Machine Learning Lecture 1

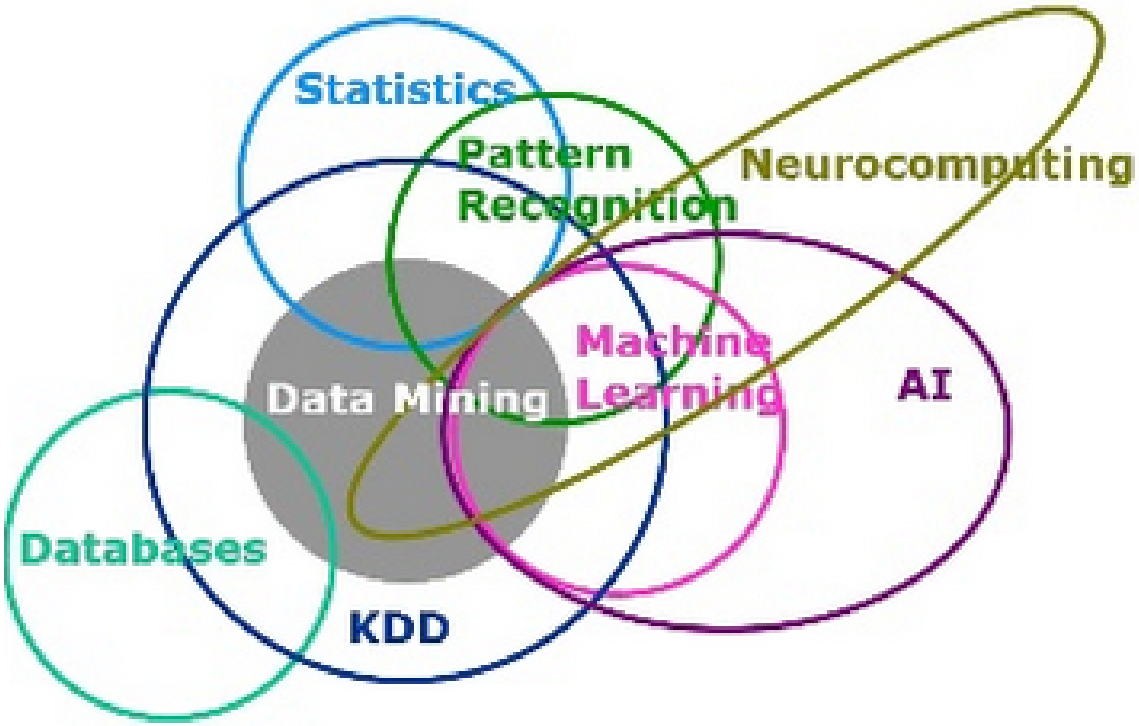
Supervised Learning: Introduction, K Nearest Neighbors

Zoltán Ádám Milacski[1]

[1]Eötvös Loránd University
Department of Software Technology and Methodology
srph25@gmail.com

April 30, 2019

MACHINE LEARNING IS...



