Natural Language Processing
Linear Models and Neural Networks

Felipe Bravo-Marquez

July 23, 2019

Supervised Learning

- The essence of supervised machine learning is the creation of mechanisms that can look at examples and produce generalizations. [Goldberg, 2017]
- We design an algorithm whose input is a set of labeled examples, and its output is a function (or a program) that receives an instance and produces the desired label.
- Example: if the task is to distinguish from spam and not-spam email, the labeled examples are emails labeled as spam and emails labeled as not-spam.
- It is expected that the resulting function will produce correct label predictions also for instances it has not seen during training.
- This approach differs from designing an algorithm to perform the task (e.g., manually designed rule-based systems).

Parameterized Functions

- Searching over the set of all possible functions is a very hard (and rather ill-defined) problem. [Goldberg, 2017]
- We often restrict ourselves to search over specific families of functions.
- Example: the space of all linear functions with d_{in} inputs and d_{out} outputs,
- Such families of functions are called **hypothesis classes**.
- By restricting ourselves to a specific hypothesis class, we are injecting the learner with **inductive bias**.
- Inductive bias: a set of assumptions about the form of the desired solution.
- Some hypothesis classes facilitate efficient procedures for searching for the solution. [Goldberg, 2017]

Linear Models

- One common hypothesis class is that of high-dimensional linear function:

$$f(\vec{x}) = \vec{x} \cdot W + \vec{b}$$
$$\vec{x} \in \mathcal{R}^{d_{in}} \quad W \in \mathcal{R}^{d_{in} \times d_{out}} \quad \vec{b} \in \mathcal{R}^{d_{out}} \tag{1}$$

- The vector $\vec{x}$ is the input to the function.
- The matrix W and the vector $\vec{b}$ are the **parameters**.
- The goal of the learner is to set the values of the parameters W and $\vec{b}$ such that the function behaves as intended on a collection of input values $\vec{x}_{1:k} = \vec{x}_1, \ldots, \vec{x}_k$ and the corresponding desired outputs $\vec{y}_{1:k} = \vec{y}_1, \ldots, \vec{y}_k$
- The task of searching over the space of functions is thus reduced to one of searching over the space of parameters. [Goldberg, 2017]

Example: Language Detection

- Consider the task of distinguishing documents written in English from documents written in German.
- This is a binary classifcation problem

$$f(\vec{x}) = \vec{x} \cdot \vec{w} + b \tag{2}$$

  $d_{out} = 1$, $\vec{w}$ is a vector, and b is a scalar.
- The range of the linear function in is $[-\infty, \infty]$.
- In order to use it for binary classification, it is common to pass the output of $f(x)$ through the *sign* function, mapping negative values to -1 (the negative class) and non-negative values to +1 (the positive class).

Example: Language Detection

- Letter frequencies make for quite good predictors (features) for this task.
- Even more informative are counts of letter bigrams, i.e., pairs of consecutive letters.
- One may think that words will also be good predictors i.e., using a bag of word representation of documents.
- Letters, or letter-bigrams are far more robust.
- We are likely to encounter a new document without any of the words we observed in the training set.
- While a document without any of the distinctive letter-bigrams is significantly less likely. [Goldberg, 2017]

Example: Language Detection

- We assume we have an alphabet of 28 letters (a–z, space, and a special symbol for all other characters including digits, punctuations, etc.)
- Documents are represented as 28×28 dimensional vectors $\vec{x} \in \mathcal{R}^{784}$.
- Each entry $\vec{x}_{[i]}$ represents a count of a particular letter combination in the document, normalized by the document's length.
- For example, denoting by $\vec{x}_{ab}$ the entry of $\vec{x}$ corresponding to the letter bigram *ab*:

$$x_{ab} = \frac{\#ab}{|D|} \tag{3}$$

where $\#ab$ is the number of times the bigram *ab* appears in the document, and $|D|$ is the total number of bigrams in the document (the document's length).

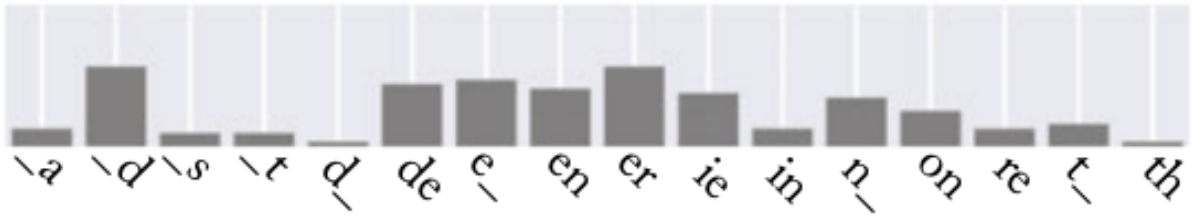
Example: Language Detection

Character-bigram histograms for documents in English (left, blue) and German(right,green). Underscores denote spaces. Only the top frequent character-bigrams are showed.

[0]Source:[Goldberg, 2017]

Example: Language Detection

- Previous figure showed clear patterns in the data, and, given a new item, such as:



- We could probably tell that it is more similar to the German group than to the English one (observe the frequency of “th” and “ie”).
- We can’t use a single definite rule such as “if it has th its English” or “if it has ie its German”.
- While German texts have considerably less “th” than English, the “th” may and does occur in German texts, and similarly the “ie” combination does occur in English.

Example: Language Detection

- The decision requires weighting different factors relative to each other.
- We can formalize the problem in a machine-learning setup using a linear model:

$$\begin{aligned} \hat{y} &= sign(f(\vec{x})) = sign(\vec{x} \cdot \vec{w} + b) \\ &= sign(\vec{x}_{aa} \times \vec{w}_{aa} + \vec{x}_{ab} \times \vec{w}_{ab} + \vec{x}_{ac} \times \vec{w}_{ac} \cdots + b) \end{aligned} \tag{4}$$

- A document will be considered English if $f(\vec{x}) \geq 0$ and as German otherwise.

Intuition

1. Learning should assign large positive values to $\vec{w}$ entries associated with letter pairs that are much more common in English than in German (i.e., “th”).
2. It should also assign negative values to letter pairs that are much more common in German than in English (“ie”, “en”).
3. Finally, it should assign values around zero to letter pairs that are either common or rare in both languages.

