Fondamenti di IA

04 - Ensembles & Neural Networks

SAPIENZA
UNIVERSITÀ DI ROMA

Fabrizio Silvestri

Intro

Learner's Input

- In the basic statistical learning setting, the learner has access to the following:
  - Domain set $\mathscr{X}$: Set of objects that we may wish to label.
  - Label set $\mathscr{Y}$. Set of possible labels.
    - We will restrict the label set to be a two-element set, usually {0,1} or {−1,+1}
  - Training data $S=((x_1,y_1)...(x_m,y_m))$ is a finite sequence of pairs in $\mathscr{X}\times\mathscr{Y}$
    - We call S the training examples or training set.

Learner's Output

- The learner is requested to output a prediction rule, h: $\mathscr{X} \to \mathscr{Y}$.
- This function is also called a *predictor*, a *hypothesis*, or a *classifier*.
- The predictor can be used to predict the label of new domain points.
- We use the notation A(S) to denote the hypothesis that a learning algorithm, A, returns upon receiving the training sequence S.

What does the learner know?

- The learner is blind to the underlying distribution $\mathscr{D}$ over the world and to the labeling function f.
- In our restaurant example, we have just arrived in a new city and we have no clue as to how waiting times are distributed and how to predict them.
- The only way the learner can interact with the environment is through observing the training set.

So far our hypotheses were represented by a single function:

what if we learn by means of a combination (ensemble) of hypothesis?

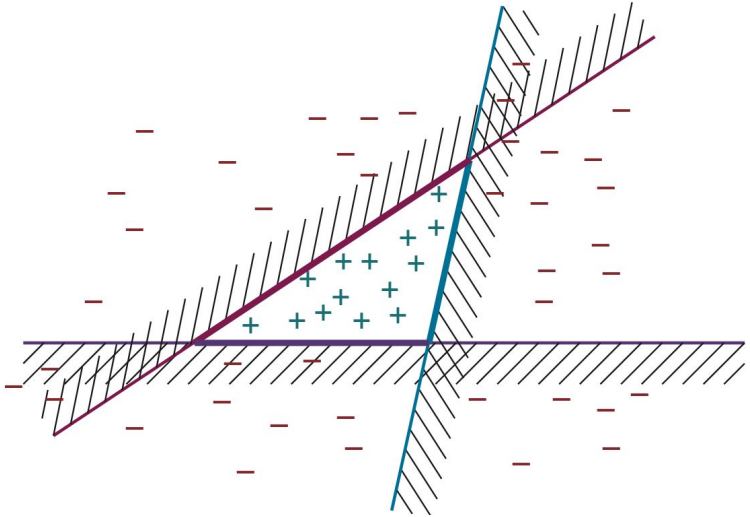
Ensemble Learning

The idea

- Base Models
  - individual hypotheses: $h_1, h_2, \ldots, h_n$
- Combine them to form $h = f(h_1, h_2, \ldots, h_n)$
  - f could be average, another ML function, sum, etc.
- Reasons to use Ensembles:
  - Reduce Bias
  - Reduce Variance

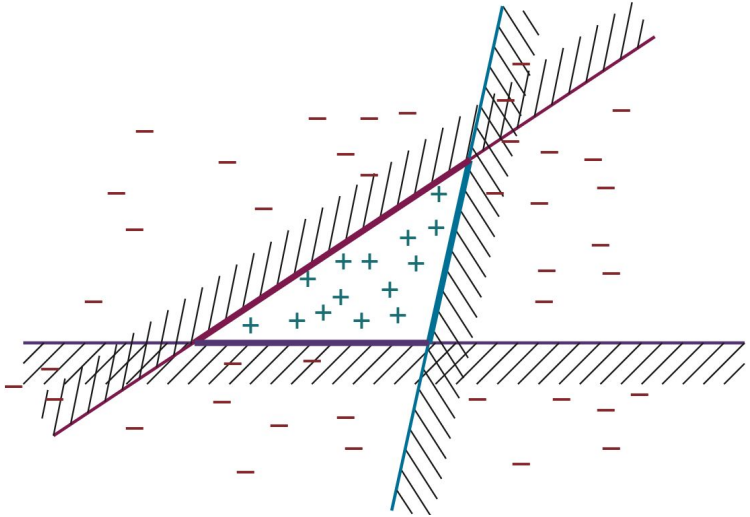
Why do they Reduce Bias

- The hypothesis space of a base model may be too restrictive, imposing a strong bias
  - such as the bias of a linear decision boundary in logistic regression
- An ensemble can be more expressive, and thus have less bias, than the base models.



Why do they Reduce Bias

- An ensemble of n linear classifiers allows more functions to be realizable, at a cost of only n times more computation
- This is often better than allowing a completely general hypothesis space that might require *exponentially more* computation.



Why do they Reduce Variance

- Consider an ensemble of K = 5 binary classifiers that we combine using majority voting.
  - For the ensemble to misclassify a new example, at least three of the five classifiers have to misclassify it.
  - The hope is that this is less likely than a single misclassification by a single classifier.
  - To quantify that, suppose you have trained a single classifier that is correct in 80% of cases.
  - Now create an ensemble of 5 classifiers, each trained on a different subset of the data so that they are independent.
  - Let’s assume this leads to some reduction in quality, and each individual classifier is correct in only 75% of cases.
  - But together, the majority vote of the ensemble will be correct in 89% of cases (and 99% with 17 classifiers), assuming true independence*.

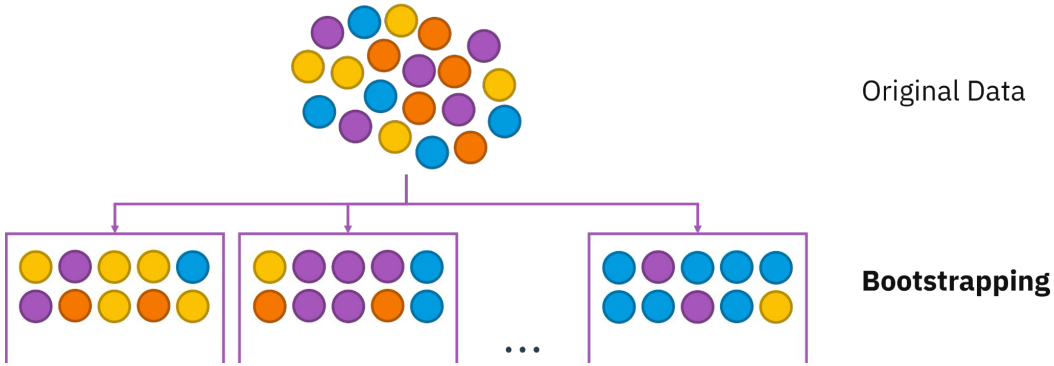
Independence Assumption (*)

- In practice the independence assumption is unreasonable—individual classifiers share some of the same data and assumptions, and thus are not completely independent, and will share some of the same errors.
- But if the component classifiers are at least somewhat uncorrelated then ensemble learning will make fewer misclassifications.
- We will now consider three ways of creating ensembles: bagging, stacking, and boosting.

Bagging and Random Forests

Data Generation

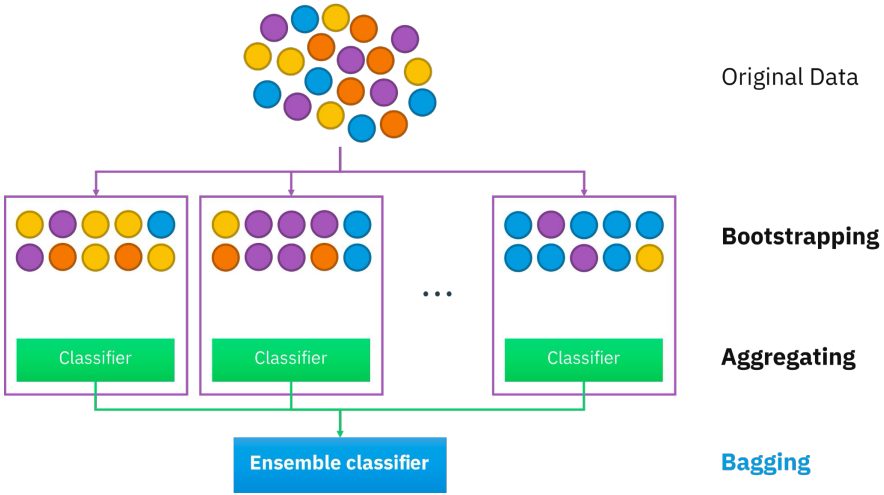
- Build K distinct training data by sampling with replacement from the original training set



  - Replacement means that we may pick the same object twice (or more). See the blue balls in the last bin.

The Classifier

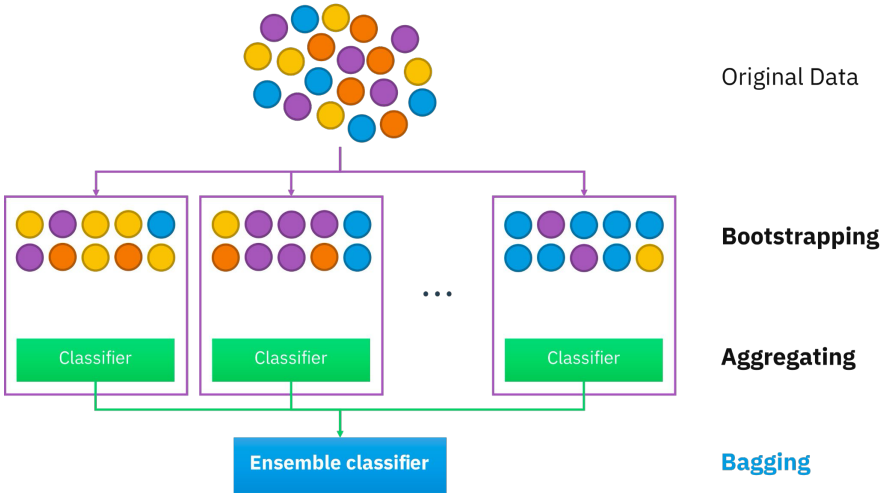
- Each training subset is used to build a separate classifier



- Each classifier output is then aggregated to produce the final classification outcome

Aggregation Strategies

- For classification, we might use majority voting
- For regression, the final output is the average: $h(\mathbf{x}) = \frac{1}{K}\sum_{i=1}^{K} h_i(\mathbf{x})$



Bagging and Decision Trees

- Bagging tends to reduce variance and is a standard approach when there is limited data or when the base model is seen to be overfitting.
- Bagging can be applied to any class of model, but is most commonly used with decision trees.
  - It is appropriate because decision trees are unstable: a slightly different set of examples can lead to a wildly different tree.
  - Bagging smoothes out this variance. If you have access to multiple computers then bagging is efficient, because the hypotheses can be computed in parallel.
- Unfortunately, bagging decision trees often ends up giving us K trees that are highly correlated
  - If there is one attribute with a very high information gain, it is likely to be the root of most of the trees.

Random Forests

- The random forest model is a form of decision tree bagging in which we take extra steps to make the ensemble of K trees more diverse, to reduce variance.
  - Random forests can be used for classification or regression.
- The key idea is to randomly vary the attribute choices.
  - At each split point in constructing the tree, we select a random sampling of attributes, and then compute which of those gives the highest information gain.
- If there are n attributes, a common default choice is that each split randomly picks
  - $\sqrt{n}$ attributes to consider for classification problems,
  - $n/3$ for regression problems.

Recapping: RF Pseudocode

- At a high-level, in pseudo-code, Random Forests algorithm follows these steps:
  - Take the original dataset and create N bagged samples of size n, with n smaller than the original dataset.
  - Train a Decision Tree with each of the N bagged datasets as input.
    - When doing a node split, don't explore all features in the dataset.
    - Randomly select a smaller number, M features, from all the features in training set.
    - Then pick the best split.
  - Aggregate the results of the individual decision trees into a single output.
    - Average the values for each observation, produced by each tree, if you're working on a Regression task.
    - Do a majority vote across all trees, for each observation, if you're working on a Classification task.

Advantages of RF

- **Random Forests require almost no input preparation**.
  - They can handle binary features, categorical features, numerical features without any need for scaling.
- **Random Forests perform implicit feature selection**
  - and provide a pretty good indicator of feature importance.
- **Random Forests are very quick to train**.
  - It's a stroke of brilliance when a performance optimization happens to enhance model precision, or vice versa. The random feature sub-setting that aims at diversifying individual trees, is at the same time a great performance optimization! Tuning down the fraction of features that is considered at any given node can let you easily work on datasets with thousands of features. (The same is applicable for row sampling if your dataset has lots of rows)
- **Random Forests are pretty tough to beat**.
  - Although you can typically find a model that beats RFs for any given dataset (typically a neural net or some boosting algorithm), it's never by much, and it usually takes much longer to build and tune said model than it took to build the Random Forest. This is why they make for excellent benchmark models.
- **It's really hard to build a bad Random Forest**!
  - Since random forests are not very sensitive to the specific hyper-parameters used, they don't require a lot of tweaking and fiddling to get a decent model, just use a large number of trees and things won't go terribly awry. Most Random Forest implementations have sensible defaults for the rest of the parameters.
- **Versatility**.
  - Random Forest are applicable to a wide variety of modeling tasks, they work well for regression tasks, work very well for classification taks(and even produce decently calibrated probability scores), and even though I've never tried it myself, they can be used for cluster analysis.
- **Simplicity**.
  - If not of the resulting model, then of the learning algorithm itself. The basic RF learning algorithm can be written in a few lines of code. There's a certain irony about that. But a sense of elegance as well.
- **Lots of excellent, free, and open-source implementations**.
  - You can find a good implementation in almost all major ML libraries and toolkits. R, scikit-learn and Weka jump to mind for having exceptionally good implementations.
- As if all of that is not enough, **Random Forests can be easily grown in parallel**.
  - The same cannot be said about boosted models or large neural networks.

Extremely Randomized Trees (ExtraTrees)

- A further improvement is to use randomness in selecting the split point value
  - for each selected attribute, we randomly sample several candidate values from a uniform distribution over the attribute's range.
  - Then we select the value that has the highest information gain.
- That makes it more likely that every tree in the forest will be different.
- Trees constructed in this fashion are called Extremely Randomized Trees (ExtraTrees).

Efficiency of Random Forests

- Random forests are efficient to create.
  - You might think that it would take K times longer to create an ensemble of K trees, but it is not that bad, for three reasons:
    - each split point runs faster because we are considering fewer attributes,
    - we can skip the pruning step for each individual tree, because the ensemble as a whole decreases overfitting, and
    - if we happen to have K computers available, we can build all the trees in parallel.
      - For example, Adele Cutler reports that for a 100-attribute problem, if we have just three CPUs we can grow a forest of K = 100 trees in about the same time as it takes to create a single decision tree on a single CPU.

Hyperparameter Tuning

- All the hyperparameters of random forests can be trained by cross-validation:
  - the number of trees K,
  - the number of examples used by each tree N (often expressed as a percentage of the complete data set),
  - the number of attributes used at each split point (often expressed as a function of the total number of attributes, such as n),
  - and the number of random split points tried if we are using ExtraTrees.
- In place of the regular cross-validation strategy, we could measure the out-of-bag error: the mean error on each example, using only the trees whose example set didn't include that particular example.

Overfitting?

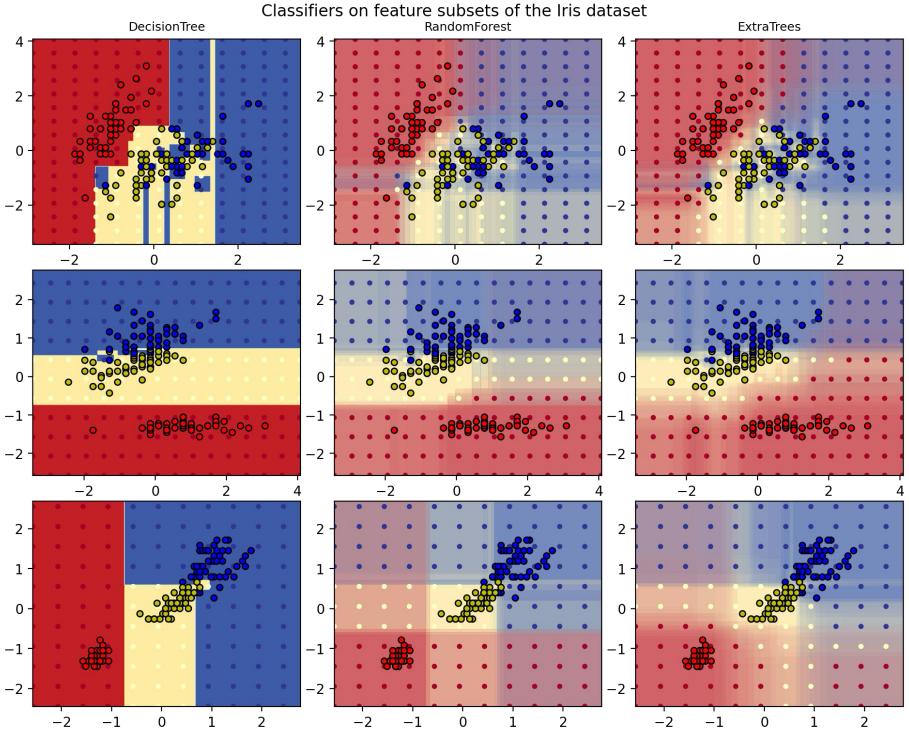
- We have been warned that more complex models can be prone to overfitting, and observed that to be true for decision trees, where we found that pruning was an answer to prevent overfitting.
- Random forests are complex, unpruned models.
  - Yet they are resistant to overfitting.
  - As you increase capacity by adding more trees to the forest they tend to improve on validation-set error rate.

Overfitting?

- Breiman in 2001 gave a mathematical proof that (in almost all cases) as you add more trees to the forest, the error converges; it does not grow.
  - One way to think of it is that the random selection of attributes yields a variety of trees, thus reducing variance, but because we don't need to prune the trees, they can cover the full input space at higher resolution.
  - Some number of trees can cover unique cases that appear only a few times in the data, and their votes can prove decisive, but they can be outvoted when they do not apply.
- Random forests are not totally immune to overfitting.
- Although the error can't increase in the limit, that does not mean that the error will go to zero.

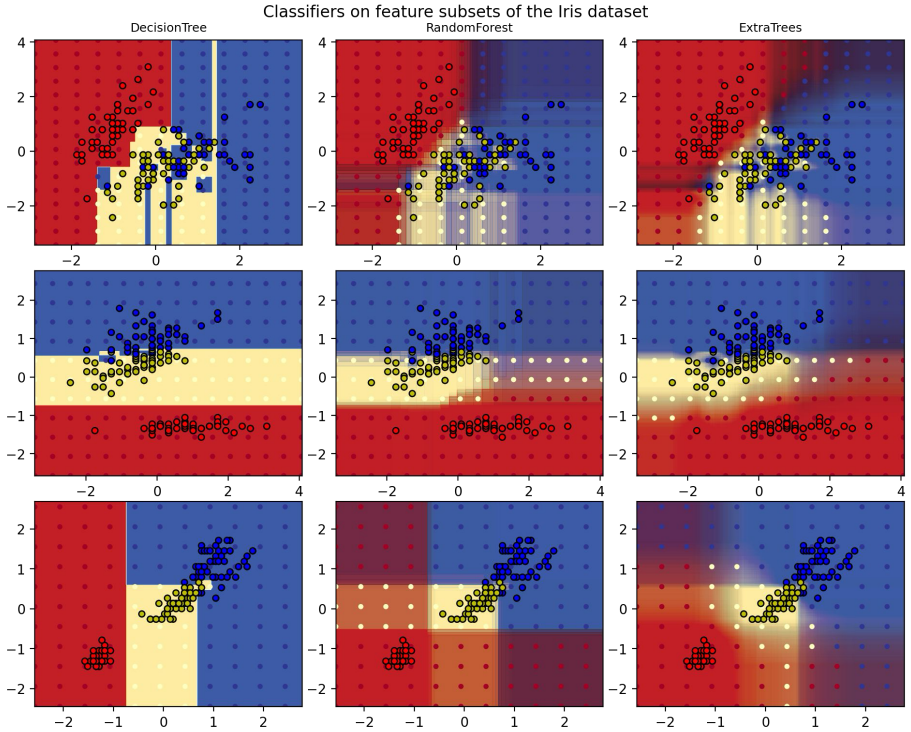
Plotting the Decision Boundaries (30 trees)

- Using this colab.



Plotting the Decision Boundaries (300 trees)

- Using this colab.