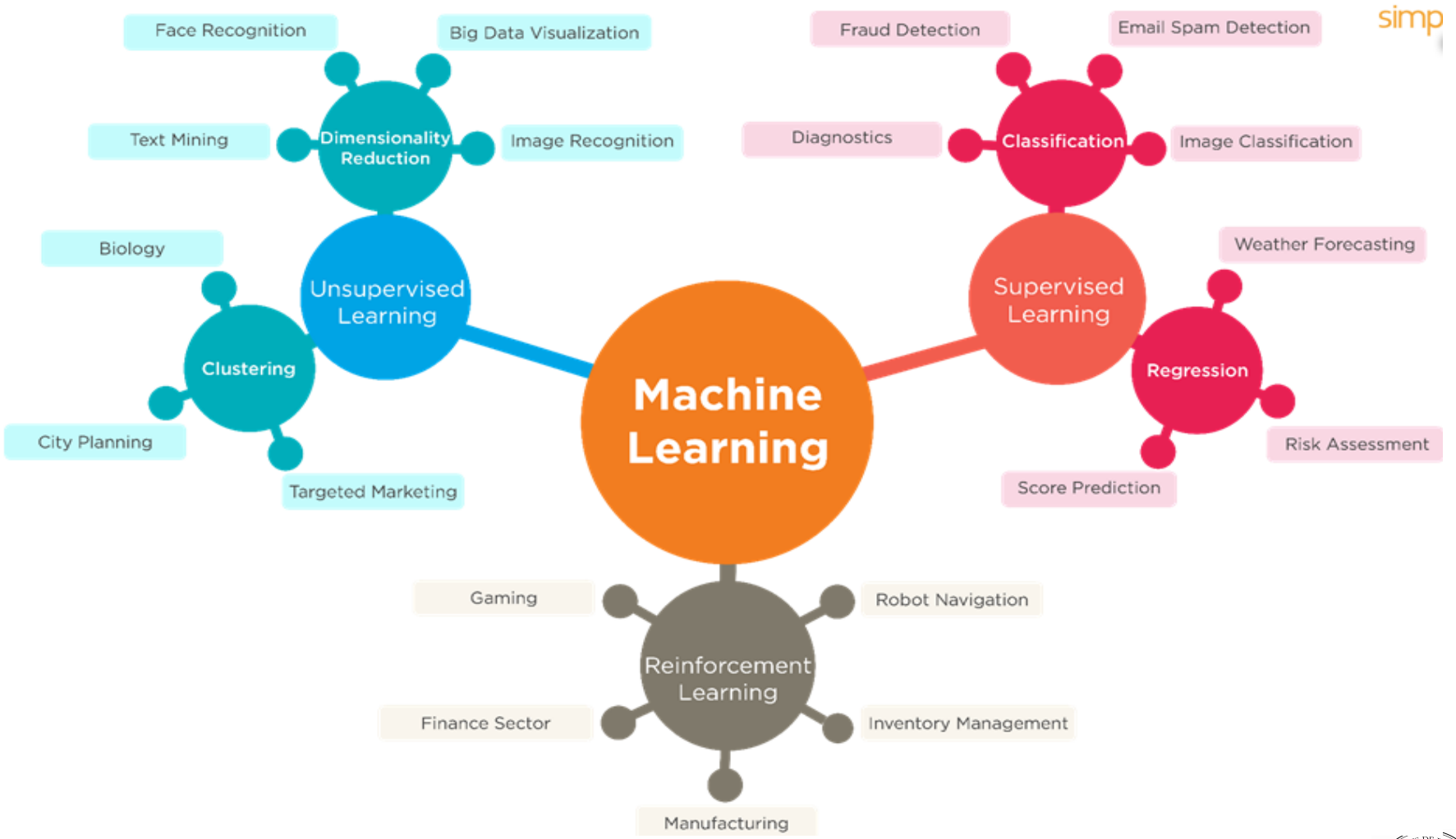
"Field of study that gives computers the ability to learn without being explicitly programmed."

– Arthur Samuel, 1959

- AI: Intelligent machines that think and act like human beings.
- ML: Systems learn things without being programmed to do so.
  Predicted next revolutionary tech (after internet, mobile) by YCombinator.

Machine Learning

What it is and what it is not



ML methods typically scale very well with the problem size (embarassingly parallel tensor operations on GPU).

Machine Learning

What it is and what it is not

- **Pure Reinforcement Learning (cherry)**
  - The machine predicts a scalar reward given once in a while.
  - **A few bits for some samples**
- **Supervised Learning (icing)**
  - The machine predicts a category or a few numbers for each input
  - **10→10,000 bits per sample**
- **Unsupervised/Predictive Learning (génoise)**
  - The machine predicts any part of its input for any observed part.
  - Predicts future frames in videos
  - **Millions of bits per sample**

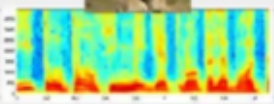
- **Unsupervised Learning is the Dark Matter (or Dark Energy) of AI**

- RL: Solved, if reward is available for training. Reproduces human knowledge.
- SL: Solved, if output is available for training. Reproduces human knowledge.
- UL: Unsolved, many open research questions. Learns without human.

