Deep Learning

Elisabetta Fersini

elisabetta.fersini@unimib.it

Main topics of the Course

- Background concepts
- Training Deep Networks:
  - Objective functions
  - Activation Functions
  - Regularization
  - Gradient-based optimization
- Deep Learning in Practice
  - Tips and Tricks
- Focus on Deep Architectures (hands on):
  - Variational Models
  - Transformers

People

- Prof. Elisabetta Fersini
  - Associate professor of CS
  - Research field:
    - Machine Learning
    - Natural Language Processing

- Dr. Cristiano De Nobili
  - Theoretical Physicist
  - Lead AI Scientist @ Pi School
  - Speaker

SCHEDULE

- **LECTURES**
  - Friday, 08/07 (14:00 – 18:00), Building U14, Room T023
  - Wednesday 13/07 (14:00– 18:00), Building U14, Room T023
  - Monday 18/07 (14:00– 18:00), Building U14, Room T024
  - Tuesday 28/07 (10:00 – 12:00, 14:00 – 16:00), Building U14, Room T023
  - Friday 29/07 (10:00 – 12:00, 14:00 – 16:00), Building U24, Room C01

- **FINAL EXAM**
  - Friday 30/09 (room and time schedule to be announced)

Final exam

- 15 minutes presentation related to:
  - An in depth analysis of a **theoretical** paper

What is Deep Learning?

Artificial Intelligence

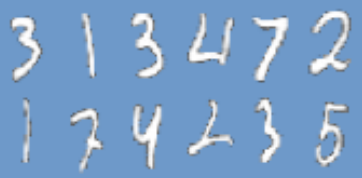
Any technique that enables computers to mimic human behavior

Machine Learning

Ability to learn without explicitly being programmed

Deep Learning

Learn underlying features in data using neural networks



Why Now?

1952 Stochastic Gradient Descent

1958 Perceptron
- Learnable Weights

...

1986 Backpropagation
- Multi-Layer Perceptron

1995 Deep Convolutional NN
- Digit Recognition

...

Neural Networks date back decades, so why the resurgence?

1. Big Data

- Larger Datasets
- Easier Collection & Storage

IMAGENET

WIKIPEDIA
The Free Encyclopedia

2. Hardware

- Graphics Processing Units (GPUs)
- Massively Parallelizable

3. Software

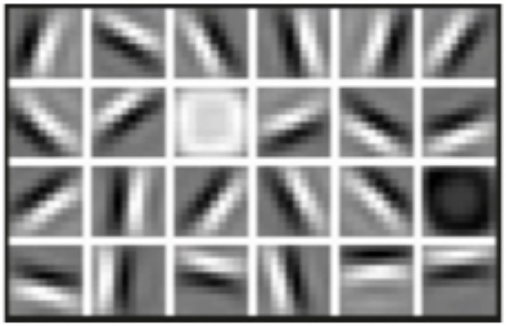
- Improved Techniques
- New Models
- Toolboxes

TensorFlow

Why Deep Learning?

- Hand engineered features are time consuming, brittle and not scalable in practice
  - Can we learn the **underlying features** directly from data?

