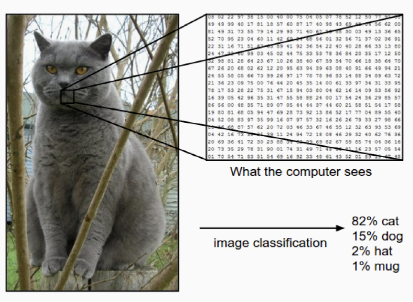


Image representation



- Represented as 2D (grayscale) or 3D (color) arrays (tensors)
- Integers between 0 and 255
- Difficulty: semantic gap between representation and its content

Image representation



- Represented as 2D (grayscale) or 3D (color) arrays (tensors)
- Integers between 0 and 255

- Difficulty: semantic gap between representation and its content

- Reminder: when designing a network, one should inject prior knowledge about intrinsic regularities of the data
- A same feature/object can be localized anywhere in the image, it should be detected regardless of its position

Convolutional neural networks

Classical networks for images?

- Computational cost!

  Ex: a 400x400 pixel RGB image as input, followed by 1000 hidden neurons, for 10-class classification

  Number of trainable parameters $\simeq$ 500 million
- Loss of context: MLP do not take into account the spatial organization of pixels
  - Non robust to image shifting
  - If the pixels are permuted, the output of the network would be the same, whereas the image would change drastically
- Confounding features: in MNIST, only one object per image, this is not the case in real images

Convolutional neural networks

Classical networks for images?

- Computational cost!

  Ex: a 400x400 pixel RGB image as input, followed by 1000 hidden neurons, for 10-class classification

  Number of trainable parameters $\simeq$ 500 million
- Loss of context: MLP do not take into account the spatial organization of pixels
  - Non robust to image shifting
  - If the pixels are permuted, the output of the network would be the same, whereas the image would change drastically
- Confounding features: in MNIST, only one object per image, this is not the case in real images

Idea

- Apply local transformation to a set of nearby pixels (spatial nature of image is used)
- Repeat this transformation over the whole image (resulting in a shift-invariant output).

Convolutional neural networks

Classical networks for images?

- Computational cost!

  Ex: a 400x400 pixel RGB image as input, followed by 1000 hidden neurons, for 10-class classification

  Number of trainable parameters $\simeq$ 500 million
- Loss of context: MLP do not take into account the spatial organization of pixels
  - Non robust to image shifting
  - If the pixels are permuted, the output of the network would be the same, whereas the image would change drastically
- Confounding features: in MNIST, only one object per image, this is not the case in real images

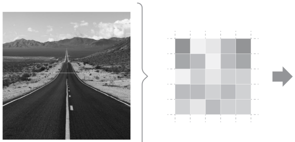
Idea

- Apply local transformation to a set of nearby pixels (spatial nature of image is used)
- Repeat this transformation over the whole image (resulting in a shift-invariant output).

Convolutional neural networks

- Replace the standard matrix products by a convolution

A 2D convolution



By Jan Kroon, from Pexels.com

5	2	1	3	5
4	3	2	3	4
0	2	1	2	2
3	3	2	3	2
2	1	3	2	2

Image piece

5	2	1
4	3	2
0	2	1

X

⊗

Kernel 3x3

1	0	1
0	1	0
1	0	1

ω

=

$\boxed{10}$

y

$$
\begin{aligned}
y \;=\;& 5x1 + 2x0 + 1x1 \\
+\;& 4x0 + 3x1 + 2x0 \\
+\;& 0x1 + 2x0 + 1x1 \;=\; 10
\end{aligned}
$$

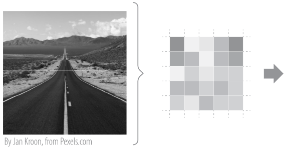
$$y = \sum_{i=1}^{n} \sum_{j=1}^{m} x_{i,j} \cdot \omega_{i,j} \quad \text{with} \begin{cases} n & \text{kernel width} \\ m & \text{kernel height} \end{cases}$$

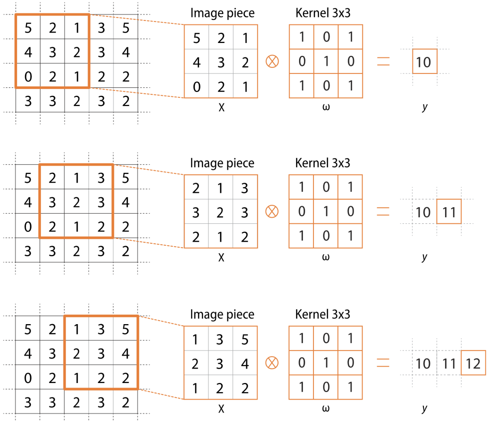
2D convolution

⊗ Hadamard product

source: fidle-cnrs

A 2D convolution

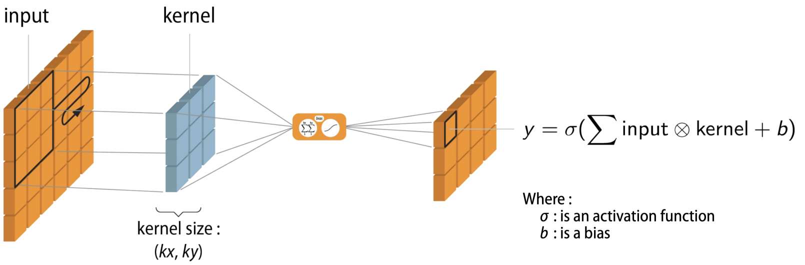




We can perform convolutions in 1, 2, 3 ...or d-dimensional spaces !

source: fidle-cnrs

A convolution layer

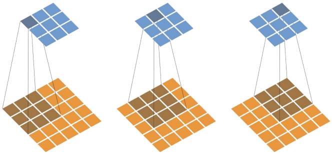


source: fidle-cnrs

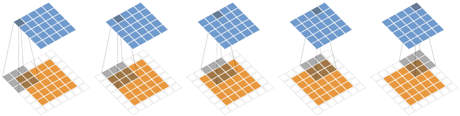
- Number of parameters in a convolution layer : $kx \times ky + 1$
- Parameter sharing instead of full connection
  - improved memory
  - statistical efficiency
  - faster computations
- The kernel will be learned (as the weights in fully connected layers)