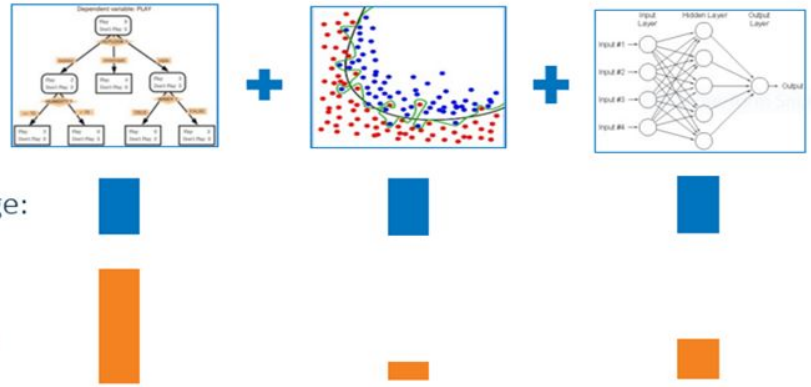


Stacking

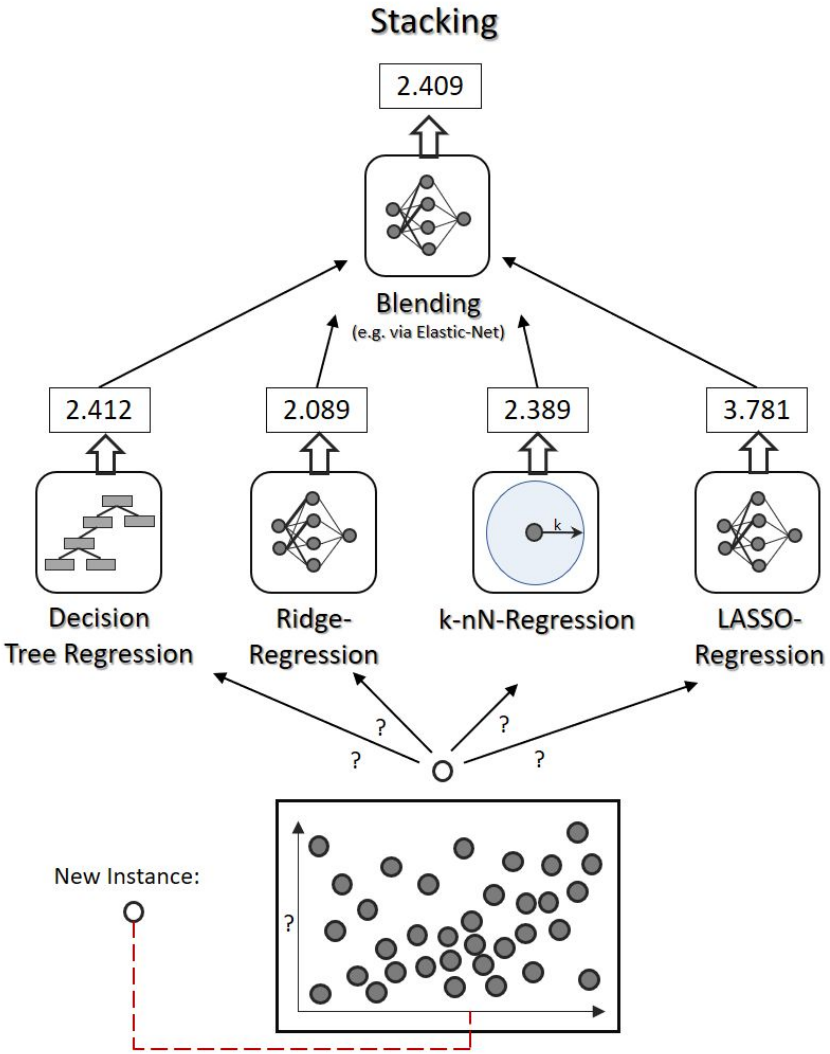
Instead of sampling data, can we sample hypothesis?

- Bagging combines multiple base models of the same model class trained on different data
- Stacked generalization (or stacking for short) combines multiple base models from different model classes trained on the same data.

Simple Stacking



Learning to Stack



Example on the Restaurant Dataset

- Split the data into train/valid/test
- Use the train dataset to train N models, say 3: SVM, Logistic Regression, DT
- Take the valid set and augment it with N additional rows (in this case 3)
  - E.g, **x** = Yes, No, No, Yes, Full, $, No, No, Thai, 30–60, **Yes, No, No**; y = No
  - The **Yes, No, No** columns are the output of the base models
- We use this new (valid) dataset to train a new model (e.g., Logistic Regression)
- Do you know what Kaggle is?
  - Stacking is frequently used in winning entries of Kaggle competitions.
- Stacking in colab.

Boosting

Weighted Training Set

- Each example has an associated weight $w_j \geq 0$ that describes how much the example should count during training
- For example, if one example had a weight of 3 and the other examples all had a weight of 1, that would be equivalent to having 3 copies of the one example in the training set.

Weighted Training Set and Boosting

- Boosting starts with equal weights $w_j = 1$ for all the examples.
- From this training set, it generates the first hypothesis, h_1.
  - In general, h_1 will classify some of the training examples correctly and some incorrectly.
  - We would like the next hypothesis to do better on the misclassified examples →
    - **increase weights of misclassified examples and decrease the weights of the correctly classified one**.
- From this new weighted training set, we generate hypothesis h_2.
- The process continues in this way until we have generated K hypotheses
  - where K is an input to the boosting algorithm.
- Examples that are difficult to classify will get increasingly larger weights until the algorithm is forced to create a hypothesis that classifies them correctly.

Drawbacks

- Note that this is a greedy algorithm in the sense that it does not backtrack; once it has chosen a hypothesis h_i it will never undo that choice; rather it will add new hypotheses.
- It is also a sequential algorithm, so we can’t compute all the hypotheses in parallel as we could with bagging.

Inference

- The final ensemble lets each hypothesis vote, as in bagging, except that each hypothesis gets a weighted number of votes
  - the hypotheses that did better on their respective weighted training sets are given more voting weight.
- For regression or binary classification we have

$$h(\mathbf{x}) = \sum_{i=1}^{K} z_i h_i(\mathbf{x})$$

- where z_i is the weight of the i^{th} hypothesis.
  - This weighting of hypotheses is distinct from the weighting of examples.

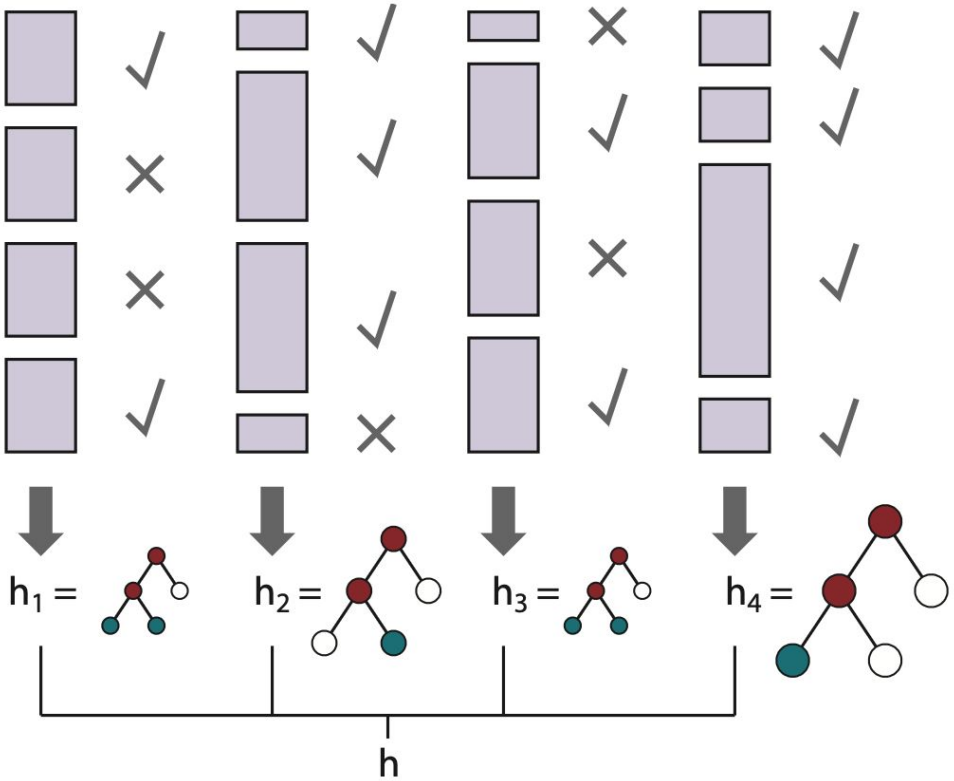


Figure 19.24 How the boosting algorithm works. Each shaded rectangle corresponds to an example; the height of the rectangle corresponds to the weight. The checks and crosses indicate whether the example was classified correctly by the current hypothesis. The size of the decision tree indicates the weight of that hypothesis in the final ensemble.

AdaBoost

- It is usually applied with decision trees as the component hypotheses; often the trees are limited in size.
- AdaBoost has a very important property:
  - if the input learning algorithm L is a weak learning algorithm
    - which means that L always returns a hypothesis with accuracy on the training set that is slightly better than random guessing (that is, 50%+ε for Boolean classification)
  - then ADABOOST will return a hypothesis that classifies the training data perfectly for large enough K.
  - Thus, the algorithm boosts the accuracy of the original learning algorithm on the training data.
- In other words, boosting can overcome any amount of bias in the base model, as long as the base model is ε better than random guessing.
  - In our pseudocode we stop generating hypotheses if we get one that is worse than random
- This result holds no matter how inexpressive the original hypothesis space and no matter how complex the function being learned.

AdaBoost

function ADABOOST(*examples*, L, K) **returns** a hypothesis
  inputs: *examples*, set of N labeled examples $(x_1, y_1), \ldots, (x_N, y_N)$
    L, a learning algorithm
    K, the number of hypotheses in the ensemble
  local variables: **w**, a vector of N example weights, initially all $1/N$
    h, a vector of K hypotheses
    z, a vector of K hypothesis weights

  $\epsilon \leftarrow$ a small positive number, used to avoid division by zero
  for $k = 1$ **to** K **do**
    $\mathbf{h}[k] \leftarrow L(examples, \mathbf{w})$
    $error \leftarrow 0$
    for $j = 1$ **to** N **do** // *Compute the total error for* $\mathbf{h}[k]$
      if $\mathbf{h}[k](x_j) \neq y_j$ **then** $error \leftarrow error + \mathbf{w}[j]$
    if $error > 1/2$ **then break** from loop
    $error \leftarrow \min(error, 1 - \epsilon)$
    for $j = 1$ **to** N **do** // *Give more weight to the examples* $\mathbf{h}[k]$ *got wrong*
      if $\mathbf{h}[k](x_j) = y_j$ **then** $\mathbf{w}[j] \leftarrow \mathbf{w}[j] \cdot error/(1 - error)$
    $\mathbf{w} \leftarrow$ NORMALIZE(**w**)
    $\mathbf{z}[k] \leftarrow \frac{1}{2} \log((1 - error)/error)$ // *Give more weight to accurate* $\mathbf{h}[k]$
  return $Function(x) : \sum \mathbf{z}_i \mathbf{h}_i(x)$

Figure 19.25 The ADABOOST variant of the boosting method for ensemble learning. The algorithm generates hypotheses by successively reweighting the training examples.

Decision Stumps and Performance on Restaurant Data

- We will choose as our original hypothesis space the class of decision stumps, which are decision trees with just one test, at the root.

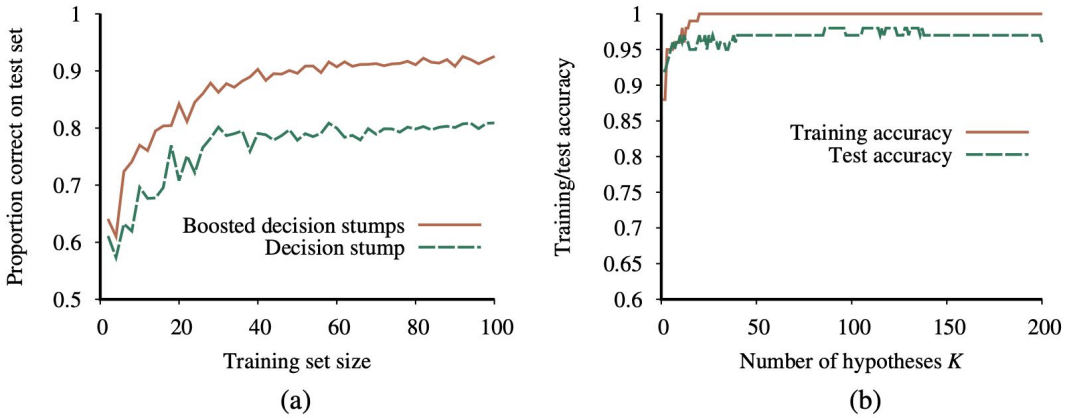


Figure 19.26 (a) Graph showing the performance of boosted decision stumps with $K = 5$ versus unboosted decision stumps on the restaurant data. (b) The proportion correct on the training set and the test set as a function of K, the number of hypotheses in the ensemble. Notice that the test set accuracy improves slightly even after the training accuracy reaches 1, i.e., after the ensemble fits the data exactly.

Decision Stumps and Generalization

- Error reaches zero when K is 20
  - that is, a weighted-majority combination of 20 decision stumps suffices to fit the 100 examples exactly; this is the interpolation point.
- As more stumps are added to the ensemble, the error remains at zero.
- The graph also shows that the test set performance continues to increase long after the training set error has reached zero.
- At K = 20, the test performance is 0.95 (or 0.05 error), and the performance increases to 0.98 as late as K = 137, before gradually dropping to 0.95.

Gradient Boosting

What is it?

- For regression and classification of factored tabular data, gradient boosting, sometimes called gradient boosting machines (GBM) or gradient boosted regression trees (GBRT), has become a very popular method.
- As the name implies, gradient boosting is a form of boosting using gradient descent.
- Recall that in ADABOOST, we start with one hypothesis h_1, and boost it with a sequence of hypotheses that pay special attention to the examples that the previous ones got wrong.
- In gradient boosting we also add new boosting hypotheses, which pay attention not to specific examples, but to the gradient between the right answers and the answers given by the previous hypotheses.

How does it work

- As in the other algorithms that used gradient descent, we start with a differentiable loss function;
  - we might use squared error for regression, or logarithmic loss for classification.
- As in ADABOOST, we then build a decision tree.
- With gradient boosting, we are not updating parameters of the existing model, we are updating the parameters of the next tree
  - but we must do that in a way that reduces the loss by moving in the right direction along the gradient.

How does it work

- Our goal is to "teach" a model F to predict values of the form $\hat{y} = F(x)$ by minimizing the mean squared error $n^{-1} \sum_i (\hat{y}_i - y_i)^2$, where i indexes over some training set of size n of actual values of the output variable y:
  - $\hat{y}$ = the predicted value $F(x_i)$
  - y_i = the observed value
  - n the number of samples in y
- Now, let us consider a gradient boosting algorithm with M stages. At each stage m ($1 \le m \le M$) of gradient boosting, suppose some imperfect model F_m (for low m this model may simply return $\hat{y}_i = \overline{y}$). In order to improve F_m, our algorithm should add some new estimator, $h_m(x)$.
  - Thus, $F_{m+1}(x) = F_m(x) + h_m(x) = y$, or equivalently $h_m(x) = y - F_m(x)$

How does it work

- Therefore, gradient boosting will fit h to the residual $y - F_m(x)$.
- As in other boosting variants, each F_{m+1} attempts to correct the errors of its predecessor F_m.
- In the case we want to optimize for the MSE loss w.r.t. F(x), we have

$$L_{\text{MSE}} = \frac{1}{n}(y - F(x))^2$$

$$-\frac{\partial L_{\text{MSE}}}{\partial F} = \frac{2}{n}(y - F(x)) = \frac{2}{n}h_m(x)$$

- Residuals $h_m(x)$ for a given model are proportional to the negative gradients of the mean squared error (MSE) loss function (with respect to F(x)).
- Gradient boosting could be specialized to a gradient descent algorithm, and generalizing it entails "plugging in" a different loss and its gradient.

XGBoost

- Gradient boosting is implemented in the popular XGBOOST (eXtreme Gradient Boosting) package, which is routinely used for both large-scale applications in industry (for problems with billions of examples), and by the winners of data science competitions (in 2015, it was used by every team in the top 10 of the KDDCup).
- XGBOOST does gradient boosting with pruning and regularization, and takes care to be efficient, carefully organizing memory to avoid cache misses, and allowing for parallel computation on multiple machines.

An Application to Search Engines

https://support.google.com

Google Support

General Help Center experience. Next. How can we help you? **Google** Chrome · **Google** Account · YouTube · Gmail · **Google** Play · **Google Search** · AdSense ...
Google Search Help · Google Account Help · Gmail · Custom Search

https://support.google.com › websearch › community

Community forum - Google Search Help - Google Support

Incorrect **Search** Results. Somebody as a cruel prank reported me missing about a year ago with some very inflammatory details · Why is **google** not allowing me ...

https://support.google.com › customsearch

Custom Search Help - Google Support

Choose sites to include in your **search** engine · Edit your **search** engine with the Control Panel · Add custom **search** to your ... Verify your site in **Search** Console.

https://www.google.com › contact

Contact us – Google

Visit Webmaster Central – the fastest way to get help with increasing traffic to your site, and see your site's crawling, indexing and **search** traffic data.

https://www.google.com › webmasters › support

Get Help and Support for your Website - Google Webmasters

Have questions about **Search** Console, **search** rankings, security issues or content on your site? Find the right **support** channel here.

https://developers.google.com › custom-search › docs › support

Support | Custom Search | Google Developers

Mar 25, 2019 - **Support**. For **support**, visit the Help Center or the Help Community. Except as otherwise noted, the content of this page is licensed under the ...

The Importance of Ranking

- Users want to look at a few results – not thousands.
- It’s very hard to write queries that produce a few results.
- Even for expert searchers
- → Ranking is important because it effectively reduces a large set of results to a very small one.

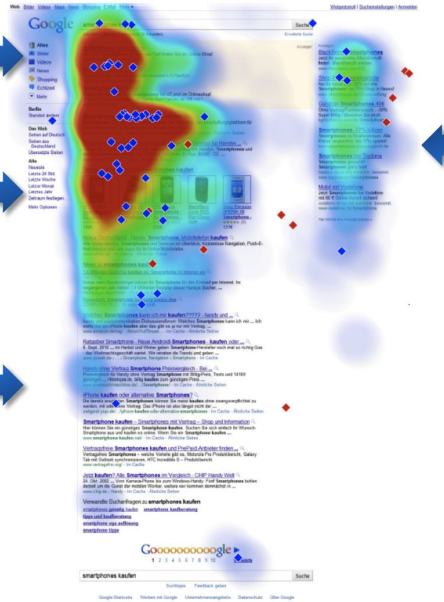
Attention of Users

Desktop search engine result page

Strong focus on the AdWords results

Rich Media elements move the attention/ fixation points down the site

Nearly no focus on the organic results below the fold



People look more intensly on the right hand side than on the organic results below the fold

thinkinsights with Google

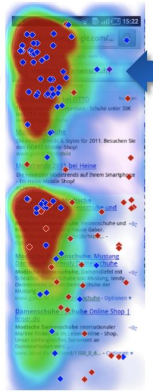
Source: Eye Square Eye Tracking Study, 2011
Base: Resondents with contact to the stationary advertising on Google (n=38 Stationary)
Info: Aggregation over three brands

Attention of Users

Mobile search engine results page

First focus on the search bar and the first AdWords ad

Second focus on the first organic result



First two screens are carefully checked

A traditional above and below the fold is missing on the mobile page

Ads at the end of the page are noticed by the respondents

thinkinsights with Google

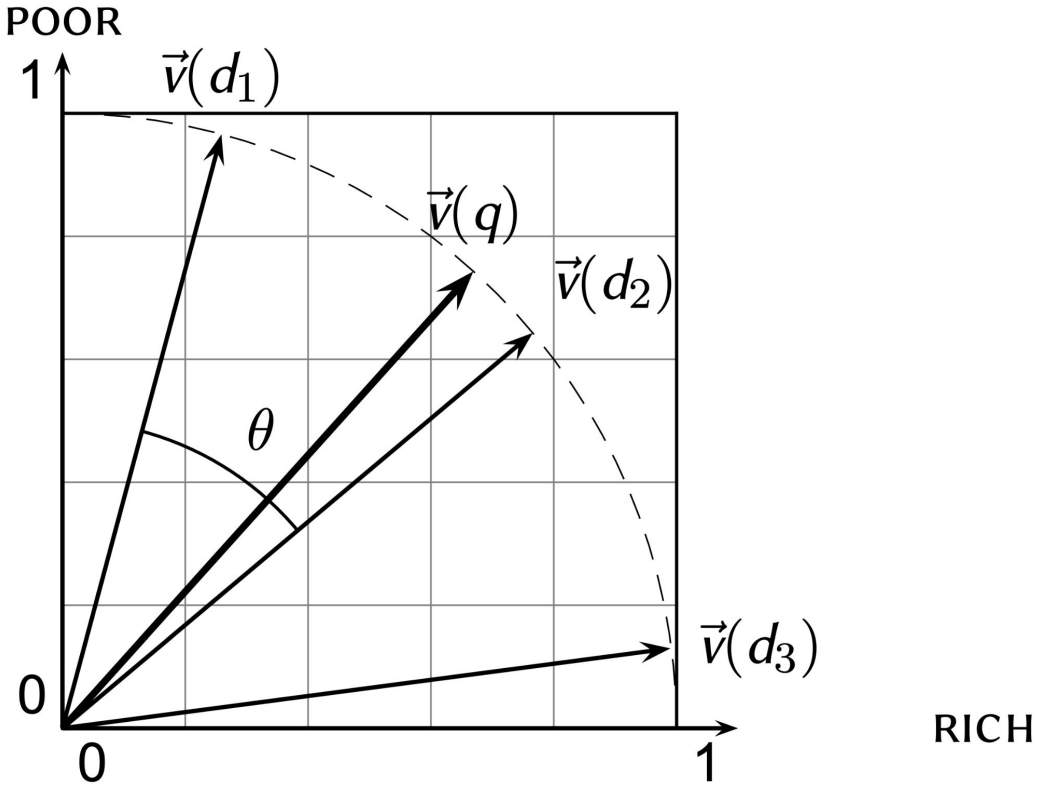
Source: Eye Square Eye Tracking Study, 2011
Base: Respondents with contact to the mobile advertising on Google (n=50 mobile)
Info: Aggregation over three brands

Cosine Similarity between Query and Document

$$\cos(\vec{q},\vec{d}) = \text{SIM}(\vec{q},\vec{d}) = \frac{\vec{q}}{|\vec{q}|} \cdot \frac{\vec{d}}{|\vec{d}|} = \sum_{i=1}^{|V|} \frac{q_i}{\sqrt{\sum_{i=1}^{|V|} q_i^2}} \cdot \frac{d_i}{\sqrt{\sum_{i=1}^{|V|} d_i^2}}$$

- q_i is the tf-idf weight of term *i* in the query.
- d_i is the tf-idf weight of term *i* in the document.
- $|\boldsymbol{q}|$ and $|\boldsymbol{d}|$ are the lengths of vectors ***q*** and ***d****, respectively*.
- $\boldsymbol{q}/|\boldsymbol{q}|$ and $\boldsymbol{d}/|\boldsymbol{d}|$ are length-1 vectors (= normalized)

Computing Scores



Other factors affecting scores

- Query-Click pairs
- Different sections of a document have different importance
- Date, time of query affects score
- Location of the searcher
- Demographic information
- History of queries
- History of clicks
- History of visited pages
- ...

