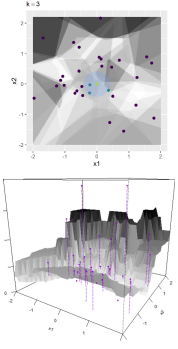
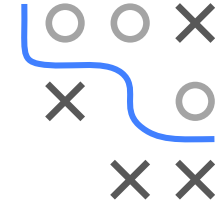
Introduction to Machine Learning

k-Nearest Neighbors

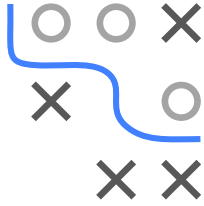




Learning goals

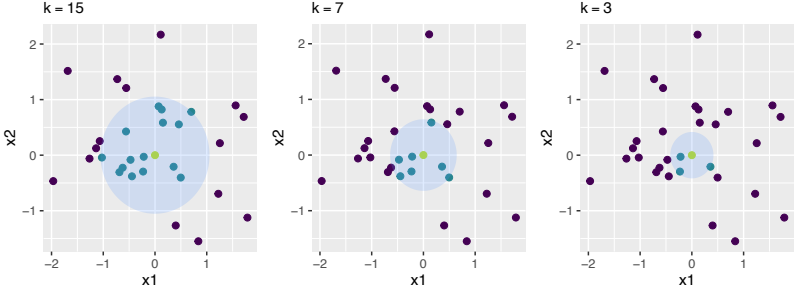
- Understand the basic idea of k-NN for regression and classification
- Understand that k-NN is a non-parametric, local model
- Know different distance measures for different scales of feature variables

K-NEAREST-NEIGHBORS

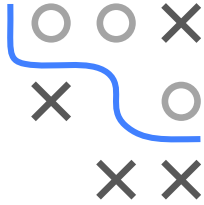


- k-**NN** can be used for regression and classification.
- Generates "similar" predictions for $\mathbf{x}$ to its k closest neighbors.
- "Closeness" requires a distance or similarity measure.
- The subset of $\mathcal{D}_{\text{train}}$ that is at least as close to $\mathbf{x}$ as its k-th closest neighbor $\mathbf{x}^{(k)}$ in $\mathcal{D}_{\text{train}}$ is called the k-**neighborhood** $N_k(\mathbf{x})$ of $\mathbf{x}$:

$$N_k(\mathbf{x}) = \{\mathbf{x}^{(i)} \in \mathcal{D}_{\text{train}} \mid d(\mathbf{x}^{(i)}, \mathbf{x}) \leq d(\mathbf{x}^{(k)}, \mathbf{x})\}$$



DISTANCE MEASURES



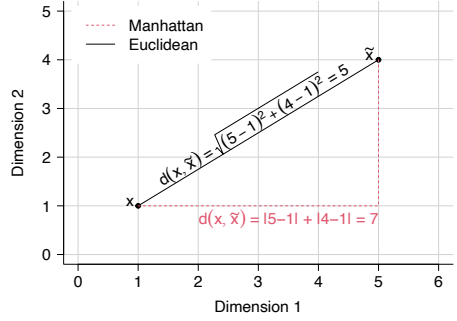
- Popular for numerical features: **Minkowski** distances of the form

$$\|\mathbf{x} - \tilde{\mathbf{x}}\|_q = \left(\sum_{j=1}^{p} |x_j - \tilde{x}_j|^q \right)^{\frac{1}{q}} \quad \text{for } \mathbf{x}, \tilde{\mathbf{x}} \in \mathcal{X} \text{ with } p \text{ numeric features}$$

- Especially, **Manhattan** ($q = 1$) and **Euclidean** ($q = 2$) distance

$$d_{Manhattan}(\mathbf{x}, \tilde{\mathbf{x}}) = \|\mathbf{x} - \tilde{\mathbf{x}}\|_1 = \sum_{j=1}^{p} |x_j - \tilde{x}_j|$$

$$d_{Euclidean}(\mathbf{x}, \tilde{\mathbf{x}}) = \|\mathbf{x} - \tilde{\mathbf{x}}\|_2 = \sqrt{\sum_{j=1}^{p} (x_j - \tilde{x}_j)^2}$$

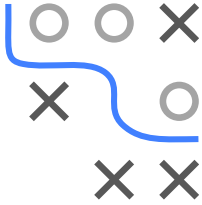


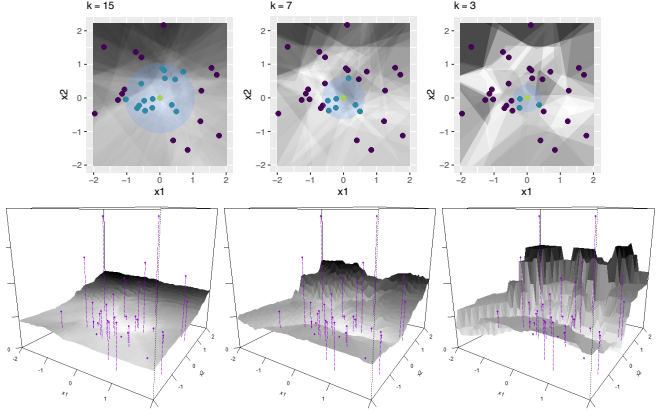
PREDICTION - REGRESSION

Compute for each point the average output y of the k-nearest neighbours in $N_k(\mathbf{x})$:

$$\hat{f}(\mathbf{x}) = \frac{1}{k} \sum_{i:\mathbf{x}^{(i)} \in N_k(\mathbf{x})} y^{(i)} \text{ or } \hat{f}(\mathbf{x}) = \frac{1}{\sum_{i:\mathbf{x}^{(i)} \in N_k(\mathbf{x})} w^{(i)}} \sum_{i:\mathbf{x}^{(i)} \in N_k(\mathbf{x})} w^{(i)} y^{(i)}$$

with neighbors weighted based on their distance to $\mathbf{x}$: $w^{(i)} = \frac{1}{d(\mathbf{x}^{(i)}, \mathbf{x})}$

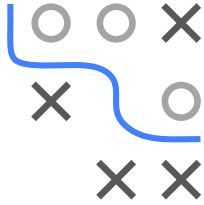




PREDICTION - CLASSIFICATION

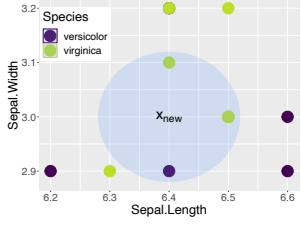
For classification in g groups, a majority vote is used:

$$\hat{h}(\mathbf{x}) = \underset{\ell \in \{1,\dots,g\}}{\arg\max} \sum_{i:\mathbf{x}^{(i)} \in N_k(\mathbf{x})} \mathbb{I}(y^{(i)} = \ell)$$



And posterior probabilities can be estimated with:

$$\hat{\pi}_\ell(\mathbf{x}) = \frac{1}{k} \sum_{i:\mathbf{x}^{(i)} \in N_k(\mathbf{x})} \mathbb{I}(y^{(i)} = \ell)$$



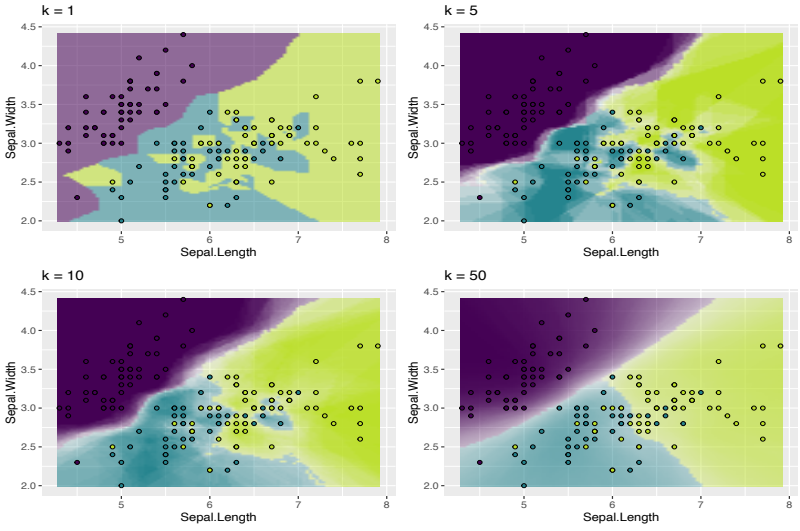
	SL	SW	Species	dist
52	6.4	3.2	versicolor	0.200
59	6.6	2.9	versicolor	0.224
75	**6.4**	**2.9**	**versicolor**	**0.100**
76	6.6	3.0	versicolor	0.200
98	6.2	2.9	versicolor	0.224
104	6.3	2.9	virginica	0.141
105	**6.5**	**3.0**	**virginica**	**0.100**
111	6.5	3.2	virginica	0.224
116	6.4	3.2	virginica	0.200
117	**6.5**	**3.0**	**virginica**	**0.100**
138	6.4	3.1	virginica	0.100
148	6.5	3.0	virginica	0.100

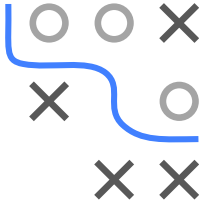
Example with subset of iris data ($k = 3$)

$\hat{\pi}_{setosa}(\mathbf{x}_{new}) = \frac{0}{3} = 0\%$, $\hat{\pi}_{versicolor}(\mathbf{x}_{new}) = \frac{1}{3} = 33\%$, $\hat{\pi}_{virginica}(\mathbf{x}_{new}) = \frac{2}{3} = 67\%$,

$\hat{h}(\mathbf{x}_{new}) = virginica$

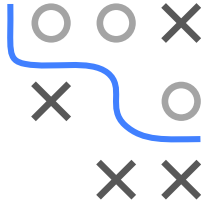
K-NN: FROM SMALL TO LARGE *K*





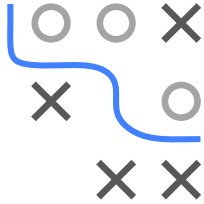
Complex, local model vs smoother, more global model

K-NN SUMMARY



- k-NN is a lazy classifier, it has no real training step, it simply stores the complete data - which are needed during prediction.
- Hence, its parameters are the training data, there is no real compression of information.
- As the number of parameters grows with the number of training points, we call k-NN a non-parametric model
- k-NN is not based on any distributional or functional assumption, and can, in theory, model data situations of arbitrary complexity.
- The smaller k, the less stable, less smooth and more “wiggly” the decision boundary becomes.
- Accuracy of k-NN can be severely degraded by the presence of noisy or irrelevant features, or when the feature scales are not consistent with their importance.

STANDARDIZATION AND WEIGHTS

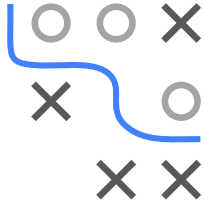


- **Standardization:** Features in k-NN are usually standardized or normalized. If two features have values on a very different range, most distances would place a higher importance on the one with a larger range, leading to an imbalanced influence of that feature.
- **Importance:** Sometimes one feature has a higher importance (maybe we know this via domain knowledge). It can now manually be upweighted to reflect this.

$$d_{Euclidean}^{weighted}(\mathbf{x}, \tilde{\mathbf{x}}) = \sqrt{\sum_{j=1}^{p} w_j (x_j - \tilde{x}_j)^2}$$

- If these weights would have to be learned in a data-driven manner, we could only do this by hyperparameter tuning in k-NN. This is inconvenient, and Gaussian processes handle this much better.

GOWER DISTANCE



- A weighted mean of univ. distances in the j-th feature.
- It can handle categoricals, missings, and different ranges.

$$d_{gower}(\mathbf{x}, \tilde{\mathbf{x}}) = \frac{\sum_{j=1}^{p} \delta_{x_j,\tilde{x}_j} \cdot d_{gower}(x_j, \tilde{x}_j)}{\sum_{j=1}^{p} \delta_{x_j,\tilde{x}_j}}.$$

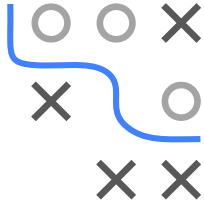
- $\delta_{x_j,\tilde{x}_j}$ is 0 or 1. It's 0 if *j*-th feature is *missing* in at least one observation, or when the feature is asymmetric binary (where "1" is more important than "0") and both values are zero. Otherwise 1.
- $d_{gower}(x_j, \tilde{x}_j)$: For nominals it's 0 if both values are equal and 1 otherwise. For integers and reals, it's the absolute difference, divided by range.

GOWER DISTANCE

Example of Gower distance with data on sex and income:

index	sex	salary
1	m	2340
2	w	2100
3	NA	2680

$$d_{gower}(\mathbf{x}, \tilde{\mathbf{x}}) = \frac{\sum_{j=1}^{p} \delta_{x_j,\tilde{x}_j} \cdot d_{gower}(x_j,\tilde{x}_j)}{\sum_{j=1}^{p} \delta_{x_j,\tilde{x}_j}}$$

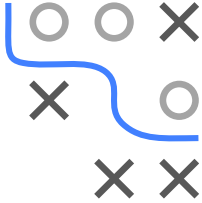


$$d_{gower}(\mathbf{x}^{(1)}, \mathbf{x}^{(2)}) = \frac{1 \cdot 1 + 1 \cdot \frac{|2340-2100|}{|2680-2100|}}{1+1} = \frac{1+\frac{240}{580}}{2} = \frac{1+0.414}{2} = 0.707$$

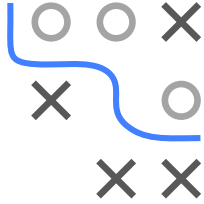
$$d_{gower}(\mathbf{x}^{(1)}, \mathbf{x}^{(3)}) = \frac{0 \cdot 1 + 1 \cdot \frac{|2340-2680|}{|2680-2100|}}{0+1} = \frac{0+\frac{340}{580}}{1} = \frac{0+0.586}{1} = 0.586$$

$$d_{gower}(\mathbf{x}^{(2)}, \mathbf{x}^{(3)}) = \frac{0 \cdot 1 + 1 \cdot \frac{|2100-2680|}{|2680-2100|}}{0+1} = \frac{0+\frac{580}{580}}{1} = \frac{0+1.000}{1} = 1$$

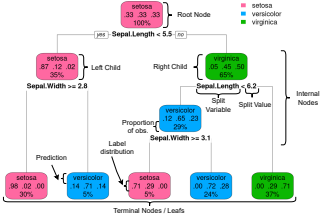
INTRODUCTION TO MACHINE LEARNING



Introduction to Machine Learning



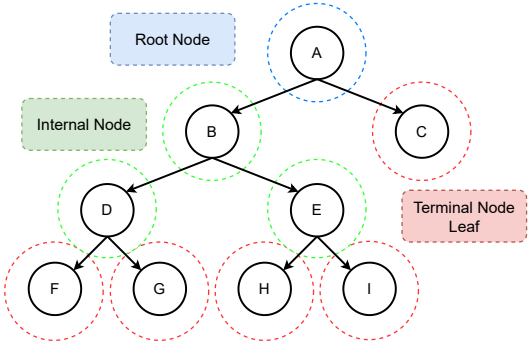
CART
Predictions with CART

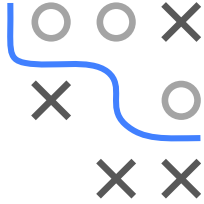


Learning goals

- Understand the basic structure of a tree model
- Understand that the basic idea of a tree model is the same for classification and regression
- Understand how the label of a new observation is predicted via CART
- Know hypothesis space of CART

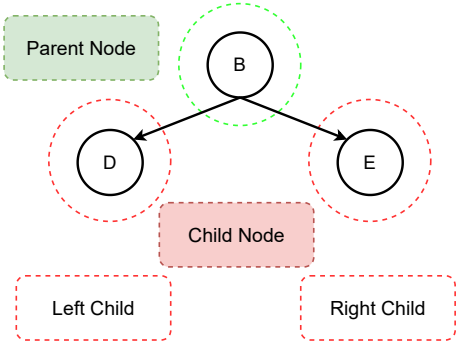
BINARY TREES

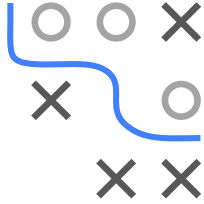




- Binary trees represent a top-down hierarchy with binary splits
- A tree is composed of different node types: a) the root node, b) internal nodes, and c) terminal nodes (also leaves).

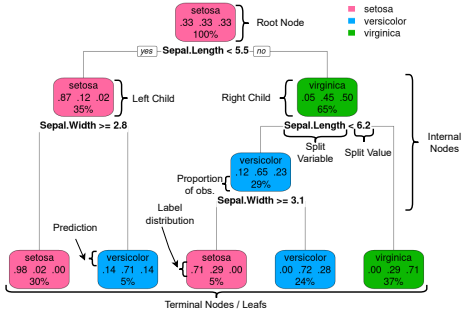
BINARY TREES

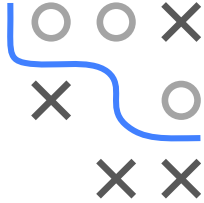




- Nodes have relative relationships, they can be:
  - Parent nodes
  - Child nodes
- Root nodes don't have parents – leaves don't have children

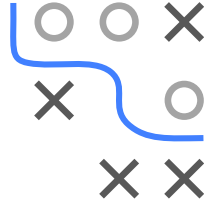
CLASSIFICATION TREES



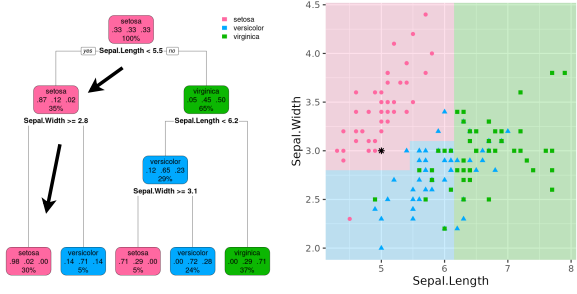


- Classification trees use the structure of a binary tree
- Binary splits are constructed top-down in a *data optimal* way
- Each split is a threshold decision for a single feature
- Each node contains the training points which follow its path
- Each leaf contains a constant prediction

CLASSIFICATION TREE MODEL AND PREDICTION

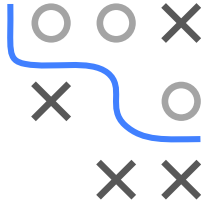


- When predicting new data (here*: `Sepal.Length` = 5, `Sepal.Width` = 3) we use the learned split points and pass an observation through the tree
- Each observation is assigned to exactly one leaf
- Classification trees can make hard-label predictions (here: `setosa`) or predict probabilities (here: 0.98, 0.02, 0.00)

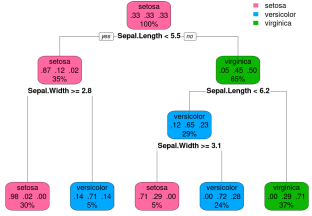


CART AS A RULE BASED MODEL

Leaf nodes can be expressed by a set of rules (left to right):

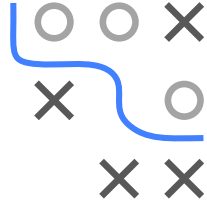


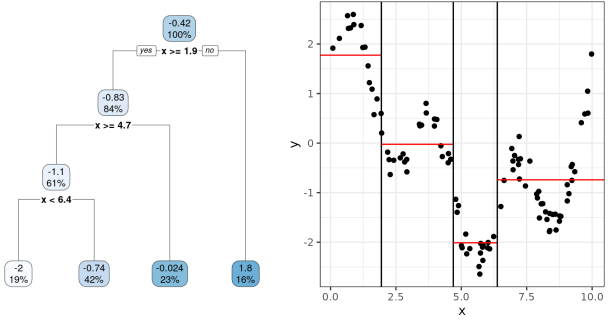
Hard label prediction	Label distribution	`Sepal.Width`	`Sepal.Length`
`setosa`	0.98, 0.02, 0.00	≥ 2.8	< 5.5
`versicolor`	0.14, 0.71, 0.14	< 2.8	< 5.5
`setosa`	0.71, 0.29, 0.00	≥ 3.1	≥ 5.5 & < 6.2
`versicolor`	0.00, 0.72, 0.28	< 3.1	≥ 5.5 & < 6.2
`virginica`	0.00, 0.29, 0.71	–	≥ 6.2



REGRESSION TREE MODEL AND PREDICTION

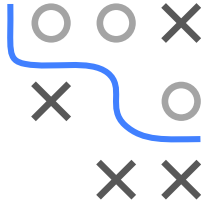
- Works the same way as for classification
- But predictions in leaf nodes are a numerical scalar





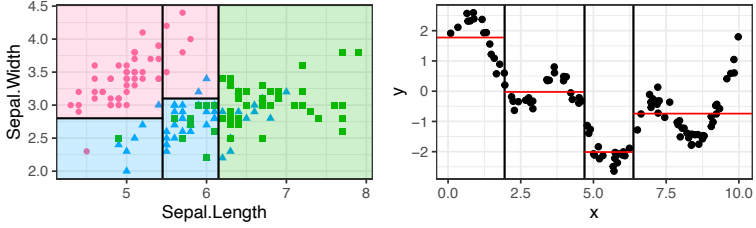
TREE AS AN ADDITIVE MODEL

Trees divide the feature space $\mathcal{X}$ into **rectangular regions**:



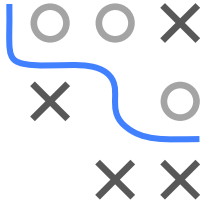
$$f(\mathbf{x}) = \sum_{m=1}^{M} c_m \mathbb{I}(\mathbf{x} \in Q_m),$$

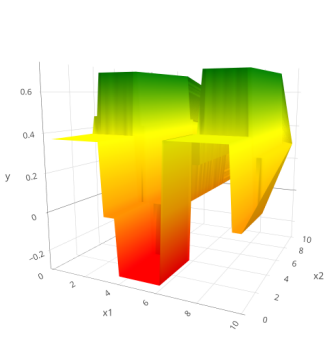
where a tree with M leaf nodes defines M "rectangles" Q_m.
c_m is the predicted numerical response, class label or class distribution in the respective leaf node.

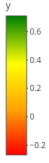


TREE AS AN ADDITIVE MODEL

A 2D regression example:

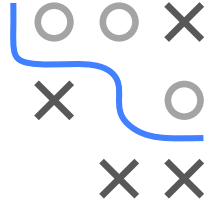






(For binary classification with probabilities, 2D surface looks similar.)

Introduction to Machine Learning

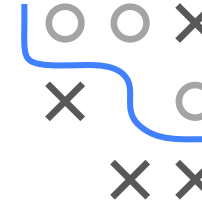


CART
Growing a Tree

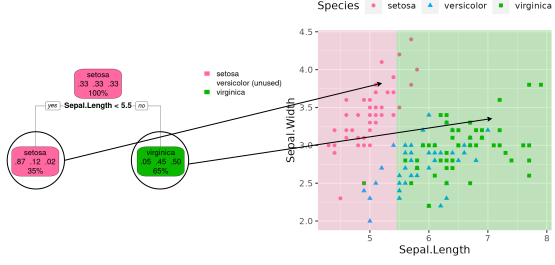
Learning goals

- Understand how a tree is grown by an exhaustive search
- Know where and how the split point is set

TREE GROWING

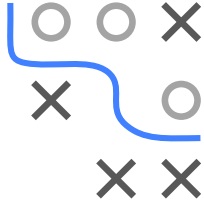


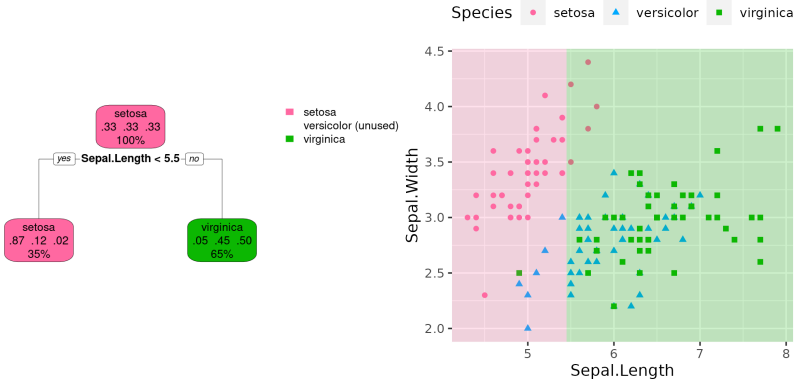
- We start with an empty tree, a root node that contains all the data. Trees are then grown by recursively applying **greedy** optimization to each node $\mathcal{N}$.
- Greedy means we do an **exhaustive search**: Ideally, all possible splits of $\mathcal{N}$ on all possible points t for all features x_j are compared in terms of their empirical risk $\mathcal{R}(\mathcal{N}, j, t)$.
- The training data is then distributed to child nodes according to the optimal split and the procedure is repeated in the child nodes.



TREE GROWING

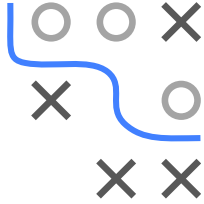
1. Start with a root node of all data.
2. Search for feature and split point that minimizes the empirical risk in child nodes – makes label distribution more homogenous.

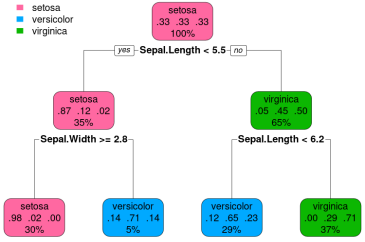


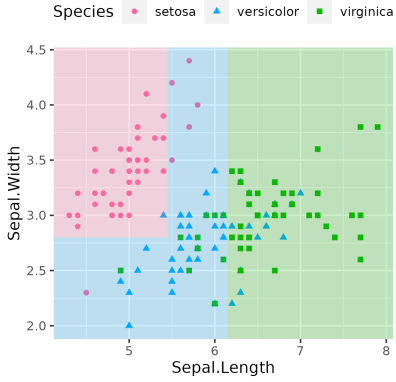


TREE GROWING

❸ Proceed recursively for each child node: Select best split and divide data from parent node into left and right child nodes.

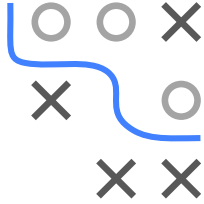


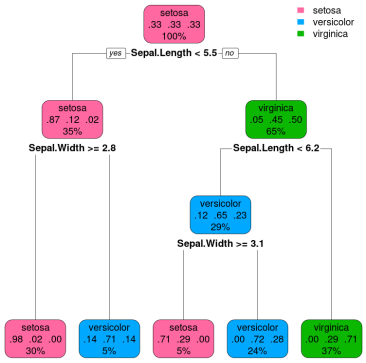


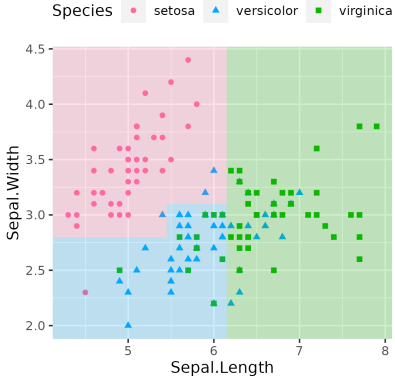


TREE GROWING

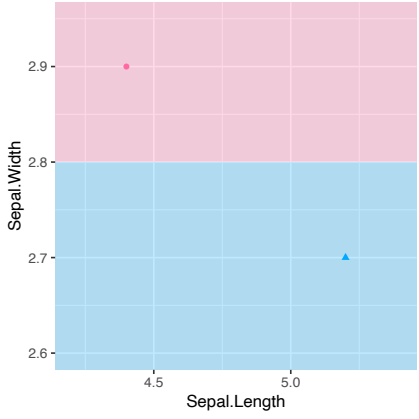
4. Repeat until we reach a stop criterion, e.g., until each leaf cannot be split further.

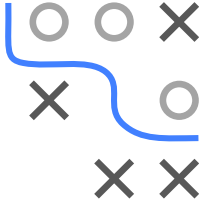






SPLIT PLACEMENT

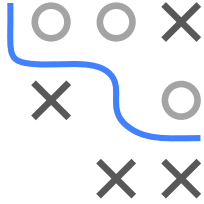


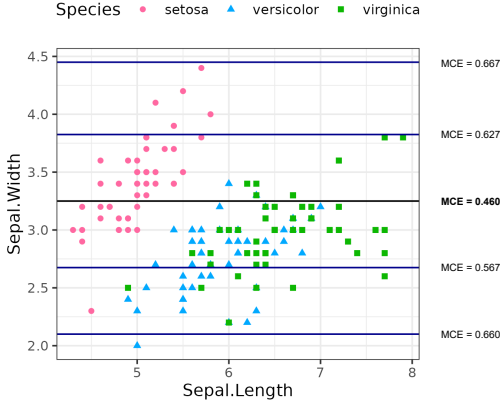


Splits are usually placed at the mid-point of the observations they split: the large margin to the next closest observations makes better generalization on new, unseen data more likely.

FINDING THE SPLIT

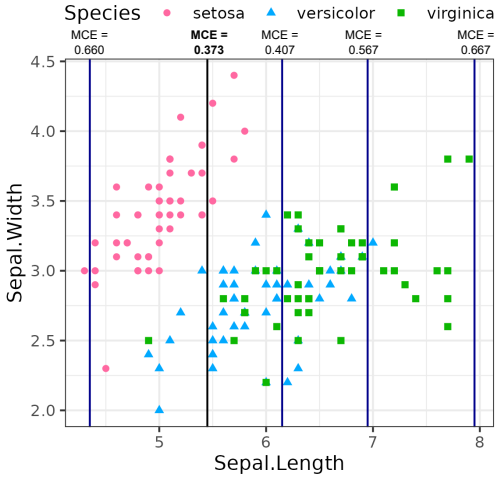
Assume we split the data so that the misclassification error (MCE) is minimal through the splitting.
First, we check a set of potential splits for `Sepal.Width`

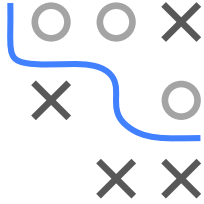




FINDING THE SPLIT

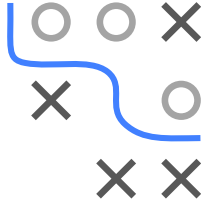
Then we check a set of potential splits for `Sepal.Length`

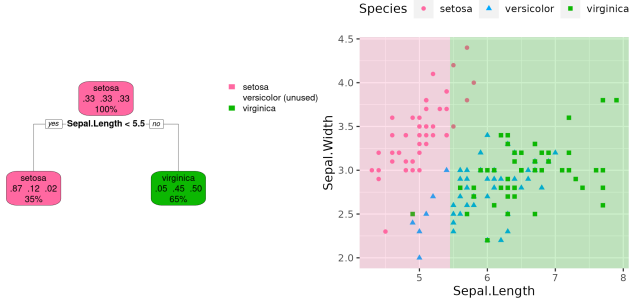




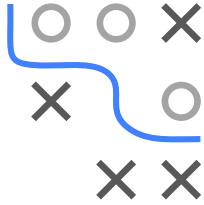
FINDING THE SPLIT

- We take the split with lowest MCE: `Sepal.Length` = 5.5
- In real life, we actually search over many more splitting points. Common strategies involve: a) Searching over all possible split points (exhaustive search), b) searching quantile-wise
- MCE is rarely used, we will cover split criteria in detail later.

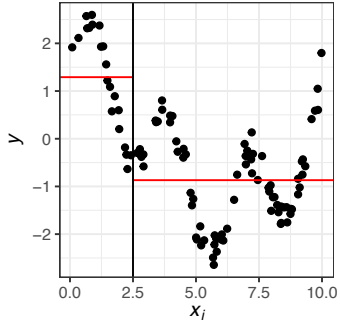




Introduction to Machine Learning



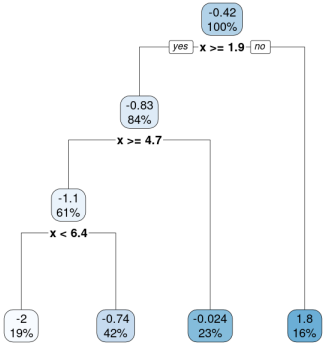
CART
Splitting Criteria for Regression

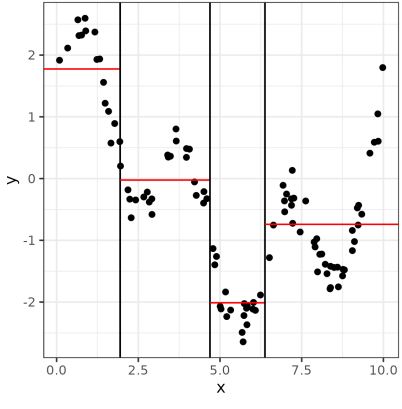


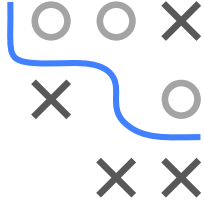
Learning goals

- Understand how to define split criteria via ERM
- Understand how to find splits in regression with L_2 loss

SPLITTING CRITERIA



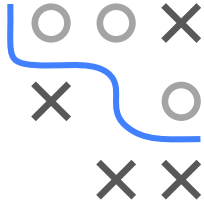




How to find good splitting rules? $\Longrightarrow$ **Empirical Risk Minimization**

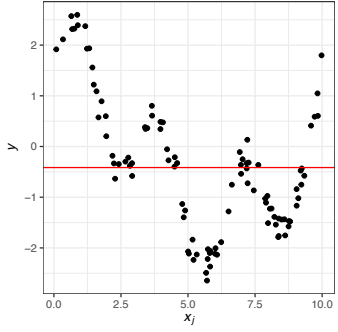
OPTIMAL CONSTANTS IN LEAVES

Idea: A split is good if each child's point predictor reflects its data well.



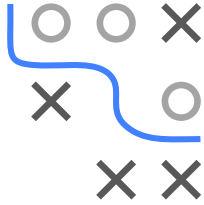
For each child $\mathcal{N}$, predict with optimal constant, e.g., the mean $c_{\mathcal{N}} = \frac{1}{|\mathcal{N}|} \sum_{(\mathbf{x},y)\in\mathcal{N}} y$ for the L_2 loss, i.e., $\mathcal{R}(\mathcal{N}) = \sum_{(\mathbf{x},y)\in\mathcal{N}} (y - c_{\mathcal{N}})^2$.

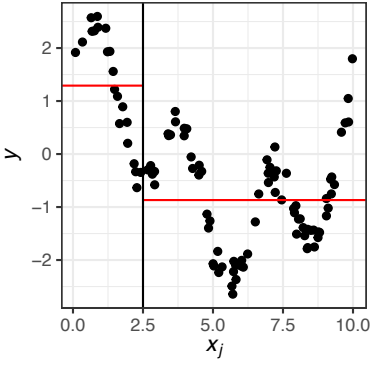
Root node:

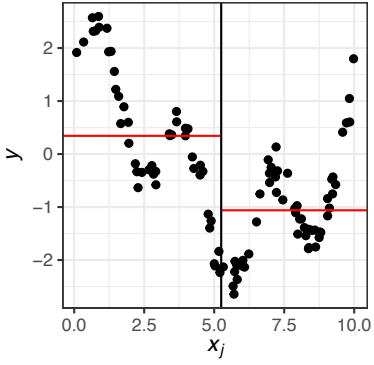


OPTIMAL CONSTANTS IN LEAVES

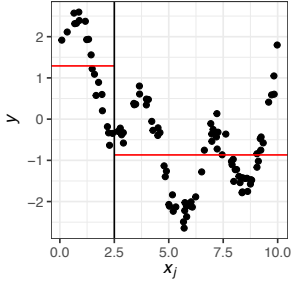
Which of these two splits is better?

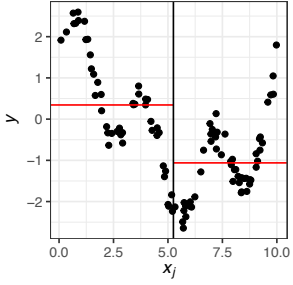


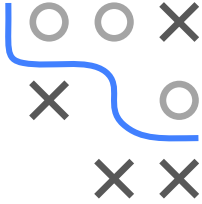




RISK OF A SPLIT







$\mathcal{R}(\mathcal{N}_1) = 23.4, \mathcal{R}(\mathcal{N}_2) = 72.4$ $\mathcal{R}(\mathcal{N}_1) = 78.1, \mathcal{R}(\mathcal{N}_2) = 46.1$

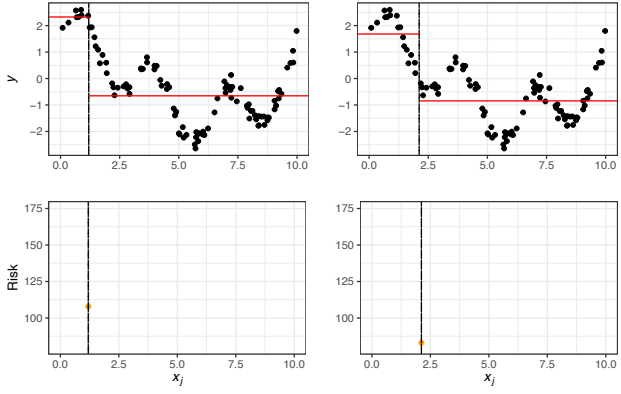
The total risk is the sum of the individual losses:

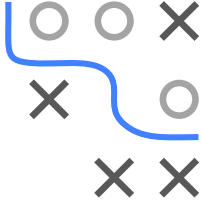
$23.4 + 72.4 = 95.8$ $78.0 + 46.1 = 124.1$

Based on the SSE, we prefer the first split.

SEARCHING THE BEST SPLIT

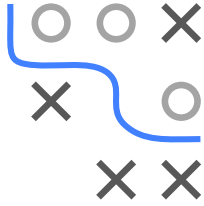
Let's find the best split for this feature by tabulating results.

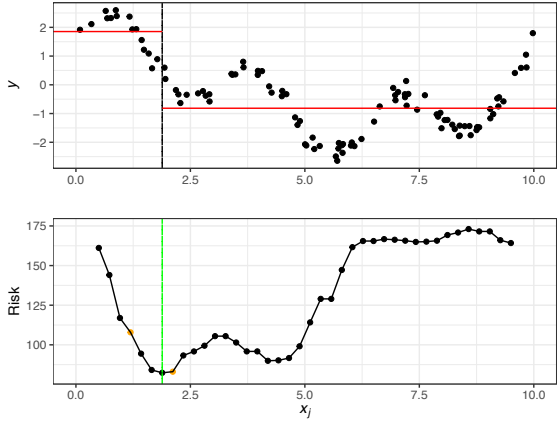




SEARCHING THE BEST SPLIT

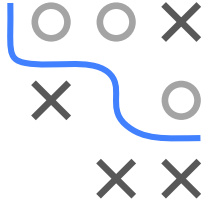
Let's iterate – quantile-wise or over all points.





We have reduced the problem to a simple loop.

FORMALIZATION



- $\mathcal{N} \subseteq \mathcal{D}$ is the data contained in this node
- Let $c_{\mathcal{N}}$ be the predicted constant for $\mathcal{N}$
- The risk $\mathcal{R}(\mathcal{N})$ for a node is:

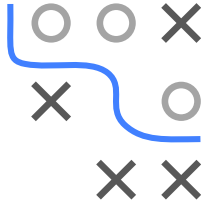
$$\mathcal{R}(\mathcal{N}) = \sum_{(\mathbf{x},y)\in\mathcal{N}} L(y, c_{\mathcal{N}})$$

- The optimal constant is $c_{\mathcal{N}} = \arg\min_{c} \sum_{(\mathbf{x},y)\in\mathcal{N}} L(y, c)$
- We often know what that is from theoretical considerations – or we can perform a simple univariate optimization

FORMALIZATION

- A split w.r.t. **feature** x_j **at split point** t divides a parent node $\mathcal{N}$ into

$$\mathcal{N}_1 = \{(\mathbf{x}, y) \in \mathcal{N} : x_j < t\} \text{ and } \mathcal{N}_2 = \{(\mathbf{x}, y) \in \mathcal{N} : x_j \geq t\}.$$



- To evaluate its quality, we compute the risk of our new, finer model

$$\begin{aligned}\mathcal{R}(\mathcal{N}, j, t) &= \mathcal{R}(\mathcal{N}_1) + \mathcal{R}(\mathcal{N}_2) \\ &= \left(\sum_{(\mathbf{x},y)\in\mathcal{N}_1} L(y, c_{\mathcal{N}_1}) + \sum_{(\mathbf{x},y)\in\mathcal{N}_2} L(y, c_{\mathcal{N}_2}) \right)\end{aligned}$$

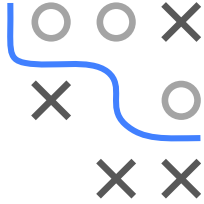
- Finding the best way to split $\mathcal{N}$ into $\mathcal{N}_1, \mathcal{N}_2$ means solving

$$\arg\min_{j,t} \mathcal{R}(\mathcal{N}, j, t)$$

FORMALIZATION

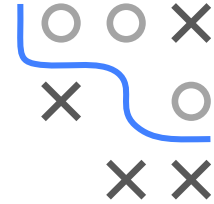
- $\mathcal{R}(\mathcal{N}, j, t) = \mathcal{R}(\mathcal{N}_1) + \mathcal{R}(\mathcal{N}_2)$, makes sense if $\mathcal{R}$ is a simple sum
- If we use averages, we have to reweight the terms to obtain a global average w.r.t. $\mathcal{N}$ as the children have different sizes

$$\bar{\mathcal{R}}(\mathcal{N}, j, t) = \frac{|\mathcal{N}_1|}{|\mathcal{N}|}\bar{\mathcal{R}}(\mathcal{N}_1) + \frac{|\mathcal{N}_2|}{|\mathcal{N}|}\bar{\mathcal{R}}(\mathcal{N}_2)$$

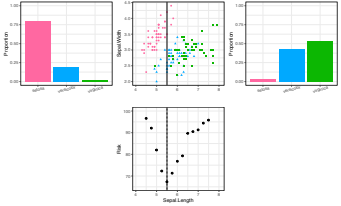


- We mention this for clarity, as quite a few texts contain only the (more complicated) weighted formula without clear explanation

Introduction to Machine Learning



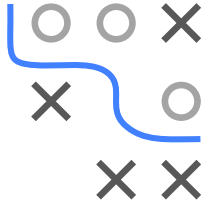
CART
Splitting Criteria for Classification



Learning goals

- Understand how to define split criteria via ERM
- Understand how to find splits in regression with L_2 loss

OPTIMAL CONSTANT MODELS



As losses in classification, we typically use:

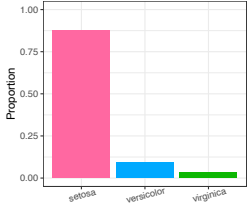
- (Multi-class) Brier score $L(y, \pi) = \sum_{k=1}^{g} (\pi_k - o_k(y))^2$,
  a.k.a. L_2 loss on probabilities
- (Multi-class) Log loss $L(y, \pi) = -\sum_{k=1}^{g} o_k(y) \log(\pi_k)$,
  as in logistic regression

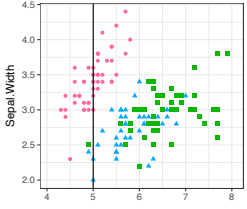
Optimal constant predictions (in a node) for both losses are simply the proportions of the contained classes:

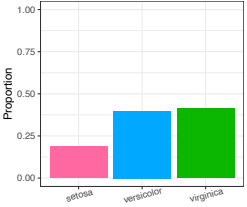
$$c_{\mathcal{N}} = (\hat{\pi}_1^{(\mathcal{N})}, \ldots, \hat{\pi}_g^{(\mathcal{N})}) \quad \text{with}$$
$$\hat{\pi}_k^{(\mathcal{N})} = \frac{1}{|\mathcal{N}|} \sum_{(\mathbf{x}, y) \in \mathcal{N}} \mathbb{I}(y = k) \quad \forall k \in \{1, \ldots, g\}$$

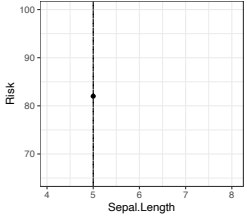
FINDING THE BEST SPLIT

Let's compute the Brier score for all splits, with optimal constant probability vectors in both children



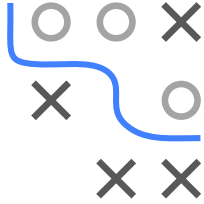


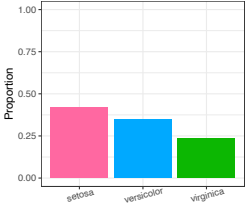


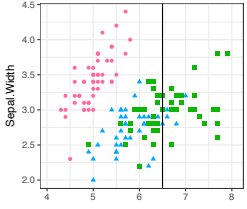


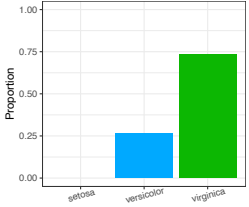
FINDING THE BEST SPLIT

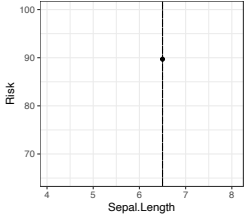
Let's compute the Brier score for all splits, with optimal constant probability vectors in both children





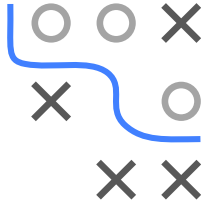


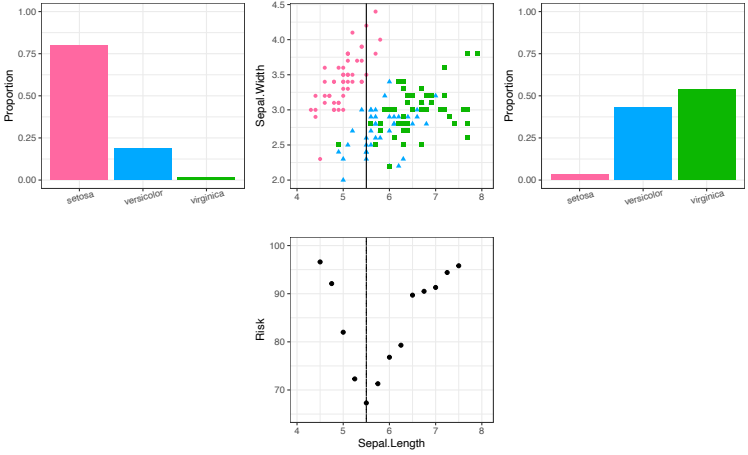




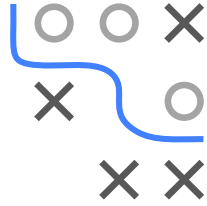
FINDING THE BEST SPLIT

The optimal split point typically creates greatest imbalance or purity of label distribution





RISK MINIMIZATION VS. IMPURITY



- Split crits are sometimes defined in terms of impurity reduction instead of ERM, where a measure of "impurity" is defined per node
- For regression trees, "impurity" is simply defined as variance of y, which is quite obviously L_2 loss
- Brier score is equivalent to Gini impurity

$$I(\mathcal{N}) = \sum_{k=1}^{g} \hat{\pi}_k^{(\mathcal{N})} \left(1 - \hat{\pi}_k^{(\mathcal{N})}\right)$$

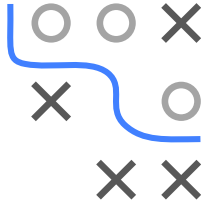
- Log loss is equivalent to entropy

$$I(\mathcal{N}) = -\sum_{k=1}^{g} \hat{\pi}_k^{(\mathcal{N})} \log \hat{\pi}_k^{(\mathcal{N})}$$

- Trees can be understood completely through the lens of ERM, so this new terminology is unnecessary and perhaps confusing

SPLITTING WITH MISCLASSIFICATION LOSS

- Often, we want to minimize the MCE in classification
- Zero-One-Loss is not differentiable, but that is a non-issue in the tree-optimization based on loops
- Brier score and Log loss more sensitive to changes in the node probs, often produce purer nodes, and are still preferred



Split 1:

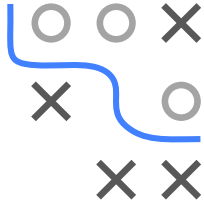
	class 0	class 1
$\mathcal{N}_1$	300	100
$\mathcal{N}_2$	100	300

Split 2:

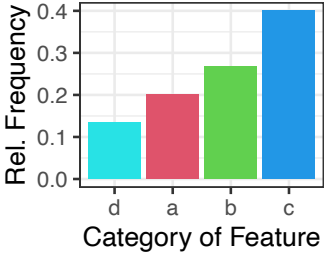
	class 0	class 1
$\mathcal{N}_1$	400	200
$\mathcal{N}_2$	0	200

- Both splits are equivalent in MCE
- But: Split 2 results in purer nodes, both Brier score (Gini) and Log loss (Entropy) prefer 2nd split

Introduction to Machine Learning



CART
Computational Aspects of Finding Splits

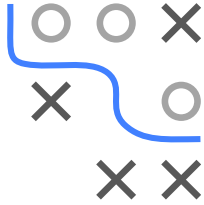


Learning goals

- Know how monotone feature transformations affect the tree
- Understand how categorical features can be treated effectively while growing a CART
- Understand how missing values can be treated in a CART

MONOTONE FEATURE TRANSFORMATIONS

Monotone transformations of one or several features will neither change the value of the splitting criterion nor the structure of the tree, only the numerical value of the split point.

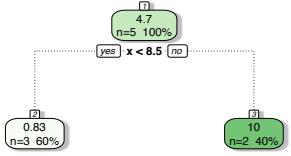


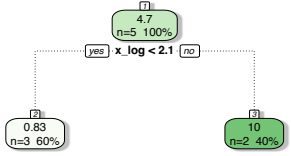
Original data

x	1.0	2.0	7.0	10.0	20.0
y	1.0	1.0	0.5	9.0	11.0

Data with log-transformed x

log(x)	0.0	0.7	1.9	2.3	3.0
y	1.0	1.0	0.5	9.0	11.0

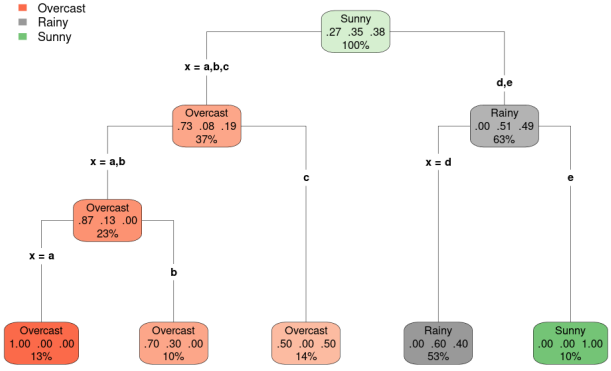




CATEGORICAL FEATURES

- A split on a categorical feature partitions the feature levels:

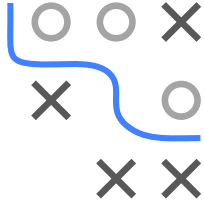
$$x_j \in \{a, b, c\} \leftarrow \mathcal{N} \rightarrow x_j \in \{d, e\}$$



CATEGORICAL FEATURES

- A split on a categorical feature partitions the feature levels:

$$x_j \in \{a, b, c\} \leftarrow \mathcal{N} \rightarrow x_j \in \{d, e\}$$

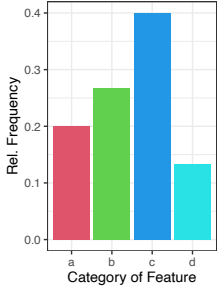


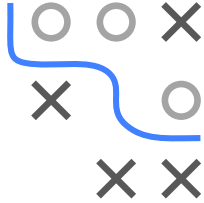
- For a feature with m levels, there are about 2^m different possible partitions of the m values into two groups ($2^{m-1} - 1$ because of symmetry and empty groups).
- Searching over all these becomes prohibitive for large values of m.
- For regression with L2 loss and for binary classification, we can define clever shortcuts.

