Artificial Intelligence

Supervised Learning

Rémi Parrot

remi.parrot@ec-nantes.fr

11 mars 2024

Learning

"An agent is **learning** if it **improves** its **performance** after making **observations** about the world.", *S. Russell and P. Norvig*, Artificial Intelligence – A Modern Approach

"An agent is **learning** if it **improves** its **performance** after making **observations** about the world.", *S. Russell and P. Norvig*, Artificial Intelligence – A Modern Approach

Induction
specific observations $\rightarrow$ general rules

$\neq$

Deduction
general axioms $\rightarrow$ specific propositions

(guaranteed to be correct)

Learning

"An agent is **learning** if it **improves** its **performance** after making **observations** about the world.", *S. Russell and P. Norvig*, Artificial Intelligence – A Modern Approach

Induction
specific observations $\rightarrow$ general rules

$\neq$

Deduction
general axioms $\rightarrow$ specific propositions

(guaranteed to be correct)

Example
the sun rose every morning in the past $\rightarrow$ the sun will rise tomorrow

Example
all squirrels are mortal and Scrat is a squirrel $\rightarrow$ Scrat is mortal

Forms of learning

Parameters

- *component* to be improved
- *prior knowledge* $\rightarrow$ *model*
- *data* and *feedback*

Forms of learning

Parameters

- *component* to be improved
- *prior knowledge* $\rightarrow$ *model*
- *data* and *feedback*

Components

- A direct mapping from conditions on the current state to actions
- A means to infer relevant properties of the world from the percept sequence
- Information about the way the world evolves and about the results of possible actions
- Utility information indicating the desirability of world states
- ...

Forms of learning

Data
$(x_1, y_1), (x_2, y_2), \dots \in X \times Y$

- **Classification** : Y is *finite* (e.g. $\{sunny, cloudy, rainy\}$ or $\{true, false\}$)
- **Regression** : Y is *infinite* (e.g. $\mathbb{N}$)

Forms of learning

Data
$(x_1, y_1), (x_2, y_2), \dots \in X \times Y$

- **Classification** : Y is *finite* (e.g. $\{sunny, cloudy, rainy\}$ or $\{true, false\}$)
- **Regression** : Y is *infinite* (e.g. $\mathbb{N}$)

Feedback

- **Supervised learning** : the agent observes *input-output* pairs (x, y) and learn $y = f(x)$
- **Unsupervised learning** : the agent learns *pattern* from *inputs*
- **Reinforcement learning** : the agent learns from a serie of reinforcements : *rewards* and *punishments*

Content

Supervised Learning

Supervised Learning - framework

Data set
$(x_1, y_1), (x_2, y_2), \dots (x_N, y_N) \in X \times Y$

Supervised Learning - framework

Data set
$(x_1, y_1), (x_2, y_2), \ldots (x_N, y_N) \in X \times Y$

Function to learn
$y = f(x) \rightarrow$ hypothesis $h \sim f$

Supervised Learning - framework

Data set
$(x_1, y_1), (x_2, y_2), \dots (x_N, y_N) \in X \times Y$

Function to learn
$y = f(x) \rightarrow$ hypothesis $h \sim f$

Stationarity assumption

- $P(E_j) = P(E_{j+1}) = P(E_{j+2}) = \dots$: each example has the same prior probability distribution
- $P(E_j) = P(E_j | E_{j-1}, E_{j-2}, \dots)$: each example is independent from previous examples

$\hookrightarrow$ *independent and identically distributed*

Model and dataset

Model

- **hypothesis space** $\mathcal{H} =$ model class

Model and dataset

Model

- **hypothesis space** $\mathcal{H} =$ model class
- **hypothesis** $h \in \mathcal{H} =$ model

Model and dataset

Model

- **hypothesis space** $\mathcal{H} =$ model class
- **hypothesis** $h \in \mathcal{H} =$ model
- **hyperparameters** : parameters of the model class (e.g : degree for polynomial)

Model and dataset

Train and Evaluate
Learn with part of the data and *evaluate* with the rest :

Model

- **hypothesis space** $\mathcal{H} =$ model class
- **hypothesis** $h \in \mathcal{H} =$ model
- **hyperparameters** : parameters of the model class (e.g : degree for polynomial)

Model and dataset

Model

- **hypothesis space** $\mathcal{H} =$ model class
- **hypothesis** $h \in \mathcal{H} =$ model
- **hyperparameters** : parameters of the model class (e.g : degree for polynomial)

Train and Evaluate

Learn with part of the data and *evaluate* with the rest :

- **training set** : to train candidate models ($\neq$ model classes and $\neq$ hyperparameters)

Model and dataset

Model

- **hypothesis space** $\mathcal{H} =$ model class
- **hypothesis** $h \in \mathcal{H} =$ model
- **hyperparameters** : parameters of the model class (e.g : degree for polynomial)

Train and Evaluate

Learn with part of the data and *evaluate* with the rest :

- **training set** : to train candidate models ($\neq$ model classes and $\neq$ hyperparameters)
- **validation set** : to evaluate candidate models and select the best

Model and dataset

Model

- **hypothesis space** $\mathcal{H} =$ model class
- **hypothesis** $h \in \mathcal{H} =$ model
- **hyperparameters** : parameters of the model class (e.g : degree for polynomial)

Train and Evaluate
Learn with part of the data and *evaluate* with the rest :

- **training set** : to train candidate models ($\neq$ model classes and $\neq$ hyperparameters)
- **validation set** : to evaluate candidate models and select the best
- **test set** : to evaluate the selected model

Model and dataset

Model

- **hypothesis space** $\mathcal{H} =$ model class
- **hypothesis** $h \in \mathcal{H} =$ model
- **hyperparameters** : parameters of the model class (e.g : degree for polynomial)

Train and Evaluate
Learn with part of the data and *evaluate* with the rest :

- **training set** : to train candidate models ($\neq$ model classes and $\neq$ hyperparameters)
- **validation set** : to evaluate candidate models and select the best
- **test set** : to evaluate the selected model

k-fold cross-validation

- split the training set into k subsets
- iterate the three steps for all $i \in [1, k]$:
    - take subset i out
    - train with $k - 1$ joint subsets
    - validate with the subset i

Evaluation

Loss function
$y = f(x)$ and $\hat{y} = h(x)$

Evaluation

Loss function
$y = f(x)$ and $\hat{y} = h(x)$

$L(x, y, \hat{y}) = \text{Utility(using } y \text{ given } x) - \text{Utility(using } \hat{y} \text{ given } x)$

Evaluation

Loss function
$y = f(x)$ and $\hat{y} = h(x)$

$L(x, y, \hat{y}) = \text{Utility}(\text{using } y \text{ given } x) - \text{Utility}(\text{using } \hat{y} \text{ given } x) \rightarrow \text{simplification} : L(y, \hat{y})$

Evaluation

Loss function
$y = f(x)$ and $\hat{y} = h(x)$

$L(x, y, \hat{y}) =$ Utility(using y given x) – Utility(using $\hat{y}$ given x) $\rightarrow$ simplification : $L(y, \hat{y})$

example : For a spam filter, $L(spam, nospam) = 1$ and $L(nospam, spam) = 10$

Evaluation

Loss function
$y = f(x)$ and $\hat{y} = h(x)$

$L(x, y, \hat{y}) = \text{Utility(using } y \text{ given } x) - \text{Utility(using } \hat{y} \text{ given } x) \rightarrow \text{simplification} : L(y, \hat{y})$

example : For a spam filter, $L(spam, nospam) = 1$ and $L(nospam, spam) = 10$

Usual loss functions

- Absolute-value loss : $L_1(y, \hat{y}) = |y - \hat{y}|$

Evaluation

Loss function
$y = f(x)$ and $\hat{y} = h(x)$

$L(x, y, \hat{y}) = \text{Utility(using } y \text{ given } x) - \text{Utility(using } \hat{y} \text{ given } x) \rightarrow \text{simplification} : L(y, \hat{y})$

example : For a spam filter, $L(spam, nospam) = 1$ and $L(nospam, spam) = 10$

Usual loss functions

- Absolute-value loss : $L_1(y, \hat{y}) = |y - \hat{y}|$
- Squared-error loss : $L_2(y, \hat{y}) = (y - \hat{y})^2$

Evaluation

Loss function
$y = f(x)$ and $\hat{y} = h(x)$

$L(x, y, \hat{y}) = \text{Utility(using } y \text{ given } x) - \text{Utility(using } \hat{y} \text{ given } x) \rightarrow \text{simplification} : L(y, \hat{y})$

example : For a spam filter, $L(spam, nospam) = 1$ and $L(nospam, spam) = 10$

Usual loss functions

- Absolute-value loss : $L_1(y, \hat{y}) = |y - \hat{y}|$
- Squared-error loss : $L_2(y, \hat{y}) = (y - \hat{y})^2$
- 0/1 loss : $L_{0/1}(y, \hat{y}) = \begin{cases} 0 & \text{if } y = \hat{y} \\ 1 & \text{else} \end{cases}$

Evaluation

Loss function
$y = f(x)$ and $\hat{y} = h(x)$

$L(x, y, \hat{y}) = \text{Utility}(\text{using } y \text{ given } x) - \text{Utility}(\text{using } \hat{y} \text{ given } x) \rightarrow \text{simplification} : L(y, \hat{y})$

example : For a spam filter, $L(spam, nospam) = 1$ and $L(nospam, spam) = 10$

Usual loss functions

- Absolute-value loss : $L_1(y, \hat{y}) = |y - \hat{y}|$
- Squared-error loss : $L_2(y, \hat{y}) = (y - \hat{y})^2$
- 0/1 loss : $L_{0/1}(y, \hat{y}) = \begin{cases} 0 & \text{if } y = \hat{y} \\ 1 & \text{else} \end{cases}$

Generalization loss
$GenLoss_L(h) = \sum\limits_{(x,y)} L(y, h(x))P(x, y)$

Evaluation

Loss function
$y = f(x)$ and $\hat{y} = h(x)$

$L(x, y, \hat{y}) = \text{Utility(using } y \text{ given } x) - \text{Utility(using } \hat{y} \text{ given } x) \rightarrow \text{simplification} : L(y, \hat{y})$

example : For a spam filter, $L(spam, nospam) = 1$ and $L(nospam, spam) = 10$

Usual loss functions

- Absolute-value loss : $L_1(y, \hat{y}) = |y - \hat{y}|$
- Squared-error loss : $L_2(y, \hat{y}) = (y - \hat{y})^2$
- 0/1 loss : $L_{0/1}(y, \hat{y}) = \begin{cases} 0 & \text{if } y = \hat{y} \\ 1 & \text{else} \end{cases}$

Empirical loss

$$\boxed{EmpLoss_{L,E}(h) = \sum_{(x,y) \in E} L(y, h(x)) \frac{1}{N}}$$

(with $|E| = N$)

Generalization loss
$GenLoss_L(h) = \sum_{(x,y)} L(y, h(x)) P(x, y)$

Loss function
$y = f(x)$ and $\hat{y} = h(x)$

$L(x, y, \hat{y}) = \text{Utility(using } y \text{ given } x) - \text{Utility(using } \hat{y} \text{ given } x) \rightarrow \text{simplification} : L(y, \hat{y})$

example : For a spam filter, $L(spam, nospam) = 1$ and $L(nospam, spam) = 10$

Usual loss functions

- Absolute-value loss : $L_1(y, \hat{y}) = |y - \hat{y}|$
- Squared-error loss : $L_2(y, \hat{y}) = (y - \hat{y})^2$
- 0/1 loss : $L_{0/1}(y, \hat{y}) = \begin{cases} 0 & \text{if } y = \hat{y} \\ 1 & \text{else} \end{cases}$

Generalization loss
$GenLoss_L(h) = \sum_{(x,y)} L(y, h(x))P(x, y)$

Empirical loss

$$\boxed{EmpLoss_{L,E}(h) = \sum_{(x,y)\in E} L(y, h(x))\frac{1}{N}}$$

(with $|E| = N$)

Regularization
Ockham's razor dictates to prefer simplicity

Evaluation

Loss function
$y = f(x)$ and $\hat{y} = h(x)$

$L(x, y, \hat{y}) = \text{Utility(using } y \text{ given } x) - \text{Utility(using } \hat{y} \text{ given } x) \rightarrow \text{simplification} : L(y, \hat{y})$

example : For a spam filter, $L(spam, nospam) = 1$ and $L(nospam, spam) = 10$

Usual loss functions

- Absolute-value loss : $L_1(y, \hat{y}) = |y - \hat{y}|$
- Squared-error loss : $L_2(y, \hat{y}) = (y - \hat{y})^2$
- 0/1 loss : $L_{0/1}(y, \hat{y}) = \begin{cases} 0 & \text{if } y = \hat{y} \\ 1 & \text{else} \end{cases}$

Generalization loss
$GenLoss_L(h) = \sum_{(x,y)} L(y, h(x)) P(x, y)$

Empirical loss

$$\boxed{EmpLoss_{L,E}(h) = \sum_{(x,y) \in E} L(y, h(x)) \frac{1}{N}}$$

(with $|E| = N$)

Regularization
Ockham's razor dictates to prefer simplicity

$$\boxed{Cost(h) = EmpLoss(h) + \lambda Complexity(h)}$$

$\hat{h}^* = \underset{h \in \mathcal{H}}{argmin} Cost(h)$

Learning failure

Realizability/Intractability

- **Realizable** : $f \in \mathcal{H}$

Learning failure

Realizability/Intractability

- **Realizable** : $f \in \mathcal{H}$
- **Computationally tractable** : there exists algorithm to explore $\mathcal{H}$ with reasonable amount of time/resources.

Realizability/Intractability

- **Realizable** : $f \in \mathcal{H}$
- **Computationally tractable** : there exists algorithm to explore $\mathcal{H}$ with reasonable amount of time/resources.

Dataset
f may be nondeterministic or **noisy** : different values of $f(x)$ for a same x

Learning failure

Realizability/Intractability

- **Realizable** : $f \in \mathcal{H}$
- **Computationally tractable** : there exists algorithm to explore $\mathcal{H}$ with reasonable amount of time/resources.

Dataset
f may be nondeterministic or **noisy** : different values of $f(x)$ for a same x

Underfitting/Overfitting

- **Overfitting** : when a function pays too much attention to the particular data it is trained on $\rightarrow$ doesn't generalize well.

Learning failure

Realizability/Intractability

- **Realizable** : $f \in \mathcal{H}$
- **Computationally tractable** : there exists algorithm to explore $\mathcal{H}$ with reasonable amount of time/resources.

Dataset

f may be nondeterministic or **noisy** : different values of $f(x)$ for a same x

Underfitting/Overfitting

- **Overfitting** : when a function pays too much attention to the particular data it is trained on $\rightarrow$ doesn't generalize well.
- **Underfitting** : when a function fails to find a pattern in the data.

Learning failure

Realizability/Intractability

- **Realizable** : $f \in \mathcal{H}$
- **Computationally tractable** : there exists algorithm to explore $\mathcal{H}$ with reasonable amount of time/resources.

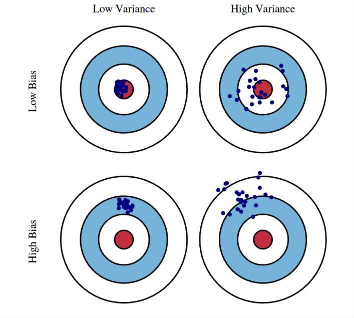
Dataset

f may be nondeterministic or **noisy** : different values of $f(x)$ for a same x

Underfitting/Overfitting

- **Overfitting** : when a function pays too much attention to the particular data it is trained on $\rightarrow$ doesn't generalize well.
- **Underfitting** : when a function fails to find a pattern in the data.

Bias-Variance tradeoff



Learning failure

Realizability/Intractability

- **Realizable** : $f \in \mathcal{H}$
- **Computationally tractable** : there exists algorithm to explore $\mathcal{H}$ with reasonable amount of time/resources.