Parameters of a convolution layer

1. Size of the kernel (K)
2. Zero-padding (P)



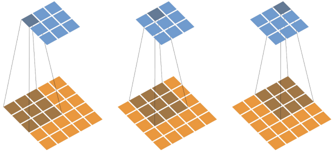
Valid convolution: no padding



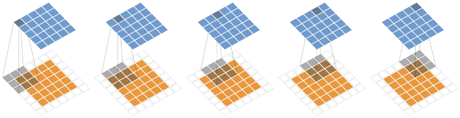
Here the size is preserved

Parameters of a convolution layer

1. Size of the kernel (K)
2. Zero-padding (P)

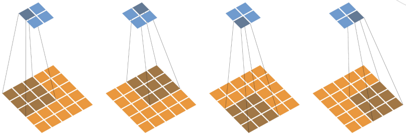


Valid convolution: no padding



Here the size is preserved

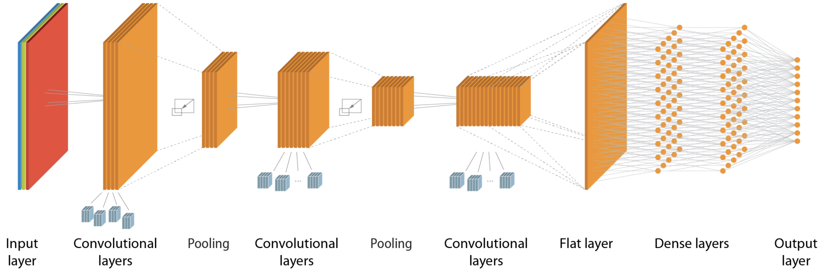
3. Stride (S): how many pixels the filter is moved horizontally and vertically



- Relation between these hyperparameters

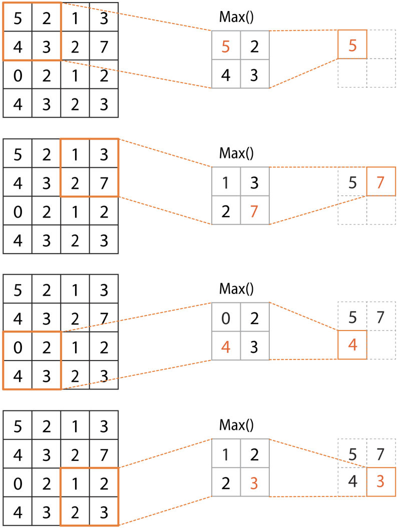
$$\text{output size} = \left\lfloor \frac{2P + \text{input size} - K}{S} \right\rfloor + 1$$

Convolutional neural networks

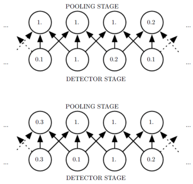


- Use several convolution kernels in parallel to get different structures in the image

Max pooling



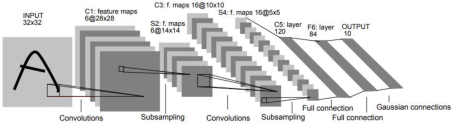
- The pooling layer operates independently on every depth slice of the input and resizes it spatially, using the max function
- Replaces the outut at a certain location by a summary statistics of neighbouring outputs
- Helps the representation to be approximately invariant to small translation in the input



Input is shifted by 1, only half of the pooling output is changed

Some famous ConvNets

- LeNet [LeCun, Bottou, Bengio, Haffner, 1998]



- AlexNet (2012), like LeNet with more layers
- VGGNet (2014), similar , bigger
- GoogleNet (2014), "all-convolutional network" (no fully connected layers anywhere, except the final classification)
- After 2015, residual blocks: use the layers to model differences. In some sense, each successive layer would predict “new information” that was not already previously extracted

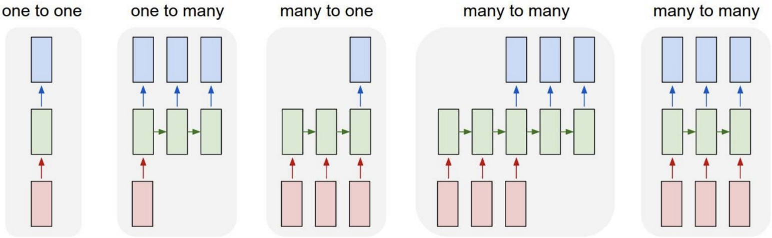
$$h_{\ell+1} = h_\ell + \sigma(W_\ell h_\ell)$$

$\rightsquigarrow$ represented by skip connections

Summary

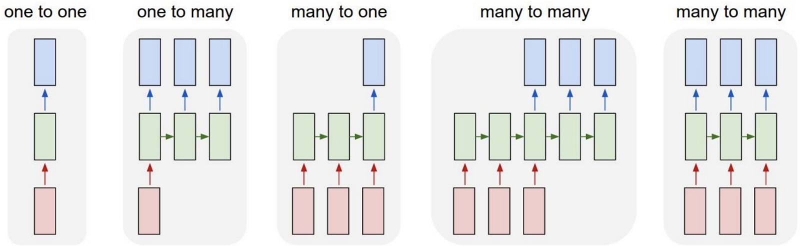
Recurrent neural networks

- Recurrent Neural Networks (RNNs) are Artificial Neural Networks that can deal with sequences of variable size.



RNNs are general computers which can learn algorithms to map input sequences to output sequences (flexible-sized vectors). The output vector's contents are influenced by the entire history of inputs.

Different uses of recurrent neural networks



- Vanilla Neural Networks
- Image classification (one-to-one)
- Image Captioning (one-to-many): image/sequence of words
- Sentiment classification (many-to-one): sequence of words/sentiment
- Translation (many-to-many): sequence of words/sequence of words
- Video classification on frame level (many-to-many): sequence of image/sequence of label

Language generating NN: training

from Charles Deledalle's lectures

How to learn "`The cat is in the kitchen drinking milk.`"?

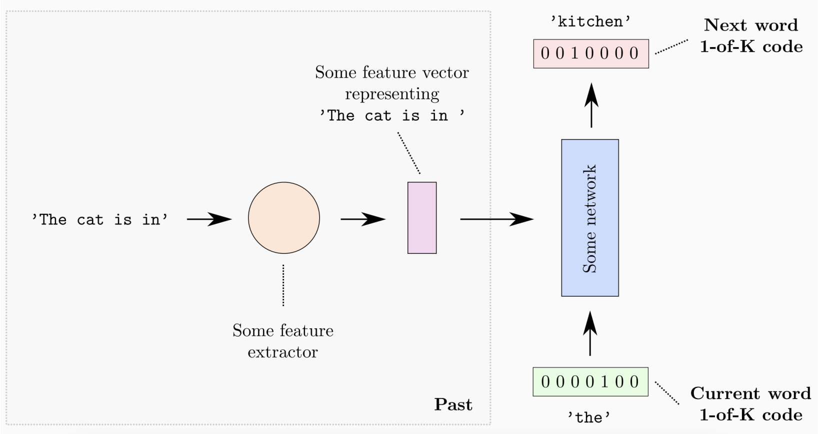
- Word: a 1-to-K code (large dictionaries of K words)
- Learn: $\mathbb{P}\,(\text{next word}|\text{current word and past})$
- Represent the past as a feature vector

Language generating NN: training

from Charles Deledalle's lectures

How to learn "`The cat is in the kitchen drinking milk.`"?