Log-linear Binary classifcation

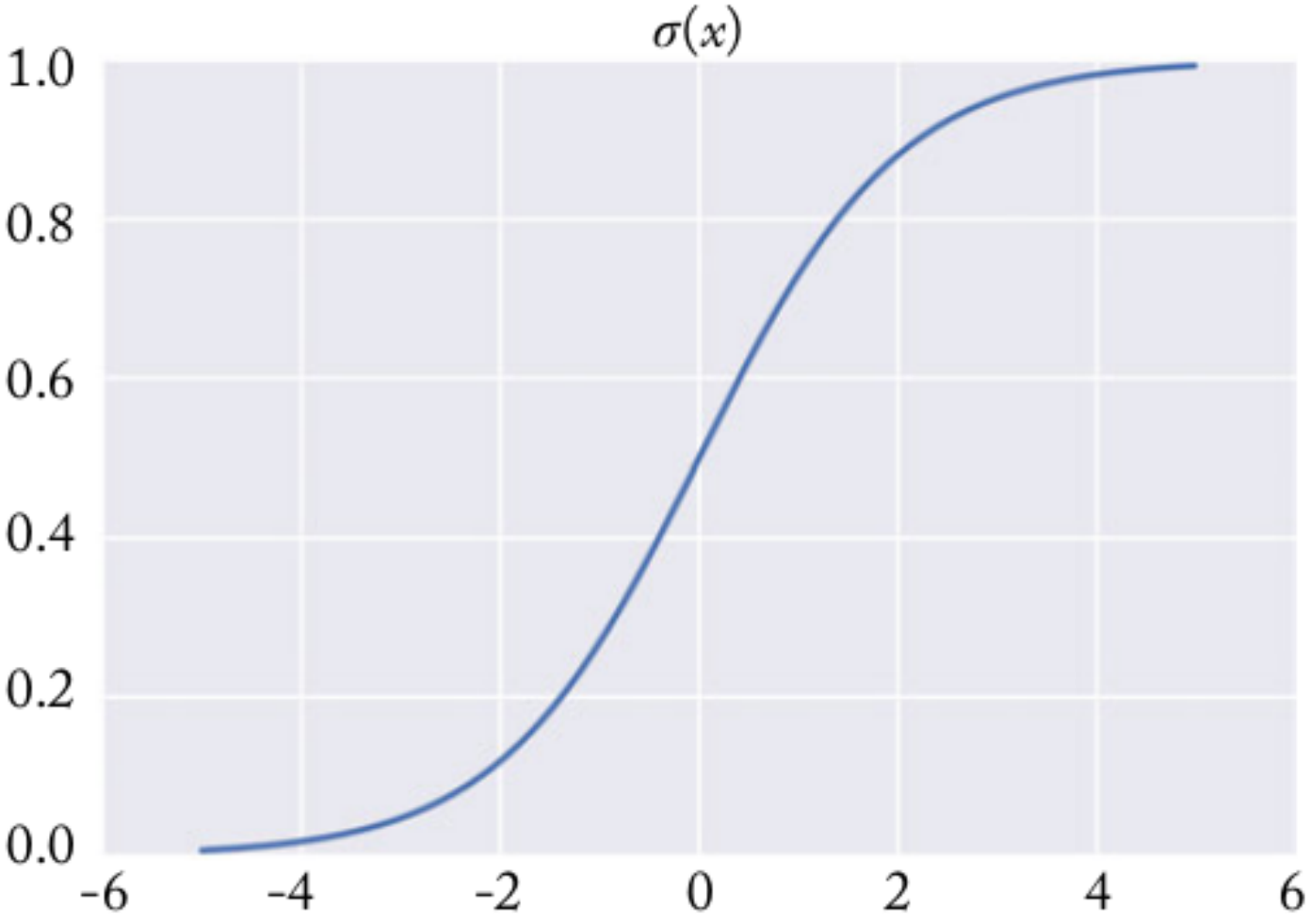
- The output $f(\vec{x})$ is in the range $[-\infty, \infty]$, and we map it to one of two classes $\{-1, +1\}$ using the *sign* function.
- This is a good fit if all we care about is the assigned class.
- We may be interested also in the confidence of the decision, or the probability that the classifier assigns to the class.
- An alternative that facilitates this is to map instead to the range $[0, 1]$, by pushing the output through a squashing function such as the sigmoid $\sigma(x)$:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{5}$$

resulting in:

$$\hat{y} = \sigma(f(\vec{x})) = \frac{1}{1 + e^{-\vec{x} \cdot \vec{w} + b}} \tag{6}$$

The Sigmoid function



The Sigmoid function

- The sigmoid function is is monotonically increasing, and maps values to the range $[0, 1]$, with 0 being mapped to $\frac{1}{2}$.
- When used with a suitable loss function (discussed later) the binary predictions made through the log-linear model can be interpreted as class membership probability estimates:

$$\sigma(f(\vec{x})) = P(\hat{y} = 1|\vec{x}) \quad \text{of } \vec{x} \text{ belonging to the positive class.} \tag{7}$$

- We also get $P(\hat{y} = 0|\vec{x}) = 1 - P(\hat{y} = 1|\vec{x}) = 1 - \sigma(f(\vec{x}))$
- The closer the value is to 0 or 1 the more certain the model is in its class membership prediction, with the value of 0.5 indicating model uncertainty.

Multi-class Classification

- Most classification problems are of a multi-class nature: examples are assigned to one of k different classes.
- Example: we are given a document and asked to classify it into one of six possible languages: English, French, German, Italian, Spanish, Other.
- Possible solution: consider six weight vectors $\vec{w}_{EN}, \vec{w}_{FR}, \ldots$ and biases (one for each language).
- Predict the language resulting in the highest score:

$$\hat{y} = f(\vec{x}) = \text{argmax}_{L \in \{EN, FR, GR, IT, SP, O\}} \quad \vec{x} \cdot \vec{w}_L + b_L \tag{8}$$

Multi-class Classification

- The six sets of parameters $\vec{w}_L \in \mathcal{R}^{784}$ and b_L can be arranged as a matrix $W \in \mathcal{R}^{784 \times 6}$ and vector $\vec{b} \in \mathcal{R}^6$, and the equation re-written as:

$$\begin{aligned} \vec{\hat{y}} = f(\vec{x}) = &\quad \vec{x} \cdot W + \vec{b} \\ \text{prediction} = \hat{y} = &\quad \text{argmax}_i \, \vec{\hat{y}}_{[i]} \end{aligned} \tag{9}$$

- Here $\vec{\hat{y}} \in \mathcal{R}^6$ is a vector of the scores assigned by the model to each language, and we again determine the predicted language by taking the argmax over the entries of $\vec{\hat{y}}$.