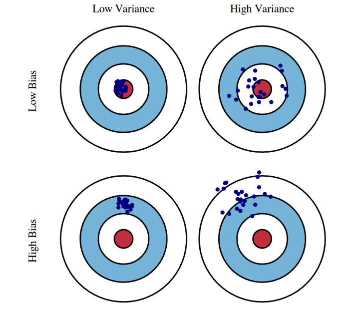
Dataset

f may be nondeterministic or **noisy** : different values of $f(x)$ for a same x

Underfitting/Overfitting

- **Overfitting** : when a function pays too much attention to the particular data it is trained on $\rightarrow$ doesn't generalize well.
- **Underfitting** : when a function fails to find a pattern in the data.

Bias-Variance tradeoff



- complex low-bias hypotheses that fit the training data well
- simple low-variance hypotheses that generalize better

Model classes

- Decision trees

Model classes

- Decision trees
- Linear regression

Model classes

- Decision trees
- Linear regression
- Linear/Logistic classification

Model classes

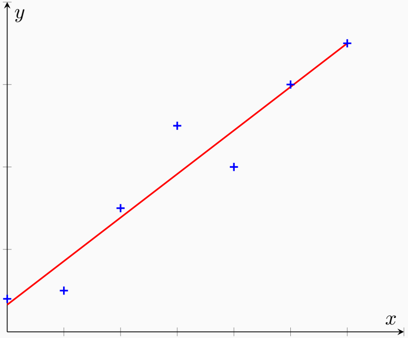
- Decision trees
- Linear regression
- Linear/Logistic classification
- Support Vector Machines

Model classes

- Decision trees
- Linear regression
- Linear/Logistic classification
- Support Vector Machines
- Neural Networks

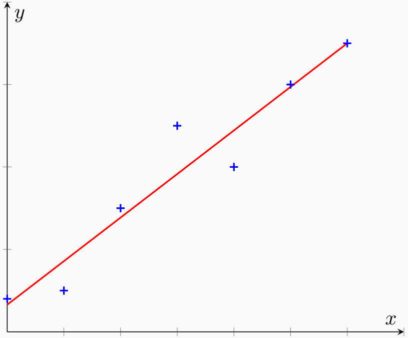
Linear Regression and Classification

Univariate linear regression



Sample set
$\{(x_1, y_1), (x_2, y_2), \ldots, (x_N, y_N)\} \subseteq \mathbb{R} \times \mathbb{R}$

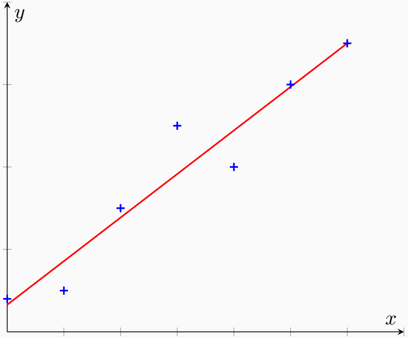
Univariate linear regression



Sample set
$\{(x_1, y_1), (x_2, y_2), \ldots, (x_N, y_N)\} \subseteq \mathbb{R} \times \mathbb{R}$

Hypothesis
$h_{\vec{w}}(x) = w_0 + w_1 x$ with $\vec{w} = (w_0, w_1)$

Univariate linear regression



Sample set

$\{(x_1, y_1), (x_2, y_2), \ldots, (x_N, y_N)\} \subseteq \mathbb{R} \times \mathbb{R}$

Hypothesis

$h_{\vec{w}}(x) = w_0 + w_1 x$ with $\vec{w} = (w_0, w_1)$

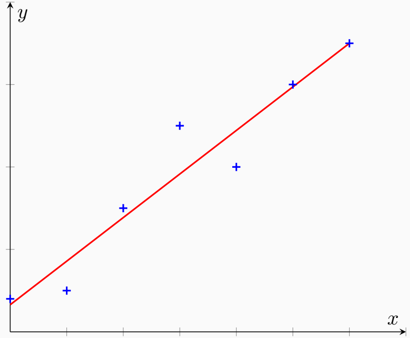
Minimize loss

Normally distributed noise $\rightarrow L_2$ (Gauss)

$$Loss(h_{\vec{w}}) = \sum_{j=1}^{N} L_2(y_j, h_{\vec{w}}(x_j)) = \sum_{j=1}^{N} (y_j - (w_0 + w_1 x_j))^2$$

Minimize $L(\vec{w}) = Loss(h_{\vec{w}})$

Univariate linear regression



Sample set
$\{(x_1, y_1), (x_2, y_2), \ldots, (x_N, y_N)\} \subseteq \mathbb{R} \times \mathbb{R}$

Hypothesis
$h_{\vec{w}}(x) = w_0 + w_1 x$ with $\vec{w} = (w_0, w_1)$

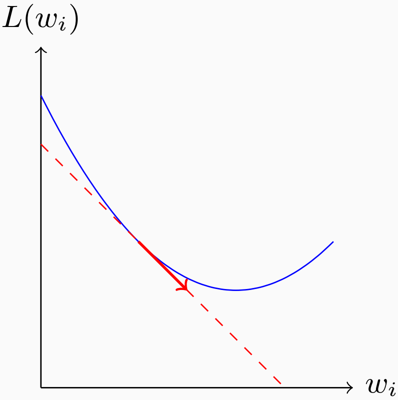
Minimize loss
Normally distributed noise $\rightarrow L_2$ (Gauss)

$$Loss(h_{\vec{w}}) = \sum_{j=1}^{N} L_2(y_j, h_{\vec{w}}(x_j)) = \sum_{j=1}^{N} (y_j - (w_0 + w_1 x_j))^2$$

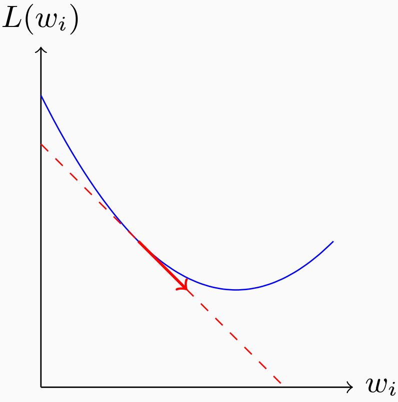
Minimize $L(\vec{w}) = Loss(h_{\vec{w}})$

Analytic solution
Show that the minimum of $L(\vec{w})$ is obtained for :
$w_1 = \frac{(\sum x_j)(\sum y_j) - N(\sum x_j y_j)}{(\sum x_j)^2 - N(\sum x_j^2)}$ and $w_0 = \frac{(\sum y_j) - w_1(\sum x_j)}{N}$

Gradient descent



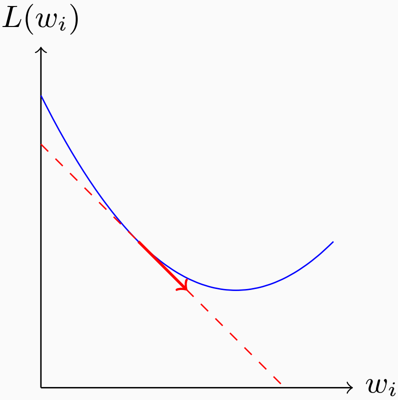
Gradient descent



Algorithm

$$\boxed{w_i \leftarrow w_i - \alpha \frac{\partial L(\vec{w})}{\partial w_i}}$$ with α the *learning rate*

Gradient descent



Algorithm

$$\boxed{w_i \leftarrow w_i - \alpha \frac{\partial L(\vec{w})}{\partial w_i}}$$ with α the *learning rate*

Univariate gradient descent

$$w_0 \leftarrow w_0 + \alpha \sum_{j=1}^{N} (y_j - h_{\vec{w}}(x_j))$$

$$w_1 \leftarrow w_1 + \alpha \sum_{j=1}^{N} (y_j - h_{\vec{w}}(x_j))x_j$$

Multivariable

Sample set
$\{(\vec{x_1}, y_1), (\vec{x_2}, y_2), \ldots, (\vec{x_N}, y_N)\} \subseteq \mathbb{R}^d \times \mathbb{R}$

Multivariable

Sample set
$\{(\vec{x_1}, y_1), (\vec{x_2}, y_2), \ldots, (\vec{x_N}, y_N)\} \subseteq \mathbb{R}^d \times \mathbb{R}$

Hypothesis

$h_{\vec{w}}(\vec{x_j}) = \vec{w}\vec{x_j} = \sum_{i=0}^{d} w_i x_{ji}$ with :

- $\vec{w} = (w_0, w_1, \ldots, w_d) \in \mathbb{R}^{d+1}$
- $\vec{x_j} = (x_{j1}, x_{j2}, \ldots, x_{jd}) \in \mathbb{R}^d$
- $x_{j0} = 1$

Multivariable

Sample set

$\{(\vec{x_1}, y_1), (\vec{x_2}, y_2), \ldots, (\vec{x_N}, y_N)\} \subseteq \mathbb{R}^d \times \mathbb{R}$

Hypothesis

$h_{\vec{w}}(\vec{x_j}) = \vec{w}\vec{x_j} = \sum_{i=0}^{d} w_i x_{ji}$ with :

- $\vec{w} = (w_0, w_1, \ldots, w_d) \in \mathbb{R}^{d+1}$
- $\vec{x_j} = (x_{j1}, x_{j2}, \ldots, x_{jd}) \in \mathbb{R}^d$
- $x_{j0} = 1$

Gradient descent

$$\boxed{w_i \leftarrow w_i - \alpha \sum_{j=1}^{N} (y_j - h_{\vec{w}}(\vec{x_j}))x_{ji}}$$

Multivariable

Sample set

$\{(\vec{x_1}, y_1), (\vec{x_2}, y_2), \ldots, (\vec{x_N}, y_N)\} \subseteq \mathbb{R}^d \times \mathbb{R}$

Hypothesis

$h_{\vec{w}}(\vec{x_j}) = \vec{w}\vec{x_j} = \sum_{i=0}^{d} w_i x_{ji}$ with :

- $\vec{w} = (w_0, w_1, \ldots, w_d) \in \mathbb{R}^{d+1}$
- $\vec{x_j} = (x_{j1}, x_{j2}, \ldots, x_{jd}) \in \mathbb{R}^d$
- $x_{j0} = 1$

Gradient descent

$$\boxed{w_i \leftarrow w_i - \alpha \sum_{j=1}^{N} (y_j - h_{\vec{w}}(\vec{x_j}))x_{ji}}$$

Analytic solution

$\boldsymbol{X}$: matrix of inputs (each row is an $\vec{x_j}$),

$\boldsymbol{y}$: vector of outputs (each row is a y_j)

$$L(\boldsymbol{w}) = \|\boldsymbol{X}.\boldsymbol{w} - \boldsymbol{y}\|^2$$

$$\nabla_{\boldsymbol{w}} L(\boldsymbol{w}) = 2\boldsymbol{X}^\top.(\boldsymbol{X}.\boldsymbol{w} - \boldsymbol{y}) = \mathbf{0}$$

$\boxed{\boldsymbol{w}^* = (\boldsymbol{X}^\top.\boldsymbol{X})^{-1}.\boldsymbol{X}^\top.\boldsymbol{y}}$: *normal equation*

Epoch poc poc

Batch gradient descent

$w_i \leftarrow w_i - \alpha \sum_{j=1}^{N} (y_j - h_{\vec{w}}(\vec{x_j}))x_{ji}$ (also called *deterministic gradient descent*)

Epoch poc poc

Batch gradient descent

$w_i \leftarrow w_i - \alpha \sum_{j=1}^{N} (y_j - h_{\vec{w}}(\vec{x_j}))x_{ji}$ (also called *deterministic gradient descent*)

Stochastic gradient descent (SGD)

1. select and remove a *minibatch* of m out of N training examples (randomly)
2. compute a step $w_i \leftarrow w_i - \alpha \sum_{j=1}^{m} (y_j - h_{\vec{w}}(\vec{x_j}))x_{ji}$
3. iterate until no more training examples

Epoch poc poc

Batch gradient descent

$w_i \leftarrow w_i - \alpha \sum_{j=1}^{N} (y_j - h_{\vec{w}}(\vec{x_j}))x_{ji}$ (also called *deterministic gradient descent*)

Stochastic gradient descent (SGD)

1. select and remove a *minibatch* of m out of N training examples (randomly)
2. compute a step $w_i \leftarrow w_i - \alpha \sum_{j=1}^{m} (y_j - h_{\vec{w}}(\vec{x_j}))x_{ji}$
3. iterate until no more training examples

Epoch
A step that covers all N training examples

Epoch poc poc

Batch gradient descent

$w_i \leftarrow w_i - \alpha \sum_{j=1}^{N} (y_j - h_{\vec{w}}(\vec{x_j}))x_{ji}$ (also called *deterministic gradient descent*)

Stochastic gradient descent (SGD)

1. select and remove a *minibatch* of m out of N training examples (randomly)
2. compute a step $w_i \leftarrow w_i - \alpha \sum_{j=1}^{m} (y_j - h_{\vec{w}}(\vec{x_j}))x_{ji}$
3. iterate until no more training examples

Epoch
A step that covers all N training examples

Complete algorithm
Iterate E epochs until *convergence*.

Regularization

Overfitting
In high-dimensional spaces, some irrelevant dimension might appear to be useful

Overfitting
In high-dimensional spaces, some irrelevant dimension might appear to be useful

Regularization
$Cost(h) = EmpLoss(h) + \lambda Complexity(h)$

Regularization

Overfitting
In high-dimensional spaces, some irrelevant dimension might appear to be useful

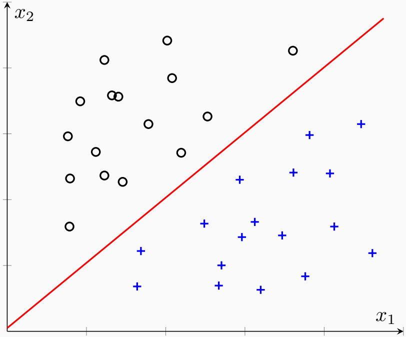
Regularization
$Cost(h) = EmpLoss(h) + \lambda Complexity(h)$

For linear functions

$$Complexity(h_{\vec{w}}) = L_q(\vec{w}) = \sum_{i=0}^{d} |w_i|^q$$

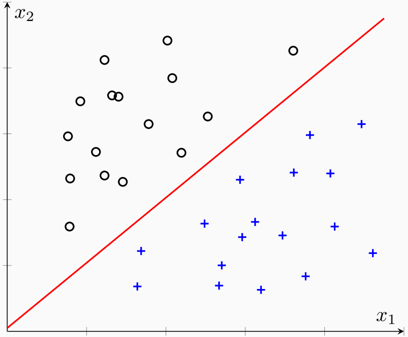
Usually, we use $q = 1$: L_1 regularization $\rightarrow$ produces *sparse model* (remove attributes)

Linear classification



Sample set
$\{(\vec{x_1}, y_1), (\vec{x_2}, y_2), \ldots, (\vec{x_N}, y_N)\} \subseteq \mathbb{R}^d \times \{0, 1\}$

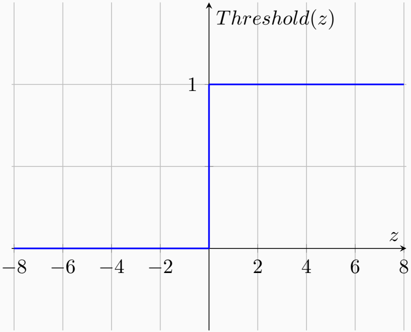
Linear classification



Sample set
$\{(\vec{x_1}, y_1), (\vec{x_2}, y_2), \ldots, (\vec{x_N}, y_N)\} \subseteq \mathbb{R}^d \times \{0, 1\}$

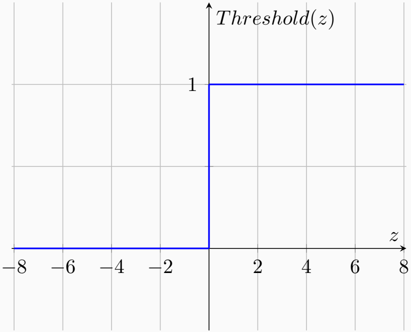
Hypothesis
The *decision boundary* is a linear separator.