Low Level Features



Lines & Edges

Mid Level Features

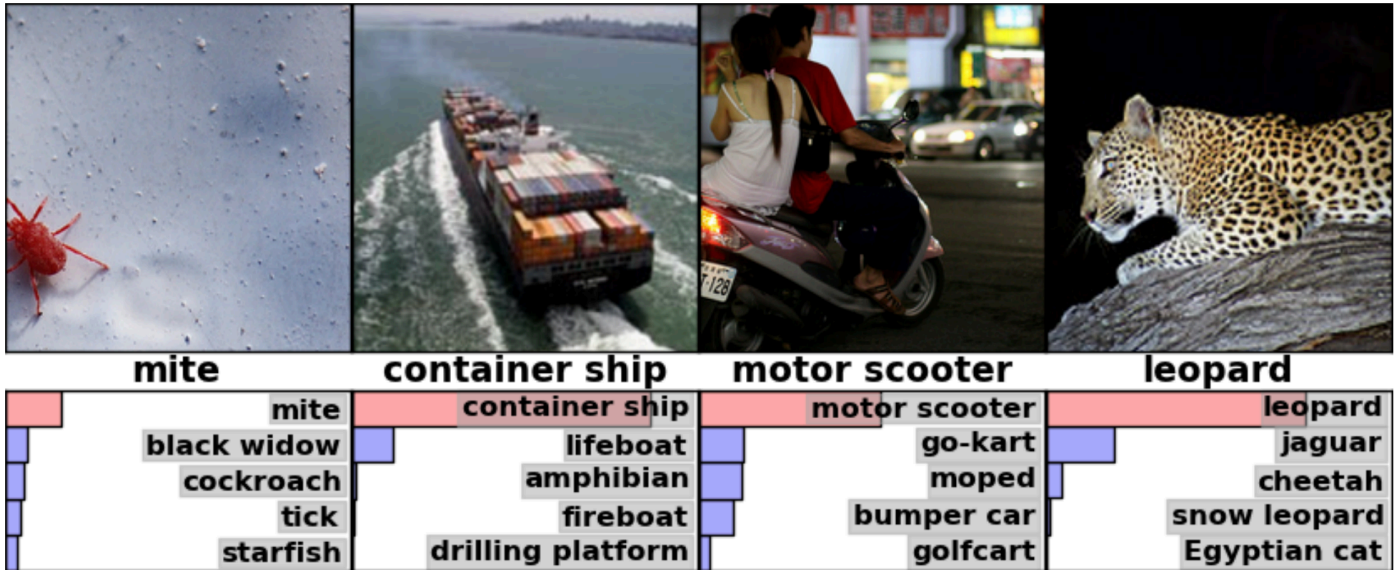
Eyes & Nose & Ears

High Level Features

Facial Structure

Domains of interest for Deep Learning

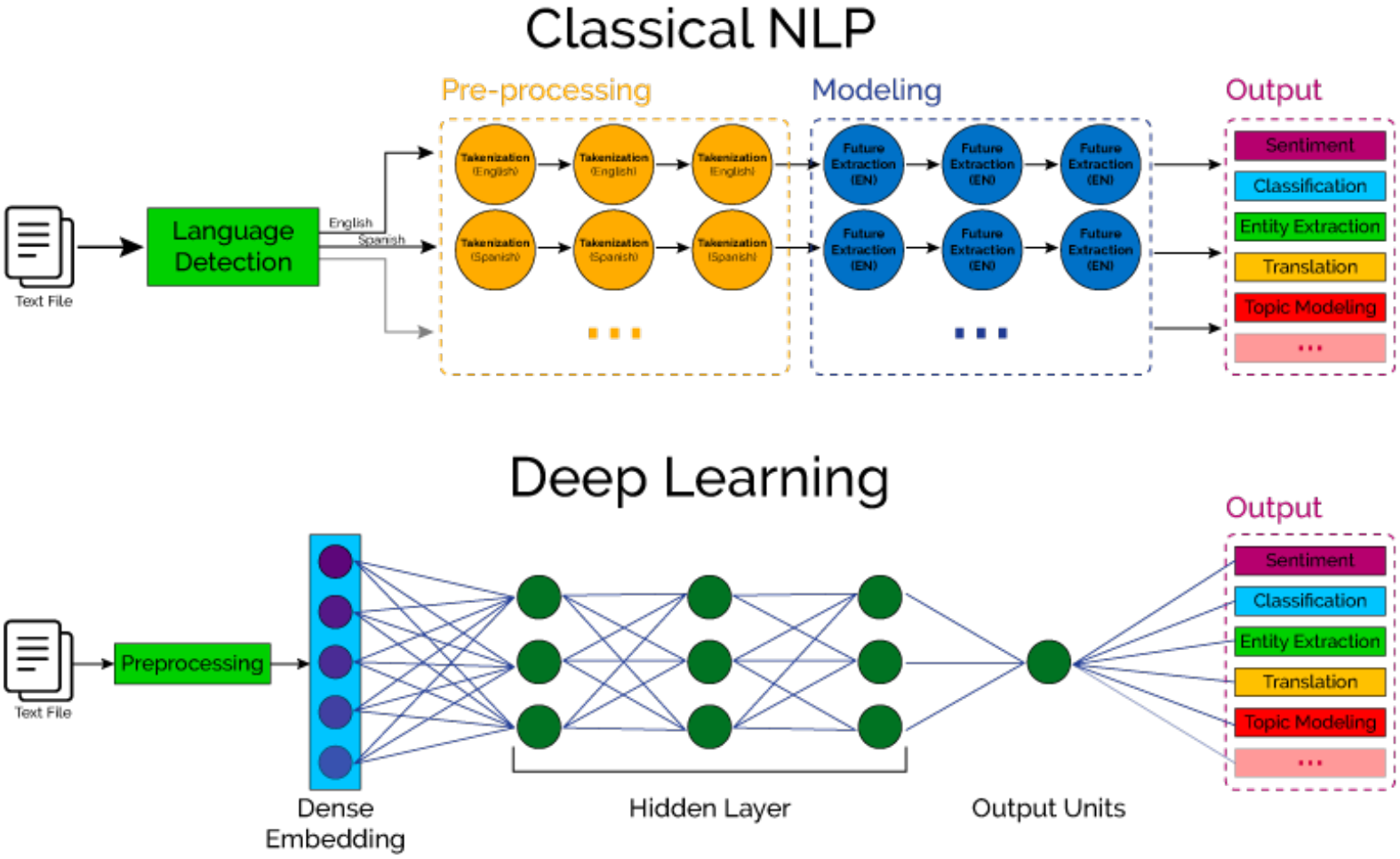
Image Recognition

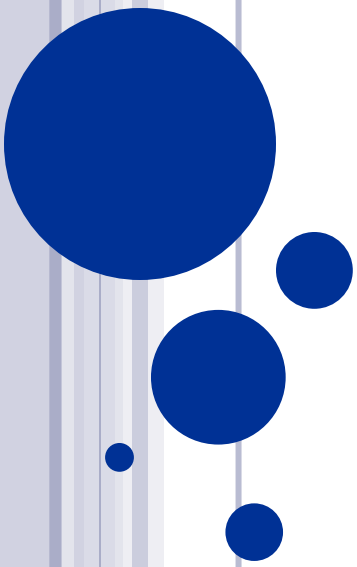


Domains of interest for Deep Learning

Music Generation

Domains of interest for Deep Learning





Background concepts

Introduction

- Deep Learning is grounded on (Artificial) Neural Networks.
- Neural Networs are a class of supervised machine learning algorithms
- We will have a quick introduction to:
  - supervised machine learning terminology and practices
  - linear and log-linear models for binary and multi-class classification

SUPERVISED LEARNING AND PARAMETERIZED FUNCTIONS

- The essence of supervised machine learning is the creation of *functions* that can look at examples (instances) and produce generalizations (good prediction on unseen data).

- As searching over the set of all possible functions is a very hard problem, we often restrict ourselves to search over specific families of functions called *hypothesis classes*

SUPERVISED LEARNING AND PARAMETERIZED FUNCTIONS

- A common hypothesis class is represented by high-dimensional linear function, i.e. functions of the form:

$$f(\boldsymbol{x}) = \boldsymbol{x} \cdot \boldsymbol{W} + \boldsymbol{b}$$

$$\boldsymbol{x} \in \mathbb{R}^{d_{in}} \quad \boldsymbol{W} \in \mathbb{R}^{d_{in} \times d_{out}} \quad \boldsymbol{b} \in \mathbb{R}^{d_{out}}.$$

input vector

Parameters Θ

- The goal of the learner is to find the values of the parameters W and b such that the function behaves as intended on a collection of input instances $x_1, \ldots, x_k$ and the corresponding output labels $y_1, \ldots, y_k$.

Linear models

- Binary classification:

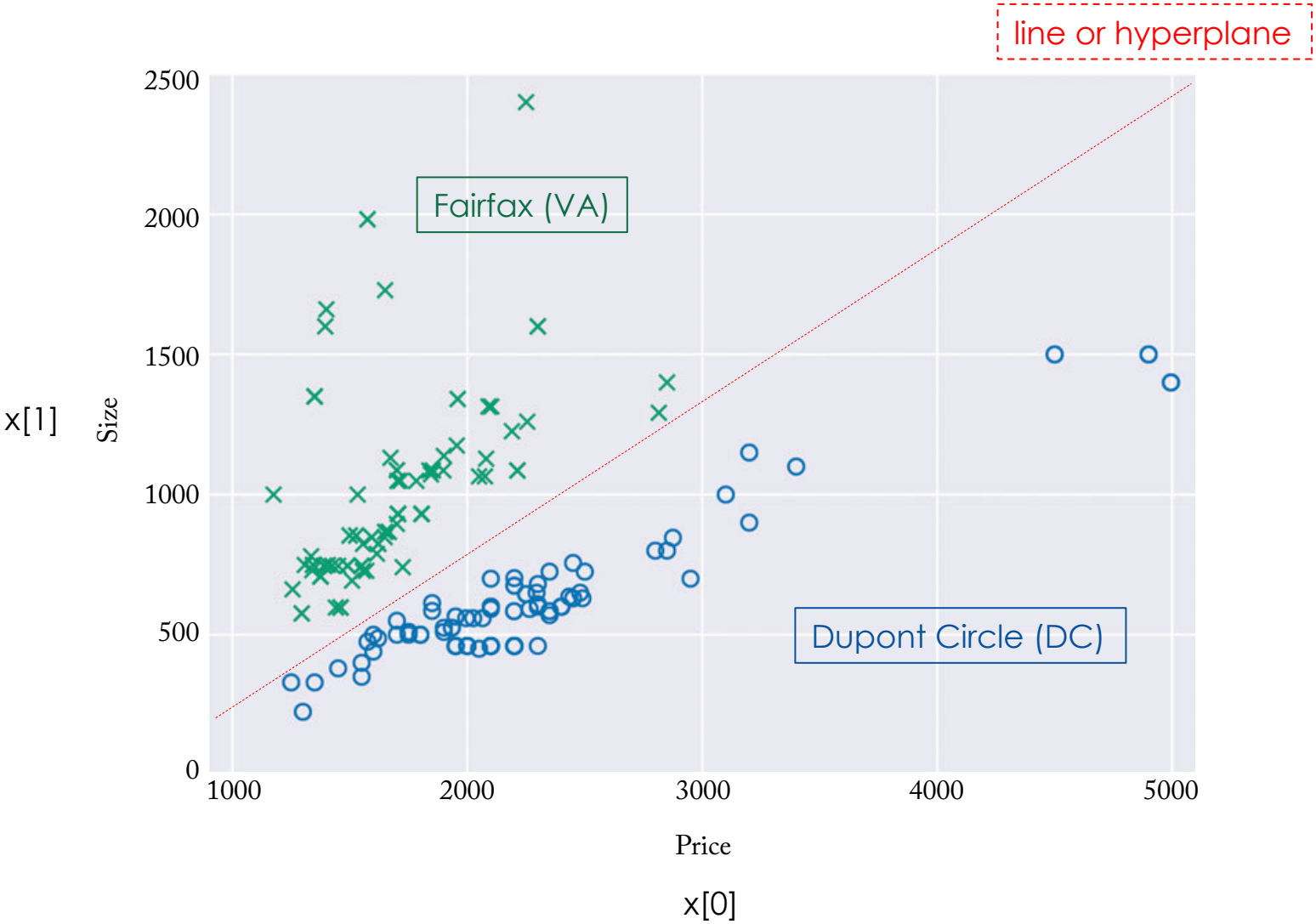
$$f(\boldsymbol{x}) = \boldsymbol{x} \cdot \boldsymbol{w} + b$$