- Word: a 1-to-K code (large dictionaries of K words)
- Learn: $\mathbb{P}$(next word|current word and past)
- Represent the past as a feature vector



Language generating NN: training

from Charles Deledalle's lectures

How to learn "`The cat is in the kitchen drinking milk.`"?

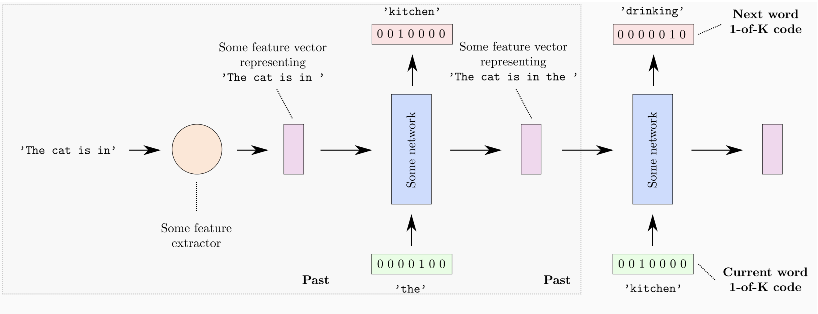
- Word: a 1-to-K code (large dictionaries of K words)
- Learn: $\mathbb{P}\left(\text{next word} \middle| \text{current word and past}\right)$
- Represent the past as a feature vector

Language generating NN: training

from Charles Deledalle's lectures

How to learn "`The cat is in the kitchen drinking milk.`"?

- Word: a 1-to-K code (large dictionaries of K words)
- Learn: $\mathbb{P}$(next word|current word and past)
- Represent the past as a feature vector

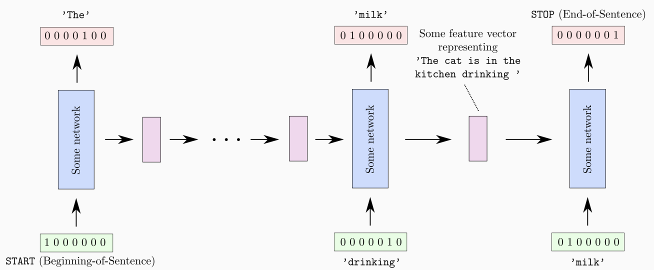


- Learn also how to represent the current sentence
- Repeat for the next word

Language generating NN: training

from Charles Deledalle's lectures

- Add two words: START and STOP to delimitate the sentence
- Learn everything end-to-end on a large corpus of sentences
- Minimize the sum of the cross-entropy of each word (maximum likelihood)
- Intermediate feature will learn how to memorize the past/context/state

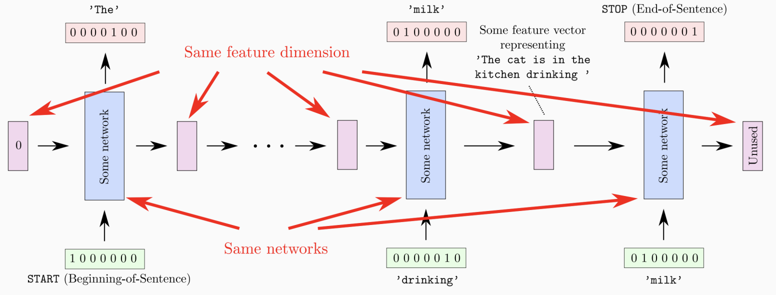


- How should the network architecture and size of intermediate features evolve with the location in the sequence?

Language generating NN: training

from Charles Deledalle's lectures

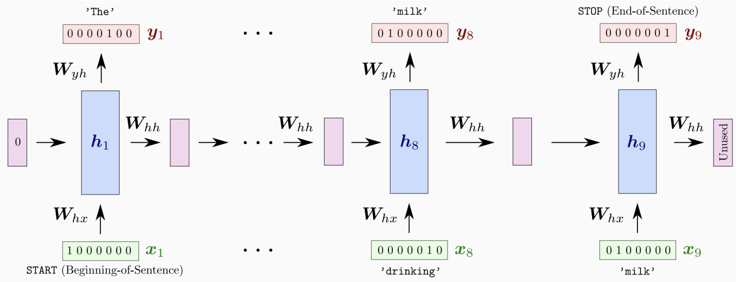
- Use the same networks and the same feature dimension
- The past is always embedded in a fix-sized feature
- Set the first feature as a zero tensor



- Allows you to learn from arbitrarily long sequences
- Sharing the architecture ⇒ fewer parameters ⇒ training requires less data and the final prediction can be expected to be more accurate

A simple shallow RNN for sentence generation

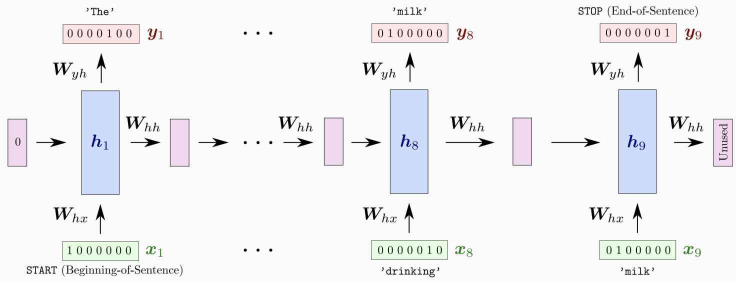
- This is an unfolded representation of an RNN



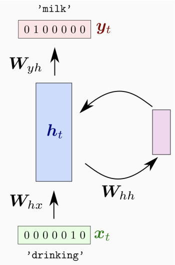
$$h_t = g\left(W_{hx}x_t + W_{hh}h_{t-1} + b_h\right)$$
$$y_t = \text{softmax}\left(W_{yh}h_t + b_y\right)$$

A simple shallow RNN for sentence generation

- This is an unfolded representation of an RNN

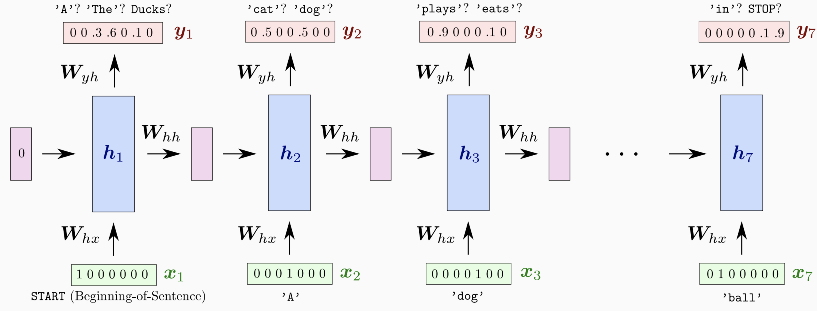


$$h_t = g\left(W_{hx}x_t + W_{hh}h_{t-1} + b_h\right)$$
$$y_t = \text{softmax}\left(W_{yh}h_t + b_y\right)$$