Hard threshold linear classifier



$$Threshold(z) = \begin{cases} 0 \text{ if } z < 0 \\ 1 \text{ else} \end{cases}$$

Hard threshold linear classifier

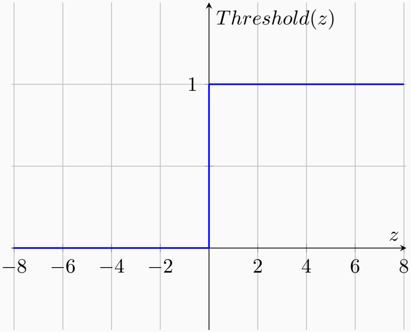


$$Threshold(z) = \begin{cases} 0 \text{ if } z < 0 \\ 1 \text{ else} \end{cases}$$

Hypothesis

$h_{\vec{w}}(\vec{x_j}) = Threshold(\vec{w}.\vec{x_j})$ with $\vec{w} \in \mathbb{R}^{d+1}$

Hard threshold linear classifier



$$Threshold(z) = \begin{cases} 0 \text{ if } z < 0 \\ 1 \text{ else} \end{cases}$$

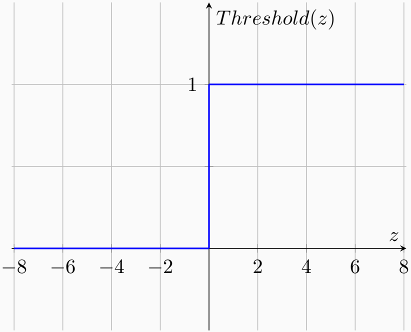
Hypothesis

$h_{\vec{w}}(\vec{x_j}) = Threshold(\vec{w}.\vec{x_j})$ with $\vec{w} \in \mathbb{R}^{d+1}$

Perceptron learning rule

$$\boxed{w_i \leftarrow w_i + \alpha(y_j - h_{\vec{w}}(\vec{x_j}))x_{ji}}$$

Hard threshold linear classifier



$$Threshold(z) = \begin{cases} 0 \text{ if } z < 0 \\ 1 \text{ else} \end{cases}$$

Hypothesis

$h_{\vec{w}}(\vec{x_j}) = Threshold(\vec{w}.\vec{x_j})$ with $\vec{w} \in \mathbb{R}^{d+1}$

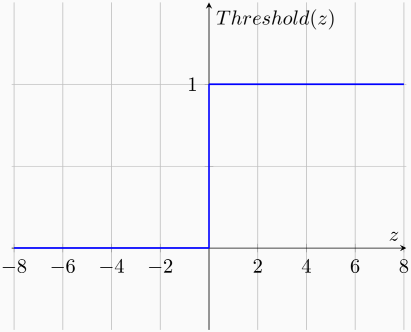
Perceptron learning rule

$$\boxed{w_i \leftarrow w_i + \alpha(y_j - h_{\vec{w}}(\vec{x_j}))x_{ji}}$$

Issue

May not converge if data is not clearly separable (without noise)

Hard threshold linear classifier



$$Threshold(z) = \begin{cases} 0 \text{ if } z < 0 \\ 1 \text{ else} \end{cases}$$

Hypothesis
$h_{\vec{w}}(\vec{x_j}) = Threshold(\vec{w}.\vec{x_j})$ with $\vec{w} \in \mathbb{R}^{d+1}$

Perceptron learning rule

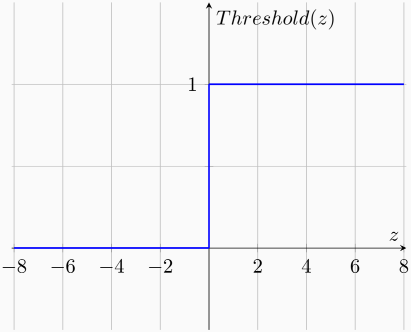
$$\boxed{w_i \leftarrow w_i + \alpha(y_j - h_{\vec{w}}(\vec{x_j}))x_{ji}}$$

Issue
May not converge if data is not clearly separable (without noise)

Dynamic learning rate
$\alpha(t) = \frac{c}{c+t}$ (decrease with time elapsing)

with c a fairly large constant

Hard threshold linear classifier



$$Threshold(z) = \begin{cases} 0 \text{ if } z < 0 \\ 1 \text{ else} \end{cases}$$

Hypothesis

$h_{\vec{w}}(\vec{x_j}) = Threshold(\vec{w}.\vec{x_j})$ with $\vec{w} \in \mathbb{R}^{d+1}$

Perceptron learning rule

$$\boxed{w_i \leftarrow w_i + \alpha(y_j - h_{\vec{w}}(\vec{x_j}))x_{ji}}$$

Issue

May not converge if data is not clearly separable (without noise)

Dynamic learning rate

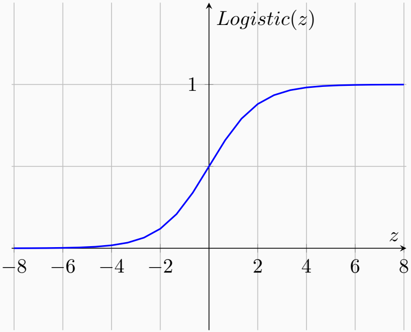
$\alpha(t) = \frac{c}{c+t}$ (decrease with time elapsing)

with c a fairly large constant

Technically, we require that :

$\sum_{t=1}^{\infty} \alpha(t) = \infty$ and $\sum_{t=1}^{\infty} \alpha(t)^2 < \infty$

Logistic linear classifier

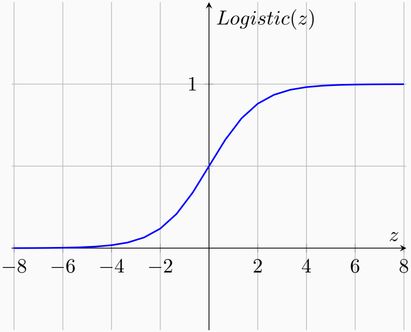


$$Logistic(z) = \frac{1}{1 + e^{-(z-\mu)/s}}$$

μ : location parameter (here $\mu = 0$)

s : scale parameter (here $s = 1$)

Logistic linear classifier



$$Logistic(z) = \frac{1}{1 + e^{-(z-\mu)/s}}$$

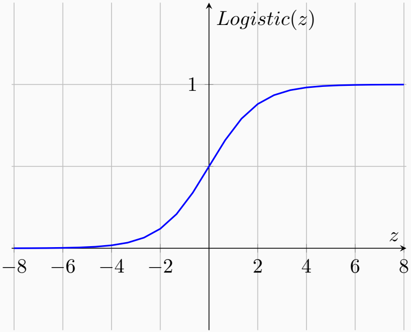
μ : location parameter (here $\mu = 0$)

s : scale parameter (here $s = 1$)

Hypothesis

$h_{\vec{w}}(\vec{x_j}) = Logistic(\vec{w}.\vec{x_j})$ with $\vec{w} \in \mathbb{R}^{d+1}$

Logistic linear classifier



$$Logistic(z) = \frac{1}{1 + e^{-(z-\mu)/s}}$$

μ : location parameter (here $\mu = 0$)

s : scale parameter (here $s = 1$)

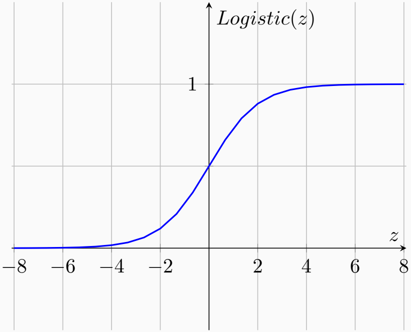
Hypothesis

$h_{\vec{w}}(\vec{x_j}) = Logistic(\vec{w}.\vec{x_j})$ with $\vec{w} \in \mathbb{R}^{d+1}$

Why though ?

Logistic function is differentiable in 0 !

Logistic linear classifier



$$Logistic(z) = \frac{1}{1 + e^{-(z-\mu)/s}}$$

μ : location parameter (here $\mu = 0$)

s : scale parameter (here $s = 1$)

Hypothesis

$h_{\vec{w}}(\vec{x_j}) = Logistic(\vec{w}.\vec{x_j})$ with $\vec{w} \in \mathbb{R}^{d+1}$

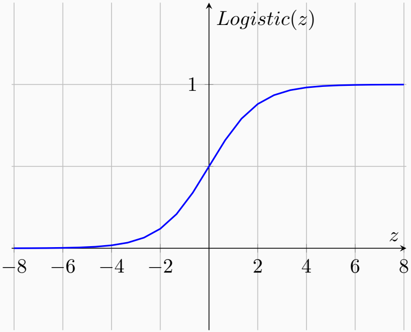
Why though ?

Logistic function is differentiable in 0 !

Gradient descent

Find the logistic learning rule from the general formula of gradient descent : $w_i \leftarrow w_i + \alpha \frac{\partial L(\vec{w})}{\partial w_i}$ (for a single example $(\vec{x}, y)$).

Logistic linear classifier



$$Logistic(z) = \frac{1}{1 + e^{-(z-\mu)/s}}$$

μ : location parameter (here $\mu = 0$)

s : scale parameter (here $s = 1$)

Hypothesis

$h_{\vec{w}}(\vec{x_j}) = Logistic(\vec{w}.\vec{x_j})$ with $\vec{w} \in \mathbb{R}^{d+1}$

Why though ?

Logistic function is differentiable in 0 !

Gradient descent

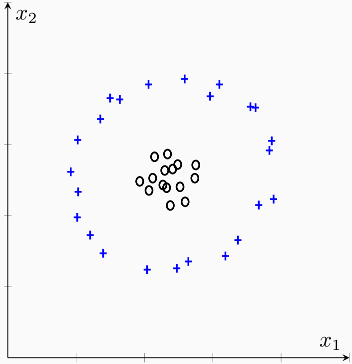
Find the logistic learning rule from the general formula of gradient descent : $w_i \leftarrow w_i + \alpha \frac{\partial L(\vec{w})}{\partial w_i}$ (for a single example $(\vec{x}, y)$).

Logistic learning rule

$$\boxed{w_i \leftarrow w_i + \alpha(y_j - h_{\vec{w}}(\vec{x_j}))h_{\vec{w}}(\vec{x_j})(1 - h_{\vec{w}}(\vec{x_j}))x_{ji}}$$

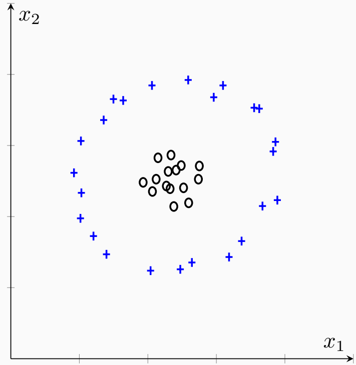
The kernel trick

What if ..
the dataset is not *linearly separable*?



The kernel trick

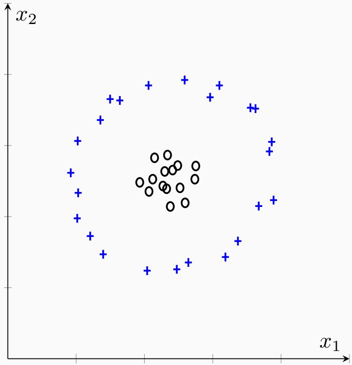
What if ..
the dataset is not *linearly separable*?



Idea
Map into another space (generally higher dimensional) where it is linearly separable : $\vec{x} \mapsto \phi(\vec{x})$

The kernel trick

What if ..
the dataset is not *linearly separable*?

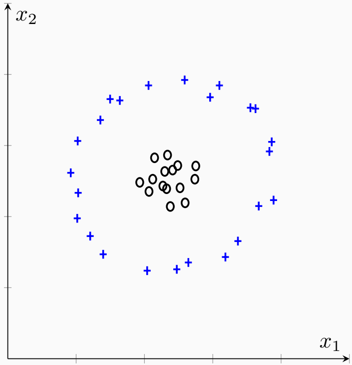


Idea
Map into another space (generally higher dimensional) where it is linearly separable : $\vec{x} \mapsto \phi(\vec{x})$

Change the space
Find a (possibly higher dimensional) space in which this dataset is linearly separable.

The kernel trick

What if ..
the dataset is not *linearly separable*?



Idea
Map into another space (generally higher dimensional) where it is linearly separable : $\vec{x} \mapsto \phi(\vec{x})$

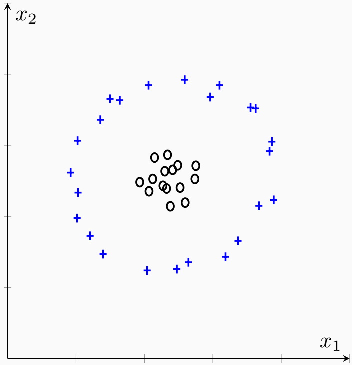
Change the space
Find a (possibly higher dimensional) space in which this dataset is linearly separable.

Kernel function

$$\boxed{K(\vec{x_k}, \vec{x_j}) = \phi(\vec{x_k}).\phi(\vec{x_j})}$$

The kernel trick

What if ..
the dataset is not *linearly separable*?



Idea
Map into another space (generally higher dimensional) where it is linearly separable : $\vec{x} \mapsto \phi(\vec{x})$

Change the space
Find a (possibly higher dimensional) space in which this dataset is linearly separable.

Kernel function

$$\boxed{K(\vec{x_k}, \vec{x_j}) = \phi(\vec{x_k}).\phi(\vec{x_j})}$$

Reformulation

We can show that $\vec{w} = \sum_{k=1}^{N} \delta_k \vec{x_k}$ then

$$h_{\vec{w}}(\vec{x_j}) = \sum_{k=1}^{N} \delta_k \vec{x_k}.\vec{x_j} \mapsto \sum_{k=1}^{N} \delta_k \phi(\vec{x_k}).\phi(\vec{x_j}) = \sum_{k=1}^{N} \delta_k K(\vec{x_k}, \vec{x_j})$$

Linear regression in the new space

$$h_{\vec{w}}(\phi(\vec{x_j})) = \sum_{k=1}^{N} \delta_k K(\vec{x_k}, \vec{x_j})$$

The kernel trick

Linear regression in the new space

$$h_{\vec{w}}(\phi(\vec{x_j})) = \sum_{k=1}^{N} \delta_k K(\vec{x_k}, \vec{x_j})$$

Kernel matrix

$K \in \mathbb{R}^N \times \mathbb{R}^N$ s.t $K_{ij} = K(\vec{x_i}, \vec{x_j})$

The kernel trick

Linear regression in the new space

$$h_{\vec{w}}(\phi(\vec{x_j})) = \sum_{k=1}^{N} \delta_k K(\vec{x_k}, \vec{x_j})$$

Kernel matrix

$K \in \mathbb{R}^N \times \mathbb{R}^N$ s.t $K_{ij} = K(\vec{x_i}, \vec{x_j})$

Algorithm

We compute $\vec{\delta}$ instead of $\vec{w}$:

$$\delta_i \leftarrow \delta_i + \alpha\gamma_i$$

The kernel trick

Linear regression in the new space

$$h_{\vec{w}}(\phi(\vec{x_j})) = \sum_{k=1}^{N} \delta_k K(\vec{x_k}, \vec{x_j})$$

Kernel matrix

$K \in \mathbb{R}^N \times \mathbb{R}^N$ s.t $K_{ij} = K(\vec{x_i}, \vec{x_j})$

Algorithm

We compute $\vec{\delta}$ instead of $\vec{w}$:

$$\delta_i \leftarrow \delta_i + \alpha\gamma_i$$

Popular kernel functions

- Linear : $K(\vec{x}, \vec{z}) = \vec{x}.\vec{z}$
- Polynomial : $K(\vec{x}, \vec{z}) = (1 + \vec{x}.\vec{z})^d$
- Radial Basis Function (RBF) : $K(\vec{x}, \vec{z}) = e^{\frac{-\|\vec{x}-\vec{z}\|^2}{\sigma^2}}$
- Laplacian Kernel : $K(\vec{x}, \vec{z}) = e^{\frac{-\|\vec{x}-\vec{z}\|}{\sigma}}$
- Sigmoïd Kernel : $K(\vec{x}, \vec{z}) = tanh(a\vec{x}.\vec{z} + b)$

demo

Linear Regression and Classification - Summary

- **Linear regression** is in pratice computed with **gradient descent**

Linear Regression and Classification - Summary

- **Linear regression** is in pratice computed with **gradient descent**
- A linear classifier with a hard threshold is called a **perceptron** and can be learnt with gradient descent if data is **linearly separable**

Linear Regression and Classification - Summary

- **Linear regression** is in pratice computed with **gradient descent**
- A linear classifier with a hard threshold is called a **perceptron** and can be learnt with gradient descent if data is **linearly separable**
- A decreasing **learning rate** improve convergence

Linear Regression and Classification - Summary

- **Linear regression** is in pratice computed with **gradient descent**
- A linear classifier with a hard threshold is called a **perceptron** and can be learnt with gradient descent if data is **linearly separable**
- A decreasing **learning rate** improve convergence
- **Logistic regression** replace the hard threshold by the logistic function

Linear Regression and Classification - Summary

- **Linear regression** is in pratice computed with **gradient descent**
- A linear classifier with a hard threshold is called a **perceptron** and can be learnt with gradient descent if data is **linearly separable**
- A decreasing **learning rate** improve convergence
- **Logistic regression** replace the hard threshold by the logistic function
- The **kernel trick** transforms input data to a higher-dimensional space where a linear separator may exists

Linear Regression and Classification - Summary

- **Linear regression** is in pratice computed with **gradient descent**
- A linear classifier with a hard threshold is called a **perceptron** and can be learnt with gradient descent if data is **linearly separable**
- A decreasing **learning rate** improve convergence
- **Logistic regression** replace the hard threshold by the logistic function
- The **kernel trick** transforms input data to a higher-dimensional space where a linear separator may exists

To go further . . .