CATEGORICAL FEATURES

For $0 - 1$ responses, in each node:

1. Calculate the proportion of 1-outcomes for each category of the feature in $\mathcal{N}$.

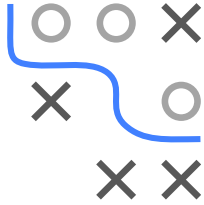


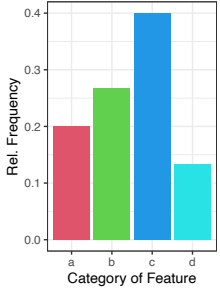


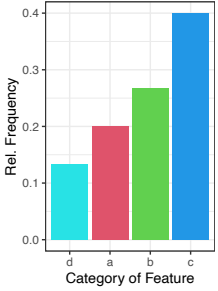
CATEGORICAL FEATURES

For $0 - 1$ responses, in each node:

1. Calculate the proportion of 1-outcomes for each category of the feature in $\mathcal{N}$.
2. Sort the categories according to these proportions.



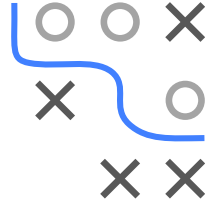


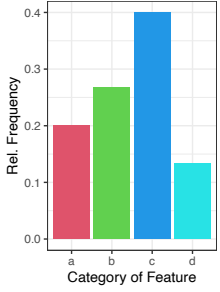


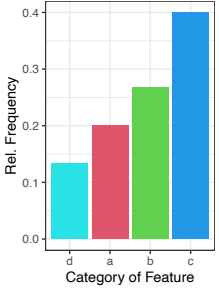
CATEGORICAL FEATURES

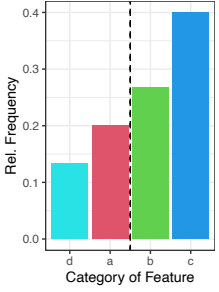
For $0 - 1$ responses, in each node:

1. Calculate the proportion of 1-outcomes for each category of the feature in $\mathcal{N}$.
2. Sort the categories according to these proportions.
3. The feature can then be treated as if it was ordinal, so we only have to investigate at most $m - 1$ splits.

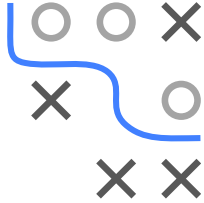








CATEGORICAL FEATURES

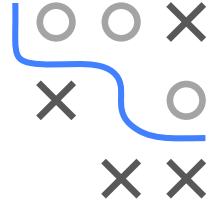


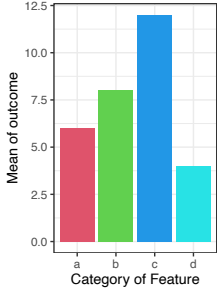
- This procedure finds the optimal split.
- This result also holds for regression trees (with L2 loss) if the levels of the feature are ordered by increasing mean of the target
- The proofs are not trivial and can be found here:
  - for 0-1 responses:
    - ▸ Breiman, 1984, Chapter 4
    - ▸ Ripley, 1996, pp. 213 et seqq.
  - for continuous responses:
    - ▸ Fisher, 1958
- There are only heuristics for the multiclass case ▸ Wright and König, 2019

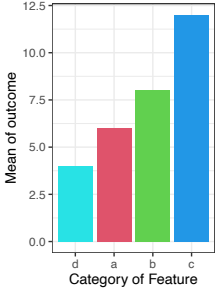
CATEGORICAL FEATURES

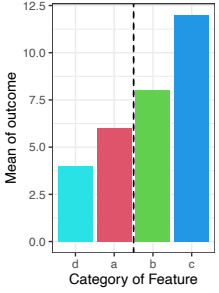
For continuous responses, in each node:

1. Calculate the mean of the outcome in each category
2. Sort the categories by increasing mean of the outcome



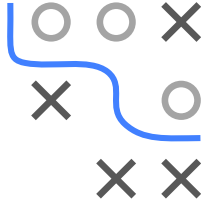






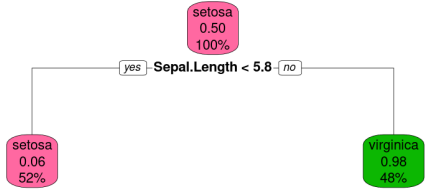
MISSING FEATURE VALUES

- When splits are evaluated, only observations for which the used feature is not missing are used. (This can actually bias splits towards using features with lots of missing values.)
- **Surrogate splits** can deal with missing values during prediction.
- Surrogate splits are created during training. They define replacement splitting rules, using a different feature, that result in almost the same child nodes as the original split.
- When observations are passed down the tree, and the feature value used in a split is missing, we use the surrogate split instead to decide to which child the data should be assigned.



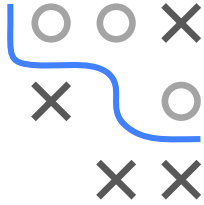
SURROGATE SPLITS

- Each surrogate split is a decision stump that tries to learn the actual splitting rule
- Consider this tree with the primary split w.r.t. `Sepal.Length` where we perform binary classification (`setosa` vs. `virginica`):



- Our surrogate split should optimize a splitting criterion w.r.t. `Sepal.Length < 5.8`

SURROGATE SPLITS

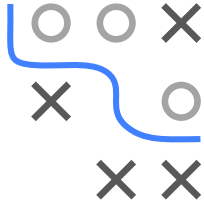


- Consider this subsample of the data used to fit the tree:

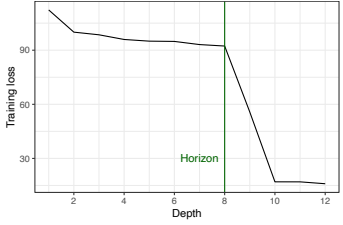
	Sepal.Length	...	Petal.Width	Species	Sepal.Length $<$ 5.8
1	5.10	...	0.20	setosa	TRUE
4	4.60	...	0.20	setosa	TRUE
9	4.40	...	0.20	setosa	TRUE
15	5.80	...	0.20	setosa	FALSE
18	5.10	...	0.30	setosa	TRUE
52	5.80	...	1.90	virginica	FALSE
57	4.90	...	1.70	virginica	TRUE
62	6.40	...	1.90	virginica	FALSE
77	6.20	...	1.80	virginica	FALSE
99	6.20	...	2.30	virginica	FALSE

- Add column that indicates whether `Sepal.Length < 5.8`
- Fit tree of depth 1 using all features but `Sepal.Length` to derive a split that explains `Sepal.Length < 5.8` best $\Rightarrow$ surrogate split
- Typically, software stores the best and a few more surrogate splits

Introduction to Machine Learning



CART
Stopping Criteria & Pruning

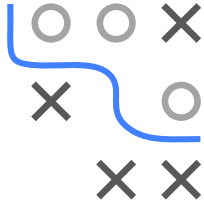


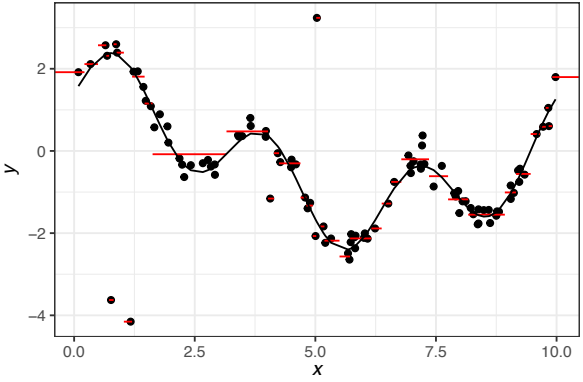
Learning goals

- Understand which problems arise when growing the tree until the end
- Know different stopping criteria
- Understand the idea of pruning

OVERFITTING TREES

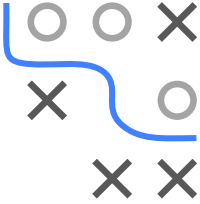
The recursive partitioning procedure used to grow a CART could run until every leaf only contains a single observation. Problem: Very complex trees will *overfit the training data.* At some point we should stop splitting nodes into ever smaller child nodes:

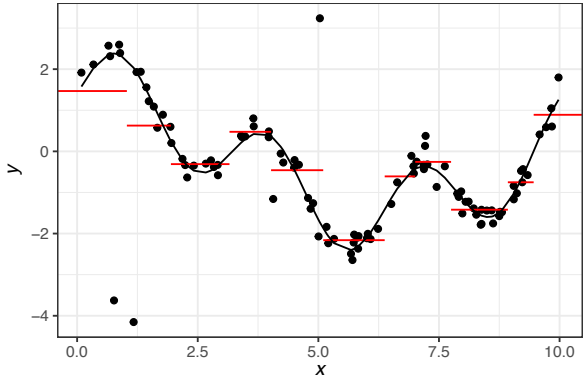




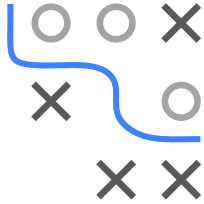
OVERFITTING TREES

We can reduce overfitting to some extent with a less deep tree:





STOPPING CRITERIA



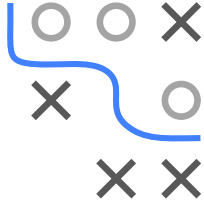
We can define different **stopping criteria**, e.g.: Don't split a node if

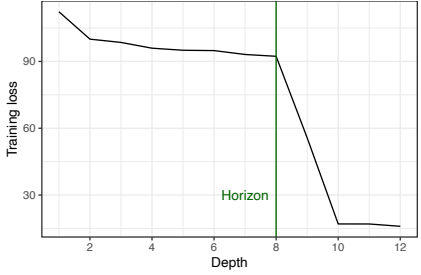
- a certain number of leaves if reached,
- it contains too few observations,
- splitting results in children with too few observations,
- splitting does not achieve a certain minimal improvement of the risk in the children, compared to the risk in the parent node,
- it already has the same target value (**pure node**) or identical feature values for all observations.

Selection of a stopping criterion and its concrete values are hyperparameters of CART.

HORIZON EFFECT

It is hard to tell where we should stop while we're growing the tree: Before we have actually tried all possible additional splits further down a branch, we can't know whether any one of them will be able to reduce the risk by a lot (*horizon effect*).

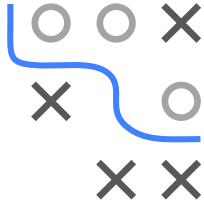




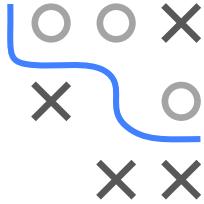
PRUNING

We try to tackle the horizon effect by **pruning**, a method to select the optimal size of a tree:

- Finding a combination of suitable strict stopping criteria (“pre-pruning”) is a hard problem (see chapter on **tuning**).
- Alternative: Grow a large tree, then remove branches so that the resulting smaller tree has lower risk (“post-pruning”).
- Often, post-pruning is meant when referring to pruning.



POST-PRUNING: CCP



- Prominent pruning method: Cost-complexity pruning (CCP)
- Idea: Grow a large tree and remove the least informative leaves
- CCP is steered with a regularization parameter α that penalizes the number of leaves in a sub tree

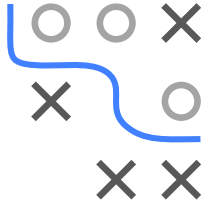
$$\mathcal{R}_{\text{reg}}(T) = \sum_{m=1}^{|T|} \sum_{i:x^{(i)} \in Q_m} L(y^{(i)}, c_m) + \alpha |T|,$$

where $|T|$ is the number of leaves of sub tree T, Q_m is the subset of the feature space related to the m-th terminal node, with its prediction c_m, and T_0 is the complete tree.

CCP

CCP performs a greedy backward search:

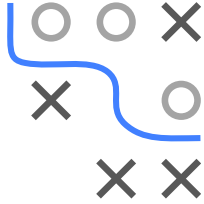
- Computes $\mathcal{R}_{\text{reg}}(T)$ with a fixed α for all possible sub trees that can be created by replacing one internal node with a leaf.
- By replacing a node we also eliminate all subsequent nodes.
- We select the sub tree with lowest risk and repeat the procedure.
- We stop if pruning does not further reduce the risk.

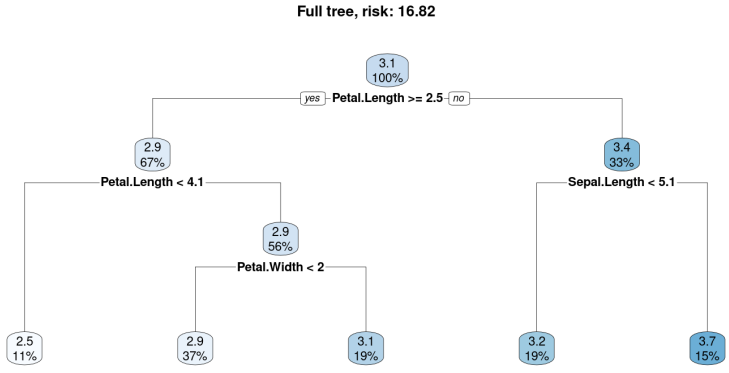


- This is proven to result in the pruned tree with the lowest risk.
- For $\alpha = 0$, we would obviously select T_0.
- Hyperparameter α is typically selected via cross-validation.
- Other prominent post-pruning methods include, e.g., reduced error pruning (REP) or pessimistic error pruning (PEP).

CCP

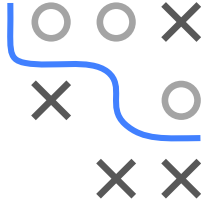
We run the CCP algorithm step-by-step with $\alpha = 1.2$:

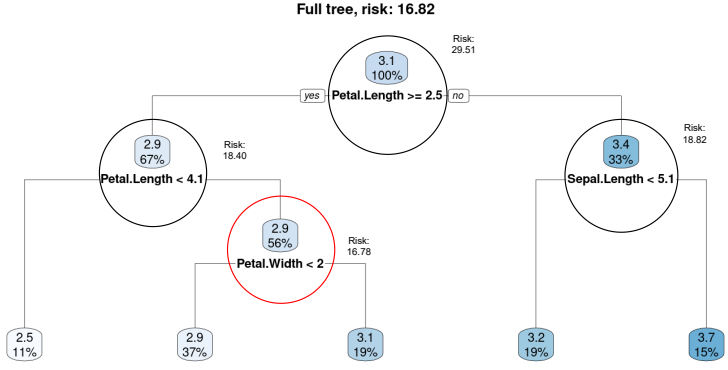




CCP

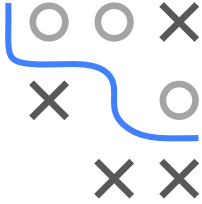
There are four possible nodes that we can eliminate to prune the tree.
We take the one replacement that results in the lowest risk (red).

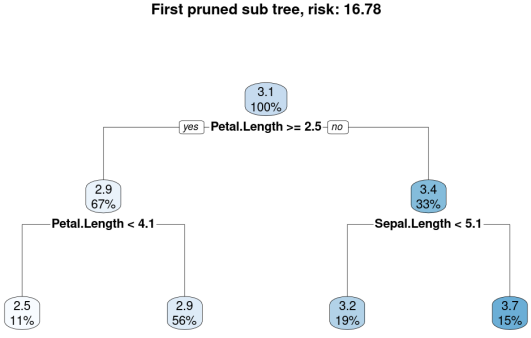




CCP

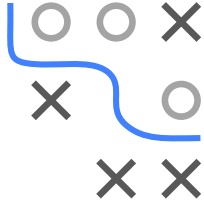
The first pruned sub tree has a lower risk than the full tree. Thus, we prefer it over the full tree.

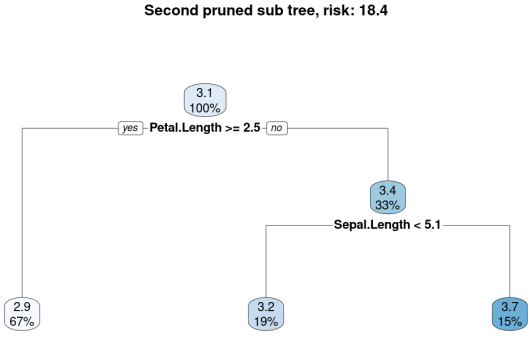




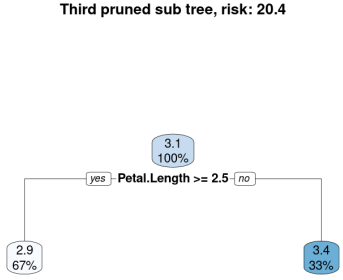
CCP

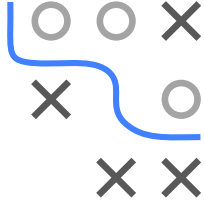
From here on, the risk increases.





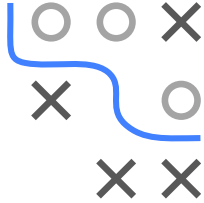
CCP


Third pruned sub tree, risk: 20.4
3.1
100%
yes
Petal.Length >= 2.5
no
2.9
67%
3.4
33%



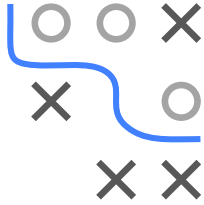
CCP

We select the first sub tree as it results in the lowest risk in the complete sequence of sub trees.

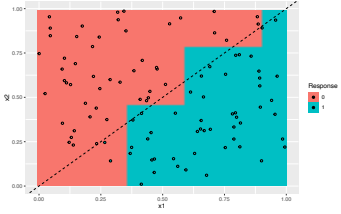


Fully pruned sub tree, risk: 29.51

Introduction to Machine Learning



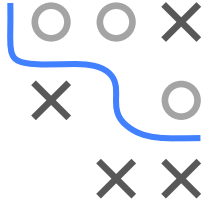
CART
Advantages & Disadvantages



Learning goals

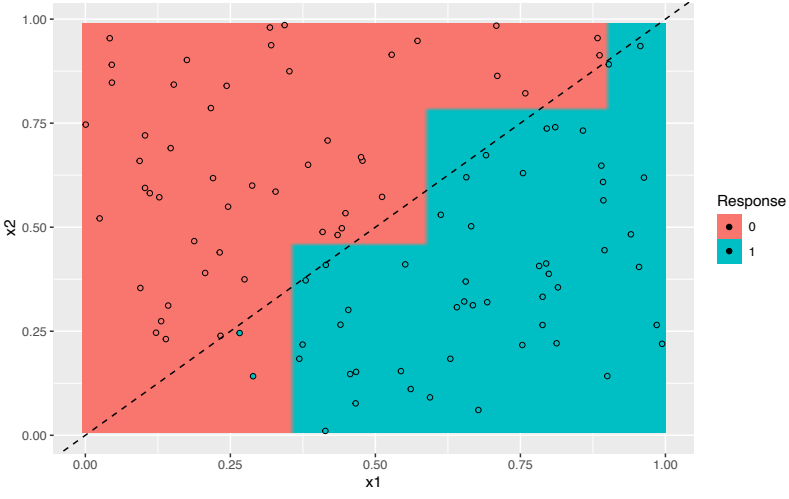
- Understand the advantages and disadvantages of CART
- Know when and where CART are applied

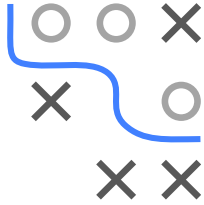
ADVANTAGES



- Fairly easy to understand, interpret and visualize.
- Not much preprocessing required:
  - Automatic handling of non-numerical features
  - Automatic handling of missing values via surrogate splits
  - No problems with outliers in features
  - Monotone transformations do not affect the model fit: scaling becomes irrelevant
- Interaction effects between features are easily possible
- Can model discontinuities and non-linearities
- Performs automatic feature selection
- Relatively fast, scales well with larger data
- Flexibility through the definition of custom split criteria or leaf-node prediction rules: clustering trees, semi-supervised trees, density estimation, etc.

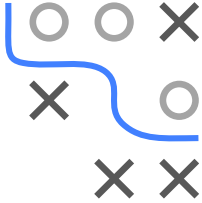
DISADVANTAGE: LINEAR DEPENDENCIES

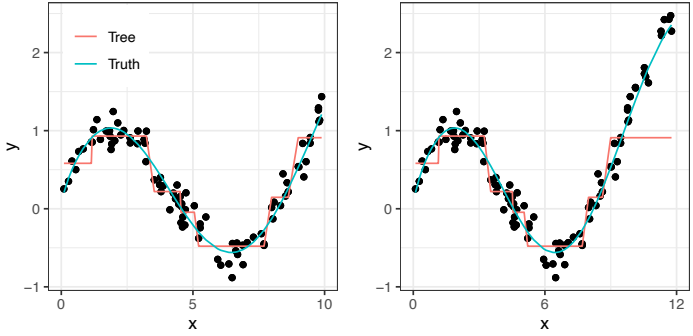




Linear dependencies must be modeled over several splits. Logistic regression would model this easily with fewer parameters.

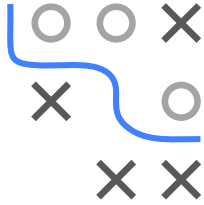
DISADVANTAGES: SMOOTH FUNCTIONS AND EXTRAPOLATION





Prediction functions of trees are never smooth as they are always step functions and do not extrapolate well beyond the training observations.

DISADVANTAGE: INSTABILITY



- High instability (variance) of the trees
- Small changes in the data may lead to very different splits/trees
- This leads to a) less trust in interpretability b) is a reason why the prediction error of trees is usually comparably high

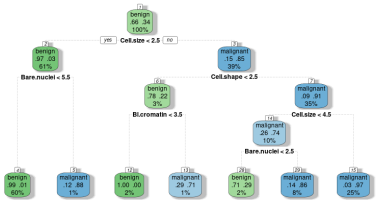
Consider the Wisconsin Breast Cancer data set with 699 observations on 9 features and binary target (“benign” vs. “malignant”). We fit two trees: (A) with the full data set and (B) where we eliminated a single observation. Results in label flip for 17 observations of the training data:

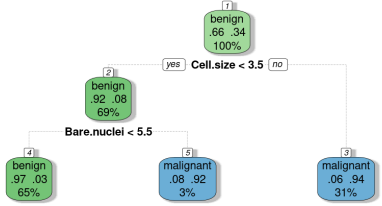
	benign	malignant
benign	445	6
malignant	11	236

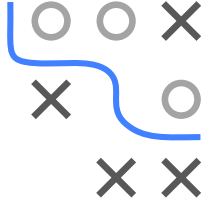
Rows: Predictions of (A), columns: Predictions of (B)

DISADVANTAGE: INSTABILITY

The resulting decision trees look very different:

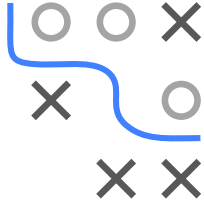




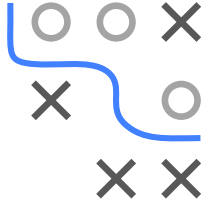


CART IN PRACTICE

- Compared to other learners CART has suboptimal predictive performance, mainly because of the problems previously shown.
- However, most disadvantages can be overcome when trained in ensembles: bagging or random forests.
- Furthermore, trees are attractive tools if an interpretable model is desired or legally required.



FURTHER TREE METHODOLOGIES

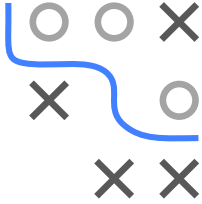


This lecture is mainly focused on Classification and Regression Trees (CART) Breiman, 1984

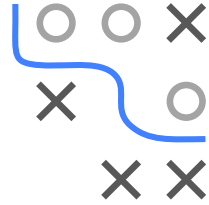
However, there are noteworthy other tree-based approaches as well:

- Automatic Interaction Detection (AID) Sonquist and Morgan, 1964 and Chi-squared Automatic Interaction Detection (CHAID) Kass, 1980: Creates all possible cross-tabulations for each categorical predictor until the best outcome is achieved and no further splitting can be performed
- C4.5 Quinlan, 1993: Not limited to binary splitting
- Unbiased Recursive Partitioning Hothorn et al., 2006: Improves the variable selection algorithm

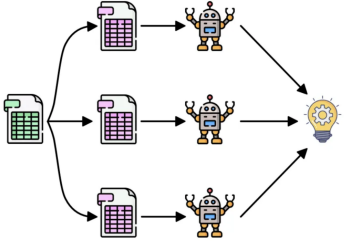
INTRODUCTION TO MACHINE LEARNING



Introduction to Machine Learning



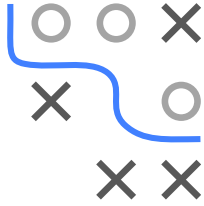
Random Forest
Bagging Ensembles



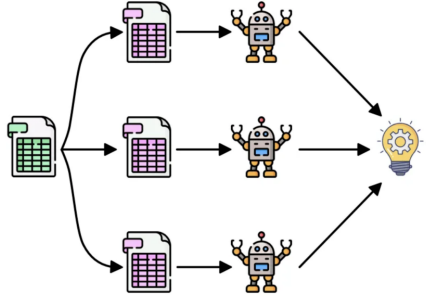
Learning goals

- Understand idea of bagging
- Be able to explain the connection between bagging and bootstrap
- Understand why bagging improves predictive performance

BAGGING



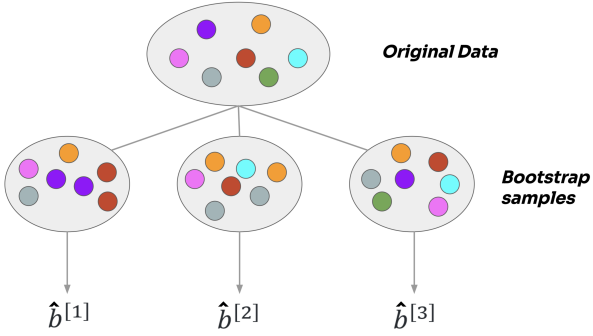
- Bagging is short for **B**ootstrap **Agg**regation
- **Ensemble method**, combines models into large “meta-model”; ensembles usually better than single **base learner**
- Homogeneous ensembles always use same BL class (e.g. CART), heterogeneous ensembles can use different classes
- Bagging is homogeneous

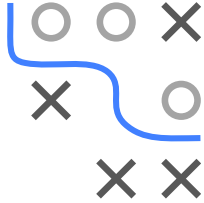


TRAINING BAGGED ENSEMBLES

Train BL on *M* **bootstrap** samples of training data $\mathcal{D}$:

- Draw *n* observations from $\mathcal{D}$ with replacement
- Fit BL on each bootstrapped data $\mathcal{D}^{[m]}$ to obtain $\hat{b}^{[m]}$

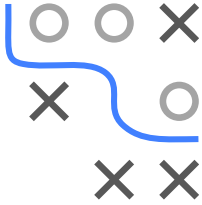


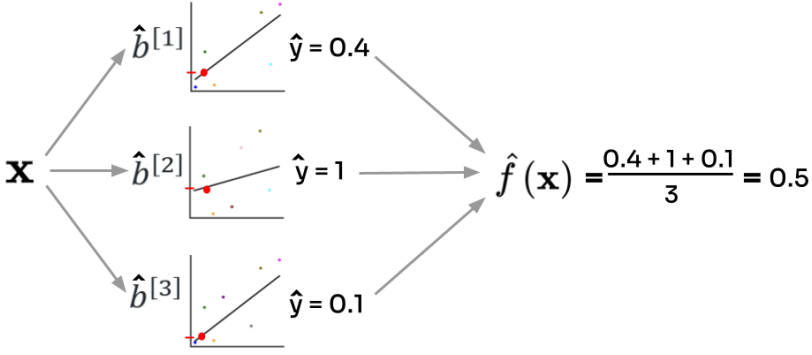


- Data sampled in one iter called “in-bag” (IB)
- Data not sampled called “out-of-bag” (OOB)

PREDICTING WITH A BAGGED ENSEMBLE

Average predictions of M fitted models for ensemble:
(here: regression)





BAGGING PSEUDO CODE

Bagging algorithm: Training

1: **Input:** Dataset $\mathcal{D}$, type of BLs, number of bootstraps M
2: **for** $m = 1 \to M$ **do**
3: Draw a bootstrap sample $\mathcal{D}^{[m]}$ from $\mathcal{D}$
4: Train BL on $\mathcal{D}^{[m]}$ to obtain model $\hat{b}^{[m]}$
5: **end for**

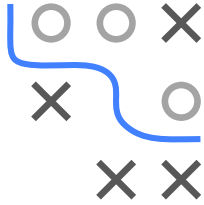
Bagging algorithm: Prediction

1: **Input:** Obs. $\mathbf{x}$, trained BLs $\hat{b}^{[m]}$ (as scores $\hat{f}^{[m]}$, hard labels $\hat{h}^{[m]}$ or probs $\hat{\pi}^{[m]}$)
2: Aggregate/Average predictions

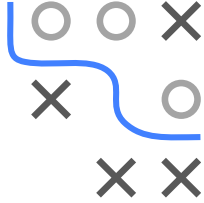
$$\hat{f}(\mathbf{x}) = \frac{1}{M} \sum_{m=1}^{M} \left(\hat{f}^{[m]}(\mathbf{x}) \right) \quad \text{(regression / decision score, use } \hat{f}_k \text{ in multi-class)}$$

$$\hat{h}(\mathbf{x}) = \underset{k \in \mathcal{Y}}{\arg\max} \sum_{m=1}^{M} \mathbb{I} \left(\hat{h}^{[m]}(\mathbf{x}) = k \right) \quad \text{(majority voting)}$$

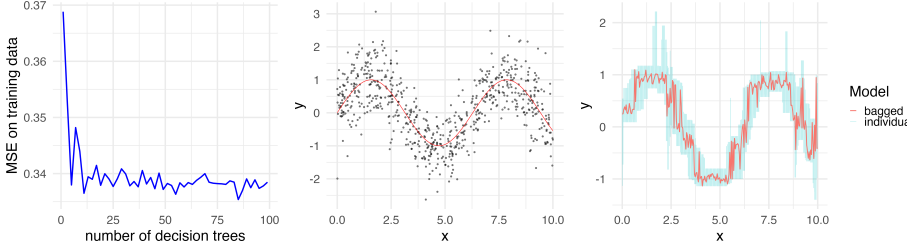
$$\hat{\pi}_k(\mathbf{x}) = \begin{cases} \frac{1}{M} \sum_{m=1}^{M} \hat{\pi}_k^{[m]}(\mathbf{x}) & \text{(probabilities through averaging)} \\ \frac{1}{M} \sum_{m=1}^{M} \mathbb{I} \left(\hat{h}^{[m]}(\mathbf{x}) = k \right) & \text{(probabilities through class frequencies)} \end{cases}$$



WHY/WHEN DOES BAGGING HELP?



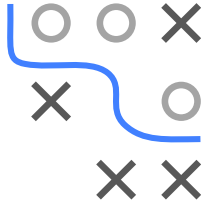
- Bagging reduces the variability of predictions by averaging the outcomes from multiple BL models
- It is particularly effective when the errors of a BL are mainly due to (random) variability rather than systematic issues

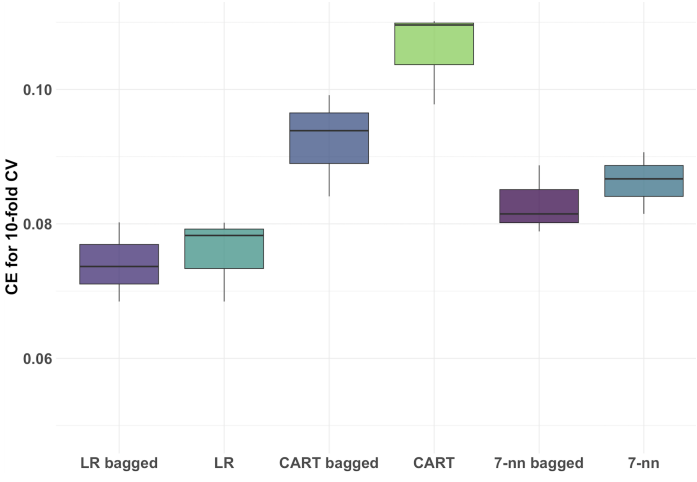


- Increasing **nr. of BLs** improves performance, up to a point, optimal ensemble size depends on inducer and data distribution

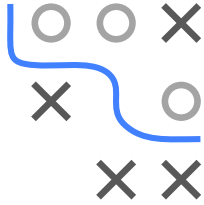
MINI BENCHMARK

Bagged ensembles with 100 BLs each on `spam`:
Bagging seems especially helpful for less stable learners like CART





Introduction to Machine Learning



Random Forest Basics

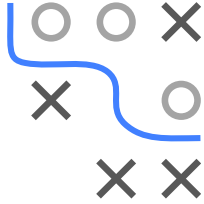
Learning goals

- Know how random forests are defined by extending the idea of bagging
- Understand general idea to decorrelate trees
- Understand effects of hyperparameters
- RFs and overfitting

MOTIVATION

CARTs offer several appealing features:

- **Interpretability**: Easy to understand and explain
- **Invariance to rank-preserving transformations**: E.g., unaffected by scaling or shifting of features
- **Versatility**: Work on categorical and numerical data
- **Robustness to missing values**: Can work with missings



Despite these benefits, CARTs are not without drawbacks:

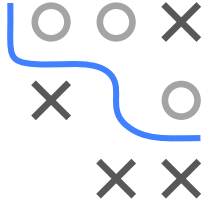
▸ Hastie, Tibshirani, and Friedman 2009

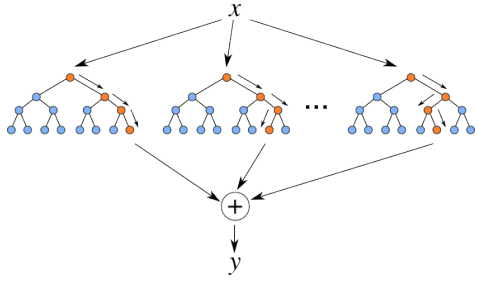
> *"Trees have one aspect that prevents them from being the ideal tool for predictive learning, namely inaccuracy."*

RANDOM FORESTS

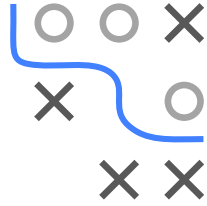
Breiman 2001

- RFs use **bagging with CARTs as BLs**
- **Random feature sampling** decorrelates the base learners
- **Fully expanded trees** further increase variability of trees





INTUITION BEHIND DECORRELATION



- Since bootstrap samples are similar, models $\hat{b}^{[m]}$ are correlated, affecting the variance of an ensemble $\hat{f}$
- We would like variance to go down linearly with ensemble size, but because of correlation we cannot really expect that
- Assuming $\text{Var}(\hat{b}^{[m]}) = \sigma^2$, $\text{Corr}(\hat{b}^{[m]}, \hat{b}^{[j]}) = \rho$, semi-formal analysis, without proper analysis of prediction error:

$$\text{Var}\left(\hat{f}\right) = \text{Var}\left(\frac{1}{M}\sum_{m=1}^{M}\hat{b}^{[m]}\right) = \frac{1}{M^2}\left(\sum_{m=1}^{M}\text{Var}(\hat{b}^{[m]}) + 2\sum_{m<j}\text{Cov}(\hat{b}^{[m]}, \hat{b}^{[j]}\right)$$

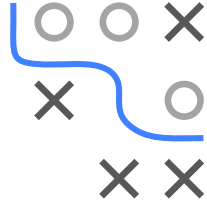
$$= \frac{1}{M^2}\left(M\sigma^2 + 2\frac{M(M-1)}{2}\rho\sigma^2\right) = (1-\rho)\frac{\sigma^2}{M} + \rho\sigma^2$$

- Ensemble variance is “convex-combo of linear-reduction and no-reduction, controlled by ρ”
- Maybe we can decorrelate trees, to reduce ensemble variance? And get less prediction error?

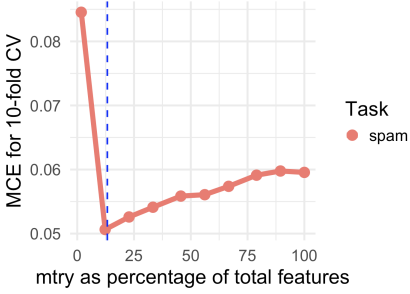
RANDOM FEATURE SAMPLING

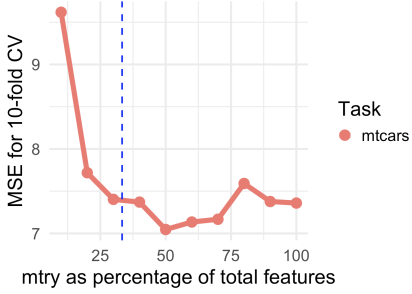
RFs decorrelate trees with a simple randomization:

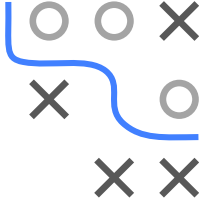
- For each node of tree, randomly draw $\texttt{mtry} \leq p$ features ($\texttt{mtry}$ = name in some implementations)
- Only consider these features for finding the best split
- Careful: Our previous analysis was simplified! The more we decorrelate by this, the more random the trees become! This also has negative effects!



EFFECT OF FEATURE SAMPLING





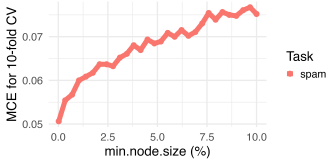


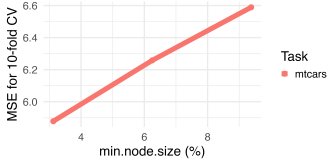
- Optimal `mtry` typically larger for regression than for classification
- Good defaults exist, but still most relevant tuning param
- Rule of thumb:
  - Classification: $\texttt{mtry} = \lfloor \sqrt{p} \rfloor$
  - Regression: $\texttt{mtry} = \lfloor p/3 \rfloor$

TREE SIZE

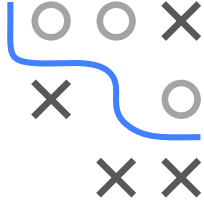
In addition to `mtry`, RFs have two other important HPs:

- Min. nr. of obs. in each decision tree node
  Default (ranger): `min.node.size = 5` ▸ Breiman 2001





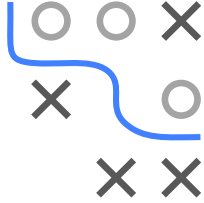
- Depth of each tree
  Default (ranger): `maxDepth` $= \infty$
- There are more alternative HPs to control depth of tree:
  minimal risk reduction, size of terminal nodes, etc.

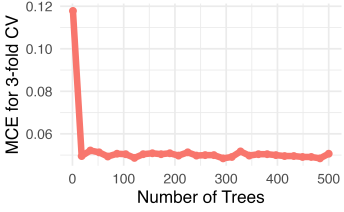


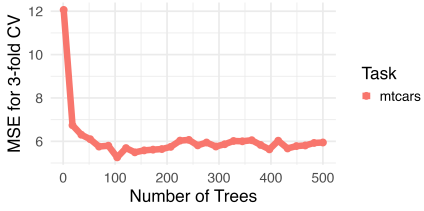
RF usually use fully expanded trees, without aggressive early stopping or pruning, to further **increase variability of each tree**. ▸ Louppe 2015

ENSEMBLE SIZE

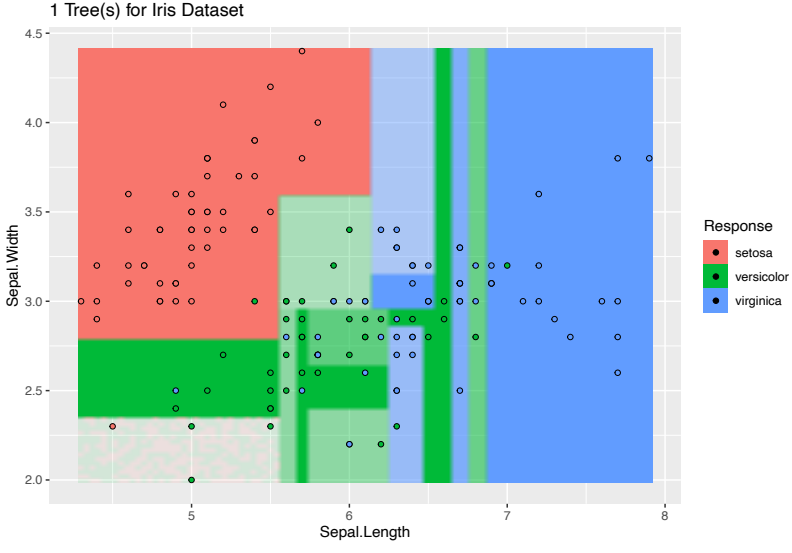
- RFs usually better if ensemble is large ▸ Breiman 2001
- But: Increases computational costs, and diminishing returns
- 100 or 500 is a sensible default
- Can also inspect the OOB error (see later)

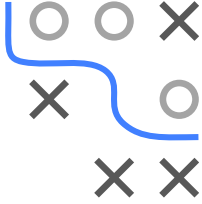




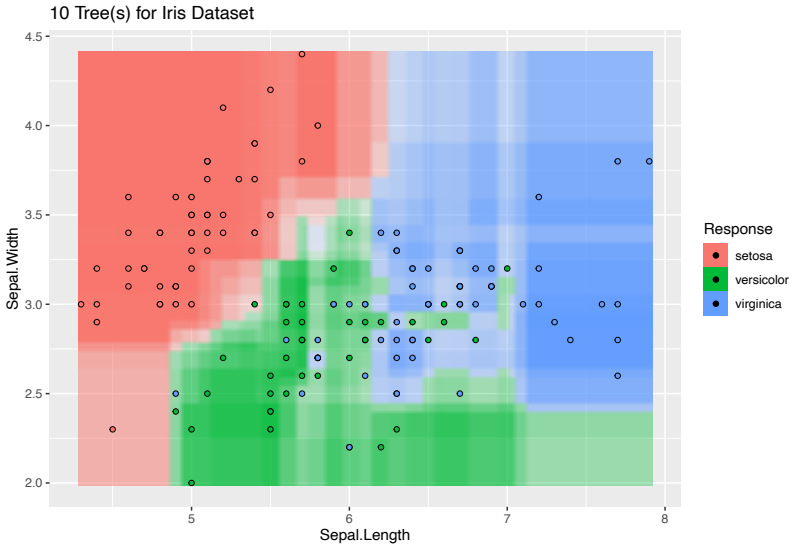


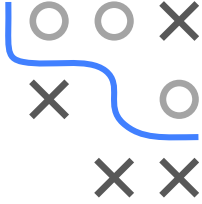
EFFECT OF ENSEMBLE SIZE



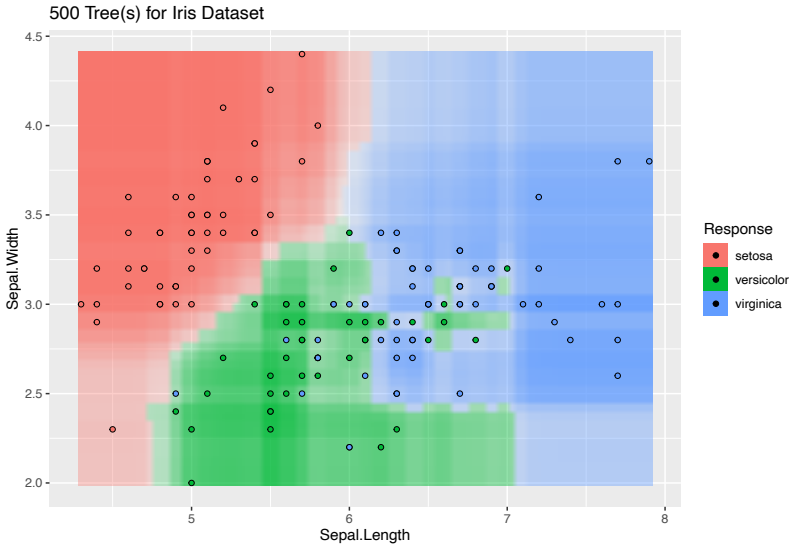


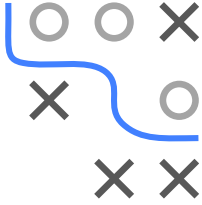
EFFECT OF ENSEMBLE SIZE





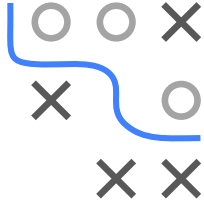
EFFECT OF ENSEMBLE SIZE

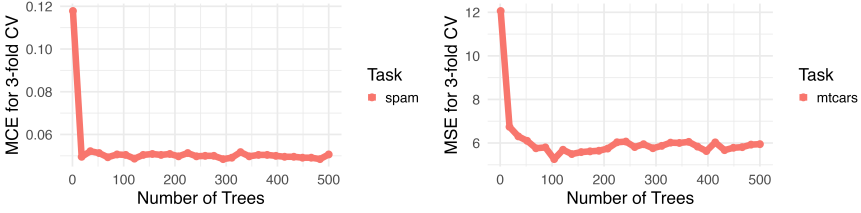




CAN RF OVERFIT? ▸ Probst and Boulesteix 2018

- Just like any other learner, RFs **can** overfit!
- However, RFs generally **less** prone to overfitting than individual CARTs.
- Overly complex trees can *still* lead to overfitting!
  If most trees capture noise, so does the RF.
- But randomization and averaging helps.

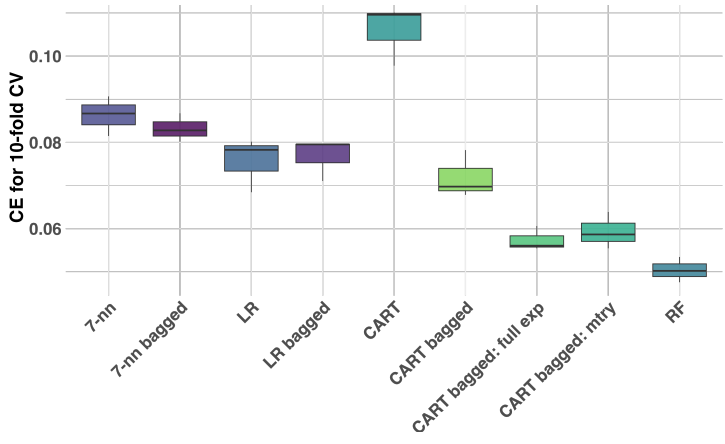


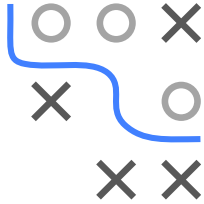


Since each tree is trained *individually and without knowledge of previously trained trees*, increasing `ntrees` generally reduces variance **without increasing the chance of overfitting!**

RF IN PRACTICE

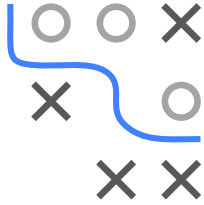
Benchmarking bagged ensembles with 100 BLs each on `spam` versus RF (`ntrees` $= 100$, `mtry` $= \sqrt{p}$, `minnode` $= 1$), we see how well RF performs!





$\Rightarrow$ RFs combine the benefits of random feature selection and fully expanded trees.

DISCUSSION



Advantages:

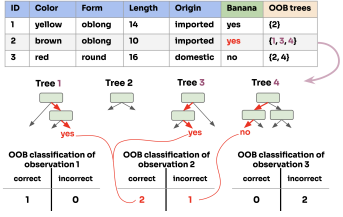
- Most advantages of trees also apply to RF: not much preprocessing required, missing value handling, etc.
- Easy to parallelize
- Often work well (enough)
- Works well on high-dimensional data
- Works well on data with irrelevant “noise” variables

Disadvantages:

- Same extrapolation problem as for trees
- Harder to interpret than trees
  (but many extra tools are nowadays available for interpreting RFs)
- Implementation can be memory-hungry
- Prediction can be computationally demanding for large ensembles

Introduction to Machine Learning

Random Forest
Out-of-Bag Error Estimate



Learning goals

- Understand the concept of out-of-bag and in-bag observations
- Learn how out-of-bag error provides an estimate of the generalization error during training

OUT-OF-BAG VS IN-BAG OBSERVATIONS

ID	Color	Form	Length	Origin	Banana
1	yellow	oblong	14	imported	yes
2	brown	oblong	10	imported	yes
3	red	round	16	domestic	no

OOB

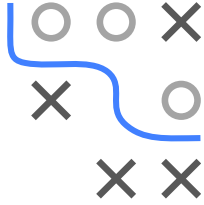
IB

predict

Bootstrapping to train tree 1

ID	Color	Form	Length	Origin	Banana
1	yellow	oblong	14	imported	yes
3	red	round	16	domestic	no
3	red	round	16	domestic	no

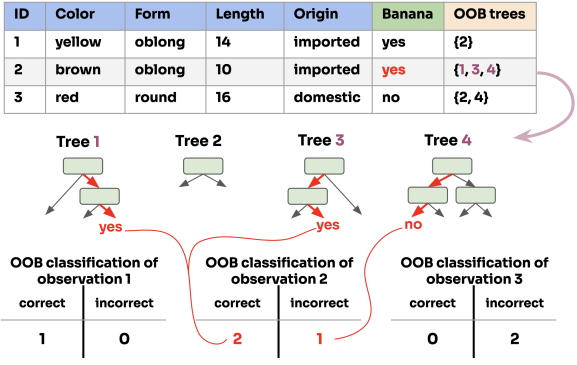
Tree 1



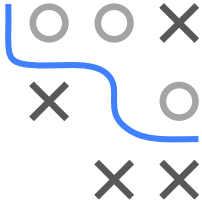
- IB observations for *m*-th bootstrap:
  $\text{IB}^{[m]} = \{i \in \{1, \ldots, n\} | \left(\mathbf{x}^{(i)}, y^{(i)}\right) \in \mathcal{D}^{[m]}\}$
- OOB observations for *m*-th bootstrap:
  $\text{OOB}^{[m]} = \{i \in \{1, \ldots, n\} | \left(\mathbf{x}^{(i)}, y^{(i)}\right) \notin \mathcal{D}^{[m]}\}$
- Nr. of trees where *i*-th observation is OOB:
  $s_{\text{OOB}}^{(i)} = \sum_{m=1}^{M} \mathbb{I}(i \in \text{OOB}^{[m]}).$

OUT-OF-BAG ERROR ESTIMATE

Predict i-th observation with all trees $\hat{b}^{[m]}$ for which it is OOB:

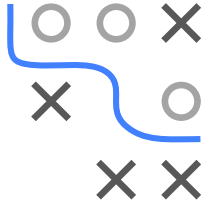


ID	Color	Form	Length	Origin	Banana	OOB trees
1	yellow	oblong	14	imported	yes	{2}
2	brown	oblong	10	imported	yes	{1, 3, 4}
3	red	round	16	domestic	no	{2, 4}



OOB prediction $\hat{\pi}^{(2)}_{\text{OOB}} = 2/3$. Evaluating all OOB predictions with some loss function L or set-based metric ρ estimates the GE.
As we do not violate the **untouched test set principle**, $\widehat{\text{GE}}$ is not *optimistically* biased.