- Folded representation: RNN $\equiv$ ANN with loops

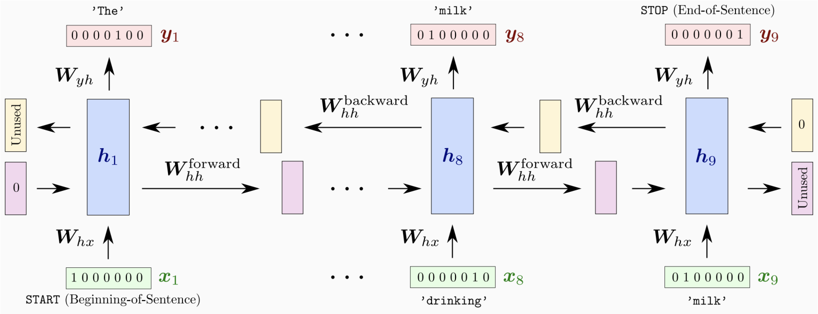
Generate a sentence in practice



- Provide `START`, get all the probabilities $\mathbb{P}\left(\text{next word}|\text{current word} = \texttt{START}\right)$
- Select one of these words according to their probabilities, let say '`A`',
- Provide '`A`' and the past, and get $\mathbb{P}\left(\text{next word}|\text{current word} = \texttt{A}\right)$
- Repeat while generating the sentence 'A dog plays with a ball'
- Stop as soon as you have picked `STOP`.

Bidirectional RNN

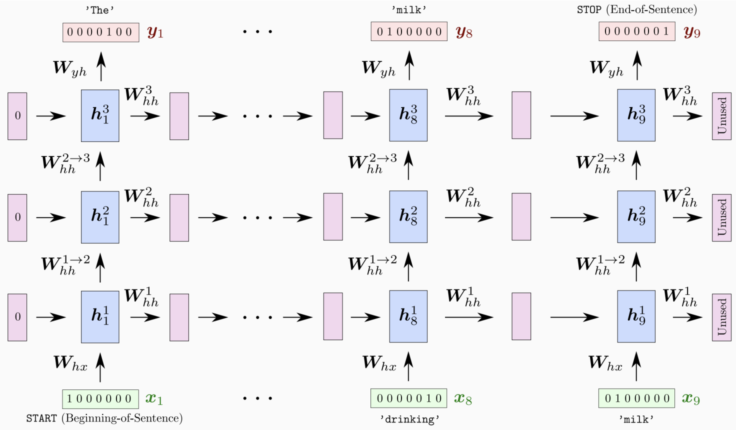
- Output at time t may not only depend on the previous elements, but also on future elements



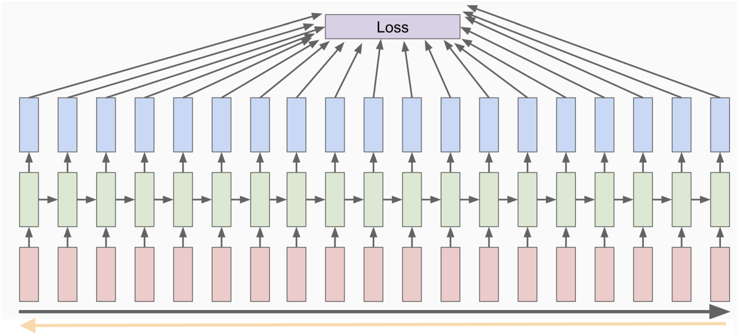
$$h_t = g\left(W_{hx}x_t + W_{hh}^{\text{forward}} h_{t-1} + W_{hh}^{\text{backward}} h_{t+1} b_h\right)$$
$$y_t = \text{softmax}\left(W_{yh}h_t + b_y\right)$$

Deep RNN

- Multiple layers per time step (a feature hierarchy)
- Higher learning capacity
- Requires a lot more training data



Learning phase for RNN

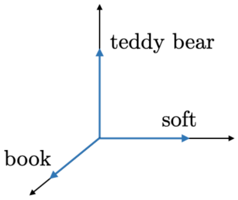
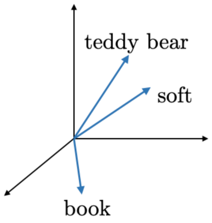


- Similar to standard backprop for training a traditional NN
- Take into account that parameters are shared by all steps in the network
- Forward through the entire sequence to compute the loss
- Backward through the entire sequence to compute gradients

What I left under the carpet: word embedding

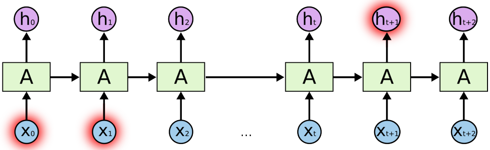
- How to represent words as input?

- Naive way: one-hot encoding
  no similarity information
- Improved way: word-embedding as `word2vec`
  takes into account words similarity

1-hot representation	Word embedding
	

Language generating RNN: limitations

- Vanilla RNN have difficulties learning long-term dependencies



`I grew up in France ...  I speak fluent ???`

(we need the context of France from further back)

- Vanishing/exploding gradient problem

$$\underbrace{\left\| \frac{\partial h_t}{\partial h_{t-1}} \right\|}_{\|W_{hh}^\top \mathrm{diag}(\sigma'(W_{hh}h_{t-1}+W_{xh}x_t))\|} \sim \eta \Longrightarrow \left\| \prod_{t=k+1}^{T} \frac{\partial h_t}{\partial h_{t-1}} \right\| \sim \eta^{T-k}$$

- As $T - k$ increases, the contribution of the k-th term to the gradient decreases exponentially fast
- Certain types of RNNs are specifically designed to get around them

GRU (Gated Recurrent Unit)

- By interpretring the state as the memory of a recurrent unit, we would like to decide whether certain units are worth memorizing (in which case the state is updated), and others are worth forgetting (in which case the state is reset)

GRU (Gated Recurrent Unit)

- By interpretring the state as the memory of a recurrent unit, we would like to decide whether certain units are worth memorizing (in which case the state is updated), and others are worth forgetting (in which case the state is reset)
- Define two gating operations, called "reset" and "update":

$$r_t = \sigma\left(W_{rx} x_t + W_{rh} h_{t-1}\right) \qquad z_t = \sigma\left(W_{zx} x_t + W_{zh} h_{t-1}\right)$$