- Other **non-parametric models** : **nearest neighbors** and **locally weighted regression**

Linear Regression and Classification - Summary

- **Linear regression** is in pratice computed with **gradient descent**
- A linear classifier with a hard threshold is called a **perceptron** and can be learnt with gradient descent if data is **linearly separable**
- A decreasing **learning rate** improve convergence
- **Logistic regression** replace the hard threshold by the logistic function
- The **kernel trick** transforms input data to a higher-dimensional space where a linear separator may exists

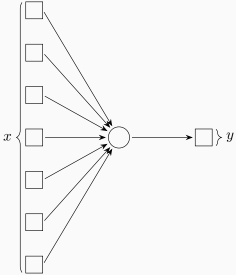
To go further ...

- Other **non-parametric models** : **nearest neighbors** and **locally weighted regression**
- **Support Vector Machines**

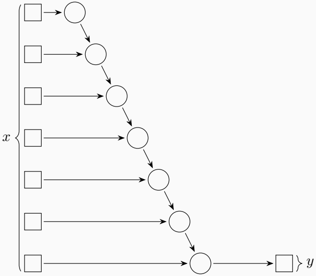
Deep Learning

Why is deep learning successful ?

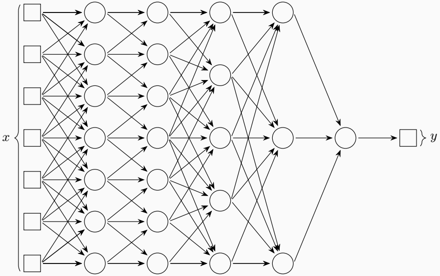
Linear regression



Decision list

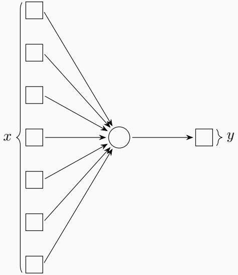


Deep learning network



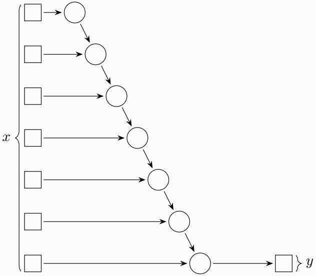
Why is deep learning successful ?

Linear regression



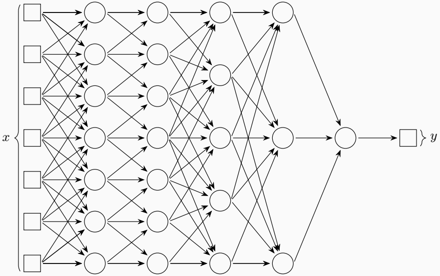
Shallow
Short computation path

Decision list



No interaction
No complex interaction between inputs

Deep learning network



Deep
Long computation path and complex interactions between many inputs

Deep Learning

Type of networks

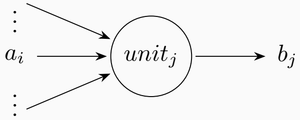
- **feedforward network** : directed acyclic graph
- **recurrent network** : loops computing intermediate or final output

Deep Learning

Type of networks

- **feedforward network** : directed acyclic graph
- **recurrent network** : loops computing intermediate or final output

Unit (a.k.a. Artificial Neuron)



$$\boxed{b_j = g_j(\sum_i w_{i,j} a_i)}$$

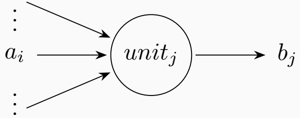
- g_j : **activation function** of unit j
- $\vec{w}_j$: weights of unit j

Deep Learning

Type of networks

- **feedforward network** : directed acyclic graph
- **recurrent network** : loops computing intermediate or final output

Unit (a.k.a. Artificial Neuron)



$$\boxed{b_j = g_j(\sum_i w_{i,j} a_i)}$$

- g_j : **activation function** of unit j
- $\vec{w}_j$: weights of unit j

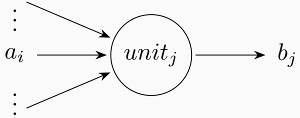
Activation functions

Deep Learning

Type of networks

- **feedforward network** : directed acyclic graph
- **recurrent network** : loops computing intermediate or final output

Unit (a.k.a. Artificial Neuron)



$$\boxed{b_j = g_j(\sum_i w_{i,j} a_i)}$$

- g_j : **activation function** of unit j
- $\vec{w}_j$: weights of unit j

Activation functions

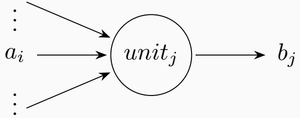
- **Logistic** or **Sigmoid** : $\sigma(x) = \frac{1}{1+e^{-x}}$

Deep Learning

Type of networks

- **feedforward network** : directed acyclic graph
- **recurrent network** : loops computing intermediate or final output

Unit (a.k.a. Artificial Neuron)



$$\boxed{b_j = g_j(\sum_i w_{i,j} a_i)}$$

- g_j : **activation function** of unit j
- $\vec{w}_j$: weights of unit j

Activation functions

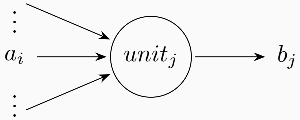
- **Logistic** or **Sigmoid** : $\sigma(x) = \frac{1}{1+e^{-x}}$
- **ReLU** (Rectified Linear Unit) : $ReLU(x) = \max(0, x)$

Deep Learning

Type of networks

- **feedforward network** : directed acyclic graph
- **recurrent network** : loops computing intermediate or final output

Unit (a.k.a. Artificial Neuron)



$$\boxed{b_j = g_j(\sum_i w_{i,j} a_i)}$$

- g_j : **activation function** of unit j
- $\vec{w}_j$: weights of unit j

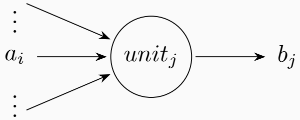
Activation functions

- **Logistic** or **Sigmoid** : $\sigma(x) = \frac{1}{1+e^{-x}}$
- **ReLU** (Rectified Linear Unit) : $ReLU(x) = \max(0, x)$
- **Softplus** (smooth ReLU) : $softplus(x) = log(1 + e^x)$

Type of networks

- **feedforward network** : directed acyclic graph
- **recurrent network** : loops computing intermediate or final output

Unit (a.k.a. Artificial Neuron)



$\boxed{b_j = g_j(\sum_i w_{i,j} a_i)}$

- g_j : **activation function** of unit j
- $\vec{w}_j$: weights of unit j

Activation functions

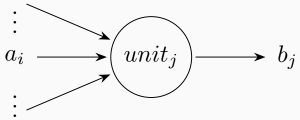
- **Logistic** or **Sigmoid** : $\sigma(x) = \frac{1}{1+e^{-x}}$
- **ReLU** (Rectified Linear Unit) : $ReLU(x) = \max(0, x)$
- **Softplus** (smooth ReLU) : $softplus(x) = log(1 + e^x)$
- **tanh** : $tanh(x) = \frac{e^{2x}-1}{e^{2x}+1}$ $(= 2\sigma(2x) - 1)$

Deep Learning

Type of networks

- **feedforward network** : directed acyclic graph
- **recurrent network** : loops computing intermediate or final output

Unit (a.k.a. Artificial Neuron)



$$\boxed{b_j = g_j(\sum_i w_{i,j} a_i)}$$

- g_j : **activation function** of unit j
- $\vec{w_j}$: weights of unit j

Activation functions

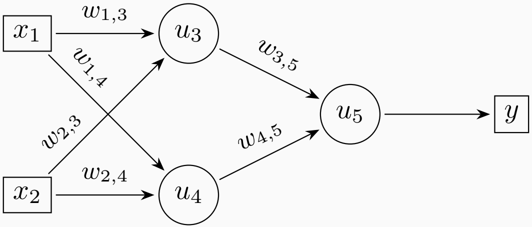
- **Logistic** or **Sigmoid** : $\sigma(x) = \frac{1}{1+e^{-x}}$
- **ReLU** (Rectified Linear Unit) : $ReLU(x) = \max(0, x)$
- **Softplus** (smooth ReLU) : $softplus(x) = log(1 + e^x)$
- **tanh** : $tanh(x) = \frac{e^{2x}-1}{e^{2x}+1}$ $(= 2\sigma(2x) - 1)$

Universal approximation theorem

A network with just two layers (one non-linear and one linear) can approximate *any continuous function* to an *arbitrary degree of accuracy*.

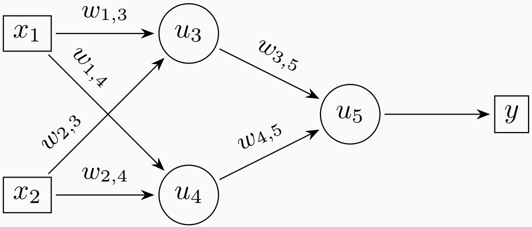
An example

Example



An example

Example



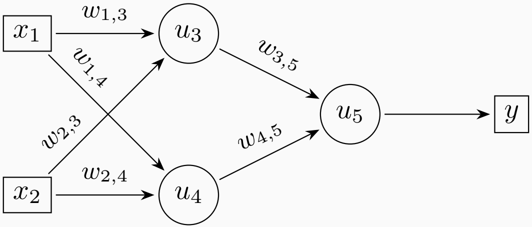
Forward computation

$$\hat{y} = g_5(w_{0,5} + w_{3,5}a_3 + w_{4,5}a_4)$$

$$\begin{aligned}\hat{y} = g_5(w_{0,5} &+ w_{3,5}g_3(w_{0,3} + w_{1,3}x_1 + w_{2,3}x_2) \\ &+ w_{4,5}g_4(w_{0,4} + w_{1,4}x_1 + w_{2,4}x_2))\end{aligned}$$

An example

Example



Forward computation

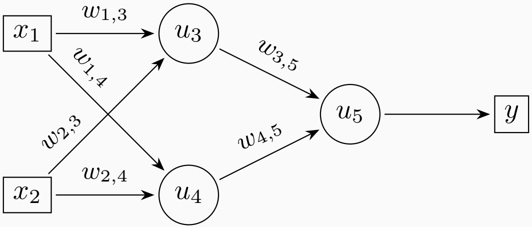
$$\hat{y} = g_5(w_{0,5} + w_{3,5}a_3 + w_{4,5}a_4)$$

$$\begin{aligned}\hat{y} = g_5(w_{0,5} &+ w_{3,5}g_3(w_{0,3} + w_{1,3}x_1 + w_{2,3}x_2) \\ &+ w_{4,5}g_4(w_{0,4} + w_{1,4}x_1 + w_{2,4}x_2))\end{aligned}$$

$$\boxed{h_{\boldsymbol{W}}(\boldsymbol{x}) = g^{(2)}(\boldsymbol{W}^{(2)}g^{(1)}(\boldsymbol{W}^{(1)}\boldsymbol{x}))}$$

An example

Example



Gradient descent

$Loss(h_{\boldsymbol{W}}) = L_2(y, h_{\boldsymbol{W}}(\boldsymbol{x})) = (y - \hat{y})^2$

Forward computation

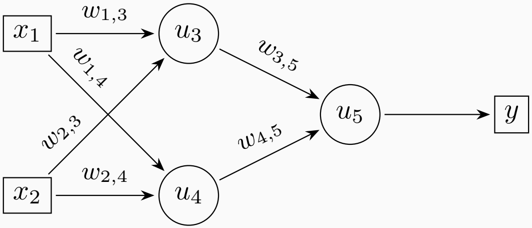
$$\hat{y} = g_5(w_{0,5} + w_{3,5}a_3 + w_{4,5}a_4)$$

$$\begin{aligned}\hat{y} = g_5(w_{0,5} + w_{3,5}g_3(w_{0,3} + w_{1,3}x_1 + w_{2,3}x_2) \\ + w_{4,5}g_4(w_{0,4} + w_{1,4}x_1 + w_{2,4}x_2))\end{aligned}$$

$$\boxed{h_{\boldsymbol{W}}(\boldsymbol{x}) = g^{(2)}(\boldsymbol{W}^{(2)}g^{(1)}(\boldsymbol{W}^{(1)}\boldsymbol{x}))}$$

An example

Example



Gradient descent

$Loss(h_{\mathbf{W}}) = L_2(y, h_{\mathbf{W}}(\mathbf{x})) = (y - \hat{y})^2$

Output layer

$\frac{\partial Loss(h_{\mathbf{W}})}{\partial w_{3,5}} = \ldots ?$

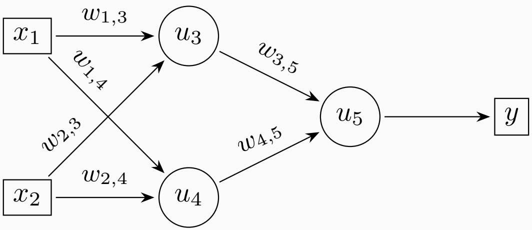
Forward computation

$$\hat{y} = g_5(w_{0,5} + w_{3,5}a_3 + w_{4,5}a_4)$$

$$\begin{aligned}\hat{y} = g_5(w_{0,5} &+ w_{3,5}g_3(w_{0,3} + w_{1,3}x_1 + w_{2,3}x_2) \\ &+ w_{4,5}g_4(w_{0,4} + w_{1,4}x_1 + w_{2,4}x_2))\end{aligned}$$

$$\boxed{h_{\mathbf{W}}(\mathbf{x}) = g^{(2)}(\mathbf{W}^{(2)}g^{(1)}(\mathbf{W}^{(1)}\mathbf{x}))}$$

Example



Forward computation

$$\hat{y} = g_5(w_{0,5} + w_{3,5}a_3 + w_{4,5}a_4)$$

$$\hat{y} = g_5(w_{0,5} + w_{3,5}g_3(w_{0,3} + w_{1,3}x_1 + w_{2,3}x_2) + w_{4,5}g_4(w_{0,4} + w_{1,4}x_1 + w_{2,4}x_2))$$

$$\boxed{h_{\mathbf{W}}(\mathbf{x}) = g^{(2)}(\mathbf{W}^{(2)}g^{(1)}(\mathbf{W}^{(1)}\mathbf{x}))}$$