input vector — weight vector — scalar

- The range of the linear function is [-∞; +∞]
  - We need to map the in order to have only values +1 and -1
    - *sign function*

$$\hat{y} = \text{sign}(f(\boldsymbol{x})) = \text{sign}(\boldsymbol{x} \cdot \boldsymbol{w} + b)$$

LINEAR MODELS



Linear models

- In our example, the linear model is as follows:

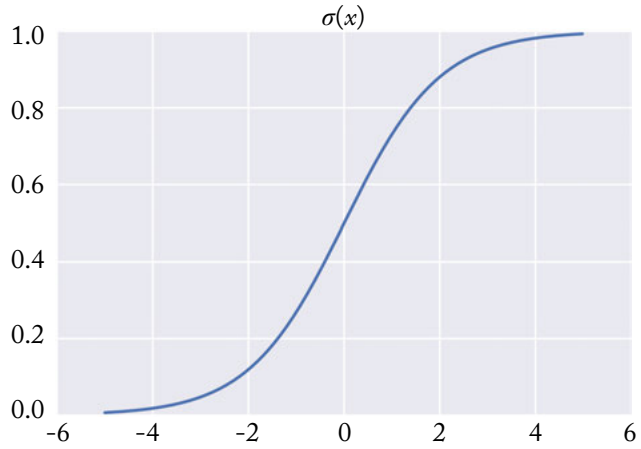
$$\hat{y} = \text{sign}(f(\boldsymbol{x})) = \text{sign}(\boldsymbol{x} \cdot \boldsymbol{w} + b)$$
$$= \text{sign}(\text{size} \times w_1 + \text{price} \times w_2 + b)$$

- where the prediction is Fairfax if $\hat{y} \geq 0$ and Dupont Circle if $\hat{y} \leq 0$.

- The goal of learning is setting the values of w_1, w_2 and b such that all $\hat{y}$ are correct for all data points we observe.
  - datasets are not always linearly separable, and therefore they are beyond the hypothesis class of linear classifiers
    - Possible solutions: move to a higher dimension, move to a richer hypothesis class, etc..

LINEAR MODELS

- Log-Linear Binary Classification: the need of a confidence of the decision
  - Instead of mapping to {-1,+1}, we map to the range [0,1] by using a squashing function such as the *sigmoid function*



  - Our prediction becomes

$$\hat{y} = \sigma(f(\boldsymbol{x})) = \frac{1}{1 + e^{-(\boldsymbol{x}\cdot\boldsymbol{w}+b)}}$$

LINEAR MODELS

- Multi-class classification:
  - we want to predict an instance as belonging to one of *k* different classes

- A possible solution is to consider *k* weight vectors w^1, w^2,…w^k and biases
  - And predict the class with the highest score:

$$\hat{y} = f(x) = \underset{k \in K}{\operatorname{argmax}} \; \boldsymbol{x} \cdot \boldsymbol{w}^k + b^k$$

  - The sets of parameters $w^k \in R^{in}$ and b^k can be arranged as a matrix $W \in R^{in \times k}$ and vector $b \in R^k$, leading to:

$$\hat{\boldsymbol{y}} = f(\boldsymbol{x}) = \boldsymbol{x} \cdot \boldsymbol{W} + \boldsymbol{b}$$

$$\text{prediction} = \hat{y} = \underset{k}{\operatorname{argmax}} \, \hat{\boldsymbol{y}}_{[k]}$$

    - where $\hat{y} \in R^k$ is a vector of the scores assigned by the model to each class

LINEAR MODELS

- Log-linear multi-class classification: transform the score vector into a probability estimate through the *softmax function*

- Our prediction becomes:

$$\hat{\boldsymbol{y}} = \text{softmax}(\boldsymbol{x}\boldsymbol{W} + \boldsymbol{b})$$

$$\hat{\boldsymbol{y}}_{[k]} = \frac{e^{(\boldsymbol{x}\boldsymbol{W}+\boldsymbol{b})_{[k]}}}{\sum_j e^{(\boldsymbol{x}\boldsymbol{W}+\boldsymbol{b})_{[j]}}}.$$

  - forcing the values in $\hat{y}$ to be positive and sum to 1, making them interpretable as a probability distribution.

Training as optimization

- The goal of a training algorithm is to return a function f() that accurately maps the input vectors to the corresponding labels
  - i.e. a function f() such that $\hat{y} = f(x)$ over the training data is accurate

- In order to measure how a function f() is accurate, we need to introduce the concept of loss function
  - Formally, a loss function $L(\hat{y}, y)$ assign a numerical score to a predicted output $\hat{y}$ given the true expected output y
  - The parameters of the learned function (W and b) are then set to minimize the loss *L* over the training samples

Training as optimization

- Given a labelled training set $(x_{1:n}, y_{1:n})$, a per-instance loss function $\mathcal{L}$ and a parameterized function $f(x;\Theta)$, we define the global loss as follows:

$$\mathcal{L}(\Theta) = \frac{1}{n}\sum_{i=1}^{n} L(f(\boldsymbol{x}_i;\Theta), \boldsymbol{y}_i)$$