OUT-OF-BAG ERROR PSEUDO CODE



Out-Of-Bag error estimation

1: **Input:** $\text{OOB}^{[m]}, \hat{b}^{[m]} \forall m \in \{1, \ldots, M\}$
2: **for** $i = 1 \rightarrow n$ **do**
3: Compute the ensemble OOB prediction for observation i, e.g., for regression:

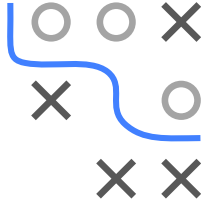
$$\hat{f}^{(i)}_{\text{OOB}} = \frac{1}{S_{\text{OOB}^{(i)}}} \sum_{m=1}^{M} \mathbb{I}(i \in \text{OOB}^{[m]}) \cdot \hat{f}^{[m]}(\mathbf{x}^{(i)})$$

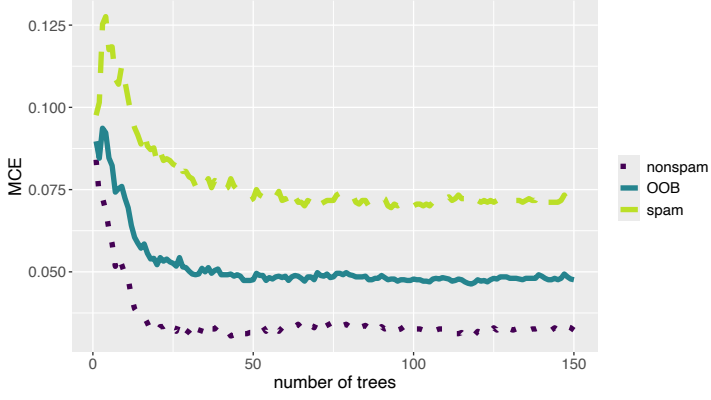
4: **end for**
5: Average losses over all observations:

$$\widehat{\text{GE}}_{\text{OOB}} = \frac{1}{n} \sum_{i=1}^{n} L(y^{(i)}, \hat{f}^{(i)}_{\text{OOB}})$$

USING THE OUT-OF-BAG ERROR ESTIMATE

- Gives us a (proper) estimator of GE, computable during training
- Can even compute this for all smaller ensemble sizes (after we fitted *M* models)

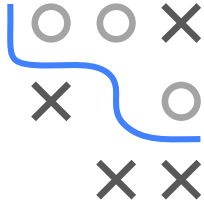




OOB ERROR: COMPARABILITY, BEST PRACTICE

OOB Size: The probability that an observation is out-of-bag (OOB) is:

$$\mathbb{P}\left(i \in \text{OOB}^{[m]}\right) = \left(1 - \frac{1}{n}\right)^n \overset{n \to \infty}{\longrightarrow} \frac{1}{e} \approx 0.37$$



$\Rightarrow$ similar to holdout or 3-fold CV (1/3 validation, 2/3 training)

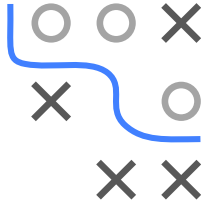
Comparability Issues:

- **OOB error** rather unique to RFs / bagging
- To compare models, we often still use CV, etc., to be consistent

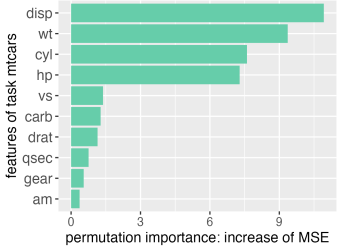
Use the OOB Error for:

- Get first impression of RF performance
- Select ensemble size
- Efficiently evaluate different RF hyperparameter configurations

Introduction to Machine Learning



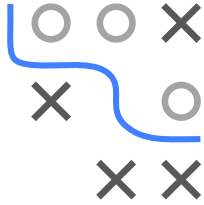
Random Forest
Feature Importance



Learning goals

- Understand that the goal of feature importance is to enhance interpretability of RF
- Understand FI based on feature permutation
- Understand FI based on improvement in splits

PERMUTATION FEATURE IMPORTANCE

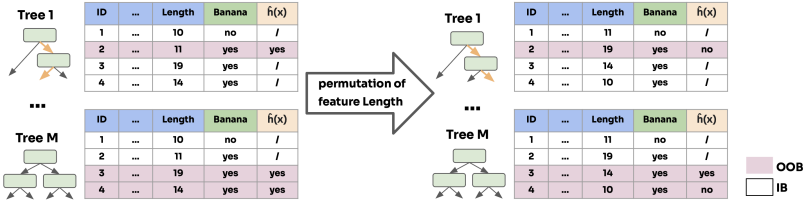


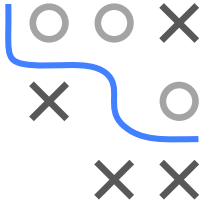
RFs improve accuracy by aggregating multiple decision trees but **lose interpretability** compared to a single tree. **Feature importance** mitigates this problem.

- How much does performance *decrease*, if feature is removed / rendered useless?
- We permute values of considered feature
- Removes association between feature and target, keeps marginal distribution
- Can obtain $\widehat{\mathrm{GE}}$ of RF (without and with permuted features) by predicting OOB data, to **efficiently compute FI during training**
- Avoids not only new models (if feature would be removed) but can already use "OOB test data" during training

ID	Color	Form	Origin	Length	Banana
1	yellow	round	domestic	10	no
2	brown	oblong	imported	11	yes
3	green	oblong	imported	19	yes
4	yellow	oblong	domestic	14	yes

PERMUTATION IMPORTANCE

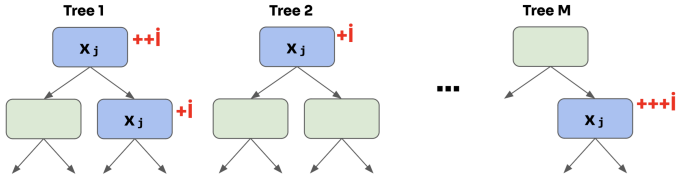


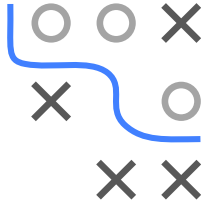


1: Calculate $\widehat{\mathrm{GE}}_{\mathrm{OOB}}$ using set-based metric ρ
2: **for** features x_j, $j = 1 \to p$ **do**
3: **for** Some statistical repetitions **do**
4: Distort feature-target relation: permute x_j with ψ_j
5: Compute all n OOB-predictions for permuted feature data, obtain all $\hat{f}^{(i)}_{\mathrm{OOB},\psi_j}$
6: Arrange predictions in $\hat{\boldsymbol{F}}_{\mathrm{OOB},\psi_j}$; Compute $\widehat{\mathrm{GE}}_{\mathrm{OOB},j} = \rho(\mathbf{y}, \hat{\boldsymbol{F}}_{\mathrm{OOB},\psi_j})$
7: Estimate importance of j-th feature: $\widehat{\mathrm{FI}}_j = \widehat{\mathrm{GE}}_{\mathrm{OOB},j} - \widehat{\mathrm{GE}}_{\mathrm{OOB}}$
8: **end for**
9: Average obtained $\widehat{\mathrm{FI}}_j$ values over reps
10: **end for**

IMPURITY IMPORTANCE

Alternative: Add up all *improvements* in splits where feature x_j is used.

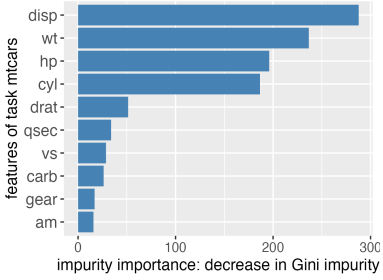


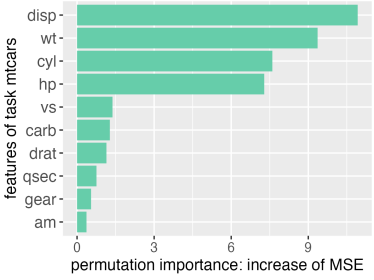


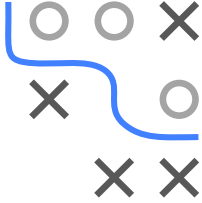
1: **for** features x_j, $j = 1 \to p$ **do**
2: **for** all models $\hat{b}^{[m]}$, $m = 1 \to M$ **do**
3: Find all splits in $\hat{b}^{[m]}$ on x_j
4: Extract improvement / risk reduction for these splits
5: Sum them up
6: **end for**
7: Add up improvements over all trees for FI of x_j
8: **end for**

IN PRACTICE / OUTLOOK

Let's compare both FI variants on `mtcars`:

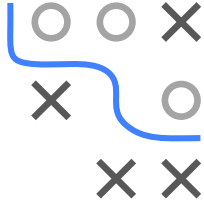




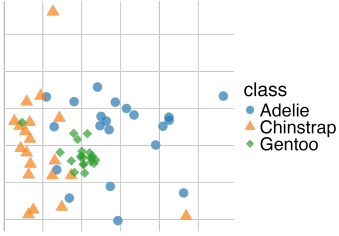


- Both methods are **biased toward features with more levels** (i.e., continuous or categoricals with many categories) ▸ Strobl et al. 2007
- More advanced versions exist
- PFI and FI have been generalized, see our lecture on IML!

Introduction to Machine Learning



Random Forest Proximities

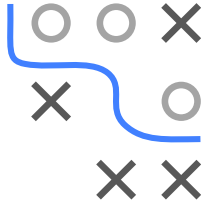


Learning goals

- Understand how RF can be used to define proximities of observations
- Know how proximities can be used for visualization, outlier detection and imputation

PROXIMITIES

RFs have built-in similarity measure for pairs of observations:

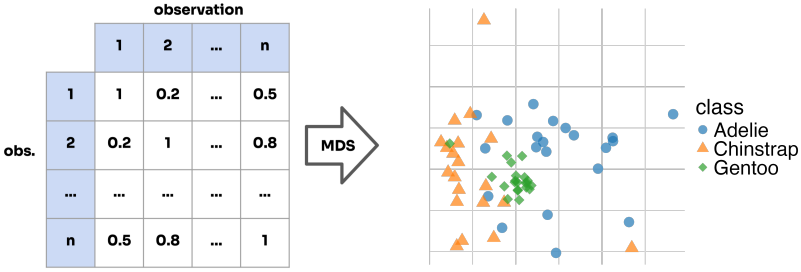


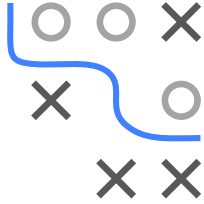
ID	Color	Form	Length
1	yellow	oblong	14

ID	Color	Form	Length
2	brown	oblong	10

- After training, push all observations through each tree
- To calculate prox $\left(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}\right)$: Percentage of how often both points are placed in **same terminal node of a tree**
- Here: prox $\left(\mathbf{x}^{(1)}, \mathbf{x}^{(2)}\right) = 2/3$
- All proximities are arranged in symmetric $n \times n$ matrix

VISUALIZING PROXIMITIES



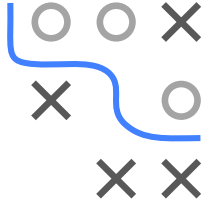


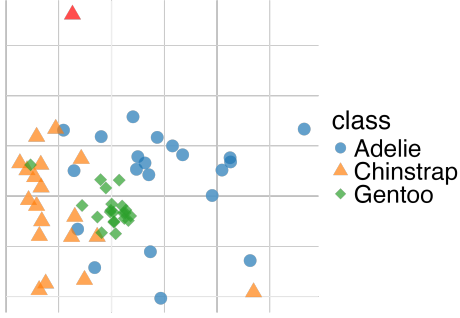
Can visualize the proximity matrix by projecting it into lower-dim. space, e.g., via multidim. scaling (might have to turn proximities into distances)

- Samples from same class usually form **identifiable clusters**
- **Offers some error-inspection**, e.g., Adelie has high within-class variance and has overlaps with other classes

OUTLIER DETECTION

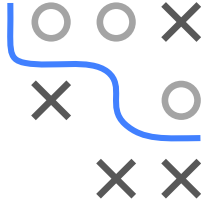
- Can also be used to **locate outliers**
- Or mislabeled points, especially in manually labeled data sets





IMPUTING MISSING DATA

ID	Color	Form	Origin	Length
1	yellow	round	domestic	14
2	brown	oblong	imported	???
3	brown	oblong	imported	19
4	???	round	domestic	14

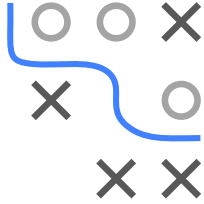


1. Replace missings per feature by median (of available values)
2. Compute proximities (NB: data has changed)
3. Replace missings in $\mathbf{x}^{(i)}$ by weighted average of non-missings; weights proportional to proximities

Steps 2 and 3 are iterated a few times.

IMPUTING MISSING DATA

ID	Color	Form	Origin	Length
1	yellow	round	domestic	14
2	brown	oblong	imported	14
3	brown	oblong	imported	19
4	brown	round	domestic	14



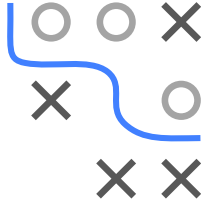
1. Replace missings per feature by median (of available values)
2. Compute proximities (NB: data has changed)
3. Replace missings in $\mathbf{x}^{(i)}$ by weighted average of non-missings; weights proportional to proximities

Steps 2 and 3 are iterated a few times.

IMPUTING MISSING DATA

ID	Color	Form	Origin	Length
1	yellow	round	domestic	14
2	brown	oblong	imported	17
3	brown	oblong	imported	19
4	brown	round	domestic	14

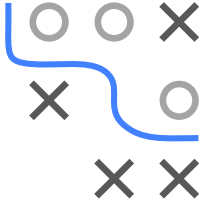
weighted average
using proximities



1. Replace missings per feature by median (of available values)
2. Compute proximities (NB: data has changed)
3. Replace missings in $\mathbf{x}^{(i)}$ by weighted average of non-missings; weights proportional to proximities

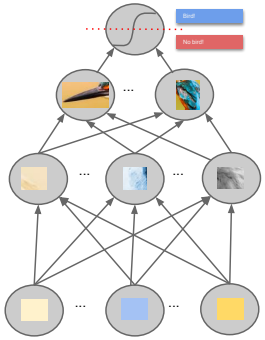
Steps 2 and 3 are iterated a few times.

INTRODUCTION TO MACHINE LEARNING



Deep Learning

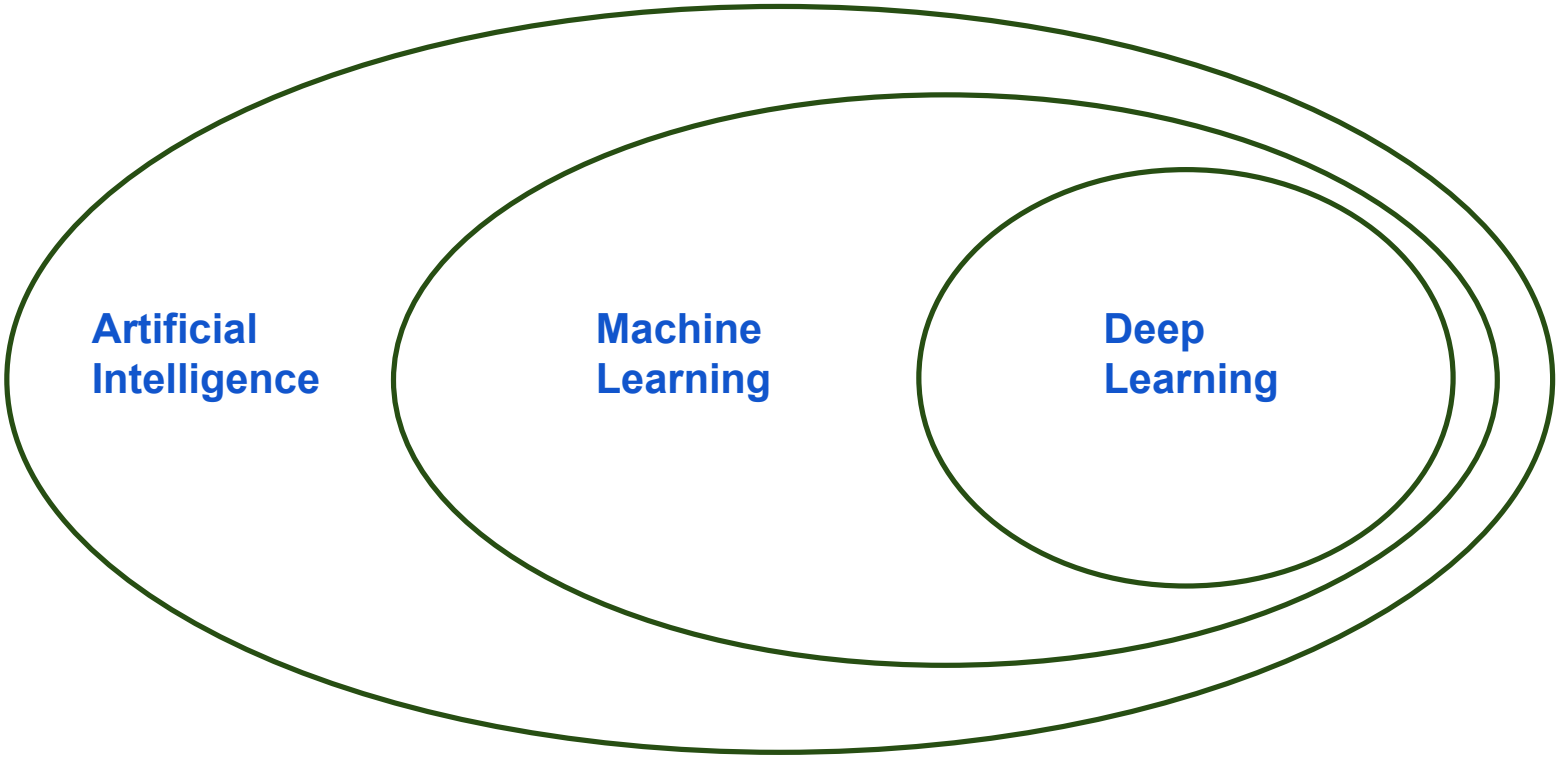
Introduction



Learning goals

- Relationship of DL and ML
- Concept of representation or feature learning
- Use-cases and data types for DL methods

WHAT IS DEEP LEARNING



- Deep learning is a subield of ML based on artificial neural networks.

DEEP LEARNING AND NEURAL NETWORKS

- Deep learning itself is not *new*:
  - Neural networks have been around since the 70s.
  - *Deep* neural networks, i.e., networks with multiple hidden layers, are not much younger.

- Why everybody is talking about deep learning now:
  1. Specialized, powerful hardware allows training of huge neural networks to push the state-of-the-art on difficult problems.
  2. Large amount of data is available.
  3. Special network architectures for image/text data.
  4. Better optimization and regularization strategies.

IMAGE CLASSIFICATION WITH NEURAL NETWORKS

“Machine learning algorithms, inspired by the brain, based on learning multiple levels of representation/abstraction.”

Y. Bengio

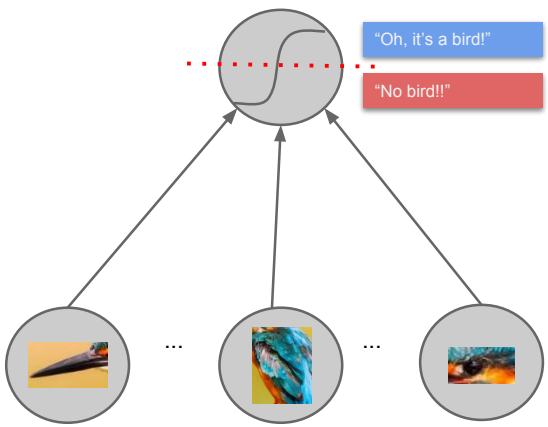


IMAGE CLASSIFICATION WITH NEURAL NETWORKS

“Machine learning algorithms, inspired by the brain, based on learning multiple levels of representation/abstraction.”

Y. Bengio

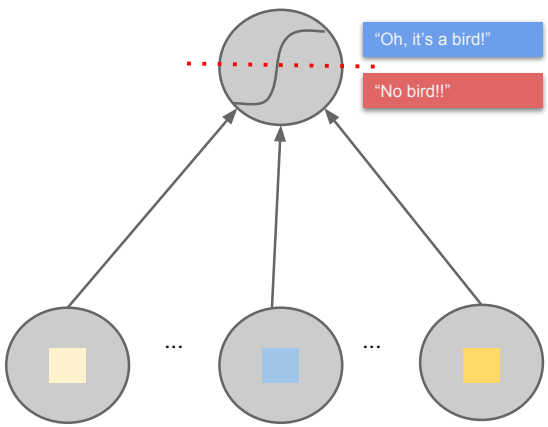


IMAGE CLASSIFICATION WITH NEURAL NETWORKS

"Machine learning algorithms, inspired by the brain, based on learning multiple levels of representation/abstraction."

Y. Bengio

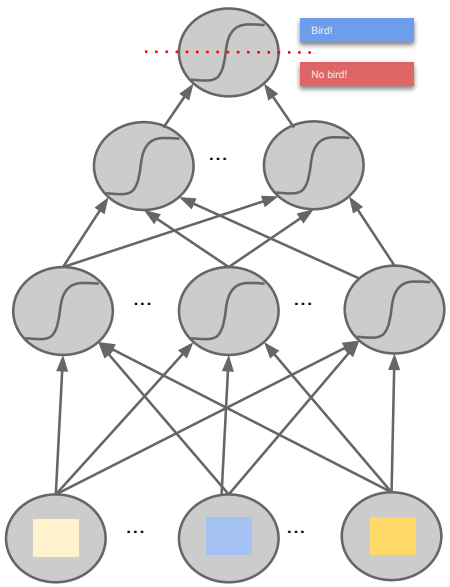
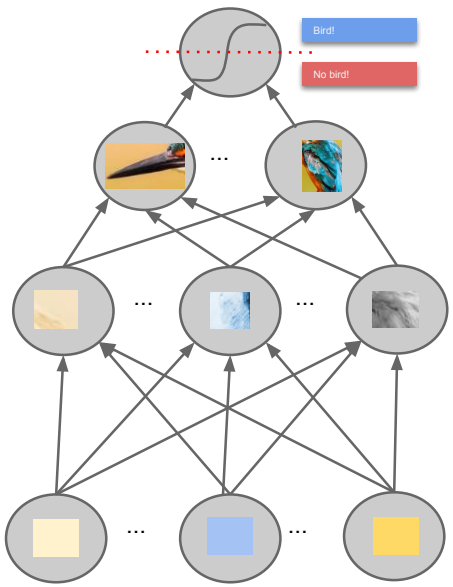


IMAGE CLASSIFICATION WITH NEURAL NETWORKS

"Machine learning algorithms, inspired by the brain, based on learning multiple levels of representation/abstraction."

Y. Bengio



POSSIBLE USE-CASES

Deep learning can be extremely valuable if the data has these properties:

- It is high dimensional.
- Each single feature itself is not very informative but only a combination of them might be.
- There is a large amount of training data.

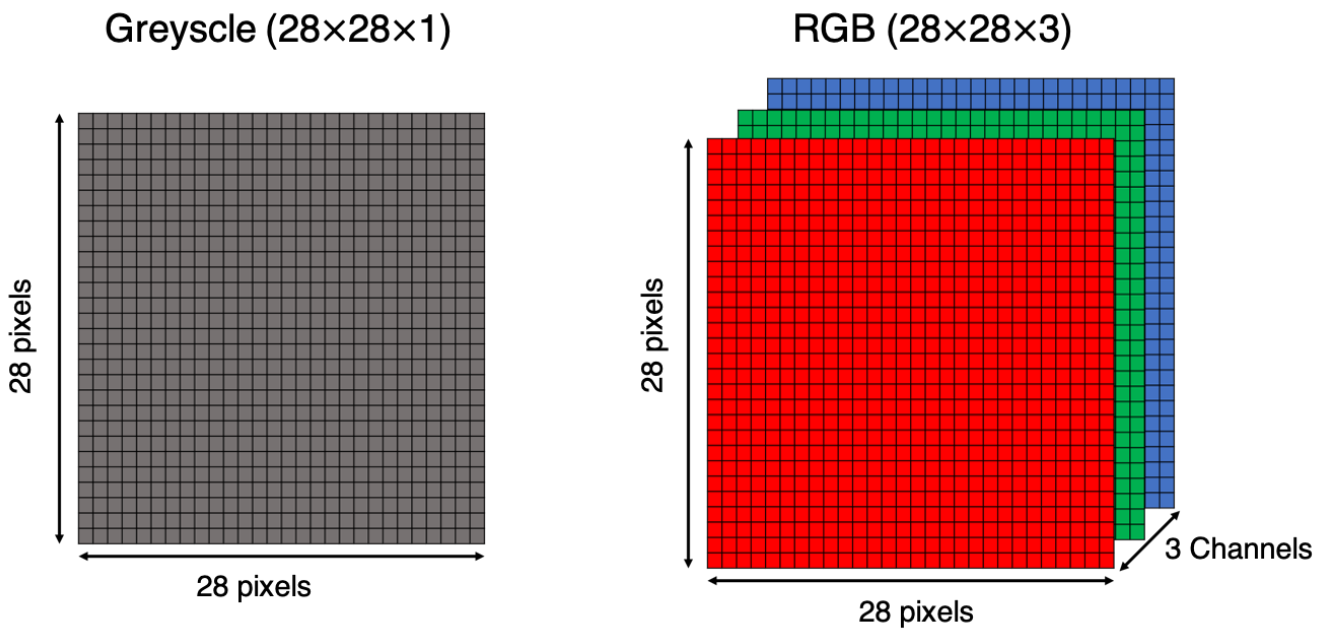
This implies that for tabular data, deep learning is rarely the correct model choice.

- Without extensive tuning, models like random forests or gradient boosting will outperform deep learning most of the time.
- One exception is data with categorical features with many levels.

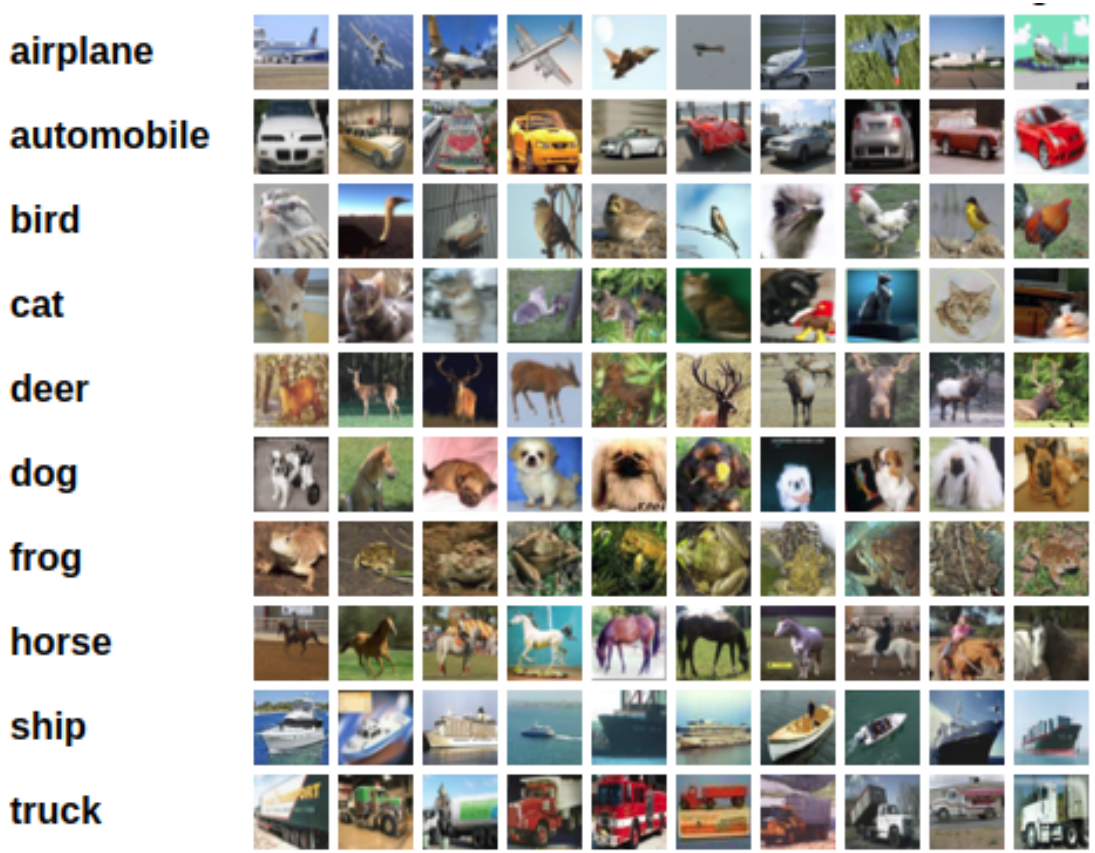
POSSIBLE USE-CASE: IMAGES

- **High Dimensional**: A color image with 255×255 (3 Colors) pixels already has 195075 features.
- **Informative**: A single pixel is not meaningful in itself.
- **Training Data**: Depending on applications huge amounts of data are available.

Architecture: **C**onvolutional **N**eural **N**etworks (CNN)



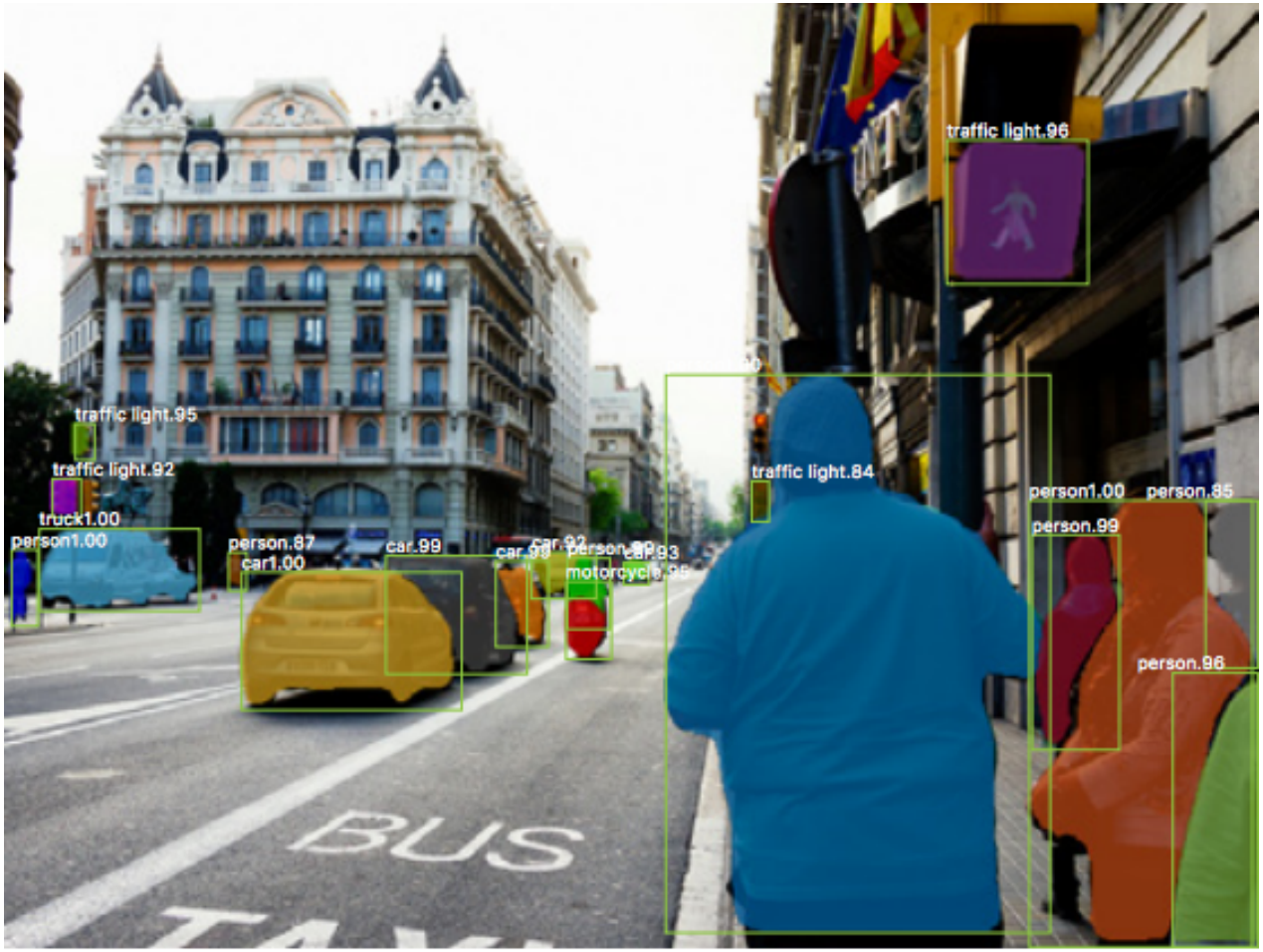
POSSIBLE USE-CASE: IMAGES



Credit: Alex Krizhevsky (2009)

Image classification tries to predict a single label for each image.
CIFAR-10 is a well-known dataset used for image classification. It consists of 60, 000 32*x*32 color images containing one of 10 object classes, with 6000 images per class.

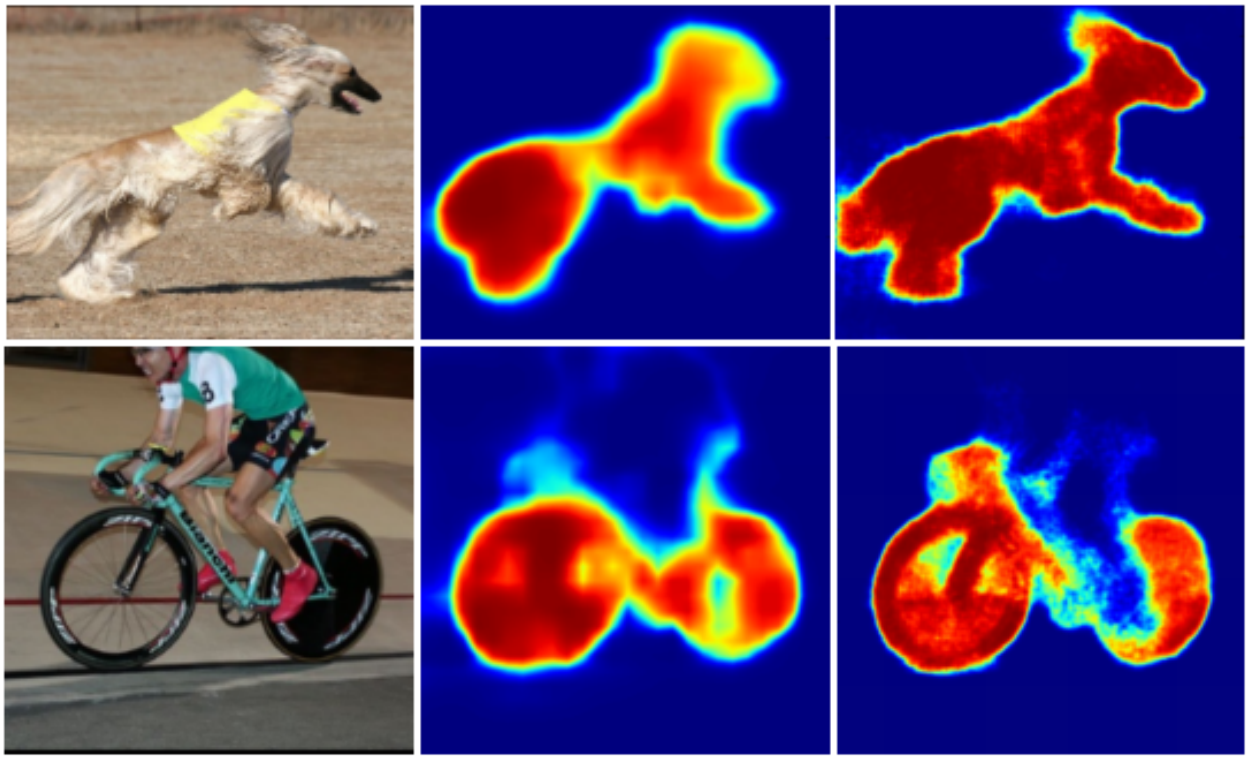
POSSIBLE USE-CASE: IMAGES



Credit: Kaiming He (2017)

Object Detection Mask R-CNN is a general framework for instance segmentation, that efficiently detects objects in an image while simultaneously generating a high-quality segmentation mask for each instance.

POSSIBLE USE-CASE: IMAGES



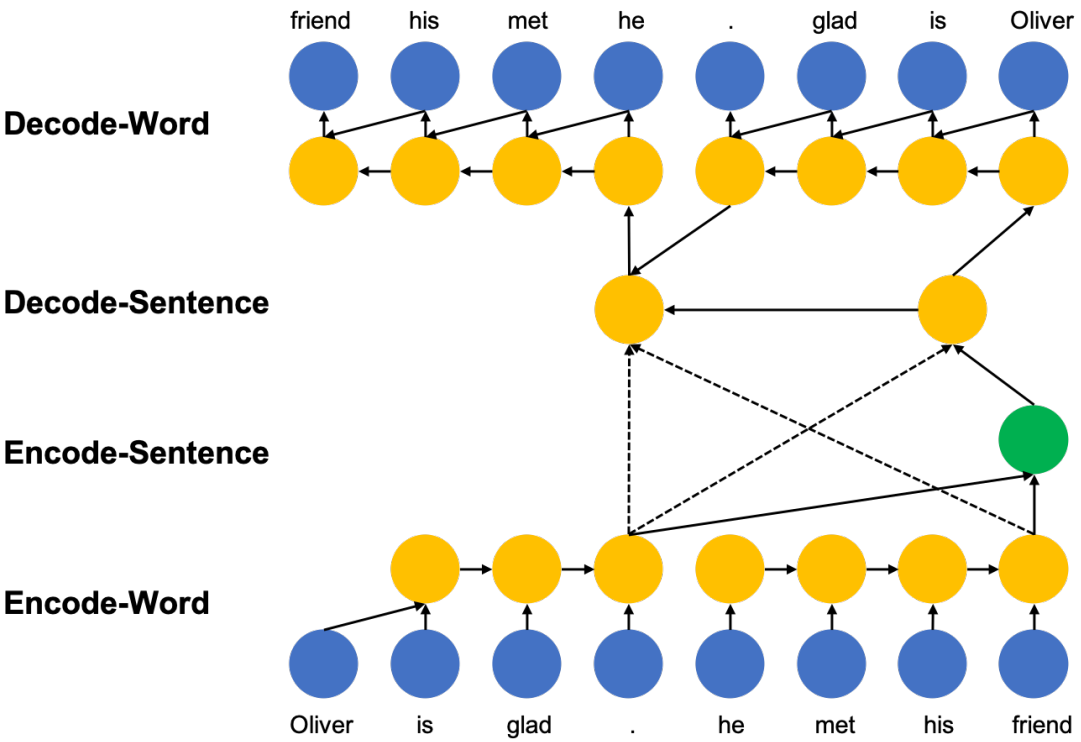
Credit: Hyeonwoo Noh (2015)

Image segmentation partitions the image into (multiple) segments.

POSSIBLE USE-CASE: TEXT

- **High Dimensional**: Each word can be a single feature (300000 words in the German language).
- **Informative**: A single word does not provide much context.
- **Training Data**: Huge amounts of text data available.

Architecture: **R**ecurrent **N**eural **N**etworks (RNN)



POSSIBLE USE-CASE: TEXT CLASSIFICATION

Positive

Great job! Your customer support is fantastic.

Neutral

Not bad, but it should be improved in the future.

Negative

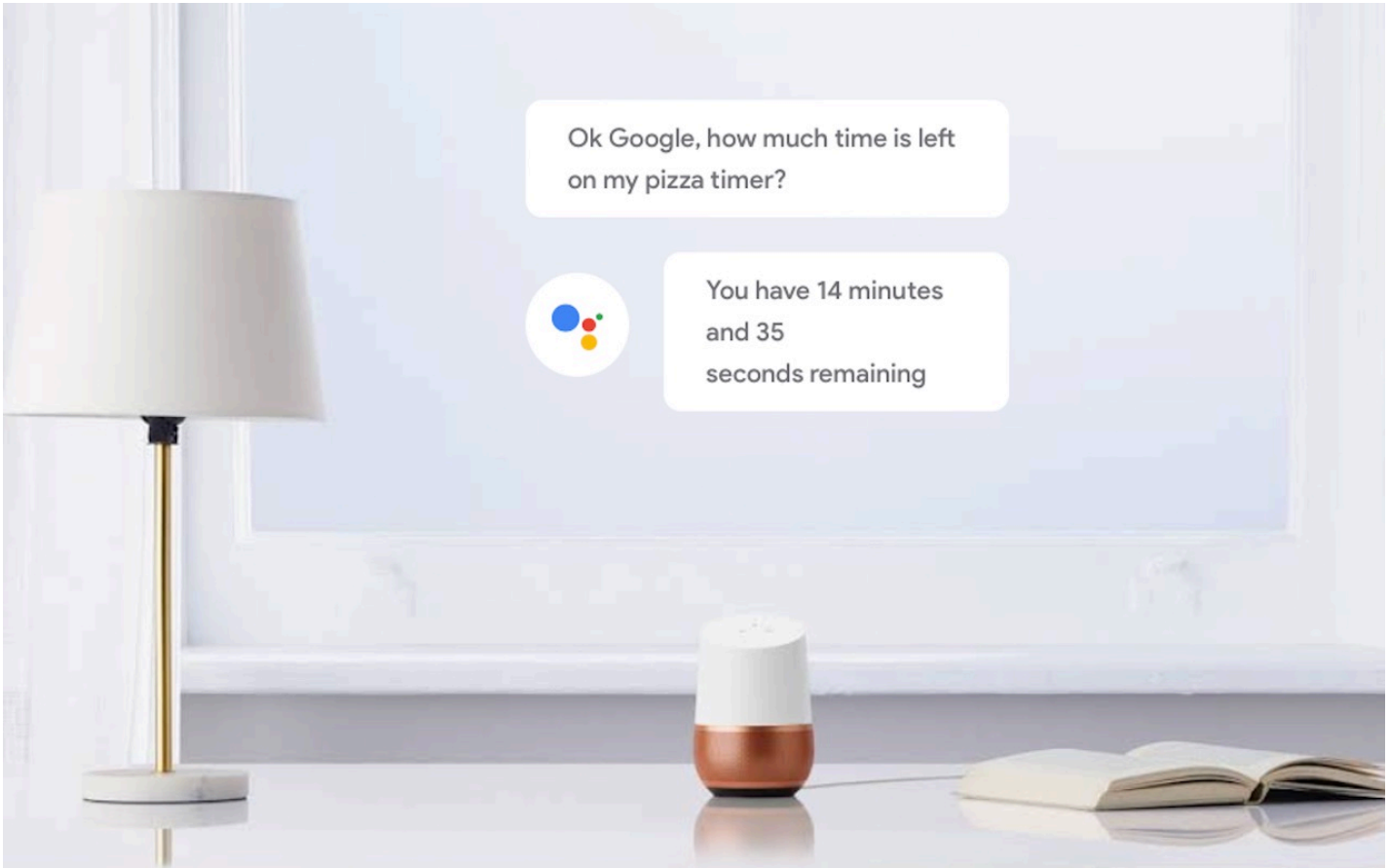
The worst customer service I have ever seen.

Sentiment Analysis is the application of natural language processing to systematically identify the emotional and subjective information in texts.

POSSIBLE USE-CASE: TEXT

Machine Translation (e.g. google translate) Neural machine translation exploits neural networks to predict the likelihood of a sequence of words, typically modeling entire sentences in a single integrated model.

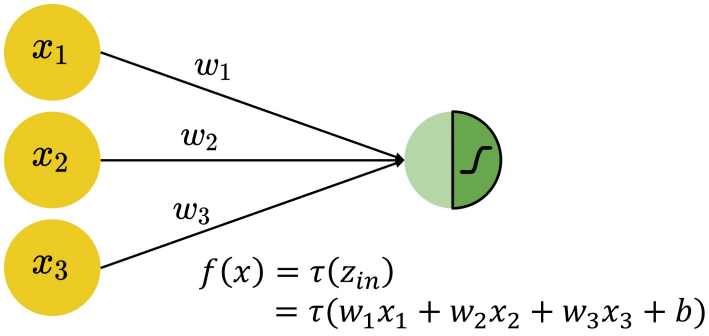
APPLICATIONS OF DEEP LEARNING: SPEECH



Speech Recognition and Generation (e.g. google assistant) Neural network extracts features from audio data for downstream tasks, e.g., to classify emotions in speech.

Deep Learning

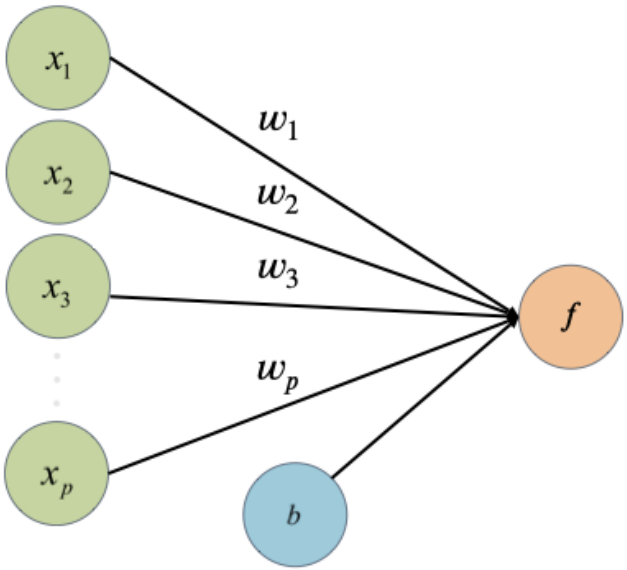
Single Neuron / Perceptron



Learning goals

- Graphical representation of a single neuron
- Affine transformations and non-linear activation functions
- Hypothesis spaces of a single neuron
- Typical loss functions

A SINGLE NEURON



Perceptron with **input features** $x_1, x_2, \ldots, x_p$, **weights** $w_1, w_2, \ldots, w_p$, **bias term** b, and **activation function** τ.

- The perceptron is a single artificial neuron and the basic computational unit of neural networks.
- It is a weighted sum of input values, transformed by τ:

$$f(x) = \tau(w_1 x_1 + \ldots + w_p x_p + b) = \tau(\mathbf{w}^T \mathbf{x} + b)$$

A SINGLE NEURON

Activation function τ: a single neuron represents different functions depending on the choice of activation function.

- The identity function gives us the simple **linear regression**:

$$f(x) = \tau(\mathbf{w}^T\mathbf{x}) = \mathbf{w}^T\mathbf{x}$$

- The logistic function gives us the **logistic regression**:

$$f(x) = \tau(\mathbf{w}^T\mathbf{x}) = \frac{1}{1 + \exp(-\mathbf{w}^T\mathbf{x})}$$

A SINGLE NEURON

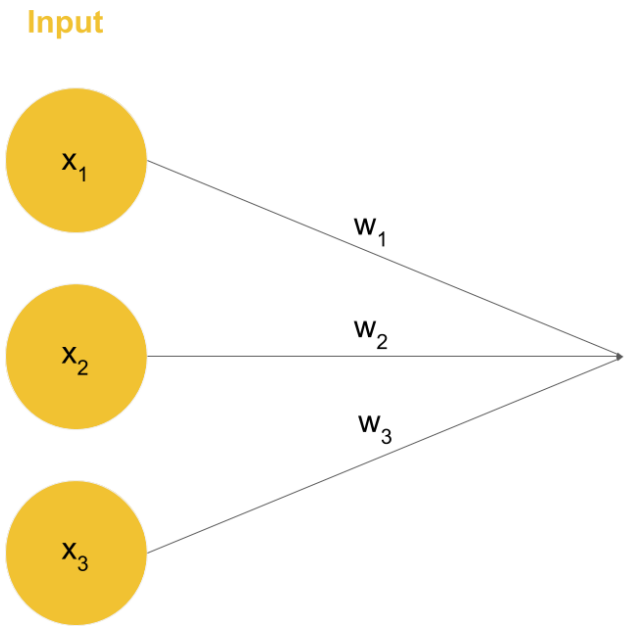
We consider a perceptron with 3-dimensional input, i.e.
$f(\mathbf{x}) = \tau(w_1 x_1 + w_2 x_2 + w_3 x_3 + b)$.

- Input features **x** are represented by nodes in the "input layer".

- In general, a p-dimensional input vector **x** will be represented by p nodes in the input layer.

A SINGLE NEURON

- Weights **w** are connected to edges from the input layer.



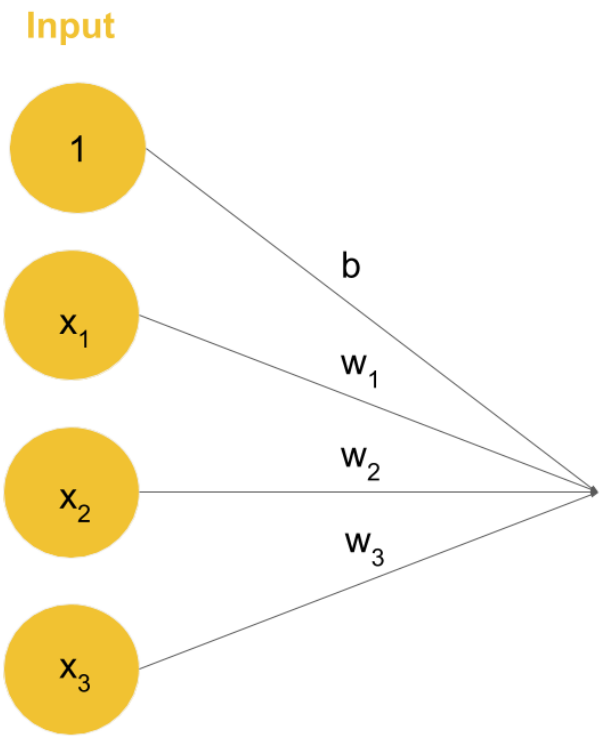
- The bias term *b* is implicit here. It is often not visualized as a separate node.

A SINGLE NEURON

For an explicit graphical representation, we do a simple trick:

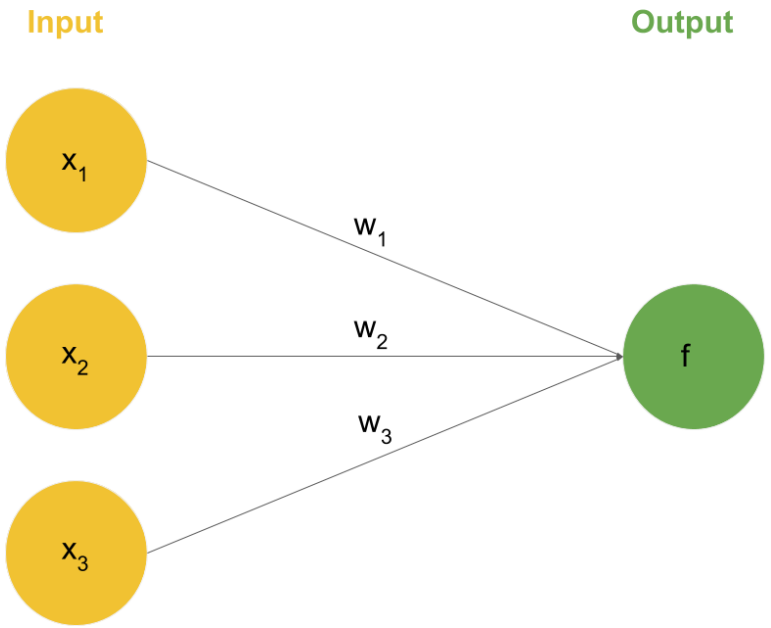
- Add a constant feature to the inputs $\tilde{\mathbf{x}} = (1, x_1, ..., x_p)^T$
- and absorb the bias into the weight vector $\tilde{\boldsymbol{w}} = (b, w_1, ..., w_p)$.

The graphical representation is then:



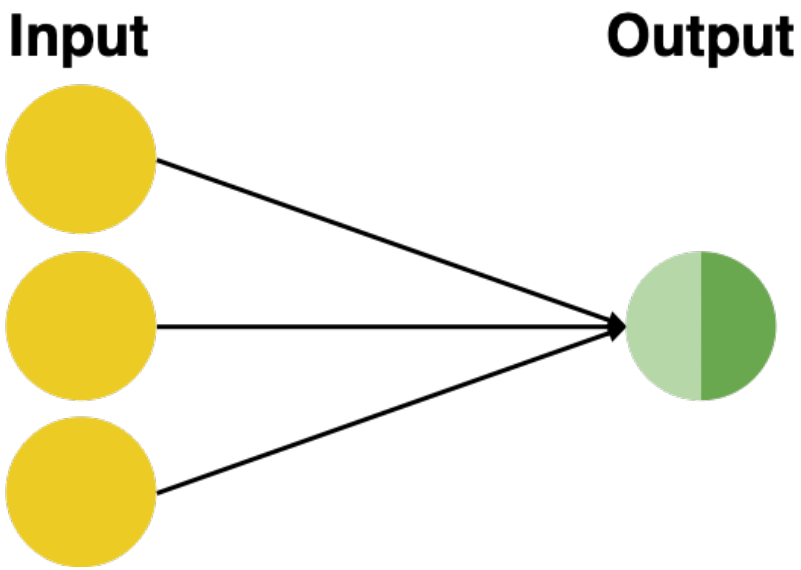
A SINGLE NEURON

- The computation $\tau(w_1 x_1 + w_2 x_2 + w_3 x_3 + b)$ is represented by the neuron in the "output layer".