- Instead of $h_t = \sigma\left(W_{hx} x_t + W_{hh} h_{t-1}\right)$, consider

$$\tilde{h}_t = \sigma\left(W_{hx} x_t + W_{hh}\left(h_{t-1} \odot r_t\right)\right)$$

  - If the reset gate $\simeq 1$, then this looks like a regular RNN unit (i.e., we retain memory)
  - If the reset gate $\simeq 0$, then this looks like a regular perceptron/dense layer (i.e., we forget)

GRU (Gated Recurrent Unit)

- By interpretring the state as the memory of a recurrent unit, we would like to decide whether certain units are worth memorizing (in which case the state is updated), and others are worth forgetting (in which case the state is reset)
- Define two gating operations, called "reset" and "update":

$$r_t = \sigma\left(W_{rx}x_t + W_{rh}h_{t-1}\right) \qquad z_t = \sigma\left(W_{zx}x_t + W_{zh}h_{t-1}\right)$$

- Instead of $h_t = \sigma\left(W_{hx}x_t + W_{hh}h_{t-1}\right)$, consider

$$\tilde{h}_t = \sigma\left(W_{hx}x_t + W_{hh}\left(h_{t-1} \odot r_t\right)\right)$$

  - If the reset gate $\simeq 1$, then this looks like a regular RNN unit (i.e., we retain memory)
  - If the reset gate $\simeq 0$, then this looks like a regular perceptron/dense layer (i.e., we forget)
- the update gate tells us how much memory retention versus forgetting needs to happen

$$h_t = h_{t-1} \odot z_t + \tilde{h}_t \odot (1 - z_t)$$

Long-Short Term Memory

$$h_t = g\left(W_{hx}x_t + W_{hh}h_{t-1} + b_h\right) \qquad \text{(memory)}$$

$$y_t = \text{softmax}\left(W_{yh}h_t + b_y\right) \qquad \text{(used as feature for prediction)}$$

Long-Short Term Memory

$$g_t = g\left(W_{cx}x_t + W_{ch}h_{t-1} + b_c\right) \qquad \text{(input modulation gate)}$$

$$c_t = g_t \qquad \text{(place memory in a cell unit c)}$$

$$h_t = c_t$$

$$y_t = \text{softmax}\left(W_{yh}h_t + b_y\right) \qquad \text{(use } h_t \text{ for prediction)}$$

Long-Short Term Memory

$$
\begin{aligned}
g_t &= g\left(W_{cx}x_t + W_{ch}h_{t-1} + b_c\right) && \text{(input modulation gate)}\\
c_t &= c_{t-1} + g_t && \text{(the cell keeps track of long term)}\\
h_t &= c_t\\
y_t &= \text{softmax}\left(W_{yh}h_t + b_y\right)
\end{aligned}
$$

Long-Short Term Memory

$$
\begin{aligned}
f_t &= \mathrm{sigm}\left(W_{fx}x_t + W_{fh}h_{t-1} + b_f\right) && \text{(forget gate)}\\
g_t &= g\left(W_{cx}x_t + W_{ch}h_{t-1} + b_c\right) && \text{(input modulation gate)}\\
c_t &= f_t \otimes c_{t-1} + g_t && \text{(but can forget some of its memories)}\\
h_t &= c_t\\
y_t &= \mathrm{softmax}\left(W_{yh}h_t + b_y\right)
\end{aligned}
$$

Long-Short Term Memory

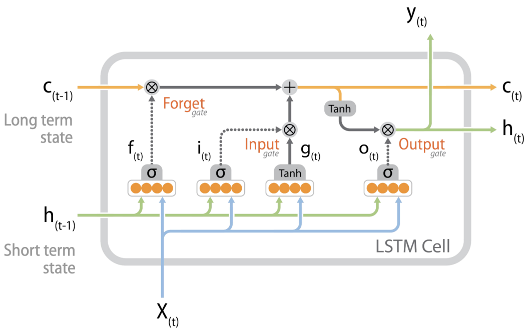
$$
\begin{aligned}
i_t &= \text{sigm}\left(W_{ix}x_t + W_{ih}h_{t-1} + b_i\right) && \text{(input gate)} \\
f_t &= \text{sigm}\left(W_{fx}x_t + W_{fh}h_{t-1} + b_f\right) && \text{(forget gate)} \\
g_t &= g\left(W_{cx}x_t + W_{ch}h_{t-1} + b_c\right) && \text{(input modulation gate)} \\
c_t &= f_t \otimes c_{t-1} + g_t && \text{(but can forget some of its memories)} \\
h_t &= c_t \\
y_t &= \text{softmax}\left(W_{yh}h_t + b_y\right)
\end{aligned}
$$

Long-Short Term Memory

$$
\begin{aligned}
o_t &= \mathrm{sigm}\left(W_{ox}x_t + W_{oh}h_{t-1} + b_o\right) && \text{(output gate)}\\
i_t &= \mathrm{sigm}\left(W_{ix}x_t + W_{ih}h_{t-1} + b_i\right) && \text{(input gate)}\\
f_t &= \mathrm{sigm}\left(W_{fx}x_t + W_{fh}h_{t-1} + b_f\right) && \text{(forget gate)}\\
g_t &= g\left(W_{cx}x_t + W_{ch}h_{t-1} + b_c\right) && \text{(input modulation gate)}\\
c_t &= f_t \otimes c_{t-1} + g_t && \text{(but can forget some of its memories)}\\
h_t &= o_t \otimes c_t && \text{(weight memory for generating feature)}\\
y_t &= \mathrm{softmax}\left(W_{yh}h_t + b_y\right)
\end{aligned}
$$

- There are many variants, but this is the general idea

Long-Short Term Memory (LSTM)



Long short-term memory (LSTM)[1]

Gated recurrent unit (GRU)[2]

$$
\begin{aligned}
f_{(t)} &= \sigma(W_{xf}^T X_{(t)} + W_{hf}^T h_{(t-1)} + b_f) \\
i_{(t)} &= \sigma(W_{xi}^T X_{(t)} + W_{hi}^T h_{(t-1)} + b_i) \\
g_{(t)} &= \tanh(W_{xg}^T X_{(t)} + W_{hg}^T h_{(t-1)} + b_g) \\
o_{(t)} &= \sigma(W_{xo}^T X_{(t)} + W_{ho}^T h_{(t-1)} + b_o) \\
c_{(t)} &= f_{(t)} \otimes c_{(t-1)} + i_{(t)} \otimes g_{(t)} \\
y_{(t)} &= h_{(t)} = o_{(t)} \otimes \tanh(c_{(t)})
\end{aligned}
$$