- The main goal of the training algorithm is to set the values of the parameters Θ such that the value of $\mathcal{L}$ is minimized:

$$\hat{\Theta} = \underset{\Theta}{\operatorname{argmin}}\, \mathcal{L}(\Theta) = \underset{\Theta}{\operatorname{argmin}}\, \frac{1}{n}\sum_{i=1}^{n} L(f(\boldsymbol{x}_i;\Theta), \boldsymbol{y}_i)$$

  - Here we are minimizing the loss at all costs, which may result in overfitting

Training as optimization

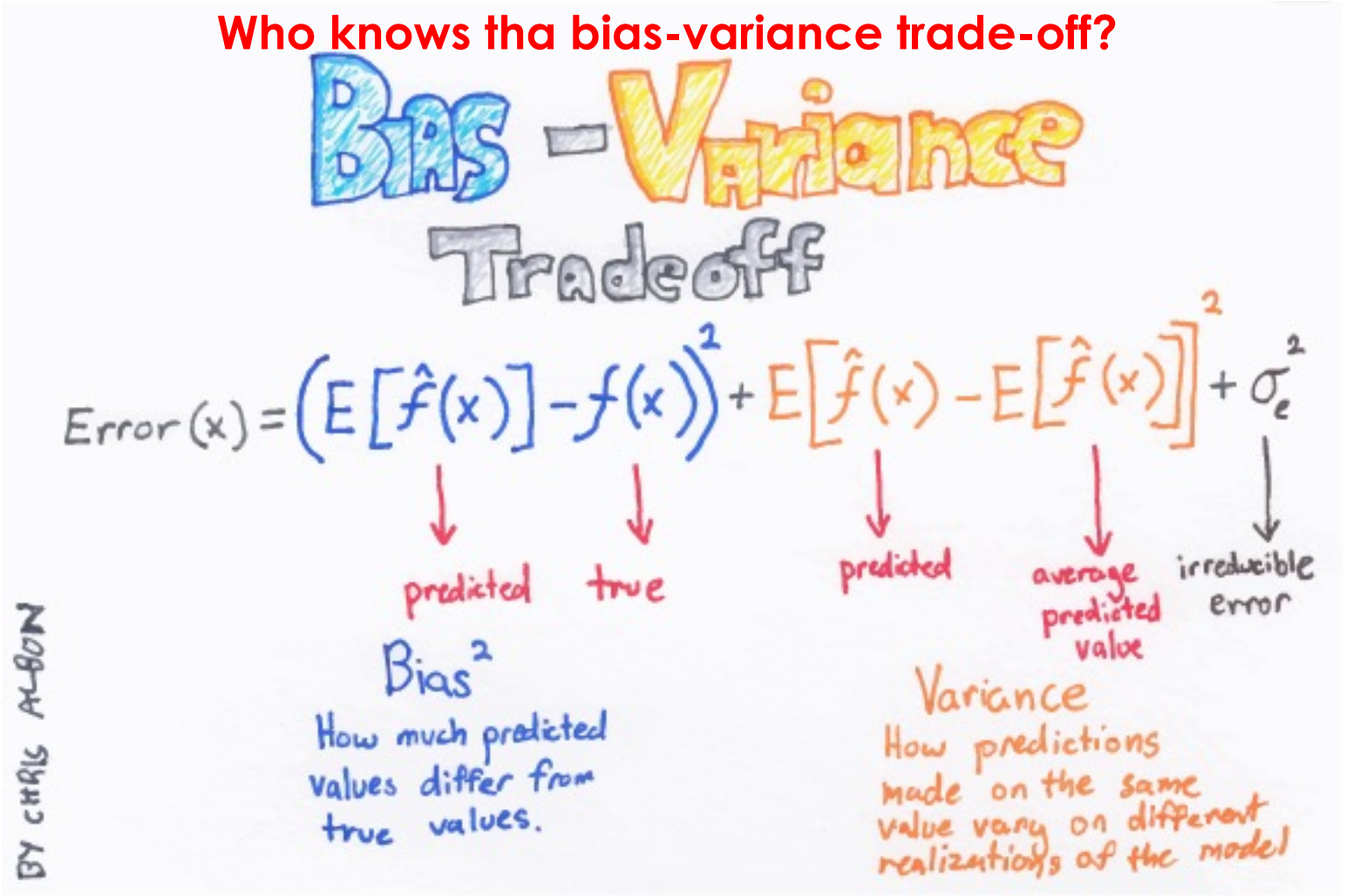
- To overcome this issue, we could pose a soft restriction on the form of the solution through a regularization function:

$$\hat{\Theta} = \underset{\Theta}{\operatorname{argmin}} \left(\overbrace{\frac{1}{n}\sum_{i=1}^{n} L(f(\boldsymbol{x}_i;\Theta), \boldsymbol{y}_i)}^{loss} + \overbrace{\lambda R(\Theta)}^{regularization} \right)$$

  - Where:
    - R(Θ) corresponds to a scalar that reflects the parameter "complexity" (that we want to keep low for efficicy reasons)
    - Λ is an hyperparameter that control the trade-off between bias and variance

BIAS-VARIANCE TRADE-OFF

Who knows tha bias-variance trade-off?



BIAS-VARIANCE TRADE-OFF: MATHEMATICALLY

- Let the variable we are trying to predict as Y and other covariates as X.

$$Y=f(X) + e$$

- The expected squared error is

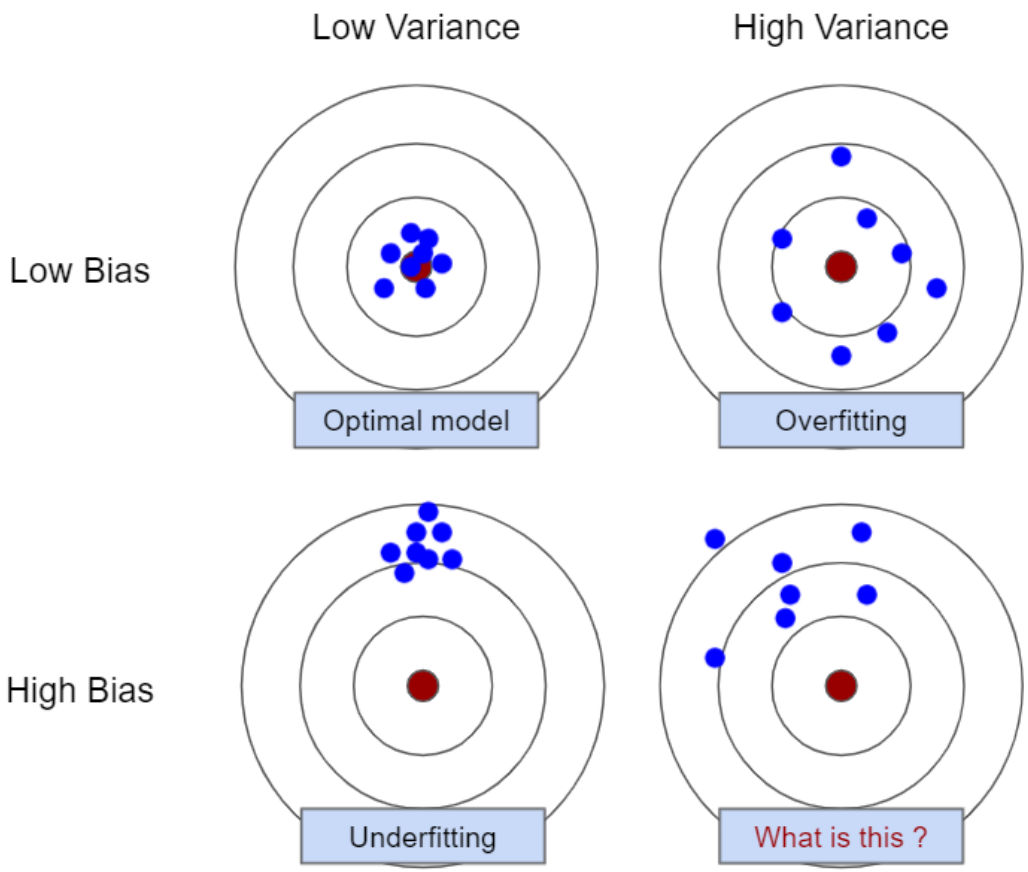
$$Err(x) = E\left[(Y - \hat{f}\,(x))^2\right]$$