A SINGLE NEURON

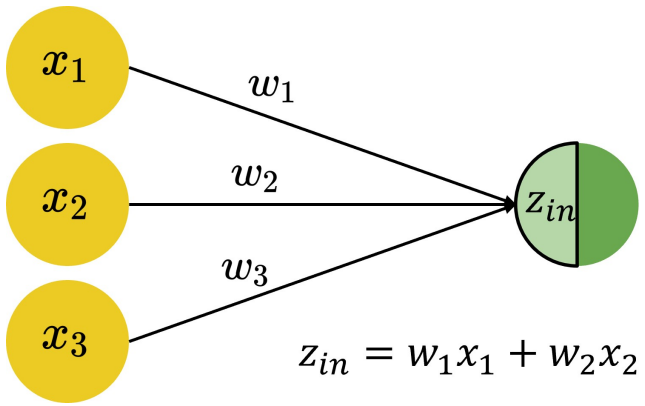
- You can picture the input vector being "fed" to neurons on the left followed by a sequence of computations performed from left to right. This is called a **forward pass**.



A SINGLE NEURON

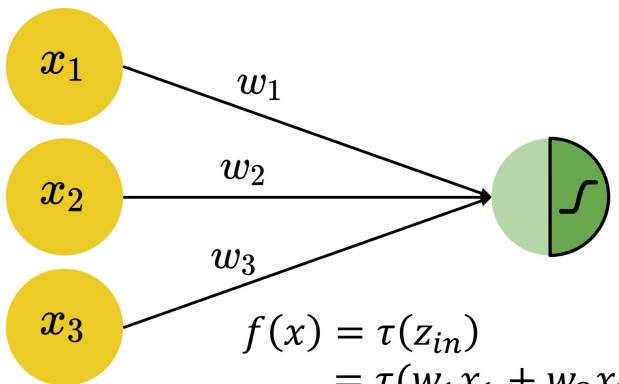
A neuron performs a 2-step computation:

1. **Affine Transformation:** weighted sum of inputs plus bias.



$$z_{in} = w_1 x_1 + w_2 x_2 + w_3 x_3 + b$$

2. **Non-linear Activation:** a non-linear transformation applied to the weighted sum.



$$f(x) = \tau(z_{in})$$
$$= \tau(w_1 x_1 + w_2 x_2 + w_3 x_3 + b)$$

A SINGLE NEURON: HYPOTHESIS SPACE

- The hypothesis space that is formed by single neuron is

$$\mathcal{H} = \left\{ f : \mathbb{R}^p \to \mathbb{R} \;\middle|\; f(\mathbf{x}) = \tau\left(\sum_{j=1}^{p} w_j x_j + b\right), \mathbf{w} \in \mathbb{R}^p, b \in \mathbb{R} \right\}.$$

- If τ is the logistic sigmoid or identity function, $\mathcal{H}$ corresponds to the hpothesis space of logistic or linear regression, respectively.

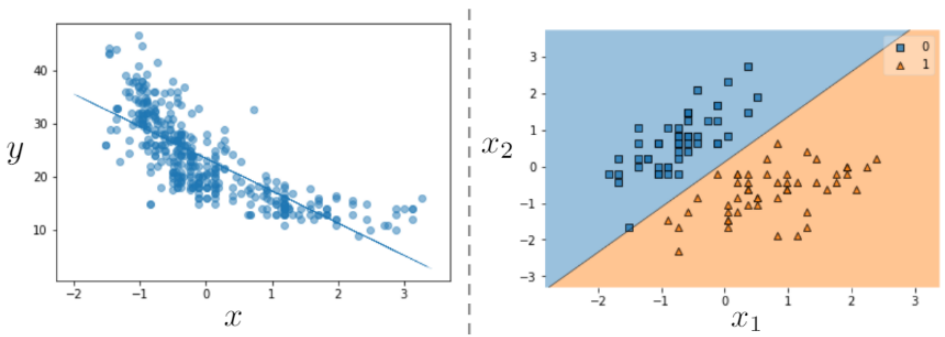


Figure: *Left*: A regression line learned by a single neuron. *Right*: A decision-boundary learned by a single neuron in a binary classification task.

A SINGLE NEURON: OPTIMIZATION

- To optimize this model, we minimize the empirical risk

$$\mathcal{R}_{\text{emp}} = \frac{1}{n} \sum_{i=1}^{n} L\left(y^{(i)}, f\left(\mathbf{x}^{(i)}\right)\right),$$

  where $L(y, f(\mathbf{x}))$ is a loss function. It compares the network's predictions $f(\mathbf{x})$ to the ground truth y.
- For regression, we typically use the L2 loss (rarely L1):

$$L(y, f(\mathbf{x})) = \frac{1}{2}(y - f(\mathbf{x}))^2$$

- For binary classification, we typically apply the cross entropy loss (also known as Bernoulli loss):

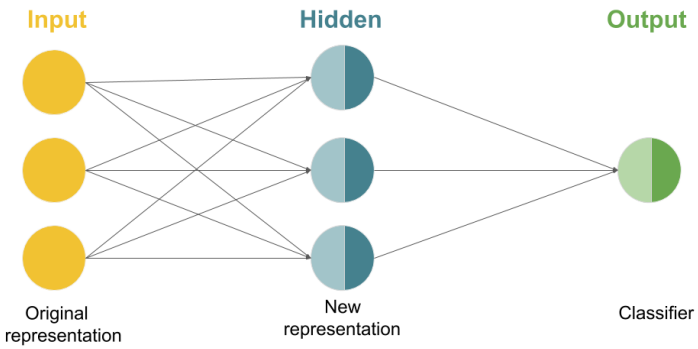
$$L(y, f(\mathbf{x})) = -(y \log f(\mathbf{x}) + (1 - y) \log(1 - f(\mathbf{x})))$$

A SINGLE NEURON: OPTIMIZATION

- For a single neuron and both choices of τ the loss function is convex.
- The global optimum can be found with an iterative algorithm like gradient descent.
- A single neuron with logistic sigmoid function trained with the Bernoulli loss yields the same result as logistic regression when trained until convergence.
- Note: In the case of regression and the L2-loss, the solution can also be found analytically using the “normal equations”. However, in other cases a closed-form solution is usually not available.

Deep Learning

Single hidden layer neural networks



Learning goals

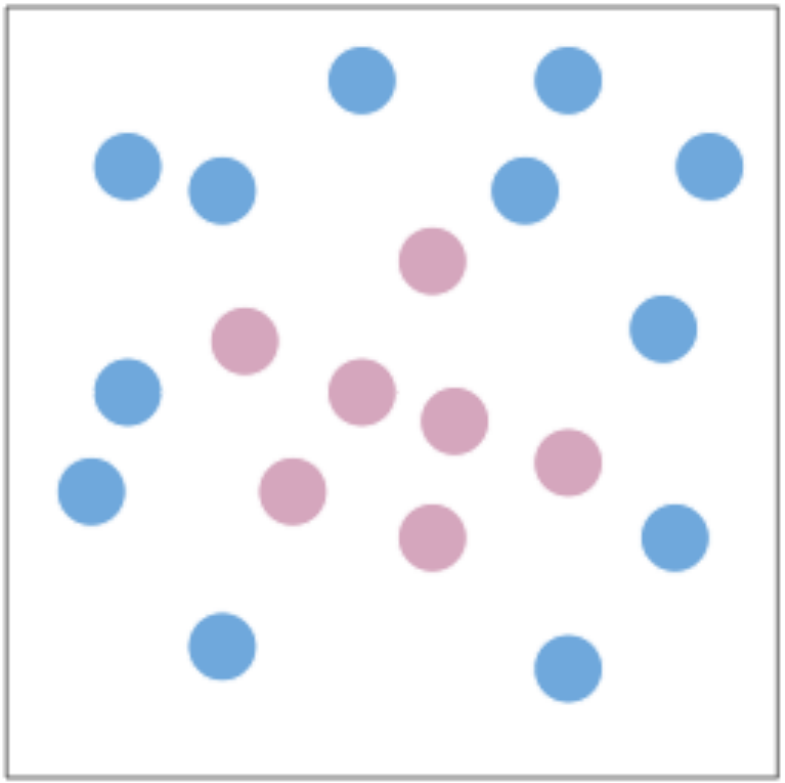
- Architecture of single hidden layer neural networks
- Representation learning/ understanding the advantage of hidden layers
- Typical (non-linear) activation functions

MOTIVATION

- The graphical way of representing simple functions/models, like logistic regression. Why is that useful?
- Because individual neurons can be used as building blocks of more complicated functions.
- Networks of neurons can represent extremely complex hypothesis spaces.
- Most importantly, it allows us to define the “right” kinds of hypothesis spaces to learn functions that are common in our universe in a data-efficient way (see Lin, Tegmark et al. 2016).

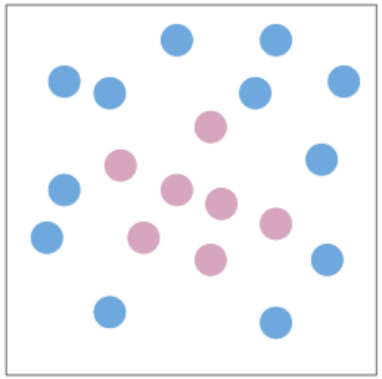
MOTIVATION

Can a single neuron perform binary classification of these points?



MOTIVATION

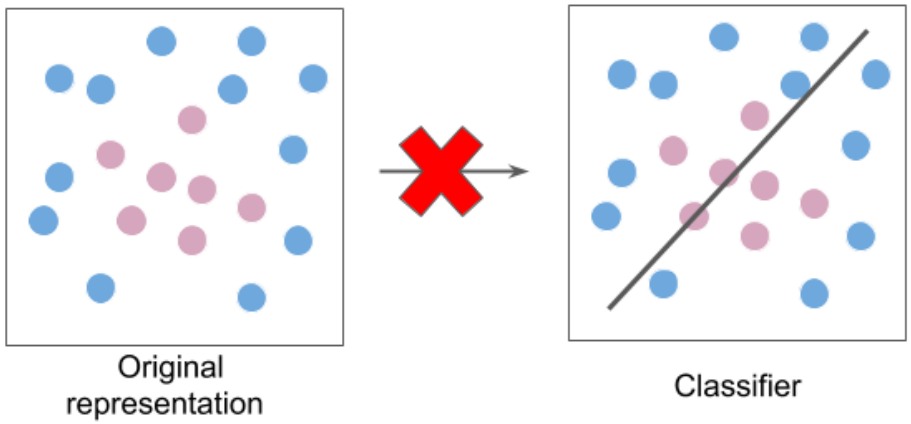
- As a single neuron is restricted to learning only linear decision boundaries, its performance on the following task is quite poor:



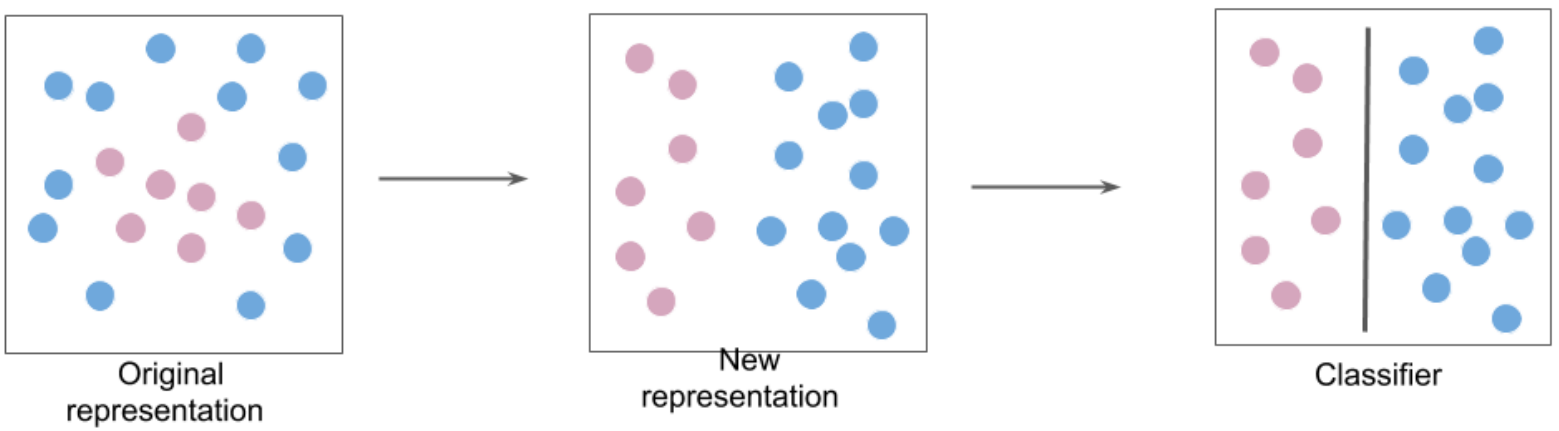
- However, the neuron can easily separate the classes if the original features are transformed (e.g., from Cartesian to polar coordinates):

MOTIVATION

- Instead of classifying the data in the original representation,

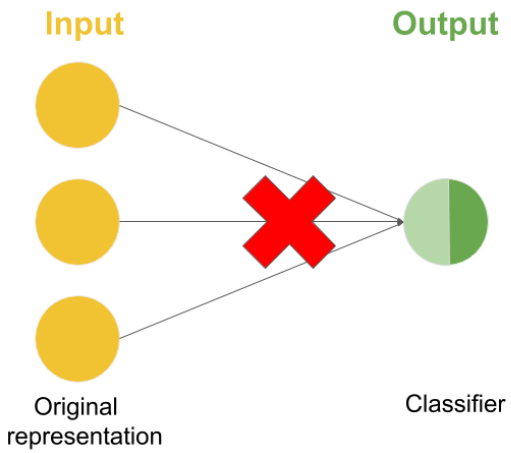


- we classify it in a new feature space.

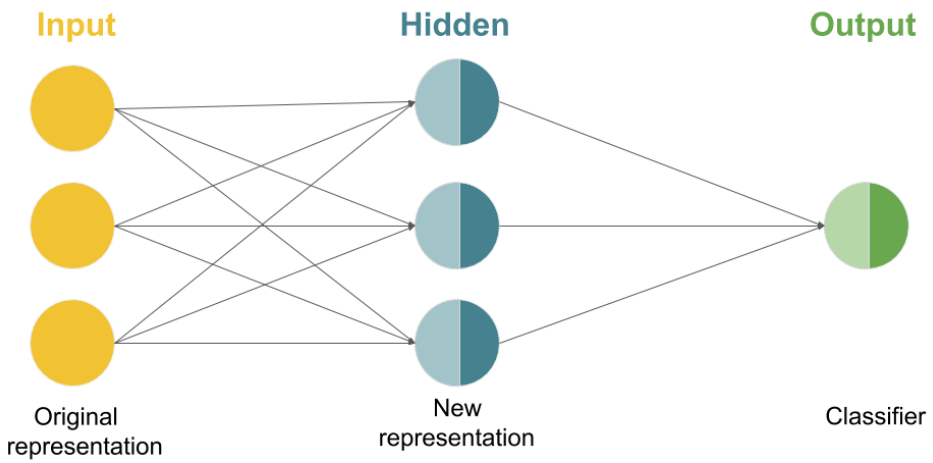


MOTIVATION

- Analogously, instead of a single neuron,



- we use more complex networks.



REPRESENTATION LEARNING

- It is *very* critical to feed a classifier the “right” features in order for it to perform well.

- Before deep learning took off, features for tasks like machine vision and speech recognition were “hand-designed” by domain experts. This step of the machine learning pipeline is called **feature engineering**.

- DL automates feature engineering. This is called **representation learning**.

SINGLE HIDDEN LAYER NETWORKS

Single neurons perform a 2-step computation:

1. **Affine Transformation:** a weighted sum of inputs plus bias.
2. **Activation:** a non-linear transformation on the weighted sum.

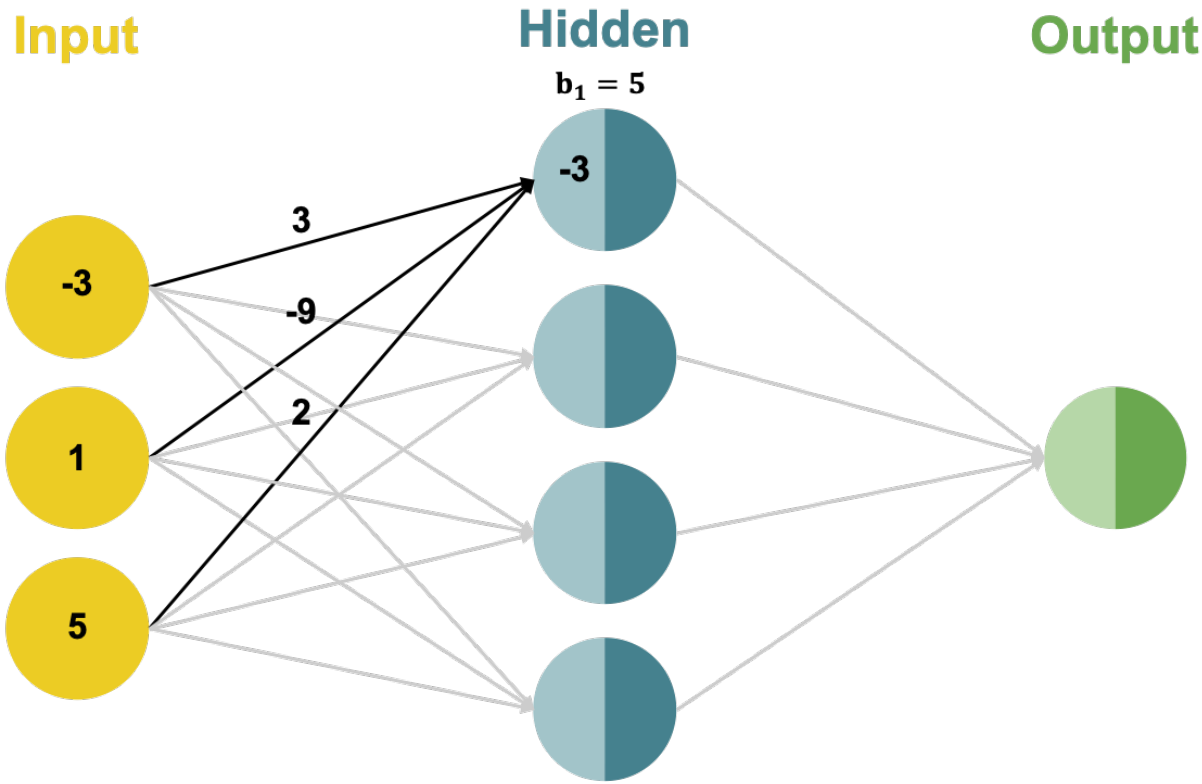
Single hidden layer networks consist of two layers (without input layer):

1. **Hidden Layer:** having a set of neurons.
2. **Output Layer:** having one or more output neurons.

- Multiple inputs are simultaneously fed to the network.
- Each neuron in the hidden layer performs a 2-step computation.
- The final output of the network is then calculated by another 2-step computation performed by the neuron in the output layer.

SINGLE HIDDEN LAYER NETWORKS: EXAMPLE

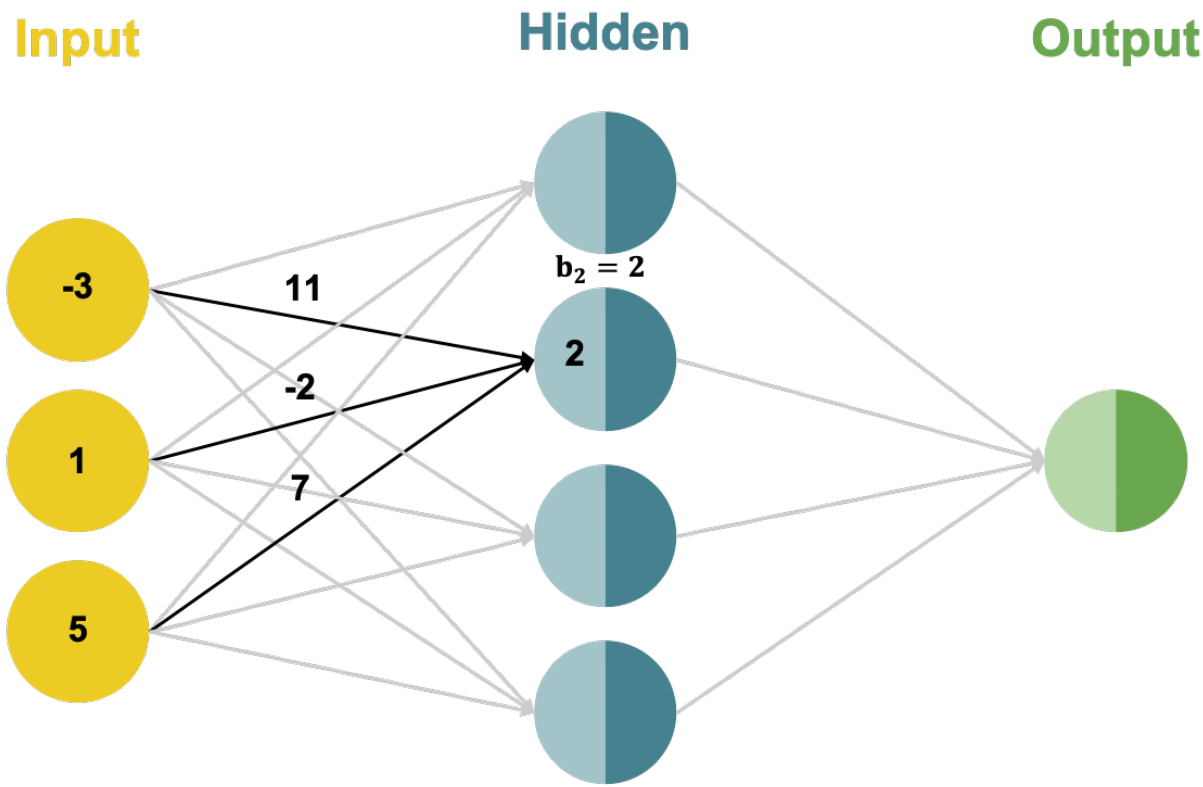
Each neuron in the hidden layer performs an **affine transformation** on the inputs:



$$z_{\text{in}}^{(1)} = w_{11}x^{(1)} + w_{21}x^{(2)} + w_{31}x^{(3)} + b_1$$
$$z_{\text{in}}^{(1)} = 3 * (-3) + (-9) * 1 + 2 * 5 + 5 = -3$$

SINGLE HIDDEN LAYER NETWORKS: EXAMPLE

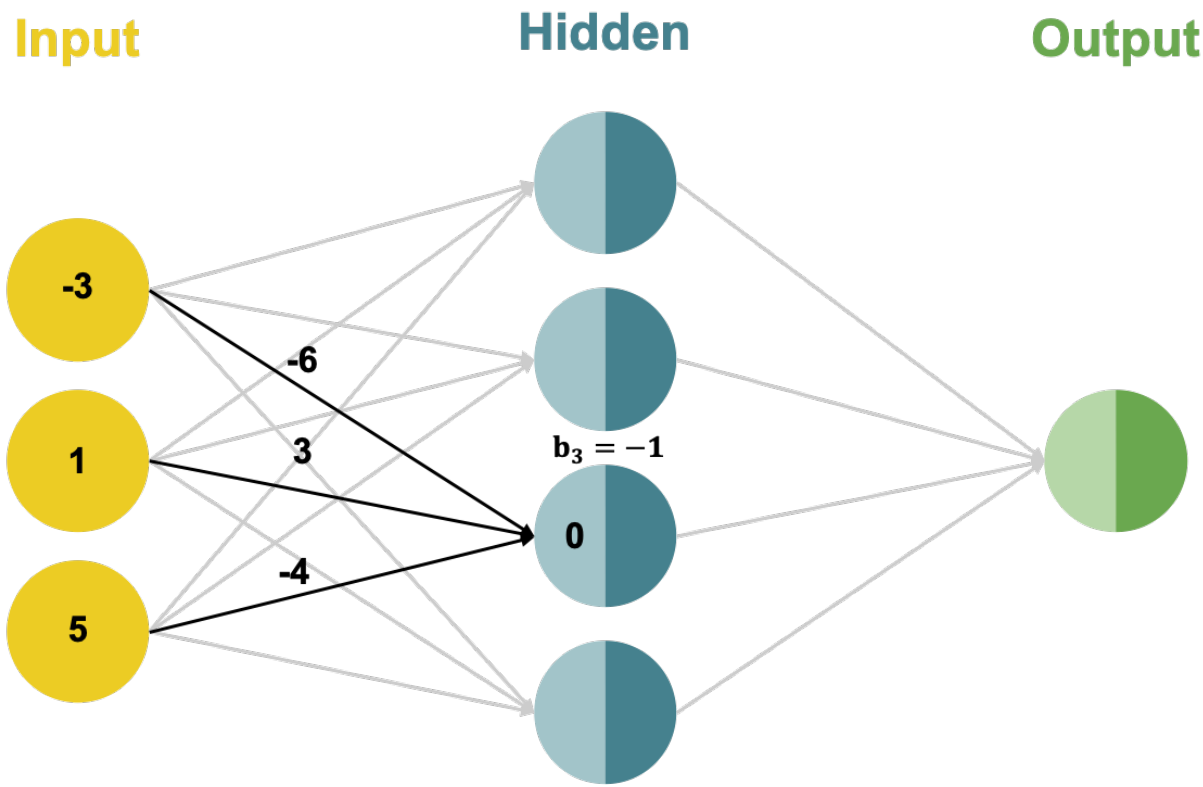
Each neuron in the hidden layer performs an **affine transformation** on the inputs:



$$z_{\text{in}}^{(2)} = w_{12}x^{(1)} + w_{22}x^{(2)} + w_{32}x^{(3)} + b_2$$
$$z_{\text{in}}^{(2)} = 11 * (-3) + (-2) * 1 + 7 * 5 + 2 = 2$$

SINGLE HIDDEN LAYER NETWORKS: EXAMPLE

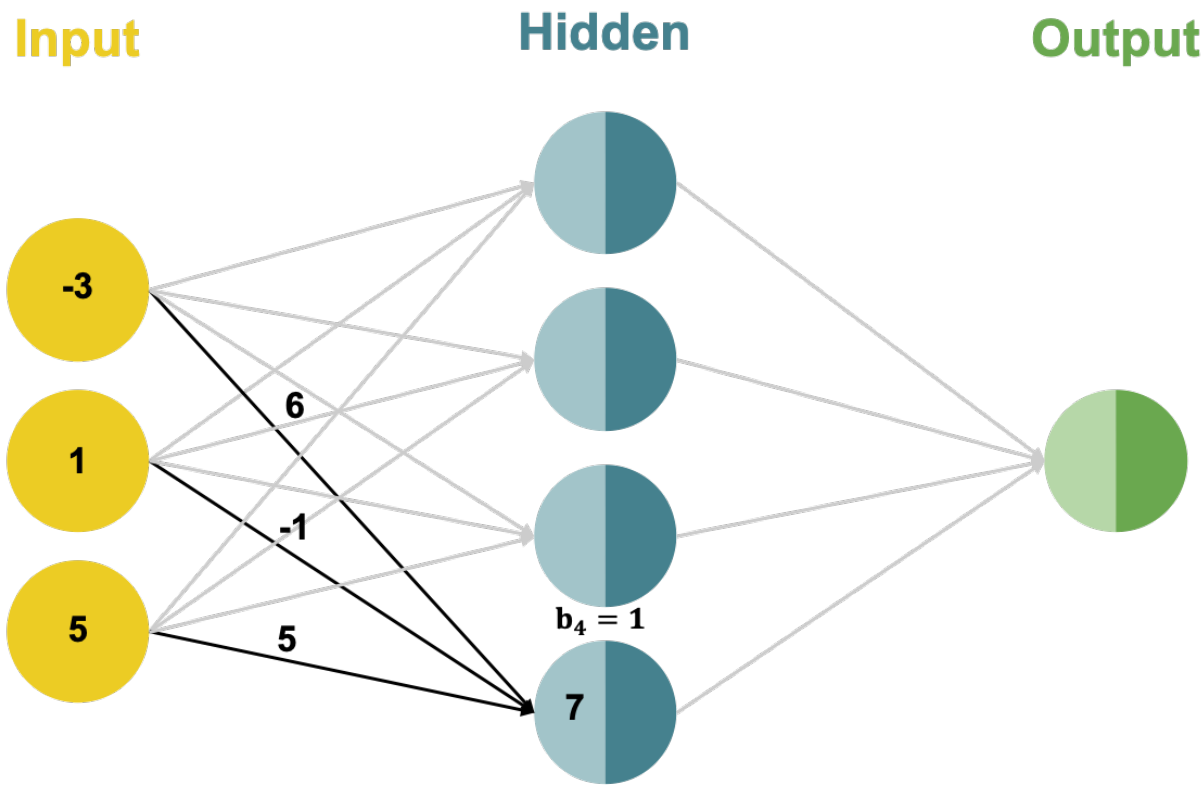
Each neuron in the hidden layer performs an **affine transformation** on the inputs:



$$z_{\text{in}}^{(3)} = w_{13}x^{(1)} + w_{23}x^{(2)} + w_{33}x^{(3)} + b_3$$
$$z_{\text{in}}^{(3)} = (-6) * (-3) + 3 * 1 + (-4) * 5 - 1 = 0$$

SINGLE HIDDEN LAYER NETWORKS: EXAMPLE

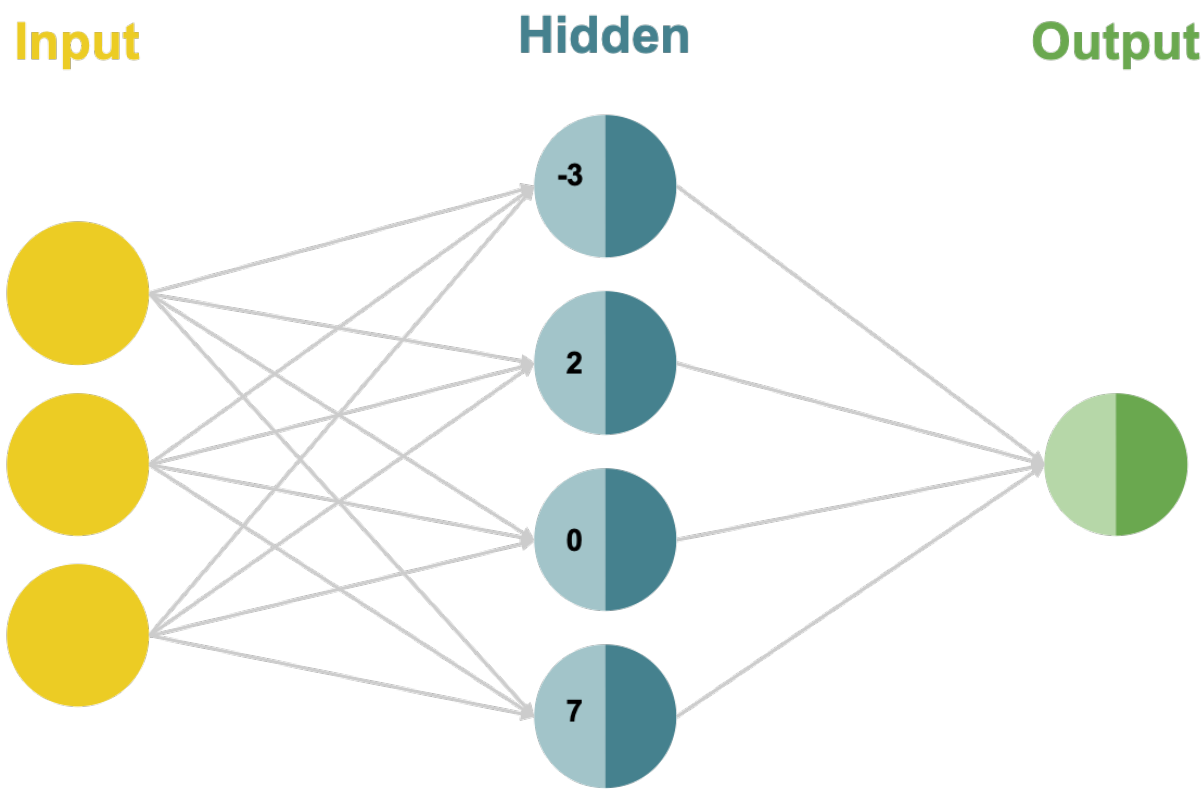
Each neuron in the hidden layer performs an **affine transformation** on the inputs:



$$z_{\text{in}}^{(4)} = w_{14}x^{(1)} + w_{24}x^{(2)} + w_{34}x^{(3)} + b_4$$
$$z_{\text{in}}^{(4)} = 6 * (-3) + (-1) * 1 + 5 * 5 + 1 = 7$$

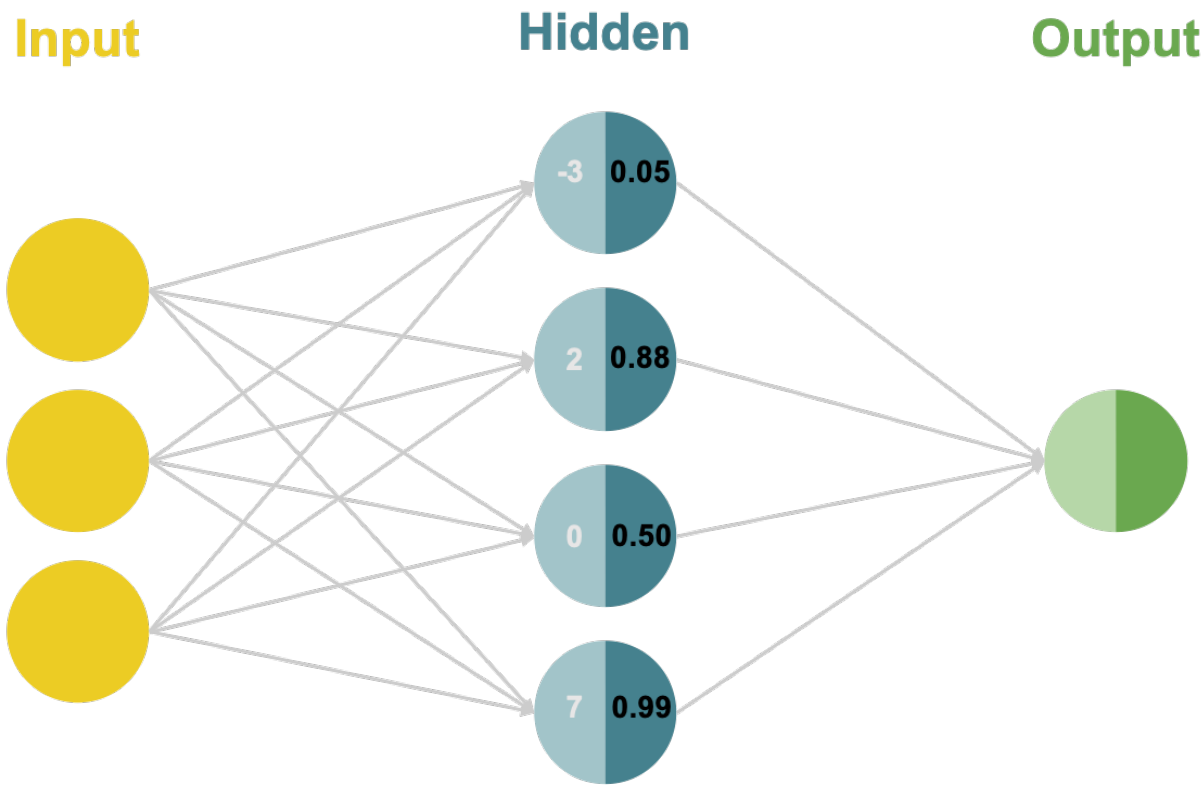
SINGLE HIDDEN LAYER NETWORKS: EXAMPLE

Each neuron in the hidden layer performs an **affine transformation** on the inputs:



SINGLE HIDDEN LAYER NETWORKS: EXAMPLE

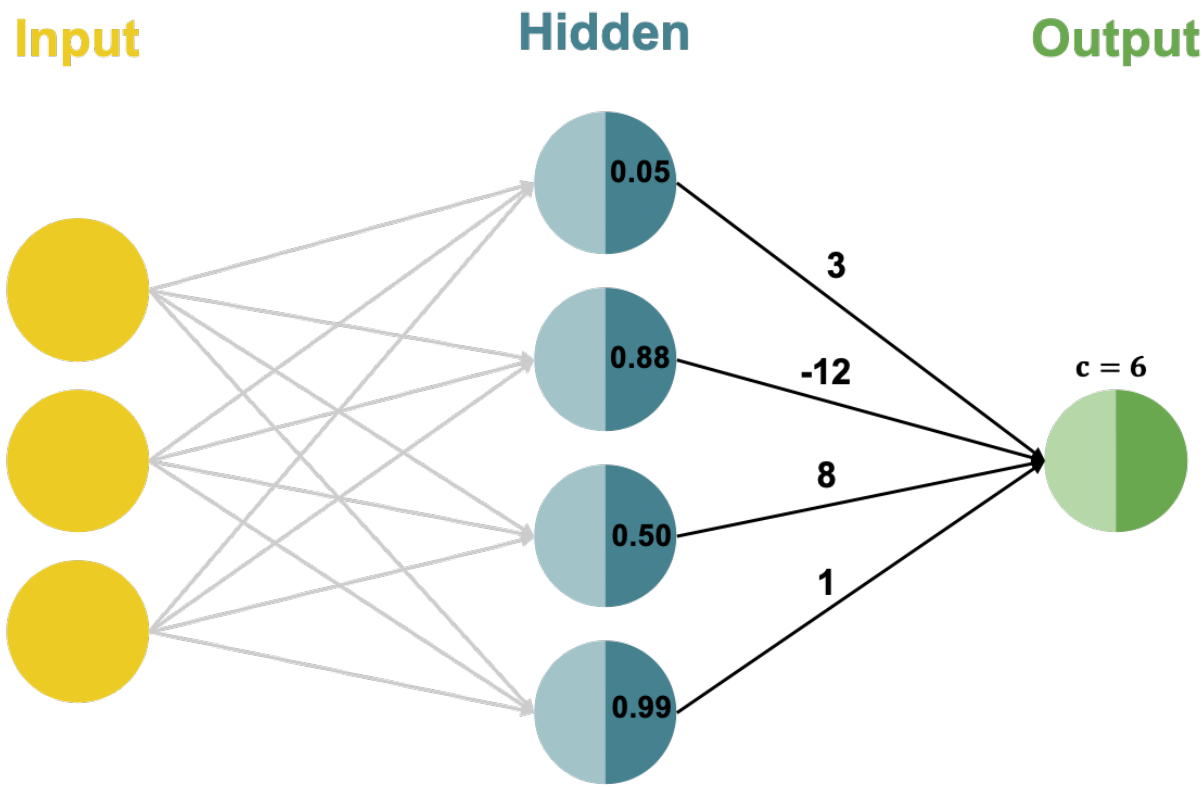
Each hidden neuron performs a non-linear **activation** transformation on the weight sum:



$$z_{\text{out}}^{(i)} = \sigma\left(z_{\text{in}}^{(i)}\right) = \frac{1}{1+e^{z_{\text{in}}^{(i)}}}$$

SINGLE HIDDEN LAYER NETWORKS: EXAMPLE

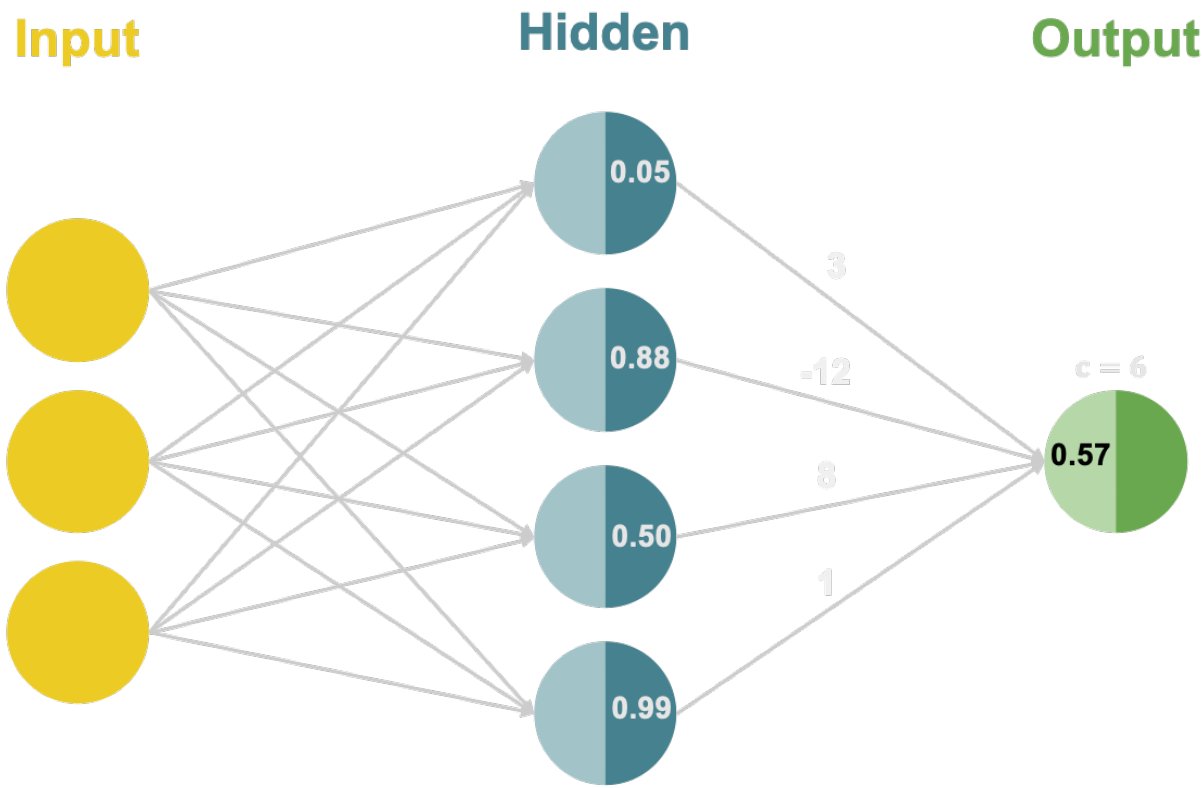
The output neuron performs an **affine transformation** on its inputs:



$$f_{\text{in}} = u_1 z_{\text{out}}^{(1)} + u_2 z_{\text{out}}^{(2)} + u_3 z_{\text{out}}^{(3)} + u_4 z_{\text{out}}^{(4)} + c$$

SINGLE HIDDEN LAYER NETWORKS: EXAMPLE

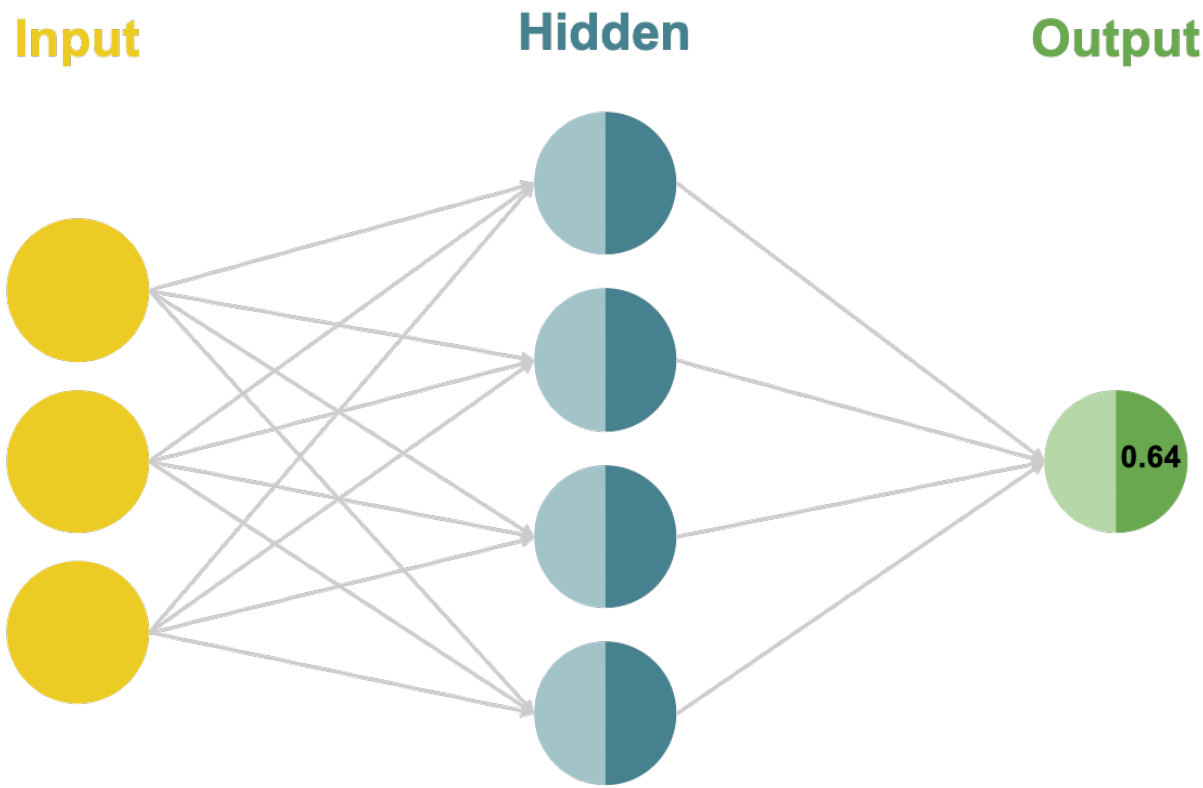
The output neuron performs an **affine transformation** on its inputs:



$$f_{\text{in}} = u_1 z_{\text{out}}^{(1)} + u_2 z_{\text{out}}^{(2)} + u_3 z_{\text{out}}^{(3)} + u_4 z_{\text{out}}^{(4)} + c$$
$$f_{\text{in}} = 3 * 0.05 + (-12) * 0.88 + 8 * 0.50 + 1 * 0.99 + 6 = 0.57$$

SINGLE HIDDEN LAYER NETWORKS: EXAMPLE

The output neuron performs a non-linear **activation** transformation on the weight sum:



$$f_{\text{out}} = \sigma(f_{\text{in}}) = \frac{1}{1+e^{f_{\text{in}}}}$$
$$f_{\text{out}} = \frac{1}{1+e^{0.57}} = 0.64$$

HIDDEN LAYER: ACTIVATION FUNCTION

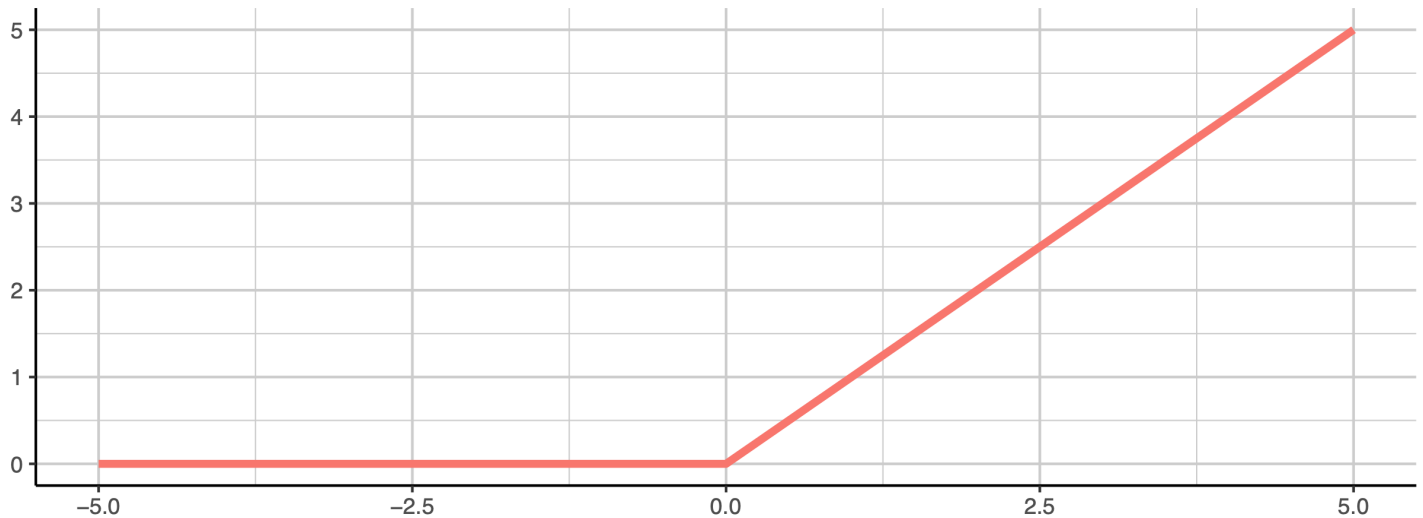
- If the hidden layer does not have a non-linear activation, the network can only learn linear decision boundaries.
- A lot of different activation functions exist.

HIDDEN LAYER: ACTIVATION FUNCTION

ReLU Activation:

- Currently the most popular choice is the ReLU (rectified linear unit):

$$\sigma(v) = \max(0, v)$$

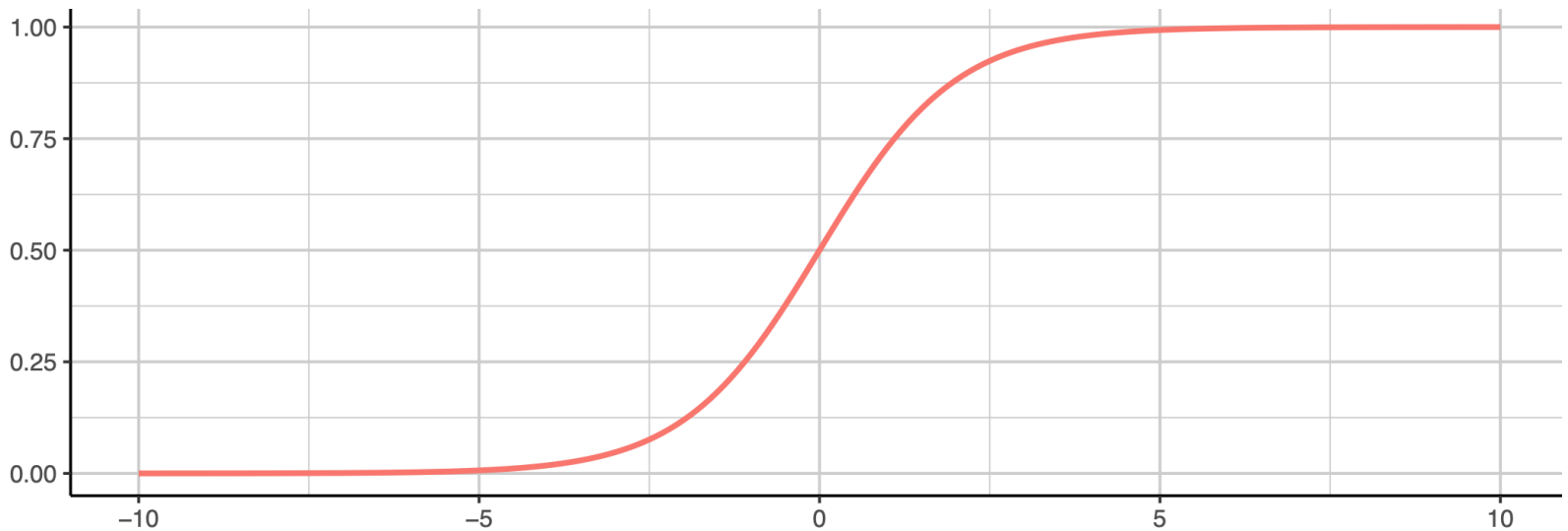


HIDDEN LAYER: ACTIVATION FUNCTION

Sigmoid Activation Function:

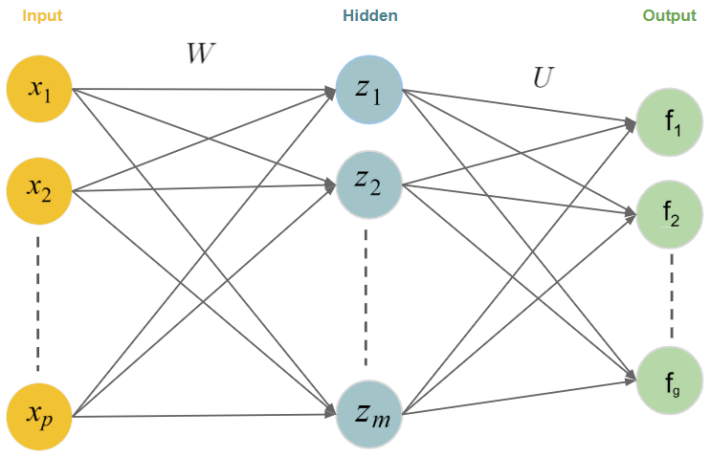
- The sigmoid function can be used even in the hidden layer:

$$\sigma(v) = \frac{1}{1 + \exp(-v)}$$



Deep Learning

Single Hidden Layer Networks for Multi-Class Classification



Learning goals

- Neural network architectures for multi-class classification
- Softmax activation function
- Softmax loss

MULTI-CLASS CLASSIFICATION

- We have only considered regression and binary classification problems so far.
- How can we get a neural network to perform multiclass classification?