Supervised Learning

Intuitively

Supervised Learning: learn the mapping from input to output, $x \mapsto y$

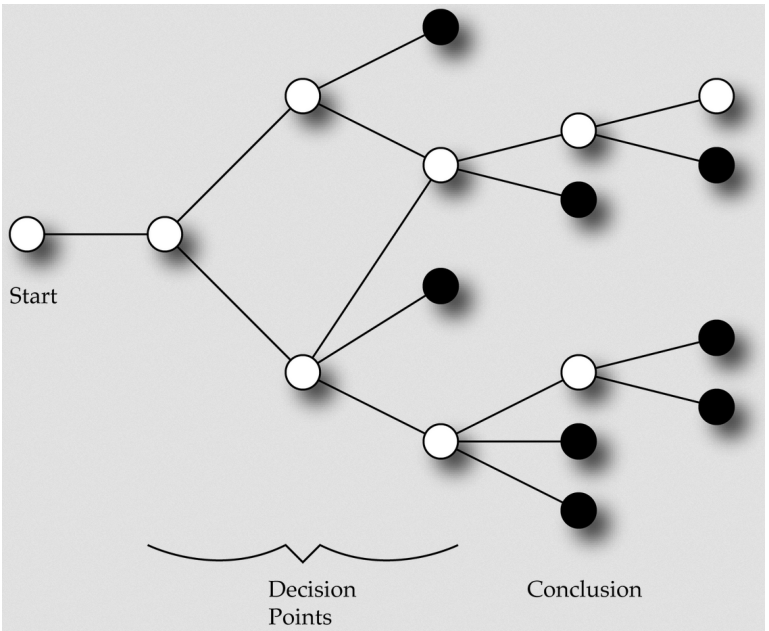
Input	Output
Pixels: 	"lion"
Audio: 	"see at tuhl res taur aun ts"
<query, doc>	P(click on doc)
"Hello, how are you?"	"Bonjour, comment allez-vous?"
Pixels: 	"A close up of a small child holding a stuffed animal"

From a finite set of sample pairs $(x_n, y_n), n = 1, \ldots, N$.
In a way that it also accurate for unseen samples pairs!

Supervised Learning

Intuitively

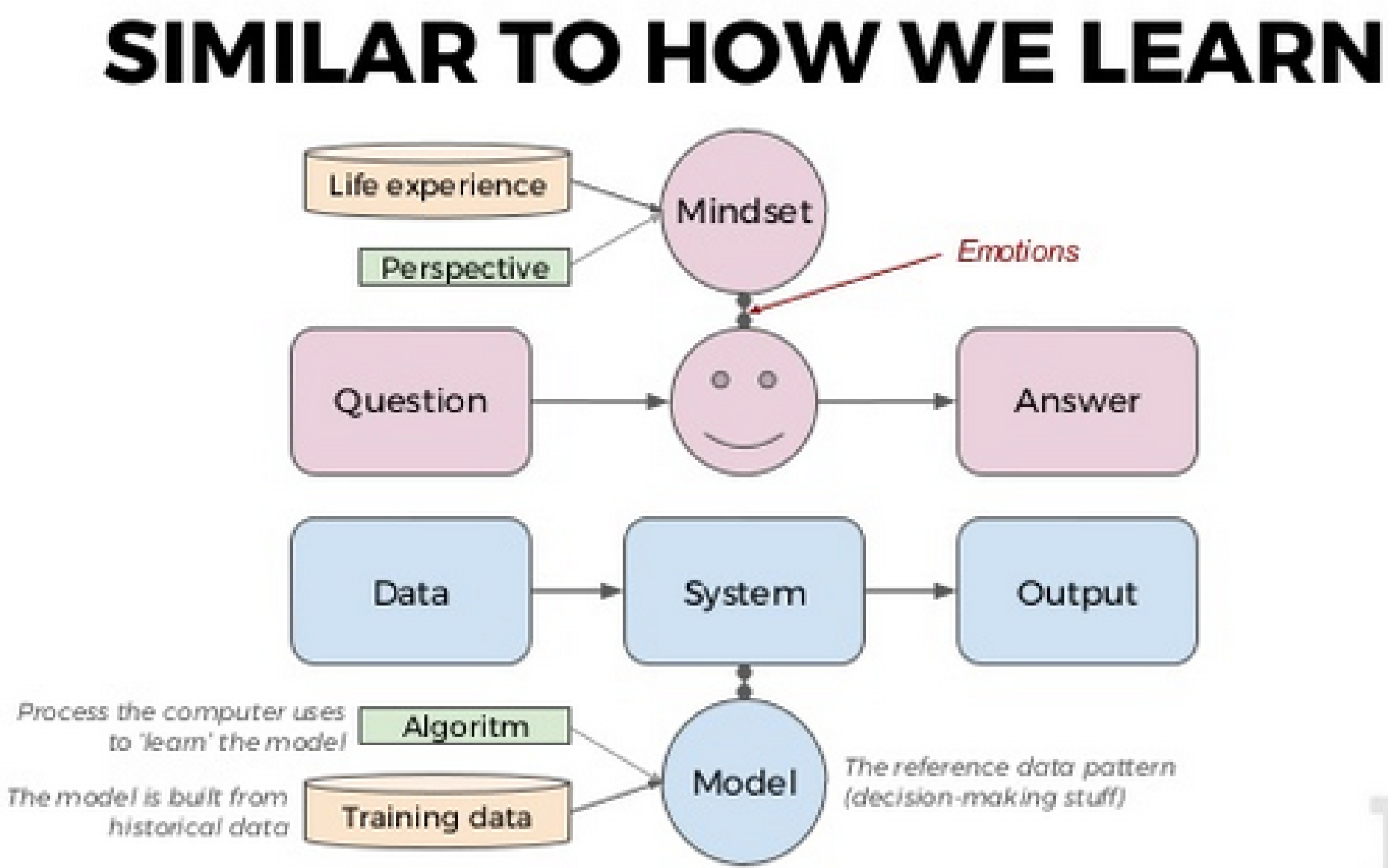
- One way to map is via branching, i.e., thresholding a feature:
  `if feature>threshold then .. else ..`
  `if` $\varphi(x) > t$ `then return` $\hat{y}_1$ `else return` $\hat{y}_2$
- Nested branching with multiple features: cases grow exponentially, hard to manually design and implement.