- Which can be further decomposed as

$$Err(x) = \left(E[\hat{f}\,(x)] - f(x)\right)^2 + E\left[\left(\hat{f}\,(x) - E[\hat{f}\,(x)]\right)^2\right] + \sigma_e^2$$

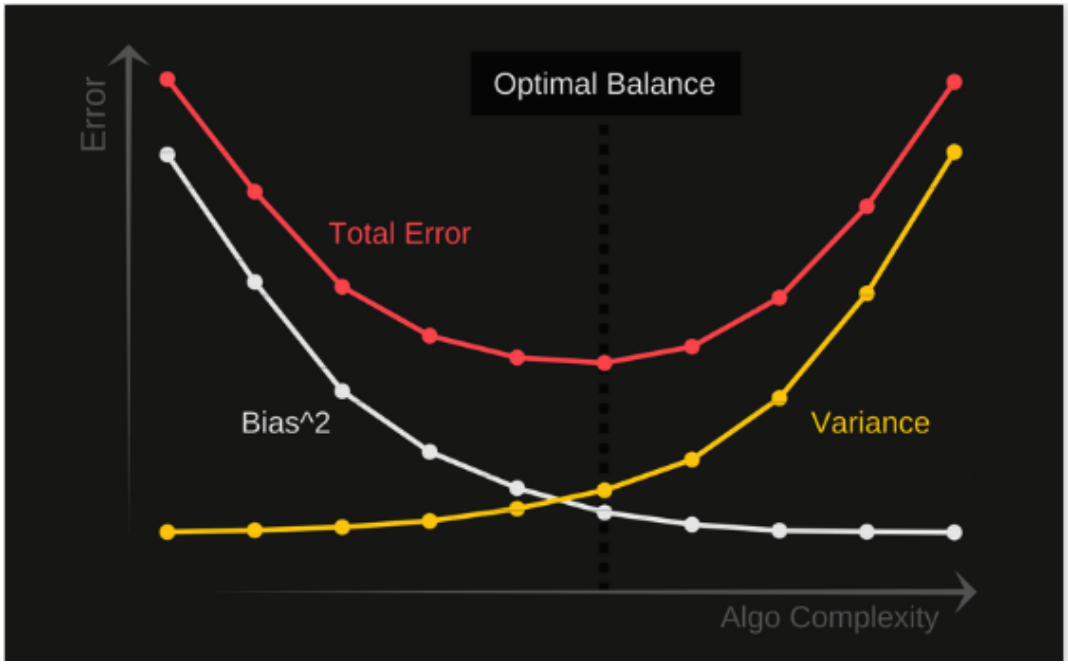
$$Err(x) = \text{Bias}^2 + \text{Variance} + \text{Irreducible Error}$$

Bias-Variance trade-off: bulls-eye diagram

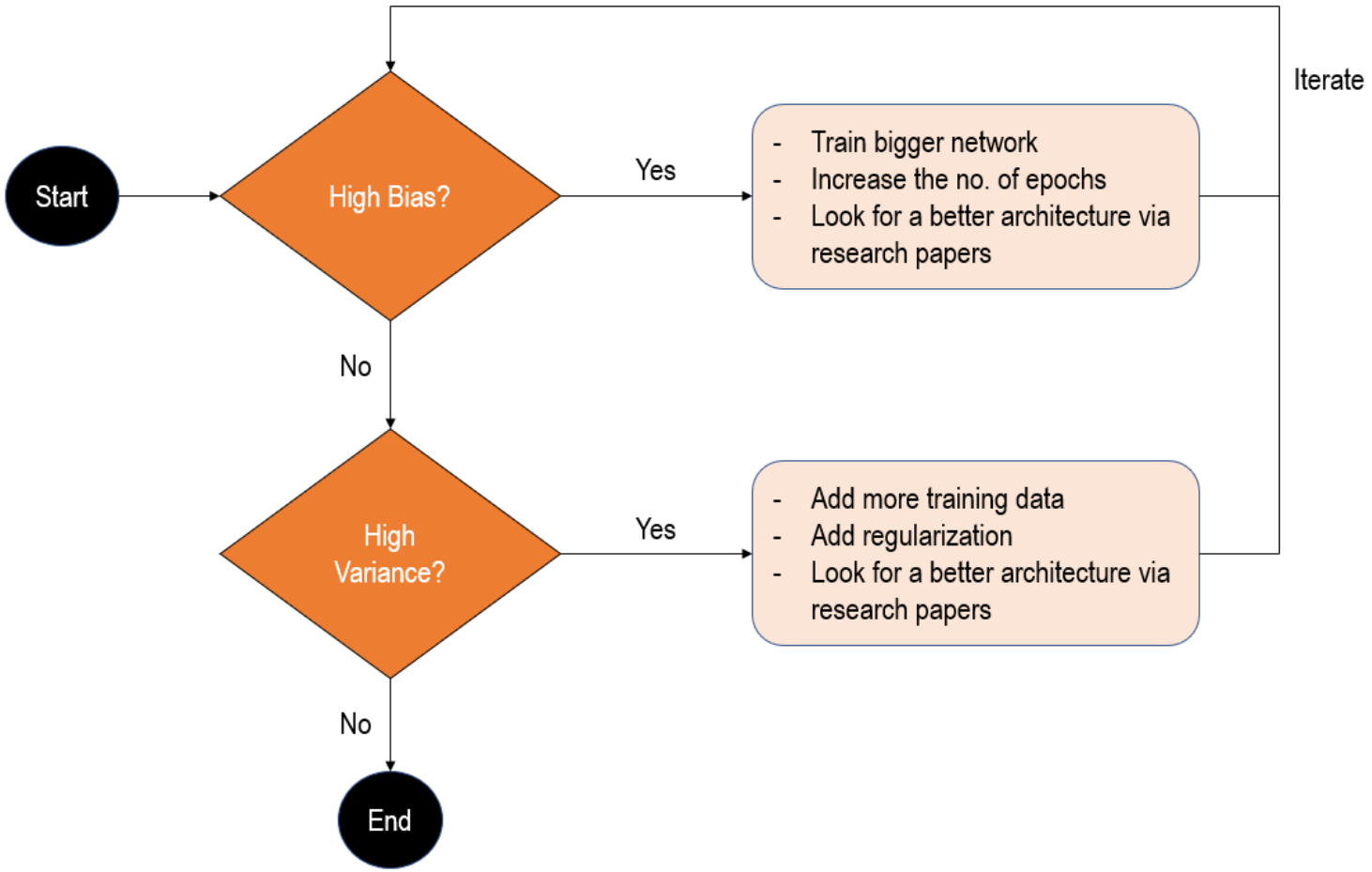


Why is Bias Variance Tradeoff?