Simple example: Classification for ad hoc IR

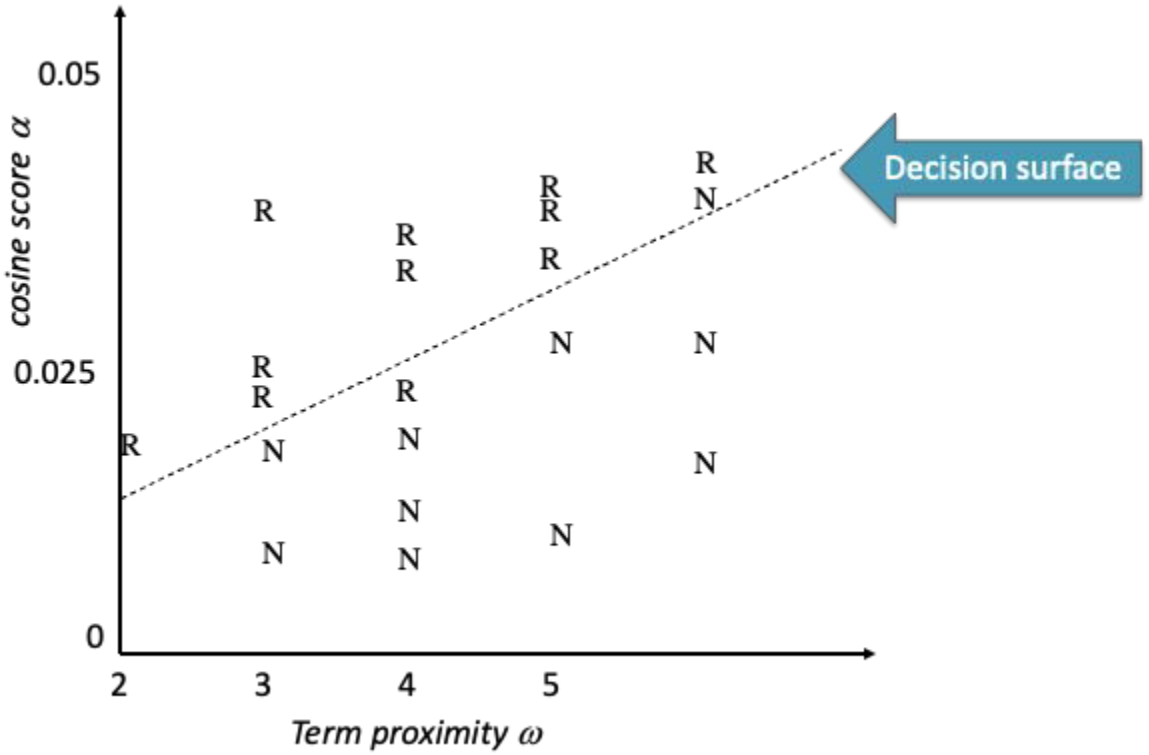
- Collect a training corpus of (q, d, r) triples
  - Relevance r is here binary (but may be multiclass, with 3–7 values)
  - Query-Document pair is represented by a feature vector
    - x = (α, ω) → α is cosine similarity, ω is minimum query window size
      - ω is the the shortest text span that includes all query words
      - Query term proximity is an important new weighting factor
  - Train a machine learning model to predict the class r of a document-query pair

example	docID	query	cosine score	ω	judgment
Φ_1	37	linux operating system	0.032	3	*relevant*
Φ_2	37	penguin logo	0.02	4	*nonrelevant*
Φ_3	238	operating system	0.043	2	*relevant*
Φ_4	238	runtime environment	0.004	2	*nonrelevant*
Φ_5	1741	kernel layer	0.022	3	*relevant*
Φ_6	2094	device driver	0.03	2	*relevant*
Φ_7	3191	device driver	0.027	5	*nonrelevant*

Simple example: Classification for ad hoc IR

- A linear score function is then
  - Score(d, q) = Score(α, ω) = aα + bω + c
- And the linear classifier is
  - Decide relevant if Score(d, q) > θ

Simple example: Classification for ad hoc IR



Using classification for search ranking

- We can generalize this to classifier functions over more features
- We can use other methods for learning the linear classifier weights

An SVM classifier for information retrieval

- Let relevance score $g(r|d,q) = wf(d,q) + b$
- Uses SVM: want $g(r|d,q) \leq -1$ for non relevant documents and $g(r|d,q) \geq 1$ for relevant documents
- SVM testing: decide relevant iff $g(r|d,q) \geq 0$

- Features are not word presence features (how would you deal with query words not in your training data?) but scores like the summed (log) tf of all query terms
- Unbalanced data (which can result in trivial always-say-nonrelevant classifiers) is dealt with by undersampling nonrelevant documents during training (just take some at random)

An SVM classifier for information retrieval

Train \ Test		Disk 3	Disk 4-5	WT10G (web)
TREC Disk 3	Lemur	**0.1785**	**0.2503**	0.2666
	SVM	0.1728	0.2432	**0.2750**
Disk 4-5	Lemur	**0.1773**	**0.2516**	0.2656
	SVM	0.1646	0.2355	**0.2675**

- At best the results are about equal to Lemur
  - Actually a little bit below
- Paper's advertisement: Easy to add more features
  - This is illustrated on a homepage finding task on WT10G:
    - Baseline Lemur 52% success@10, baseline SVM 58%
    - SVM with URL-depth, and in-link features: 78% success@10

Is it really learning to “rank”?

- Classification probably isn’t the right way to think about approaching ad hoc IR:
  - Classification problems: Map to an unordered set of classes
  - Regression problems: Map to a real value
  - Ordinal regression (or “ranking”) problems: Map to an ordered set of classes

- This formulation gives extra power:
  - Relations between relevance levels are modeled
  - **Documents are good versus other documents for a query given collection**; not an absolute scale of goodness

Multiple Additive Regression Trees (MART)

Algorithm 1 Multiple Additive Regression Trees.

Initialize $F_0(\mathbf{x}) = \arg\min_\gamma \sum_{i=1}^{N} L(y_i, \gamma)$
for $m = 1, ..., M$ **do**
  for $i = 1, ..., N$ **do**
    $\tilde{y}_{im} = -\left[\frac{\partial L(y_i, F(\mathbf{x}_i))}{\partial F(\mathbf{x}_i)}\right]_{F(\mathbf{x})=F_{m-1}(\mathbf{x})}$
  end for
  $\{R_{km}\}_{k=1}^{K}$ // Fit a regression tree to targets $\tilde{y}_{im}$
  for $k = 1, ..., K_m$ **do**
    $\gamma_{km} = \arg\min_\gamma \sum_{x_i \in R_{jm}} L(y_i, F_{m-1}(\mathbf{x}_i) + \gamma)$
  end for
  $F_m(\mathbf{x}) = F_{m-1}(\mathbf{x}) + \eta \sum_{k=1}^{K_m} \gamma_{km} \mathbf{1}(\mathbf{x}_i \in R_{km})$
end for
Return $F_M(\mathbf{x})$

Historical Path to LambdaMART via RankNet (NN)

- Have differentiable function with model parameters w:
  - $x_i \rightarrow f(x;w)=s_i$
- For query q, learn probability of different ranking class for documents $d_i > d_j$ via:
  - $P_{ij}=P(d_i>d_j)=1/(1+e^{(-\sigma(si-sj))})$
- Cost function calculates cross entropy loss:
  - $C=-\bar{P}_{ij} \log P_{ij} - (1-\bar{P}_{ij}) \log(1-P_{ij})$
- Where $\bar{P}_{ij}$ is the model probability; P_{ij} the actual probability (0 or 1 for categorical judgments)

Historical Path to LambdaMART via RankNet (NN)

- Combining these equations gives
- $C=1/2(1-S_{ij})\sigma(s_i-s_j)+\log(1+e^{(-\sigma(si-sj))})$
- where, for a given query, $S_{ij} \in \{0,+1,-1\}$
  1 if d_i is more relevant than d_j; –1 if the reverse, and 0 if the they have the same label

$$\frac{\partial C}{\partial s_i} = \sigma\left(\frac{1}{2}(1-S_{ij}) - \frac{1}{1+e^{\sigma(s_i-s_j)}}\right) = -\frac{\partial C}{\partial s_j}$$

$$\frac{\partial C}{\partial w_k} = \frac{\partial C}{\partial s_i}\frac{\partial s_i}{\partial w_k} + \frac{\partial C}{\partial s_j}\frac{\partial s_j}{\partial w_k} = \sigma\left(\frac{1}{2}(1-S_{ij}) - \frac{1}{1+e^{\sigma(s_i-s_j)}}\right)\left(\frac{\partial s_i}{\partial w_k} - \frac{\partial s_j}{\partial w_k}\right)$$
$$= \lambda_{ij}\left(\frac{\partial s_i}{\partial w_k} - \frac{\partial s_j}{\partial w_k}\right)$$

SAPIENZA
UNIVERSITÀ DI ROMA

Historical Path to LambdaMART via RankNet (NN)

- The crucial part of the update is

$$\frac{\partial C}{\partial w_k} = \frac{\partial C}{\partial s_i}\frac{\partial s_i}{\partial w_k} + \frac{\partial C}{\partial s_j}\frac{\partial s_j}{\partial w_k} = \lambda_{ij}\left(\frac{\partial s_i}{\partial w_k} - \frac{\partial s_j}{\partial w_k}\right)$$

- λ_{ij} describes the desired change of scores for the pair of documents d_i and d_j
- The sum of all λ_{ij}'s and λ_{ji}'s of a query-doc vector x_i w.r.t. all other differently labelled documents for q is

$$\lambda_i = \sum_{j:\{i,j\}\in I} \lambda_{ij} - \sum_{k:\{k,i\}\in I} \lambda_{ki}$$

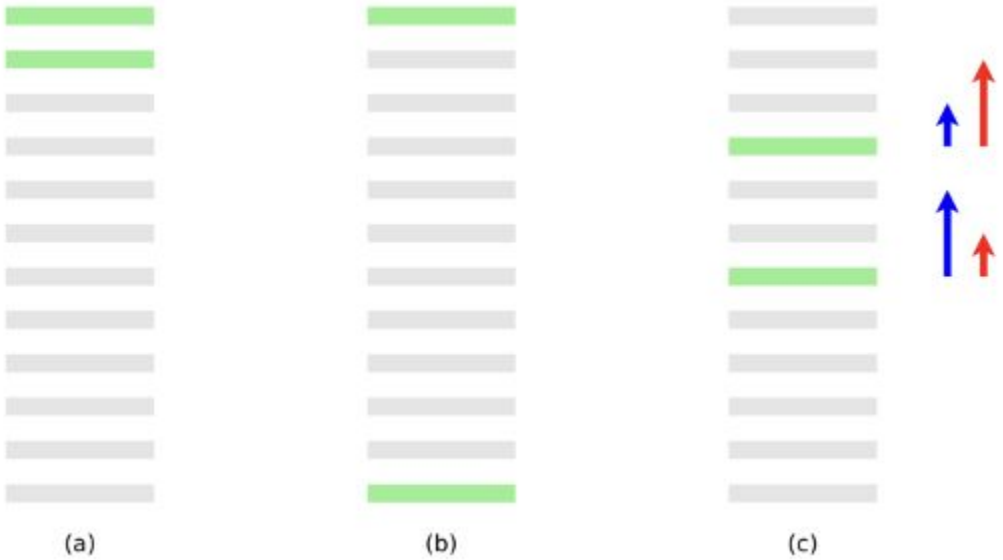
- λ_i is (sort of) a gradient of the pairwise loss of vector x_i

RankNet lambdas

- (a) is the perfect ranking, (b) is a ranking with 10 pairwise errors, (c) is a ranking with 8 pairwise errors. Each blue arrow represents the λ_i for each query-document vector x_i

RankNet lambdas

- Problem: RankNet is based on pairwise error, while modern IR measures emphasize higher ranking positions. Red arrows show better λ's for modern IR, esp. web search.



From RankNet to LambdaRank

- Rather than working with pairwise ranking errors, scale by effect a change has on NDCG
- Idea: Multiply λ's by $|\Delta Z|$, the difference of an IR measure when d_i and d_j are swapped
- E.g. $|\Delta NDCG|$ is the change in NDCG when swapping d_i and d_j giving

$$\lambda_{ij} = \frac{\partial C(s_i - s_j)}{\partial s_i} = \frac{-\sigma}{1+e^{\sigma(s_i - s_j)}} |\Delta NDCG|$$

- Burges et al. "prove" (partly theory, partly empirical) that this change is sufficient for model to optimize NDCG

From RankNet to LambdaRank

- LambdaRank models gradients
- MART can be trained with gradients (“gradient boosting”)
- Combine both to get LambdaMART
  - MART with specified gradients and optimization step

LambdaRank Algorithm

set number of trees N, number of training samples m, number of leaves per tree L, learning rate η
for $i = 0$ to m **do**
  $F_0(x_i) = \text{BaseModel}(x_i)$ //If BaseModel is empty, set $F_0(x_i) = 0$
end for
for $k = 1$ to N **do**
  for $i = 0$ to m **do**
    $y_i = \lambda_i$
    $w_i = \frac{\partial y_i}{\partial F_{k-1}(x_i)}$
  end for
  $\{R_{lk}\}_{l=1}^{L}$ // Create L leaf tree on $\{x_i, y_i\}_{i=1}^{m}$ R_{lk} is data items at leaf node l
  $\gamma_{lk} = \frac{\sum_{x_i \in R_{lk}} y_i}{\sum_{x_i \in R_{lk}} w_i}$ // Assign leaf values based on Newton step.
  $F_k(x_i) = F_{k-1}(x_i) + \eta \sum_l \gamma_{lk} I(x_i \in R_{lk})$ // Take step with learning rate η.
end for

Yahoo! Learning to rank challenge

- Goal was to validate learning to rank methods on a large, “real” web search problem
  - Previous work was mainly driven by LETOR datasets
    - Great as first public learning-to-rank data
    - Small: 10s of features, 100s of queries, 10k’s of docs
- Only feature vectors released
  - Not URLs, queries, nor feature descriptions
    - Wanting to keep privacy and proprietary info safe
  - But included web graph features, click features, page freshness and page classification features as well as text match features

https://colab.research.google.com/drive/1hFgEXszkAYmOObJ4YW-8qXxOohuTvSLe

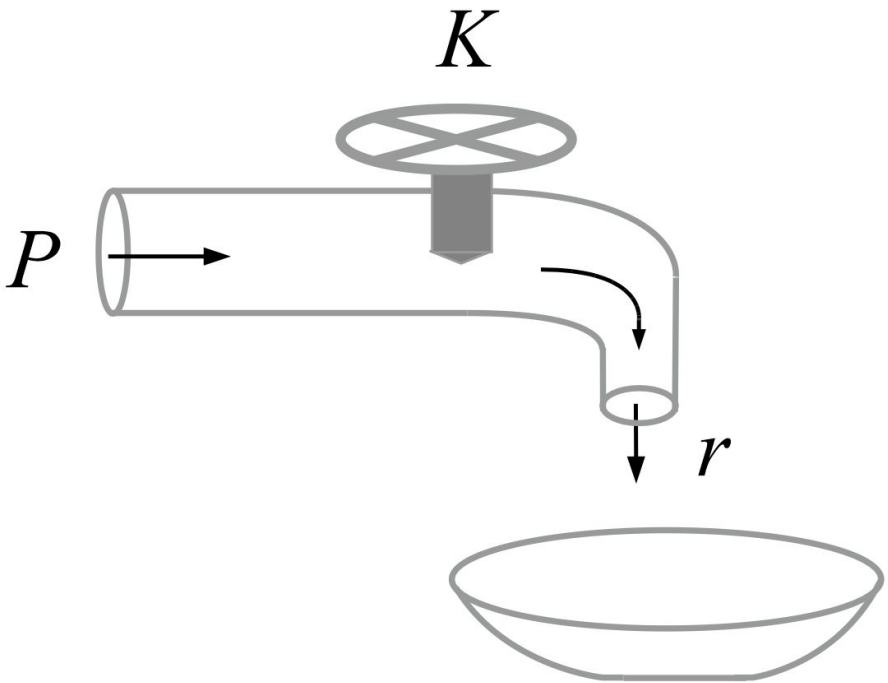
Neural Networks

Water in a Sink

A pipe is supplied with water at a given pressure P.

The knob K can adjust the water pressure, providing a variable outgoing water flow at rate r.

The knob influences the rate by introducing a nonnegative control w such that P = r/w.

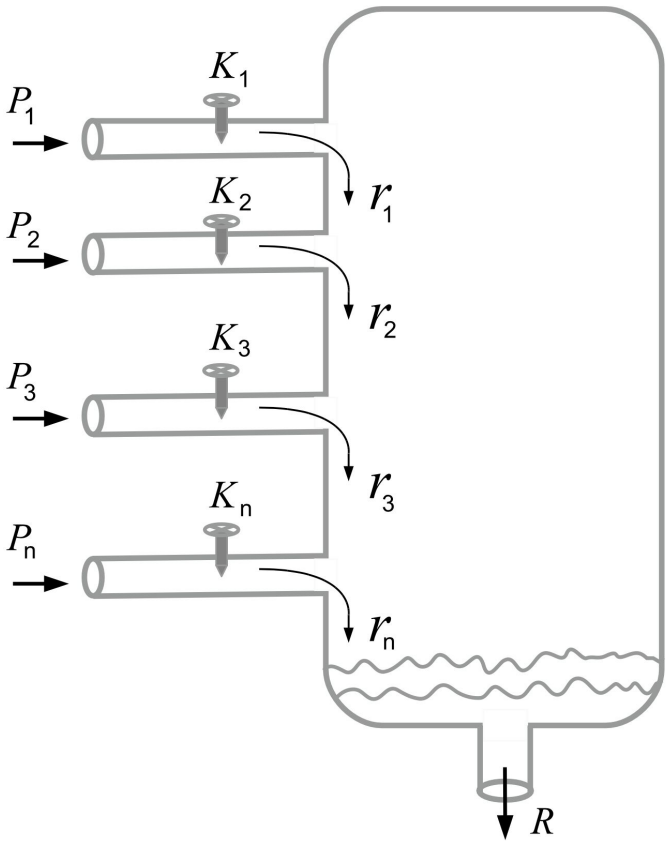


Water in a Sink

A certain number of pipes of this type are used to pour water into a tank.

Simultaneously, the tank is drained at a rate R.

Problem:
Given the water pressure supplies $P_1, \ldots, P_n$, how can one adjust the knobs $K_1, \ldots, K_n$ such that after an a priori fixed interval of time t there is a predetermined volume V of water in the tank?