Gradient descent

$Loss(h_{\mathbf{W}}) = L_2(y, h_{\mathbf{W}}(\mathbf{x})) = (y - \hat{y})^2$

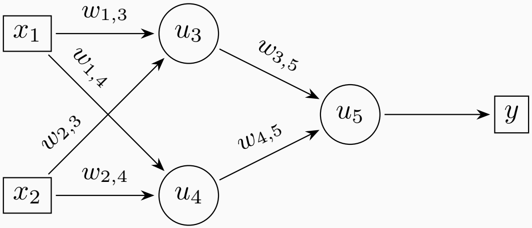
Output layer

$\frac{\partial Loss(h_{\mathbf{W}})}{\partial w_{3,5}} = \dots$

$-2(y - \hat{y})g_5'(w_{0,5} + w_{3,5}a_3 + w_{4,5}a_4)a_3 = \Delta_5 a_3$

An example

Example



Forward computation

$$\hat{y} = g_5(w_{0,5} + w_{3,5}a_3 + w_{4,5}a_4)$$

$$\hat{y} = g_5(w_{0,5} + w_{3,5}g_3(w_{0,3} + w_{1,3}x_1 + w_{2,3}x_2) + w_{4,5}g_4(w_{0,4} + w_{1,4}x_1 + w_{2,4}x_2))$$

$$\boxed{h_{\mathbf{W}}(\mathbf{x}) = g^{(2)}(\mathbf{W}^{(2)}g^{(1)}(\mathbf{W}^{(1)}\mathbf{x}))}$$

Gradient descent

$Loss(h_{\mathbf{W}}) = L_2(y, h_{\mathbf{W}}(\mathbf{x})) = (y - \hat{y})^2$

Output layer

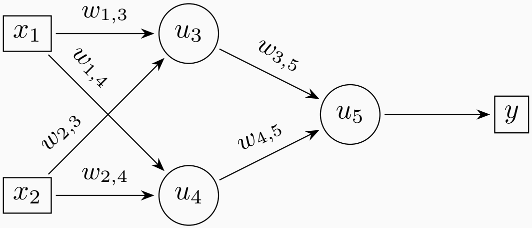
$\frac{\partial Loss(h_W)}{\partial w_{3,5}} = \ldots$

$-2(y - \hat{y})g_5'(w_{0,5} + w_{3,5}a_3 + w_{4,5}a_4)a_3 = \Delta_5 a_3$

Hidden layer

$\frac{\partial Loss(h_W)}{\partial w_{1,3}} = \ldots ?$

Example



Forward computation

$$\hat{y} = g_5(w_{0,5} + w_{3,5}a_3 + w_{4,5}a_4)$$

$$\hat{y} = g_5(w_{0,5} + w_{3,5}g_3(w_{0,3} + w_{1,3}x_1 + w_{2,3}x_2) + w_{4,5}g_4(w_{0,4} + w_{1,4}x_1 + w_{2,4}x_2))$$

$$\boxed{h_{\boldsymbol{W}}(\boldsymbol{x}) = g^{(2)}(\boldsymbol{W}^{(2)}g^{(1)}(\boldsymbol{W}^{(1)}\boldsymbol{x}))}$$

Gradient descent

$Loss(h_{\boldsymbol{W}}) = L_2(y, h_{\boldsymbol{W}}(\boldsymbol{x})) = (y - \hat{y})^2$

Output layer

$\frac{\partial Loss(h_{W})}{\partial w_{3,5}} = \dots$

$-2(y - \hat{y})g_5'(w_{0,5} + w_{3,5}a_3 + w_{4,5}a_4)a_3 = \Delta_5 a_3$

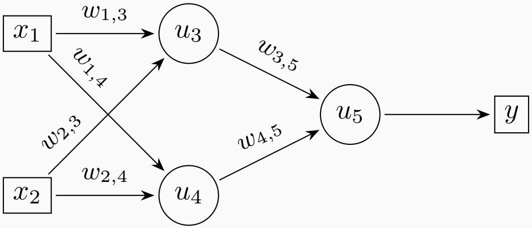
Hidden layer

$\frac{\partial Loss(h_{W})}{\partial w_{1,3}} = \dots$

$\Delta_5 w_{3,5} g_3'(w_{0,3} + w_{1,3}x_1 + w_{2,3}x_2)x_1 = \Delta_3 x_1$

An example

Example



Forward computation

$$\hat{y} = g_5(w_{0,5} + w_{3,5}a_3 + w_{4,5}a_4)$$

$$\hat{y} = g_5(w_{0,5} + w_{3,5}g_3(w_{0,3} + w_{1,3}x_1 + w_{2,3}x_2) + w_{4,5}g_4(w_{0,4} + w_{1,4}x_1 + w_{2,4}x_2))$$

$$\boxed{h_{\mathbf{W}}(\mathbf{x}) = g^{(2)}(\mathbf{W}^{(2)}g^{(1)}(\mathbf{W}^{(1)}\mathbf{x}))}$$

Gradient descent

$Loss(h_{\mathbf{W}}) = L_2(y, h_{\mathbf{W}}(\mathbf{x})) = (y - \hat{y})^2$

Output layer

$\frac{\partial Loss(h_{\mathbf{W}})}{\partial w_{3,5}} = \dots$

$-2(y - \hat{y})g_5'(w_{0,5} + w_{3,5}a_3 + w_{4,5}a_4)a_3 = \Delta_5 a_3$

Hidden layer

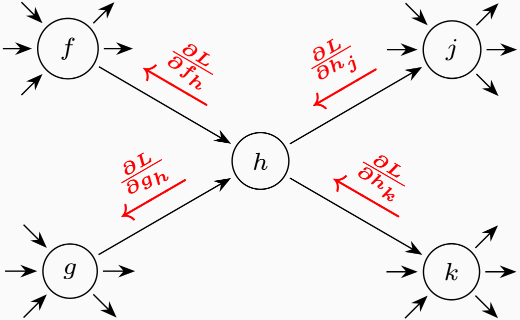
$\frac{\partial Loss(h_{\mathbf{W}})}{\partial w_{1,3}} = \dots$

$\Delta_5 w_{3,5} g_3'(w_{0,3} + w_{1,3}x_1 + w_{2,3}x_2)x_1 = \Delta_3 x_1$

Vanishing gradient

When $g_i'(in_i) \approx 0 \rightarrow$ learning stops

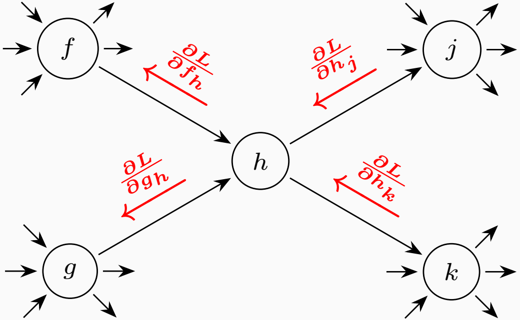
Backpropagation



h_j : message from node h to node j

$(h_j = h(f_h, g_h))$

Backpropagation



h_j : message from node h to node j

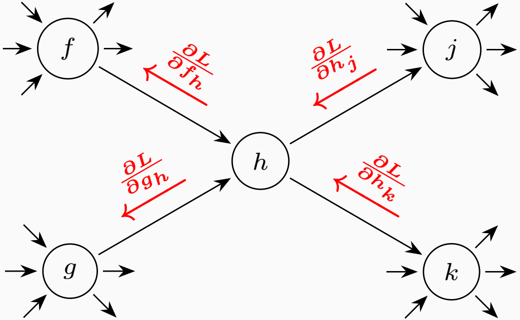
$(h_j = h(f_h, g_h))$

Contribution of h on L

$$\boxed{\frac{\partial L}{\partial h} = \frac{\partial L}{\partial h_j} + \frac{\partial L}{\partial h_k}}$$

Learning algorithms - Backpropagation

Backpropagation



h_j : message from node h to node j
$(h_j = h(f_h, g_h))$

Contribution of h on L

$$\boxed{\frac{\partial L}{\partial h} = \frac{\partial L}{\partial h_j} + \frac{\partial L}{\partial h_k}}$$

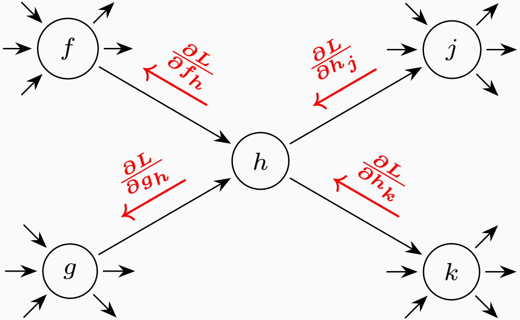
Backpropagate

$$\boxed{\frac{\partial L}{\partial f_h} = \frac{\partial L}{\partial h}\frac{\partial h}{\partial f_h}} \text{ and } \boxed{\frac{\partial L}{\partial g_h} = \frac{\partial L}{\partial h}\frac{\partial h}{\partial g_h}}$$

- $\frac{\partial L}{\partial h}$: already computed at previous step
- $\frac{\partial h}{\partial f_h}$: specific to the type of node h

Learning algorithms - Backpropagation

Backpropagation



h_j : message from node h to node j

$(h_j = h(f_h, g_h))$

Contribution of h on L

$$\boxed{\frac{\partial L}{\partial h} = \frac{\partial L}{\partial h_j} + \frac{\partial L}{\partial h_k}}$$

Backpropagate

$$\boxed{\frac{\partial L}{\partial f_h} = \frac{\partial L}{\partial h}\frac{\partial h}{\partial f_h}} \text{ and } \boxed{\frac{\partial L}{\partial g_h} = \frac{\partial L}{\partial h}\frac{\partial h}{\partial g_h}}$$

- $\frac{\partial L}{\partial h}$: already computed at previous step
- $\frac{\partial h}{\partial f_h}$: specific to the type of node h

Until ..

.. we reach a node corresponding to a parameter w : $\frac{\partial L}{\partial w} \rightarrow$ update w

Learning algorithms - Enhancements

General gradient descent
$\boldsymbol{W} \leftarrow \boldsymbol{W} - \alpha \nabla_{\boldsymbol{W}} L(\boldsymbol{W})$

Learning algorithms - Enhancements

General gradient descent
$\boldsymbol{W} \leftarrow \boldsymbol{W} - \alpha \nabla_{\boldsymbol{W}} L(\boldsymbol{W})$

Batches

- When $\boldsymbol{W}$ dimensionality and the training set are very large $\rightarrow$ **minibatch**
- Gradient contributions of each batch are independent $\rightarrow$ **parallel** computing (GPU or TPU)

Learning algorithms - Enhancements

General gradient descent
$\boldsymbol{W} \leftarrow \boldsymbol{W} - \alpha \nabla_{\boldsymbol{W}} L(\boldsymbol{W})$

Batches

- When $\boldsymbol{W}$ dimensionality and the training set are very large $\rightarrow$ **minibatch**
- Gradient contributions of each batch are independent $\rightarrow$ **parallel** computing (GPU or TPU)

Decreasing learning rate
$\alpha(t)$ **decreasing** function $\rightarrow$ find the right schedule

Learning algorithms - Enhancements

General gradient descent
$\boldsymbol{W} \leftarrow \boldsymbol{W} - \alpha \nabla_{\boldsymbol{W}} L(\boldsymbol{W})$

Batches

- When $\boldsymbol{W}$ dimensionality and the training set are very large $\rightarrow$ **minibatch**
- Gradient contributions of each batch are independent $\rightarrow$ **parallel** computing (GPU or TPU)

Decreasing learning rate
$\alpha(t)$ **decreasing** function $\rightarrow$ find the right schedule

Gradient has high variance on small batches
and thus may point to a wrong direction ..

Learning algorithms - Enhancements

General gradient descent
$\boldsymbol{W} \leftarrow \boldsymbol{W} - \alpha \nabla_{\boldsymbol{W}} L(\boldsymbol{W})$

Batches

- When $\boldsymbol{W}$ dimensionality and the training set are very large $\rightarrow$ **minibatch**
- Gradient contributions of each batch are independent $\rightarrow$ **parallel** computing (GPU or TPU)

Decreasing learning rate
$\alpha(t)$ **decreasing** function $\rightarrow$ find the right schedule

Gradient has high variance on small batches
and thus may point to a wrong direction ..

- increase minibatch size as training proceeds

Learning algorithms - Enhancements

General gradient descent
$\boldsymbol{W} \leftarrow \boldsymbol{W} - \alpha \nabla_{\boldsymbol{W}} L(\boldsymbol{W})$

Batches

- When $\boldsymbol{W}$ dimensionality and the training set are very large $\rightarrow$ **minibatch**
- Gradient contributions of each batch are independent $\rightarrow$ **parallel** computing (GPU or TPU)

Decreasing learning rate
$\alpha(t)$ **decreasing** function $\rightarrow$ find the right schedule

Gradient has high variance on small batches
and thus may point to a wrong direction ..

- increase minibatch size as training proceeds
- **momemtum** : keep a running average of the gradient

Learning algorithms - Enhancements

General gradient descent
$\boldsymbol{W} \leftarrow \boldsymbol{W} - \alpha \nabla_{\boldsymbol{W}} L(\boldsymbol{W})$

Batches

- When $\boldsymbol{W}$ dimensionality and the training set are very large $\rightarrow$ **minibatch**
- Gradient contributions of each batch are independent $\rightarrow$ **parallel** computing (GPU or TPU)

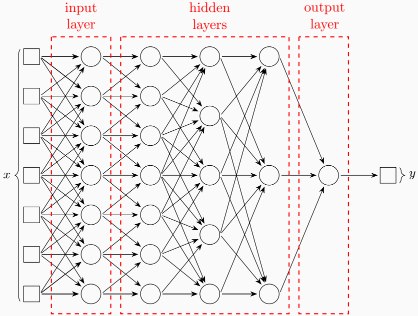
Decreasing learning rate
$\alpha(t)$ **decreasing** function $\rightarrow$ find the right schedule

Gradient has high variance on small batches
and thus may point to a wrong direction ..

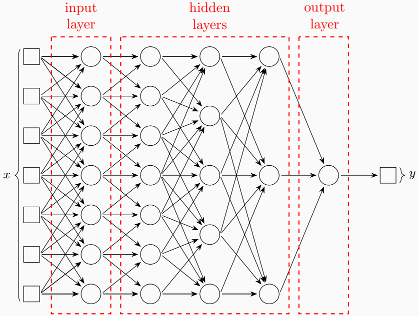
- increase minibatch size as training proceeds
- **momemtum** : keep a running average of the gradient

Batch normalization
For each example i of the minibatch, replace each output z_i of each node by $\hat{z}_i = \gamma \frac{z_i - \mu}{\sqrt{\varepsilon + \sigma^2}} + \beta$ (μ : mean, σ : standard deviation, within the minibatch) ($\varepsilon > 0$) (γ and β : new parameters)

Layers



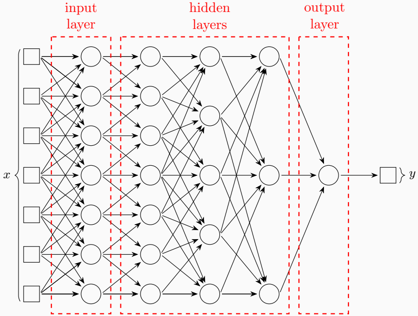
Layers



Input encoding

- generally straighforward : $\{\top, \bot\} \rightarrow \{0, 1\}$, $\mathbb{R} \rightarrow \mathbb{R}$, *log* scale for big magnitudes, ...