Representations

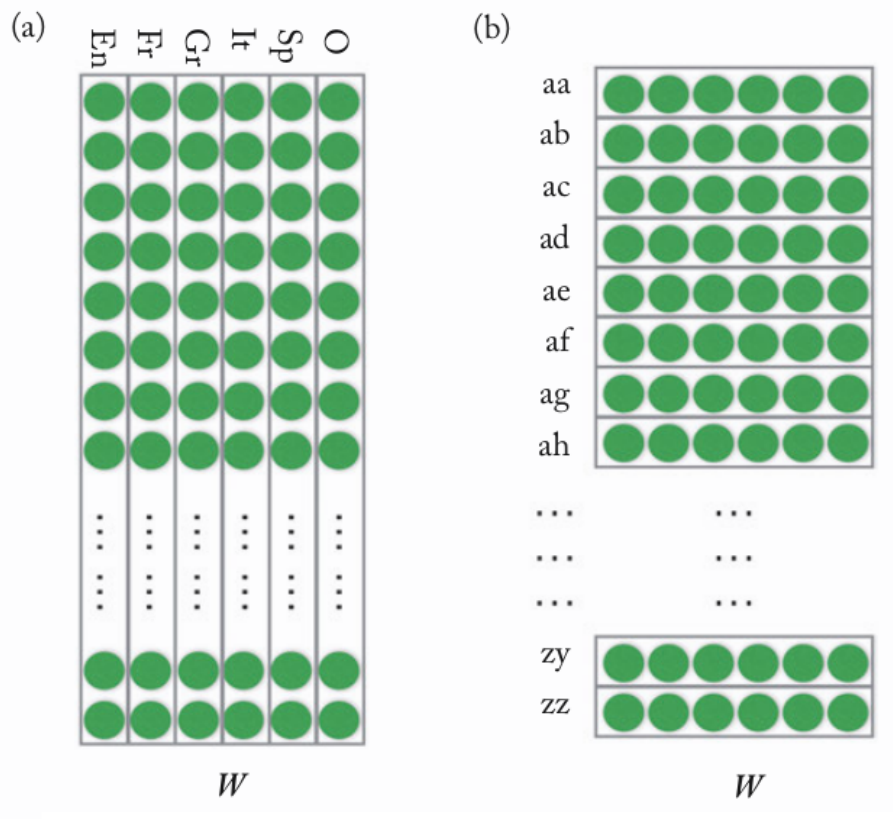
- Consider the vector $\vec{\hat{y}}$ resulting from applying a trained model to a document.
- The vector can be considered as a representation of the document.
- It captures the properties of the document that are important to us: the scores of the different languages.
- The representation $\vec{\hat{y}}$ contains strictly more information than the prediction $\text{argmax}_i \, \vec{\hat{y}}_{[i]}$.
- Example: $\vec{\hat{y}}$ can be used to distinguish documents in which the main language is German, but which also contain a sizeable amount of French words.
- Clustering documents based on $\vec{\hat{y}}$ could help to discover documents written in regional dialects, or by multilingual authors.

Representations

- The vectors $\vec{x}$ containing the normalized letter-bigram counts for the documents are also representations of the documents.
- Arguably containing a similar kind of information to the vectors $\vec{\hat{y}}$.
- However, the representations in $\vec{\hat{y}}$ is more compact (6 entries instead of 784) and more specialized for the language prediction objective.
- Clustering by the vectors $\vec{x}$ would likely reveal document similarities that are not due to a particular mix of languages, but perhaps due to the document's topic or writing styles.

Representations

- The trained matrix $W \in \mathcal{R}^{784 \times 6}$ can also be considered as containing learned representations.
- We can consider two views of W, as rows or as columns.



Two views of the W matrix. (a) Each column corresponds to a language. (b) Each row corresponds to a letter bigram. Source: [Goldberg, 2017].

Representations

- A column of W can be taken to be a 784-dimensional vector representation of a language in terms of its characteristic letter-bigram patterns.
- We can then cluster the 6 language vectors according to their similarity.
- Each of the 784 rows of W provide a 6-dimensional vector representation of that bigram in terms of the languages it prompts.

Representations

- Representations are central to deep learning.
- One could argue that the main power of deep-learning is the ability to learn good representations.
- In the linear case, the representations are interpretable.
- We can assign a meaningful interpretation to each dimension in the representation vector.
- For example: each dimension corresponds to a particular language or letter-bigram.

Representations

- Deep learning models, on the other hand, often learn a cascade of representations of the input that build on top of each other.
- These representations are often not interpretable.
- We do not know which properties of the input they capture.
- However, they are still very useful for making predictions.

One-Hot Vector Representation

- The input vector $\vec{x}$ in our language classification example contains the normalized bigram counts in the document D.
- This vector can be decomposed into an average of $|D|$ vectors, each corresponding to a particular document position i:

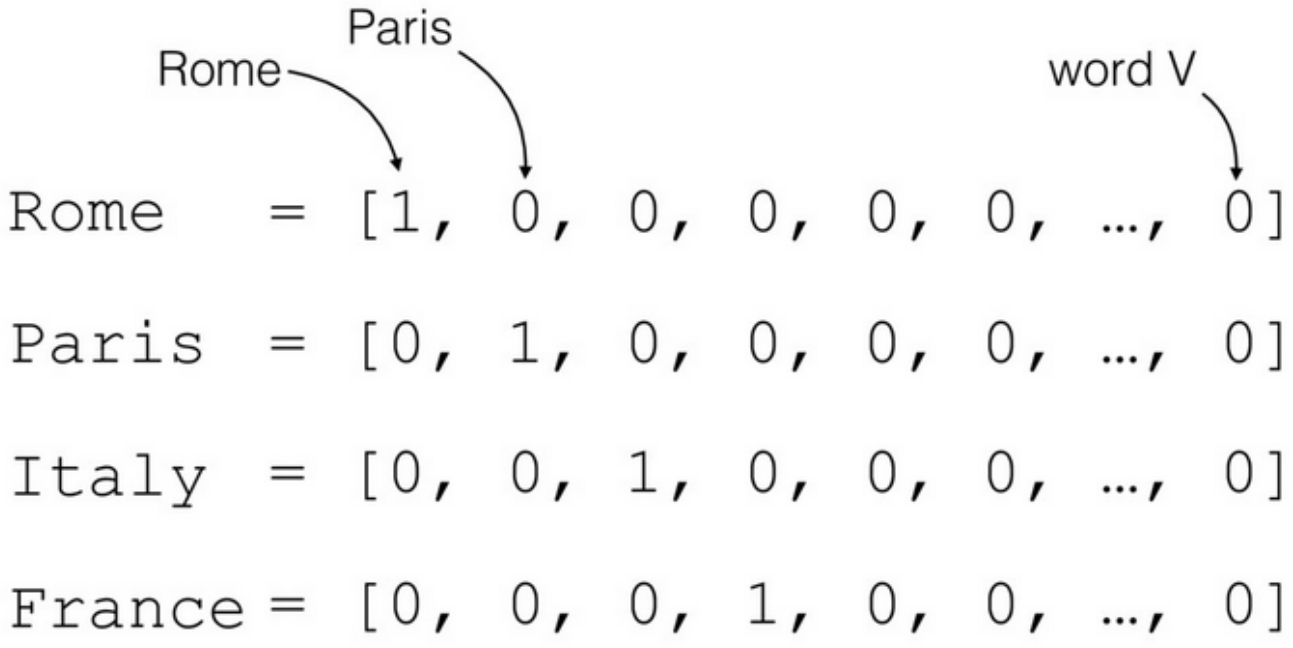
$$\vec{x} = \frac{1}{|D|} \sum_{i=1}^{|D|} \vec{x}^{D_{[i]}} \tag{10}$$

- Here, $D_{[i]}$ is the bigram at document position i.
- Each vector $\vec{x}^{D_{[i]}} \in \mathcal{R}^{784}$ is a one-hot vector.

One-Hot Vector Representation

- A one-hot vector: all entries are zero except the single entry corresponding to the letter bigram $D_{[i]}$, which is 1.
- The resulting vector $\vec{x}$ is commonly referred to as an averaged bag of bigrams (more generally averaged bag of words , or just bag of words).
- Bag-of-words (BOW) representations contain information about the identities of all the “words” (here, bigrams) of the document, without considering their order.
- A one-hot representation can be considered as a bag-of-a-single-word.

One-Hot Vector Representation



One-hot vectors of words. Source: `https://medium.com/@athif.shaffy/one-hot-encoding-of-text-b69124bef0a7`.

Log-linear Multi-class Classification

- In the binary case, we transformed the linear prediction into a probability estimate by passing it through the sigmoid function, resulting in a log-linear model.
- The analog for the multi-class case is passing the score vector through the **softmax** function:

$$\text{softmax}(\vec{x})_{[i]} = \frac{e^{\vec{x}_{[i]}}}{\sum_j e^{\vec{x}_{[j]}}} \tag{11}$$

Resulting in:

$$\begin{aligned} \hat{\vec{y}} &= \text{softmax}(\vec{x} \cdot W + \vec{b}) \\ \hat{\vec{y}}_{[i]} &= \frac{e^{(\vec{x} \cdot W + \vec{b})_{[i]}}}{\sum_j e^{(\vec{x} \cdot W + \vec{b})_{[j]}}} \end{aligned} \tag{12}$$

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

Limitations of linear models: the XOR problem

- The hypothesis class of linear (and log-linear) models is severely restricted.
- For example, it cannot represent the XOR function, defined as:

$$\begin{aligned} \text{xor}(0,0) &= 0 \\ \text{xor}(1,0) &= 1 \\ \text{xor}(0,1) &= 1 \\ \text{xor}(1,1) &= 0 \end{aligned} \tag{13}$$

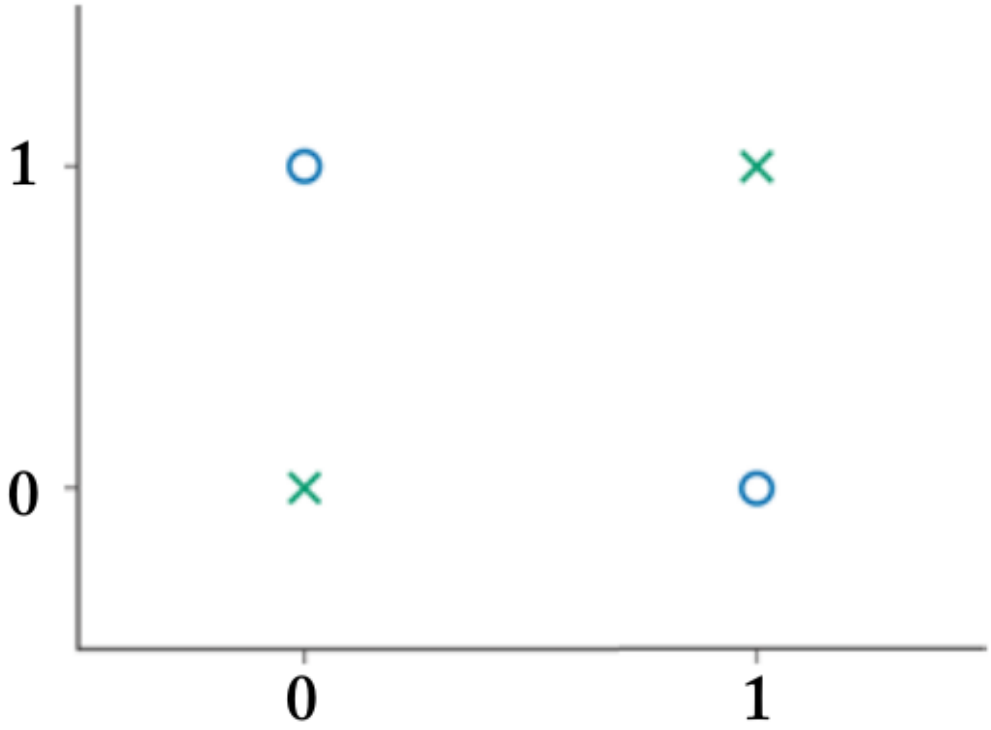
Limitations of linear models: the XOR problem