- Tradeoff in complexity is why there is a tradeoff between bias and variance.



GOOD TRADE-OFF: TIPS

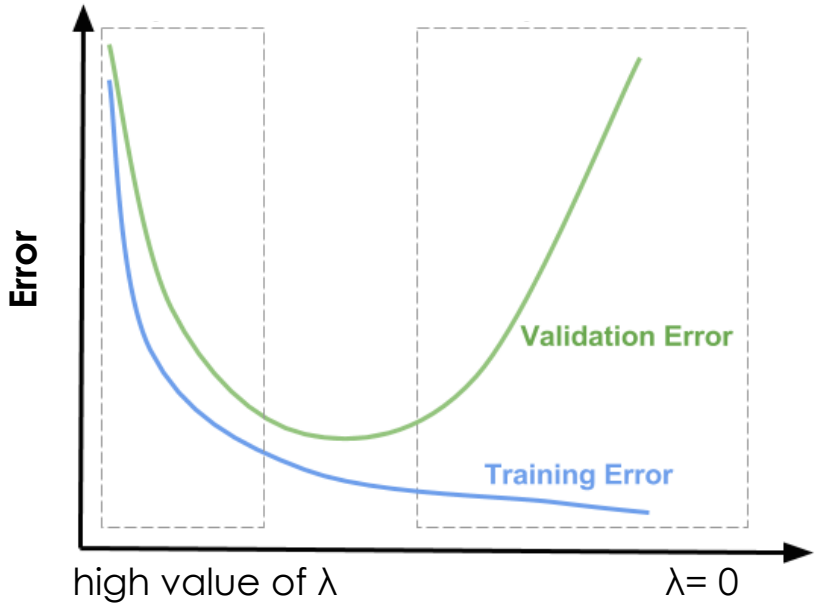


READING

Yang, Z., Yu, Y., You, C., Steinhardt, J., & Ma, Y. (2020, November). **Rethinking bias-variance trade-off for generalization of neural networks**. In *International Conference on Machine Learning* (pp. 10767-10777).

If we go back to the regularization

- The optimal value for λ will be probably somewhere around the minimum of the Cross Validation loss function



Nielsen, M. A. Neural networks and deep learning. Vol. 2018. San Francisco, CA, USA: Determination.
http://neuralnetworksanddeeplearning.com

Loss functions

- The loss can be an arbitrary function mapping two vectors to a scalar

- HINGE (binary) LOSS:
  - the classifier's output is a single scalar $\tilde{y}$ and the true label is in {+1,-1};
  - the classification rule is $\hat{y} = sign(\tilde{y})$ and a prediction is correct if

$$y \cdot \tilde{y} \geq 0 \qquad \rightarrow \quad \text{they share they same } sign$$

  - The Hinge Loss is defined as:

$$L_{\text{hinge(binary)}}(\tilde{y}, y) = \max(0, 1 - y \cdot \tilde{y})$$

LOSS FUNCTIONS

- HINGE (multi-class) LOSS:
  - Let $\hat{y} = \hat{y}_{[1]},\ldots,\hat{y}_{[k]}$ be the classifier's output vector, and y be the one-hot vector for the true label.
  - The classification rule is defined as selecting the class with the highest score

$$\text{prediction} = \underset{i}{\text{argmax}}\, \hat{\boldsymbol{y}}_{[i]}$$

  - The Hinge Loss is defined as:

$$L_{\text{hinge(multi-class)}}(\hat{\boldsymbol{y}}, \boldsymbol{y}) = \max(0, 1 - (\hat{\boldsymbol{y}}_{[t]} - \hat{\boldsymbol{y}}_{[j]}))$$

TIPS: In general HINGE LOSSES are useful when we require a hard decision rule, do not considering the class membership probability.

LOSS FUNCTIONS

- LOGISTIC LOSS (BINARY CROSS ENTROPY):
  - It is used in binary classification with conditional probability outputs
  - We assume a set of two target classes labelled 0 and 1, with correct label $y \in \{0,1\}$
  - The classifier's output $\tilde{y}$ is transfomed using the *sigmoid* (aka logistic) function $\sigma(x) = 1/(1+e^{-x})$ to the range [0,1]
    - It is intepreted as the conditional probability $\hat{y} = \sigma(\tilde{y}) = P(y=1|\boldsymbol{x})$
  - The prediction rule is:

$$\text{prediction} = \begin{cases} 0 & \hat{y} < 0.5 \\ 1 & \hat{y} \geq 0.5 \end{cases}$$

  - The training algorithm tries to maximize the log conditional probability *P(y=1 | x)* for each training example (x,y):

$$L_{\text{logistic}}(\hat{y}, y) = -y \log \hat{y} - (1-y) \log(1-\hat{y})$$

LOSS FUNCTIONS

- NEGATIVE LOG LIKELIHOOD (CATEGORICAL CROSS-ENTROPY LOSS):
  - It is used when a probabilistic distribution of scores is desired
  - Let $\boldsymbol{y} = \boldsymbol{y}_{[1]}, \ldots, \boldsymbol{y}_{[k]}$ be a vector denoting the true multinomial distribution over the labels *1,…,k*
  - Let $\hat{\boldsymbol{y}} = \hat{\boldsymbol{y}}_{[1]}, \ldots, \hat{\boldsymbol{y}}_{[k]}$ be the linear classifier's predictions (transformed by the softmax function) that represent the class memebership conditional probability distribution $\hat{y}_{[i]} = P(y = i \mid x)$
  - The negative log likelihood loss measures the dissimilarity between the true label distribution and the predicted one:

$$L_{\text{negative log likelihood}} = (\hat{\boldsymbol{y}}, \boldsymbol{y}) = -\sum_{i} \boldsymbol{y}_{[i]} \log(\hat{\boldsymbol{y}}_{[i]})$$

$$L_{\text{cross-entropy(hard classification)}}(\hat{\boldsymbol{y}}, \boldsymbol{y}) = -\log(\hat{\boldsymbol{y}}_{[t]}),$$