MULTI-CLASS CLASSIFICATION

- The first step is to add additional neurons to the output layer.
- Each neuron in the layer will represent a specific class (number of neurons in the output layer = number of classes).

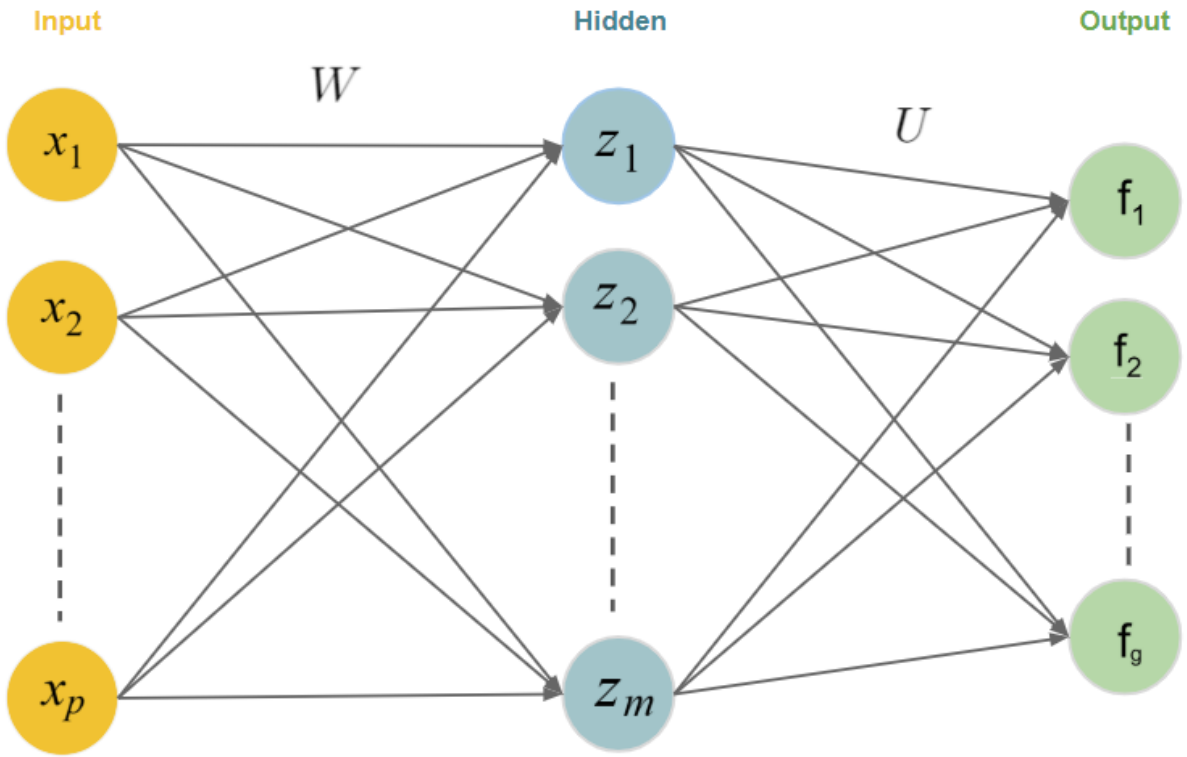


Figure: Structure of a single hidden layer, feed-forward neural network for g-class classification problems (bias term omitted).

MULTI-CLASS CLASSIFICATION

Notation:

- For g-class classification, g output units:

$$\mathbf{f} = (f_1, \ldots, f_g)$$

- m hidden neurons $z_1, \ldots, z_m$, with

$$z_j = \sigma(\mathbf{W}_j^T \mathbf{x}), \quad j = 1, \ldots, m.$$

- Compute linear combinations of derived features z:

$$f_{in,k} = \boldsymbol{U}_k^T \mathbf{z}, \quad \mathbf{z} = (z_1, \ldots, z_m)^T, \quad k = 1, \ldots, g$$

MULTI-CLASS CLASSIFICATION

- The second step is to apply a **softmax** activation function to the output layer.

- This gives us a probability distribution over g different possible classes:

$$f_{out,k} = \tau_k(f_{in,k}) = \frac{\exp(f_{in,k})}{\sum_{k'=1}^{g} \exp(f_{in,k'})}$$

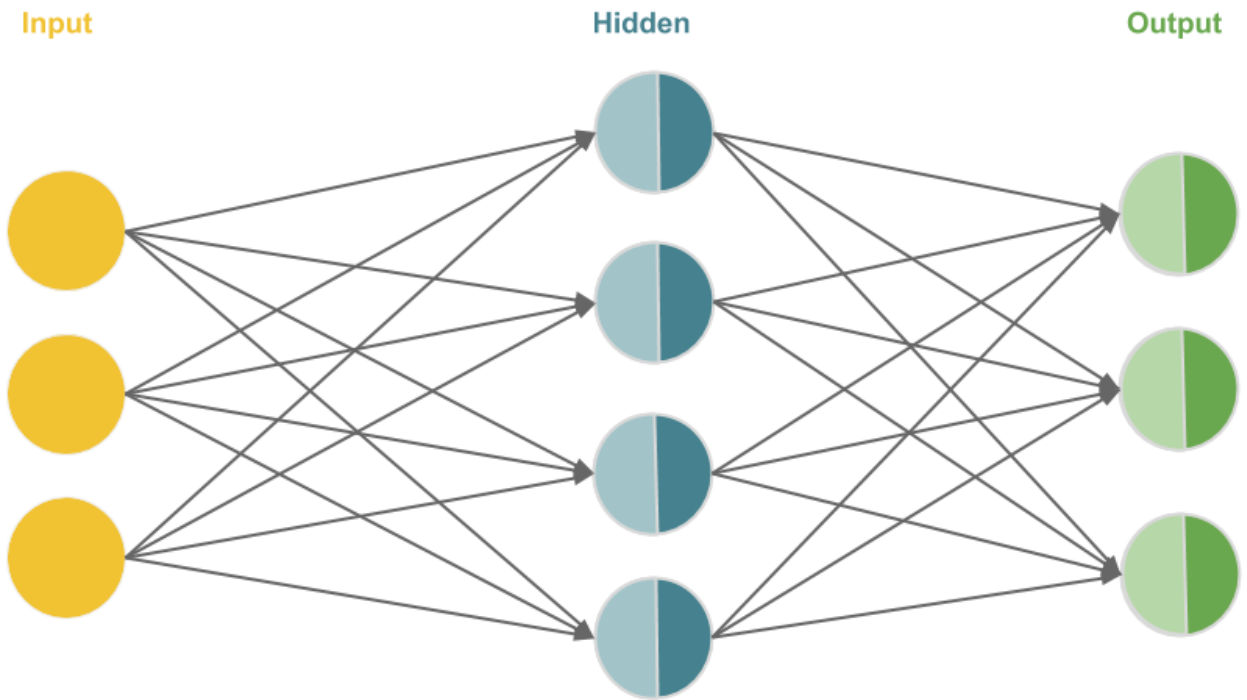
- This is the same transformation used in softmax regression!

- Derivative $\frac{\partial \tau(\mathbf{f}_{in})}{\partial \mathbf{f}_{in}} = \text{diag}(\tau(\mathbf{f}_{in})) - \tau(\mathbf{f}_{in})\tau(\mathbf{f}_{in})^T$

- It is a "smooth" approximation of the argmax operation, so $\tau((1, 1000, 2)^T) \approx (0, 1, 0)^T$ (picks out 2nd element!).

MULTI-CLASS CLASSIFICATION: EXAMPLE

Forward pass (Hidden: Sigmoid, Output: Softmax).



$$\begin{pmatrix} 3 & -9 & 2 \\ 11 & -2 & 7 \\ -6 & 3 & -4 \\ 6 & -1 & 5 \end{pmatrix} \begin{pmatrix} 5 \\ 2 \\ -1 \\ 1 \end{pmatrix}$$

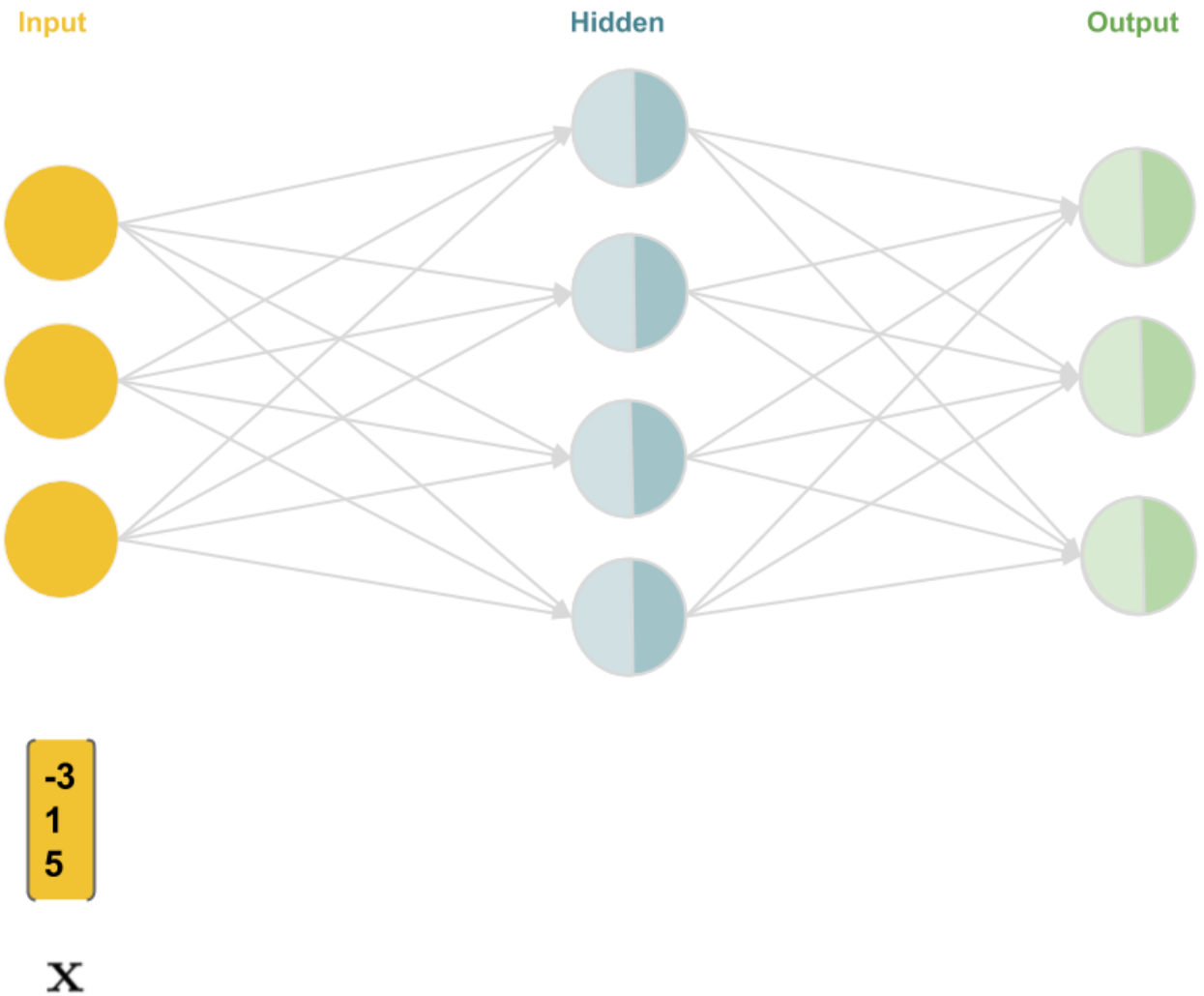
W^T $\mathbf{b}$

$$\begin{pmatrix} 3 & -12 & 8 & 1 \\ 2 & -3 & 9 & 1 \\ -5 & 1 & -1 & 7 \end{pmatrix} \begin{pmatrix} 6 \\ 0 \\ -8 \end{pmatrix}$$

U^T $\mathbf{c}$

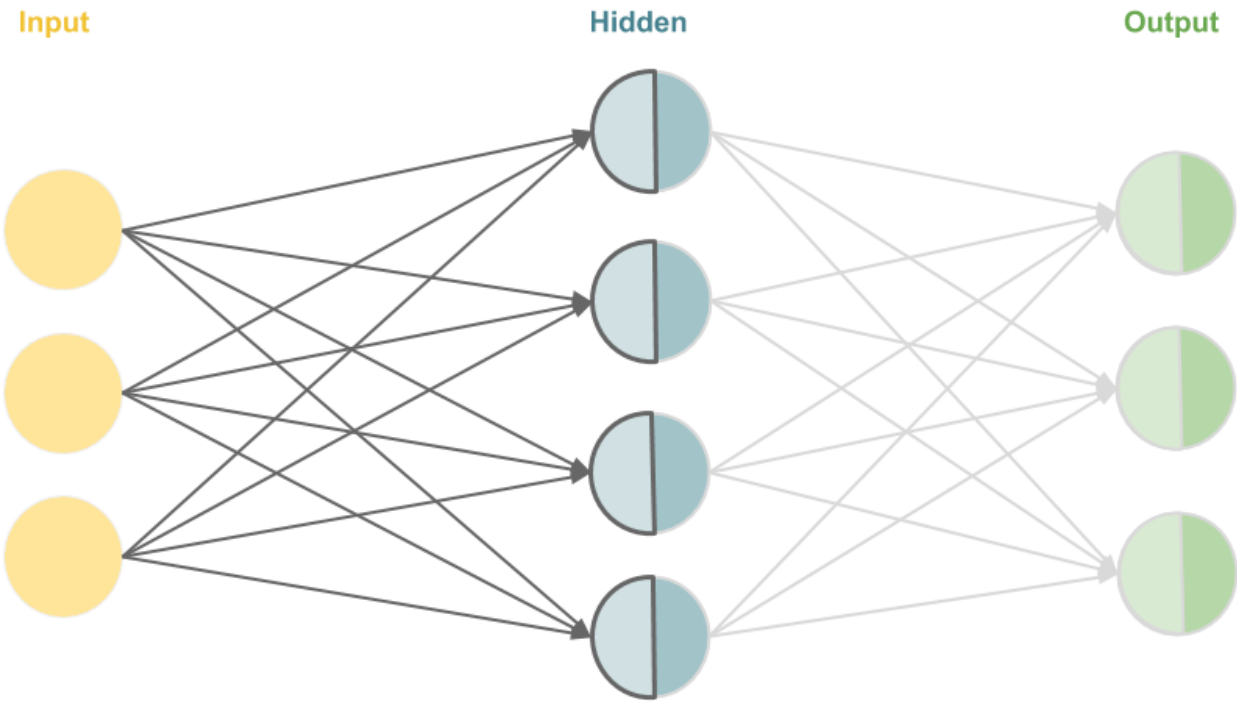
MULTI-CLASS CLASSIFICATION: EXAMPLE

Forward pass (Hidden: Sigmoid, Output: Softmax).



MULTI-CLASS CLASSIFICATION: EXAMPLE

Forward pass (Hidden: Sigmoid, Output: Softmax).

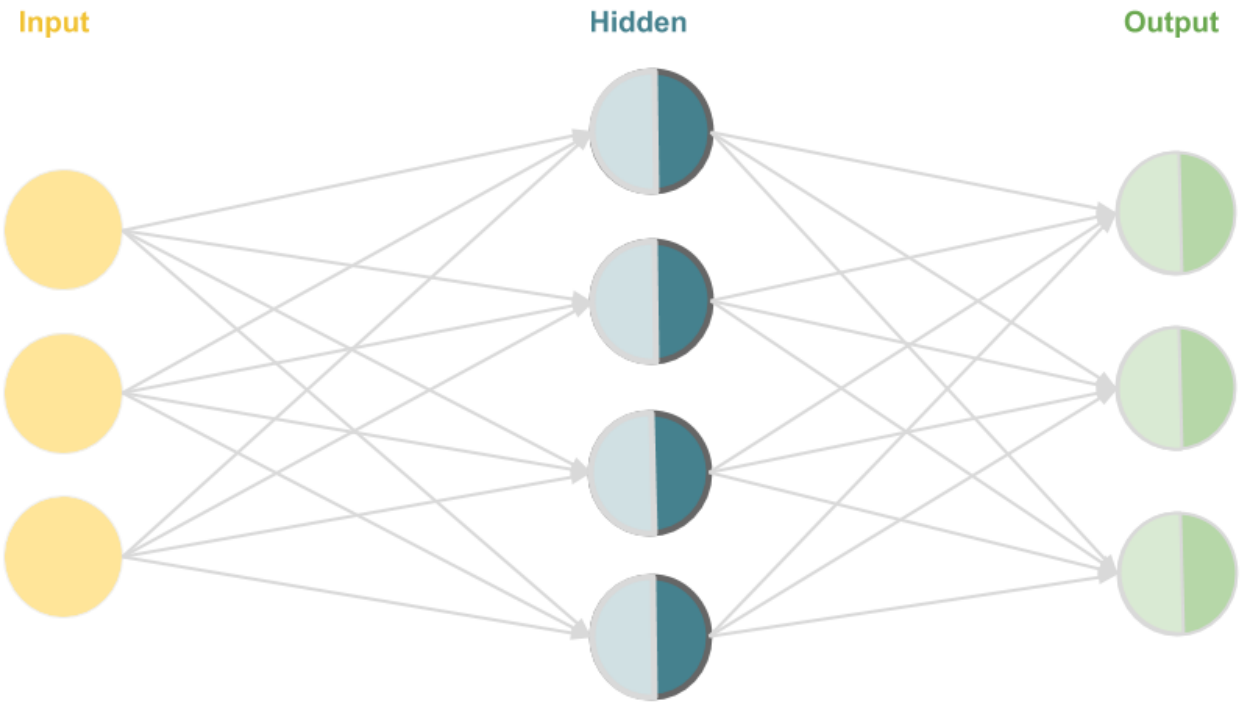


$$\begin{bmatrix} -3 \\ 1 \\ 5 \end{bmatrix} \quad \begin{bmatrix} (-3)*3 + 1*(-9) + 5*2 + 5 \\ (-3)*11 + 1*(-2) + 5*7 + 2 \\ (-3)*(-6) + 1*3 + 5*(-4) + (-1) \\ (-3)*6 + 1*(-1) + 5*5 + 1 \end{bmatrix}$$

$$\mathbf{x} \qquad \mathbf{z}_{in} = \mathbf{w}^T\mathbf{x} + \mathbf{b}$$

MULTI-CLASS CLASSIFICATION: EXAMPLE

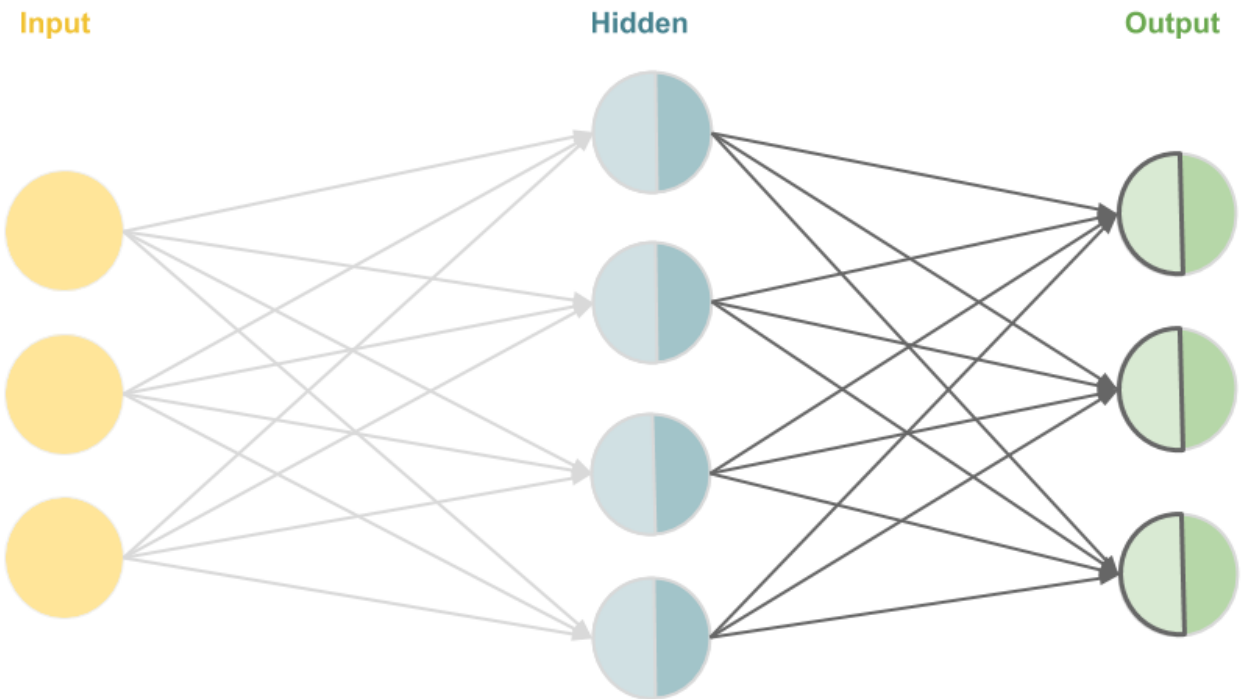
Forward pass (Hidden: Sigmoid, Output: Softmax).



$\mathbf{x}$ $\mathbf{z}_{in}$ $\mathbf{z} = \mathbf{z}_{out} = \sigma(\mathbf{z}_{in})$

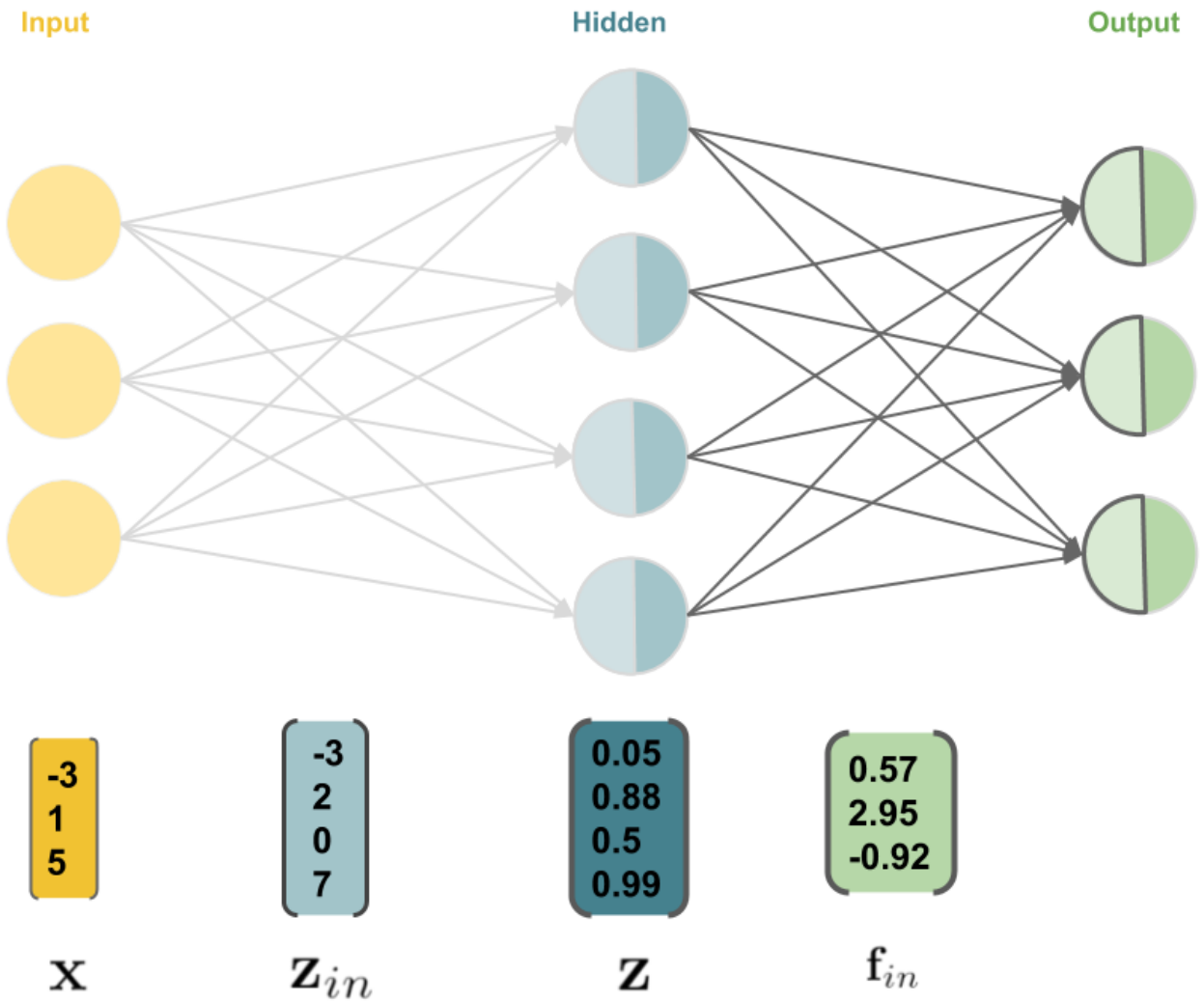
MULTI-CLASS CLASSIFICATION: EXAMPLE

Forward pass (Hidden: Sigmoid, Output: Softmax).



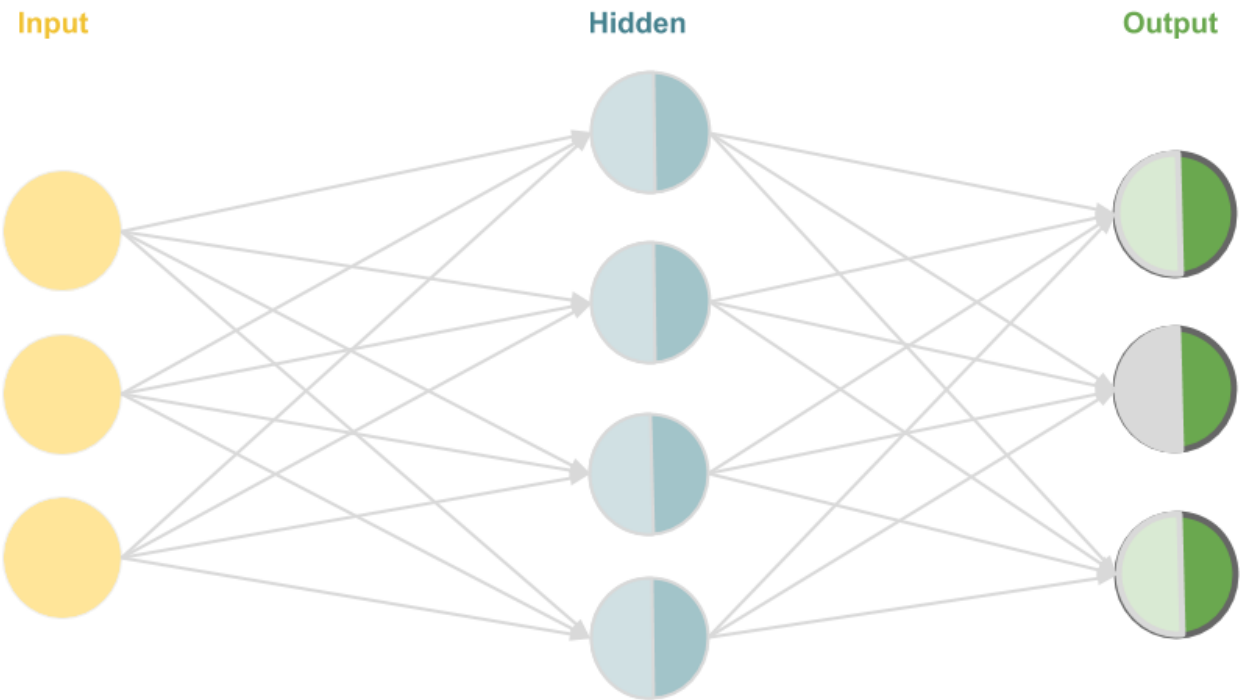
MULTI-CLASS CLASSIFICATION: EXAMPLE

Forward pass (Hidden: Sigmoid, Output: Softmax).



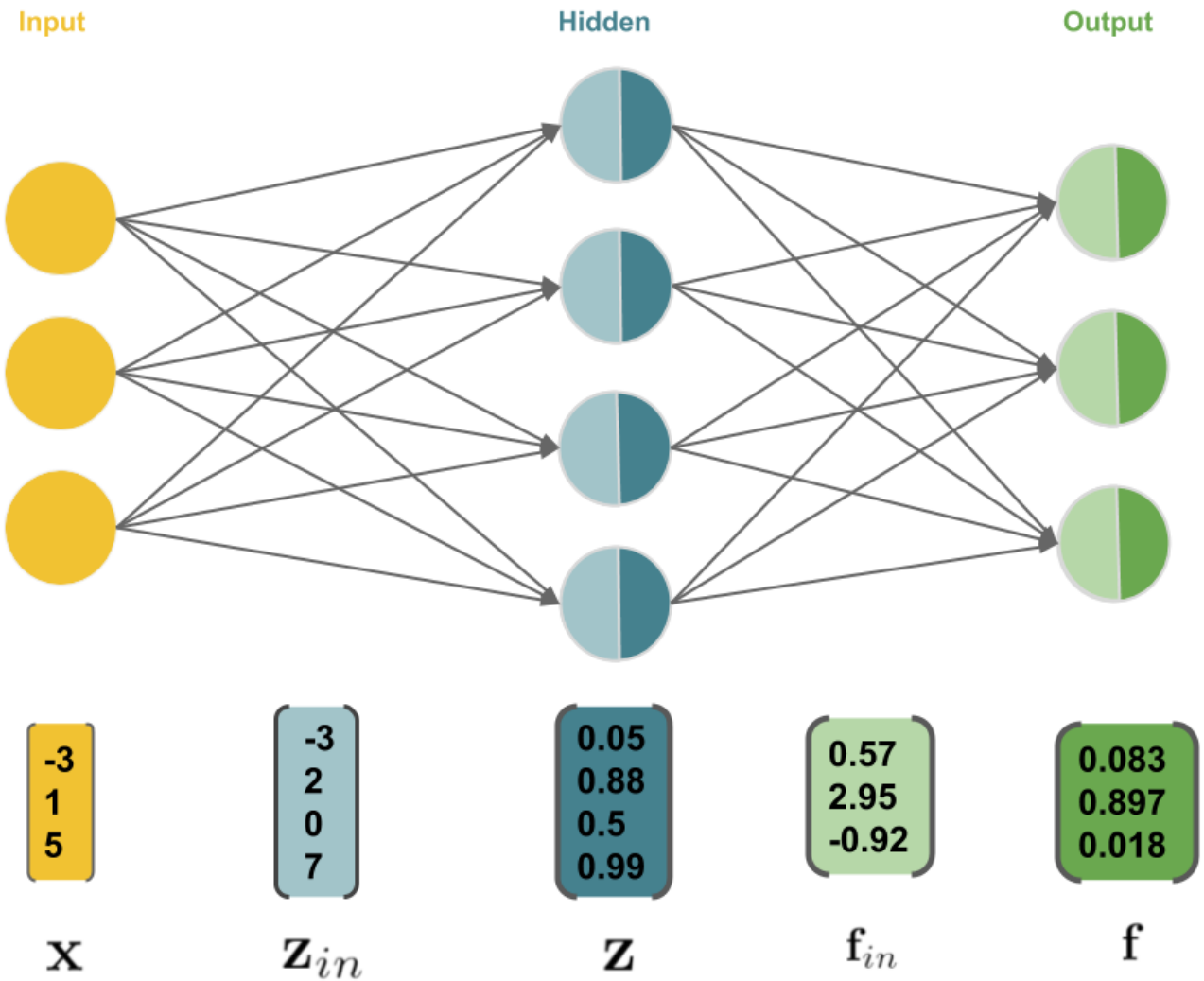
MULTI-CLASS CLASSIFICATION: EXAMPLE

Forward pass (Hidden: Sigmoid, Output: Softmax).



MULTI-CLASS CLASSIFICATION: EXAMPLE

Forward pass (Hidden: Sigmoid, Output: Softmax).



OPTIMIZATION: SOFTMAX LOSS

- The loss function for a softmax classifier is

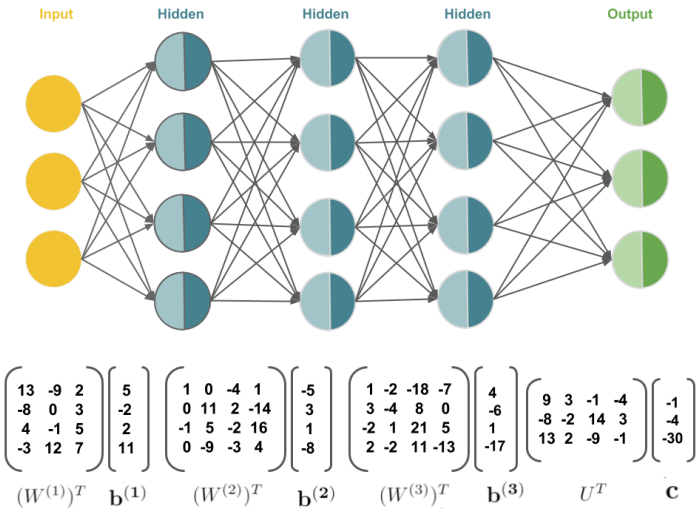
$$L(y, f(\mathbf{x})) = -\sum_{k=1}^{g} [y = k] \log \left(\frac{\exp(f_{in,k})}{\sum_{k'=1}^{g} \exp(f_{in,k'})} \right)$$

where $[y = k] = \begin{cases} 1 & \text{if } y = k \\ 0 & \text{otherwise} \end{cases}$.

- This is equivalent to the cross-entropy loss when the label vector $\boldsymbol{y}$ is one-hot coded (e.g. $\mathbf{y} = (0, 0, 1, 0)^T$).
- Optimization: Again, there is no analytic solution.

Deep Learning

MLP – Multi-Layer Feedforward Neural Networks



Learning goals

- Architectures of deep neural networks
- Deep neural networks as chained functions

FEEDFORWARD NEURAL NETWORKS

- We will now extend the model class once again, such that we allow an arbitrary amount l of hidden layers.

- The general term for this model class is (multi-layer) **feedforward networks** (inputs are passed through the network from left to right, no feedback-loops are allowed)

FEEDFORWARD NEURAL NETWORKS

- We can characterize those models by the following chain structure:

$$f(\mathbf{x}) = \tau \circ \phi \circ \sigma^{(l)} \circ \phi^{(l)} \circ \sigma^{(l-1)} \circ \phi^{(l-1)} \circ \ldots \circ \sigma^{(1)} \circ \phi^{(1)}$$

  where $\sigma^{(i)}$ and $\phi^{(i)}$ are the activation function and the weighted sum of hidden layer i, respectively. τ and ϕ are the corresponding components of the output layer.

- Each hidden layer has:
  - an associated weight matrix $\mathbf{W}^{(i)}$, bias $\mathbf{b}^{(i)}$, and activations $\mathbf{z}^{(i)}$ for $i \in \{1 \ldots l\}$.
  - $\mathbf{z}^{(i)} = \sigma^{(i)}(\phi^{(i)}) = \sigma^{(i)}(\mathbf{W}^{(i)T}\mathbf{z}^{(i-1)} + \mathbf{b}^{(i)})$, where $\mathbf{z}^{(0)} = \mathbf{x}$.

- Again, without non-linear activations in the hidden layers, the network can only learn linear decision boundaries.

FEEDFORWARD NEURAL NETWORKS

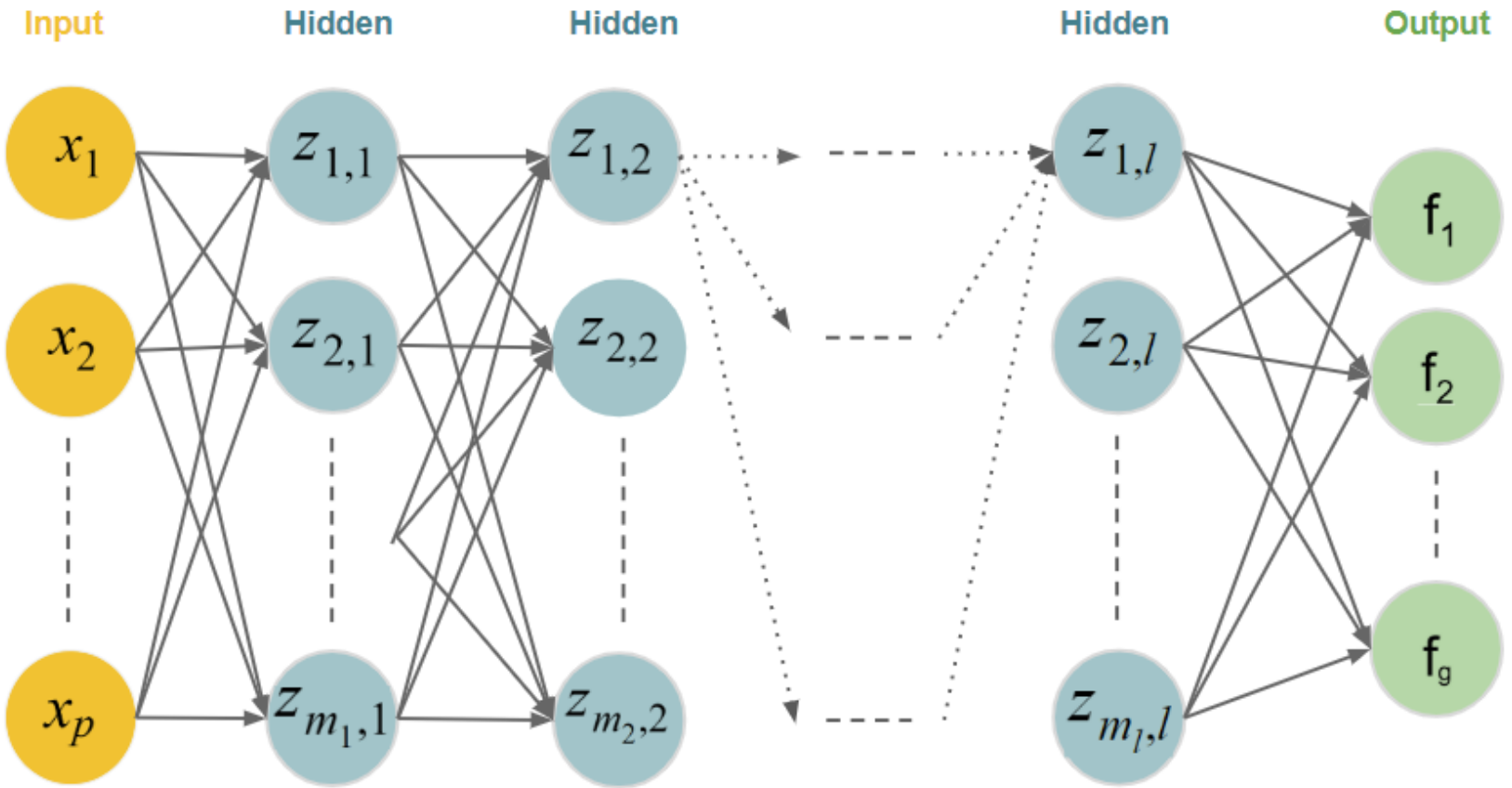
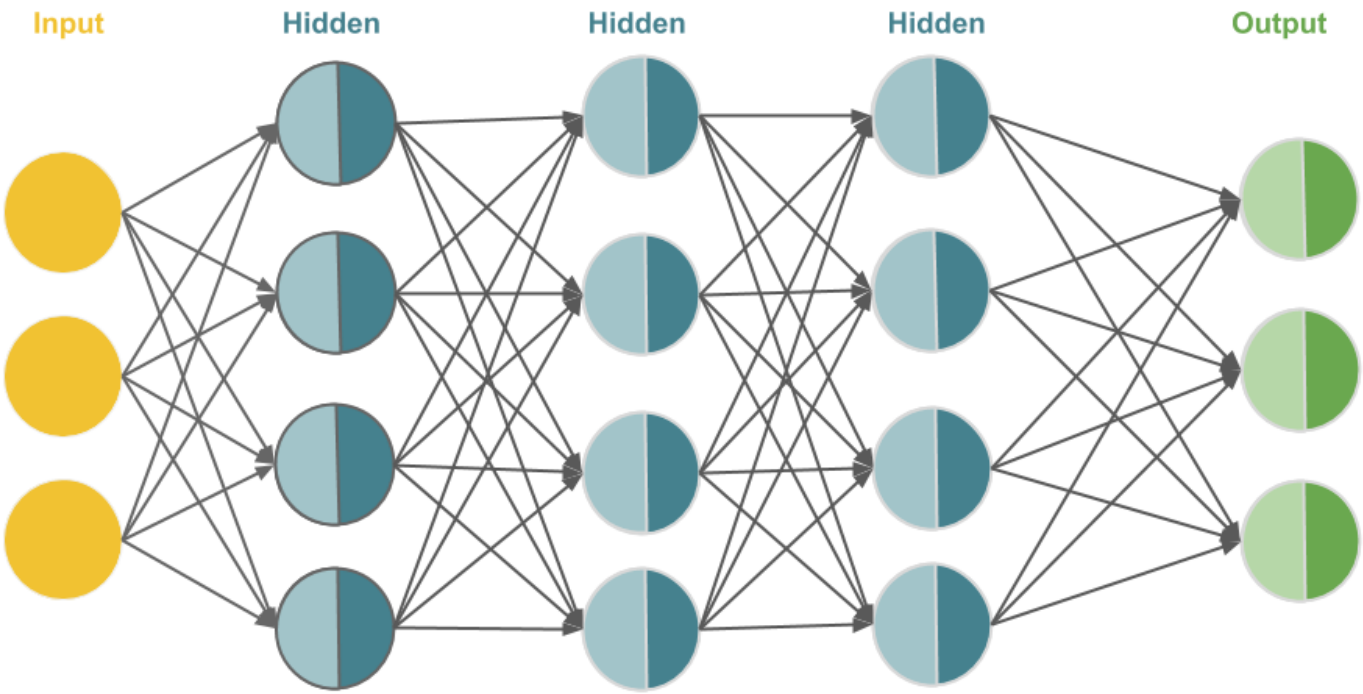


Figure: Structure of a deep neural network with *l* hidden layers (bias terms omitted).

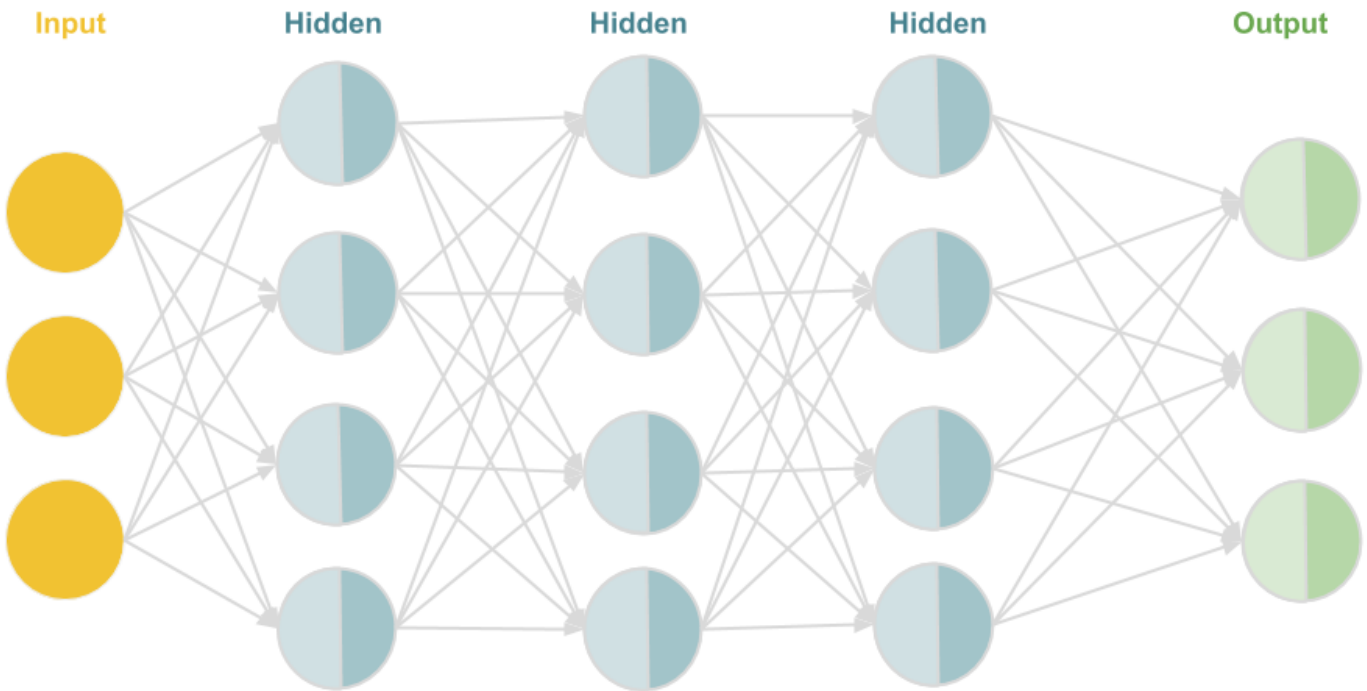
FEEDFORWARD NEURAL NETWORKS: EXAMPLE



$$\begin{pmatrix} 13 & -9 & 2 \\ -8 & 0 & 3 \\ 4 & -1 & 5 \\ -3 & 12 & 7 \end{pmatrix} \begin{pmatrix} 5 \\ -2 \\ 2 \\ 11 \end{pmatrix} \quad \begin{pmatrix} 1 & 0 & -4 & 1 \\ 0 & 11 & 2 & -14 \\ -1 & 5 & -2 & 16 \\ 0 & -9 & -3 & 4 \end{pmatrix} \begin{pmatrix} -5 \\ 3 \\ 1 \\ -8 \end{pmatrix} \quad \begin{pmatrix} 1 & -2 & -18 & -7 \\ 3 & -4 & 8 & 0 \\ -2 & 1 & 21 & 5 \\ 2 & -2 & 11 & -13 \end{pmatrix} \begin{pmatrix} 4 \\ -6 \\ 1 \\ -17 \end{pmatrix} \quad \begin{pmatrix} 9 & 3 & -1 & -4 \\ -8 & -2 & 14 & 3 \\ 13 & 2 & -9 & -1 \end{pmatrix} \begin{pmatrix} -1 \\ -4 \\ -30 \end{pmatrix}$$

$(W^{(1)})^T \quad \mathbf{b}^{(1)} \quad (W^{(2)})^T \quad \mathbf{b}^{(2)} \quad (W^{(3)})^T \quad \mathbf{b}^{(3)} \quad U^T \quad \mathbf{c}$

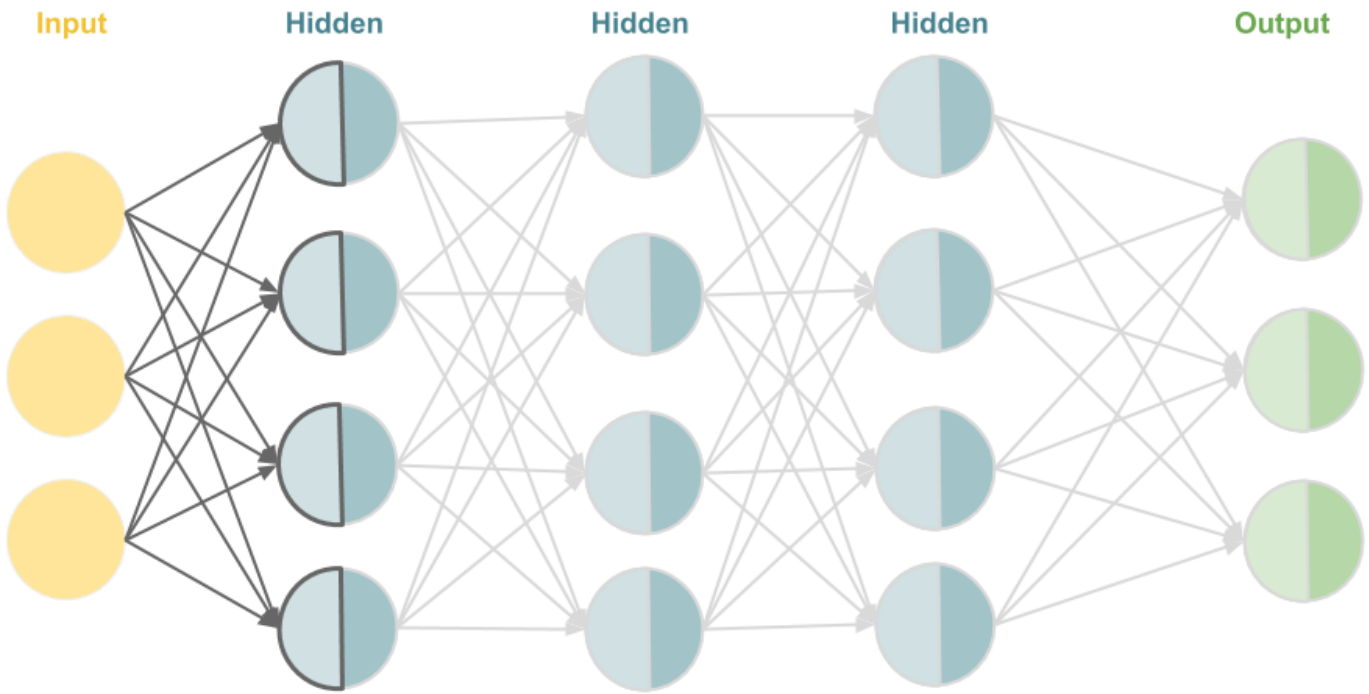
FEEDFORWARD NEURAL NETWORKS: EXAMPLE



7
1
-4

x

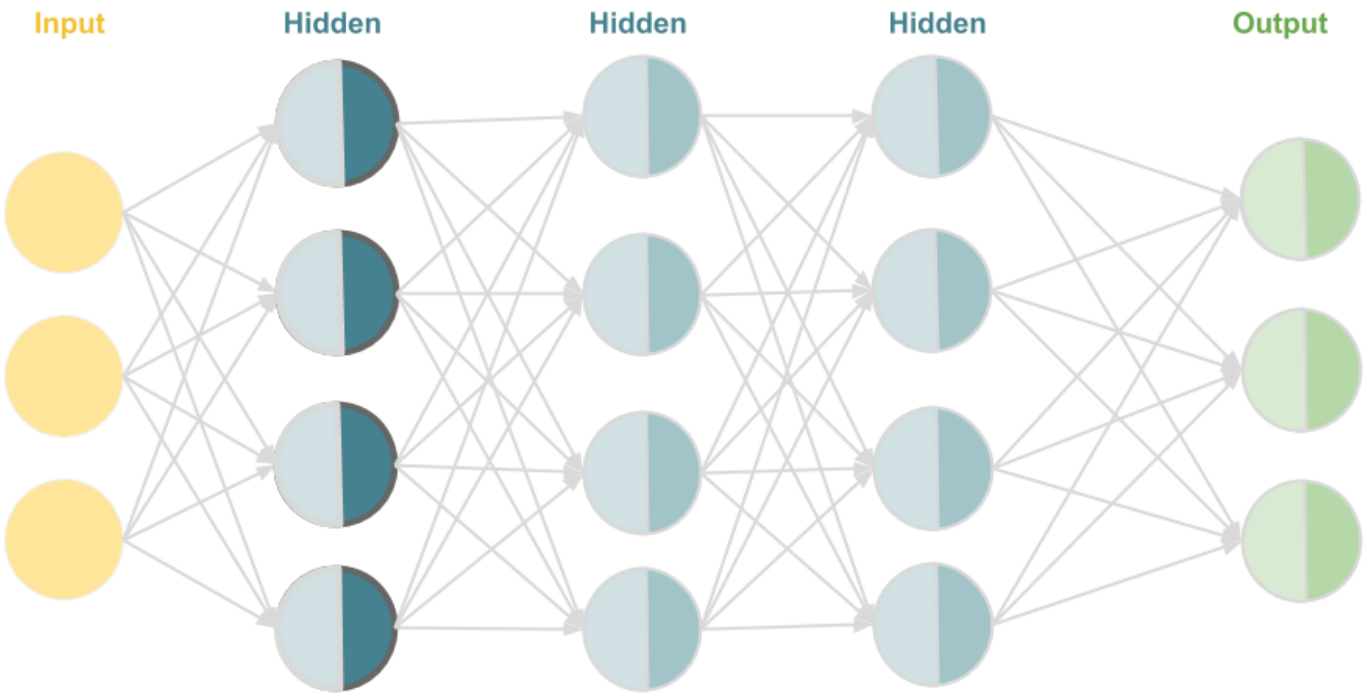
FEEDFORWARD NEURAL NETWORKS: EXAMPLE

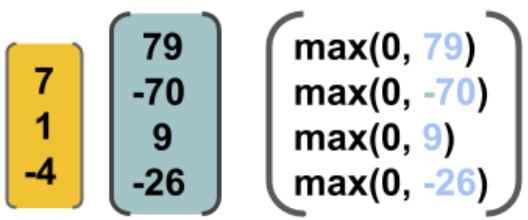


$$\begin{matrix} 7 \\ 1 \\ -4 \end{matrix} \begin{pmatrix} 7*13 + 1*(-9) + (-4)*2 + 5 \\ 7*(-8) + 1*0 + (-4)*3 + (-2) \\ 7*4 + 1*(-1) + (-4)*5 + 2 \\ 7*(-3) + 1*12 + (-4)*7 + 11 \end{pmatrix}$$