Water in a Sink

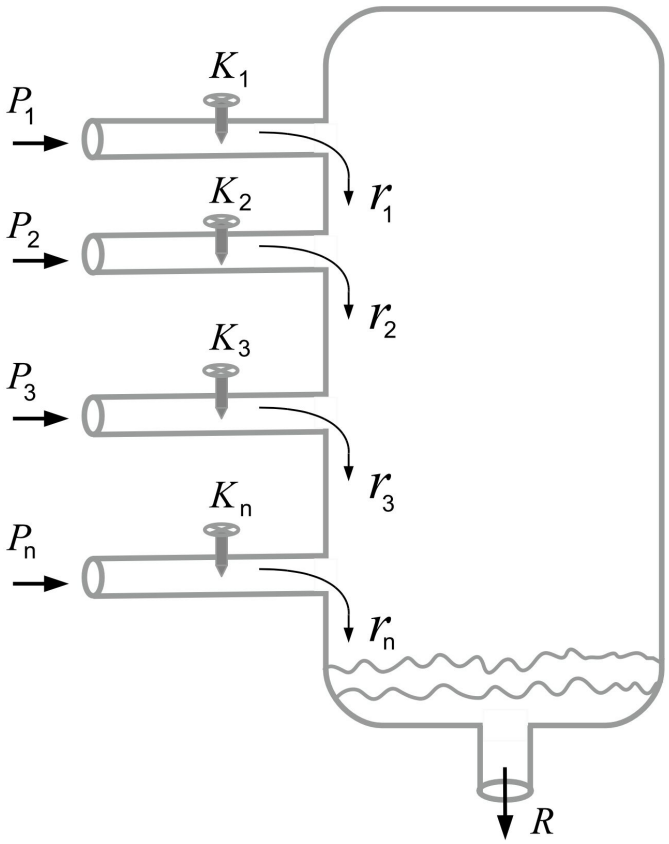
Denote by $r_1,...,r_n$ the inflow rates adjusted by the knobs $K_1,...,K_n$.

If the outflow rate exceeds the total inflow rate, i.e., if $R > \sum_{i=1}^{n} r_i$, then no water will accumulate in the tank, i.e., $V = 0$ at any future time.

Otherwise, if the total inflow rate is larger than the outflow rate, i.e., if $\sum_{i=1}^{n} r_i > R$, then the water accumulates at the rate given by the difference $\sum_{i=1}^{n} r_i - R$ tes , accumulating over time t an amount of water $(\sum_{i=1}^{n} r_i - R)t$.

The resulting water amount can be written as a piecewise function

$$V = \begin{cases} 0, & \text{if } R > \sum_{i=1}^{n} r_i \\ (\sum_{i=1}^{n} r_i - R)t, & \text{otherwise.} \end{cases}$$



Water in a Sink

Alternati
provided
pipe sup

This will be, later on, known as ReLU activation function

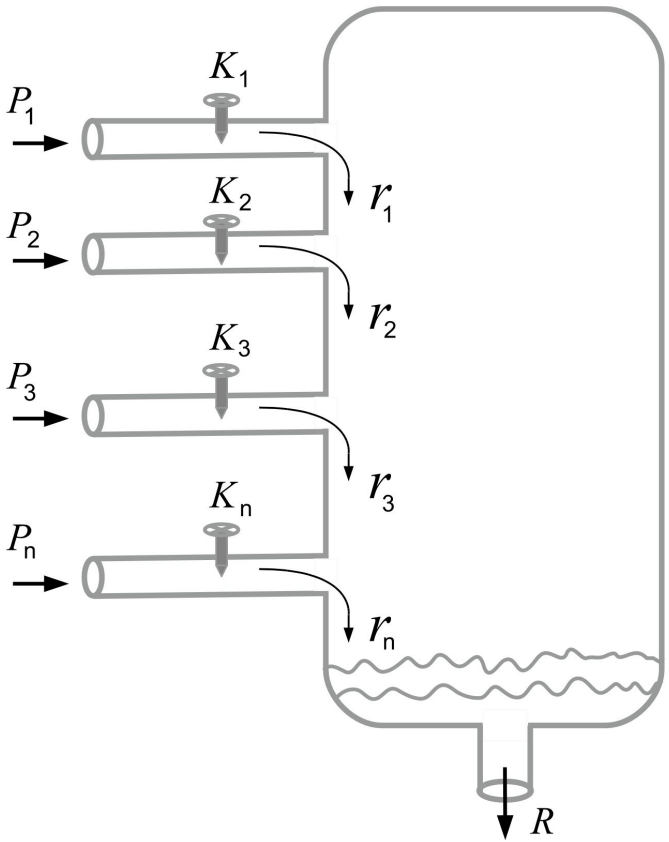
Consider
(defined later)

$$\varphi_t(x) = \begin{cases} 0, & \text{if } x < 0 \\ xt, & \text{otherwise} \end{cases}$$

which depends on the time parameter $t > 0$.

Now, the volume of water, V, can be written in terms of the pressures P_i, controls w_i, and function φ_t as

$$V = \varphi_t\left(\sum_{i=1}^{n} P_i w_i - R\right)$$



Water in a Sink

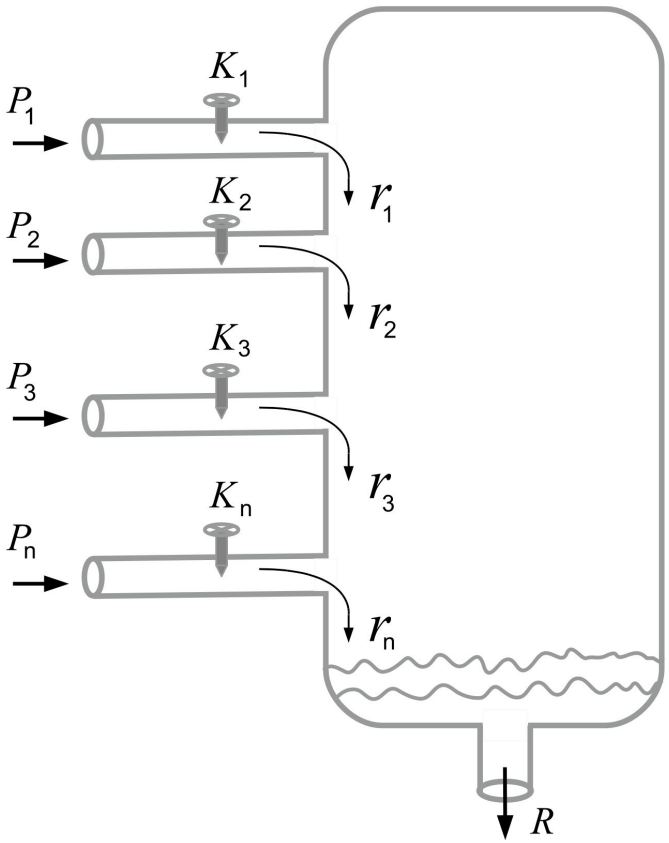
The problem reduces now to find a solution to

$$V = \varphi_t\left(\sum_{i=1}^{n} P_i w_i - R\right)$$

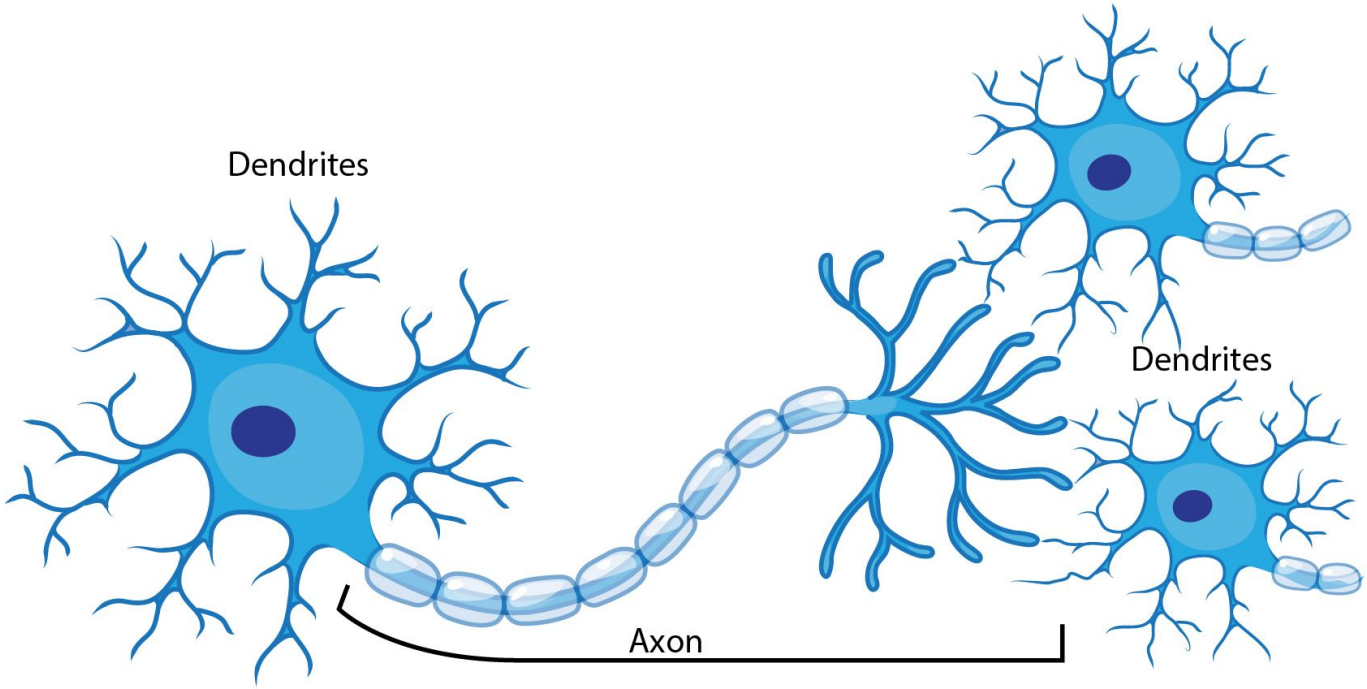
having the unknowns w_i and R.

In practice, we just need to obtain an approximation of the solution by choosing the controls wi and the bias R such that the following proximity function is minimized

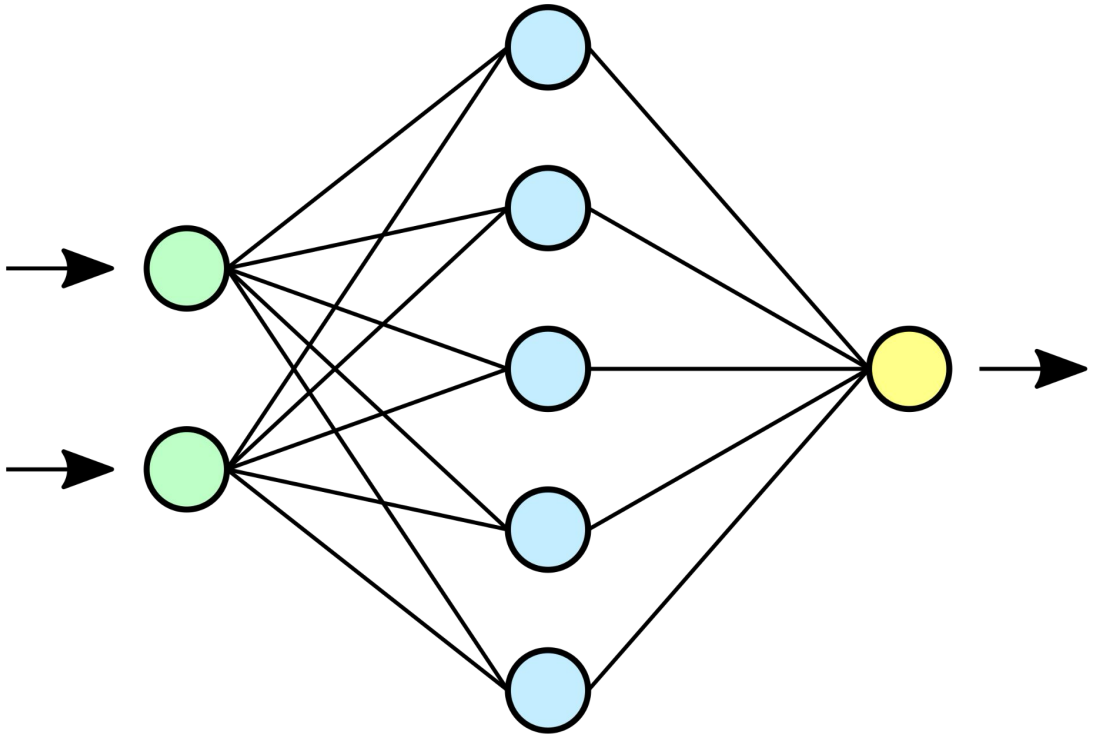
$$L(w_1, \ldots, w_n, R) = \frac{1}{2}\left(\varphi_t\left(\sum_{i=1}^{n} P_i w_i - R\right) - V\right)^2$$



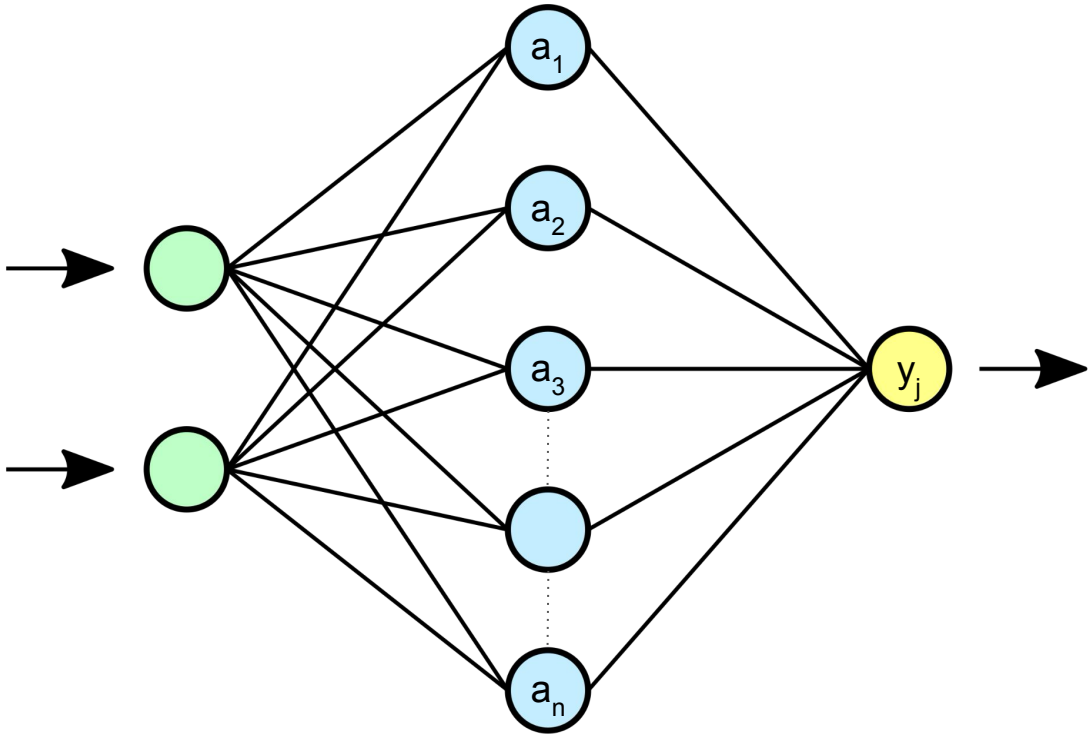
Biological Inspiration



Simple Feedforward Neural Networks

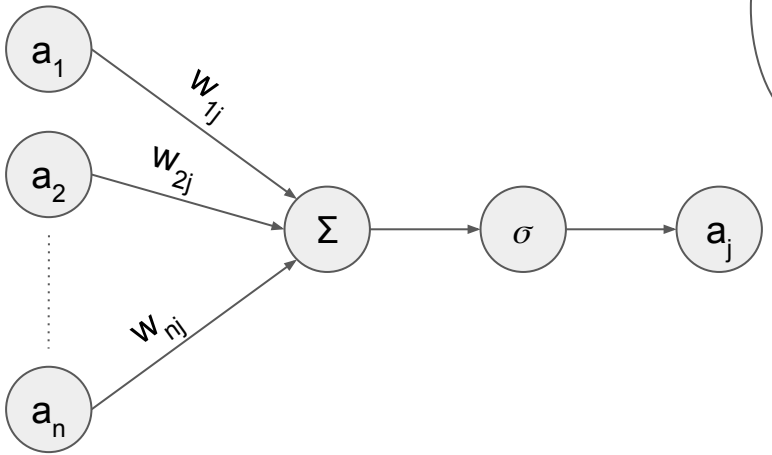


Simple Feedforward Neural Networks



Simple Feedforward Neural Networks

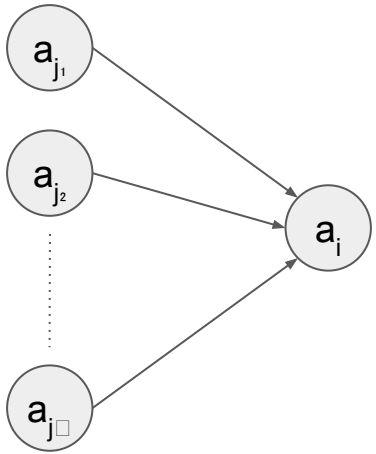
$$a_j = \sigma\left(\sum_i w_{ij} a_i\right)$$



Activation function, e.g., Sigmoid, ReLU, Tanh, etc.

Notation

- For simplicity, in the discussion following, we assume nodes are indexed in a way that
  if a_i depends on a_j then $i > j$.

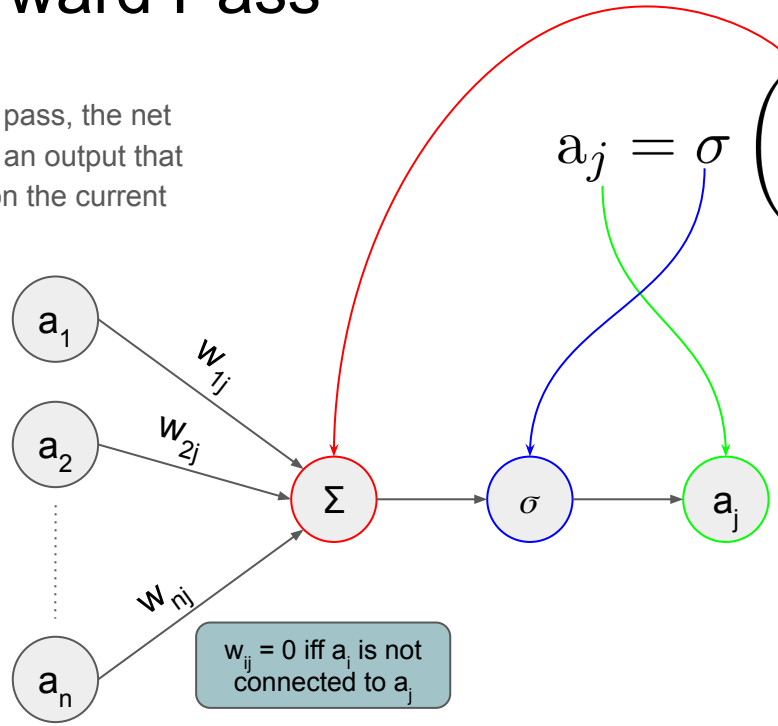


$$j_m < i \qquad \forall m \in [1, k] \cap \mathbb{N}$$

The Forward Pass

- In the fwd pass, the net computes an output that is based on the current input.

$$a_j = \sigma\left(\sum_i w_{ij} a_i\right)$$

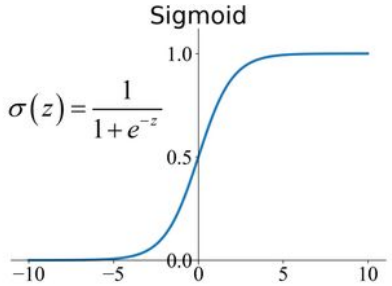


σ has to be:
- Bounded
- Monotonic
- Squashing

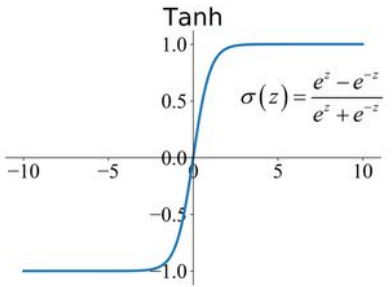
For instance:
- Sigmoid
- Tanh
- ~~ReLU~~

Non Linearities

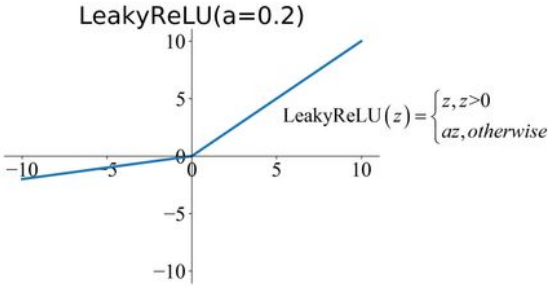
- Has to be:
  - Bounded
  - Monotonic
  - Squashing



(a)