- Solution: learn the features $\varphi(x)$ and the thresholds t directly from data!
- Remember: ML is software that writes itself...

Supervised Learning

Intuitively



Supervised Learning

Intuitively

Algorithm: Learn Something

Input: Data, Mental Model

Output: Updated Mental Model

```
while Mental Model Makes Bad Predictions do
    make a guess at answer
    measure error
    if error is acceptably small then
        break
    else
        propose model adjustment
        Mental Model ← Mental Model + adjustment
```

Algorithm: Gradient Descent

Input: $Y, \Theta, \mathbf{X}, \alpha$, tolerance, max iterations

Output: Θ

```
for i = 0; i < max iterations; i++ do
    current cost = Cost(Y, X, Θ)
    if current cost < tolerance then
        break
    else
        gradient = Gradient(Y, X, Θ)
        θ_j ← θ_j − α · gradient
```

Supervised Learning

Formally: Optimization Problem

- Given pairs $(x_n, y_n), n = 1, \ldots, N$, find optimal parameters θ^* of parametric mapping $f(\theta, \cdot)$ such that $f(\theta^*, x_n) \approx y_n$ (approximate the mapping $x_n \mapsto y_n$):

$$\min_\theta f_0(\theta, x_n, y_n) = \frac{1}{N} \sum_{n=1}^{N} l\Big(\underbrace{f(\theta, x_n)}_{\hat{y}_n}, y_n\Big),$$

where $l(\hat{y}_n, y_n)$ is the loss function and θ is the vector of parameters.
Note: need to choose $l(\hat{y}_n, y_n)$, $f(\theta, x_n)$ and the optimization method for finding θ^*.
Inference (computing $f(\theta^*, x_n)$ given θ^*) is very fast!
Training (minimization finding θ^*) is very slow!
In general: nonconvex optimization problem, hard to find globally optimal θ^*.

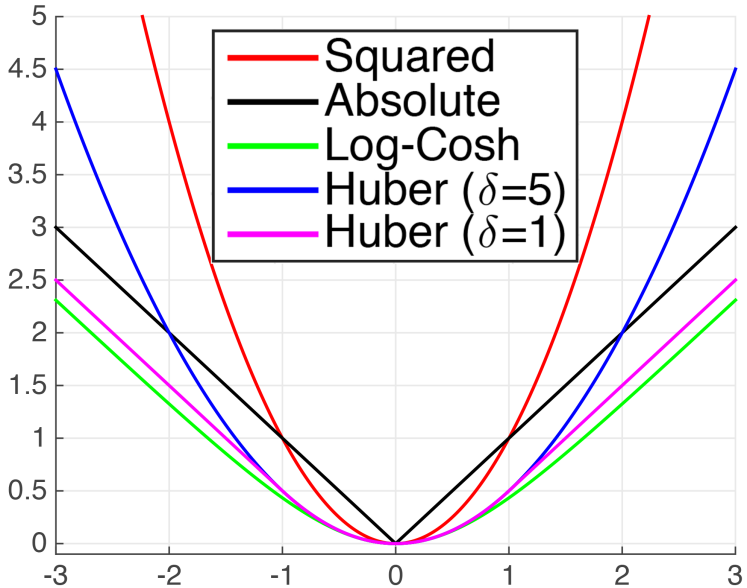
- Good news: it works, but only if...
  - ... N is big enough! Human work, expensive to collect y_n... (UL?),
  - ... proper $l(\hat{y}_n, y_n)$ and $f(\theta, x_n)$ are chosen! Human work, hard... (UL?),
  - ... θ is close to θ^*! Only empirical evidence... (θ^* overfits anyway...).