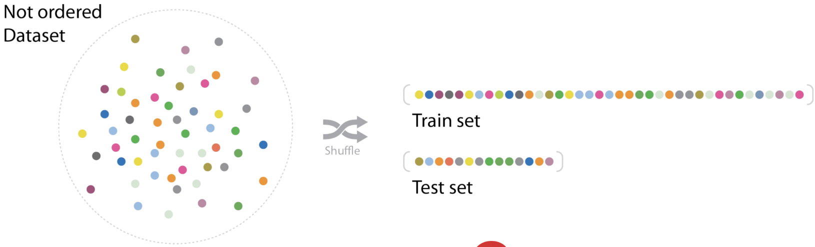
with :

$X_{(t)} \in \mathbb{R}^d$	input vector
$f_{(t)} \in \mathbb{R}^h$	forget gate's activation vector
$i_{(t)} \in \mathbb{R}^h$	input gate's activation vector
$o_{(t)} \in \mathbb{R}^h$	output gate's activation vector
$g_{(t)} \in \mathbb{R}^h$,	current entry vector
$h_{(t)}, y_{(t)} \in \mathbb{R}^h$	hidden state or output vector
$c_{(t)} \in \mathbb{R}^h$	cell state vector
$\otimes$	Hadamard product
σ	sigmoid function
W_k	weights matrix
b_k	bias vector

LSTM being a generalization of GRU

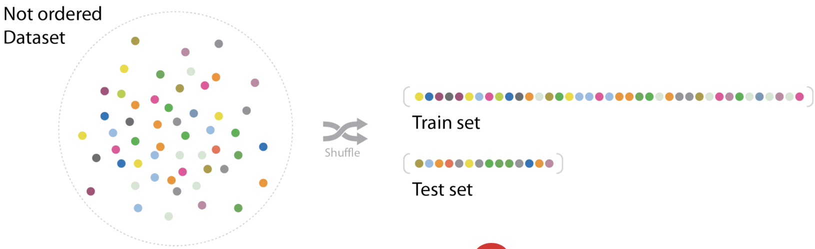
- The LSTM units give the network memory cells with read, write and reset operations. During training, the network can learn when it should remember data and when it should throw it away
- Well-suited to learn from experience to classify, process and predict time series when there are very long time lags of unknown size between important events

Preparation of sequence data



The distribution of data between trains and test series must be comparable.

Preparation of sequence data

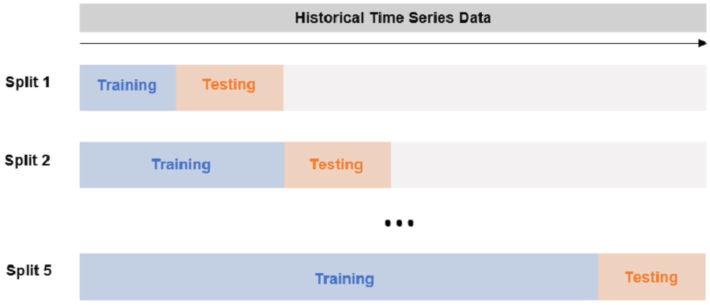


The distribution of data between trains and test series must be comparable.

- Can you predict the past with the future? Beware of splitting time series!

source: fidle-cnrs

Cross-validation for time series



Summary

Self-attention for what?

So far

- Convnets maps an image to a single output
- RNN maps a sequence to a single output or a sequence
- Self-attention maps a set of inputs $\{x_1, \ldots, x_N\}$ to a set of outputs $\{y_1, \ldots, y_N\}$
- This is an embedding

A preliminary version of self-attention

$$y_i = \sum_{j=1}^{N} w_{ij} x_j$$

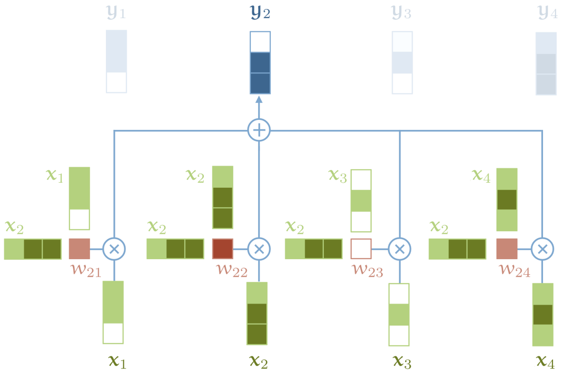
- Each output is a weighted average of all inputs where the weights W_{ij} are row-normalized such that they sum to 1
- The weights are directly derived from the inputs, e.g.

$$w'_{ij} = x_i^\top x_j \qquad \left. w_{ij} = \frac{\exp(w'_{ij})}{\sum_{j'} \exp(w'_{ij'})} \right\} \text{softmax}\left((w'_{ij})_j\right)$$

- Here, everything is deterministic, for now nothing is learned
- The operation is permutation-invariant (but this can be fixed, see later)

A preliminary version of self-attention

from `http://peterbloem.nl/blog/transformers`



- A few other ingredients are needed for a complete transformer
- But this is the only operation in the whole architecture that propagates information between vectors
  - Every other operation in the transformer is applied to each vector in the input sequence without interactions between vectors

A preliminary version of self-attention

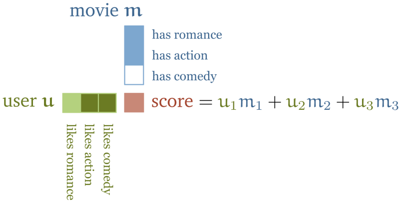
What's the point?

- Restriction of self-attention to linear models
- Example of Neural Machine Translation (NMT)
- Task: translate "`the dog sat on the couch`" from English to French
  - A lot of redundancy in natural languages
  - '`the`' '`on`' are common, not informative, not correlated
  - '`dog`' '`couch`' are similar, both nouns, can be grouped according to subject-object relationships or subject-predicate relationships
- It would be useful if the model automatically "grouped" similar words together
- Possible by the scalar products

A preliminary version of self-attention

Another example: movie recommendation

1. create manual features for movies and for users
   - how much romance there is in the movie, and how much action,
   - how much they enjoy romantic movies and how much they enjoy action-based movies

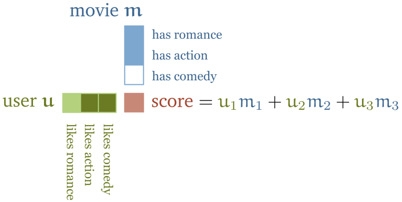


2. The dot product between the two feature vectors gives a score for how well the attributes of the movie match what the user enjoys

A preliminary version of self-attention

Another example: movie recommendation

1. create manual features for movies and for users
   - how much romance there is in the movie, and how much action,
   - how much they enjoy romantic movies and how much they enjoy action-based movies