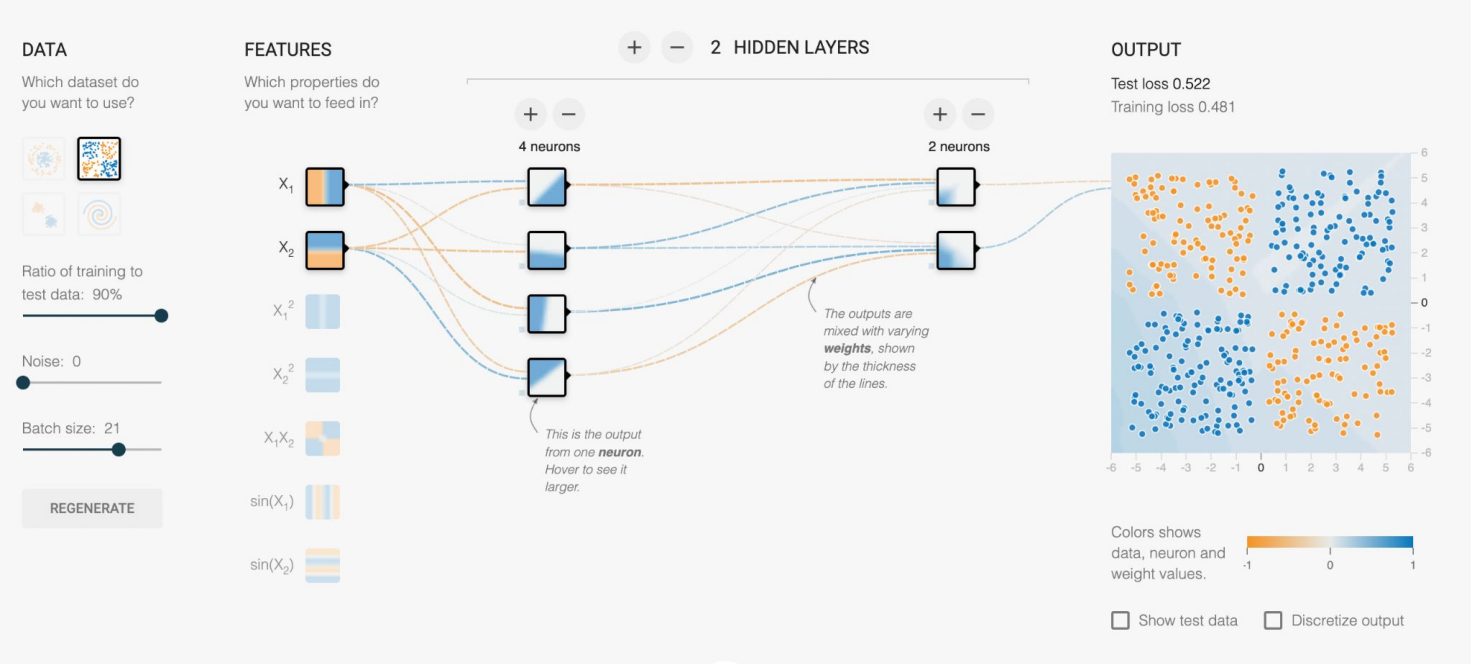


(b)



(d)

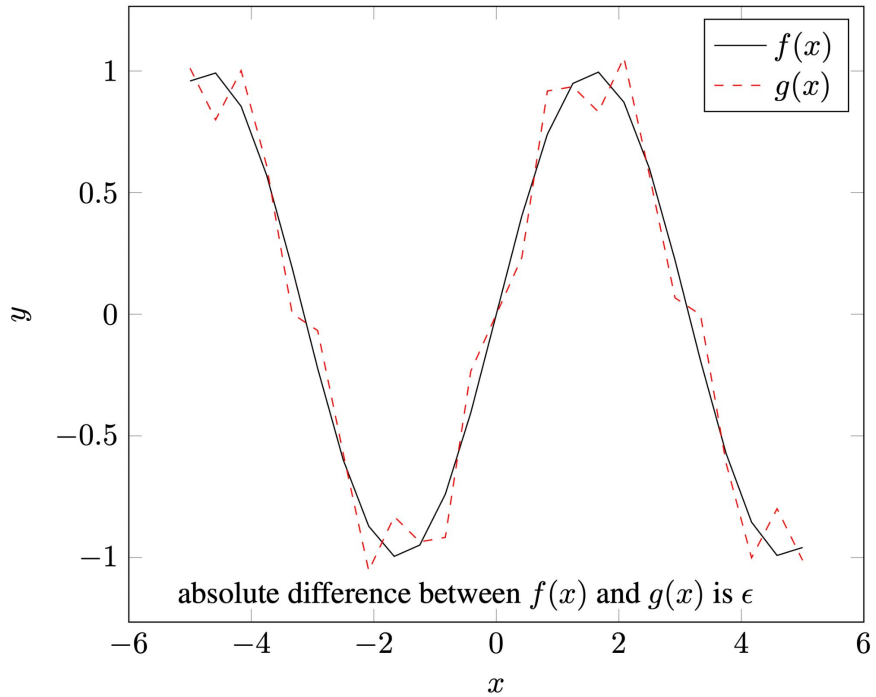
Effect of Nonlinearity in NN



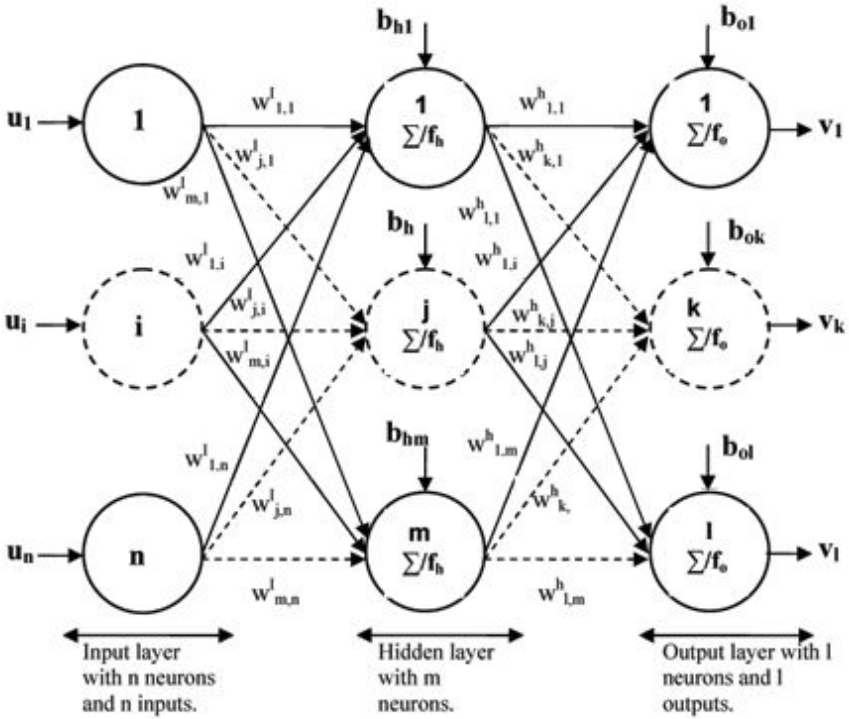
- Play with: https://playground.tensorflow.org/

Universality of NN

- The universal approximation theorem states that any continuous function $f:[0,1]^n \rightarrow [0,1]$ can be approximated arbitrarily well by a neural network with at least 1 hidden layer with a finite number of weights



Multilayer NNs



Error (Loss) Calculation

- You can use any loss that we have encountered so far. We will base our discussion on MSE loss

$$\mathrm{E} = 1/2 \sum_{p} \sum_{i} (d_{pi} - y_{pi})^2$$

where p indexes the patterns (samples) in the training set, and i indexes the output nodes

d_{pi} and y_{pi} are, respectively, the desired target and network output for the i^{th} output node on the p^{th} pattern (sample)

Error (Loss) Calculation

- You can use any loss that we have encountered so far. We will base our discussion on MSE loss

$$\mathrm{E} = \sum_p E_p \qquad E_p = 1/2 \sum_i (d_{pi} - y_{pi})^2$$

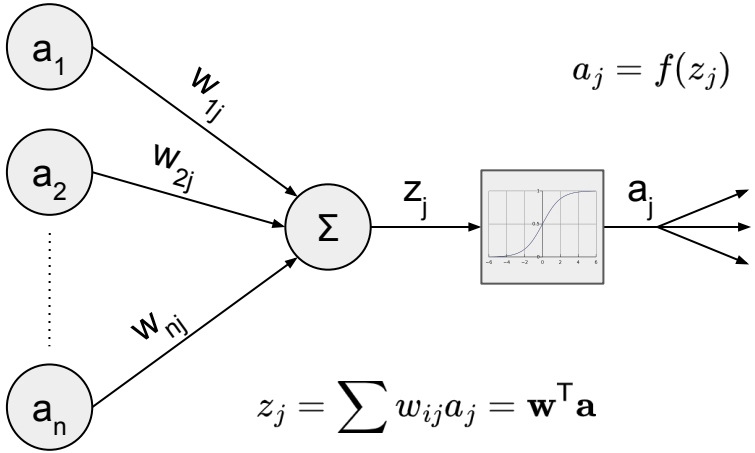
where p indexes the patterns (samples) in the training set, and i indexes the output nodes

d_{pi} and y_{pi} are, respectively, the desired target and network output for the i^{th} output node on the p^{th} pattern (sample)

How do you optimize a Neural Network Model?

- Compute the gradient of the loss w.r.t. parameters and apply (Batched/Stochastic) Gradient Descent
- This is easy in the case of a “simple” NN (e.g. 1 hidden layer).
  - Things can get hairy in the case of complex (deep) NNs.
- Things improved when Backpropagation was started to be used
- "Isn't backpropagation just the chain rule of Leibniz (1676) & L'Hopital (1696)?" No, it is the efficient way of applying the chain rule to big networks with differentiable nodes

Forward Pass



Backpropagation

- We have to compute the gradient of the loss function w.r.t. all the weights w_{ij}.

$$\frac{\partial E}{\partial w_{ij}} = \sum_p \frac{\partial E_p}{\partial w_{ij}}$$

- Backprop decomposes the calculation

$$\frac{\partial E_p}{\partial w_{ij}} = \sum_k \frac{\partial E_p}{\partial a_k} \frac{\partial a_k}{w_{ij}}$$

where k runs over all output nodes, and a_j is the weighted-sum input for node j as it follows from

$$\mathrm{a}_j = \sigma \left(\sum_{j<i} w_{ij} a_i \right)$$

Backpropagation for Output Nodes

- We first define an auxiliary variable δ_i for each node i of the network

$$\delta_i = \frac{\partial E_p}{\partial a_i} = \frac{\partial E_p}{\partial y_i}\frac{\partial y_i}{\partial a_i}$$

- Measures the contribution of a_i to the error on the current sample (pattern)
- For output nodes $\frac{\partial E_p}{\partial a_k}$ is obtained as (f is the non-linearity function)

$$\delta_k = \frac{\partial E_p}{\partial a_k} = -(d_{pk} - y_{pk}) f'_k$$

where:

- We have that $-(d_{pk} - y_{pk}) = \frac{\partial E_p}{\partial y_k}$, and
- $f'(a_k) \overset{\text{def}}{=} f'_k = \frac{\partial y_k}{\partial a_k}$

In the case of sigmoid non-linearity we have:

$$\delta_k = -(d_{pk} - y_{pk})\left[\sigma(a_k)\left(1 - \sigma(a_k)\right)\right]$$

Backpropagation for Intermediate Nodes

- Hidden nodes can influence the error only through their effect on the nodes k to which they send output connections

$$\delta_i = \frac{\partial E_p}{\partial a_i} = \sum_k \frac{\partial E_p}{\partial a_k} \frac{\partial a_k}{\partial a_i}$$

- So given that $\delta_k = \partial E_p / \partial a_k$ then

$$\delta_i = \sum_k \delta_k \frac{\partial a_k}{\partial a_i}$$

- If node i connects directly to node k then $\frac{\partial a_k}{\partial a_i} = f_i' w_{ki}$ otherwise zero.

- Overall, we have $\delta_i = f_i' \sum_k w_{ki} \delta_k$

Backpropagation: Summing Up

- Overall, we can write δ_i as

$$\delta_i = \begin{cases} f_i' \sum_k w_{ki}\delta_k & \text{for hidden nodes} \\ -(d_{pi} - y_{pi}) f_i' & \text{for output nodes} \end{cases}$$

- So, backpropagation is

$$\frac{\partial E_p}{\partial w_{ij}} = \delta_i y_j$$

where

$$y_j = \frac{\partial a_i}{\partial w_{ij}}$$

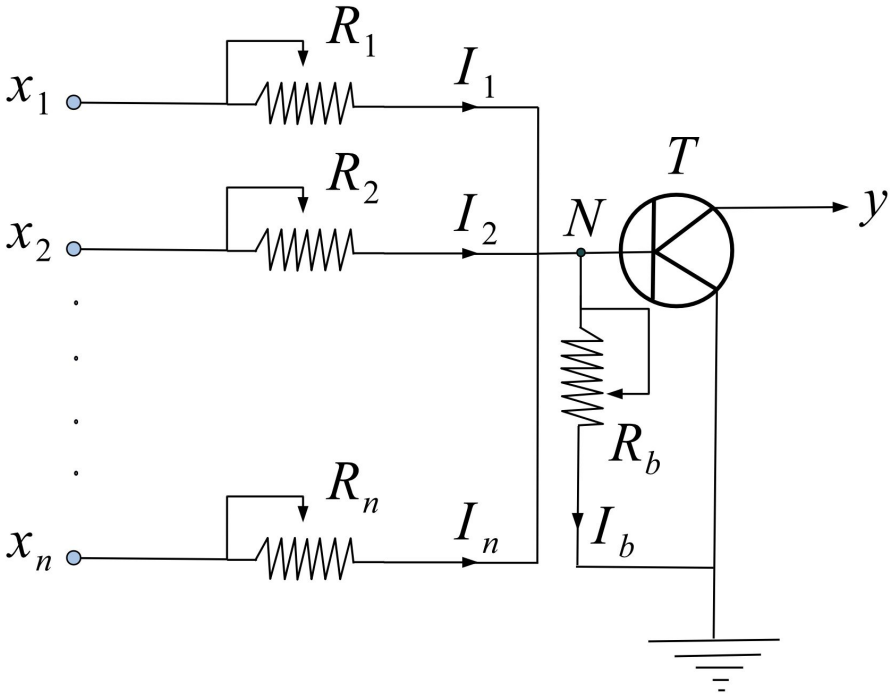
Electronic Circuit

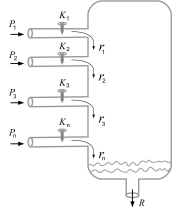
Remember Water in a Sink?

If we apply the following correspondences

Fluids	pressure	rate	knob
Electronics	voltage	intensity	resistivity

We can turn the problem from a fluid-dynamic one into electronic by considering the following circuit, which is, in fact, a physically implemented artificial neuron 👀





SAPIENZA UNIVERSITÀ DI ROMA

Shallow Neural Networks

(Chapter 3)

SAPIENZA
UNIVERSITÀ DI ROMA

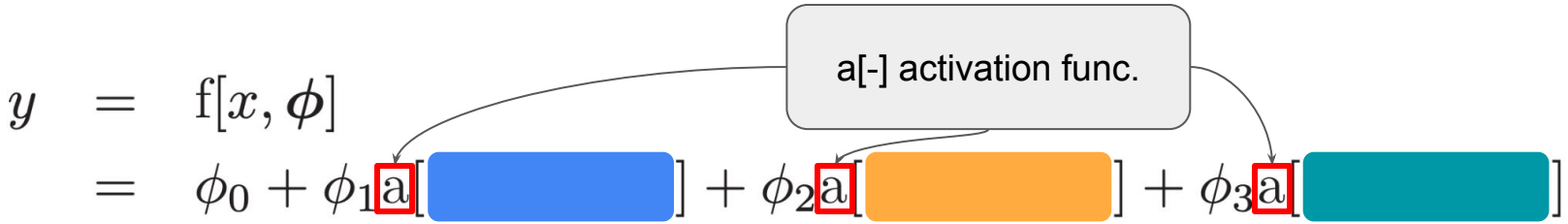
Introduction to Shallow Neural Networks

- Shallow neural networks are functions **y** = f[**x**, **ϕ**] with parameters **ϕ** that map multivariate inputs **x** to multivariate outputs **y**.
- We introduce the main ideas using an example network f[**x**, **ϕ**] that maps a scalar input **x** to a scalar output **y** and has ten parameters **ϕ** = {ϕ_0, ϕ_1, ϕ_2, ϕ_3, θ_{10}, θ_{11}, θ_{20}, θ_{21}, θ_{30}, θ_{31}}:

$$
\begin{aligned}
y &= f[x, \boldsymbol{\phi}] \\
&= \phi_0 + \phi_1 \quad + \phi_2 \quad + \phi_3
\end{aligned}
$$

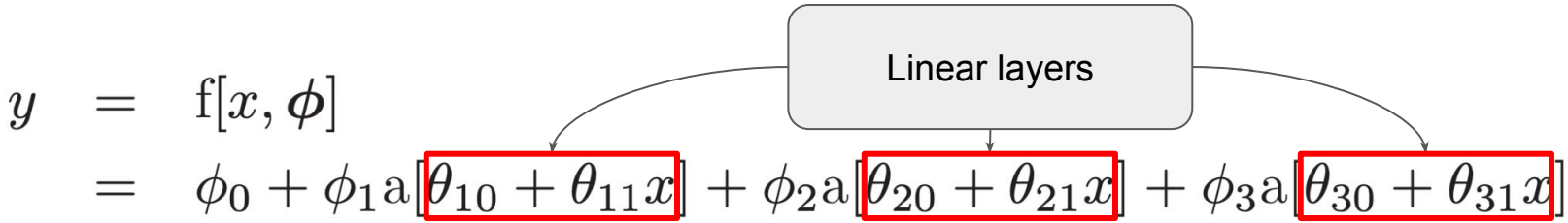
Introduction to Shallow Neural Networks

- Shallow neural networks are functions **y** = f[**x**, **ϕ**] with parameters **ϕ** that map multivariate inputs **x** to multivariate outputs **y**.
- We introduce the main ideas using an example network f[**x**, **ϕ**] that maps a scalar input **x** to a scalar output **y** and has ten parameters **ϕ** = {ϕ_0, ϕ_1, ϕ_2, ϕ_3, θ_{10}, θ_{11}, θ_{20}, θ_{21}, θ_{30}, θ_{31}}:



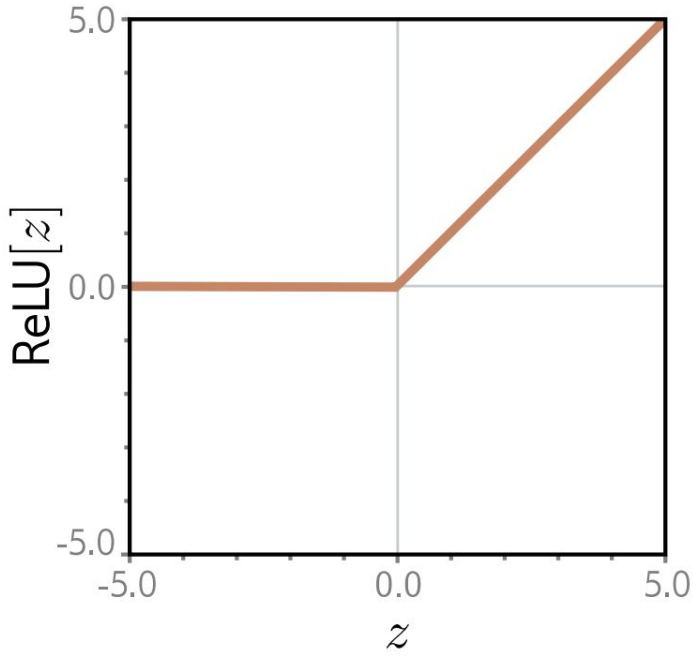
Introduction to Shallow Neural Networks

- Shallow neural networks are functions **y** = f[**x**, **ϕ**] with parameters **ϕ** that map multivariate inputs **x** to multivariate outputs **y**.
- We introduce the main ideas using an example network f[**x**, **ϕ**] that maps a scalar input **x** to a scalar output **y** and has ten parameters **ϕ** = {ϕ_0, ϕ_1, ϕ_2, ϕ_3, θ_{10}, θ_{11}, θ_{20}, θ_{21}, θ_{30}, θ_{31}}:



ReLU Activation

- It's the most common choice $\mathrm{a}[z] = \mathrm{ReLU}[z] = \begin{cases} 0 & z < 0 \\ z & z \geq 0 \end{cases}$