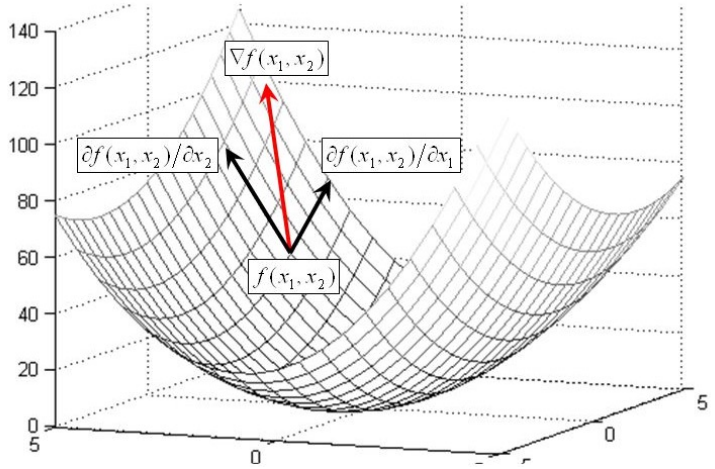
Gradient-based Optimization

- In order to train the model, we need to solve the optimization problem:

$$\hat{\Theta} = \underset{\Theta}{\operatorname{argmin}}\, \mathcal{L}(\Theta) + \lambda R(\Theta)$$

$$= \underset{\Theta}{\operatorname{argmin}}\, \frac{1}{n} \sum_{i=1}^{n} L(f(\boldsymbol{x}_i; \Theta), \boldsymbol{y}_i) + \lambda R(\Theta)$$

- Solution: gradient-based methods
  - Estimates the loss $\mathcal{L}$ over the training set
  - Computes the gradients Θ w.r.t. the loss $\mathcal{L}$
  - Moves the parameters directions the gradient



  - When optimizing the loss $\mathcal{L}$, the parameters are considered as input, while the training examples are treated as constants

Stochastic gradient descent

- Stochastic Gradient Descent (SGD)
  - It is a general optimization algorithm
  - It receives a function *f* parameterized by **Θ**, a loss function $\mathcal{L}$, input-output pairs $x_{1:n}, y_{1:n}$
  - The main goal is to set the parameters **Θ** such that the cumulative loss of *f* on the training examples is small

$$\mathcal{L}(\Theta) = \sum_{i=1}^{n} L(f(\boldsymbol{x_i}; \theta), \boldsymbol{y_i})$$

STOCHASTIC GRADIENT DESCENT

Algorithm 2.1 Online stochastic gradient descent training.

Input:
- Function $f(\boldsymbol{x}; \Theta)$ parameterized with parameters Θ.
- Training set of inputs $\boldsymbol{x_1}, \ldots, \boldsymbol{x_n}$ and desired outputs $\boldsymbol{y_1}, \ldots, \boldsymbol{y_n}$.
- Loss function L.

1: **while** stopping criteria not met **do**
2: Sample a training example $\boldsymbol{x_i}, \boldsymbol{y_i}$
3: Compute the loss $L(f(\boldsymbol{x_i}; \Theta), \boldsymbol{y_i})$
4: $\hat{\boldsymbol{g}} \leftarrow$ gradients of $L(f(\boldsymbol{x_i}; \Theta), \boldsymbol{y_i})$ w.r.t Θ
5: $\Theta \leftarrow \Theta - \eta_t \hat{\boldsymbol{g}}$
6: **return** Θ

It starts by sampling a training example

STOCHASTIC GRADIENT DESCENT

Algorithm 2.1 Online stochastic gradient descent training.

Input:
- Function $f(\boldsymbol{x}; \Theta)$ parameterized with parameters Θ.
- Training set of inputs $\boldsymbol{x_1}, \ldots, \boldsymbol{x_n}$ and desired outputs $\boldsymbol{y_1}, \ldots, \boldsymbol{y_n}$.
- Loss function L.

1: **while** stopping criteria not met **do**
2: Sample a training example $\boldsymbol{x_i}, \boldsymbol{y_i}$
3: Compute the loss $L(f(\boldsymbol{x_i}; \Theta), \boldsymbol{y_i})$
4: $\hat{\boldsymbol{g}} \leftarrow$ gradients of $L(f(\boldsymbol{x_i}; \Theta), \boldsymbol{y_i})$ w.r.t Θ
5: $\Theta \leftarrow \Theta - \eta_t \hat{\boldsymbol{g}}$
6: **return** Θ

It computes the loss

STOCHASTIC GRADIENT DESCENT

Algorithm 2.1 Online stochastic gradient descent training.

Input:
- Function $f(\boldsymbol{x}; \Theta)$ parameterized with parameters Θ.
- Training set of inputs $\boldsymbol{x_1}, \ldots, \boldsymbol{x_n}$ and desired outputs $\boldsymbol{y_1}, \ldots, \boldsymbol{y_n}$.
- Loss function L.

1: **while** stopping criteria not met **do**
2: Sample a training example $\boldsymbol{x_i}, \boldsymbol{y_i}$
3: Compute the loss $L(f(\boldsymbol{x_i}; \Theta), \boldsymbol{y_i})$
4: $\hat{\boldsymbol{g}} \leftarrow$ gradients of $L(f(\boldsymbol{x_i}; \Theta), \boldsymbol{y_i})$ w.r.t Θ
5: $\Theta \leftarrow \Theta - \eta_t \hat{\boldsymbol{g}}$
6: **return** Θ

It computes the gradient w.r.t Θ

STOCHASTIC GRADIENT DESCENT

Algorithm 2.1 Online stochastic gradient descent training.

Input:
- Function $f(\boldsymbol{x}; \Theta)$ parameterized with parameters Θ.
- Training set of inputs $\boldsymbol{x_1}, \ldots, \boldsymbol{x_n}$ and desired outputs $\boldsymbol{y_1}, \ldots, \boldsymbol{y_n}$.
- Loss function L.

1: **while** stopping criteria not met **do**
2: Sample a training example $\boldsymbol{x_i}, \boldsymbol{y_i}$
3: Compute the loss $L(f(\boldsymbol{x_i}; \Theta), \boldsymbol{y_i})$
4: $\hat{\boldsymbol{g}} \leftarrow$ gradients of $L(f(\boldsymbol{x_i}; \Theta), \boldsymbol{y_i})$ w.r.t Θ
5: $\Theta \leftarrow \Theta - \eta_t \hat{\boldsymbol{g}}$
6: **return** Θ

It computes the gradient w.r.t Θ

Input and preditions are assumed to be fixed and the loss is treated as a function of Θ

STOCHASTIC GRADIENT DESCENT

Algorithm 2.1 Online stochastic gradient descent training.

Input:
- Function $f(\boldsymbol{x}; \Theta)$ parameterized with parameters Θ.
- Training set of inputs $\boldsymbol{x_1}, \ldots, \boldsymbol{x_n}$ and desired outputs $\boldsymbol{y_1}, \ldots, \boldsymbol{y_n}$.
- Loss function L.