Supervised Learning

Formally: Loss Function

Loss function:

- Supervised:
  - Regression ($y \in \mathbb{R}^m$):
    squared error: $l(\hat{y}_n, y_n) = \|\hat{y}_n - y_n\|_2^2$
    absolute error: $l(\hat{y}_n, y_n) = \|\hat{y}_n - y_n\|_1$
  - Classification ($y \in \{0,1\}^m$):
    categorical cross-entropy: $l(\hat{y}_n, y_n) = -\sum_{i=1} y_{n,i} \log \hat{y}_{n,i}$
- Unsupervised: when annotation y_n is not provided by human, let $y_n = x_n$
  - $l(\hat{x}_n, x_n) = \|\hat{x}_n - x_n\|_2^2$



Supervised Learning

Formally: Parametric Mapping and Optimization Method

Parametric mapping: we'll get back to this in a moment.

- K Nearest Neighbors: in detail today,
- Random Forests (Decision Trees): in detail today,
- Deep Neural Networks: in detail in next lecture.

Optimization method: to find θ^* varies with loss and parametric mapping.
In general training looks like this:

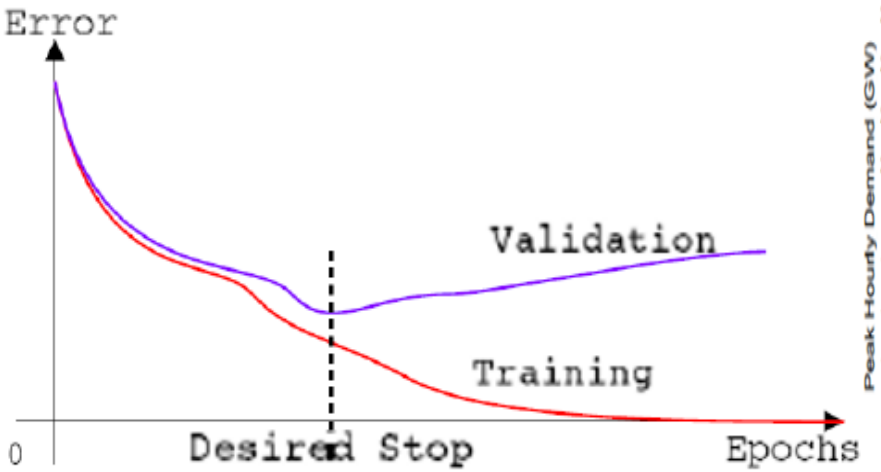
```
def train_and_val(hyp, train_set, val_set):
      model = build_model(hyp)
      for _ in range(300):
            train_losses.append(model.train(train_set))
            val_losses.append(model.test(val_set))
            plot_learning_curves(train_losses, val_losses)
            if val_losses[-1] < best_val_loss:
                  best_val_loss = val_losses[-1]
                  best_weights = weights
      return best_val_loss, best_weights
```

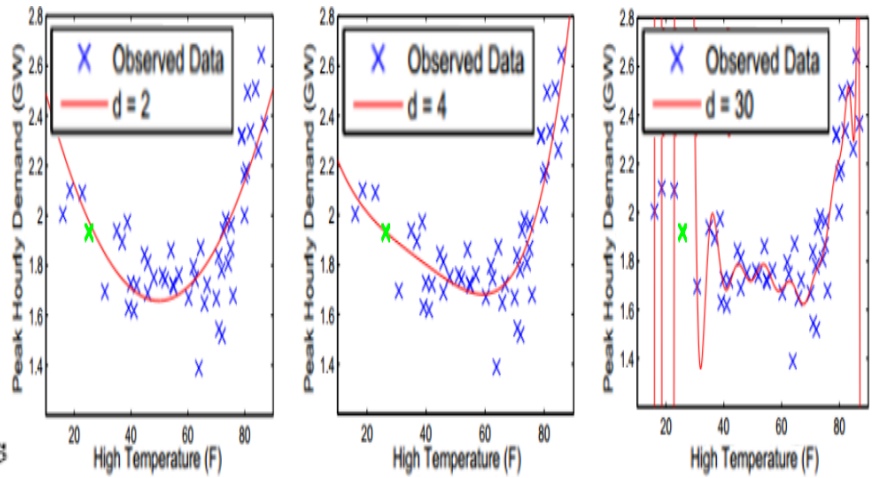
Supervised Learning

Overfitting

It does not generalize well! We optimized only for the pairs $(x_n, y_n), n = 1, \ldots, N$, it could learn characteristics of this set! We call this overfitting.