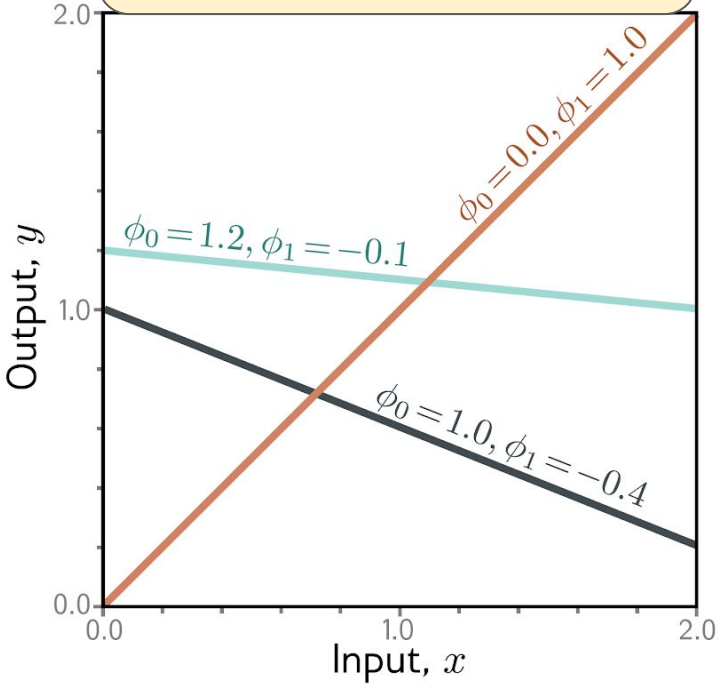


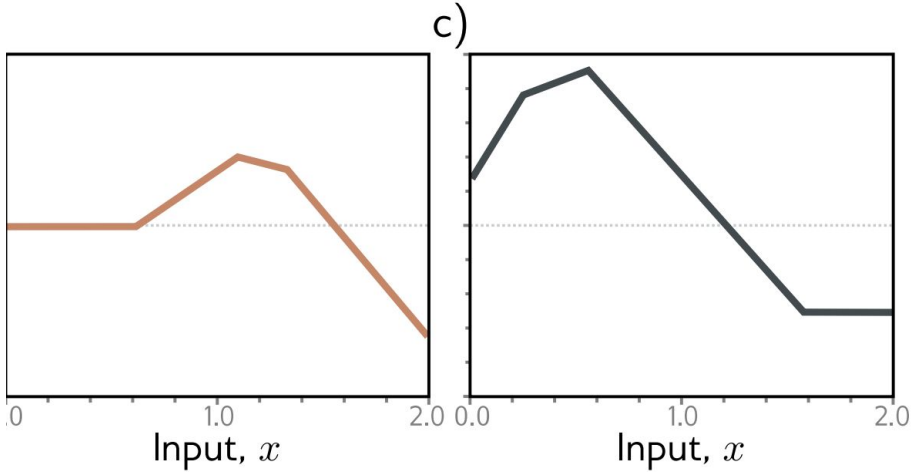
Family of functions defined by the

Compare with…

What's the main observation we can make about the "shape" of the three functions?

$$\ldots {}_1x] + \phi_2 a[\theta_{20} + \theta \ldots$$





Recapping…

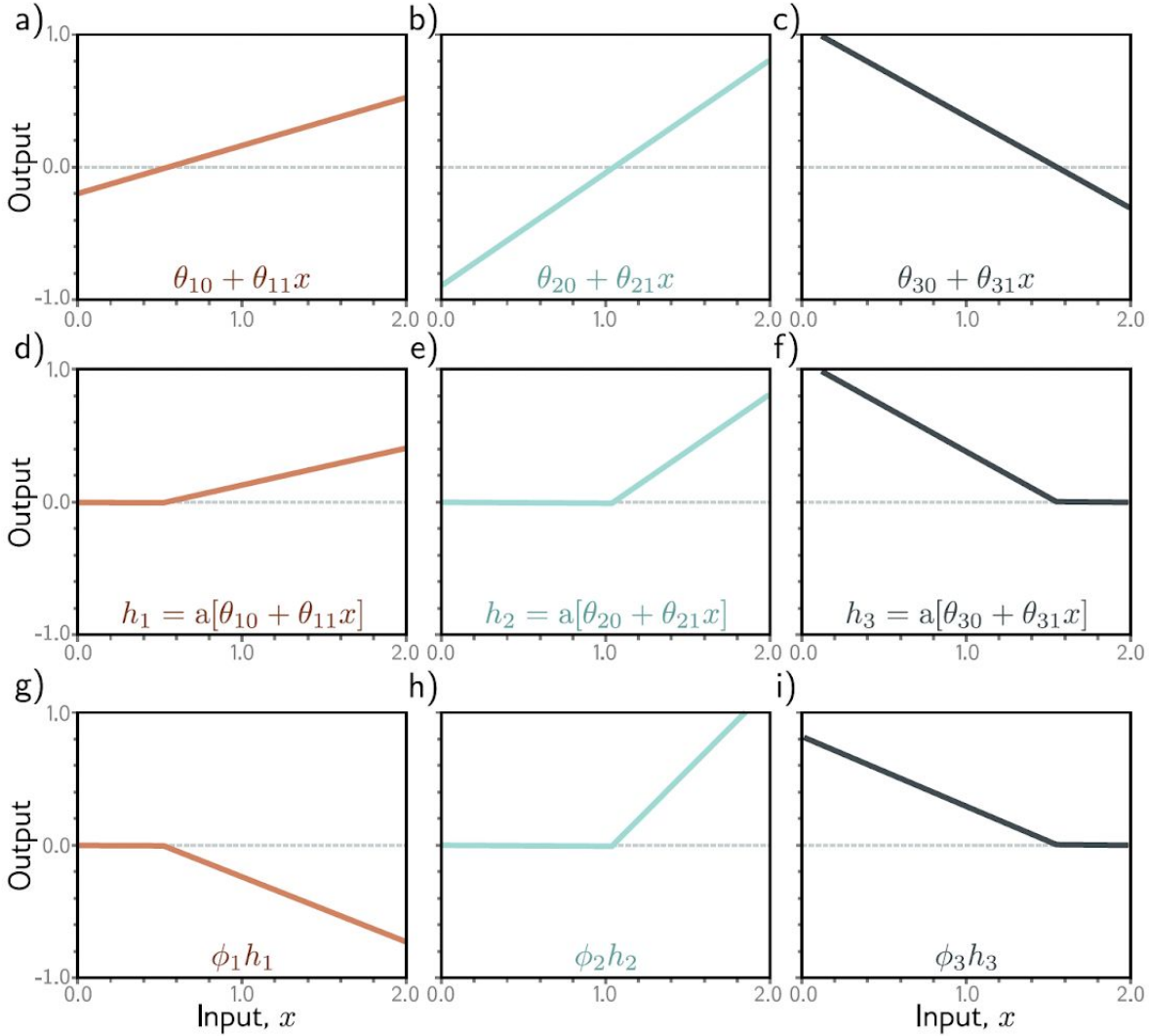
- We introduce three temp variables

$$\begin{aligned} h_1 &= a[\theta_{10} + \theta_{11}x] \\ h_2 &= a[\theta_{20} + \theta_{21}x] \\ h_3 &= a[\theta_{30} + \theta_{31}x] \end{aligned}$$

  we refer to h_1, h_2, and h_3 to as **Hidden Units**.
- By combining these hidden units with a linear function we get:

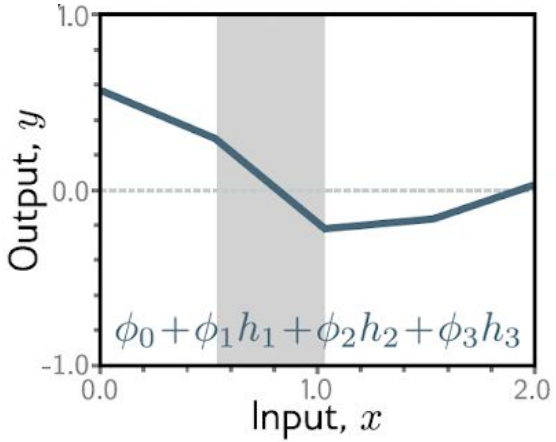
$$y = \phi_0 + \phi_1 h_1 + \phi_2 h_2 + \phi_3 h_3$$

- Each hidden unit contains a linear function $\theta_{\bullet 0} + \theta_{\bullet 1}x$ of the input, and that line is clipped by the ReLU function $a[\bullet]$ below zero. The positions where the three lines cross zero become the three “joints” in the final output. The three clipped lines are then weighted by ϕ_1, ϕ_2, and ϕ_3, respectively. Finally, the offset ϕ_0 is added, which controls the overall height of the final function.



Summing up (pun intended 👀)…

- The clipped and weighted functions are summed, and an offset ϕ_0 that controls the height is added. Each of the four linear regions corresponds to a different activation pattern in the hidden units. In the shaded region, h_2 is inactive (clipped), but h_1 and h_3 are both active.



Summing up (pun intended 👀)…

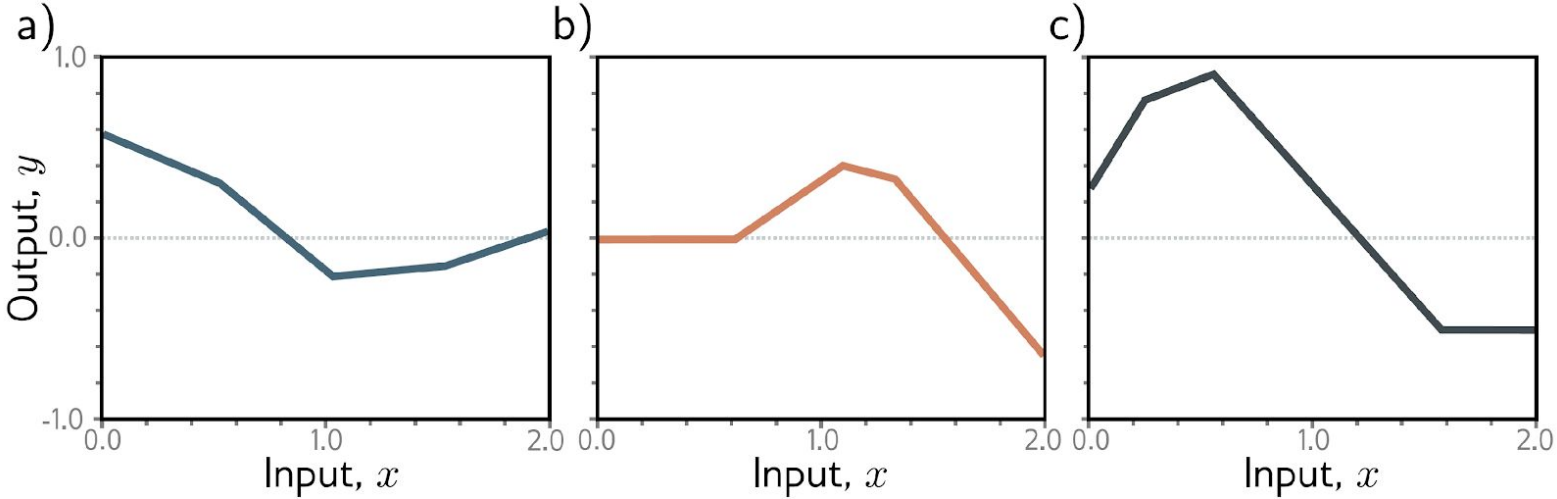
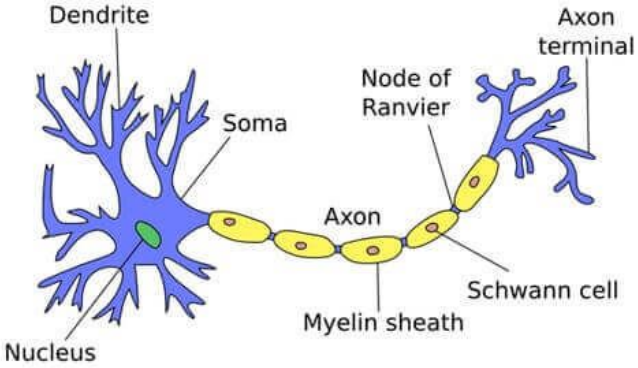
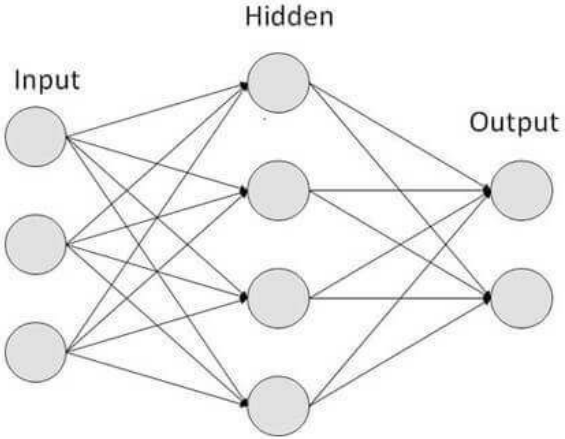


Figure 3.2 Family of functions defined by equation 3.1. a–c) Functions for three different choices of the ten parameters ϕ. In each case, the input/output relation is piecewise linear. However, the positions of the joints, the slopes of the linear regions between them, and the overall height vary.

Maybe more usually depicted like this…

Let’s stop using this…





Colab time: Shallow Networks I

https://colab.research.google.com/drive/1kxlRuMPhOy_I_y5XyMbFBXIINaB65_Id

Colab time: Shallow Networks I

01 Solution Shallow_Networks_I.ipynb

Colab time: Shallow Networks II

https://colab.research.google.com/drive/16G7zO9XE6d7Q99CCjISIz0CSnRmqjRvB

Colab time: Shallow Networks II

02 Solution_Shallow_Networks_II.ipynb

Universal Approximation Theorem

- Consider the case with D hidden units where the d^{th} hidden unit is:

$$h_d = \mathrm{a}[\theta_{d0} + \theta_{d1}x]$$

  and these are combined linearly to create the output:

$$y = \phi_0 + \sum_{d=1}^{D} \phi_d h_d$$

- The number of hidden units in a shallow network is a measure of the network capacity. With ReLU activation functions, the output of a network with D hidden units has at most D joints and so is a piecewise linear function with at most D + 1 linear regions.
- As we add more hidden units, the model can approximate more complex functions.

Universal Approximation Theorem

- With enough capacity (hidden units), a shallow network can describe any continuous 1D function defined on a compact subset of the real line to arbitrary precision.
- To see this, consider that every time we add a hidden unit, we add another linear region to the function. As these regions become more numerous, they represent smaller sections of the function, which are increasingly well approximated by a line.
- The **universal approximation theorem** proves that for any continuous function, there exists a *shallow* network that can approximate this function to any specified precision.

Universal Approximation Theorem

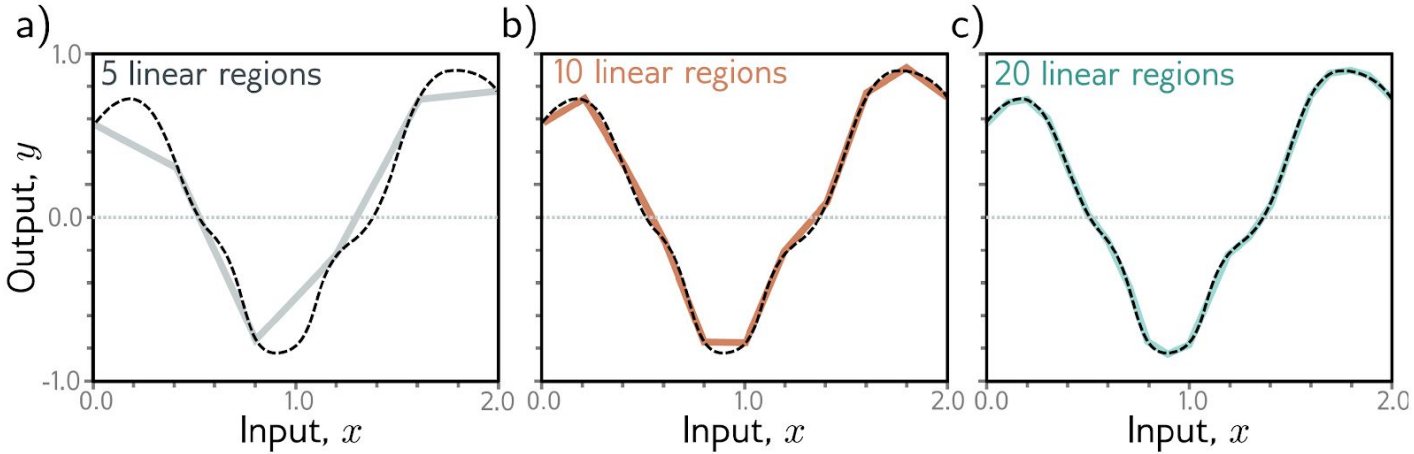
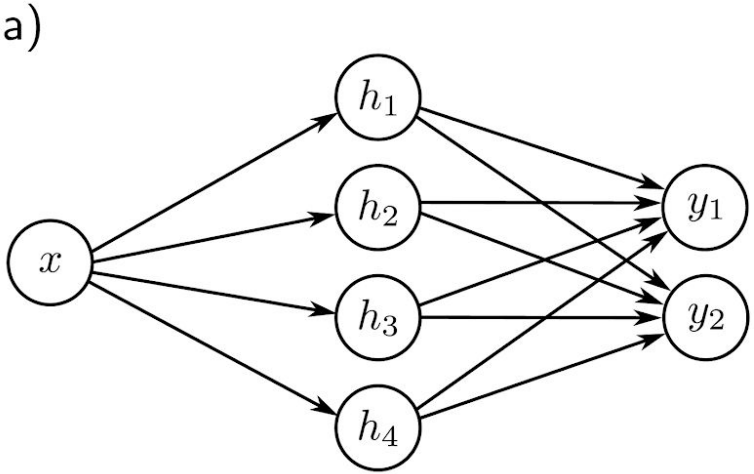
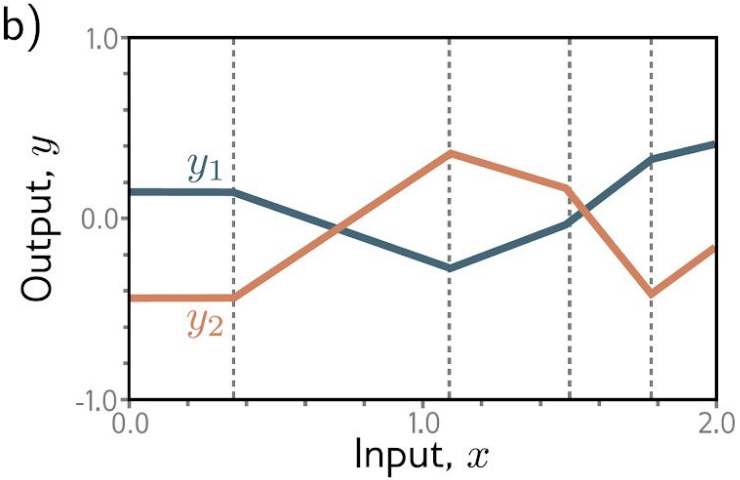


Figure 3.5 Approximation of a 1D function (dashed line) by a piecewise linear model. a–c) As the number of regions increases, the model becomes closer and closer to the continuous function. A neural network with a scalar input creates one extra linear region per hidden unit. The universal approximation theorem proves that, with enough hidden units, there exists a shallow neural network can describe any given continuous function defined on a compact subset of $\mathbb{R}^D$ to arbitrary precision.

Multivariate Outputs





$$y_1 = \phi_{10} + \phi_{11}h_1 + \phi_{12}h_2 + \phi_{13}h_3 + \phi_{14}h_4$$

$$y_2 = \phi_{20} + \phi_{21}h_1 + \phi_{22}h_2 + \phi_{23}h_3 + \phi_{24}h_4.$$

units, but the slopes and overall height may differ.

Multivariate Inputs

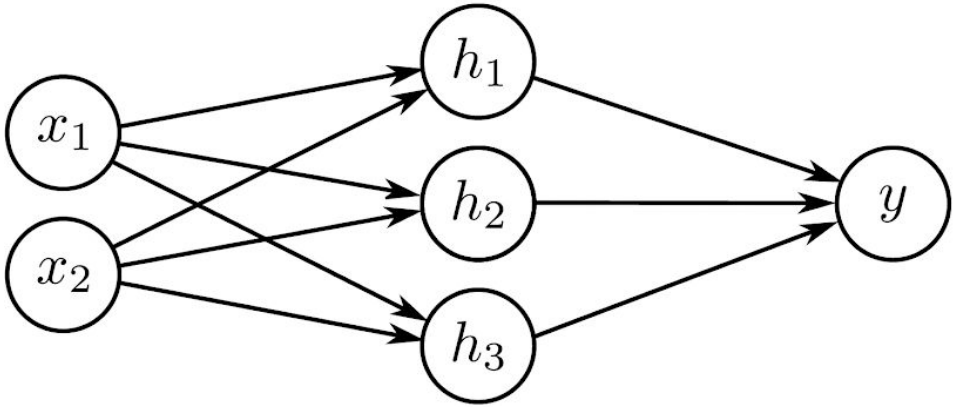
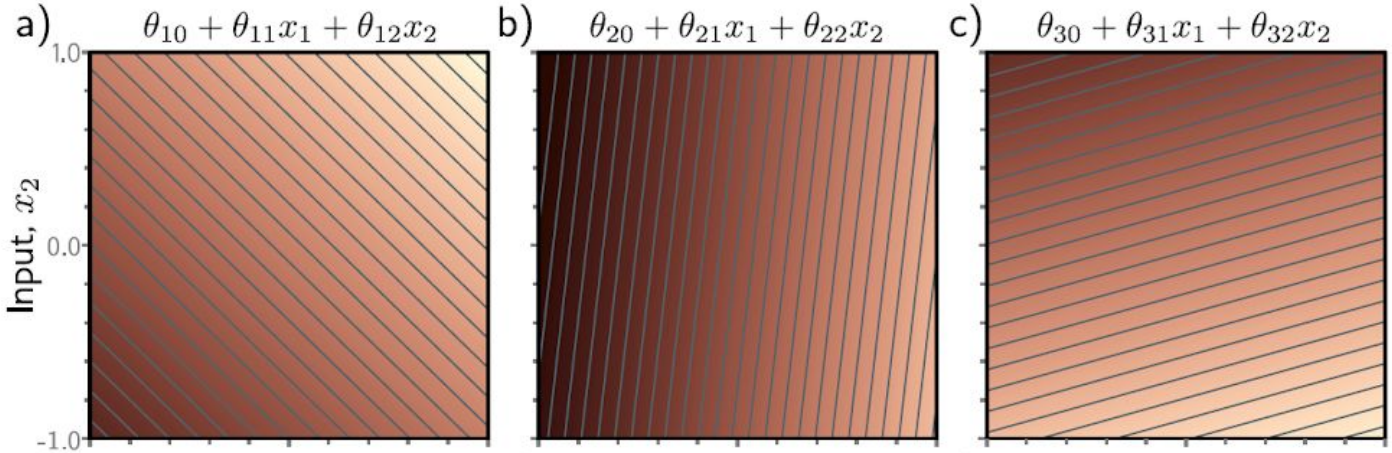


Figure 3.7 Visualization of neural network with 2D multivariate input $\mathbf{x} = [x_1, x_2]^T$ and scalar output y.