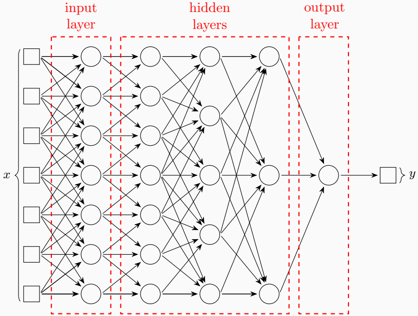
Layers



Input encoding

- generally straighforward : $\{\top, \bot\} \rightarrow \{0, 1\}$, $\mathbb{R} \rightarrow \mathbb{R}$, *log* scale for big magnitudes, ...
- categories $\rightarrow$ **one-hot encoding**

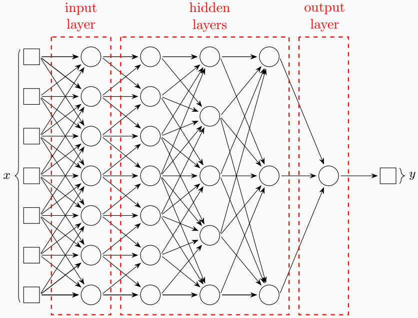
Layers



Input encoding

- generally straighforward : $\{\top, \bot\} \to \{0, 1\}$, $\mathbb{R} \to \mathbb{R}$, *log* scale for big magnitudes, ...
- categories $\to$ **one-hot encoding**
- images $\to$ array-like structure to represent **adjacency**

Layers



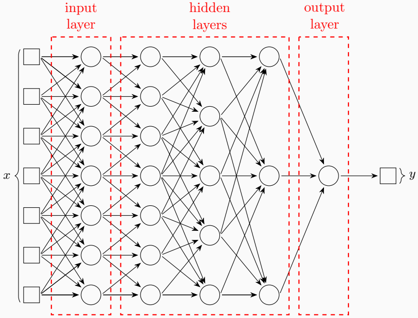
Input encoding

- generally straightforward : $\{\top, \bot\} \to \{0, 1\}$, $\mathbb{R} \to \mathbb{R}$, *log* scale for big magnitudes, ...
- categories $\to$ **one-hot encoding**
- images $\to$ array-like structure to represent **adjacency**

Output encoding

- multiclass $\to$ one-hot encoding : probability to be in the class k
  softmax layer : $softmax(\vec{in})_k = \frac{e^{in_k}}{\sum_{k'} e^{in_{k'}}}$

Layers



Input encoding

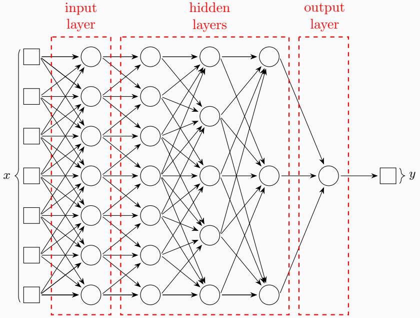
- generally straightforward : $\{\top, \bot\} \to \{0, 1\}$, $\mathbb{R} \to \mathbb{R}$, *log* scale for big magnitudes, ...
- categories $\to$ **one-hot encoding**
- images $\to$ array-like structure to represent **adjacency**

Output encoding

- multiclass $\to$ one-hot encoding : probability to be in the class k

  softmax layer : $softmax(\vec{in})_k = \frac{e^{in_k}}{\sum_{k'} e^{in_{k'}}}$
- regression $\to$ linear layer

Layers



Hidden layer

- 1985-2010 : *sigmoid* or *tanh*
- now : *ReLU* and *softplus* more popular (vanishing gradient)

Input encoding

- generally straighforward : $\{\top, \bot\} \rightarrow \{0, 1\}$, $\mathbb{R} \rightarrow \mathbb{R}$, *log* scale for big magnitudes, ...
- categories $\rightarrow$ **one-hot encoding**
- images $\rightarrow$ array-like structure to represent **adjacency**

Output encoding

- multiclass $\rightarrow$ one-hot encoding : probability to be in the class k
  softmax layer : $softmax(\vec{in})_k = \frac{e^{in_k}}{\sum_{k'} e^{in_{k'}}}$
- regression $\rightarrow$ linear layer

Cross-entropy

Multiclass Classification
Interpret $\hat{y}$ as probabilities

Cross-entropy

Multiclass Classification
Interpret $\hat{y}$ as probabilities

Cross-entropy
Measure of dissimilarity between two distributions P and Q :

$$H(P, Q) = -E_{\boldsymbol{z} \sim P(\boldsymbol{z})}(\log Q(\boldsymbol{z})) = -\int P(\boldsymbol{z}) \log Q(\boldsymbol{z}) d\boldsymbol{z}$$

Cross-entropy

Multiclass Classification
Interpret $\hat{y}$ as probabilities

Cross-entropy
Measure of dissimilarity between two distributions P and Q :

$$H(P, Q) = -E_{\boldsymbol{z} \sim P(\boldsymbol{z})}(\log Q(\boldsymbol{z})) = -\int P(\boldsymbol{z}) \log Q(\boldsymbol{z}) d\boldsymbol{z}$$

For classification

- P : the true distribution over training examples
- Q : the predictive hypothesis

Cross-entropy

Multiclass Classification
Interpret $\hat{y}$ as probabilities

Cross-entropy
Measure of dissimilarity between two distributions P and Q :

$$H(P, Q) = -E_{\boldsymbol{z} \sim P(\boldsymbol{z})}(\log Q(\boldsymbol{z})) = -\int P(\boldsymbol{z}) \log Q(\boldsymbol{z}) d\boldsymbol{z}$$

For classification

- P : the true distribution over training examples
- Q : the predictive hypothesis

Binary classification

- probability of output $y = 1$: $q_{y=1} = \hat{y}$
- probability of output $y = 0$: $q_{y=0} = 1 - \hat{y}$

$$H(p, q) = -\sum_i p_i \log q_i = -y \log \hat{y} - (1 - y) \log(1 - \hat{y})$$

Cross-entropy

Multiclass Classification
Interpret $\hat{y}$ as probabilities

Cross-entropy
Measure of dissimilarity between two distributions P and Q :

$$H(P, Q) = -E_{\boldsymbol{z}\sim P(\boldsymbol{z})}(\log Q(\boldsymbol{z})) =$$
$$-\int P(\boldsymbol{z}) \log Q(\boldsymbol{z}) d\boldsymbol{z}$$

For classification

- P : the true distribution over training examples
- Q : the predictive hypothesis

Binary classification

- probability of output $y = 1$: $q_{y=1} = \hat{y}$
- probability of output $y = 0$: $q_{y=0} = 1 - \hat{y}$

$$H(p, q) = -\sum_i p_i \log q_i =$$
$$-y \log \hat{y} - (1 - y) \log(1 - \hat{y})$$

Cross-entropy loss

$$L(\boldsymbol{w}) = \frac{1}{N} \sum_{k=1}^{N} H(p_k, q_k)$$

$$\boxed{L(\boldsymbol{w}) = -\frac{1}{N} \sum_{k=1}^{N} (y_k \log \hat{y_k} + (1 - y_k) \log(1 - \hat{y_k}))}$$

Image specificities

Convolutional Networks

Image specificities

- **adjacency** $\rightarrow$ units should receive input from a *small local* region

Convolutional Networks

Image specificities

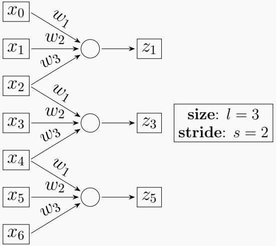
- **adjacency** $\rightarrow$ units should receive input from a *small local* region
- **space invariance** $\rightarrow$ units should share their weights

Convolutional Networks

Image specificities

- **adjacency** $\rightarrow$ units should receive input from a *small local* region
- **space invariance** $\rightarrow$ units should share their weights

1D example



Convolution

- **kernel** : pattern of weights that is *replicated*

Convolutional Networks

Image specificities

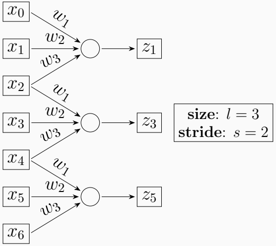
- **adjacency** $\rightarrow$ units should receive input from a *small local* region
- **space invariance** $\rightarrow$ units should share their weights

Convolution

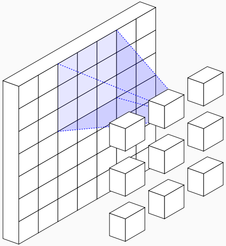
- **kernel** : pattern of weights that is *replicated*
- **convolution** : apply a kernel $\boldsymbol{k}$ of size l :

$$\boxed{\boldsymbol{z} = \boldsymbol{x} * \boldsymbol{k}} \rightarrow z_i = \sum_{j=1}^{l} k_j x_{j+i-(l+1)/2}$$

1D example



2D pattern



Convolutional Networks

Image specificities

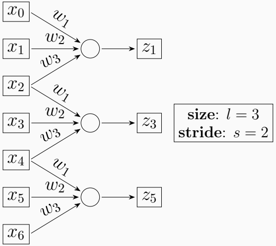
- **adjacency** $\rightarrow$ units should receive input from a *small local* region
- **space invariance** $\rightarrow$ units should share their weights

Convolution

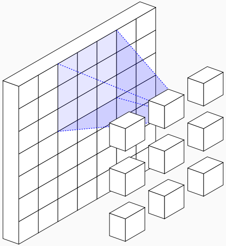
- **kernel** : pattern of weights that is *replicated*
- **convolution** : apply a kernel $\boldsymbol{k}$ of size l :

$$\boxed{\boldsymbol{z} = \boldsymbol{x} * \boldsymbol{k}} \rightarrow z_i = \sum_{j=1}^{l} k_j x_{j+i-(l+1)/2}$$

1D example



2D pattern



Pooling

- **average pooling** : $\boldsymbol{k} = (\frac{1}{l}, \ldots, \frac{1}{l})$ (if $s > 1$: *downsampling*)

Image specificities

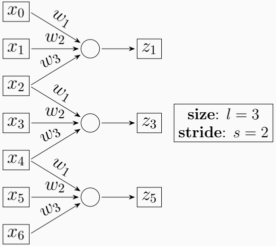
- **adjacency** $\rightarrow$ units should receive input from a *small local* region
- **space invariance** $\rightarrow$ units should share their weights

Convolution

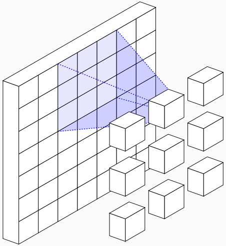
- **kernel** : pattern of weights that is *replicated*
- **convolution** : apply a kernel $\boldsymbol{k}$ of size l :

$$\boxed{\boldsymbol{z} = \boldsymbol{x} * \boldsymbol{k}} \rightarrow z_i = \sum_{j=1}^{l} k_j x_{j+i-(l+1)/2}$$

1D example



2D pattern



Pooling

- **average pooling** : $\boldsymbol{k} = (\frac{1}{l}, \ldots, \frac{1}{l})$ (if $s > 1$: *downsampling*)
- **max-pooling** : $z_i = \max_{1 \leq j \leq l}(x_{j+i-(l+1)/2})$

Tensor
Multidimensional arrays of any dimension :

- 1D : vector
- 2D : matrix
- ...

Tensor
Multidimensional arrays of any dimension :

- 1D : vector
- 2D : matrix
- ...

Example

input
minibatch of 64 images RGB 256x256
256x256x3x64

Tensor

Tensor
Multidimensional arrays of any dimension :

- 1D : vector
- 2D : matrix
- ...

Example

input	$\longrightarrow$	output
minibatch of 64 images RGB 256x256	96 kernels 5x5x3 with $s = 2$	**feature map**
256x256x3x64	$\longrightarrow$	

Tensor

Tensor
Multidimensional arrays of any dimension :

- 1D : vector
- 2D : matrix
- ...

Example

input	$\longrightarrow$	output
minibatch of 64 images RGB 256x256	96 kernels 5x5x3 with $s = 2$	**feature map**
256x256x3x64	$\longrightarrow$	128x128x96x64

Residual Networks

Idea
To avoid vanishing gradient in very deep networks $\rightarrow$ keep information of the previous layer

Residual Networks

Idea
To avoid vanishing gradient in very deep networks $\rightarrow$ keep information of the previous layer

Residual

Instead of $\boldsymbol{z}^{(i)} = h(\boldsymbol{z}^{(i-1)}) = g^{(i)}(\boldsymbol{W}^{(i)}\boldsymbol{z}^{(i-1)}) \rightarrow \boxed{\boldsymbol{z}^{(i)} = g_r^{(i)}(\boldsymbol{z}^{(i-1)} + f(\boldsymbol{z}^{(i-1)}))}$

Residual Networks

Idea
To avoid vanishing gradient in very deep networks $\rightarrow$ keep information of the previous layer

Residual
Instead of $\boldsymbol{z}^{(i)} = h(\boldsymbol{z}^{(i-1)}) = g^{(i)}(\boldsymbol{W}^{(i)}\boldsymbol{z}^{(i-1)}) \rightarrow \boxed{\boldsymbol{z}^{(i)} = g_r^{(i)}(\boldsymbol{z}^{(i-1)} + f(\boldsymbol{z}^{(i-1)}))}$

- $g_r^{(i)}$: activation function

Residual Networks

Idea
To avoid vanishing gradient in very deep networks $\rightarrow$ keep information of the previous layer

Residual

Instead of $\boldsymbol{z}^{(i)} = h(\boldsymbol{z}^{(i-1)}) = g^{(i)}(\boldsymbol{W}^{(i)}\boldsymbol{z}^{(i-1)}) \rightarrow \boxed{\boldsymbol{z}^{(i)} = g_r^{(i)}(\boldsymbol{z}^{(i-1)} + f(\boldsymbol{z}^{(i-1)}))}$

- $g_r^{(i)}$: activation function
- f typically a linear + non-linear function : $f(\boldsymbol{z}) = \boldsymbol{V}g(\boldsymbol{W}\boldsymbol{z})$

Residual Networks

Idea
To avoid vanishing gradient in very deep networks $\rightarrow$ keep information of the previous layer

Residual
Instead of $\boldsymbol{z}^{(i)} = h(\boldsymbol{z}^{(i-1)}) = g^{(i)}(\boldsymbol{W}^{(i)}\boldsymbol{z}^{(i-1)}) \rightarrow \boxed{\boldsymbol{z}^{(i)} = g_r^{(i)}(\boldsymbol{z}^{(i-1)} + f(\boldsymbol{z}^{(i-1)}))}$

- $g_r^{(i)}$: activation function
- f typically a linear + non-linear function : $f(\boldsymbol{z}) = \boldsymbol{V}g(\boldsymbol{W}\boldsymbol{z})$

Disable a layer
We can make layers that can be disabled by setting $\boldsymbol{V} = \boldsymbol{0}$: if $g_r = ReLU$ (at least for layers $i-1$ and i), $\boldsymbol{z}^{(i-1)} = ReLU(\boldsymbol{in}^{(i-1)})$ then
$\boldsymbol{z}^{(i)} = ReLU(\boldsymbol{z}^{(i-1)}) = ReLU(ReLU(\boldsymbol{in}^{(i-1)})) = ReLU(\boldsymbol{in}^{(i-1)}) = \boldsymbol{z}^{(i-1)}$

Recurrent Networks - Basic

Time series
A sequence of inputs $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T$ and observed outputs $\boldsymbol{y}_1, \ldots, \boldsymbol{y}_T$.

Recurrent Networks - Basic

Time series
A sequence of inputs $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T$ and observed outputs $\boldsymbol{y}_1, \ldots, \boldsymbol{y}_T$.

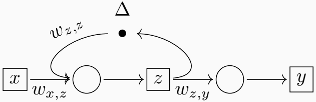
Signal or Text processing

- **time series** $\rightarrow$ we need a **memory** $\boldsymbol{z}$
- **time invariance** $\rightarrow$ share weights at each time step

Recurrent Networks - Basic

Time series
A sequence of inputs $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T$ and observed outputs $\boldsymbol{y}_1, \ldots, \boldsymbol{y}_T$.

Basic RNN



Signal or Text processing

- **time series** $\rightarrow$ we need a **memory** $\boldsymbol{z}$
- **time invariance** $\rightarrow$ share weights at each time step

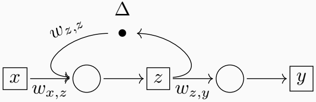
Time series

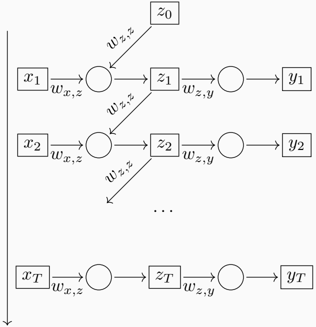
A sequence of inputs $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T$ and observed outputs $\boldsymbol{y}_1, \ldots, \boldsymbol{y}_T$.