1: **while** stopping criteria not met **do**
2: Sample a training example $\boldsymbol{x_i}, \boldsymbol{y_i}$
3: Compute the loss $L(f(\boldsymbol{x_i}; \Theta), \boldsymbol{y_i})$
4: $\hat{\boldsymbol{g}} \leftarrow$ gradients of $L(f(\boldsymbol{x_i}; \Theta), \boldsymbol{y_i})$ w.r.t Θ
5: $\Theta \leftarrow \Theta - \eta_t \hat{\boldsymbol{g}}$
6: **return** Θ

Parameters Θ are updated in the opposite direction of the gradient

STOCHASTIC GRADIENT DESCENT

Algorithm 2.1 Online stochastic gradient descent training.

Input:
- Function $f(\boldsymbol{x}; \Theta)$ parameterized with parameters Θ.
- Training set of inputs $\boldsymbol{x_1}, \ldots, \boldsymbol{x_n}$ and desired outputs $\boldsymbol{y_1}, \ldots, \boldsymbol{y_n}$.
- Loss function L.

1: **while** stopping criteria not met **do**
2: Sample a training example $\boldsymbol{x_i}, \boldsymbol{y_i}$
3: Compute the loss $L(f(\boldsymbol{x_i}; \Theta), \boldsymbol{y_i})$
4: $\hat{\boldsymbol{g}} \leftarrow$ gradients of $L(f(\boldsymbol{x_i}; \Theta), \boldsymbol{y_i})$ w.r.t Θ
5: $\Theta \leftarrow \Theta - \eta_t \hat{\boldsymbol{g}}$
6: **return** Θ

Parameters Θ are updated in the opposite direction of the gradient

scaled by a learning rate η_t

STOCHASTIC GRADIENT DESCENT

Algorithm 2.1 Online stochastic gradient descent training.

Input:
- Function $f(\boldsymbol{x}; \Theta)$ parameterized with parameters Θ.
- Training set of inputs $\boldsymbol{x_1}, \ldots, \boldsymbol{x_n}$ and desired outputs $\boldsymbol{y_1}, \ldots, \boldsymbol{y_n}$.
- Loss function L.

1: **while** stopping criteria not met **do**
2: Sample a training example $\boldsymbol{x_i}, \boldsymbol{y_i}$
3: Compute the loss $L(f(\boldsymbol{x_i}; \Theta), \boldsymbol{y_i})$
4: $\hat{\boldsymbol{g}} \leftarrow$ gradients of $L(f(\boldsymbol{x_i}; \Theta), \boldsymbol{y_i})$ w.r.t Θ
5: $\Theta \leftarrow \Theta - \eta_t \hat{\boldsymbol{g}}$
6: **return** Θ

Parameters Θ are updated in the opposite direction of the gradient

scaled by a learning rate η_t

TIPS: The learning rate can be either fixed or decay as function of the time step *t*

STOCHASTIC GRADIENT DESCENT

Algorithm 2.1 Online stochastic gradient descent training.

Input:
- Function $f(\boldsymbol{x}; \Theta)$ parameterized with parameters Θ.
- Training set of inputs $\boldsymbol{x_1}, \ldots, \boldsymbol{x_n}$ and desired outputs $\boldsymbol{y_1}, \ldots, \boldsymbol{y_n}$.
- Loss function L.

1: **while** stopping criteria not met **do**
2: Sample a training example $\boldsymbol{x_i}, \boldsymbol{y_i}$
3: Compute the loss $L(f(\boldsymbol{x_i}; \Theta), \boldsymbol{y_i})$
4: $\hat{\boldsymbol{g}} \leftarrow$ gradients of $L(f(\boldsymbol{x_i}; \Theta), \boldsymbol{y_i})$ w.r.t Θ
5: $\Theta \leftarrow \Theta - \eta_t \hat{\boldsymbol{g}}$
6: **return** Θ

The error is based on a single training example

It is a rough estimation of the entire training set that we want to minimize

This may result in inaccurate gradients

STOCHASTIC GRADIENT DESCENT

- A common way to reduce this noise is to estimate the error and the gradients based on a sample of *m* examples:
  - Minibatch Stochastic Gradient Descent

Algorithm 2.2 Minibatch stochastic gradient descent training.

Input:
- Function $f(\boldsymbol{x}; \Theta)$ parameterized with parameters Θ.
- Training set of inputs $\boldsymbol{x_1}, \ldots, \boldsymbol{x_n}$ and desired outputs $\boldsymbol{y_1}, \ldots, \boldsymbol{y_n}$.
- Loss function L.

1: **while** stopping criteria not met **do**
2: Sample a minibatch of m examples $\{(\boldsymbol{x_1}, \boldsymbol{y_1}), \ldots, (\boldsymbol{x_m}, \boldsymbol{y_m})\}$
3: $\hat{\boldsymbol{g}} \leftarrow 0$
4: **for** $i = 1$ to m **do**
5: Compute the loss $L(f(\boldsymbol{x_i}; \Theta), \boldsymbol{y_i})$
6: $\hat{\boldsymbol{g}} \leftarrow \hat{\boldsymbol{g}} +$ gradients of $\frac{1}{m} L(f(\boldsymbol{x_i}; \Theta), \boldsymbol{y_i})$ w.r.t Θ
7: $\Theta \leftarrow \Theta - \eta_t \hat{\boldsymbol{g}}$
8: **return** Θ

Estimate of the training loss based *on the minibatch*

STOCHASTIC GRADIENT DESCENT

- A common way to reduce this noise is to estimate the error and the gradients based on a sample of *m* examples:
  - Minibatch Stochastic Gradient Descent

Algorithm 2.2 Minibatch stochastic gradient descent training.

Input:
- Function $f(\boldsymbol{x}; \Theta)$ parameterized with parameters Θ.
- Training set of inputs $\boldsymbol{x_1}, \ldots, \boldsymbol{x_n}$ and desired outputs $\boldsymbol{y_1}, \ldots, \boldsymbol{y_n}$.
- Loss function L.

1: **while** stopping criteria not met **do**
2: Sample a minibatch of m examples $\{(\boldsymbol{x_1}, \boldsymbol{y_1}), \ldots, (\boldsymbol{x_m}, \boldsymbol{y_m})\}$
3: $\hat{\boldsymbol{g}} \leftarrow 0$
4: **for** $i = 1$ to m **do**
5: Compute the loss $L(f(\boldsymbol{x_i}; \Theta), \boldsymbol{y_i})$
6: $\hat{\boldsymbol{g}} \leftarrow \hat{\boldsymbol{g}} +$ gradients of $\frac{1}{m} L(f(\boldsymbol{x_i}; \Theta), \boldsymbol{y_i})$ w.r.t Θ
7: $\Theta \leftarrow \Theta - \eta_t \hat{\boldsymbol{g}}$
8: **return** Θ

Final gradient estimate

Parameters are updated toward *the gradient*

STOCHASTIC GRADIENT DESCENT

- Minibatch Stochastic Gradient Descent
  - The minibatch can vary in size from *m=1* to *m=n*
    - High values of *m* provide better estimates of the training gradients
    - Small values of *m* allow more updates and faster convergence
    - It provides also efficient training:
      - for small values of m some computing architectures (e.g. GPU) allow an efficient parallel implementation of the gradient computation