$\mathbf{x}$ $\quad \mathbf{z}_{in}^{(1)} = W^{(1)T}\mathbf{x} + \mathbf{b}^{(1)}$

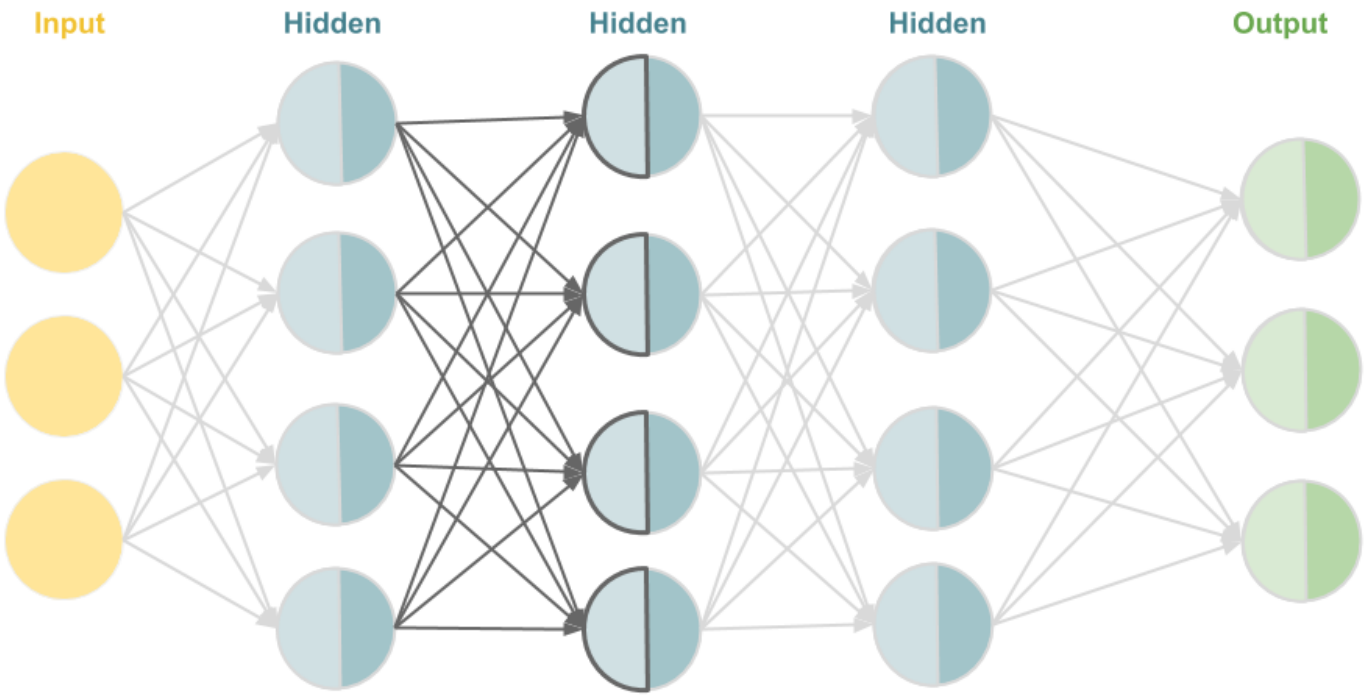
FEEDFORWARD NEURAL NETWORKS: EXAMPLE





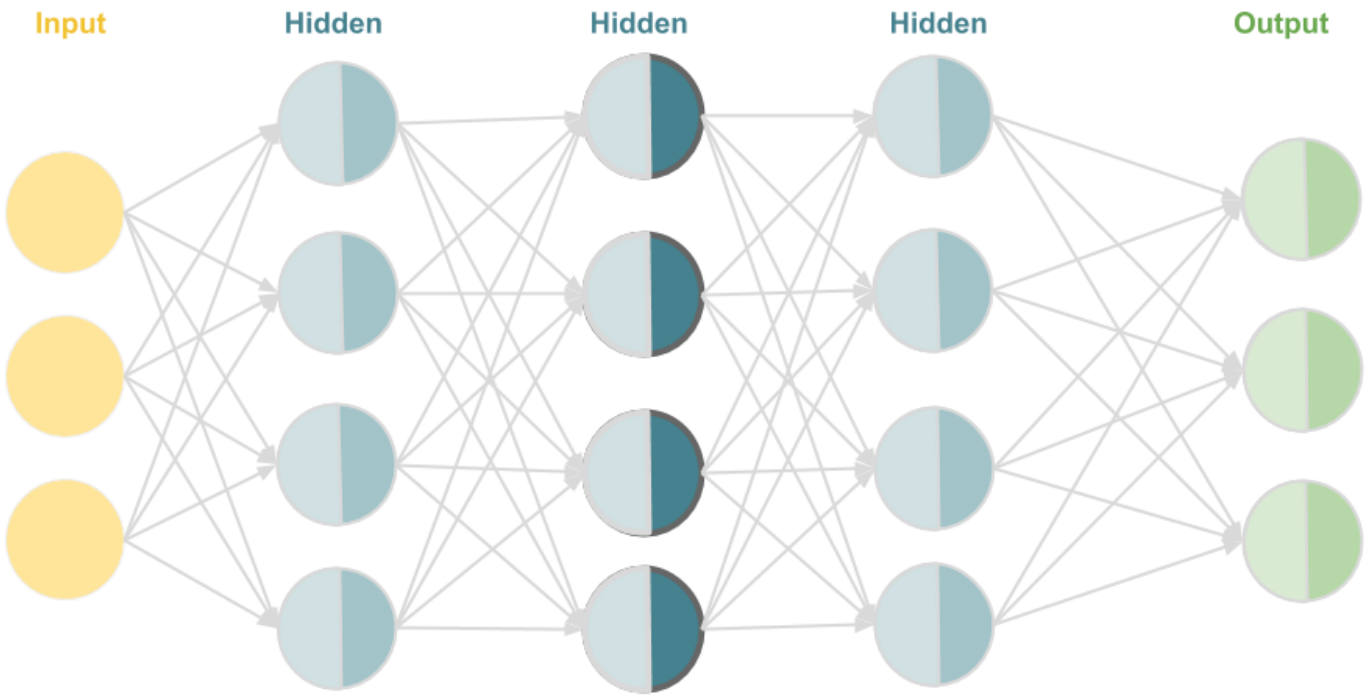
$\mathbf{x} \quad \mathbf{z}_{in}^{(1)} \quad \mathbf{z}^{(1)} = \mathbf{z}_{out}^{(1)} = \sigma(\mathbf{z}_{in}^{(1)})$

FEEDFORWARD NEURAL NETWORKS: EXAMPLE



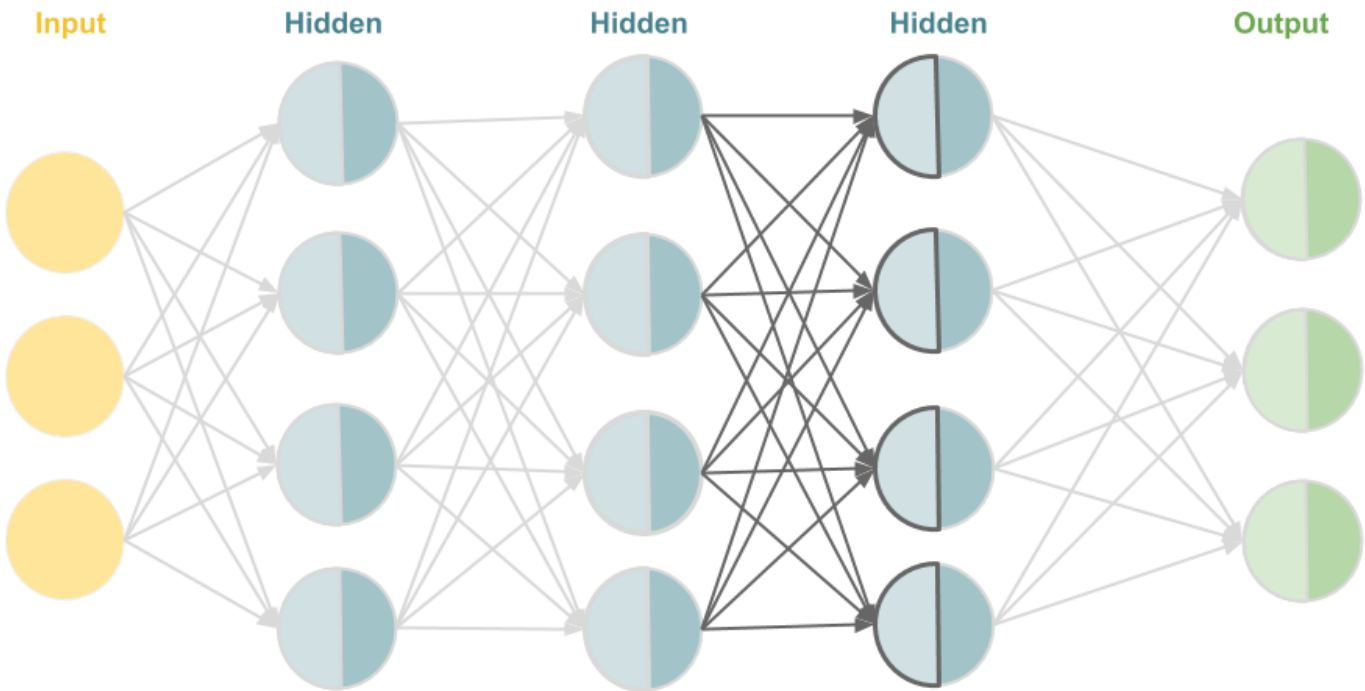
$\mathbf{x}$	$\mathbf{z}_{in}^{(1)}$	$\mathbf{z}^{(1)}$	$\mathbf{z}_{in}^{(2)} = W^{(2)T}\mathbf{z}^{(1)} + \mathbf{b}^{(2)}$
7 1 -4	79 -70 9 -26	79 0 9 0	79*1 + 0*0 + 9*(-4) + 0*1 + (-5) 79*0 + 0*11 + 9*2 + 0*(-14) + 3 79*(-1) + 0*5 + 9*(-2) + 0*16 + 1 79*0 + 0*(-9) + 9*(-3) + 0*4 + (-8)

FEEDFORWARD NEURAL NETWORKS: EXAMPLE



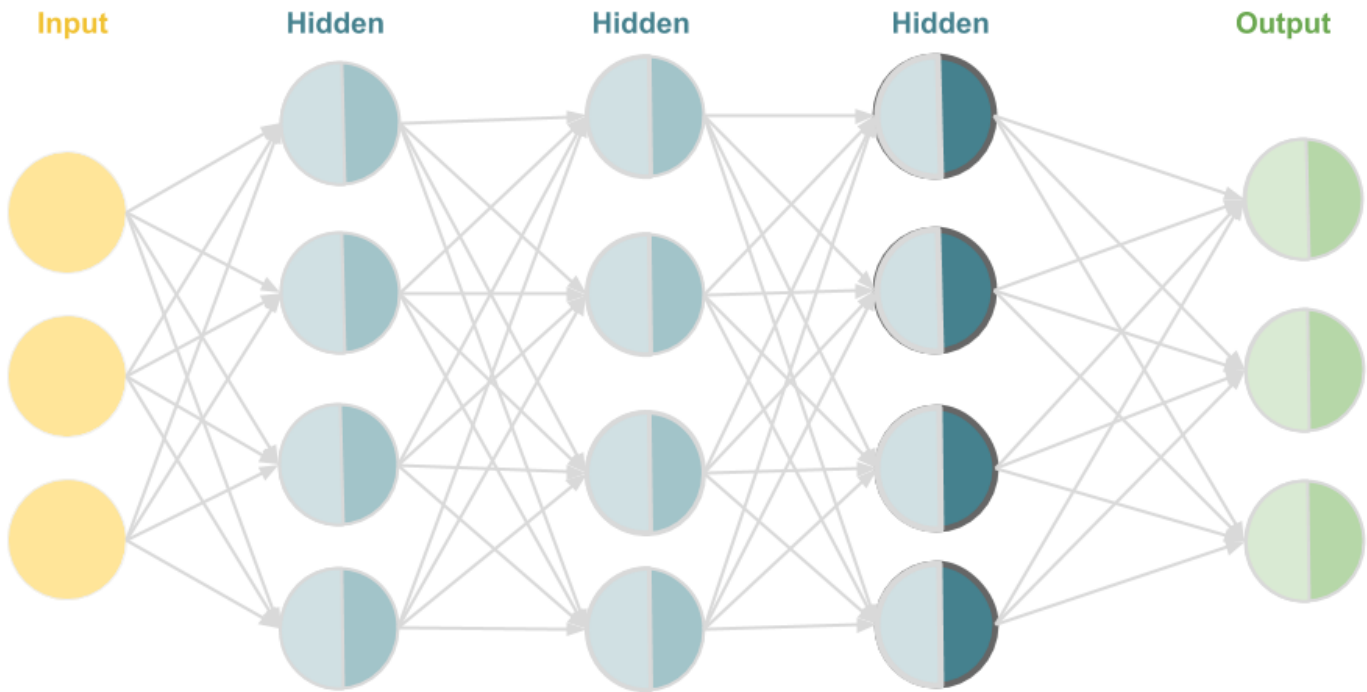
$\mathbf{x}$	$\mathbf{z}_{in}^{(1)}$	$\mathbf{z}^{(1)}$	$\mathbf{z}_{in}^{(2)}$	$\mathbf{z}^{(2)} = \mathbf{z}_{out}^{(2)} = \sigma(\mathbf{z}_{in}^{(2)})$
7	79	79	38	max(0, 38)
1	-70	0	21	max(0, 21)
-4	9	9	-96	max(0, -96)
	-26	0	-35	max(0, -36)

FEEDFORWARD NEURAL NETWORKS: EXAMPLE



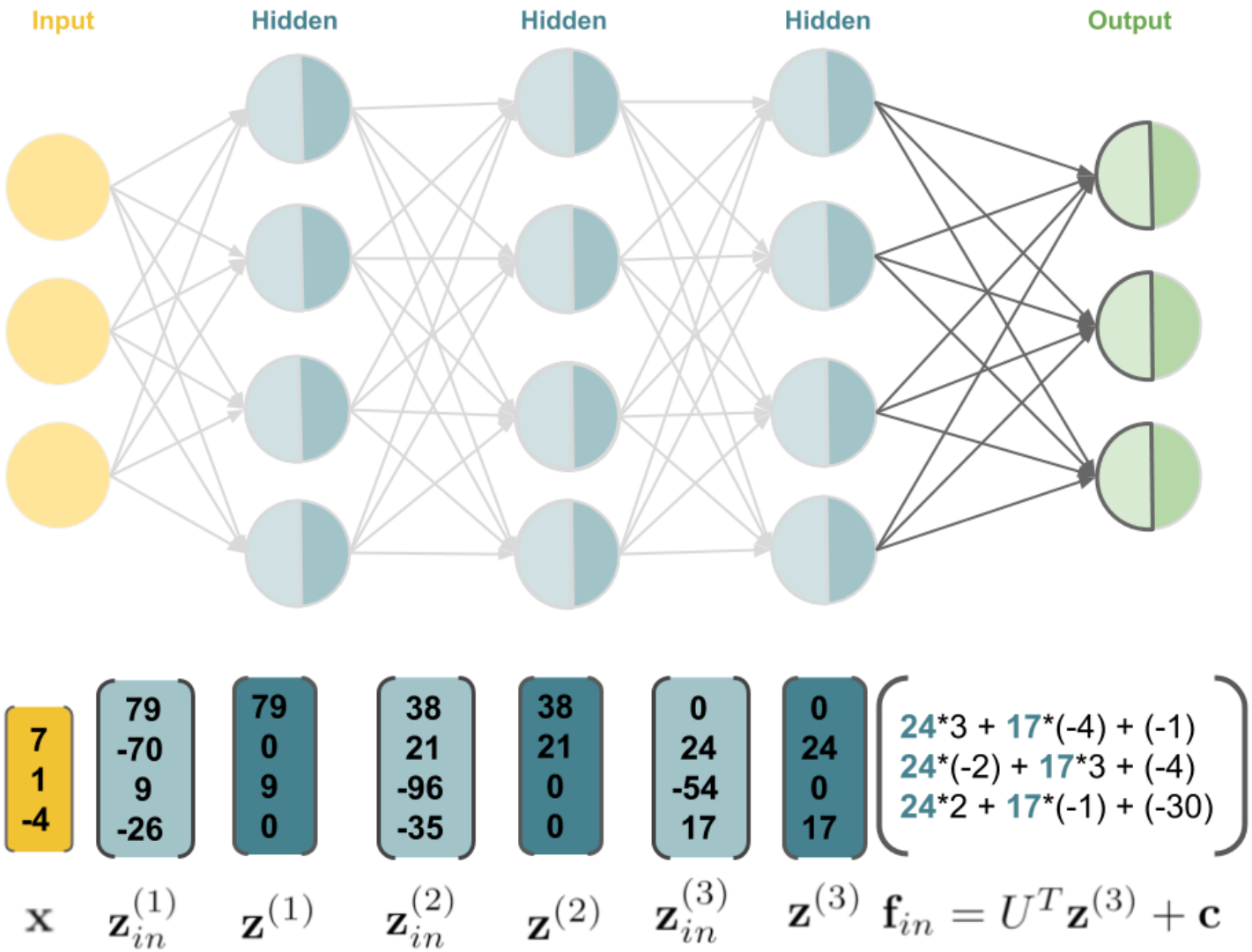
$\mathbf{x}$	$\mathbf{z}_{in}^{(1)}$	$\mathbf{z}^{(1)}$	$\mathbf{z}_{in}^{(2)}$	$\mathbf{z}^{(2)}$	$\mathbf{z}_{in}^{(3)} = W^{(3)T}\mathbf{z}^{(2)} + \mathbf{b}^{(3)}$
7	79	79	38	38	38*1 + 21*(-2) + 0*(-18) + 0*(-7) + 4
1	-70	0	21	21	38*3 + 21*(-4) + 0*8 + 0*0 + (-6)
-4	9	9	-96	0	38*(-2) + 21*1 + 0*21 + 0*5 + 1
	-26	0	-35	0	38*2 + 21*(-2) + 0*11 + 0*(-13) + (-17)

FEEDFORWARD NEURAL NETWORKS: EXAMPLE

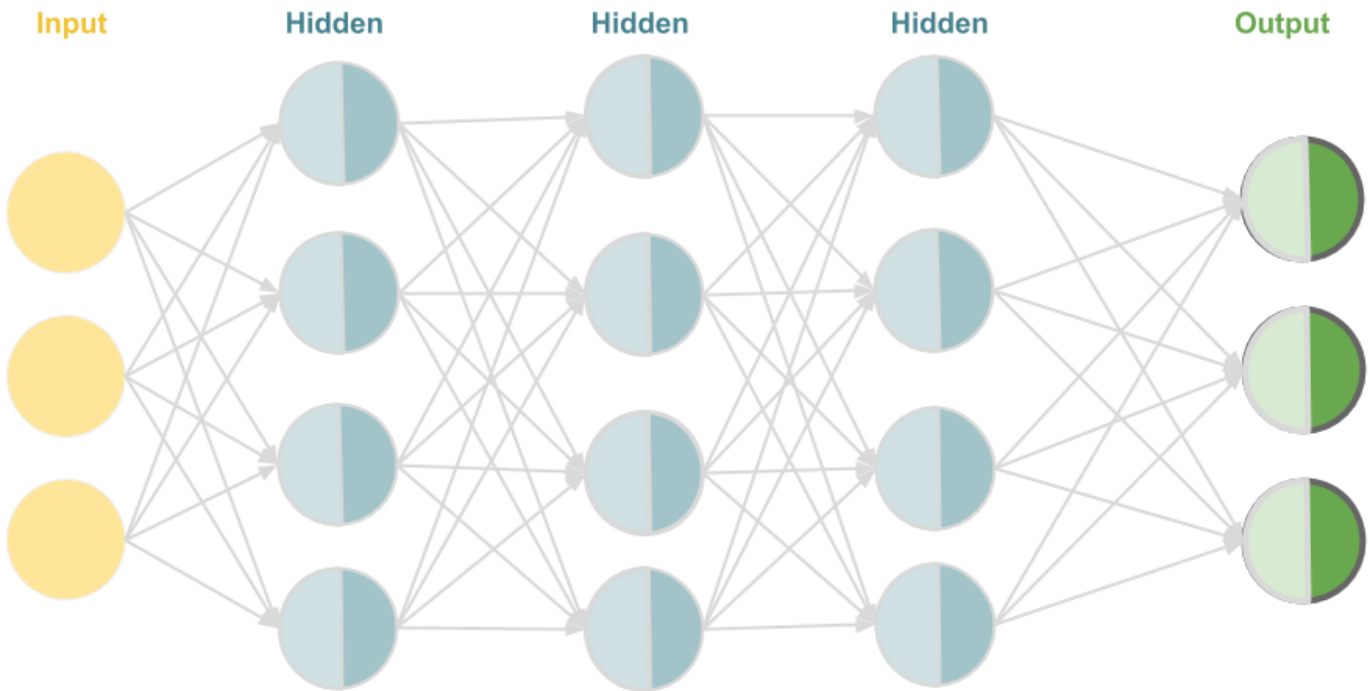


$\mathbf{x}$	$\mathbf{z}_{in}^{(1)}$	$\mathbf{z}^{(1)}$	$\mathbf{z}_{in}^{(2)}$	$\mathbf{z}^{(2)}$	$\mathbf{z}_{in}^{(3)}$	$\mathbf{z}^{(3)} = \mathbf{z}_{out}^{(3)} = \sigma(\mathbf{z}_{in}^{(3)})$
7	79	79	38	38	0	max(0, 0)
1	-70	0	21	21	24	max(0, 24)
-4	9	9	-96	0	-54	max(0, -54)
	-26	0	-35	0	17	max(0, 17)

FEEDFORWARD NEURAL NETWORKS: EXAMPLE

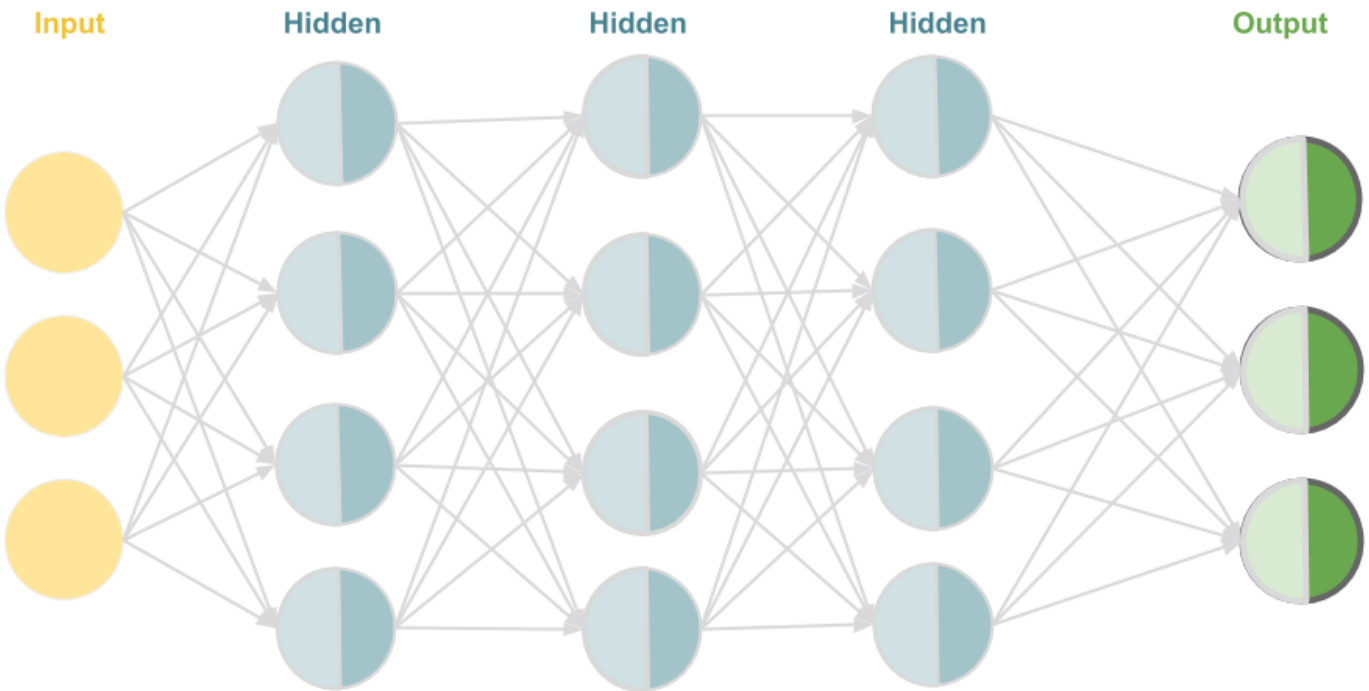


FEEDFORWARD NEURAL NETWORKS: EXAMPLE



$\mathbf{x}$	$\mathbf{z}_{in}^{(1)}$	$\mathbf{z}^{(1)}$	$\mathbf{z}_{in}^{(2)}$	$\mathbf{z}^{(2)}$	$\mathbf{z}_{in}^{(3)}$	$\mathbf{z}^{(3)}$	$\mathbf{f}_{in}$
7	79	79	38	38	0	0	3
1	-70	0	21	21	24	24	-1
-4	9	9	-96	0	-54	0	1
	-26	0	-35	0	17	17	

FEEDFORWARD NEURAL NETWORKS: EXAMPLE



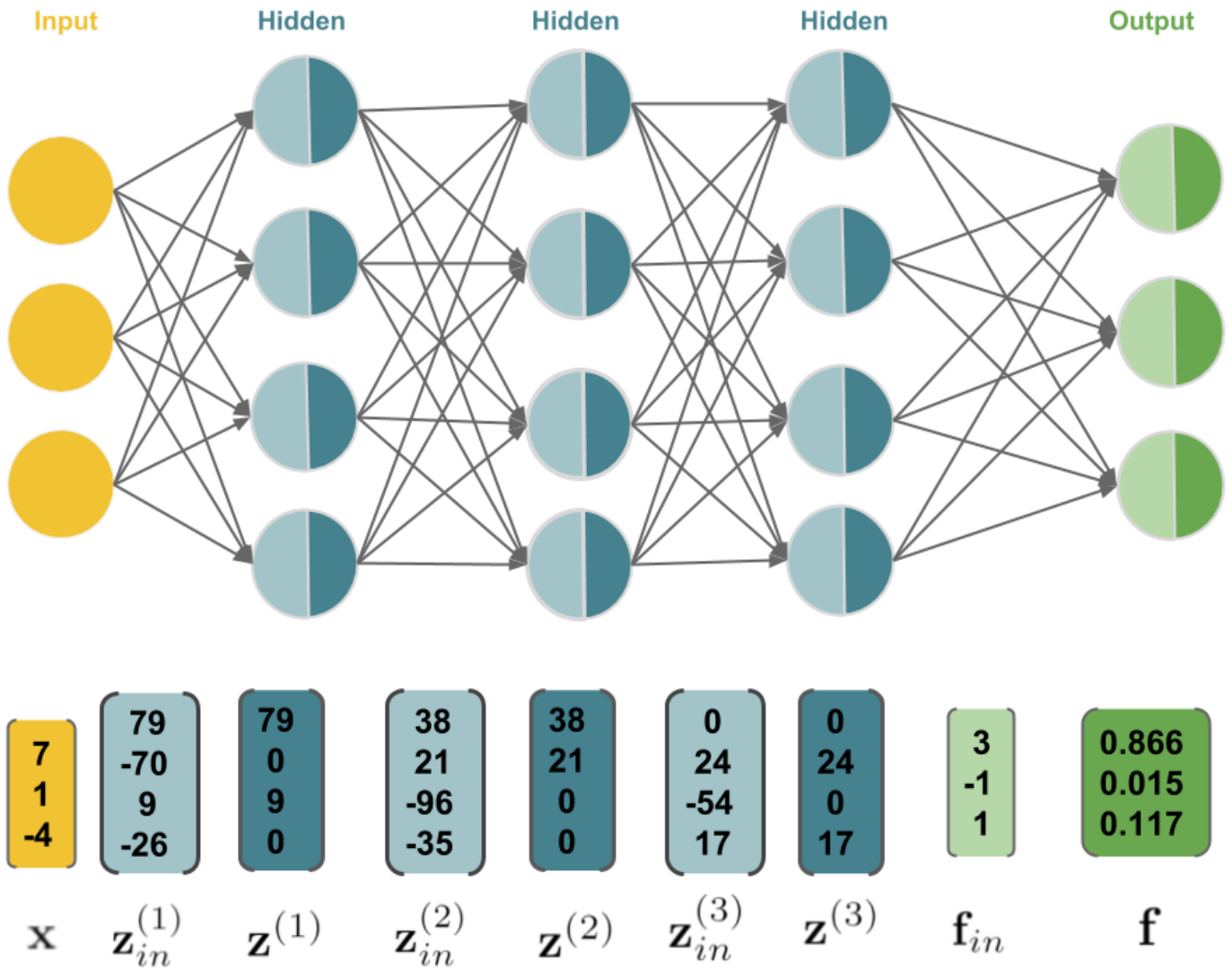
$$\mathbf{f}_{in}$$

3
-1
1

$$\frac{\exp(3)}{\exp 3+\exp(-1)+\exp(1)}$$
$$\frac{\exp(-1)}{\exp 3+\exp(-1)+\exp(1)}$$
$$\frac{\exp(1)}{\exp 3+\exp(-1)+\exp(1)}$$

$$\mathbf{f} = \mathbf{f}_{out} = \tau(\mathbf{f}_{in})$$

FEEDFORWARD NEURAL NETWORKS: EXAMPLE



WHY ADD MORE LAYERS?

- Multiple layers allow for the extraction of more and more abstract representations.
- Each layer in a feed-forward neural network adds its own degree of non-lnearity to the model.

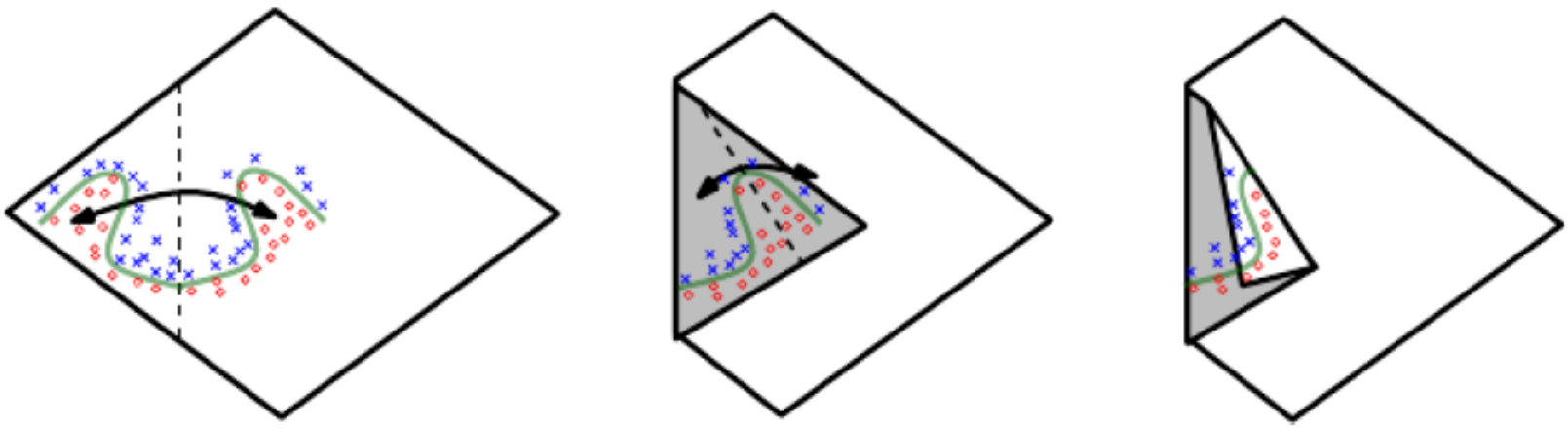


Figure: An intuitive, geometric explanation of the exponential advantage of deeper networks formally (Montúfar et al. (2014)).

DEEP NEURAL NETWORKS

Neural networks today can have hundreds of hidden layers. The greater the number of layers, the "deeper" the network. Historically DNNs were very challenging to train and not popular until the late ’00s for several reasons:

- The use of sigmoid activations (e.g., logistic sigmoid and tanh) significantly slowed down training due to a phenomenon known as “vanishing gradients”. The introduction of the ReLU activation largely solved this problem.
- Training DNNs on CPUs was too slow to be practical. Switching over to GPUs cut down training time by more than an order of magnitude.
- When dataset sizes are small, other models (such as SVMs) and techniques (such as feature engineering) often outperform them.

DEEP NEURAL NETWORKS

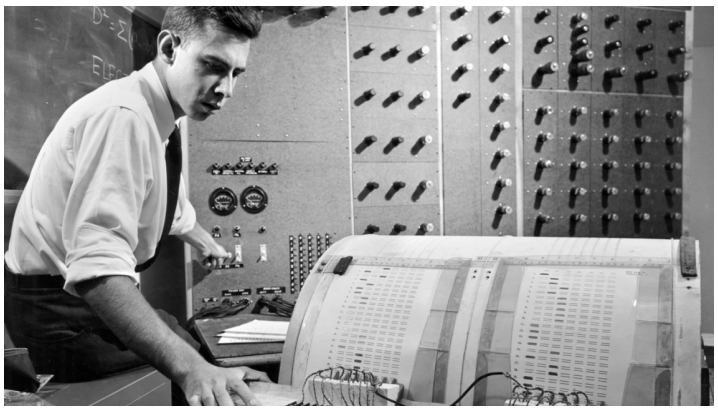
- The availability of large datasets and novel architectures that are capable of handling even complex tensor-shaped data (e.g. CNNs for image data), faster hardware, and better optimization and regularization methods made it feasible to successfully implement deep neural networks.

- An increase in depth often translates to an increase in performance on a given task. State-of-the-art neural networks, however, are much more sophisticated than the simple architectures we have encountered so far.

The term "**deep learning**" encompasses all of these developments and refers to the field as a whole.

Deep Learning

Brief History

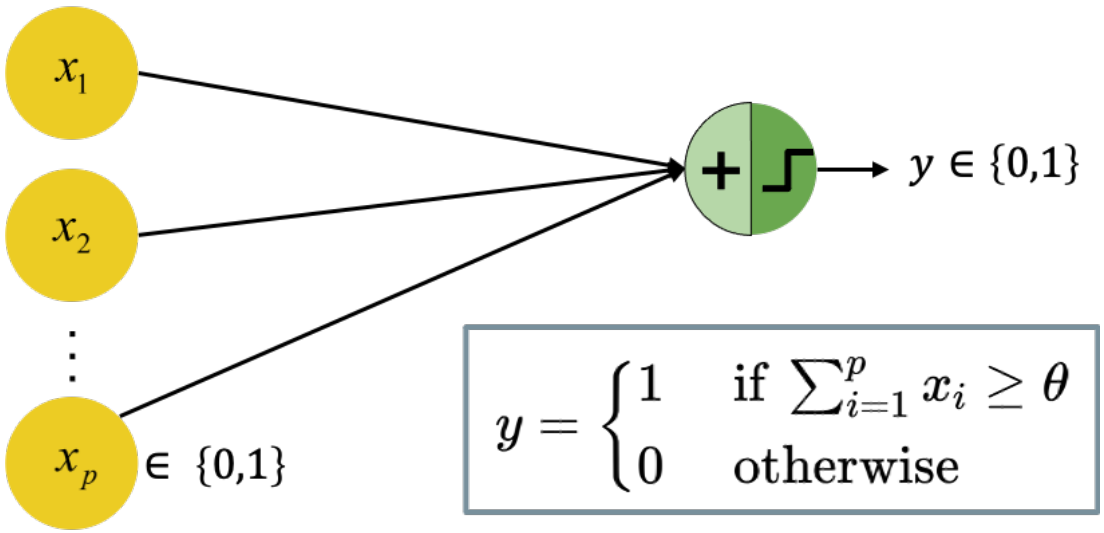


Learning goals

- Predecessors of modern (deep) neural networks
- History of DL as a field

A BRIEF HISTORY OF NEURAL NETWORKS

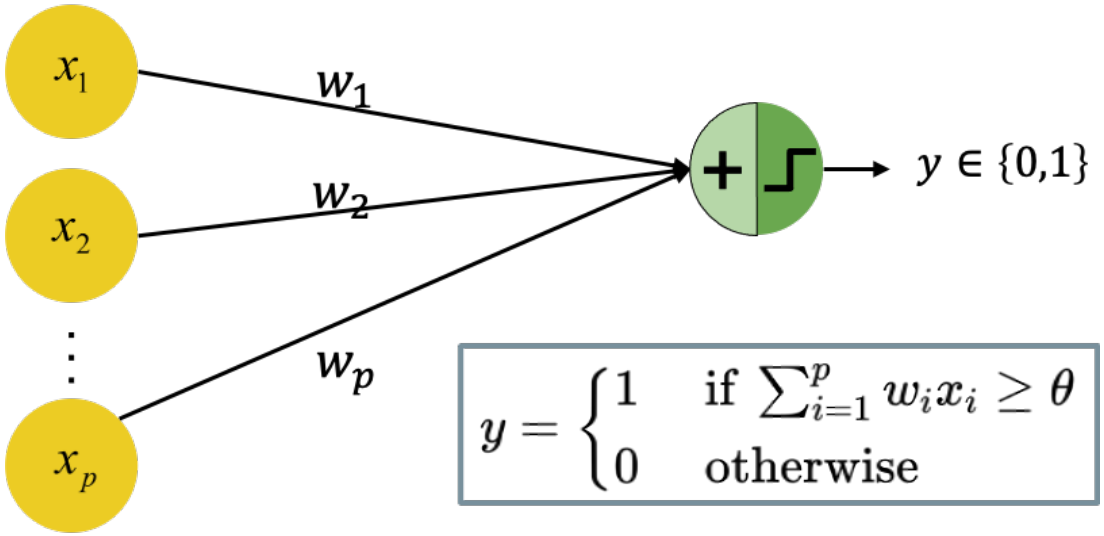
- **1943:** The first artificial neuron, the "Threshold Logic Unit (TLU)", was proposed by Warren McCulloch & Walter Pitts.



  - The model is limited to binary inputs.
  - It fires/outputs $+1$ if the input exceeds a certain threshold θ.
  - The weight are not adjustable, so learning could only be achieved by changing the threshold θ.

A BRIEF HISTORY OF NEURAL NETWORKS

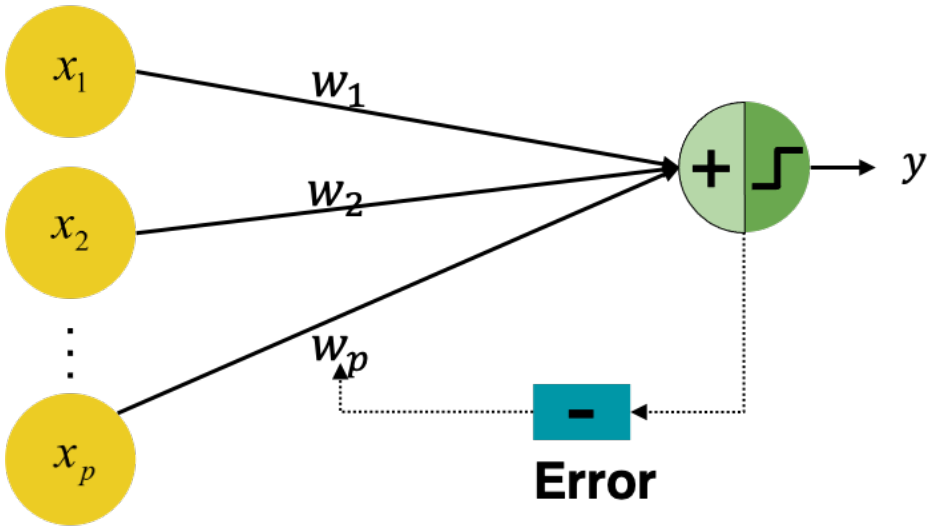
- **1957:** The perceptron was invented by Frank Rosenblatt.



  - The inputs are not restricted to be binary.
  - The weights are adjustable and can be learned by learning algorithms.
  - As for the TLU, the threshold is adjustable and decision boundaries are linear.

A BRIEF HISTORY OF NEURAL NETWORKS

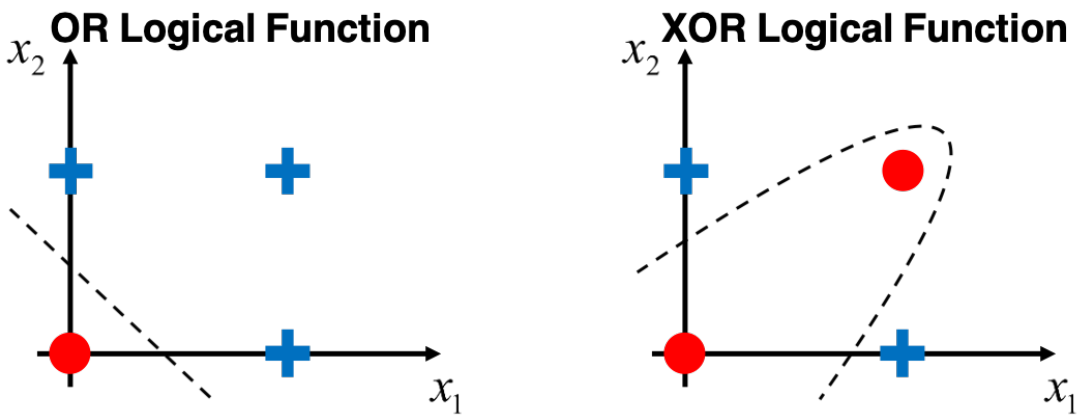
- **1960:** Adaptive Linear Neuron (ADALINE) was invented by Bernard Widrow & Ted Hoff; weights are now adjustable according to the weighted sum of the inputs.



- **1965:** Group method of data handling (also known as polynomial neural networks) by Alexey Ivakhnenko. The first learning algorithm for supervised deep feedforward multilayer perceptrons.

A BRIEF HISTORY OF NEURAL NETWORKS

- **1969:** The first “AI Winter” kicked in.
  - Marvin Minsky & Seymour Papert proved that a perceptron cannot solve the XOR-Problem (linear separability).
  - Less funding $\Rightarrow$ Standstill in AI/DL research.



- **1985:** Multilayer perceptron with backpropagation by David Rumelhart, Geoffrey Hinton, and Ronald Williams.
  - Efficiently compute derivatives of composite functions.
  - Backpropagation was developed already in 1970 by Linnainmaa.

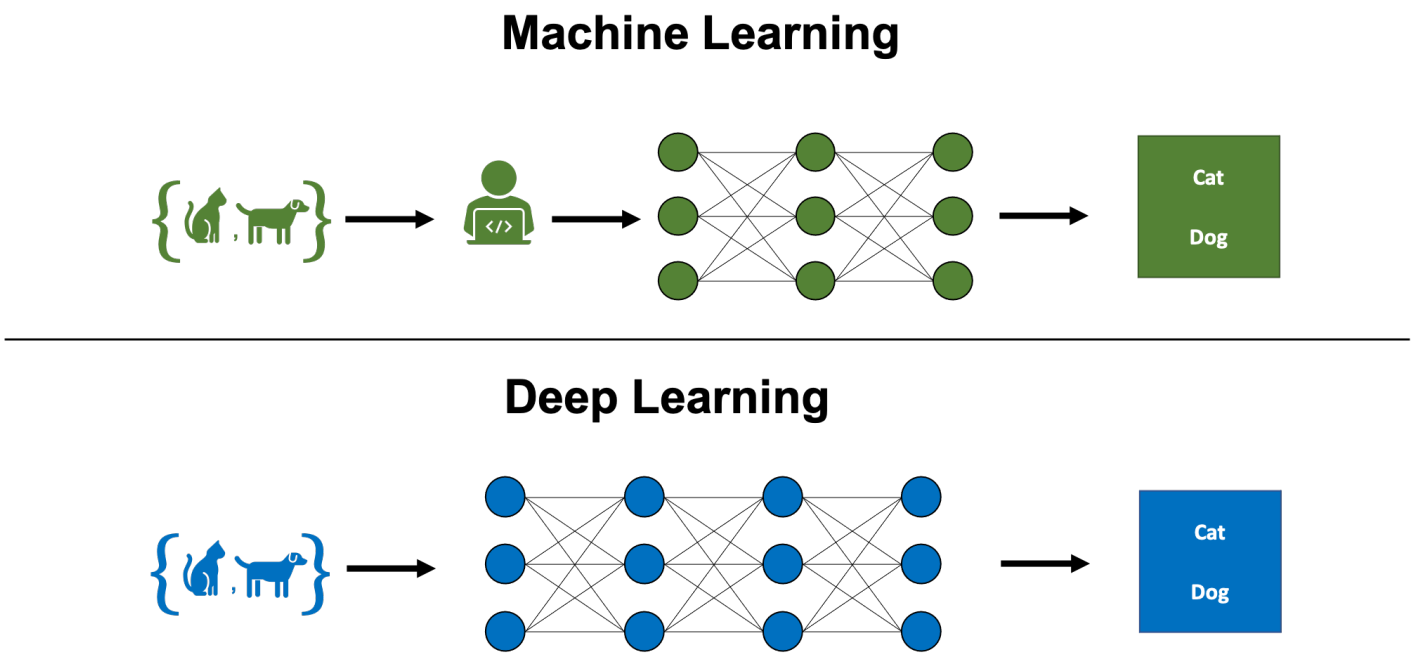
A BRIEF HISTORY OF NEURAL NETWORKS

- **1985:** The second “AI Winter” kicked in.
  - Overly optimistic expectations concerning potential of AI/DL.
  - The phrase “AI” even reached a pseudoscience status.
  - Kernel machines and graphical models both achieved good results on many important tasks.
  - Some fundamental mathematical difficulties in modeling long sequences were identified.

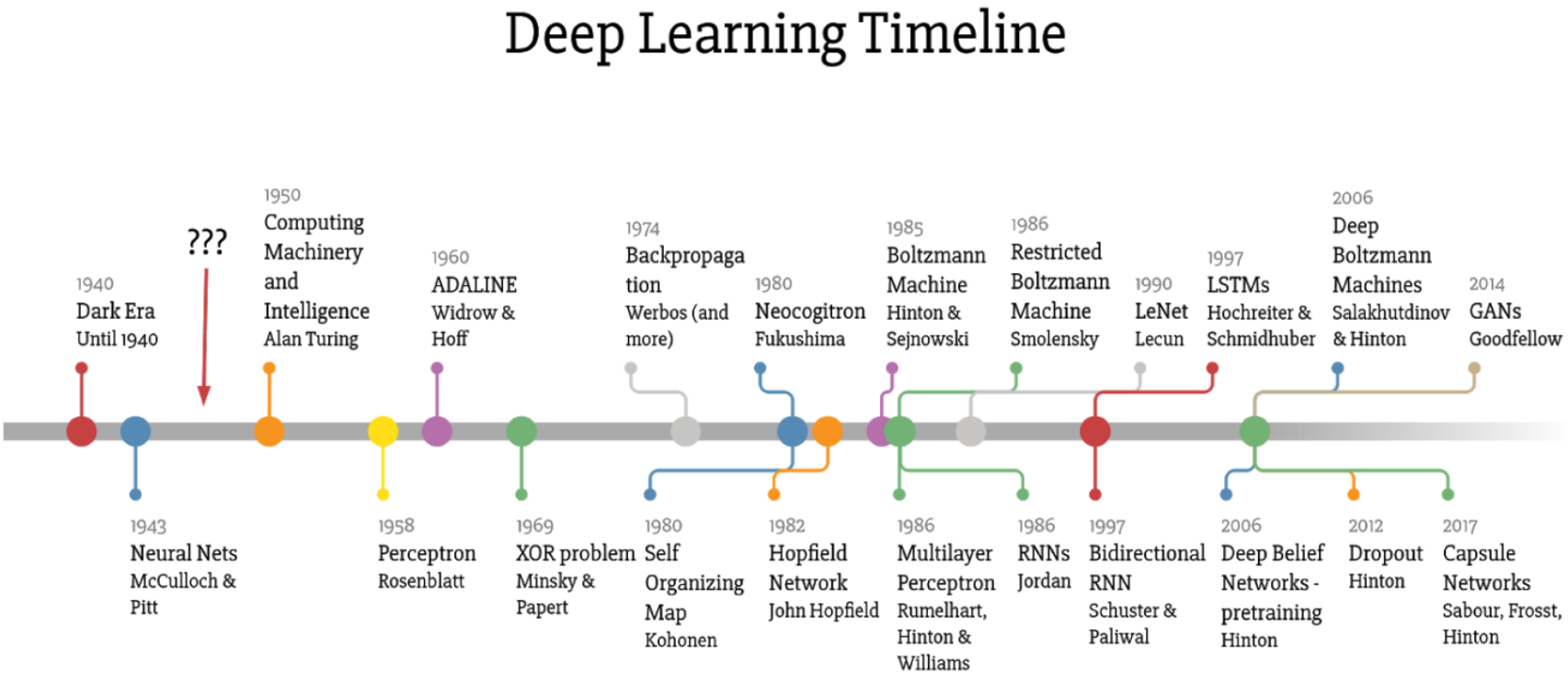
Credit: https://emerj.com/ai-executive-guides/will-there-be-another-artificial-intelligence-winter-probably-not/

A BRIEF HISTORY OF NEURAL NETWORKS

- **2006:** Age of deep neural networks began.
  - Geoffrey Hinton showed that a deep belief network could be efficiently trained using *greedy layer-wise pretraining*.
  - This wave of research popularized the use of the term deep learning to emphasize that researchers were now able to train deeper neural networks than had been possible before.
  - At this time, deep neural networks outperformed competing AI systems based on other ML technologies as well as hand-designed functionality.