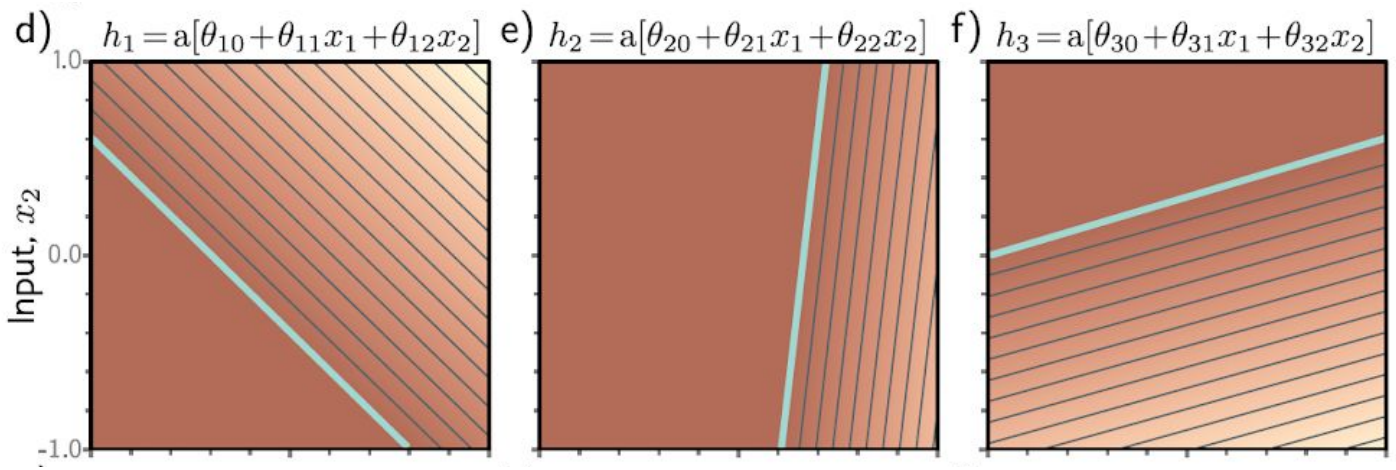
What happens when we add inputs and outputs?

- Processing in network with two inputs $\mathbf{x} = [x_1, x_2]^T$, three hidden units h_1, h_2, h_3, and one output y.
- The input to each hidden unit is a linear function of the two inputs, which corresponds to an oriented plane. Brightness indicates function output.
- For example, in panel (a), the brightness represents $\theta_{10} + \theta_{11}x_1 + \theta_{12}x_2$. Thin lines are contours.



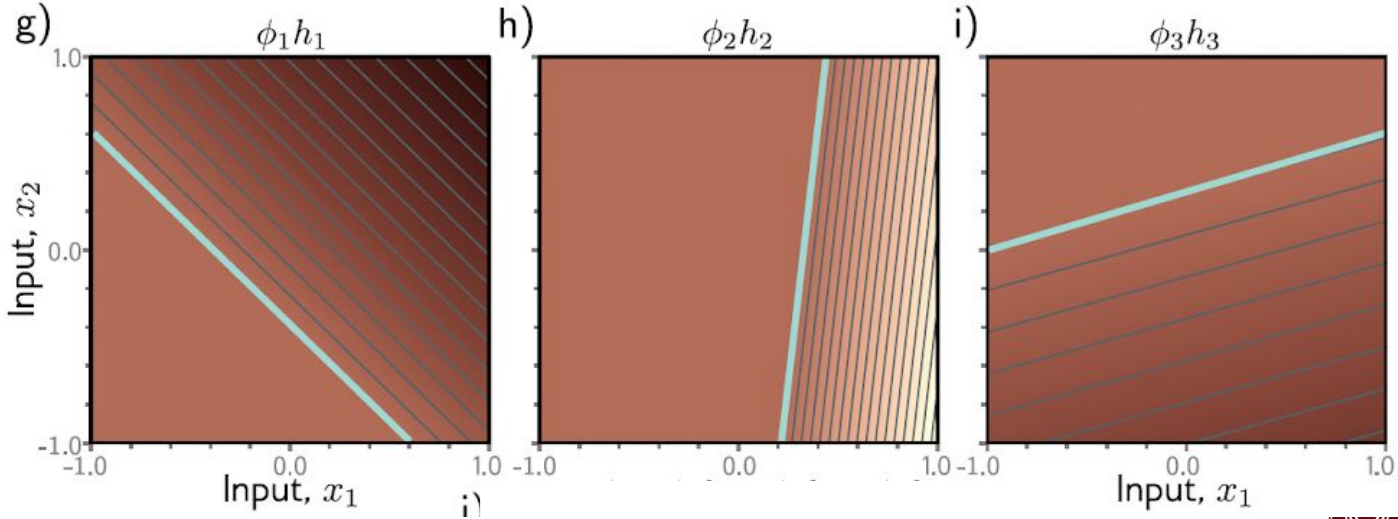
What happens when we add inputs and outputs?

- Each plane is clipped by the ReLU activation function
- Cyan lines are equivalent to “joints” in the 1-D case



What happens when we add inputs and outputs?

- The clipped planes are then weighted



Summing up (pun again intended 👀)…

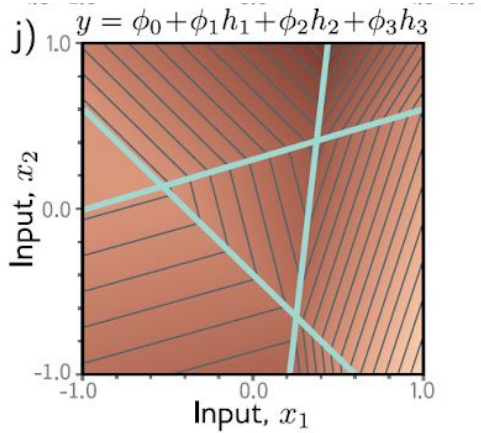


Figure 3.8 Processing in network with two inputs $\mathbf{x} = [x_1, x_2]^T$, three hidden units h_1, h_2, h_3, and one output y. a–c) The input to each hidden unit is a linear function of the two inputs, which corresponds to an oriented plane. Brightness indicates function output. For example, in panel (a), the brightness represents $\theta_{10} + \theta_{11}x_1 + \theta_{12}x_2$. Thin lines are contours. d–f) Each plane is clipped by the ReLU activation function (cyan lines are equivalent to "joints" in figures 3.3d–f). g-i) The clipped planes are then weighted, and j) summed together with an offset that determines the overall height of the surface. The result is a continuous surface made up of convex piecewise linear polygonal regions.

Hidden Units vs. Convex Piece-wise Linear Regions

- Each hidden unit defines a hyperplane that delineates the part of space where this unit is active from the Shallow network regions where it is not (cyan lines in d–f above).
- If we had the same number of hidden units as input dimensions D_i, we could align each hyperplane with one of the coordinate axes.
  - For two input dimensions, this would divide the space into four quadrants.
  - For three dimensions, this would create eight octants, and
  - for D_i dimensions, this would create 2^{D_i} orthants.
- Shallow neural networks usually have more hidden units than input dimensions, so they typically create more than 2^{D_i} linear regions.

Hidden Units vs. Convex Piece-wise Linear Regions

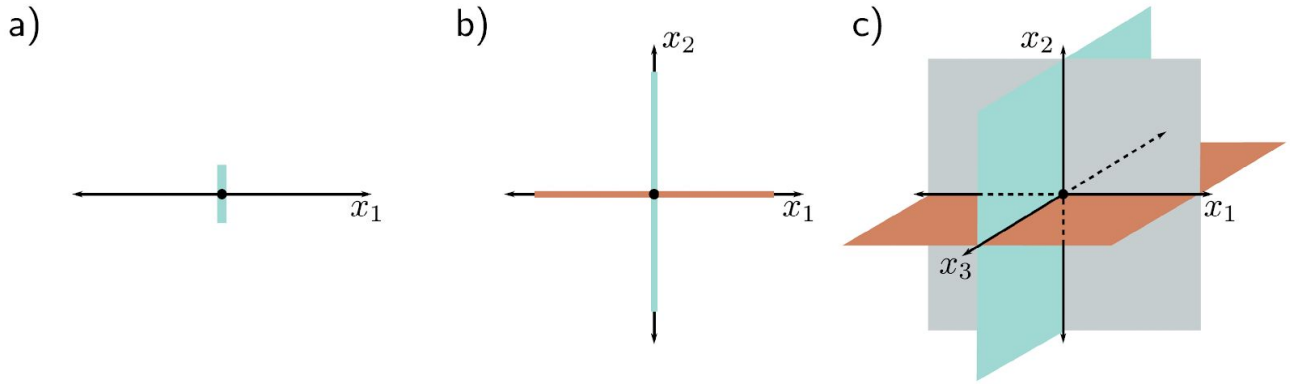


Figure 3.10 Number of linear regions vs. input dimensions. a) With a single input dimension, a model with one hidden unit creates one joint, which divides the axis into two linear regions. b) With two input dimensions, a model with two hidden units can divide the input space using two lines (here aligned with axes) to create four regions. c) With three input dimensions, a model with three hidden units can divide the input space using three planes (again aligned with axes) to create eight regions. Continuing this argument, it follows that a model with D_i input dimensions and D_i hidden units can divide the input space with D_i hyperplanes to create 2^{D_i} linear regions.

Hidden Units vs. Convex Piece-wise Linear Regions

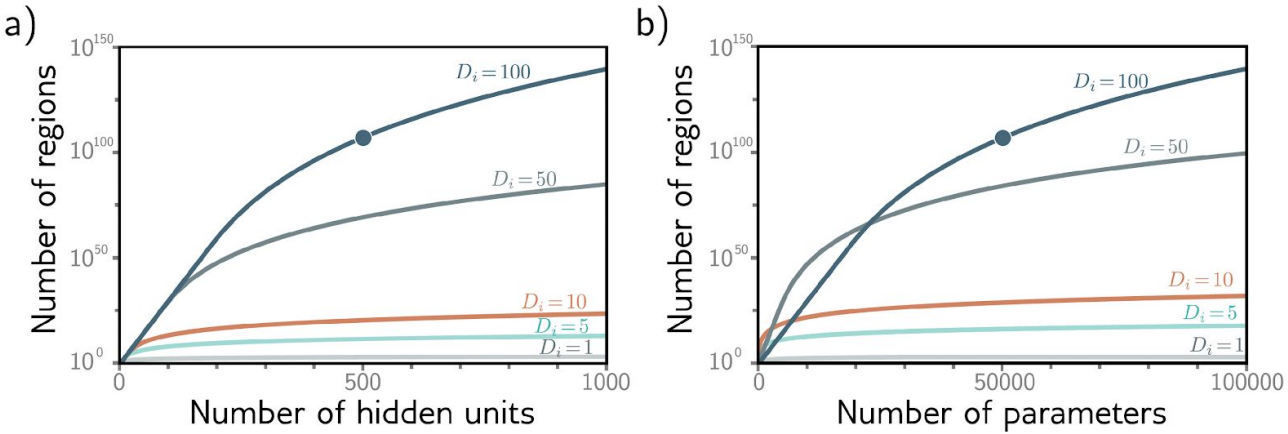


Figure 3.9 Linear regions vs. hidden units. a) Maximum possible regions as a function of the number of hidden units for five different input dimensions $D_i = \{1, 5, 10, 50, 100\}$. The number of regions increases rapidly in high dimensions; with $D = 500$ units and input size $D_i = 100$, there can be greater than 10^{107} regions (solid circle). b) The same data are plotted as a function of the number of parameters. The solid circle represents the same model as in panel (a) with $D = 500$ hidden units. This network has $51,001$ parameters and would be considered very small by modern standards.

Colab time: Number of Hidden Layers

https://colab.research.google.com/drive/1Y9_tCB73A4h5O77QQhPSG0AlJU-I3hDQ

Colab time: Number of Hidden Layers

03 Solution Shallow_Network_Regions.ipynb

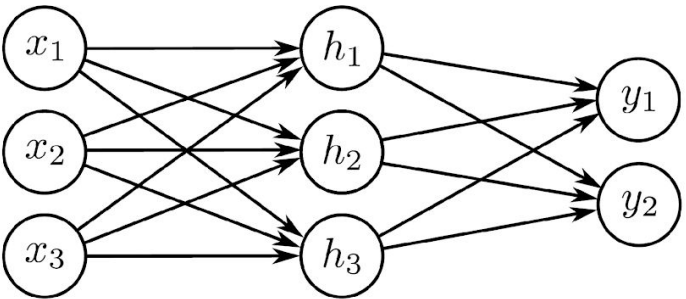
Shallow Networks: The Most General Case

- We now define a general equation for a shallow neural network **y** = f[**x**, **ϕ**]
  - **x** ∈ $\mathbb{R}^{D_i}$ to a multi-dimensional output **y** ∈ $\mathbb{R}^{D_o}$, **h** ∈ $\mathbb{R}^{D}$

$$h_d = a\left[\theta_{d0} + \sum_{i=1}^{D_i} \theta_{di} x_i\right]$$

- Combined linearly to create the output:

$$y_j = \phi_{j0} + \sum_{d=1}^{D} \phi_{jd} h_d$$



Terminology

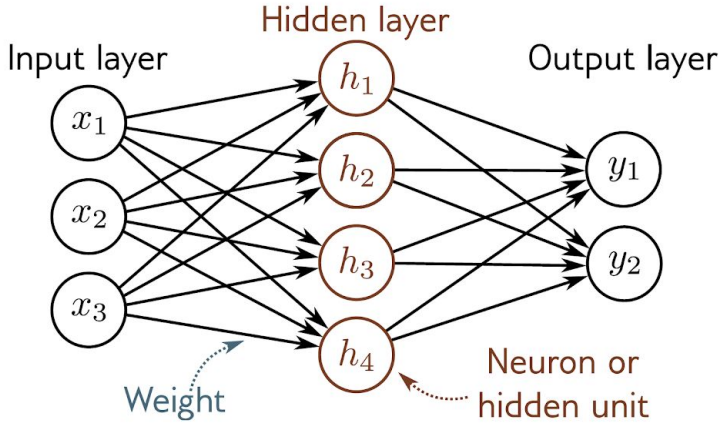


Figure 3.12 Terminology. A shallow network consists of an input layer, a hidden layer, and an output layer. Each layer is connected to the next by forward connections (arrows). For this reason, these models are referred to as feed-forward networks. When every variable in one layer connects to every variable in the next, we call this a fully connected network. Each connection represents a slope parameter in the underlying equation, and these parameters are termed weights. The variables in the hidden layer are termed neurons or hidden units. The values feeding into the hidden units are termed pre-activations, and the values at the hidden units (i.e., after the ReLU function is applied) are termed activations.

Activation Function (or Nonlinearities)

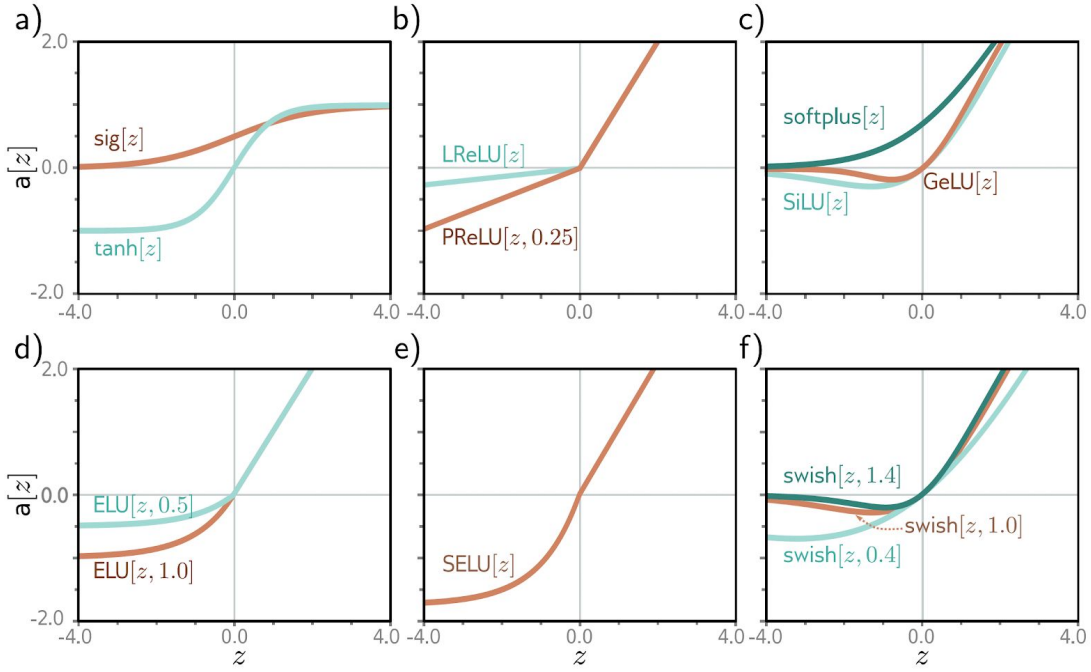


Figure 3.13 Activation functions. a) Logistic sigmoid and tanh functions. b) Leaky ReLU and parametric ReLU with parameter 0.25. c) SoftPlus, Gaussian error linear unit, and sigmoid linear unit. d) Exponential linear unit with parameters 0.5 and 1.0, e) Scaled exponential linear unit. f) Swish with parameters 0.4, 1.0, and 1.4.

Colab time: Activation Functions (or nonlinearities)

https://colab.research.google.com/drive/1QPn1q5AKXnw3eKDDK1xLBS2rl9zodgs5

Colab time: Activation Functions (or nonlinearities)

04 Solution Activation_Functions.ipynb

Fitting Models

(Chapter 6)

SAPIENZA
UNIVERSITÀ DI ROMA

Recapping

- The loss we saw in the previous class depends on the network parameters, and here we show how to find the parameter values that minimize this loss.
  - This is known as learning the network's parameters or simply as training or fitting the model.
- The process is to choose initial parameter values and then iterate the following two steps:
  - compute the derivatives (gradients) of the loss with respect to the parameters
  - adjust the parameters based on the gradients to decrease the loss.
- After many iterations, we ***hope*** to reach the overall minimum of the loss function.
- We will show algorithms that adjust the parameters to decrease the loss.

Gradient Descent

- Starts with initial parameters $\boldsymbol{\phi} = [\phi_0, \phi_1, \ldots, \phi_N]^T$ and iterates two steps:

Step 1. Compute the derivatives of the loss with respect to the parameters:

$$\mathrm{L}[\boldsymbol{\phi}] = \sum_{i=1}^{I} \ell_i \qquad \frac{\partial L}{\partial \boldsymbol{\phi}} = \begin{bmatrix} \frac{\partial L}{\partial \phi_0} \\ \frac{\partial L}{\partial \phi_1} \\ \vdots \\ \frac{\partial L}{\partial \phi_N} \end{bmatrix}.$$

Step 2. Update the parameters according to the rule:

$$\boldsymbol{\phi} \longleftarrow \boldsymbol{\phi} - \alpha \frac{\partial L}{\partial \boldsymbol{\phi}},$$

Learning Rate

where the positive scalar α determines the magnitude of the change.