- Early Stopping: Monitor the loss on a left-out validation set, terminate the optimization when it increases.





- Regularization: Add $\lambda\|\theta\|_2^2$ to $f_0(\theta, x_n, y_n)$ to promote simple models.
- Denoising: Add random noise to the representation during optimization, the parametric mapping must learn to bypass it.
- Data Augmentation: Generate more training samples on the fly, e.g. crop, mirror, rotate images and change their contrast.
- Fine-tuning: Initialize with parameters that were trained on a large dataset previously.

Supervised Learning

Hyperparameters

ML has many hyperparameters: mapping parameter count, optimization parameters, regularization parameters (λ), noise level, etc. How to set these properly?

- Copy the settings of a research paper.
- Optimize with Random Search or Bayesian Optimization: Train with many hyperparameter settings on the training set, greedily choose the best on the validation set and use it on the test set. Unfortunately this multiplies the already massive computational cost...

```
train_set , val_set, test_set = load_dataset()
for _ in range(100):
      hyps.append(pick_hyperparams(hyps, val_losses))
      val_losses.append(train_and_val(hyps[-1], train_set,
val_set))
      if val_losses[-1] < best_val_loss:
            best_val_loss = val_losses[-1]
            best_hyp = hyps[-1]
train_and_val(best_hyp, train_set + val_set, test_set)
```

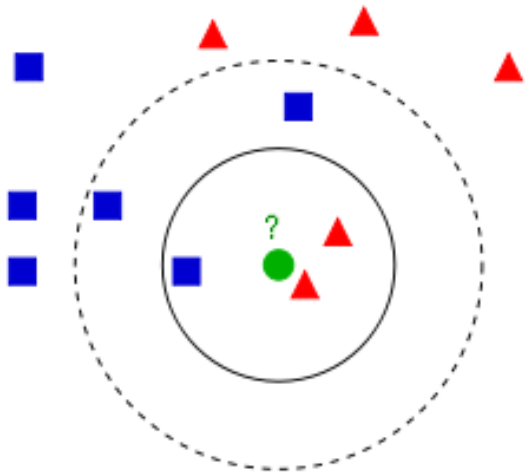
K Nearest Neighbors

Theory

- K Nearest Neighbor (KNN): (Non-)parametric mapping, pick K, predict the average of the K closest training sample outputs:

$$f(\theta, x) = f(x_n, y_n, x) = \sum_{n=1}^{N} \underbrace{\frac{I(x_n \text{ is one of the } K \text{ points closest to } x)}{K}}_{w_n} y_n.$$

Since $\theta = \{(x_n, y_n) \,|\, n = 1, \ldots, N\}$, there is no optimization to be done.
Simply store all training pairs as they are. Closest: $\operatorname{argsort}_n \|x_n - x\|_2^2$.
Pro: Simple to implement. Con: If N is large, storing and sorting is costly.



K Nearest Neighbors

Practice

Implementation is avaiable in sklearn (Python):
sklearn.neighbors.KNeighborsClassifier

```
>>> X = [[0], [1], [2], [3]]
>>> y = [0, 0, 1, 1]
>>> from sklearn.neighbors import KNeighborsClassifier
>>> neigh = KNeighborsClassifier(n_neighbors=3)
>>> neigh.fit(X, y)
KNeighborsClassifier(...)
>>> print(neigh.predict([[1.1]]))
[0]
>>> print(neigh.predict_proba([[0.9]]))
[[ 0.66666667  0.33333333]]
```

sklearn.neighbors.KNeighborsRegressor

```
>>> X = [[0], [1], [2], [3]]
>>> y = [0, 0, 1, 1]
>>> from sklearn.neighbors import KNeighborsRegressor
>>> neigh = KNeighborsRegressor(n_neighbors=2)
>>> neigh.fit(X, y)
KNeighborsRegressor(...)
>>> print(neigh.predict([[1.5]]))
[ 0.5]
```