- There is no parameterization $\vec{w} \in \mathcal{R}^2, b \in \mathcal{R}$ such that:

$$\begin{aligned} (0,0) \cdot \vec{w} + b &< 0 \\ (0,0) \cdot \vec{w} + b &< 0 \\ (0,0) \cdot \vec{w} + b &< 0 \\ (0,0) \cdot \vec{w} + b &< 0 \end{aligned} \tag{14}$$

Limitations of linear models: the XOR problem

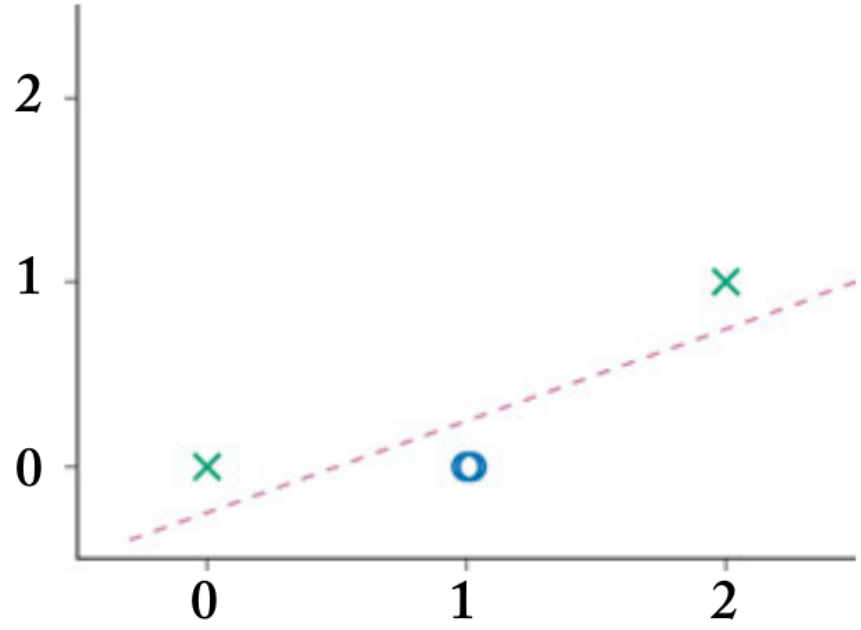
- To see why, consider the following plot of the XOR function, where blue Os denote the positive class and green Xs the negative class.



- It is clear that no straight line can separate the two classes.

Nonlinear input transformations

- If we transform the points by feeding each of them through the nonlinear function $\phi(x_1, x_2) = [x_1 \times x_2, x_1 x_2]$, the XOR problem becomes linearly separable.



- The function ϕ mapped the data into a representation that is suitable for linear classification.

Nonlinear input transformations

- We can now easily train a linear classifier to solve the XOR problem.

$$\hat{y} = f(\vec{x}) = \phi(\vec{x}) \cdot \vec{w} + b \tag{15}$$

- Problem: we need to manually define the function ϕ.
- This process is dependent on the particular dataset, and requires a lot of human intuition.
- Solution: define a trainable nonlinear mapping function, and train it in conjunction with the linear classifier.
- Finding the suitable representation becomes the responsibility of the training algorithm.

Trainable mapping functions

- The mapping function can take the form of a parameterized linear model.
- Followed by a nonlinear activation function g that is applied to each of the output dimensions:

$$\begin{aligned} \hat{y} = f(\vec{x}) &= \phi(\vec{x}) \cdot \vec{w} + b \\ \phi(\vec{x}) &= g(\vec{x}W' + \vec{b}') \end{aligned} \tag{16}$$

Trainable mapping functions

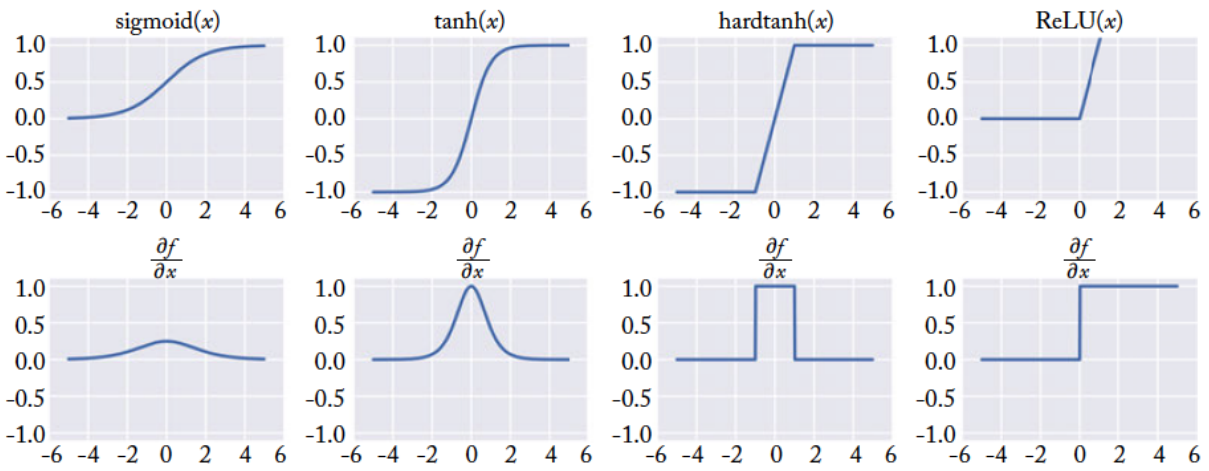
- By taking $g(x) = \max(0, x)$ and $W' = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$, $\vec{b}' = \begin{pmatrix} -1 & 0 \end{pmatrix}$.
- We get an equivalent mapping to $[x_1 \times x_2, x_1 + x_2]$ for the our points of interest (0,0), (0,1), (1,0), and (1,1).
- This successfuly solves the XOR problem!
- Learning both the representation function and the linear classifier on top of it at the same time is the main idea behind deep learning and neural networks.
- In fact, previous equation describes a very common neural network architecture called a multi-layer perceptron (MLP).