Gradient Descent: Linear Regression Example

- The model $y = f[x,\boldsymbol{\phi}]$ maps a scalar input x to a scalar output y and has parameters $\boldsymbol{\phi} = [\phi_0, \phi_1]^T$, which represent the y-intercept and the slope:

$$\begin{aligned} y &= f[x, \boldsymbol{\phi}] \\ &= \phi_0 + \phi_1 x \end{aligned}$$

- Given a dataset $\{x_i, y_i\}$ containing I input/outp ast squares loss function:

$\ell_i = (\phi_0 + \phi_1 x_i - y_i)^2$ is the individual contribution to the loss from the i^{th} training example

$$\begin{aligned} L[\boldsymbol{\phi}] &= \sum_{i=1}^{I} \ell_i = \sum_{i=1}^{I} (f[x_i, \boldsymbol{\phi}] - y_i)^2 \\ &= \sum_{i=1}^{I} (\phi_0 + \phi_1 x_i - y_i)^2 \end{aligned}$$

Gradient Descent: Linear Regression Example

- The derivative of the loss function with respect to the parameters can be decomposed into the sum of the derivatives of the individual contributions:

$$\frac{\partial L}{\partial \boldsymbol{\phi}} = \frac{\partial}{\partial \boldsymbol{\phi}} \sum_{i=1}^{I} \ell_i = \sum_{i=1}^{I} \frac{\partial \ell_i}{\partial \boldsymbol{\phi}}$$

- where these are given by (take 5 mins to compute it):

$$\ell_i = (\phi_0 + \phi_1 x_i - y_i)^2$$

$$\frac{\partial \ell_i}{\partial \boldsymbol{\phi}} = \begin{bmatrix} \frac{\partial \ell_i}{\partial \phi_0} \\ \frac{\partial \ell_i}{\partial \phi_1} \end{bmatrix} = \begin{bmatrix} 2(\phi_0 + \phi_1 x_i - y_i) \\ 2x_i(\phi_0 + \phi_1 x_i - y_i) \end{bmatrix}$$

Gradient Descent: Linear Regression Example

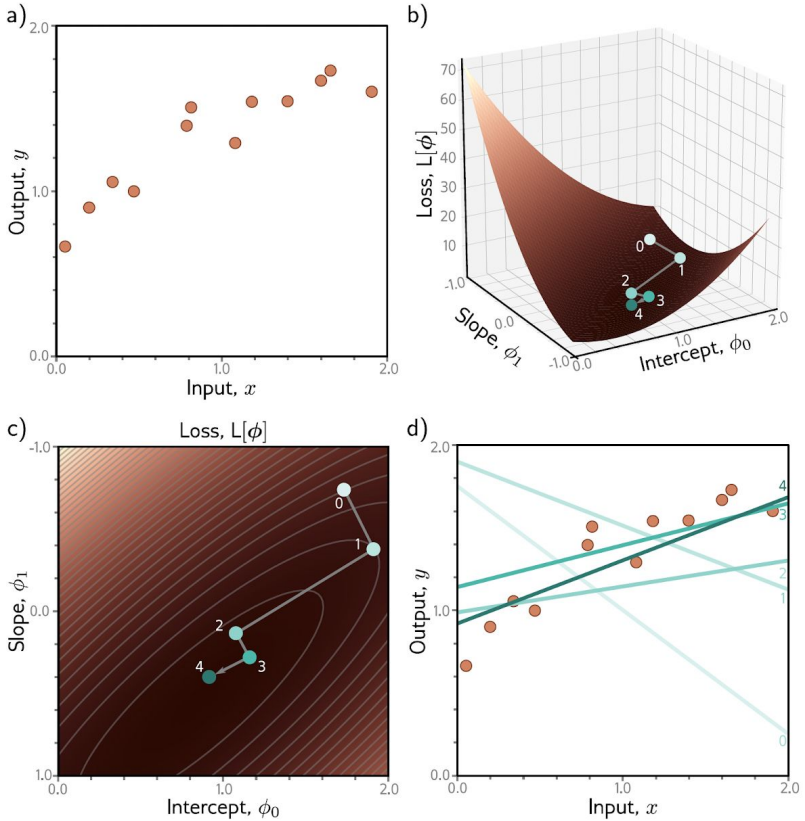


Figure 6.1 Gradient descent for the linear regression model. a) Training set of $I = 12$ input/output pairs $\{x_i, y_i\}$. b) Loss function showing iterations of gradient descent. We start at point 0 and move in the steepest downhill direction until we can improve no further to arrive at point 1. We then repeat this procedure. We measure the gradient at point 1 and move downhill to point 2 and so on. c) This can be visualized better as a heatmap, where the brightness represents the loss. After only four iterations, we are already close to the minimum. d) The model with the parameters at point 0 (lightest line) describes the data very badly, but each successive iteration improves the fit. The model with the parameters at point 4 (darkest line) is already a reasonable description of the training data.

Is that it?!?!

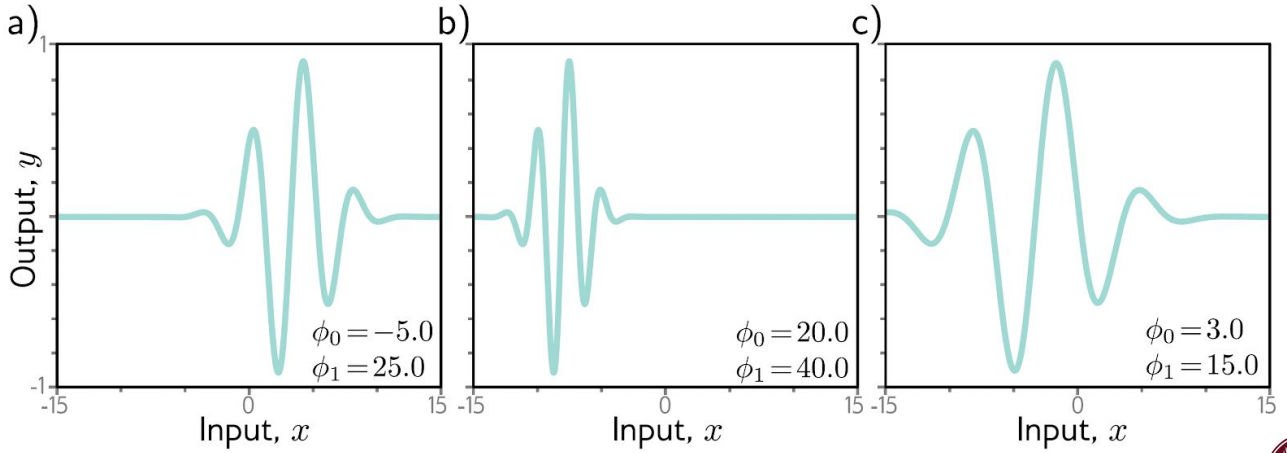
- For linear models, e.g. linear regression, the situation is usually good:
  - Convex functions always have a single optimum

Gabor Model

- Consider the following model (Gabor):

$$\mathrm{f}[x, \boldsymbol{\phi}] = \sin[\phi_0 + 0.06 \cdot \phi_1 x] \cdot \exp\left(-\frac{(\phi_0 + 0.06 \cdot \phi_1 x)^2}{32.0}\right)$$

- This Gabor model maps scalar input x to scalar output y and consists of a sinusoidal component (creating an oscillatory function) multiplied by a negative exponential component (causing the amplitude to decrease as we move from the center).



Running Example for the Gabor Model

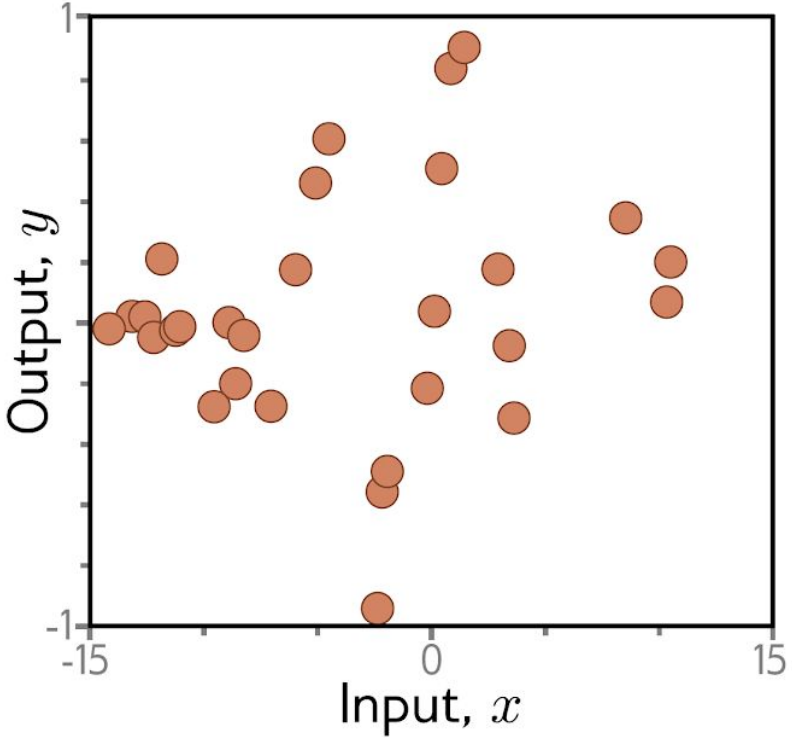


Figure 6.3 Training data for fitting the Gabor model. The training dataset contains 28 input/output examples $\{x_i, y_i\}$. These were created by uniformly sampling $x_i \in [-15, 15]$, passing the samples through a Gabor model with parameters $\boldsymbol{\phi} = [0.0, 16.6]^T$, and adding normally distributed noise.

Loss Landscape of MSE Applied to the Gabor Model

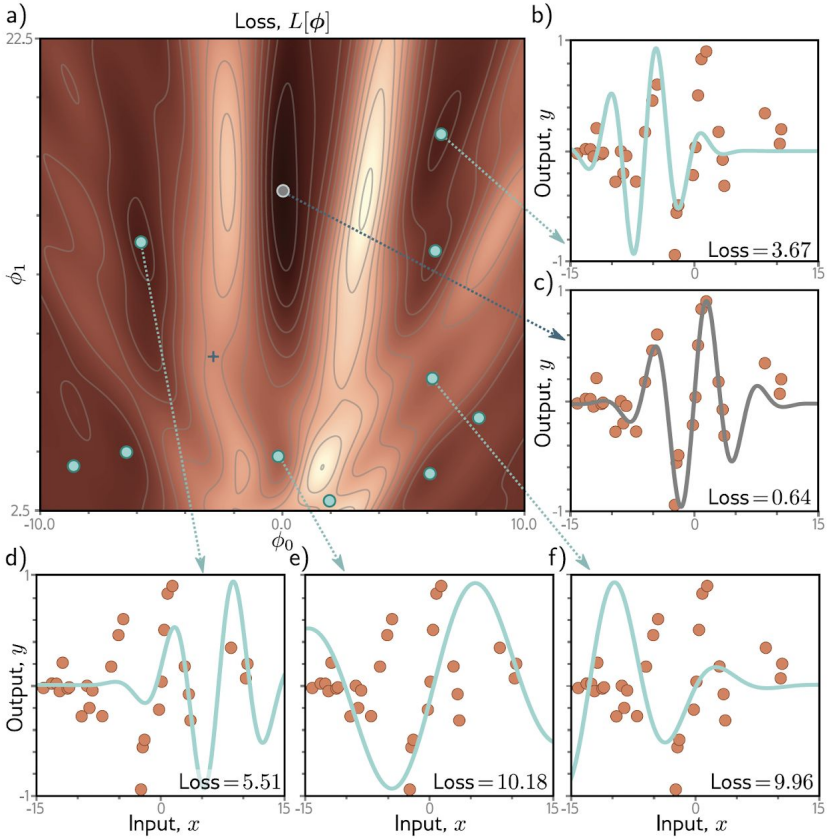
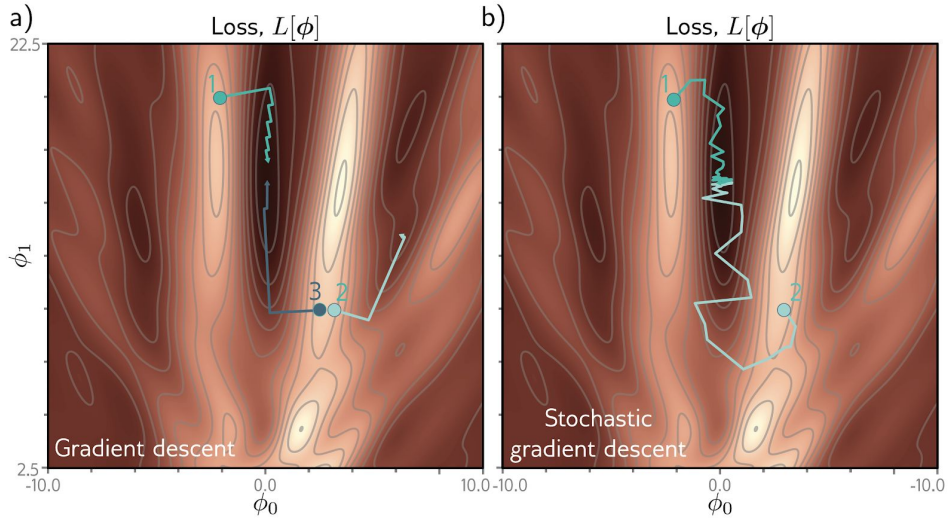


Figure 6.4 Loss function for the Gabor model. a) The loss function is non-convex, with multiple local minima (cyan circles) in addition to the global minimum (gray circle). It also contains saddle points where the gradient is locally zero, but the function increases in one direction and decreases in the other. The blue cross is an example of a saddle point; the function decreases as we move horizontally in either direction but increases as we move vertically. b–f) Models associated with the different minima. In each case, there is no small change that decreases the loss. Panel (c) shows the global minimum, which has a loss of 0.64.

Stochastic Gradient Descent

- Starts with initial parameters $\boldsymbol{\phi} = [\phi_0, \phi_1, \ldots, \phi_N]^T$ and iterates the same two steps of Gradient Descent, except…
- … The gradient is computed for each element (x_i, y_i) in the dataset: $\frac{\partial \ell_i}{\partial \boldsymbol{\phi}}$



SAPIENZA
UNIVERSITÀ DI ROMA

Batches and Epochs

- At each iteration, the algorithm chooses a random subset of the training data and computes the gradient from these examples alone.
  - This subset is known as a **minibatch** or **batch** for short.
- The update rule for the model parameters ϕ_t at iteration t is hence:

$$\phi_{t+1} \leftarrow \phi_t - \alpha \cdot \sum_{i \in \mathcal{B}_t} \frac{\partial \ell_i[\phi_t]}{\partial \phi}$$

Minibatch (or Batch)

- The batches are usually drawn from the dataset without replacement.
- The algorithm works through the training examples until it has used all the data, at which point it starts sampling from the full training dataset again.
- A *single pass through the entire training dataset is referred to as an epoch*.

SGD with Adam

- Gradient descent with a fixed step size has the following undesirable property: it makes large adjustments to parameters associated with large gradients (where perhaps we should be more cautious) and small adjustments to parameters associated with small gradients (where perhaps we should explore further).
- When the gradient of the loss surface is much steeper in one direction than another, it is difficult to choose a learning rate that (i) makes good progress in both directions and (ii) is stable

SGD with Adam

- A straightforward approach is to normalize the gradients so that we move a fixed distance (governed by the learning rate) in each direction. To do this, we first measure the gradient m_{t+1} and the pointwise squared gradient v_{t+1}:

$$\begin{aligned} \mathbf{m}_{t+1} &\leftarrow \frac{\partial L[\boldsymbol{\phi}_t]}{\partial \boldsymbol{\phi}} \\ \mathbf{v}_{t+1} &\leftarrow \left(\frac{\partial L[\boldsymbol{\phi}_t]}{\partial \boldsymbol{\phi}}\right)^2 \end{aligned}$$

- Then we apply the update rule:

$$\boldsymbol{\phi}_{t+1} \leftarrow \boldsymbol{\phi}_t - \alpha \cdot \frac{\mathbf{m}_{t+1}}{\sqrt{\mathbf{v}_{t+1}} + \epsilon}$$

SGD with Adam

- Adam extends this idea by refining m_{t+1} and v_{t+1} calculation

$$\begin{aligned}\mathbf{m}_{t+1} &\leftarrow \beta \cdot \mathbf{m}_t + (1-\beta)\frac{\partial L[\boldsymbol{\phi}_t]}{\partial \boldsymbol{\phi}} \\ \mathbf{v}_{t+1} &\leftarrow \gamma \cdot \mathbf{v}_t + (1-\gamma)\left(\frac{\partial L[\boldsymbol{\phi}_t]}{\partial \boldsymbol{\phi}}\right)^2\end{aligned}$$

  where β and γ are the momentum coefficients for the two statistics.
- Using momentum is equivalent to taking a weighted average over the history of each of these statistics. At the start of the procedure, all the previous measurements are effectively zero, resulting in unrealistically small estimates. Consequently, we modify these statistics using the rule:

$$\begin{aligned}\tilde{\mathbf{m}}_{t+1} &\leftarrow \frac{\mathbf{m}_{t+1}}{1-\beta^{t+1}} \\ \tilde{\mathbf{v}}_{t+1} &\leftarrow \frac{\mathbf{v}_{t+1}}{1-\gamma^{t+1}}.\end{aligned}$$

SGD with Adam

- Finally, we update the parameters as before, but with the modified terms:

$$\boldsymbol{\phi}_{t+1} \leftarrow \boldsymbol{\phi}_t - \alpha \cdot \frac{\tilde{\mathbf{m}}_{t+1}}{\sqrt{\tilde{\mathbf{v}}_{t+1}} + \epsilon}.$$

- In stochastic settings

$$\mathbf{m}_{t+1} \leftarrow \beta \cdot \mathbf{m}_t + (1-\beta) \sum_{i \in \mathcal{B}_t} \frac{\partial \ell_i[\boldsymbol{\phi}_t]}{\partial \boldsymbol{\phi}}$$

$$\mathbf{v}_{t+1} \leftarrow \gamma \cdot \mathbf{v}_t + (1-\gamma) \sum_{i \in \mathcal{B}_t} \left(\frac{\partial \ell_i[\boldsymbol{\phi}_t]}{\partial \boldsymbol{\phi}} \right)^2$$

SGD with Adam

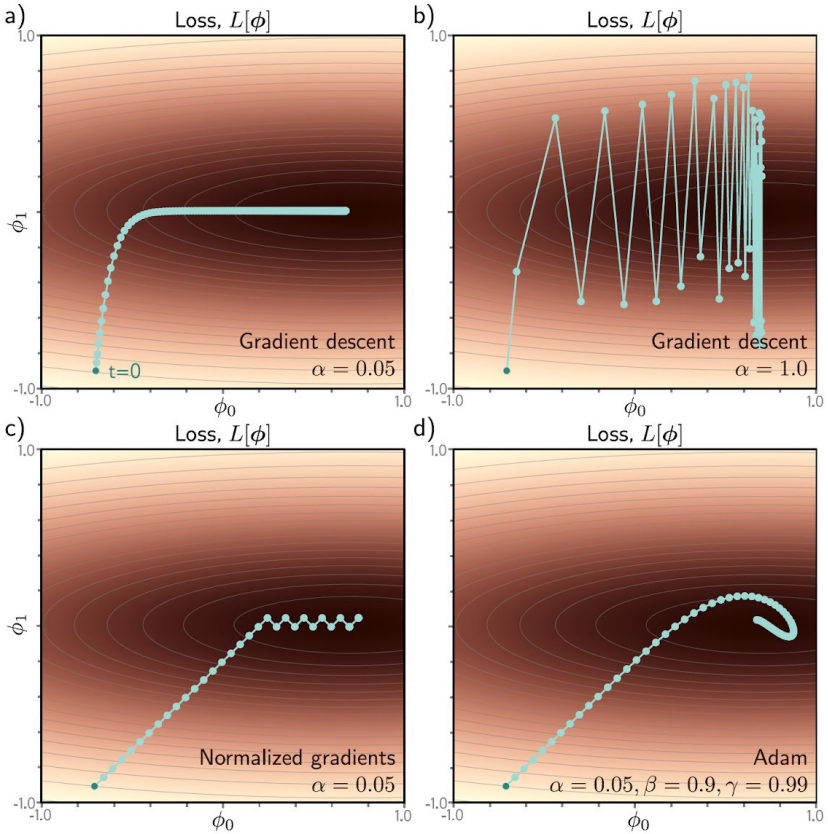


Figure 6.9 Adaptive moment estimation (Adam). a) This loss function changes quickly in the vertical direction but slowly in the horizontal direction. If we run full-batch gradient descent with a learning rate that makes good progress in the vertical direction, then the algorithm takes a long time to reach the final horizontal position. b) If the learning rate is chosen so that the algorithm makes good progress in the horizontal direction, it overshoots in the vertical direction and becomes unstable. c) A straightforward approach is to move a fixed distance along each axis at each step so that we move downhill in both directions. This is accomplished by normalizing the gradient magnitude and retaining only the sign. However, this does not usually converge to the exact minimum but instead oscillates back and forth around it (here between the last two points). d) The Adam algorithm uses momentum in both the estimated gradient and the normalization term, which creates a smoother path.

Hyperparameters of the Training Algorithms

- The choices of learning algorithm, batch size, learning rate schedule, and momentum coefficients are all considered hyperparameters of the training algorithm
    - these directly affect the final model performance but are distinct from the model parameters.
- Choosing these can be more art than science, and it's common to train many models with different hyperparameters and choose the best one.
- This is known as *hyperparameter search*.

Fondamenti di IA

End of Lecture

04 - Ensembles and Neural Networks

SAPIENZA
UNIVERSITÀ DI ROMA

Fabrizio Silvestri