2. The dot product between the two feature vectors gives a score for how well the attributes of the movie match what the user enjoys

Dot product $\equiv$ relations between objects

A step further

- This is the basic principle at work in the self-attention

Going back to the NMT example:

- Input: a sequence of words $x_1, \ldots x_N$

A step further

- This is the basic principle at work in the self-attention

Going back to the NMT example:

- Input: a sequence of words $x_1, \dots x_N$
- Embedding layer: apply to each word x_i an embedding v_i (the values that we will learn)
  $\rightsquigarrow$ Learning the values v_i is learning how "related" two words are
  $\rightsquigarrow$ Entirely determined by the learning task

A step further

- This is the basic principle at work in the self-attention

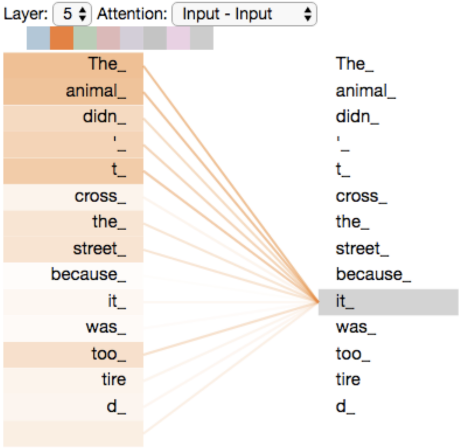
Going back to the NMT example:

- Input: a sequence of words $x_1, \dots x_N$
- Embedding layer: apply to each word x_i an embedding v_i (the values that we will learn)
  - $\rightsquigarrow$ Learning the values v_i is learning how "related" two words are
  - $\rightsquigarrow$ Entirely determined by the learning task

Ex: `"The dog sleeps on the couch"`

- '`The`': not very relevant to the interpretation of the other words in the sentence
- Desire 1: the embedding $v_{\texttt{The}}$ should have a zero or negative scalar product with the other words
- Helpful to interpret who sleeps
- Desire 2: for nouns like '`dog`' and verbs like '`sleeps`', learn an embedding $v_{\texttt{dog}}$ and $v_{\texttt{sleeps}}$ that have a high, positive dot product

Learning the embedding: attention weights



- Showing the scalar products between the learned embedding v
- As we are encoding the word "`it`", part of the attention mechanism was focusing on "`the animal`"

Towards a real self-attention layer

In the toy self-attention version, every input vector x_i is used in three different ways in the self attention operation

- (Query) x_i is compared to every other vector to establish the weights for its own output y_i
- (Key) x_i is compared to every other vector to establish the weights for the output of the j-th vector y_j
- (Value) x_i is used as part of the weighted sum to compute each output vector once the weights have been established.

These three roles are called the query, key, and value.

Towards a real self-attention layer

Make these roles distinct by adding a few dummy variables:

$$
\begin{aligned}
q_i &= x_i \qquad \text{(Query)} \\
k_i &= x_i \qquad \text{(Key)} \\
v_i &= x_i \qquad \text{(Value)}
\end{aligned}
$$

and then write out the output as:

$$w'_{ij} = q_i^\top k_j \qquad w_{ij} = \text{softmax}((w'_{ij})) \qquad y_i = \sum_{j=1}^{N} w_{ij} v_j$$

Towards a real self-attention layer

Make these roles distinct by adding a few dummy variables:

$$
\begin{aligned}
q_i &= x_i && \text{(Query)} \\
k_i &= x_i && \text{(Key)} \\
v_i &= x_i && \text{(Value)}
\end{aligned}
$$

and then write out the output as:

$$
w'_{ij} = q_i^\top k_j \qquad w_{ij} = \mathrm{softmax}((w'_{ij})) \qquad y_i = \sum_{j=1}^{N} w_{ij} v_j
$$

Then, we can learnable parameters for each of these roles, for instance:

$$
\begin{aligned}
q_i &= W_q x_i && \text{(Query)} \\
k_i &= W_k x_i && \text{(Key)} \\
v_i &= W_v x_i && \text{(Value)}
\end{aligned}
$$

where W_q, W_k, W_v are learnable projection matrices that defines the roles of each data point

An attention head

from `http://peterbloem.nl/blog/transformers`

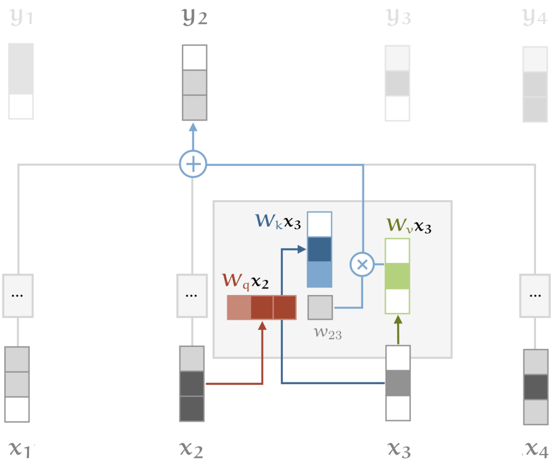


Fig.: Illustration of the self-attention with key, query and value transformations

Scaling the dot product

- The dot product in attention weights is usually scaled

$$w'_{ij} = \frac{1}{\sqrt{\text{dimension of the embedding}}} q_i^\top k_j$$

where dimension of the embedding $=$ size of q_i, k_i, v_i

- The softmax function can be sensitive to very large input values

  $\rightsquigarrow$ vanishing gradient / slow training

- The average value of the dot product grows with the embedding dimension

Multi-head self attention layer

- Concatenate different self-attention mechanisms to give it more flexibility
- Same analogy as choosing multiple filters in a convnet layer

Index each head with $r = 1, 2, \ldots$

$$q_i^r = W_q^r x_i \qquad k_i^r = W_k^r x_i \qquad v_i^r = W_v^r x_i$$

$$(w')_{ij}^r = (q_i^r)^\top k_j^r \qquad w_{ij}^r = \text{softmax}((w')_{ij}^r)) \qquad y_i^r = \sum_{j=1}^{N} w_{ij}^r v_j^r$$

$$y_i = W_y \text{concat}\left(y_i^1, y_i^2, \ldots\right)$$

$$(y_1, \ldots, y_N) = \text{Attn}\left(x_1, \ldots, x_N\right)$$

On the vocabulary

- ‘key’, ‘query’, ‘value’ come from a key-value data structure (search engine)
  If we give a query key and match it to a database of available keys, then the data structure returns the corresponding matched value
- Similar here
  - matching done by scalar products
  - softmax ensures a soft-matching
  - keys are matched to queries in some extent

On the vocabulary

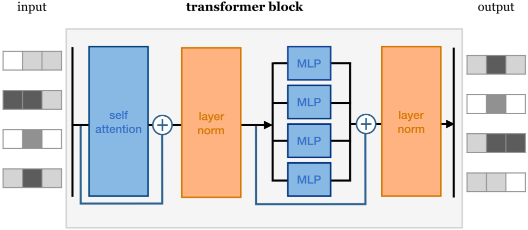
- 'key', 'query', 'value' come from a key-value data structure (search engine)
  If we give a query key and match it to a database of available keys, then the data structure returns the corresponding matched value
- Similar here
  - matching done by scalar products
  - softmax ensures a soft-matching
  - keys are matched to queries in some extent

- "Self-attention"? The self-attention mechanism allows the inputs
  1. to interact with each other ("self")
  2. to find out who they should pay more attention to ("attention")

  The outputs are aggregates of these interactions and attention scores.