Introduction to Neural Networks

- Very popular machine learning models formed by units called **neurons**.
- A neuron is a computational unit that has scalar inputs and outputs.
- Each input has an associated weight w.
- The neuron multiplies each input by its weight, and then sums them (other functions such as **max** are also possible).
- It applies an activation function g (usually non-linear) to the result, and passes it to its output.
- Multiple layers can be stacked.

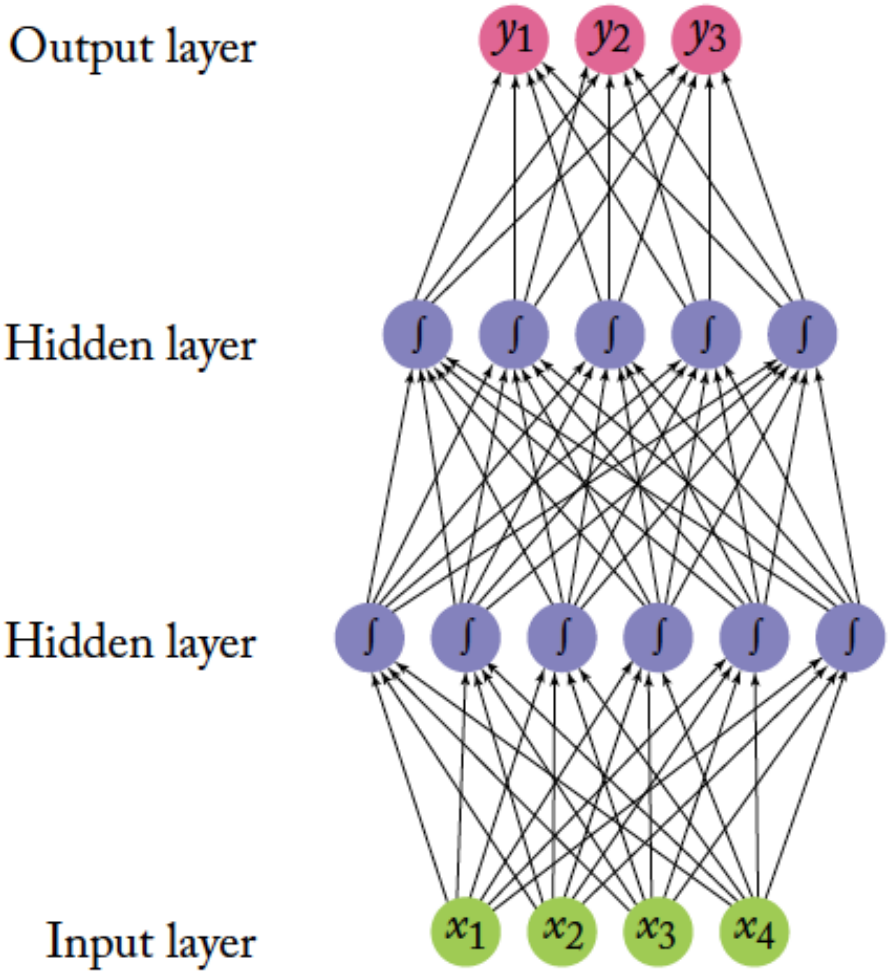
Activation Functions

- The nonlinear activation function *g* has a crucial role in the network's ability to represent complex functions.
- Without the nonlinearity in g, the neural network can only represent linear transformations of the input.



[0]Source:[Goldberg, 2017]

Feedforward Network with two Layers



[0]Source:[Goldberg, 2017]

Brief Introduction to Neural Networks

- The feedforward network from the picture is a stack of linear models separated by nonlinear functions.
- The values of each row of neurons in the network can be thought of as a vector.
- The input layer is a 4-dimensional vector ($\vec{x}$), and the layer above it is a 6-dimensional vector ($\vec{h}^1$).
- The fully connected layer can be thought of as a linear transformation from 4 dimensions to 6 dimensions.
- A fully connected layer implements a vector-matrix multiplication, $\vec{h} = \vec{x}W$.
- The weight of the connection from the i-th neuron in the input row to the j-th neuron in the output row is $W_{[i,j]}$.
- The values of $\vec{h}$ are transformed by a nonlinear function g that is applied to each value before being passed on as input to the next layer.

[0] Vectors are assumed to be row vectors and superscript indices correspond to network layers.

Brief Introduction to Neural Networks

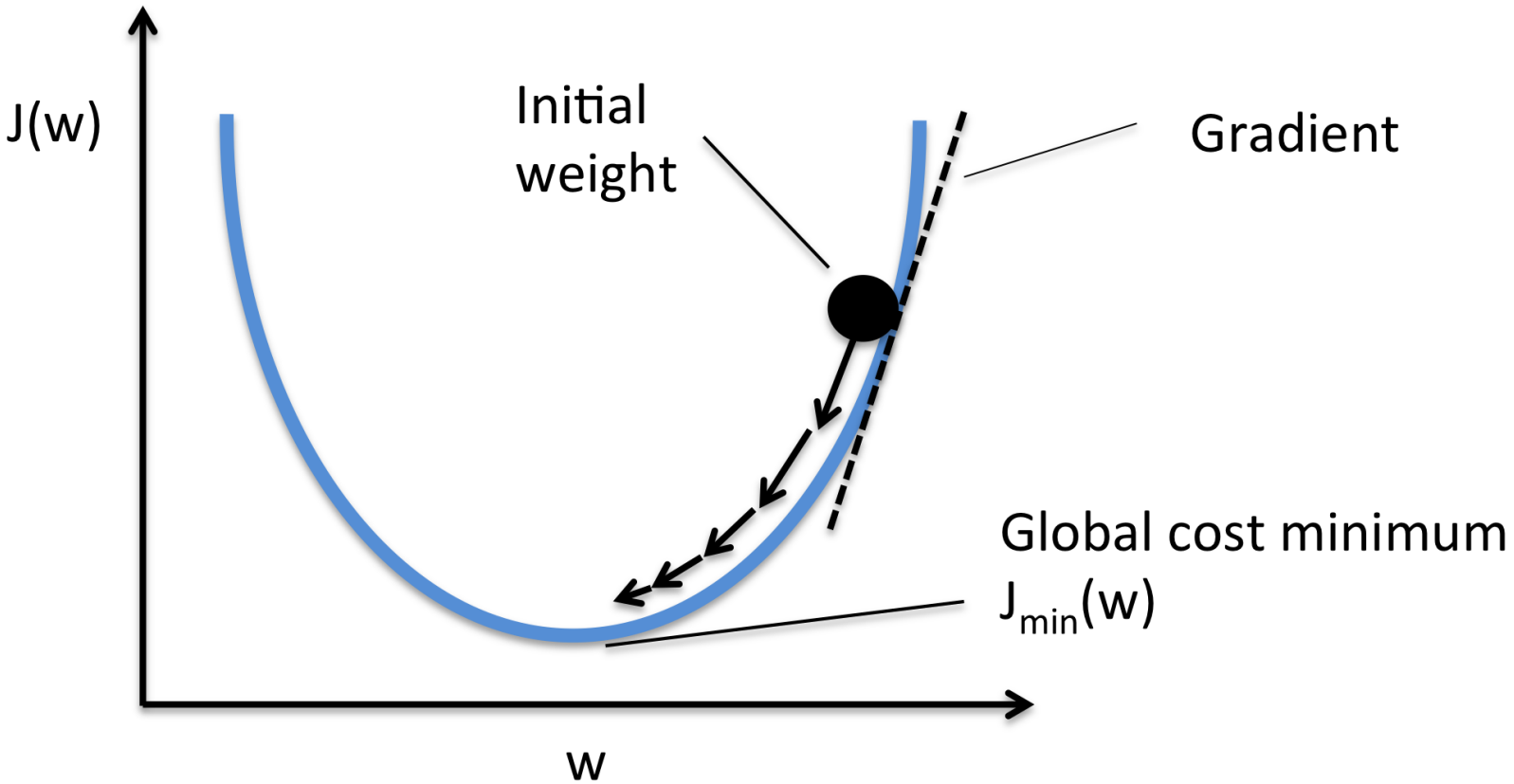
- The Multilayer Perceptron (MLP) from the figure can be written as the following mathematical function:

$$
\begin{aligned}
NN_{MLP2}(\vec{x}) &= \vec{y} \\
\vec{h}^1 &= g^1(\vec{x}W^1 + \vec{b}^1) \\
\vec{h}^2 &= g^2(\vec{h}^1 W^2 + \vec{b}^2) \\
\vec{y} &= \vec{h}^2 W^3 \\
\vec{y} &= (g^2(g^1(\vec{x}W^1 + \vec{b}^1)W^2 + \vec{b}^2))W^3.
\end{aligned}
\tag{17}
$$

Training

- When training a parameterized function (e.g., a linear model, a neural network) one defines a loss function $L(\hat{y}, y)$, stating the loss of predicting $\hat{y}$ when the true output is y.
- The training objective is then to minimize the loss across the different training examples.
- Functions are trained using gradient-based methods.
- They work by repeatedly computing an estimate of the loss L over the training set.
- They compute gradients of the parameters with respect to the loss estimate, and moving the parameters in the opposite directions of the gradient.
- Different optimization methods differ in how the error estimate is computed, and how moving in the opposite direction of the gradient is defined.

Gradient Descent



[0]Source: https://sebastianraschka.com/images/faq/closed-form-vs-gd/ball.png

Online Stochastic Gradient Descent

Algorithm 2.1 Online stochastic gradient descent training.

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

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

- The learning rate can either be fixed throughout the training process, or decay as a function of the time step t.
- The error calculated in line 3 is based on a single training example, and is thus just a rough estimate of the corpus-wide loss L that we are aiming to minimize.
- The noise in the loss computation may result in inaccurate gradients (single examples may provide noisy information).

[0]Source:[Goldberg, 2017]

Mini-batch Stochastic Gradient Descent

- A common way of reducing this noise is to estimate the error and the gradients based on a sample of *m* examples.
- This gives rise to the minibatch SGD algorithm

Algorithm 2.2 Minibatch stochastic gradient descent training.

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

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

- Higher values of *m* provide better estimates of the corpus-wide gradients, while smaller values allow more updates and in turn faster convergence.
- For modest sizes of *m* , some computing architectures (i.e., GPUs) allow an efficient parallel implementation of the computation in lines 3-6.

[0]Source:[Goldberg, 2017]

Some Loss Functions

- Hinge (or SVM loss): for binary classification problems, the classifier's output is a single scalar $\tilde{y}$ and the intended output y is in $\{+1, -1\}$. The classification rule is $\hat{y} = sign(\tilde{y})$, and a classification is considered correct if $y \cdot \tilde{y} > 0$.

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

- Binary cross entropy (or logistic loss): is used in binary classification with conditional probability outputs. The classifier's output $\tilde{y}$ is transformed using the sigmoid function to the range $[0, 1]$, and is interpreted as the conditional probability $P(y = 1|x)$.

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

Some Loss Functions

- Categorical cross-entropy loss: is used when a probabilistic interpretation of multi-class scores is desired. It measures the dissimilarity between the true label distribution y and the predicted label distribution $\tilde{y}$.

$$L_{\text{cross-entropy}}(\hat{y}, y) = -\sum_i y_{[i]} \log(\hat{y}_{[i]})$$

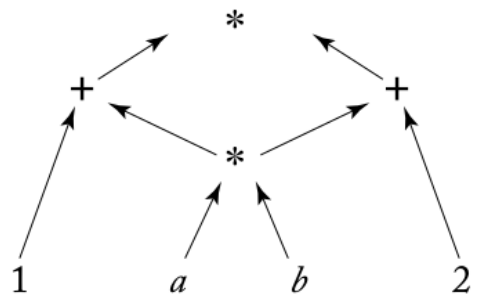
- The predicted label distribution of the categorical cross-entropy loss ($\hat{y}$) is obtained by applying the softmax function the last layer of the network $\tilde{y}$:

$$\hat{y}_{[i]} = \text{softmax}(\tilde{y})_{[i]} = \frac{e^{\tilde{y}_{[i]}}}{\sum_j e^{\tilde{y}_{[j]}}}$$

- The softmax function squashes the k-dimensional output to values in the range (0,1) with all entries adding up to 1. Hence, $\hat{y}_{[i]} = P(y = i|x)$ represent the class membership conditional distribution.