Signal or Text processing

- **time series** $\rightarrow$ we need a **memory** $\boldsymbol{z}$
- **time invariance** $\rightarrow$ share weights at each time step

Basic RNN





Recurrent Networks - Basic

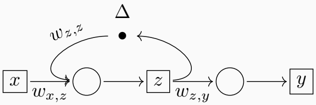
Time series

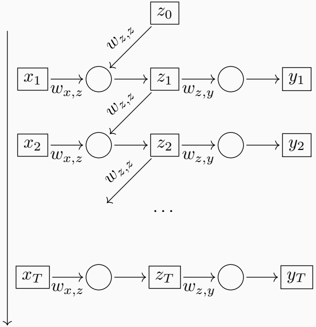
A sequence of inputs $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T$ and observed outputs $\boldsymbol{y}_1, \ldots, \boldsymbol{y}_T$.

Signal or Text processing

- **time series** $\rightarrow$ we need a **memory** $\boldsymbol{z}$
- **time invariance** $\rightarrow$ share weights at each time step

Basic RNN





Forward

$z_t = g_z(w_{z,z} z_{t-1} + w_{x,z} x_t)$

and $\hat{y}_t = g_y(w_{y,z} z_t)$

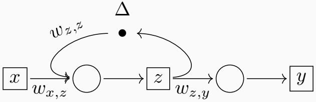
Time series

A sequence of inputs $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T$ and observed outputs $\boldsymbol{y}_1, \ldots, \boldsymbol{y}_T$.

Signal or Text processing

- **time series** $\rightarrow$ we need a **memory** $\boldsymbol{z}$
- **time invariance** $\rightarrow$ share weights at each time step

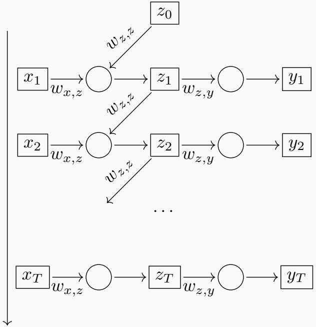
Basic RNN



Forward

$z_t = g_z(w_{z,z} z_{t-1} + w_{x,z} x_t)$

and $\hat{y}_t = g_y(w_{y,z} z_t)$



Backpropagation

$$\frac{\partial L}{\partial w_{z,z}} = \sum_{t=1}^{T} -2(y_t - \hat{y}_t) g_y'(in_{y,t}) w_{z,y} \frac{\partial z_t}{w_{z,z}}$$

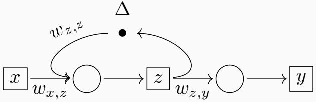
$$\frac{\partial z_t}{\partial w_{z,z}} = g_z'(in_{z,t})(z_{t-1} + w_{z,z} \frac{\partial z_{t-1}}{w_{z,z}})$$

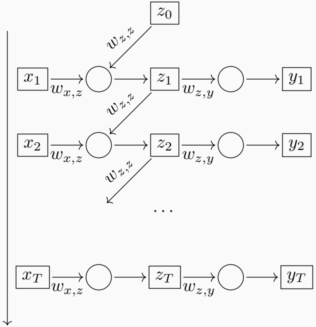
Time series
A sequence of inputs $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T$ and observed outputs $\boldsymbol{y}_1, \ldots, \boldsymbol{y}_T$.

Signal or Text processing

- **time series** $\rightarrow$ we need a **memory** $\boldsymbol{z}$
- **time invariance** $\rightarrow$ share weights at each time step

Basic RNN





Forward
$z_t = g_z(w_{z,z} z_{t-1} + w_{x,z} x_t)$
and $\hat{y}_t = g_y(w_{y,z} z_t)$

Backpropagation

$$\frac{\partial L}{\partial w_{z,z}} = \sum_{t=1}^{T} -2(y_t - \hat{y}_t) g_y'(in_{y,t}) w_{z,y} \frac{\partial z_t}{w_{z,z}}$$

$$\frac{\partial z_t}{\partial w_{z,z}} = g_z'(in_{z,t})(z_{t-1} + w_{z,z} \frac{\partial z_{t-1}}{w_{z,z}})$$

Issue
Gradient at step T will include terms proportional to $w_{z,z} \prod_{t=1}^{T} g_z'(in_{z,t})$

$\hookrightarrow$ vanishing ($w_{z,z} < 1$) or *exploding* ($w_{z,z} > 1$) gradient

Long Short-Term Memory (LSTM)

- **memory cell** $\boldsymbol{c}$: *copied* at each time step

Long Short-Term Memory (LSTM)

- **memory cell $\boldsymbol{c}$** : *copied* at each time step

Gating units :

Long Short-Term Memory (LSTM)

- **memory cell $\boldsymbol{c}$** : *copied* at each time step

Gating units :

- **forget gate $\boldsymbol{f}$** : elements of the memory to *forget/remember*

Long Short-Term Memory (LSTM)

- **memory cell $\boldsymbol{c}$** : *copied* at each time step

Gating units :

- **forget gate $\boldsymbol{f}$** : elements of the memory to *forget/remember*
- **input gate $\boldsymbol{i}$** : elements of the memory to *update with new info* from the inputs

Long Short-Term Memory (LSTM)

- **memory cell $\boldsymbol{c}$** : *copied* at each time step

Gating units :

- **forget gate $\boldsymbol{f}$** : elements of the memory to *forget/remember*
- **input gate $\boldsymbol{i}$** : elements of the memory to *update with new info* from the inputs
- **output gate $\boldsymbol{o}$** : elements of the memory to *transfer* to the short-term memory

Long Short-Term Memory (LSTM)

- **memory cell $\boldsymbol{c}$** : *copied* at each time step

Gating units :

- **forget gate $\boldsymbol{f}$** : elements of the memory to *forget/remember*
- **input gate $\boldsymbol{i}$** : elements of the memory to *update with new info* from the inputs
- **output gate $\boldsymbol{o}$** : elements of the memory to *transfer* to the short-term memory
- **short-term memory $\boldsymbol{z}$** : as for basic RNN

Long Short-Term Memory (LSTM)

- **memory cell $\boldsymbol{c}$** : *copied* at each time step

Gating units :

- **forget gate $\boldsymbol{f}$** : elements of the memory to *forget/remember*
- **input gate $\boldsymbol{i}$** : elements of the memory to *update with new info* from the inputs
- **output gate $\boldsymbol{o}$** : elements of the memory to *transfer* to the short-term memory
- **short-term memory $\boldsymbol{z}$** : as for basic RNN

Gating units

- $\boldsymbol{f}_t = \sigma(\boldsymbol{W}_{x,f}\boldsymbol{x}_t + \boldsymbol{W}_{z,f}\boldsymbol{z}_{t-1})$

Long Short-Term Memory (LSTM)

- **memory cell $\boldsymbol{c}$** : *copied* at each time step

Gating units :

- **forget gate $\boldsymbol{f}$** : elements of the memory to *forget/remember*
- **input gate $\boldsymbol{i}$** : elements of the memory to *update with new info* from the inputs
- **output gate $\boldsymbol{o}$** : elements of the memory to *transfer* to the short-term memory
- **short-term memory $\boldsymbol{z}$** : as for basic RNN

Gating units

- $\boldsymbol{f}_t = \sigma(\boldsymbol{W}_{x,f}\boldsymbol{x}_t + \boldsymbol{W}_{z,f}\boldsymbol{z}_{t-1})$
- $\boldsymbol{i}_t = \sigma(\boldsymbol{W}_{x,i}\boldsymbol{x}_t + \boldsymbol{W}_{z,i}\boldsymbol{z}_{t-1})$

Long Short-Term Memory (LSTM)

- **memory cell $\boldsymbol{c}$** : *copied* at each time step

Gating units :

- **forget gate $\boldsymbol{f}$** : elements of the memory to *forget/remember*
- **input gate $\boldsymbol{i}$** : elements of the memory to *update with new info* from the inputs
- **output gate $\boldsymbol{o}$** : elements of the memory to *transfer* to the short-term memory
- **short-term memory $\boldsymbol{z}$** : as for basic RNN

Gating units

- $\boldsymbol{f}_t = \sigma(\boldsymbol{W}_{x,f}\boldsymbol{x}_t + \boldsymbol{W}_{z,f}\boldsymbol{z}_{t-1})$
- $\boldsymbol{i}_t = \sigma(\boldsymbol{W}_{x,i}\boldsymbol{x}_t + \boldsymbol{W}_{z,i}\boldsymbol{z}_{t-1})$
- $\boldsymbol{o}_t = \sigma(\boldsymbol{W}_{x,o}\boldsymbol{x}_t + \boldsymbol{W}_{z,o}\boldsymbol{z}_{t-1})$

Long Short-Term Memory (LSTM)

- **memory cell $\boldsymbol{c}$** : *copied* at each time step

Gating units :

- **forget gate $\boldsymbol{f}$** : elements of the memory to *forget/remember*
- **input gate $\boldsymbol{i}$** : elements of the memory to *update with new info* from the inputs
- **output gate $\boldsymbol{o}$** : elements of the memory to *transfer* to the short-term memory
- **short-term memory $\boldsymbol{z}$** : as for basic RNN

Gating units

- $\boldsymbol{f}_t = \sigma(\boldsymbol{W}_{x,f}\boldsymbol{x}_t + \boldsymbol{W}_{z,f}\boldsymbol{z}_{t-1})$
- $\boldsymbol{i}_t = \sigma(\boldsymbol{W}_{x,i}\boldsymbol{x}_t + \boldsymbol{W}_{z,i}\boldsymbol{z}_{t-1})$
- $\boldsymbol{o}_t = \sigma(\boldsymbol{W}_{x,o}\boldsymbol{x}_t + \boldsymbol{W}_{z,o}\boldsymbol{z}_{t-1})$
- $\boldsymbol{c}_t = \boldsymbol{c}_{t-1} \odot \boldsymbol{f}_t + \boldsymbol{i}_t \odot tanh(\boldsymbol{W}_{x,c}\boldsymbol{x}_t + \boldsymbol{W}_{z,c}\boldsymbol{z}_{t-1})$

Recurrent Networks - LSTM

Long Short-Term Memory (LSTM)

- **memory cell $\boldsymbol{c}$** : *copied* at each time step

Gating units :

- **forget gate $\boldsymbol{f}$** : elements of the memory to *forget/remember*
- **input gate $\boldsymbol{i}$** : elements of the memory to *update with new info* from the inputs
- **output gate $\boldsymbol{o}$** : elements of the memory to *transfer* to the short-term memory
- **short-term memory $\boldsymbol{z}$** : as for basic RNN

Gating units

- $\boldsymbol{f}_t = \sigma(\boldsymbol{W}_{x,f}\boldsymbol{x}_t + \boldsymbol{W}_{z,f}\boldsymbol{z}_{t-1})$
- $\boldsymbol{i}_t = \sigma(\boldsymbol{W}_{x,i}\boldsymbol{x}_t + \boldsymbol{W}_{z,i}\boldsymbol{z}_{t-1})$
- $\boldsymbol{o}_t = \sigma(\boldsymbol{W}_{x,o}\boldsymbol{x}_t + \boldsymbol{W}_{z,o}\boldsymbol{z}_{t-1})$
- $\boldsymbol{c}_t = \boldsymbol{c}_{t-1} \odot \boldsymbol{f}_t + \boldsymbol{i}_t \odot tanh(\boldsymbol{W}_{x,c}\boldsymbol{x}_t + \boldsymbol{W}_{z,c}\boldsymbol{z}_{t-1})$
- $\boldsymbol{z}_t = tanh(\boldsymbol{c}_t) \odot \boldsymbol{o}_t$

Improve generalization – Design the architecture

Specialized architecture

- **Convolutional** : images
- **Recurrent** : text and audio signals

Improve generalization – Design the architecture

Specialized architecture

- **Convolutional** : images
- **Recurrent** : text and audio signals

Empirical result
For a fixed number of weights : *the deeper the better*

Improve generalization – Design the architecture

Specialized architecture

- **Convolutional** : images
- **Recurrent** : text and audio signals

Neural architecture search
Optimisation problem with **hyperparameters** : depth, width, connectivity, . . .

Empirical result
For a fixed number of weights : *the deeper the better*

Improve generalization – Design the architecture

Specialized architecture

- **Convolutional** : images
- **Recurrent** : text and audio signals

Neural architecture search
Optimisation problem with **hyperparameters** : depth, width, connectivity, . . .

- Grid search

Empirical result
For a fixed number of weights : *the deeper the better*

Improve generalization – Design the architecture

Specialized architecture

- **Convolutional** : images
- **Recurrent** : text and audio signals

Neural architecture search
Optimisation problem with **hyperparameters** : depth, width, connectivity, . . .

- Grid search
- Evolutionary algorithm : recombination (joining parts of two networks) + mutation (adding/removing a layer or changing a parameter value)

Empirical result
For a fixed number of weights : *the deeper the better*

Improve generalization – Design the architecture

Specialized architecture

- **Convolutional** : images
- **Recurrent** : text and audio signals

Neural architecture search
Optimisation problem with **hyperparameters** : depth, width, connectivity, . . .

- Grid search
- Evolutionary algorithm : recombination (joining parts of two networks) + mutation (adding/removing a layer or changing a parameter value)
- Hill climbing with mutations

Empirical result
For a fixed number of weights : *the deeper the better*

Improve generalization – Design the architecture

Specialized architecture

- **Convolutional** : images
- **Recurrent** : text and audio signals

Neural architecture search
Optimisation problem with **hyperparameters** : depth, width, connectivity, ...

- Grid search
- Evolutionary algorithm : recombination (joining parts of two networks) + mutation (adding/removing a layer or changing a parameter value)
- Hill climbing with mutations
- Reinforcement learning

Empirical result
For a fixed number of weights : *the deeper the better*

Improve generalization – Design the architecture

Specialized architecture

- **Convolutional** : images
- **Recurrent** : text and audio signals

Neural architecture search
Optimisation problem with **hyperparameters** : depth, width, connectivity, . . .

- Grid search
- Evolutionary algorithm : recombination (joining parts of two networks) + mutation (adding/removing a layer or changing a parameter value)
- Hill climbing with mutations
- Reinforcement learning
- Bayesian optimization

Empirical result
For a fixed number of weights : *the deeper the better*

Improve generalization – Design the architecture

Specialized architecture

- **Convolutional** : images
- **Recurrent** : text and audio signals

Neural architecture search

Optimisation problem with **hyperparameters** : depth, width, connectivity, . . .

- Grid search
- Evolutionary algorithm : recombination (joining parts of two networks) + mutation (adding/removing a layer or changing a parameter value)
- Hill climbing with mutations
- Reinforcement learning
- Bayesian optimization
- Gradient descent

Empirical result

For a fixed number of weights : *the deeper the better*

Improve generalization – Design the architecture

Specialized architecture

- **Convolutional** : images
- **Recurrent** : text and audio signals

Neural architecture search
Optimisation problem with **hyperparameters** : depth, width, connectivity, ...

- Grid search
- Evolutionary algorithm : recombination (joining parts of two networks) + mutation (adding/removing a layer or changing a parameter value)
- Hill climbing with mutations
- Reinforcement learning
- Bayesian optimization
- Gradient descent

Empirical result
For a fixed number of weights : *the deeper the better*

Train and evaluate
Reduce time of estimation : train on *test set* + evaluate on *validation set*

Improve generalization – Design the architecture

Specialized architecture

- **Convolutional** : images
- **Recurrent** : text and audio signals

Neural architecture search
Optimisation problem with **hyperparameters** : depth, width, connectivity, ...

- Grid search
- Evolutionary algorithm : recombination (joining parts of two networks) + mutation (adding/removing a layer or changing a parameter value)
- Hill climbing with mutations
- Reinforcement learning
- Bayesian optimization
- Gradient descent

Empirical result
For a fixed number of weights : *the deeper the better*

Train and evaluate
Reduce time of estimation : train on *test set* + evaluate on *validation set*

- Smaller training set

Improve generalization – Design the architecture

Specialized architecture

- **Convolutional** : images
- **Recurrent** : text and audio signals

Neural architecture search
Optimisation problem with **hyperparameters** : depth, width, connectivity, ...