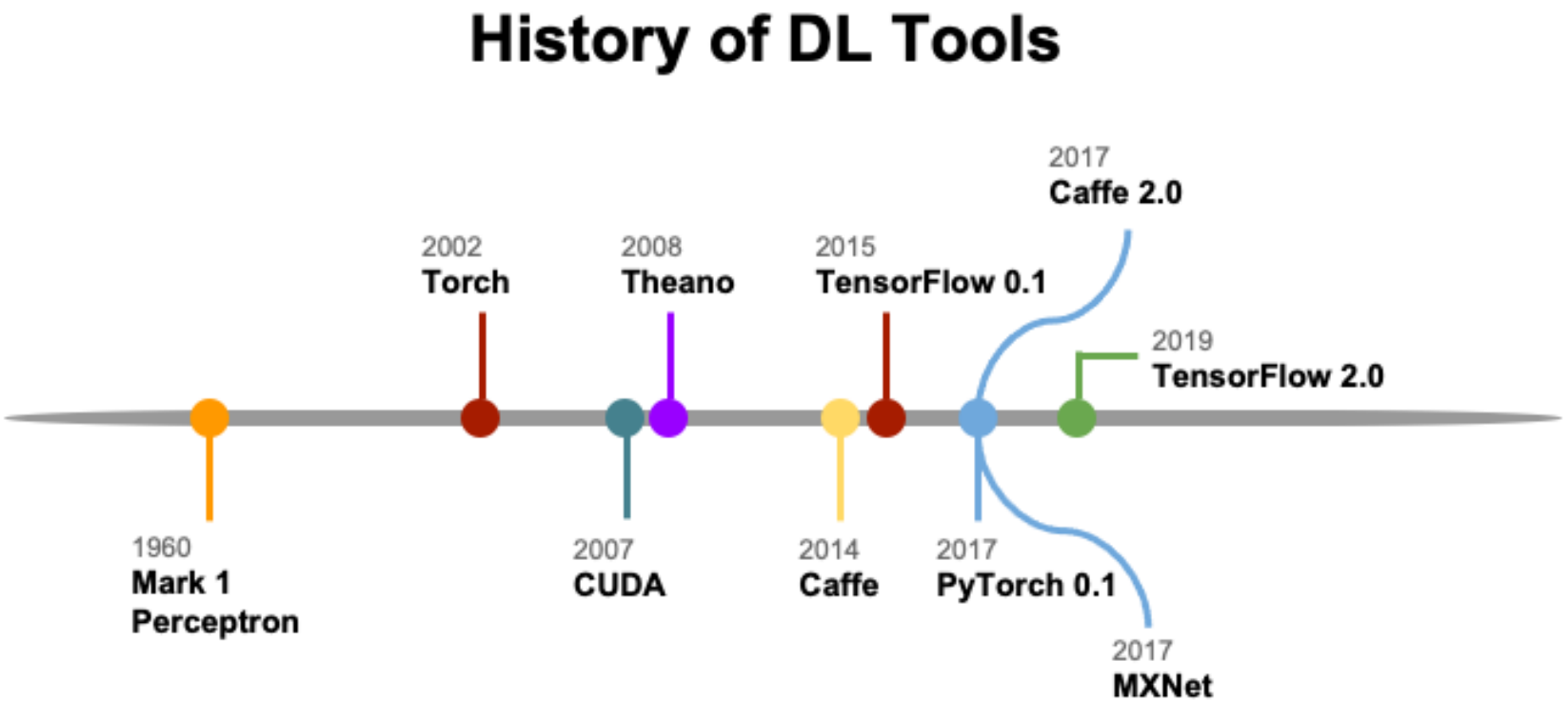


A BRIEF HISTORY OF NEURAL NETWORKS



Credit: https://towardsdatascience.com/a-weird-introduction-to-deep-learning-7828803693b0

A BRIEF HISTORY OF NEURAL NETWORKS



A BRIEF HISTORY OF NEURAL NETWORKS

Figure: IBM Supercomputer

- Watson is a question-answering system capable of answering questions posed in natural language, developed in IBM's DeepQA project.
- In 2011, Watson competed on *Jeopardy!* against champions Brad Rutter and Ken Jennings, winning the first place prize of $1 million.

A BRIEF HISTORY OF NEURAL NETWORKS

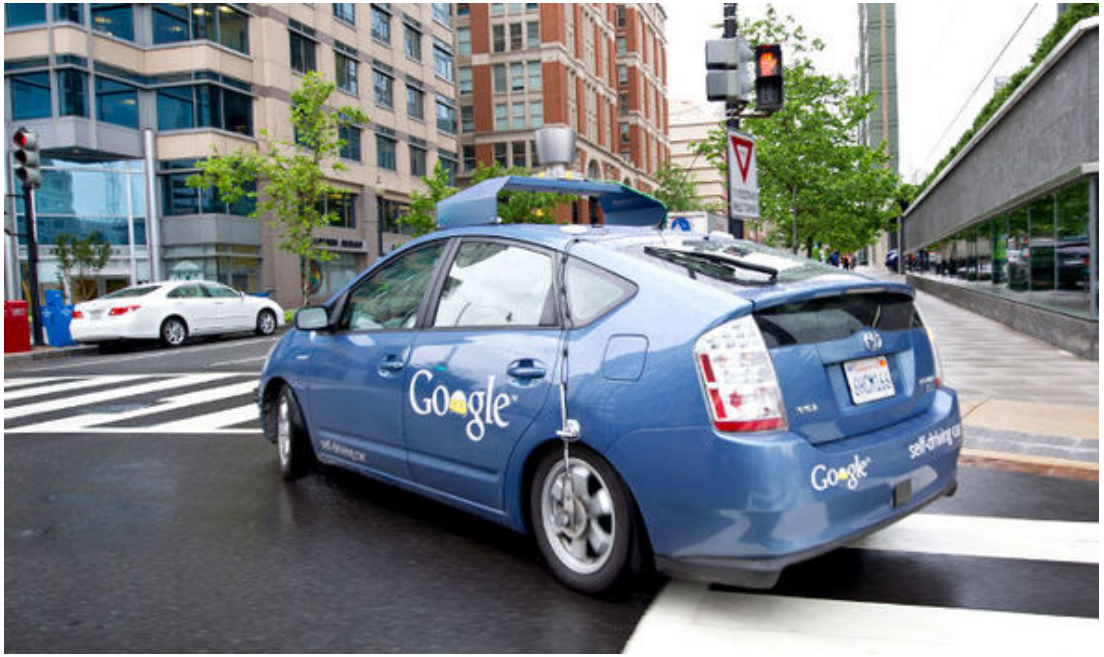


Figure: Google self driving car (Waymo)

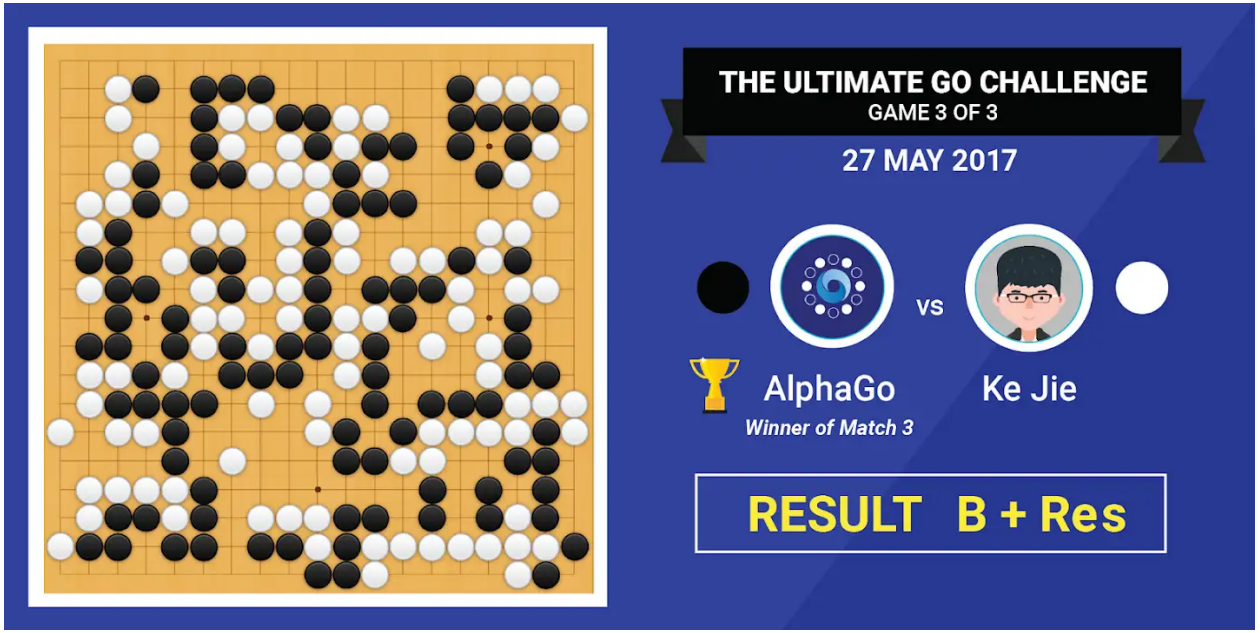
- Google's development of self-driving technology began on January 17, 2009, at the company's secretive X lab.
- By January 2020, 20 million miles of self-driving on public roads had been completed by Waymo.

A BRIEF HISTORY OF NEURAL NETWORKS

Credit: DeepMind

- **AlphaFold** is a deep learning system, developed by Google DeepMind, for determining a protein's 3D shape from its amino-acid sequence.
- In 2018 and 2020, AlphaFold placed first in the overall rankings of the Critical Assessment of Techniques for Protein Structure Prediction (CASP).

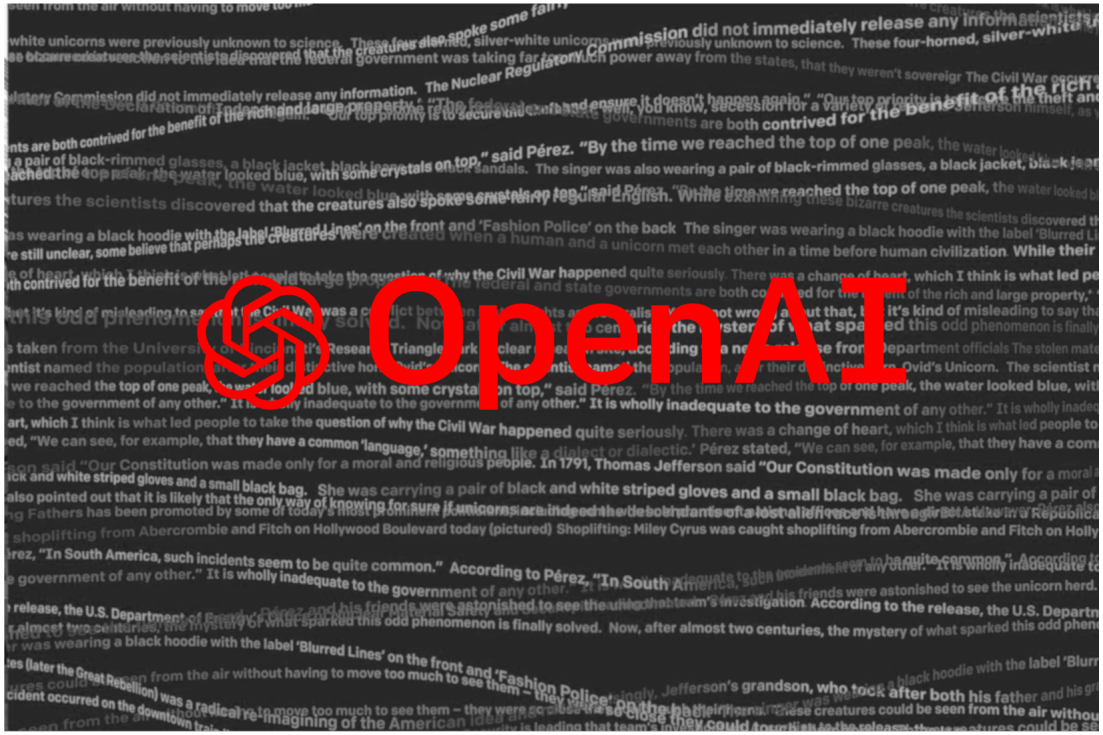
A BRIEF HISTORY OF NEURAL NETWORKS



Credit: DeepMind

- **AlphaGo**, originally developed by DeepMind, is a deep learning system that plays the board game Go. In 2017, the Master version of AlphaGo beat Ke Jie, the number one ranked player in the world at the time.
- While there are several extensions to AlphaGo (e.g., Master AlphaGo, AlphaGo Zero, AlphaZero, and MuZero), the main idea is the same: search for optimal moves based on knowledge acquired by machine learning.

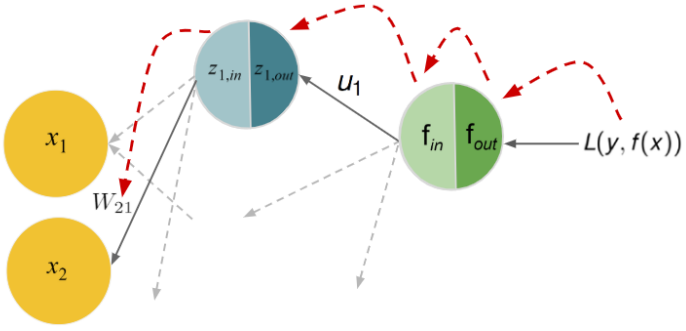
A BRIEF HISTORY OF NEURAL NETWORKS



- **Generative Pre-trained Transformer 3 (GPT-3)** is the third generatation of the GPT model, introduced by OpenAI in May 2020, to produce human-like text.
- There are 175 billion parameters to be learned by the algorithm, but the quality of the generated text is so high that it is hardly possible to distinguish it from a human-written text.

Deep Learning

Basic Backpropagation 1



Learning goals

- Forward and backward passes
- Chain rule
- Details of backprop

BACKPROPAGATION: BASIC IDEA

We would like to run ERM by GD on:

$$\mathcal{R}_{\text{emp}}(\boldsymbol{\theta}) = \frac{1}{n} \sum_{i=1}^{n} L\left(y^{(i)}, f\left(\mathbf{x}^{(i)} \mid \boldsymbol{\theta}\right)\right)$$

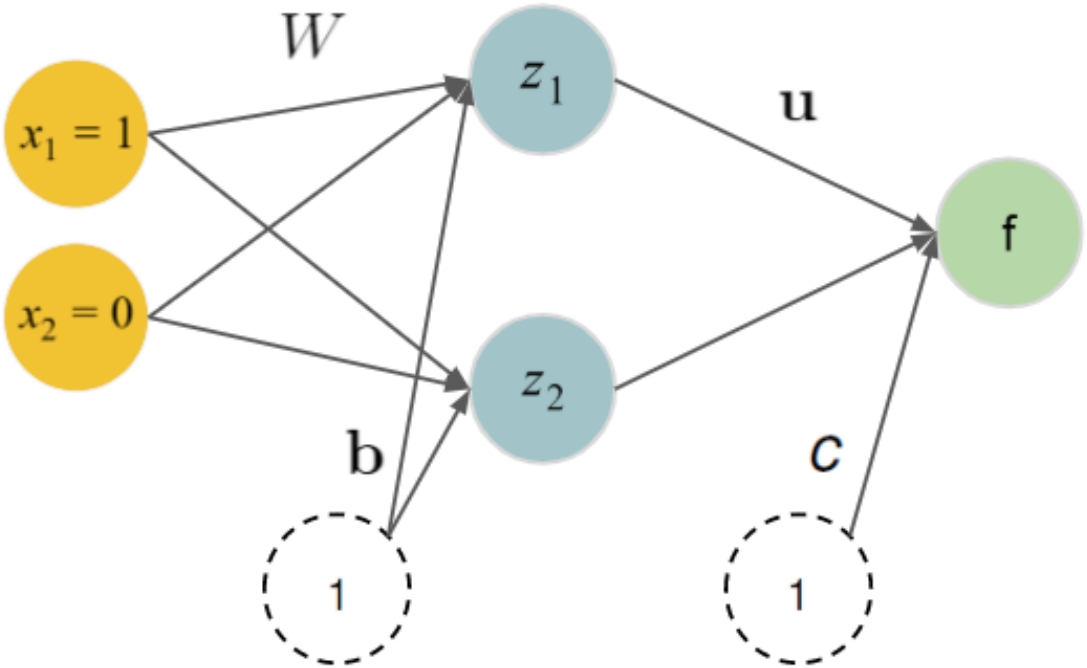
Backprop training of NNs runs in 2 alternating steps, for one $\mathbf{x}$:

1. **Forward pass:** Inputs flow through model to outputs. We then compute the observation loss. We covered that.
2. **Backward pass:** Loss flows backwards to update weights so error is reduced, as in GD.

We will see: This is simply (S)GD in disguise, cleverly using the chain rule, so we can reuse a lot of intermediate results.

XOR EXAMPLE

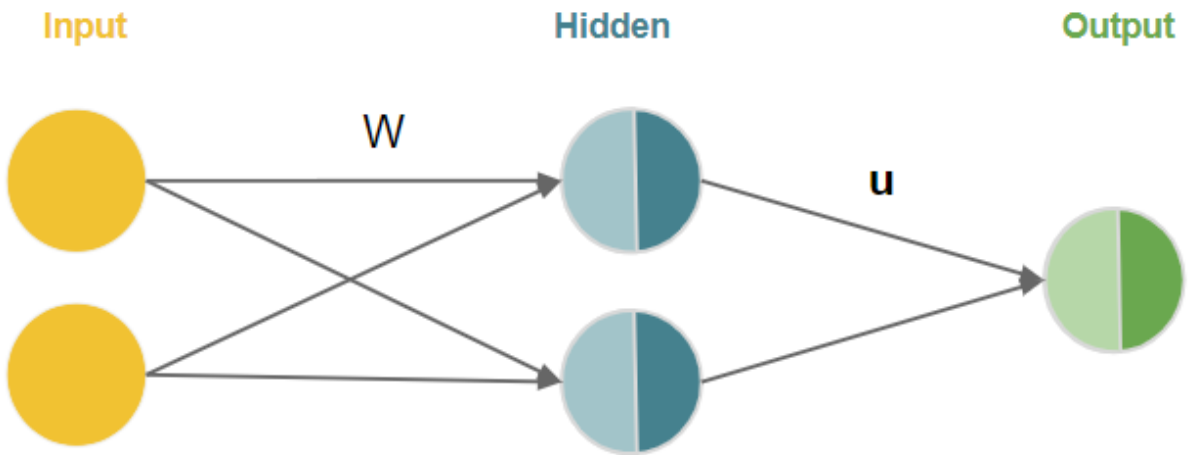
- As activations (hidden and outputs) we use the logistic.
- We run one FP and BP on $\mathbf{x} = (1, 0)^T$ with $y = 1$.
- We use L2 loss between 0-1 labels and the predicted probabilities. This is a bit uncommon, but computations become simpler.



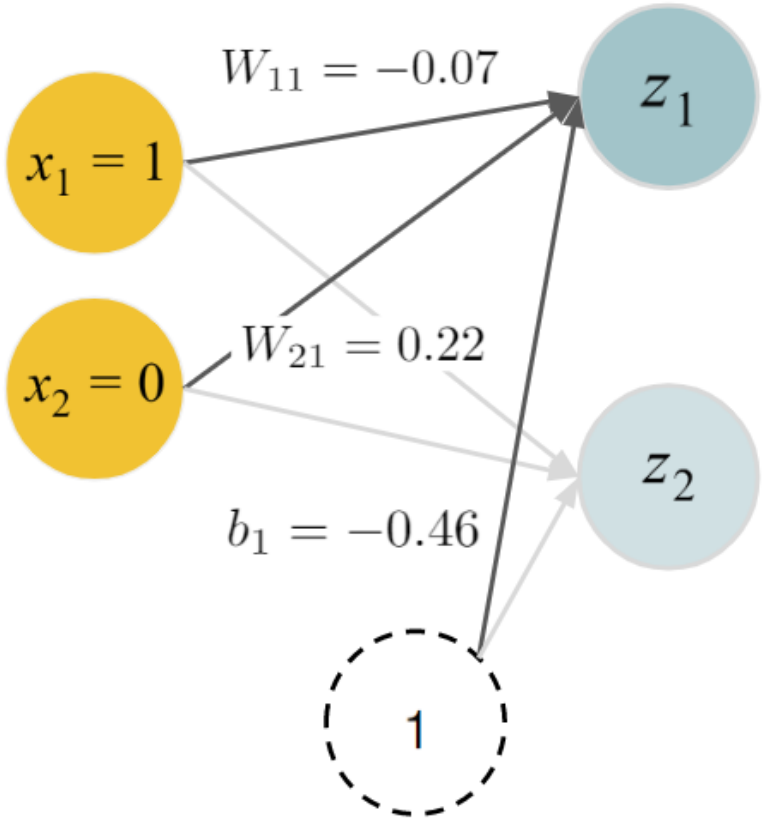
Note: We will only show rounded decimals.

FORWARD PASS

- We will divide the FP into four steps:
  - the inputs of z_i: $z_{i,in}$
  - the activations of z_i: $z_{i,out}$
  - the input of f: f_{in}
  - and finally the activation of f: f_{out}



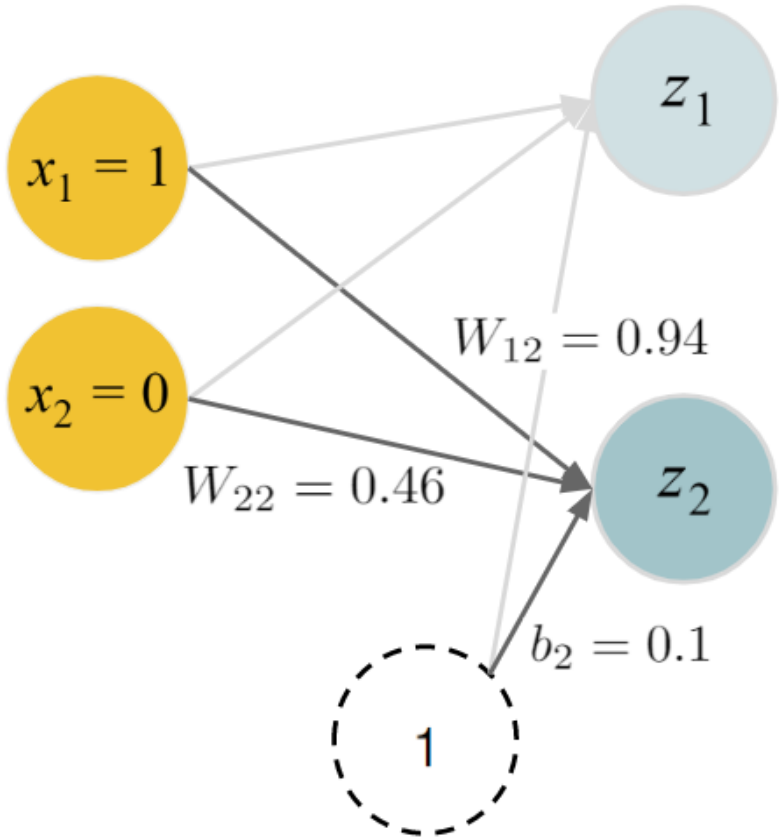
FORWARD PASS



$$z_{1,in} = \mathbf{W}_1^T \mathbf{x} + b_1 = 1 \cdot (-0.07) + 0 \cdot 0.22 + 1 \cdot (-0.46) = -0.53$$

$$z_{1,out} = \sigma(z_{1,in}) = \frac{1}{1 + \exp(-(-0.53))} = 0.3705$$

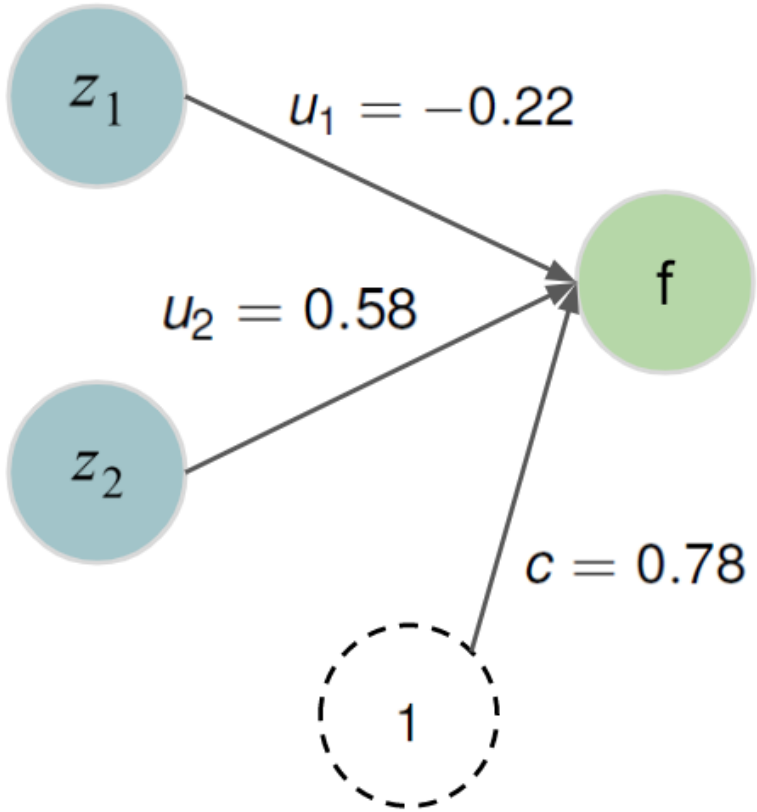
FORWARD PASS



$$z_{2,in} = \mathbf{W}_2^T \mathbf{x} + b_2 = 1 \cdot 0.94 + 0 \cdot 0.46 + 1 \cdot 0.1 = 1.04$$

$$z_{2,out} = \sigma\left(z_{2,in}\right) = \frac{1}{1 + \exp(-1.04)} = 0.7389$$

FORWARD PASS



$$f_{in} = \boldsymbol{u}^T \boldsymbol{z} + c = 0.3705 \cdot (-0.22) + 0.7389 \cdot 0.58 + 1 \cdot 0.78 = 1.1122$$

$$f_{out} = \tau(f_{in}) = \frac{1}{1 + \exp(-1.1122)} = 0.7525$$

FORWARD PASS

- The FP predicted $f_{out} = 0.7525$
- Now, we compare the prediction $f_{out} = 0.7525$ and the true label $y = 1$ using the L2-loss:

$$\begin{aligned} L(y, f(\mathbf{x})) &= \frac{1}{2}(y - f\left(\mathbf{x}^{(i)} \mid \boldsymbol{\theta}\right))^2 = \frac{1}{2}(y - f_{out})^2 \\ &= \frac{1}{2}(1 - 0.7525)^2 = 0.0306 \end{aligned}$$

- The calculation of the gradient is performed backwards (starting from the output layer), so that results can be reused.

BACKWARD PASS

The main ingredients of the backward pass are:

- to reuse the results of the forward pass
  (here: $z_{i,in}$, $z_{i,out}$, f_{in}, f_{out})
- reuse the intermediate results from the chain rule
- the derivative of the activations and some affine functions

BACKWARD PASS

- Let's start to update u_1. We recursively apply the chain rule:

$$\frac{\partial L(y, f(\mathbf{x}))}{\partial u_1} = \frac{\partial L(y, f(\mathbf{x}))}{\partial f_{out}} \cdot \frac{\partial f_{out}}{\partial f_{in}} \cdot \frac{\partial f_{in}}{\partial u_1}$$

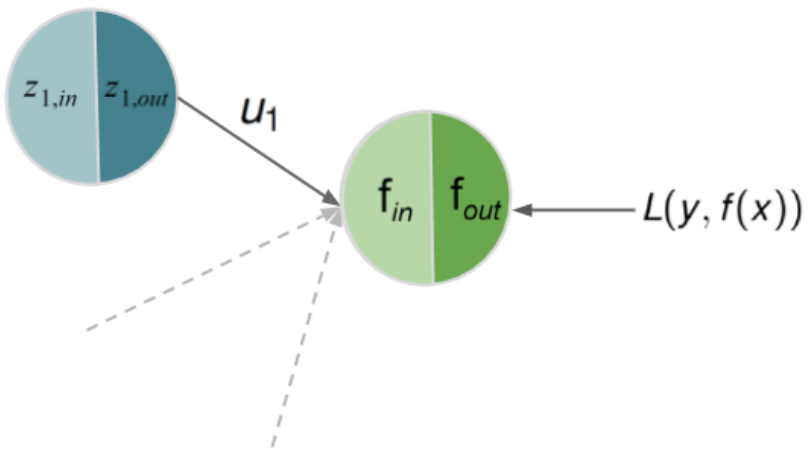


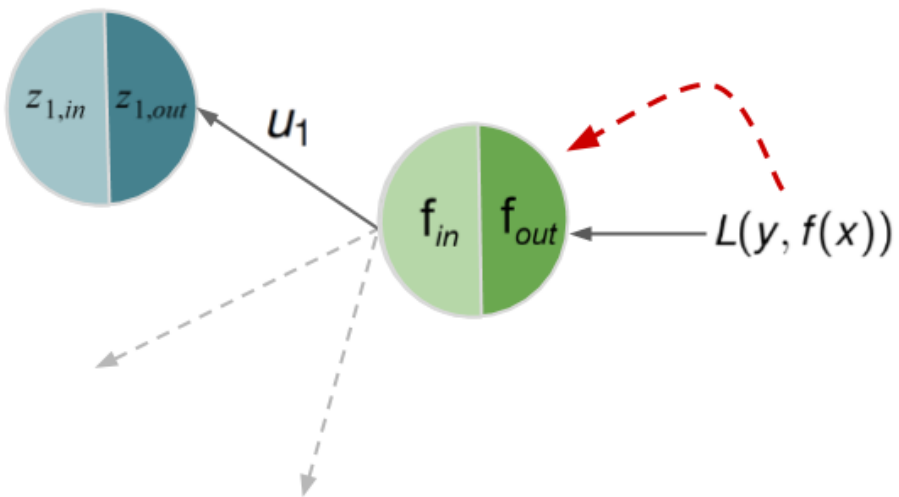
Figure: Snippet from our NN, with backward path for u_1.

BACKWARD PASS

- 1st step: The derivative of L2 is easy; we know f_{out} from FP.

$$\frac{\partial L(y, f(\mathbf{x}))}{\partial f_{out}} = \frac{d}{\partial f_{out}} \frac{1}{2}(y - f_{out})^2 = -\underbrace{(y - f_{out})}_{\hat{=}\text{residual}}$$

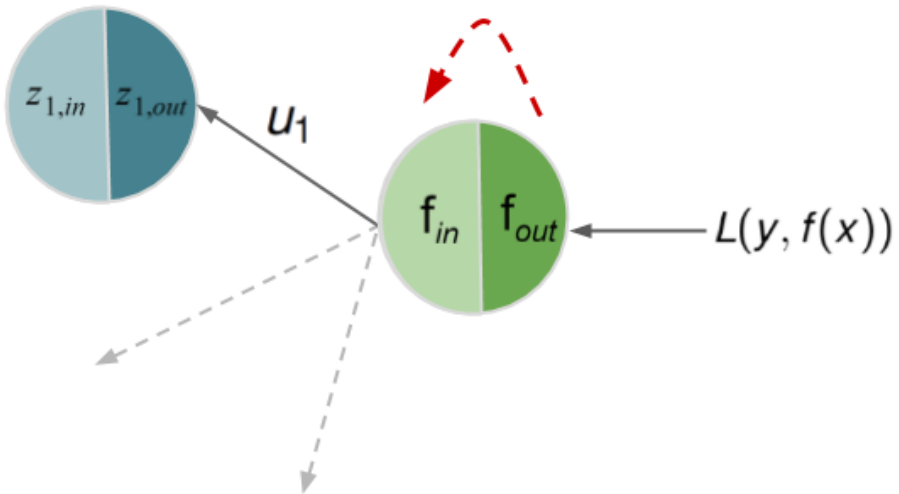
$$= -(1 - 0.7525) = -0.2475$$



BACKWARD PASS

- 2nd step. $f_{out} = \sigma(f_{in})$, use rule for σ', use f_{in} from FP.

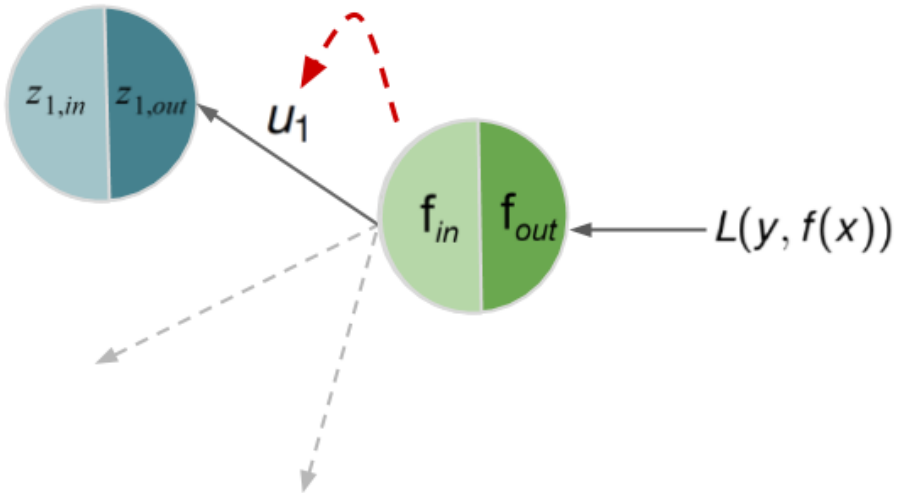
$$\begin{aligned} \frac{\partial f_{out}}{\partial f_{in}} &= \sigma(f_{in}) \cdot (1 - \sigma(f_{in})) \\ &= 0.7525 \cdot (1 - 0.7525) = 0.1862 \end{aligned}$$



BACKWARD PASS

- 3rd step. Derivative of the linear input is easy; use $z_{1,out}$ from FP.

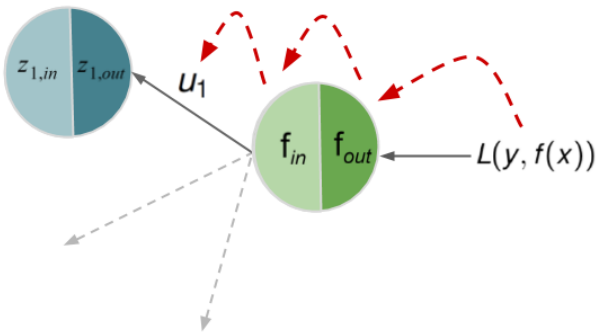
$$\frac{\partial f_{in}}{\partial u_1} = \frac{\partial(u_1 \cdot z_{1,out} + u_2 \cdot z_{2,out} + c \cdot 1)}{\partial u_1} = z_{1,out} = 0.3705$$



BACKWARD PASS

- Plug it together:

$$\begin{aligned}\frac{\partial L(y, f(\mathbf{x}))}{\partial u_1} &= \frac{\partial L(y, f(\mathbf{x}))}{\partial f_{out}} \cdot \frac{\partial f_{out}}{\partial f_{in}} \cdot \frac{\partial f_{in}}{\partial u_1} \\ &= -0.2475 \cdot 0.1862 \cdot 0.3705 = -0.0171\end{aligned}$$



- With LR $\alpha = 0.5$:

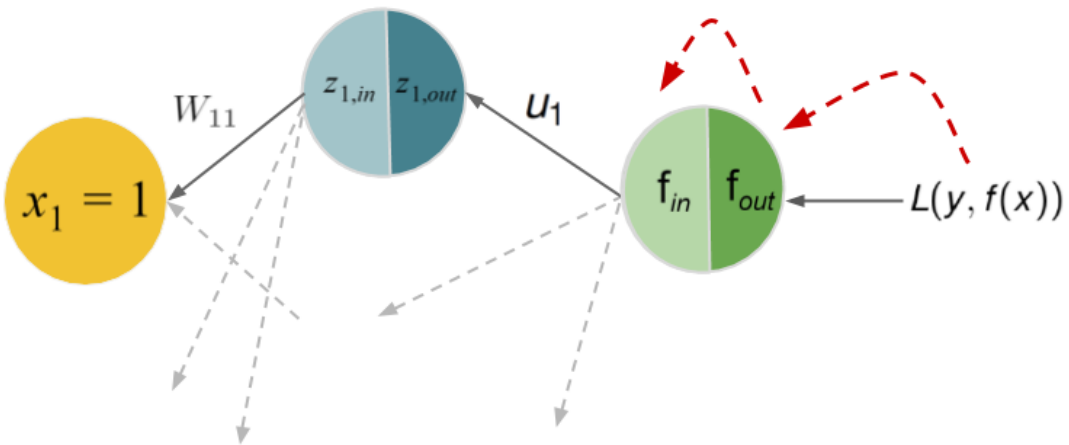
$$\begin{aligned}u_1^{[new]} &= u_1^{[old]} - \alpha \cdot \frac{\partial L(y, f(\mathbf{x}))}{\partial u_1} \\ &= -0.22 - 0.5 \cdot (-0.0171) = -0.2115\end{aligned}$$

BACKWARD PASS

- Now for W_{11}:

$$\frac{\partial L(y, f(\mathbf{x}))}{\partial W_{11}} = \frac{\partial L(y, f(\mathbf{x}))}{\partial f_{out}} \cdot \frac{\partial f_{out}}{\partial f_{in}} \cdot \frac{\partial f_{in}}{\partial z_{1,out}} \cdot \frac{\partial z_{1,out}}{\partial z_{1,in}} \cdot \frac{\partial z_{1,in}}{\partial W_{11}}$$

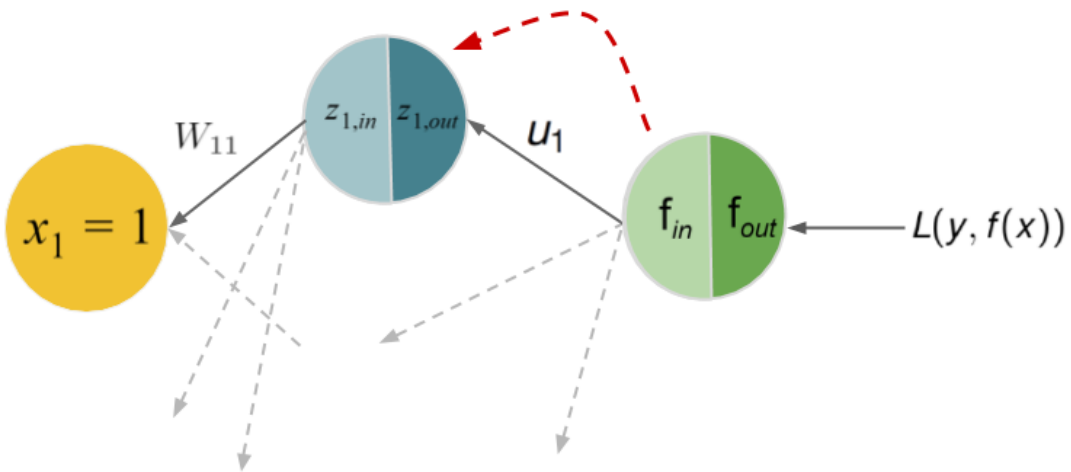
- We know $\frac{\partial L(y,f(\mathbf{x}))}{\partial f_{out}}$ and $\frac{\partial f_{out}}{\partial f_{in}}$ from BP for u_1.



BACKWARD PASS

- $f_{in} = u_1 \cdot z_{1,out} + u_2 \cdot z_{2,out} + c \cdot 1$ is linear, easy and we know u_1 :

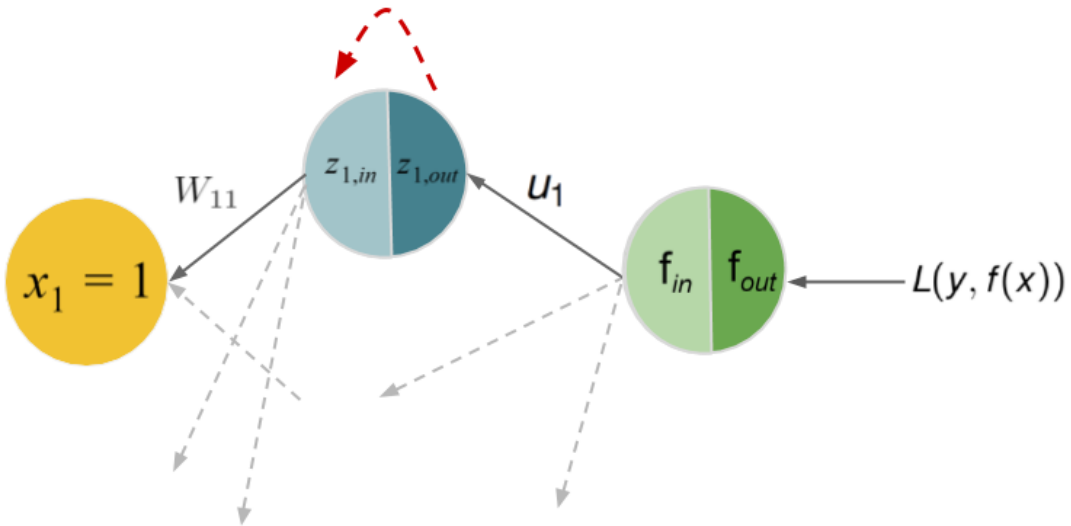
$$\frac{\partial f_{in}}{\partial z_{1,out}} = u_1 = -0.22$$



BACKWARD PASS

- Next. Use rule for σ' and FP results:

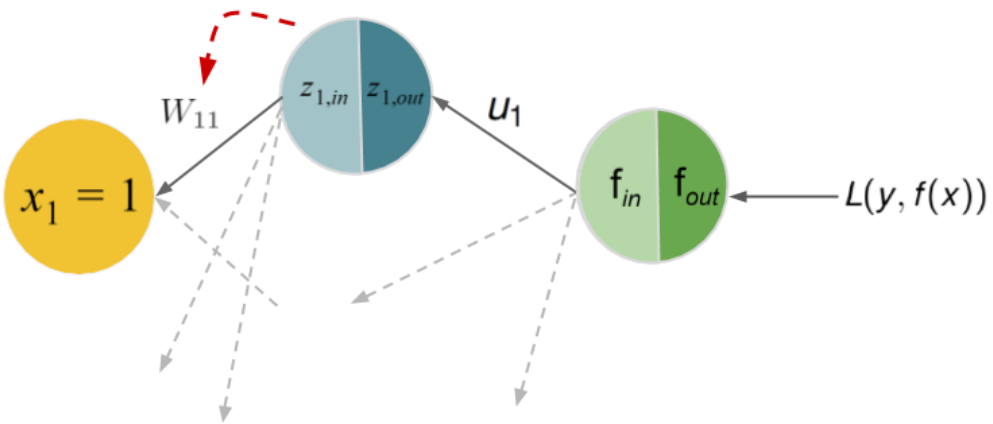
$$\begin{aligned} \frac{\partial z_{1,out}}{\partial z_{1,in}} &= \sigma(z_{1,in}) \cdot (1 - \sigma(z_{1,in})) \\ &= 0.3705 \cdot (1 - 0.3705) = 0.2332 \end{aligned}$$



BACKWARD PASS

- $z_{1,in} = x_1 \cdot W_{11} + x_2 \cdot W_{21} + b_1 \cdot 1$ is linear and depends on inputs:

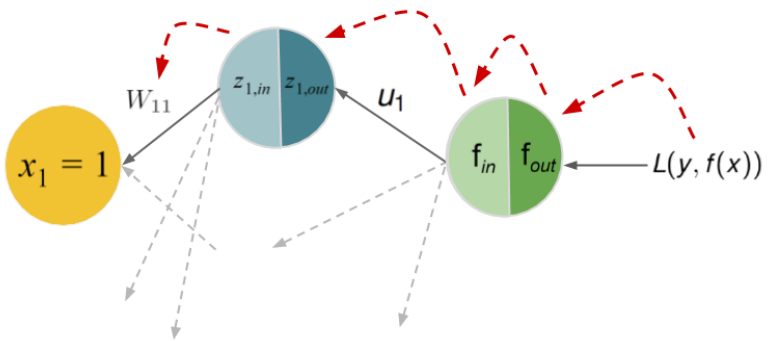
$$\frac{\partial z_{1,in}}{\partial W_{11}} = x_1 = 1$$



BACKWARD PASS

- Plugging together:

$$\begin{aligned}
\frac{\partial L(y, f(\mathbf{x}))}{\partial W_{11}} &= \frac{\partial L(y, f(\mathbf{x}))}{\partial f_{out}} \cdot \frac{\partial f_{out}}{\partial f_{in}} \cdot \frac{\partial f_{in}}{\partial z_{1,out}} \cdot \frac{\partial z_{1,out}}{\partial z_{1,in}} \cdot \frac{\partial z_{1,in}}{\partial W_{11}} \\
&= (-0.2475) \cdot 0.1862 \cdot (-0.22) \cdot 0.2332 \cdot 1 \\
&= 0.0024
\end{aligned}$$



- Full SGD update:

$$\begin{aligned}
W_{11}^{[new]} &= W_{11}^{[old]} - \alpha \cdot \frac{\partial L(y, f(\mathbf{x}))}{\partial W_{11}} \\
&= -0.07 - 0.5 \cdot 0.0024 = -0.0712
\end{aligned}$$

RESULT

- We can do this for all weights:

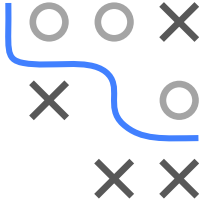
$$W = \begin{pmatrix} -0.0712 & 0.9426 \\ 0.22 & 0.46 \end{pmatrix}, b = \begin{pmatrix} -0.4612 \\ 0.1026 \end{pmatrix},$$

$$u = \begin{pmatrix} -0.2115 \\ 0.5970 \end{pmatrix} \text{ and } c = 0.8030.$$

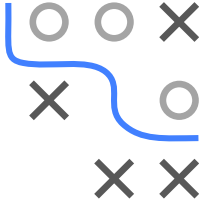
- Yields $f(\mathbf{x} \mid \boldsymbol{\theta}^{[new]}) = 0.7615$ and loss $\frac{1}{2}(1 - 0.7615)^2 = 0.0284$.
- Before, we had $f(\mathbf{x} \mid \boldsymbol{\theta}^{[old]}) = 0.7525$ and higher loss 0.0306.

Now rinse and repeat. This was one training iter, we do thousands.

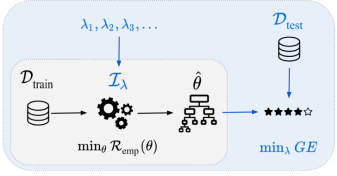
INTRODUCTION TO MACHINE LEARNING



Introduction to Machine Learning



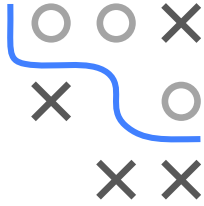
Hyperparameter Tuning
Introduction



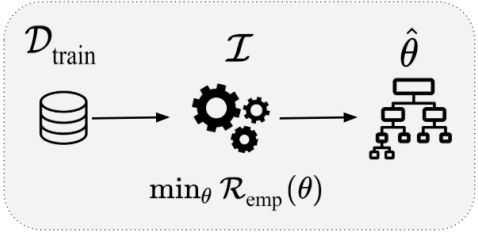
Learning goals

- Understand the difference between model parameters and hyperparameters
- Know different types of hyperparameters
- Be able to explain the goal of hyperparameter tuning

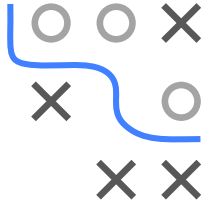
MOTIVATING EXAMPLE



- Given a data set, we want to train a classification tree.
- We feel that a maximum tree depth of 4 has worked out well for us previously, so we decide to set this hyperparameter to 4.
- The learner ("inducer") $\mathcal{I}$ takes the input data, internally performs **empirical risk minimization**, and returns a fitted tree model $\hat{f}(\mathbf{x}) = f(\mathbf{x}, \hat{\theta})$ of at most depth $\lambda = 4$ that minimizes empirical risk.

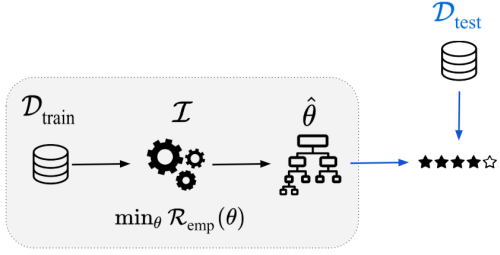


MOTIVATING EXAMPLE

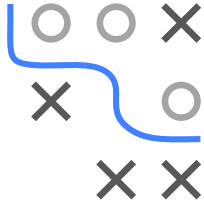


- We are **actually** interested in the **generalization performance** $\mathrm{GE}\left(\hat{f}\right)$ of the estimated model on new, previously unseen data.
- We estimate the generalization performance by evaluating the model $\hat{f} = \mathcal{I}(\mathcal{D}_{\text{train}}, \boldsymbol{\lambda})$ on a test set $\mathcal{D}_{\text{test}}$:

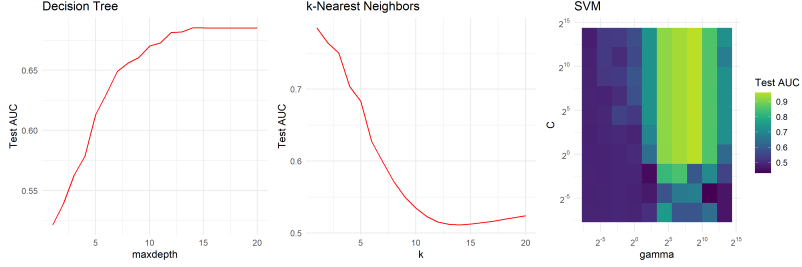
$$\widehat{\mathrm{GE}}_{\mathcal{D}_{\text{train}}, \mathcal{D}_{\text{test}}}(\mathcal{I}, \boldsymbol{\lambda}, n_{\text{train}}, \rho) = \rho\left(\mathbf{y}_{\mathcal{D}_{\text{test}}}, \boldsymbol{F}_{\mathcal{D}_{\text{test}}, \hat{f}}\right)$$



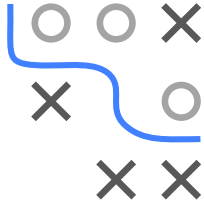
MOTIVATING EXAMPLE



- But many ML algorithms are sensitive w.r.t. a good setting of their hyperparameters, and generalization performance might be bad if we have chosen a suboptimal configuration.
- Consider a simulation example of 3 ML algorithms below, where we use the dataset *mlbench.spiral* and 10,000 testing points. As can be seen, variating hyperparameters can lead to big difference in model's generalization performance.



MOTIVATING EXAMPLE

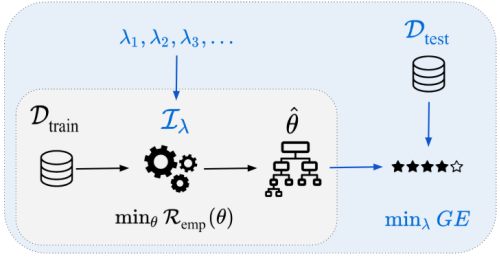


For our example this could mean:

- Data too complex to be modeled by a tree of depth 4
- Data much simpler than we thought, a tree of depth 4 overfits

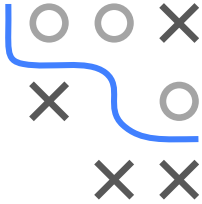
$\Longrightarrow$ Algorithmically try out different values for the tree depth. For each maximum depth λ, we have to train the model **to completion** and evaluate its performance on the test set.