Transformer?

- This is an architecture



from http://peterbloem.nl/blog/transformers

- Combining self-attention, residual connections, layer normalizations and standard MLPs
- Normalization and residual connections are standard tricks used to help deep neural networks train faster and more accurately
- The layer normalization is applied over the embedding dimension only

Positional encoding

- Unlike sequence models (such as RNNs or LSTMs), self-attention layers are permutation-equivariant
- Meaning that

$$\begin{cases} \texttt{`The dog chases the cat'} \\ \texttt{`The cat chases the dog'} \end{cases}$$

will learn the same features

Positional encoding

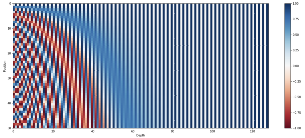
- Unlike sequence models (such as RNNs or LSTMs), self-attention layers are permutation-equivariant
- Meaning that

$$\begin{cases} \texttt{`The dog chases the cat'} \\ \texttt{`The cat chases the dog'} \end{cases}$$

  will learn the same features
- Solution: positional embedding/encoding
  - One-hot encoding
  - Sinusoidal encoding

$$\text{position } t \to (\sin(\omega_1 t), \sin(\omega_2 t), \ldots, \sin(\omega_d t))$$

with $\omega_k = \frac{1}{10000^{k/d}}$ (float continuous counterparts of binary values)



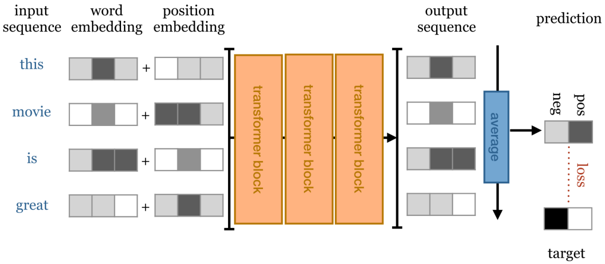
The 128-dimensional positional encoding for a sentence with a maximum length of 50. Each row represents the encoding vector.

Simple sequence classification transformer

- Goal: build a sequence classifier for sentiment analysis
- IMDb sentiment classification dataset
  - (input) movie reviews (sequences of words)
  - (output) classification labels: positive or negative

Simple sequence classification transformer

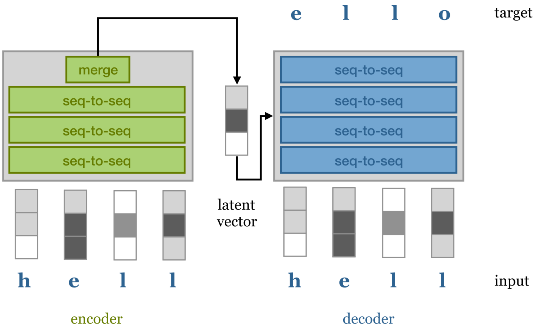
- Goal: build a sequence classifier for sentiment analysis
- IMDb sentiment classification dataset
  - (input) movie reviews (sequences of words)
  - (output) classification labels: positive or negative



from http://peterbloem.nl/blog/transformers

The original transformer

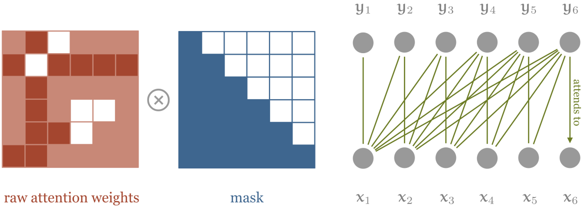
- "Attention is all you need" by Vaswani et al. (2017)



- A sequence-to-sequence structure by encoder-decoder architecture with teacher forcing
  - encoder: takes the input sequence and maps it to a latent representation
  - decoder: unpacks it to the desired target sequence (for instance, language translation)
  - teacher forcing: the decoder also has access to the input sequence

Focus on the decoder

- The decoder also has access to the input sequence
- In an autoregressive manner



from http://peterbloem.nl/blog/transformers

Masking the self-attention scores to ensure that elements can only attend to input elements that precede them in the sequence

- The decoder can use
  - word-for-word sampling to take care of the low-level structure like syntax and grammar
  - the latent vector to capture more high-level semantic structure

Modern transformers

- BERT (Bidirectional Encoder Representations from Transformers): reaches human-level performance on a variety of language based tasks: question answering, sentiment classification or classifying whether two sentences naturally follow one another
  - simple stack of transformer blocks
  - pre-trained on a large general-domain corpus (English books and wikipedia)
  - pre-training possible through masking or next-sequence classification
- GPT-2: prediction of the next word
- Transformer-XL: for long sequence of text
- Sparse transformers: uses sparse attention matrices

Wrapping up

- 4 different NN architectures
- for different purposes
- for different inputs
- Back-propagation in all of them: this is the learning phase

There are a lot of things we did not talk about

- NLP
- GAN reproducing "realistic" data
- Auto-encoder (unsupervised ML) learning a low-dimensional representation of data

Further reading

Some cheat sheets / online lectures / book

- https://stanford.edu/~shervine/teaching/cs-230/
- https://stanford.edu/~shervine/teaching/cs-229/cheatsheet-deep-learning
- https://chinmayhegde.github.io/dl-notes/
- https://cloud.univ-grenoble-alpes.fr/index.php/s/wxCztjYBbQ6zwd6
- https://www.deeplearningbook.org/

Further reading

Some cheat sheets / online lectures / book

- https://stanford.edu/~shervine/teaching/cs-230/
- https://stanford.edu/~shervine/teaching/cs-229/cheatsheet-deep-learning
- https://chinmayhegde.github.io/dl-notes/
- https://cloud.univ-grenoble-alpes.fr/index.php/s/wxCztjYBbQ6zwd6
- https://www.deeplearningbook.org/

Thank you!