- Grid search
- Evolutionary algorithm : recombination (joining parts of two networks) + mutation (adding/removing a layer or changing a parameter value)
- Hill climbing with mutations
- Reinforcement learning
- Bayesian optimization
- Gradient descent

Empirical result
For a fixed number of weights : *the deeper the better*

Train and evaluate
Reduce time of estimation : train on *test set* + evaluate on *validation set*

- Smaller training set
- Fewer batches + prediction of improvement

Improve generalization – Design the architecture

Specialized architecture

- **Convolutional** : images
- **Recurrent** : text and audio signals

Neural architecture search
Optimisation problem with **hyperparameters** : depth, width, connectivity, . . .

- Grid search
- Evolutionary algorithm : recombination (joining parts of two networks) + mutation (adding/removing a layer or changing a parameter value)
- Hill climbing with mutations
- Reinforcement learning
- Bayesian optimization
- Gradient descent

Empirical result
For a fixed number of weights : *the deeper the better*

Train and evaluate
Reduce time of estimation : train on *test set* + evaluate on *validation set*

- Smaller training set
- Fewer batches + prediction of improvement
- Reduced version of the network

Improve generalization – Design the architecture

Specialized architecture

- **Convolutional** : images
- **Recurrent** : text and audio signals

Neural architecture search
Optimisation problem with **hyperparameters** : depth, width, connectivity, . . .

- Grid search
- Evolutionary algorithm : recombination (joining parts of two networks) + mutation (adding/removing a layer or changing a parameter value)
- Hill climbing with mutations
- Reinforcement learning
- Bayesian optimization
- Gradient descent

Empirical result
For a fixed number of weights : *the deeper the better*

Train and evaluate
Reduce time of estimation : train on *test set* + evaluate on *validation set*

- Smaller training set
- Fewer batches + prediction of improvement
- Reduced version of the network
- Focus on subgraph

Improve generalization – Design the architecture

Specialized architecture

- **Convolutional** : images
- **Recurrent** : text and audio signals

Neural architecture search
Optimisation problem with **hyperparameters** : depth, width, connectivity, . . .

- Grid search
- Evolutionary algorithm : recombination (joining parts of two networks) + mutation (adding/removing a layer or changing a parameter value)
- Hill climbing with mutations
- Reinforcement learning
- Bayesian optimization
- Gradient descent

Empirical result
For a fixed number of weights : *the deeper the better*

Train and evaluate
Reduce time of estimation : train on *test set* + evaluate on *validation set*

- Smaller training set
- Fewer batches + prediction of improvement
- Reduced version of the network
- Focus on subgraph
- Learn heuristic evaluation function

Improve generalization

Weight decay

Regularization with penalty $\boxed{\lambda \sum_{i,j} \boldsymbol{W}_{i,j}^2}$, typically $\lambda = 10^{-4}$

$\hookrightarrow$ Encourage small weights
(to stay in the linear part for sigmoid activation)

Improve generalization

Weight decay

Regularization with penalty $\boxed{\lambda \sum_{i,j} \boldsymbol{W}_{i,j}^2}$, typically $\lambda = 10^{-4}$

$\hookrightarrow$ Encourage small weights
(to stay in the linear part for sigmoid activation)

Dropout
At each step of training *deactivate a random set of units*

- Encourage the detection of more features
- Make it more robust to noise

Neural Network Applications

Vision
Deep convolutional networks (since 1990s)

ImageNet competition : classification 1200000 images in 1000 categories

In 2012, *AlexNet* : error rate $< 15.3\%$ (2^{nd} : 25%) (now, error rate $< 2\%$)

Neural Network Applications

Vision
Deep convolutional networks (since 1990s)

ImageNet competition : classification 1200000 images in 1000 categories

In 2012, *AlexNet* : error rate $< 15.3\%$ (2^{nd} : 25%) (now, error rate $< 2\%$)

Natural Langage processing
Translation problems :

- Two networks : from L1 to IR + from IR to L2
- One end-to-end network $\leftarrow$ performs better

Speech recognition : representation of words with high-dimensional vectors $\rightarrow$ *word embeddings*

Neural Network Applications

Vision
Deep convolutional networks (since 1990s)

ImageNet competition : classification 1200000 images in 1000 categories

In 2012, *AlexNet* : error rate $< 15.3\%$ (2^{nd} : 25%) (now, error rate $< 2\%$)

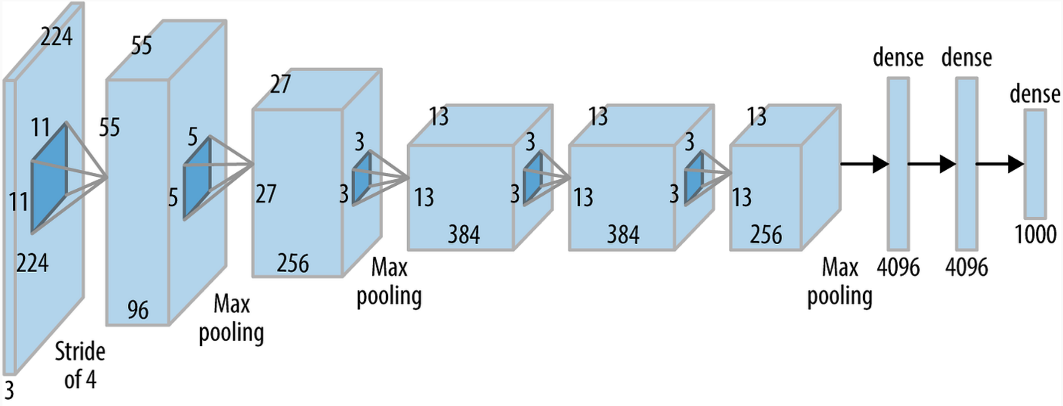
Natural Langage processing
Translation problems :

- Two networks : from L1 to IR + from IR to L2
- One end-to-end network $\leftarrow$ performs better

Speech recognition : representation of words with high-dimensional vectors $\rightarrow$ *word embeddings*

Reinforcement learning
Optimise the sum of *future rewards* : learn a value function, Q-function, policy, ...$\rightarrow$ *deep reinforcement learning*

DeepMind : *DQN* an Atari-playing agent (2013) and *AlphaGo* (2014)



Architecture of Alexnet. From left to right (input to output) five convolutional layers with Max Pooling after layers 1,2, and 5, followed by a three layer fully connected classifier (layers 6-8). The number of neurons in the output layer is equal to the designed number of output classes.

Deep Learning - Summary

- **Neural Networks** $=$ computation graph composed of parameterized linear-threshold units

Deep Learning - Summary

- **Neural Networks** = computation graph composed of parameterized linear-threshold units
- A neural network can represent complex nonlinear functions

Deep Learning - Summary

- **Neural Networks** = computation graph composed of parameterized linear-threshold units
- A neural network can represent complex nonlinear functions
- **Backpropagation** = gradient descent for neural networks

Deep Learning - Summary

- **Neural Networks** = computation graph composed of parameterized linear-threshold units
- A neural network can represent complex nonlinear functions
- **Backpropagation** = gradient descent for neural networks
- Deep learning is suited for visual object recognition, speech recognition, natural language processing and reinforcement learning

Deep Learning - Summary

- **Neural Networks** = computation graph composed of parameterized linear-threshold units
- A neural network can represent complex nonlinear functions
- **Backpropagation** = gradient descent for neural networks
- Deep learning is suited for visual object recognition, speech recognition, natural language processing and reinforcement learning
- **Convolutional** networks $\rightarrow$ data with grid topology (e.g. images)

Deep Learning - Summary

- **Neural Networks** = computation graph composed of parameterized linear-threshold units
- A neural network can represent complex nonlinear functions
- **Backpropagation** = gradient descent for neural networks
- Deep learning is suited for visual object recognition, speech recognition, natural language processing and reinforcement learning
- **Convolutional** networks $\rightarrow$ data with grid topology (e.g. images)
- **Recurrent** networks $\rightarrow$ sequence data (e.g. language modeling and machine translation)

Deep Learning - Summary

- **Neural Networks** = computation graph composed of parameterized linear-threshold units
- A neural network can represent complex nonlinear functions
- **Backpropagation** = gradient descent for neural networks
- Deep learning is suited for visual object recognition, speech recognition, natural language processing and reinforcement learning
- **Convolutional** networks $\rightarrow$ data with grid topology (e.g. images)
- **Recurrent** networks $\rightarrow$ sequence data (e.g. language modeling and machine translation)

To go further ...

- **Transfer learning** : re-train a pretrained network for a specific task

Deep Learning - Summary

- **Neural Networks** = computation graph composed of parameterized linear-threshold units
- A neural network can represent complex nonlinear functions
- **Backpropagation** = gradient descent for neural networks
- Deep learning is suited for visual object recognition, speech recognition, natural language processing and reinforcement learning
- **Convolutional** networks → data with grid topology (e.g. images)
- **Recurrent** networks → sequence data (e.g. language modeling and machine translation)

To go further . . .

- **Transfer learning** : re-train a pretrained network for a specific task
- **Generative Adversarial Networks** : a *generator* network + a *discriminator* network

Model Selection and Optimisation

Ensemble learning

Learn several hypothesis $h_1, h_2, \ldots, h_K$ and use a combination $h^* = \{h_1, h_2, \ldots, h_K\}$

- reduce **bias** of each base model by **combining**
- reduce **variance** of learning by **voting**

Ensemble learning

Learn several hypothesis $h_1, h_2, \dots, h_K$ and use a combination $h^* = \{h_1, h_2, \dots, h_K\}$

- reduce **bias** of each base model by **combining**
- reduce **variance** of learning by **voting**

Bagging

$\boxed{h^*(\boldsymbol{x}) = \frac{1}{K}\sum_{i=1}^{K} h_i(\boldsymbol{x})}$: voting in the same model class

Example : **random forests**

Ensemble learning

Learn several hypothesis $h_1, h_2, \ldots, h_K$ and use a combination $h^* = \{h_1, h_2, \ldots, h_K\}$

- reduce **bias** of each base model by **combining**
- reduce **variance** of learning by **voting**

Bagging

$$\boxed{h^*(\boldsymbol{x}) = \frac{1}{K}\sum_{i=1}^{K} h_i(\boldsymbol{x})}$$: voting in the same model class

Example : **random forests**

Stacking

Train a new hypothesis on validation set augmented with the predictions of $h_1, h_2, \ldots, h_K$

Learn several hypothesis $h_1, h_2, \dots, h_K$ and use a combination $h^* = \{h_1, h_2, \dots, h_K\}$

- reduce **bias** of each base model by **combining**
- reduce **variance** of learning by **voting**

Bagging

$\boxed{h^*(\boldsymbol{x}) = \frac{1}{K}\sum_{i=1}^{K} h_i(\boldsymbol{x})}$: voting in the same model class

Example : **random forests**

Stacking
Train a new hypothesis on validation set augmented with the predictions of $h_1, h_2, \dots, h_K$

- learn a weight for each hypothesis h_i : trust

Ensemble learning

Learn several hypothesis $h_1, h_2, \ldots, h_K$ and use a combination $h^* = \{h_1, h_2, \ldots, h_K\}$

- reduce **bias** of each base model by **combining**
- reduce **variance** of learning by **voting**

Bagging

$$\boxed{h^*(\boldsymbol{x}) = \frac{1}{K}\sum_{i=1}^{K} h_i(\boldsymbol{x})}$$: voting in the same model class

Example : **random forests**

Stacking

Train a new hypothesis on validation set augmented with the predictions of $h_1, h_2, \ldots, h_K$

- learn a weight for each hypothesis h_i : trust
- we can add metadata (e.g. time to compute) and stack several layers

Ensemble learning

Learn several hypothesis $h_1, h_2, \ldots, h_K$ and use a combination $h^* = \{h_1, h_2, \ldots, h_K\}$

- reduce **bias** of each base model by **combining**
- reduce **variance** of learning by **voting**

Bagging

$$\boxed{h^*(\boldsymbol{x}) = \frac{1}{K}\sum_{i=1}^{K} h_i(\boldsymbol{x})}$$: voting in the same model class

Example : **random forests**

Stacking

Train a new hypothesis on validation set augmented with the predictions of $h_1, h_2, \ldots, h_K$

- learn a weight for each hypothesis h_i : trust
- we can add metadata (e.g. time to compute) and stack several layers

Boosting

1. Boost incorrectly classified training example by increasing its weight (number of occurences), iterate after learning each h_i
2. Weighted voting : $h^*(\boldsymbol{x}) = \sum_{i=1}^{K} z_i h_i(\boldsymbol{x})$

Ensemble learning

Learn several hypothesis $h_1, h_2, \ldots, h_K$ and use a combination $h^* = \{h_1, h_2, \ldots, h_K\}$

- reduce **bias** of each base model by **combining**
- reduce **variance** of learning by **voting**

Bagging

$$\boxed{h^*(\boldsymbol{x}) = \frac{1}{K}\sum_{i=1}^{K} h_i(\boldsymbol{x})}$$: voting in the same model class

Example : **random forests**

Stacking
Train a new hypothesis on validation set augmented with the predictions of $h_1, h_2, \ldots, h_K$

- learn a weight for each hypothesis h_i : trust
- we can add metadata (e.g. time to compute) and stack several layers

Boosting

1. Boost incorrectly classified training example by increasing its weight (number of occurences), iterate after learning each h_i
2. Weighted voting : $h^*(\boldsymbol{x}) = \sum_{i=1}^{K} z_i h_i(\boldsymbol{x})$

Gradient boosting
Boosting with *gradient descent* to find the weight on training examples

Model Selection

- **Random Forests** : lot of categorical features and many irrelevant

Model Selection

- **Random Forests** : lot of categorical features and many irrelevant
- **Non-parametric models** : lot of data and no prior knowledge $\rightarrow$ resulting hypothesis are expensive to compute

Model Selection

- **Random Forests** : lot of categorical features and many irrelevant
- **Non-parametric models** : lot of data and no prior knowledge $\rightarrow$ resulting hypothesis are expensive to compute
- **Logistic Regression** performs similarly than **SVM**

Model Selection

- **Random Forests** : lot of categorical features and many irrelevant
- **Non-parametric models** : lot of data and no prior knowledge $\rightarrow$ resulting hypothesis are expensive to compute
- **Logistic Regression** performs similarly than **SVM**
- **SVM** : is better for not too large dataset with high dimension

Model Selection

- **Random Forests** : lot of categorical features and many irrelevant
- **Non-parametric models** : lot of data and no prior knowledge $\rightarrow$ resulting hypothesis are expensive to compute
- **Logistic Regression** performs similarly than **SVM**
- **SVM** : is better for not too large dataset with high dimension
- **Deep Neural Network** : for complex pattern recognition (e.g. image or speech processing)

Data enhancement

- Not enough data $\rightarrow$ **data augmentation** (example : image cropping/rotating/...)

Data enhancement

- Not enough data → **data augmentation** (example : image cropping/rotating/...)
- Unbalanced classes in data (example : unbalanced representation of negative vs. positive examples) → **undersample** or **oversample**

Data enhancement

- Not enough data $\rightarrow$ **data augmentation** (example : image cropping/rotating/...)
- Unbalanced classes in data (example : unbalanced representation of negative vs. positive examples) $\rightarrow$ **undersample** or **oversample**
- **Outliers** : points far from the majority $\rightarrow$ some model classes are less susceptible : decision trees

Summary

Summary

- **Supervised learning** is learning on labelled datasets
- **Regression** is learning a function with infinite output values
- **Classification** is learning a function with finite output values
- **Linear/Logistic regression** is a simple yet powerful model class for supervised learning
- **Deep Neural Networks** are computation graphs composed of units made of a non-linear and a linear function
- **Deep learning** is well suited for visual object recognition, speech recognition, natural language processing and reinforcement learning

Sources

- Artificial Intelligence : A Modern Approach, *Stuart Russell* and *Peter Norvig*
- Lecture of *Didier Lime* (2022-2023)
- Lecture of *Kilian Weinberger* : `https://courses.cis.cornell.edu/cs4780/2017sp/`