The Computation Graph Abstraction

- One can compute the gradients of the various parameters of a network by hand and implement them in code.
- This procedure is cumbersome and error prone.
- For most purposes, it is preferable to use automatic tools for gradient computation [Bengio, 2012].
- A computation graph is a representation of an arbitrary mathematical computation (e.g., a neural network) as a graph.
- Consider for example a graph for the computation of $(a * b + 1) * (a * b + 2)$:



- The computation of $a * b$ is shared.
- The graph structure defines the order of the computation in terms of the dependencies between the different components.

The Computation Graph Abstraction

- Te computation graph abstraction allows us to:
  1. Easily construct arbitrary networks.
  2. Evaluate their predictions for given inputs (forward pass)

Algorithm 5.3 Computation graph forward pass.

1: **for** i = 1 to N **do**
2: Let $a_1, \ldots, a_m = \pi^{-1}(i)$
3: $v(i) \leftarrow f_i(v(a_1), \ldots, v(a_m))$

  3. Compute gradients for their parameters with respect to arbitrary scalar losses (backward pass or backpropagation).

Algorithm 5.4 Computation graph backward pass (backpropagation).

1: $d(N) \leftarrow 1$ ▷ $\frac{\partial N}{\partial N} = 1$
2: **for** i = N-1 to 1 **do**
3: $d(i) \leftarrow \sum_{j \in \pi(i)} d(j) \cdot \frac{\partial f_j}{\partial i}$ ▷ $\frac{\partial N}{\partial i} = \sum_{j \in \pi(i)} \frac{\partial N}{\partial j} \frac{\partial j}{\partial i}$

- The backpropagation algorithm (backward pass) is essentially following the chain-rule of differentiation[1].

[1] A comprehensive tutorial on the backpropagation algorithm over the computational graph abstraction:
https://colah.github.io/posts/2015-08-Backprop/

Train, Test, and Validation Sets

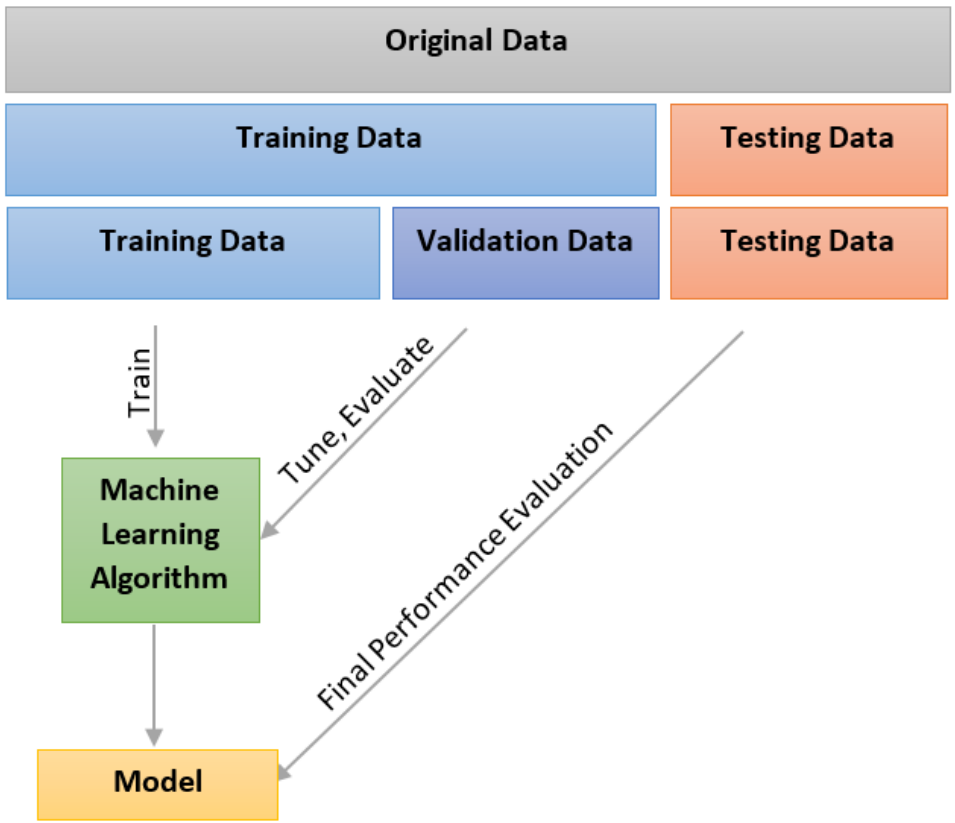
- Parameterized function are prone to overfit the data.
- Hence, performance on training data can be misleading.
- Held-out set: split training set into training and testing subsets (80% and 20% splits). Train on training and compute accuracy on testing.
- Problem: in practice you often train several models, compare their quality, and select the best one.
- Selecting the best model according to the held-out set's accuracy will result in an overly optimistic estimate of the model's quality.
- You don't know if the chosen settings of the final classifier are good in general, or are just good for the particular examples in the held-out sets.

Train, Test, and Validation Sets

- The accepted methodology is to use a three-way split of the data into train, validation (also called development), and test sets [2].
- This gives you two held-out sets: a validation set (also called development set), and a test set.
- All the experiments, tweaks, error analysis, and model selection should be performed based on the validation set.
- Then, a single run of the final model over the test set will give a good estimate of its expected quality on unseen examples.
- It is important to keep the test set as pristine as possible, running as few experiments as possible on it.
- Some even advocate that you should not even look at the examples in the test set, so as to not bias the way you design your model.

[2] An alternative approach is cross-validation, but it doesn't scale well for training deep neural networks.

Train, Test, and Validation Sets



[2]source: https://www.codeproject.com/KB/AI/1146582/validation.PNG

Deep Learning Frameworks

Several software packages implement the computation-graph model. All these packages support all the essential components (node types) for defining a wide range of neural network architectures.

- TensorFlow (`https://www.tensorflow.org/`): an open source software library for numerical computation using data-flow graphs originally developed by the Google Brain Team.
- Keras: High-level neural network API that runs on top of Tensorflow as well as other backends (`https://keras.io/`).
- PyTorch: open source machine learning library for Python, based on Torch, developed by Facebook's artificial-intelligence research group. It supports dynamic graph construction, a different computation graph is created from scratch for each training sample. (`https://pytorch.org/`)

Questions?

Thanks for your Attention!

References I


Goldberg, Y. (2017).
Neural network methods for natural language processing.
Synthesis Lectures on Human Language Technologies, 10(1):1–309.