- We choose the tree depth λ that is **optimal** w.r.t. the generalization error of the model.



MODEL PARAMETERS VS. HYPERPARAMETERS

It is critical to understand the difference between model parameters and hyperparameters.



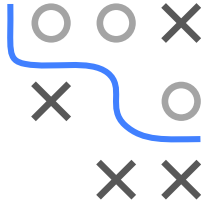
Model parameters θ are optimized during training. They are an **output** of the training.

Examples:

- The splits and terminal node constants of a tree learner
- Coefficients θ of a linear model $f(\mathbf{x}) = \theta^\top \mathbf{x}$

MODEL PARAMETERS VS. HYPERPARAMETERS

In contrast, **hyperparameters** (HPs) λ are not optimized during training. They must be specified in advance, are an **input** of the training. Hyperparameters often control the complexity of a model, i.e., how flexible the model is. They can in principle influence any structural property of a model or computational part of the training process.



The process of finding the best hyperparameters is called **tuning**.

Examples:

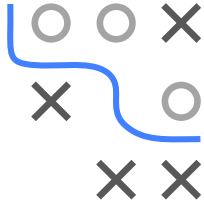
- Maximum depth of a tree
- k and which distance measure to use for k-NN
- Number and maximal order of interactions to be included in a linear regression model
- Number of optimization steps if the empirical risk minimization is done via gradient descent

TYPES OF HYPERPARAMETERS

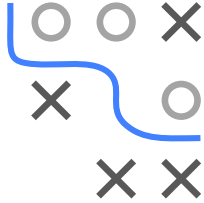
We summarize all hyperparameters we want to tune in a vector $\boldsymbol{\lambda} \in \boldsymbol{\Lambda}$ of (possibly) mixed type. HPs can have different types:

- Real-valued parameters, e.g.:
  - Minimal error improvement in a tree to accept a split
  - Bandwidths of the kernel density estimates for Naive Bayes
- Integer parameters, e.g.:
  - Neighborhood size k for k-NN
  - *mtry* in a random forest
- Categorical parameters, e.g.:
  - Which split criterion for classification trees?
  - Which distance measure for k-NN?

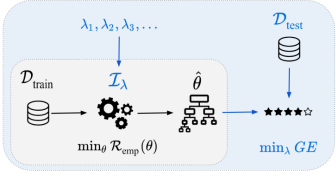
Hyperparameters are often **hierarchically dependent** on each other, e.g., *if* we use a kernel-density estimate for Naive Bayes, what is its width?



Introduction to Machine Learning



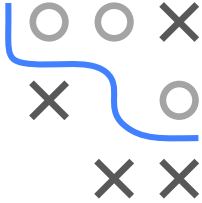
Hyperparameter Tuning
Problem Definition



Learning goals

- Definition of HPO objective and components
- Understand its properties
- What makes tuning challenging

HYPERPARAMETER OPTIMIZATION



Hyperparameters (HP) $\boldsymbol{\lambda}$ are parameters that are *inputs* to learner $\mathcal{I}$ which performs ERM on training data set to find optimal **model parameters** $\boldsymbol{\theta}$. HPs can influence the generalization performance in a non-trivial and subtle way.

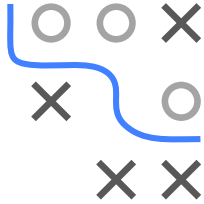
Hyperparameter optimization (HPO) / **Tuning** is the process of finding a well-performing hyperparameter configuration (HPC) $\boldsymbol{\lambda} \in \tilde{\boldsymbol{\Lambda}}$ for an learner $\mathcal{I}_{\boldsymbol{\lambda}}$.

OBJECTIVE AND SEARCH SPACE

Search space $\tilde{\boldsymbol{\Lambda}} \subset \boldsymbol{\Lambda}$ with all optimized HPs and ranges:

$$\tilde{\boldsymbol{\Lambda}} = \tilde{\boldsymbol{\Lambda}}_1 \times \tilde{\boldsymbol{\Lambda}}_2 \times \cdots \times \tilde{\boldsymbol{\Lambda}}_I$$

where $\tilde{\boldsymbol{\Lambda}}_i$ is a bounded subset of the domain of the i-th HP $\boldsymbol{\Lambda}_i$, and can be either continuous, discrete, or categorical.

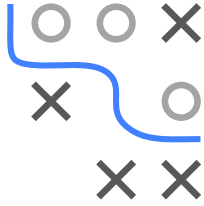


The general HPO problem is defined as:

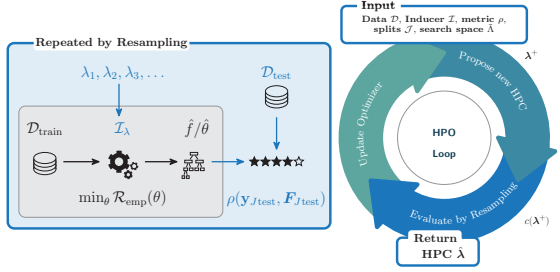
$$\boldsymbol{\lambda}^* \in \arg\min_{\boldsymbol{\lambda} \in \tilde{\boldsymbol{\Lambda}}} c(\boldsymbol{\lambda}) = \arg\min_{\boldsymbol{\lambda} \in \tilde{\boldsymbol{\Lambda}}} \widehat{\mathrm{GE}}(\mathcal{I}, \mathcal{J}, \rho, \boldsymbol{\lambda})$$

with $\boldsymbol{\lambda}^*$ as theoretical optimum, and $c(\boldsymbol{\lambda})$ is short for estim. gen. error when $\mathcal{I}$, resampling splits $\mathcal{J}$, performance measure ρ are fixed.

OBJECTIVE AND SEARCH SPACE



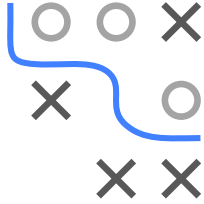
$$\boldsymbol{\lambda}^* \in \underset{\boldsymbol{\lambda} \in \tilde{\boldsymbol{\Lambda}}}{\arg\min}\, c(\boldsymbol{\lambda}) = \underset{\boldsymbol{\lambda} \in \tilde{\boldsymbol{\Lambda}}}{\arg\min}\, \widehat{\mathrm{GE}}(\mathcal{I}, \mathcal{J}, \rho, \boldsymbol{\lambda})$$



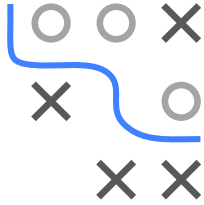
- Evals are stored in **archive**
  $\mathcal{A} = ((\boldsymbol{\lambda}^{(1)}, c(\boldsymbol{\lambda}^{(1)})), (\boldsymbol{\lambda}^{(2)}, c(\boldsymbol{\lambda}^{(2)})), \ldots)$, with
  $\mathcal{A}^{[t+1]} = \mathcal{A}^{[t]} \cup (\boldsymbol{\lambda}^+, c(\boldsymbol{\lambda}^+))$.
- We can define tuner as function $\tau : (\mathcal{D}, \mathcal{I}, \tilde{\boldsymbol{\Lambda}}, \mathcal{J}, \rho) \mapsto \hat{\boldsymbol{\lambda}}$

WHY IS TUNING SO HARD?

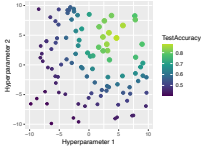
- Tuning is usually **black box**: No derivatives of the objective are availabe. We can only eval the performance for a given HPC via a computer program (CV of learner on data).
- Every evaluation can require multiple train and predict steps, hence it's **expensive**.
- Even worse: the answer we get from that evaluation is **not exact, but stochastic** in most settings, as we use resampling.
- **Categorical and dependent hyperparameters** aggravate our difficulties: the space of hyperparameters we optimize over can have non-metric, complicated structure.
- Many standard optimization algorithms cannot handle these properties.



Introduction to Machine Learning



Hyperparameter Tuning
Basic Techniques

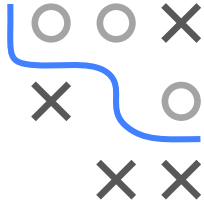


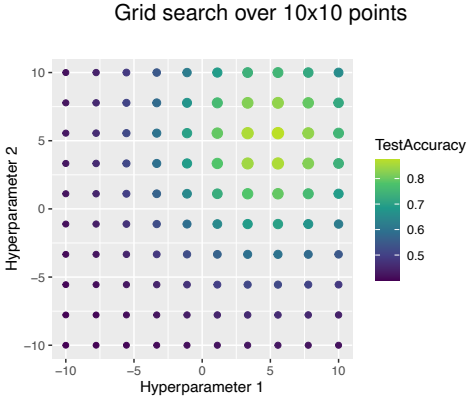
Learning goals

- Understand the idea of grid search
- Understand the idea of random search
- Be able to discuss advantages and disadvantages of the two methods

GRID SEARCH

- Simple technique which is still quite popular, tries all HP combinations on a multi-dimensional discretized grid
- For each hyperparameter a finite set of candidates is predefined
- Then, we simply search all possible combinations in arbitrary order

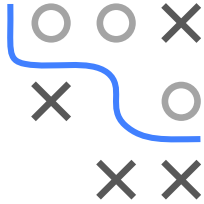




GRID SEARCH

Advantages

- Very easy to implement
- All parameter types possible
- Parallelizing computation is trivial

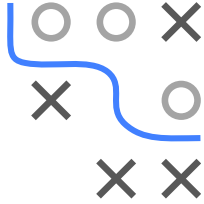


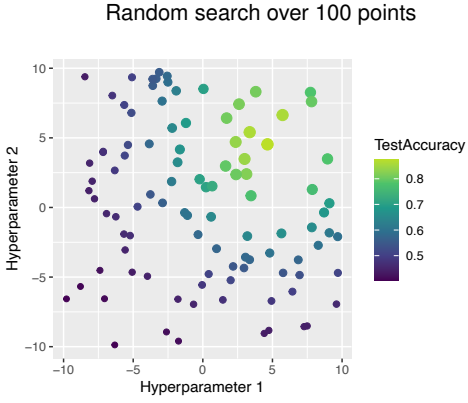
Disadvantages

- Scales badly: combinatorial explosion
- Inefficient: searches large irrelevant areas
- Arbitrary: which values / discretization?

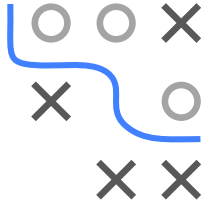
RANDOM SEARCH

- Small variation of grid search
- Uniformly sample from the region-of-interest





RANDOM SEARCH



Advantages

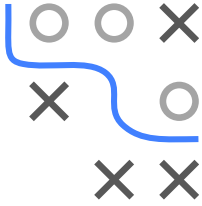
- Like grid search: very easy to implement, all parameter types possible, trivial parallelization
- Anytime algorithm: can stop the search whenever our budget for computation is exhausted, or continue until we reach our performance goal.
- No discretization: each individual parameter is tried with a different value every time

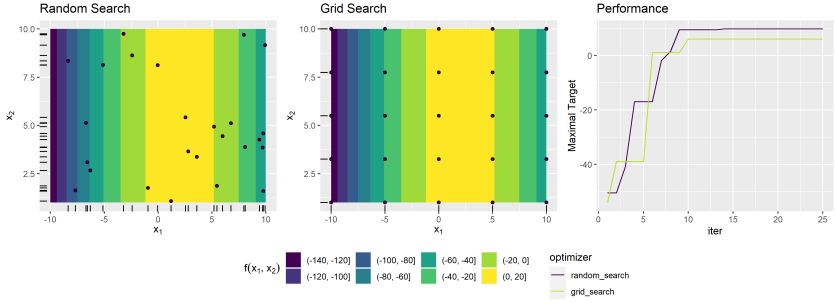
Disadvantages

- Inefficient: many evaluations in areas with low likelihood for improvement
- Scales badly: high-dimensional hyperparameter spaces need *lots* of samples to cover.

RANDOM SEARCH VS. GRID SEARCH

We consider a maximization problem on the function $f(x_1, x_2) = g(x_1) + h(x_2) \approx g(x_1)$, i.e. in order to maximize the target, x_1 should be the parameter to focus on.

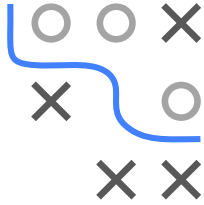




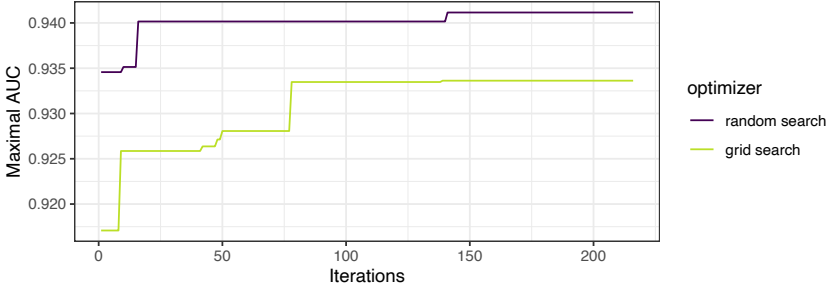
$\Rightarrow$ In this setting, random search is more superior as we get a better coverage for the parameter x_1 in comparison with grid search, where we only discover 5 distinct values for x_1.

TUNING EXAMPLE

Tuning random forest with grid search/random search and 5CV on the `sonar` data set for AUC:

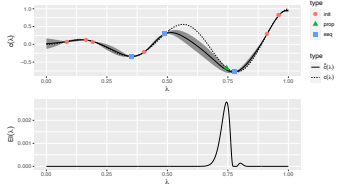


Hyperparameter	Type	Min	Max
`num.trees`	integer	3	500
`mtry`	integer	5	50
`min.node.size`	integer	10	100



Introduction to Machine Learning

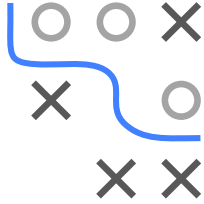
Hyperparameter Tuning
Advanced Tuning Techniques



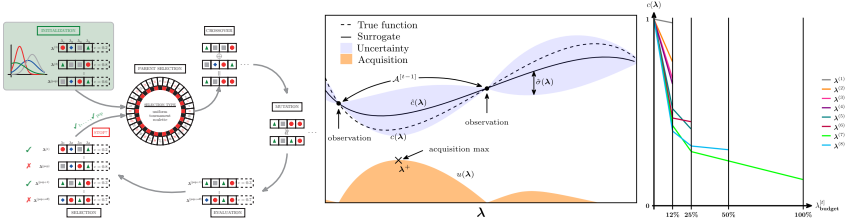
Learning goals

- Basic idea of evolutionary algorithms
- and Bayesian Optimization
- and hyperband

HPO – MANY APPROACHES



- Evolutionary algorithms
- Bayesian / model-based optimization
- Multi-fidelity optimization, e.g. Hyperband

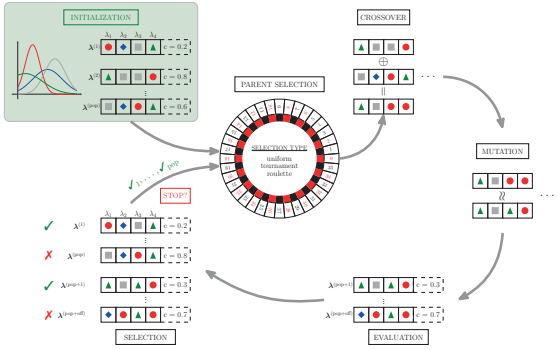


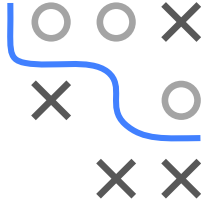
HPO methods can be characterized by:

- how the exploration vs. exploitation trade-off is handled
- how the inference vs. search trade-off is handled

Further aspects: Parallelizability, local vs. global behavior, handling of noisy observations, multifidelity and search space complexity.

EVOLUTIONARY STRATEGIES

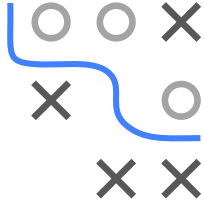




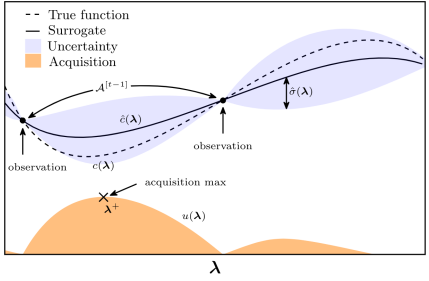
- Are a class of stochastic population-based optimization methods inspired by the concepts of biological evolution
- Are applicable to HPO since they do not require gradients
- Mutation is the (randomized) change of one or a few HP values in a configuration.
- Crossover creates a new HPC by (randomly) mixing the values of two other configurations.

BAYESIAN OPTIMIZATION

BO sequentially iterates:



1. **Approximate** $\boldsymbol{\lambda} \mapsto c(\boldsymbol{\lambda})$ by (nonlin) regression model $\hat{c}(\boldsymbol{\lambda})$, from evaluated configurations (archive)
2. **Propose candidates** via optimizing an acquisition function that is based on the surrogate $\hat{c}(\boldsymbol{\lambda})$
3. **Evaluate** candidate(s) proposed in 2, then go to 1

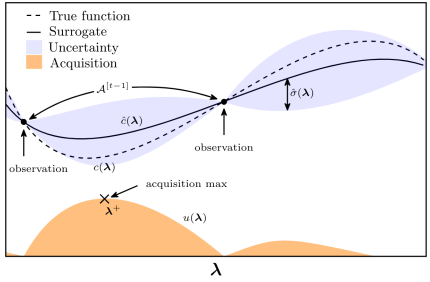


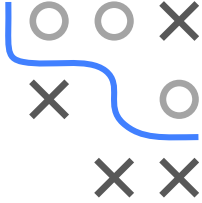
Important trade-off: **Exploration** (evaluate candidates in under-explored areas) vs. **exploitation** (search near promising areas)

BAYESIAN OPTIMIZATION

Surrogate Model:

- Probabilistic modeling of $C(\boldsymbol{\lambda}) \sim (\hat{c}(\boldsymbol{\lambda}), \hat{\sigma}(\boldsymbol{\lambda}))$ with posterior mean $\hat{c}(\boldsymbol{\lambda})$ and uncertainty $\hat{\sigma}(\boldsymbol{\lambda})$.
- Typical choices for numeric spaces are Gaussian Processes; random forests for mixed spaces

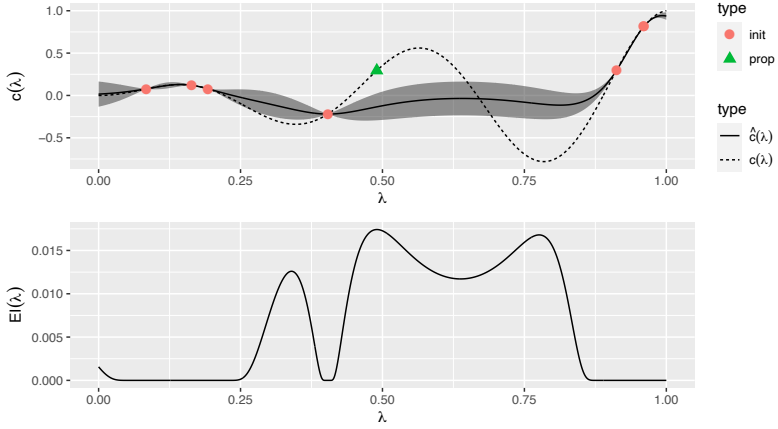


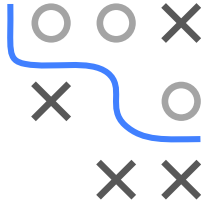


Acquisition Function:

- Balance exploration (high $\hat{\sigma}$) vs. exploitation (low $\hat{c}$).
- Lower confidence bound (LCB): $a(\boldsymbol{\lambda}) = \hat{c}(\boldsymbol{\lambda}) - \kappa \cdot \hat{\sigma}(\boldsymbol{\lambda})$
- Expected improvement (EI): $a(\boldsymbol{\lambda}) = \mathbb{E}\left[\max\left\{c_{\min} - C(\boldsymbol{\lambda}), 0\right\}\right]$ where ($c_{\min}$ is best cost value from archive)
- Optimizing $a(\boldsymbol{\lambda})$ is still difficult, but cheap(er)

BAYESIAN OPTIMIZATION

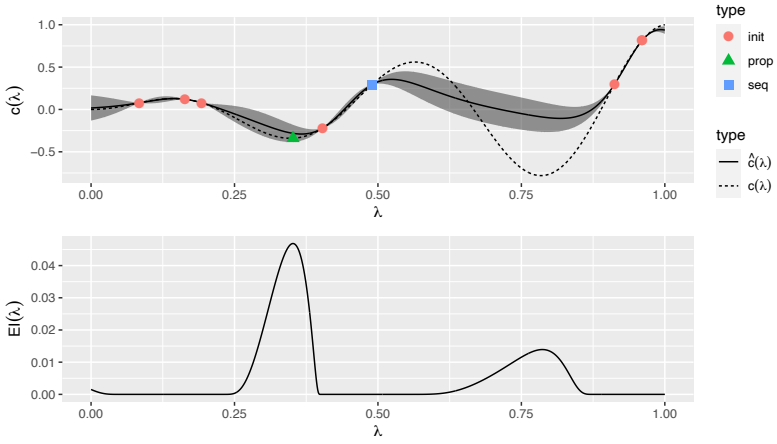


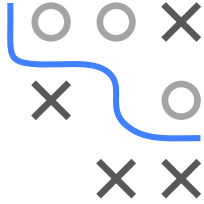


Upper plot: The surrogate model (black, solid) models the *unknown* relationship between input and output (black, dashed) based on the initial design (red points).

Lower plot: Mean and variance of the surrogate model are used to derive the expected improvement (EI) criterion. The point that maximizes the EI is proposed (green point).

BAYESIAN OPTIMIZATION

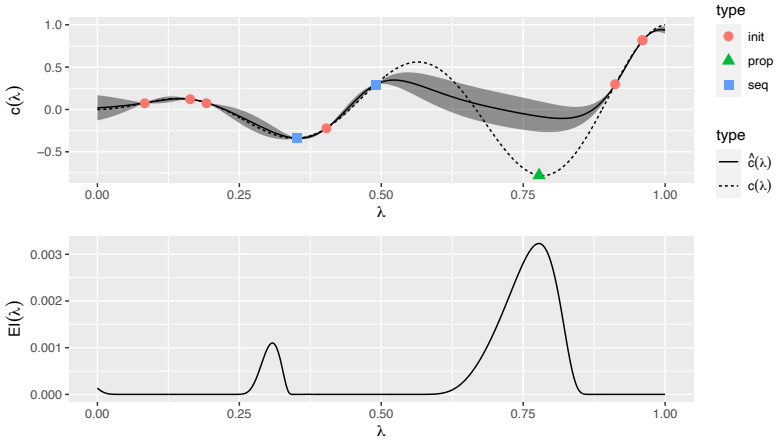


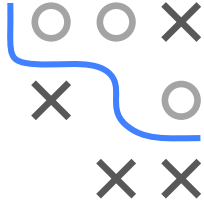


Upper plot: The surrogate model (black, solid) models the *unknown* relationship between input and output (black, dashed) based on the initial design (red points).

Lower plot: Mean and variance of the surrogate model are used to derive the expected improvement (EI) criterion. The point that maximizes the EI is proposed (green point).

BAYESIAN OPTIMIZATION

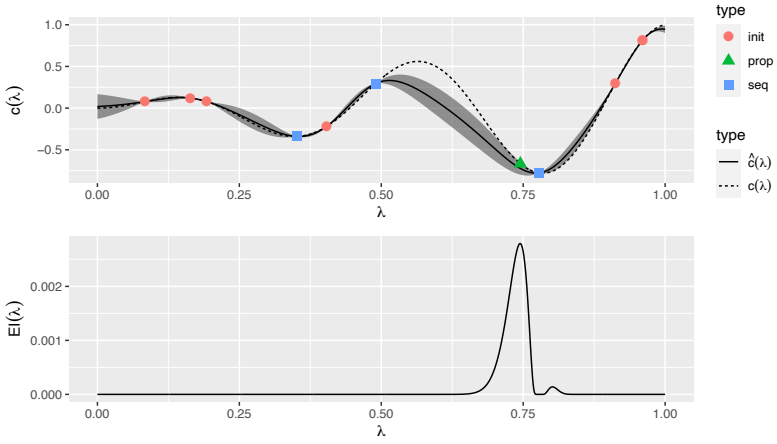


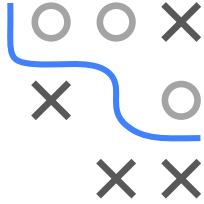


Upper plot: The surrogate model (black, solid) models the *unknown* relationship between input and output (black, dashed) based on the initial design (red points).

Lower plot: Mean and variance of the surrogate model are used to derive the expected improvement (EI) criterion. The point that maximizes the EI is proposed (green point).

BAYESIAN OPTIMIZATION



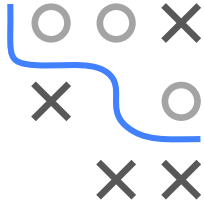


Upper plot: The surrogate model (black, solid) models the *unknown* relationship between input and output (black, dashed) based on the initial design (red points).

Lower plot: Mean and variance of the surrogate model are used to derive the expected improvement (EI) criterion. The point that maximizes the EI is proposed (green point).

BAYESIAN OPTIMIZATION

Since we use the sequentially updated surrogate model predictions of performance to propose new configurations, we are guided to "interesting" regions of $\mathbf{\Lambda}$ and avoid irrelevant evaluations:



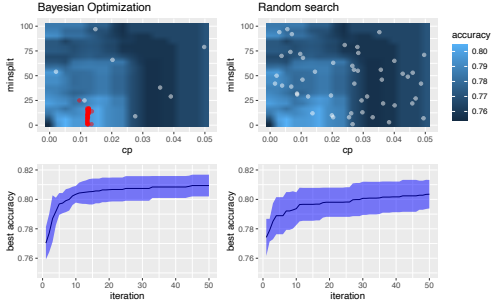
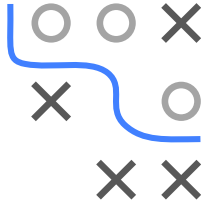


Figure: Tuning complexity and minimal node size for splits for CART on the `titanic` data (10-fold CV maximizing accuracy).
Left panel: BO, 50 configurations; right panel: random search, 50 iterations.
Top panel: one run (initial design of BO is white); bottom panel: mean $\pm$ std of 10 runs.

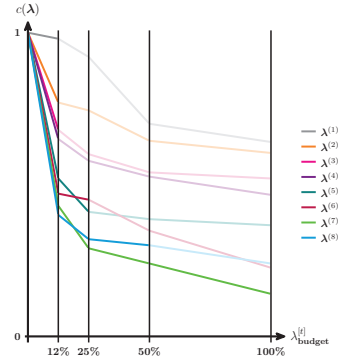
MULTIFIDELITY OPTIMIZATION

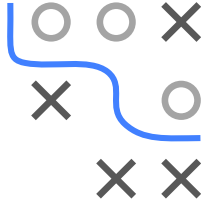


- Prerequiste: Fidelity HP λ_{fid}, i.e., a component of $\boldsymbol{\lambda}$, which influences the computational cost of the fitting procedure in a monotonically increasing manner
- Methods of multifidelity optimization in HPO are all tuning approaches that can efficiently handle a $\mathcal{I}$ with a HP λ_{fid}
- The lower we set λ_{fid}, the more points we can explore in our search space, albeit with much less reliable information w.r.t. their true performance.
- We assume to know box-constraints of λ_{fid}, so $\lambda_{\text{fid}} \in [\lambda_{\text{fid}}^{\text{low}}, \lambda_{\text{fid}}^{\text{upp}}]$, where the upper limit implies the highest fidelity returning values closest to the true objective value at the highest computational cost.

SUCCESSIVE HALVING

- Races down set of HPCs to the best
- Idea: Discard bad configurations early
- Train HPCs with fraction of full budget (SGD epochs, training set size); the control param for this is called **multi-fidelity HP**
- Continue with better $1/\eta$ fraction of HPCs (w.r.t $\widehat{\mathrm{GE}}$); with η times budget (usually $\eta = 2, 3$)
- Repeat until budget depleted or single HPC remains





MULTIFIDELITY OPTIMIZATION – HYPERBAND

Problem with SH

- Good HPCs could be killed off too early, depends on evaluation schedule

Solution: Hyperband

- Repeat SH with different start budgets $\lambda_{\text{fid}}^{[0]}$ and initial number of HPCs $p^{[0]}$
- Each SH run is called bracket
- Each bracket consumes ca. the same budget

For $\eta = 4$

bracket 3

t	$\lambda_{\text{fid}}^{[t]}$	$p_3^{[t]}$
0	1	82
1	4	20
2	16	5
3	64	1

bracket 2

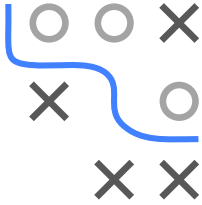
t	$\lambda_{\text{fid}}^{[t]}$	$p_2^{[t]}$
0	4	27
1	16	6
2	64	1

bracket 1

t	$\lambda_{\text{fid}}^{[t]}$	$p_1^{[t]}$
0	16	10
1	64	2

bracket 0

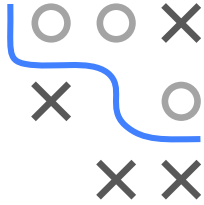
t	$\lambda_{\text{fid}}^{[t]}$	$p_0^{[t]}$
0	64	5



MORE TUNING ALGORITHMS:

Other advanced techniques besides model-based optimization and the hyperband algorithm are:

- Stochastic local search, e.g., simulated annealing
- Genetic algorithms / CMAES
- Iterated F-Racing
- Many more . . .



For more information see *Hyperparameter Optimization: Foundations, Algorithms, Best Practices and Open Challenges*, Bischl (2021)

Introduction to Machine Learning

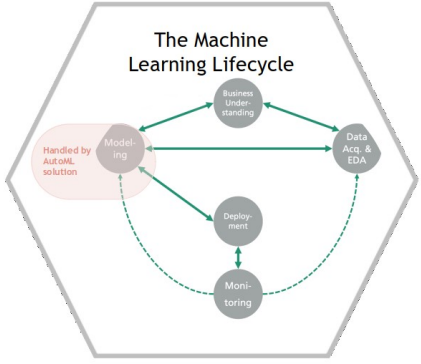
Hyperparameter Tuning Pipelines and AutoML

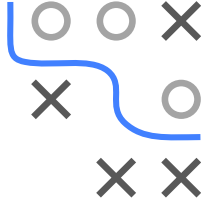
Learning goals

- Pipelines as connected steps of learnable operations
- Sequential pipeline
- Pipelines and DAGs

CASE FOR AUTOML

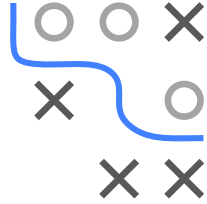
- More and more tasks are approached via data driven methods.
- Data scientists often rely on trial-and-error.
- The process is especially tedious for similar, recurring tasks.
- Not the entire machine learning lifecycle can be automated.

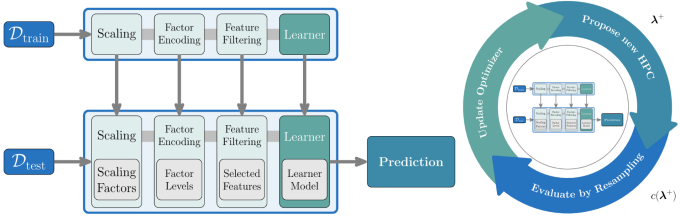




PIPELINES AND AUTOML

- ML typically has several data transformation steps before model fit
- If steps are in succession, data flows through sequential pipeline
- NB: Each node has a train and predict step and learns params
- And usually has HPs

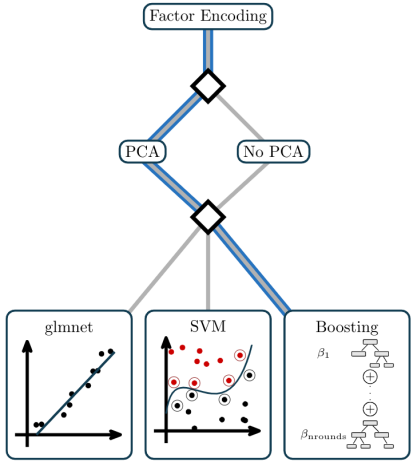


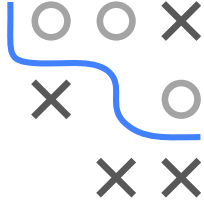


Pipelines are required to embed full model building into CV to avoid overfitting and biased evaluation!

PIPELINES AND AUTOML

- Further flexibility by representing pipeline as DAG
- Single source accepts $\mathcal{D}_{\text{train}}$, single sink returns predictions
- Each node represents a preprocessing operation, a learner, a postprocessing operation or controls data flow
- Can be used to implement ensembles, operator selection, ...



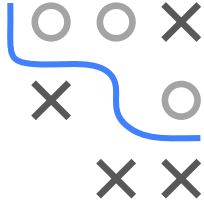


PIPELINES AND AUTOML

- HPs of pipeline are the joint set of all HPs of its contained nodes:
  $\tilde{\boldsymbol{\Lambda}} = \tilde{\boldsymbol{\Lambda}}_{\text{op},1} \times \cdots \times \tilde{\boldsymbol{\Lambda}}_{\text{op},k} \times \tilde{\boldsymbol{\Lambda}}_{\mathcal{I}}$
- HP space of a DAG is more complex: Depending on branching / selection different nodes and HPs are active
  $\rightarrow$ **hierarchical search space**

Search Space $\tilde{\Lambda}$

Name	Type	Bounds/Values	Trafo
`encoding`	C	one-hot, impact	
◇ `pca`	C	PCA, no PCA	
◇ `learner`	C	glmnet, SVM, Boosting	
`if learner = glmnet`			
`s`	R	$[-12, 12]$	2^x
`alpha`	R	$[0, 1]$	–
`if learner = SVM`			
`cost`	R	$[-12, 12]$	2^x
`gamma`	R	$[-12, 12]$	2^x
`if learner = Boosting`			
`eta`	R	$[-4, 0]$	10^x
`nrounds`	I	$\{1, \ldots, 5000\}$	–
`max_depth`	I	$\{1, \ldots, 20\}$	–

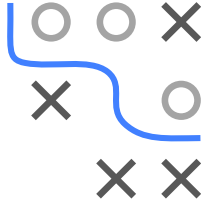


A graph that includes many preprocessing steps and learner types can be flexible enough to work on a large number of data sets

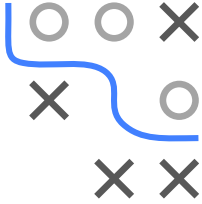
Combining such graph with an efficient tuner is key in AutoML

AUTOML – CHALLENGES

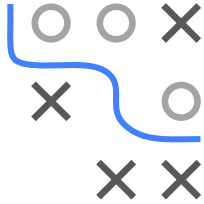
- Most efficient approach?
- How to integrate human a-priori knowledge?
- How can we best (computationally) transfer “experience” into AutoML? Warmstarts, learned search spaces, etc.
- Multi-Objective goals, including model intepretability
- AutoML as a process is too much of a black-box, hurts adoption.



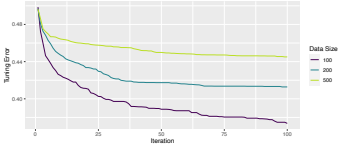
INTRODUCTION TO MACHINE LEARNING



Introduction to Machine Learning



Nested Resampling Motivation

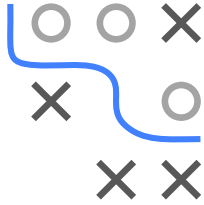


Learning goals

- Understand the problem of overtuning
- Be able to explain the untouched test set principle and how it motivates the idea of nested resampling

MOTIVATION

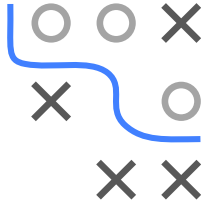
Selecting the best model from a set of potential candidates (e.g., different classes of learners, different hyperparameter settings, different feature sets, different preprocessing, ...) is an important part of most machine learning problems.



Problem

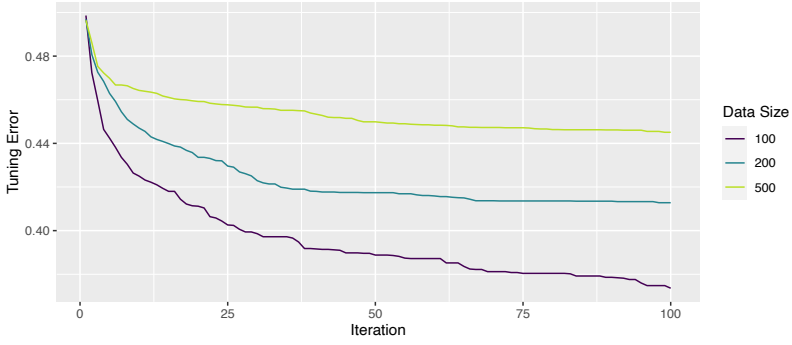
- We cannot evaluate our finally selected learner on the same resampling splits that we have used to perform model selection for it, e.g., to tune its hyperparameters.
- By repeatedly evaluating the learner on the same test set, or the same CV splits, information about the test set “leaks” into our evaluation.
- Danger of overfitting to the resampling splits / overtuning!
- The final performance estimate will be optimistically biased.
- One could also see this as a problem similar to multiple testing.

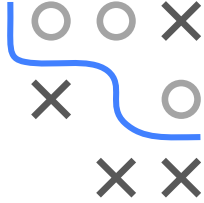
INSTRUCTIVE AND PROBLEMATIC EXAMPLE



- Assume a binary classification problem with equal class sizes.
- Assume a learner with hyperparameter λ.
- Here, the learner is a (nonsense) feature-independent classifier, where λ has no effect. The learner simply predicts random labels with equal probability.
- Of course, its true generalization error is 50%.
- A cross-validation of the learner (with any fixed λ) will easily show this (given that the partitioned data set for CV is not too small).
- Now let's "tune" it, by trying out 100 different λ values.
- We repeat this experiment 50 times and average results.

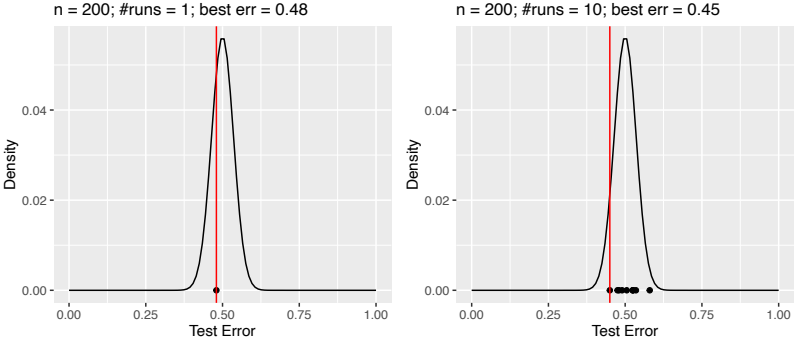
INSTRUCTIVE AND PROBLEMATIC EXAMPLE

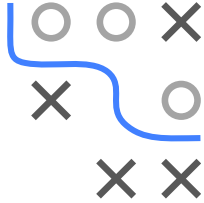




- Plotted is the best “tuning error” (i.e. the performance of the model with fixed λ as evaluated by the cross-validation) after k tuning iterations.
- We have performed the experiment for different sizes of learning data that were cross-validated.

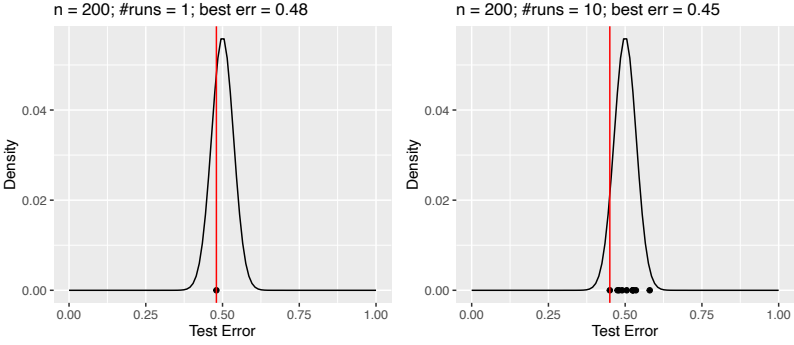
INSTRUCTIVE AND PROBLEMATIC EXAMPLE

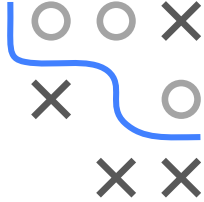




- For 1 experiment, the CV score will be nearly 0.5, as expected
- We basically sample from a (rescaled) binomial distribution when we calculate error rates
- And multiple experiment scores are also nicely arranged around the expected mean 0.5

INSTRUCTIVE AND PROBLEMATIC EXAMPLE

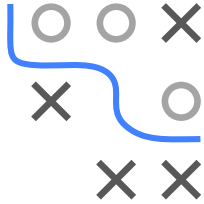




- But in tuning we take the minimum of those! So we don't really estimate the "average performance" anymore, we get an estimate of "best case" performance instead.
- The more we sample, the more "biased" this value becomes.

UNTOUCHED TEST SET PRINCIPLE

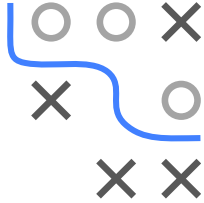
Countermeasure: simulate what actually happens in model application.



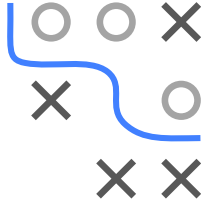
- All parts of the model building (including model selection, preprocessing) should be embedded in the model-finding process **on the training data**.
- The test set should only be touched once, so we have no way of "cheating". The test data set is only used once *after* a model is completely trained, after deciding, for example, on specific hyperparameters.
  Only if we do this are the performance estimates we obtained from the test set **unbiased estimates** of the true performance.

UNTOUCHED TEST SET PRINCIPLE

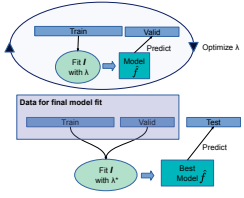
- For steps that themselves require resampling (e.g., hyperparameter tuning) this results in **nested resampling**, i.e., resampling strategies for both
  - tuning: an inner resampling loop to find what works best based on training data
  - outer evaluation on data not used for tuning to get honest estimates of the expected performance on new data



Introduction to Machine Learning



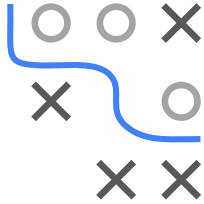
Nested Resampling
Training - Validation - Test



Learning goals

- Understand how to fulfill the untouched test set principle by a 3-way split of the data
- Understand how thereby the tuning step can be seen as part of a more complex training procedure

TUNING PROBLEM



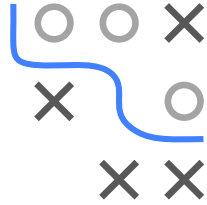
Remember:

We need to

- **select an optimal learner**
- without compromising the **accuracy of the performance estimate** for that learner

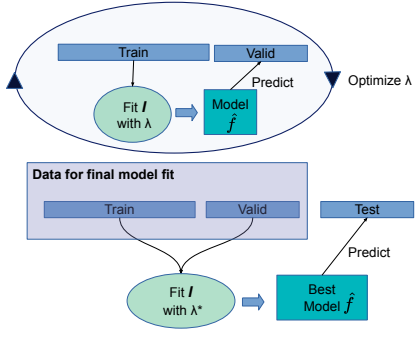
– for that we need an **untouched test set**!

TRAIN - VALIDATION - TEST



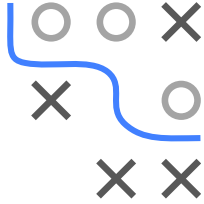
Simplest method to achieve this: a 3-way split

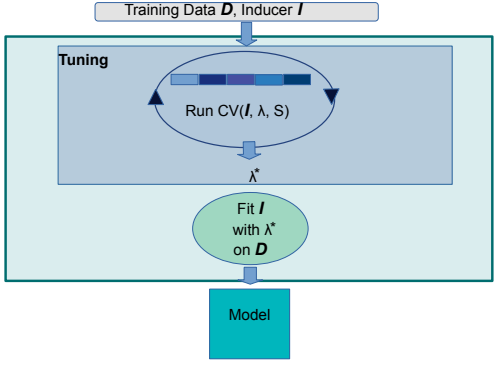
- During tuning, a learner is trained on the **training set**, evaluated on the **validation set**
- After the best model configuration $\boldsymbol{\lambda}^*$ has been selected, we re-train on the joint (training+validation) set and evaluate the model's performance on the **test set**.



TUNING AS PART OF MODEL BUILDING

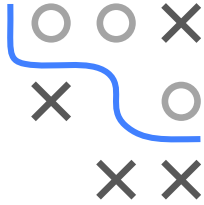
- Effectively, the tuning step is now simply part of a more complex training procedure.
- We could see this as removing the hyperparameters from the inputs of the algorithm and making it “self-tuning”.

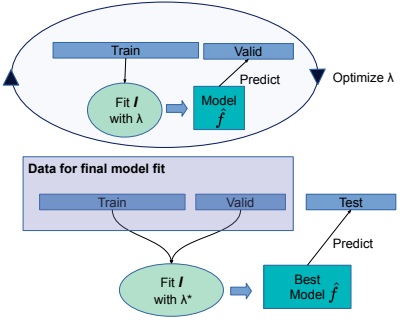


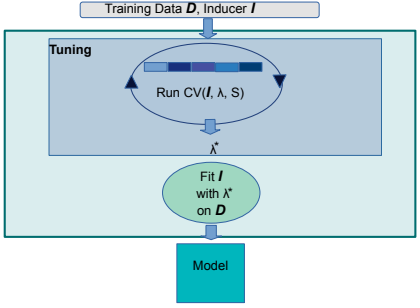


TUNING AS PART OF MODEL BUILDING

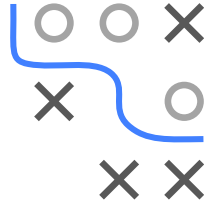
More precisely: the combined training & validation set is actually the training set for the “self-tuning” endowed algorithm.



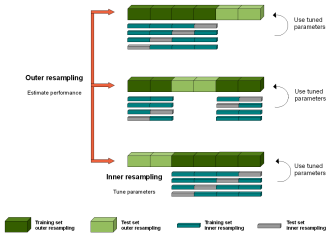




Introduction to Machine Learning



Nested Resampling
Nested Resampling Procedure

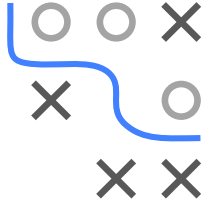


Learning goals

- Understand how the 3-way split of the data can be generalized to nested resampling
- Understand the goal of nested resampling
- Be able to explain how resampling allows to estimate the generalization error

NESTED RESAMPLING

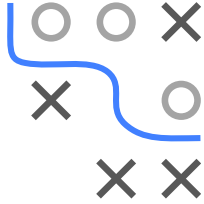
Just like we can generalize hold-out splitting to resampling to get more reliable estimates of the predictive performance, we can generalize the training/validation/test approach to **nested resampling**.

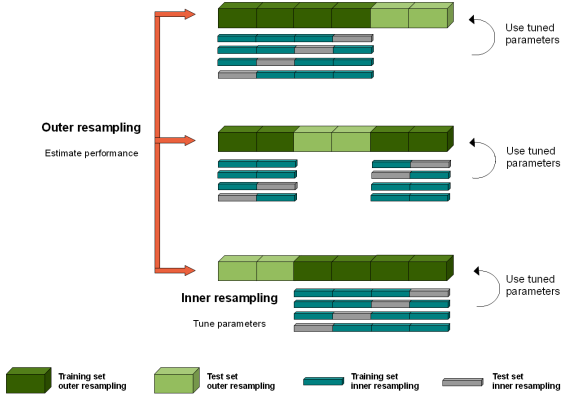


This results in two nested resampling loops, i.e., resampling strategies for both tuning and outer evaluation.

NESTED RESAMPLING

Assume we want to tune over a set of candidate HP configurations λ_i; $i = 1, \ldots$ with 4-fold CV in the inner resampling and 3-fold CV in the outer loop. The outer loop is visualized as the light green and dark green parts.

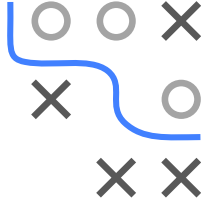


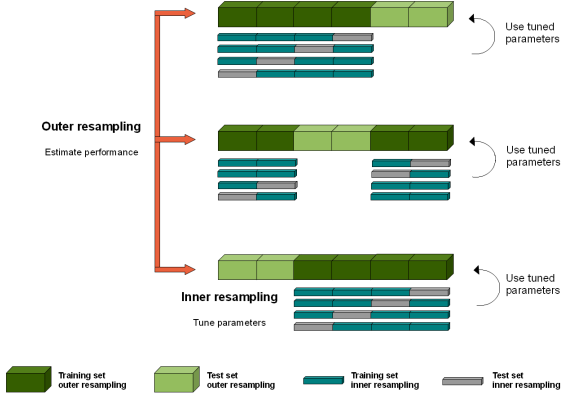


NESTED RESAMPLING

In each iteration of the outer loop we:

- Split off the light green testing data
- Run the tuner on the dark green part of the data, e.g., evaluate each λ_i through fourfold CV on the dark green part

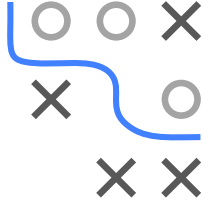


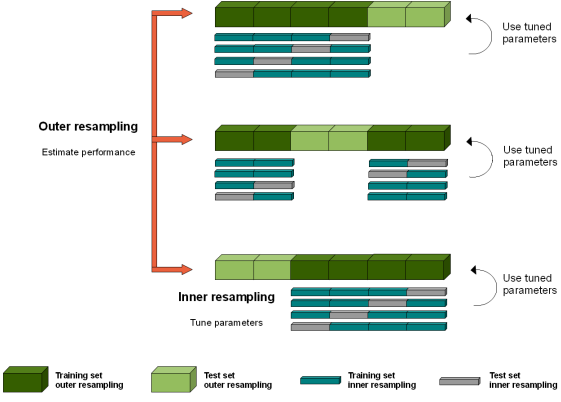


NESTED RESAMPLING

In each iteration of the outer loop we:

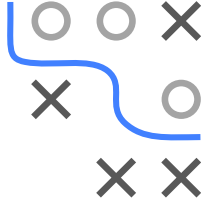
- Return the winning λ^* that performed best on the grey inner test sets
- Re-train the model on the full outer dark green train set
- Evaluate it on the outer light green test set

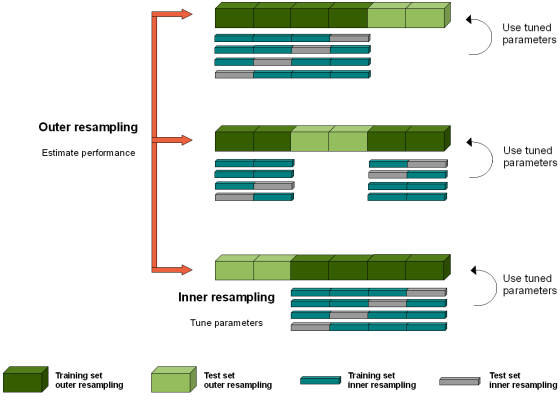




NESTED RESAMPLING

The error estimates on the outer samples (light green) are unbiased because this data was strictly excluded from the model-building process of the model that was tested on.





NESTED RESAMPLING - INSTRUCTIVE EXAMPLE

Taking again a look at the motivating example and adding a nested resampling outer loop, we get the expected behavior: