6.86x Machine Learning with Python

This is a cheat sheet for machine learning based on the online course given by Prof. Tommi Jaakkola and Prof. Regina Barzilay. Compiled by Janus B. Advincula.

Last Updated January 14, 2020

Linear Classifiers

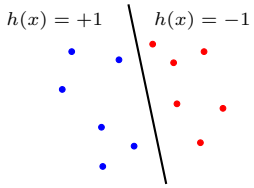
Introduction to Machine Learning

What is machine learning? Machine learning as a discipline aims to design, understand and apply computer programs that learn from experience (i.e., data) for the purpose of modeling, prediction or control.

Types of Machine Learning

- **Supervised learning:** prediction based on examples of correct behavior
- **Unsupervised learning:** no explicit target, only data, goal is to model/discover
- **Semi-supervised learning:** supplement limited annotations with unsupervised learning
- **Active learning:** learn to query the examples actually needed for learning
- **Transfer learning:** how to apply what you have learned from A to B
- **Reinforcement learning:** learning to act, not just predict; goal is to optimize the consequences of actions

Linear Classifier and Perceptron



Key Concepts

- **feature vectors, labels:**

$$x \in \mathbb{R}^d, \quad y \in \{-1, +1\}$$

- **training set:**

$$S_n = \left\{ \left(x^{(i)}, y^{(i)} \right), i = 1, \ldots, n \right\}$$

- **classifier:**

$$h : \mathbb{R}^d \to \{-1, +1\}$$
$$\chi^+ = \left\{ x \in \mathbb{R}^d : h(x) = +1 \right\}$$
$$\chi^- = \left\{ x \in \mathbb{R}^d : h(x) = -1 \right\}$$

- **training error:**

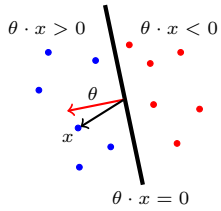
$$\mathcal{E}_n(h) = \frac{1}{n} \sum_{i=1}^{n} \left[\left[h\left(x^{(i)} \right) \neq y^{(i)} \right] \right]$$

$$\left[\left[h\left(x^{(i)} \right) \neq y^{(i)} \right] \right] = \begin{cases} 1 & \text{if error} \\ 0 & \text{otherwise} \end{cases}$$

- **test error:** $\mathcal{E}(h)$
- **set of classifiers:** $h \in \mathcal{H}$

Linear Classifiers through the Origin We consider functions of the form

$$h(x; \theta) = \text{sign}\left(\theta_1 x_1 + \cdots + \theta_d x_d \right) = \text{sign}\left(\theta \cdot x \right).$$



Linear Classifiers with Offset We can consider functions of the form

$$h(x; \theta) = \text{sign}\left(\theta \cdot x + \theta_0 \right)$$

where θ_0 is the offset parameter.

Linear Separation Training examples S_n are *linearly separable* if there exists a parameter vector $\widehat{\theta}$ and offset parameter $\widehat{\theta}_0$ such that $y^{(i)} \left(\widehat{\theta} \cdot x^{(i)} + \widehat{\theta}_0 \right) > 0$ for all $i = 1, \ldots, n$.

Training Error The training error for a linear classifier is

$$\mathcal{E}_n\left(\theta, \theta_0 \right) = \frac{1}{n} \sum_{i=1}^{n} \left[\left[y^{(i)} \left(\theta \cdot x^{(i)} + \theta_0 \right) \leq 0 \right] \right].$$

Perceptron Algorithm

procedure PERCEPTRON$\left(\{ (x^{(i)}, y^{(i)}), i = 1, \ldots, n \}, T \right)$
  $\theta = 0$ (vector)
  for $t = 1, \ldots, T$ **do**
    for $i = 1, \ldots, n$ **do**
      if $y^{(i)} \left(\theta \cdot x^{(i)} \right) \leq 0$ **then**
        $\theta = \theta + y^{(i)} x^{(i)}$
  return θ

Perceptron Algorithm (with offset)

procedure PERCEPTRON$\left(\{ (x^{(i)}, y^{(i)}), i = 1, \ldots, n \}, T \right)$
  $\theta = 0$ (vector)
  for $t = 1, \ldots, T$ **do**
    for $i = 1, \ldots, n$ **do**
      if $y^{(i)} \left(\theta \cdot x^{(i)} + \theta_0 \right) \leq 0$ **then**
        $\theta = \theta + y^{(i)} x^{(i)}$
        $\theta_0 = \theta_0 + y^{(i)}$
  return θ, θ_0

Convergence Assumptions:

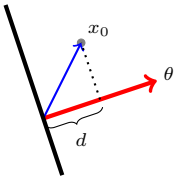
- There exists θ^* such that $\frac{y^{(i)} \left(\theta^* \cdot x^{(i)} \right)}{\left\| x^{(i)} \right\|} \geq \gamma$ for all $i = 1, \ldots, n$ for some $\gamma > 0$.
- All examples are bounded $\left\| x^{(i)} \right\| \leq R, i = 1, \ldots, n$.

Then the number k of updates made by the perceptron algorithm is bounded by $\frac{R^2}{\gamma^2}$.

Hinge Loss, Margin Boundaries and Regularization

Distance from a Line to a Point The perpendicular distance from a line with equation $\theta \cdot x + \theta_0 = 0$ to a point with coordinates x_0 is

$$d = \frac{|\theta \cdot x_0 + \theta_0|}{\|\theta\|}$$

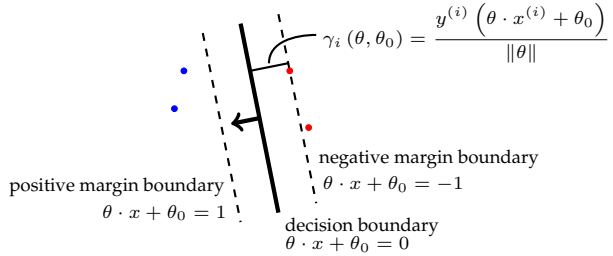


Decision Boundary The decision boundary is the set of points x which satisfy

$$\theta \cdot x + \theta_0 = 0.$$

Margin Boundary The margin boundary is the set of points x which satisfy

$$\theta \cdot x + \theta_0 = \pm 1.$$



Hinge Loss

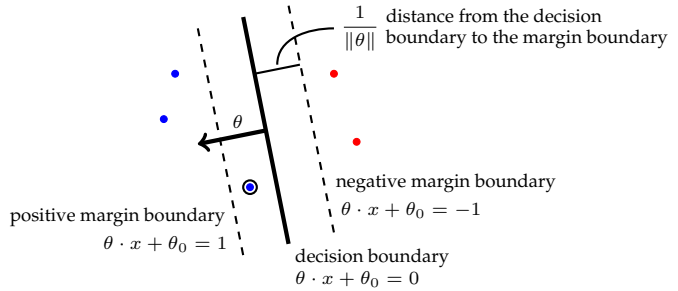
$$\text{Loss}_h\left(z \right) = \begin{cases} 0 & \text{if } z \geq 1 \\ 1 - z & \text{if } z < 1 \end{cases}$$

with $z = y^{(i)} \left(\theta \cdot x^{(i)} + \theta_0 \right)$.

Regularization Maximize margin

$$\max \frac{1}{\|\theta\|} \quad \Rightarrow \quad \min \frac{1}{2} \|\theta\|^2$$

Linear Classification and Generalization



Objective function

$$J\left(\theta, \theta_0 \right) = \frac{1}{n} \sum_{i=1}^{n} \left[\text{Loss}_h \left(y^{(i)} \left(\theta \cdot x^{(i)} + \theta_0 \right) \right) + \frac{\lambda}{2} \|\theta\|^2 \right]$$

λ is the regularization factor.

Stochastic Gradient Descent Select $i \in \{1, \ldots, n\}$ at random

$$\theta \quad \leftarrow \quad \theta - \eta_t \nabla_\theta \left[\text{Loss}_h \left(\theta \cdot x^{(i)} + \theta_0 \right) + \frac{\lambda}{2} \|\theta\|^2 \right]$$

η_t is the learning rate which can vary at every iteration.

Support Vector Machine

- Support Vector Machine finds the maximum margin linear separator by solving the quadratic program that corresponds to $J(\theta, \theta_0)$
- In the realizable case, if we disallow any margin violations, the quadratic program we have to solve is:

  Find θ, θ_0 that minimize $\frac{1}{2}\|\theta\|^2$ subject to

$$y^{(i)} \left(\theta \cdot x^{(i)} + \theta_0 \right) \geq 1, \quad i = 1, \ldots, n$$

Nonlinear Classification, Linear Regression, Collaborative Filtering

Linear Regression

Empirical Risk

$$R_n(\theta) = \frac{1}{n} \sum_{i=1}^{n} \frac{1}{2} \left(y^{(i)} - \theta \cdot x^{(i)} \right)^2 \qquad \text{squared error}$$

Gradient-based Approach We can use stochastic gradient descent to find the minima of the empirical risk.

Algorithm Initialize $\theta = 0$.
Randomly pick $i = \{1, \ldots, n\}$.
$\theta = \theta + \eta \left(y^{(i)} - \theta \cdot x^{(i)} \right) x^{(i)}$.

η is the learning rate.

Closed Form Solution Let

$$A = \frac{1}{n} \sum_{i=1}^{n} x^{(i)} \left(x^{(i)} \right)^\intercal \quad \text{and} \quad B = \frac{1}{n} \sum_{i=1}^{n} y^{(i)} x^{(i)}.$$

Then,

$$\widehat{\theta} = A^{-1} B.$$

In matrix notation, this is

$$\widehat{\theta} = \left(\mathbb{X}^\intercal \mathbb{X} \right)^{-1} \mathbb{X}^\intercal \mathbb{Y}.$$

Generalization and Regularization

Ridge Regression: The loss function is

$$J_{\lambda,n} = \frac{\lambda}{2} \|\theta\|^2 + R_n(\theta)$$

where λ is the regularization factor. We can find its minima using gradient-based approach.

Algorithm Initialize $\theta = 0$.
Randomly pick $i = \{1, \ldots, n\}$.
$\theta = (1 - \eta\lambda)\theta + \eta \left(y^{(i)} - \theta \cdot x^{(i)} \right) x^{(i)}$.

Nonlinear Classification

Feature Transformation

$$x \mapsto \phi(x)$$
$$\theta \cdot x \to \theta' \cdot \phi(x)$$

Non-linear Classification

$$h(x; \theta, \theta_0) = \text{sign}(\theta \cdot \phi(x) + \theta_0)$$

Kernel Function A kernel function is simply an inner product between two feature vectors. Using kernels is advantageous when the inner products are faster to evaluate than using explicit vectors (e.g., when the vectors would be infinite dimensional).

$$K(x, x') = \phi(x) \cdot \phi(x')$$

Perceptron

$\theta = 0$
for $i = 1, \ldots, n$ **do**
  if $y^{(i)} \theta \cdot \phi\left(x^{(i)}\right) \leq 0$ **then**
    $\theta \leftarrow \theta + y^{(i)} \phi\left(x^{(i)}\right)$

This algorithm gives

$$\theta = \sum_{j=1}^{n} \alpha_j y^{(j)} \phi\left(x^{(j)}\right)$$

where α_j is the number of mistakes. For the offset parameter, we get

$$\theta_0 = \sum_{j=1}^{n} \alpha_j y^{(j)}.$$

Kernel Perceptron Algorithm We can reformulate the perceptron algorithm so that we initialize and update α_j's, instead of θ.

$$\theta \cdot \phi\left(x^{(i)}\right) = \sum_{j=1}^{n} \alpha_j y^{(j)} \underbrace{\phi\left(x^{(j)}\right) \cdot \phi\left(x^{(i)}\right)}_{K\left(x^{(j)}, x^{(i)}\right)}$$

procedure KERNEL PERCEPTRON$\left(\{(x^{(i)}, y^{(i)}), i = 1, \ldots, n\}, T\right)$
  Initialize $\alpha_1, \ldots, \alpha_n$ to some values
  for $t = 1, \ldots, T$ **do**
    for $i = 1, \ldots, n$ **do**
      if $y^{(i)} \sum_{j=1}^{n} \alpha_j y^{(j)} K\left(x^{(j)}, x^{(i)}\right) \leq 0$ **then**
        $\alpha_j = \alpha_j + 1$

The initilization $\theta = 0$ is equivalent to $\alpha_1 = \cdots = \alpha_n = 0$.

Composition rules:

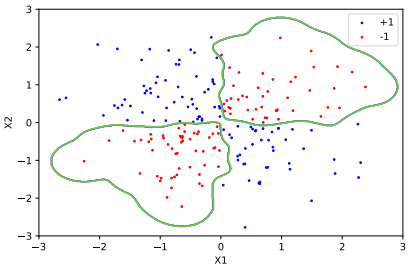
1. $K(x, x') = 1$ is a kernel function.
2. Let $f : \mathbb{R}^d \to \mathbb{R}$ and $K(x, x')$ is a kernel. Then so is $\widetilde{K}(x, x') = f(x) K(x, x') f(x')$
3. If $K_1(x, x')$ and $K_2(x, x')$ are kernels, then $K(x, x') = K_1(x, x') + K_2(x, x')$ is a kernel.
4. If $K_1(x, x')$ and $K_2(x, x')$ are kernels, then $K(x, x') = K_1(x, x') K_2(x, x')$ is a kernel.

Decision Boundary The decision boundary satisfies

$$\sum_{j=1}^{n} \alpha_j y^{(j)} K\left(x^{(j)}, x\right) = 0.$$

Radial Basis Kernel

$$K(x, x') = \exp\left(-\frac{1}{2} \|x - x'\|^2 \right)$$



Other non-linear classifiers

- We can get non-linear classifiers or regression methods by simply mapping examples into feature vectors non-linearly, and applying a linear method on the resulting vectors.
- These feature vectors can be high dimensional.
- We can turn the linear methods into kernel methods by casting the computations in terms of inner products.

Recommender Systems

Problem Description We are given a matrix where each row corresponds to a user's rating of movies, for example, and each column corresponds to the user ratings for a particular movie. It can also be product ratings, etc. This matrix will be very sparse. The goal is to predict user ratings for those movies that are yet to be rated.

$$\begin{array}{c} \\ n \text{ users} \end{array} \begin{array}{c} \overset{m \text{ movies}}{\overbrace{\qquad\qquad}} \\ \begin{pmatrix} \\ Y_{ai} \\ \\ \end{pmatrix} \end{array}$$

K-Nearest Neighbor Method The K-Nearest Neighbor method makes use of ratings by K other *similar* users when predicting Y_{ai}. Let $\text{KNN}(a)$ be the set of K users *similar to* user a, and let $\text{sim}(a, b)$ be a **similarity measure** between users a and $b \in \text{KNN}(a)$. The KNN method predicts a rating Y_{ai} to be

$$\widehat{Y}_{ai} = \frac{\sum_{b \in \text{KNN}(a)} \text{sim}(a, b) Y_{bi}}{\sum_{b \in \text{KNN}(a)} \text{sim}(a, b)}$$

The similarity measure $\text{sim}(a, b)$ could be any distance function between the feature vectors x_a and x_b.

- Euclidean distance: $\|x_a - x_b\|$
- Cosine similarity: $\cos\theta = \dfrac{x_a \cdot x_b}{\|x_a\| \|x_b\|}$

Collaborative Filtering Our goal is to come up with a matrix X that has no blank entries and whose $(a,i)^{\text{th}}$ entry X_{ai} is the prediction of user a's rating to movie i. Let D be the set of all (a,i)'s for which a user rating Y_{ai} exists. A naive approach is to minimize the objective function

$$J(X) = \sum_{(a,i)\in D} \frac{1}{2}(Y_{ai} - X_{ai})^2 + \frac{\lambda}{2}\sum_{(a,i)} X_{ai}^2.$$

The results are

$$\widehat{X}_{ai} = \frac{Y_{ai}}{1+\lambda} \quad \text{for } (a,i) \in D$$
$$\widehat{X}_{ai} = 0 \quad \text{for } (a,i) \notin D.$$

The problem with this approach is that there is no connection between the entries of X. We can impose additional constraint on X:

$$X = UV^\intercal$$

for some $n \times d$ matrix U and $d \times m$ matrix $V^\intercal$, where d is the *rank* of the matrix X.

Alternating Minimization Assume that U and V are rank k matrices. Then, we can write the objective function as

$$J(X) = \sum_{(a,i)\in D} \frac{1}{2}\left(Y_{ai} - [UV^\intercal]_{ai}\right)^2 + \frac{\lambda}{2}\left(\sum_{a,k} U_{ak}^2 + \sum_{i,k} V_{ik}^2\right).$$

To find the solution, we fix (initialize) U (or V) and minimize the objective with respect to V (or U). We plug-in the result back to the objective and minimize it with respect to U (or V). We repeat this alternating process until there is no change in the objective function.

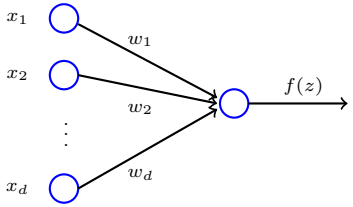
Example Consider the case $k = 1$. Then, $U_{a1} = u_a$ and $V_{i1} = v_i$. If we initialize u_a to some values, then we have to optimize the function

$$\sum_{(a,i)\in D} \frac{1}{2}(Y_{ai} - u_a v_i)^2 + \frac{\lambda}{2}\sum_i v_i^2.$$

Neural Networks

Introduction to Feedforward Neural Networks

A Unit in a Neural Network A **neural network unit** is a primitive neural network that consists of only the *input layer*, and an output layer with only one output.



A neural network unit computes a non-linear weighted combination of its input:

$$\widehat{y} = f(z) \quad \text{where} \quad z = w_0 + \sum_{i=1}^{d} x_i w_i$$

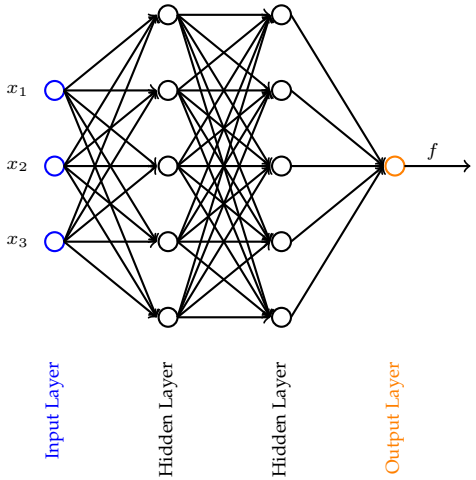
where w_i are the **weights**, z is a number and is the weighted sum of the inputs x_i, and f is generally a non-linear function called the **activation function**.

Linear Function $f(z) = z$

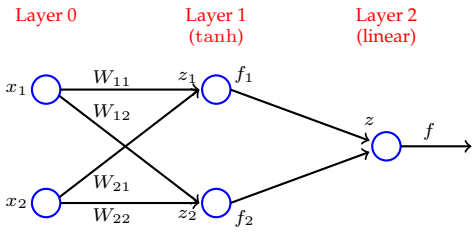
Rectified Linear Function (ReLU) $f(z) = \max\{0, z\}$

Hyperbolic Tangent Function $\tanh(z) = \dfrac{e^z - e^{-z}}{e^z - e^{-z}} = 1 - \dfrac{2}{e^{2z}+1}$

Deep Neural Networks A **deep (feedforward) neural network** refers to a neural network that contains not only the input and output layers, but also hidden layers in between. Below is a deep feedforward neural network of 2 hidden layers, with each hidden layer consisting of 5 units:

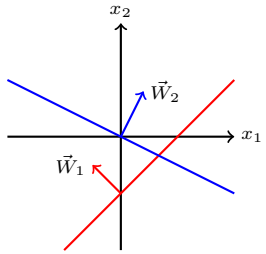


One Hidden Layer Model



$$z_1 = \sum_{j=1}^{2} x_j W_{j1} + W_{01} \qquad z_2 = \sum_{j=1}^{2} x_j W_{j2} + W_{02}$$
$$f_1 = f(z_1) = \tanh(z_1) \qquad f_2 = f(z_2) = \tanh(z_2)$$
$$z = f_1 w_1' + f_2 w_2' \qquad f = f(z) = z$$

Neural Signal Transformation We can visualize what the hidden layer is doing similarly to a linear classifier.



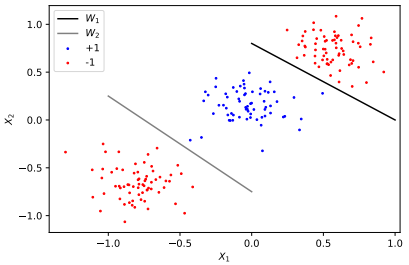
In the figure,

$$\vec{W}_1 = \begin{pmatrix} W_{11} \\ W_{21} \end{pmatrix} \quad \text{and} \quad \vec{W}_2 = \begin{pmatrix} W_{21} \\ W_{22} \end{pmatrix}.$$

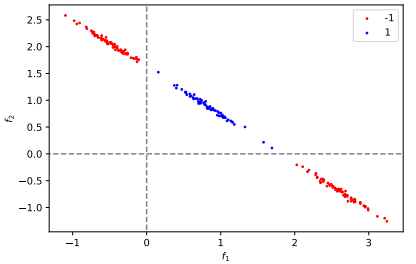
They map the input onto the f_1-f_2 axes.

Hidden Layer Representation

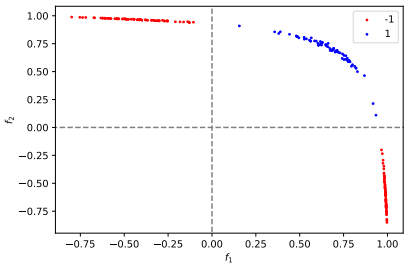
- Hidden Layer Units



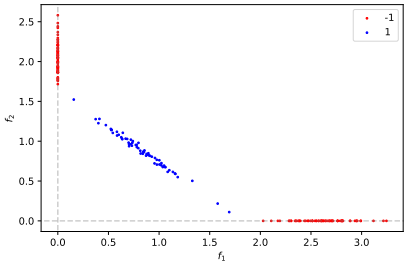
- Linear Activation



- $tanh$ Activation



- ReLU Activation

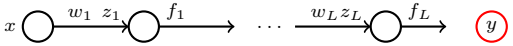


Summary

- Units in neural networks are linear classifiers, just with different output non-linearity.
- The units in feedforward neural networks are arranged in layers.
- By learning the parameters associated with the hidden layer units, we learn how to represent examples (as hidden layer activations).
- The representations in neural networks are learned directly to facilitate the end-to-end task.
- A simple classifier (output unit) suffices to solve complex classification tasks if it operates on the hidden layer representations.

Feedforward Neural Networks, Back Propagation, and Stochastic Gradient Descent (SGD)

Simple Example This simple neural network is made up of L hidden layers, but each layer consists of only one unit, and each unit has activation function f.



$$
\begin{aligned}
z_1 &= xw_1 \\
f_1 &= \tanh(xw_1) \\
&\vdots \\
f_L &= \tanh(f_{L-1}w_L) \\
\mathcal{L}(y, f_L) &= \text{Loss}\,(y, f_L) = \tfrac{1}{2}\,(y - f_L)^2
\end{aligned}
$$

For $i = 2, \ldots, L$: $z_i = f_{i-1}w_i$ where $f_{i-1} = f(z_{i-1})$. Also, y is the true value and f_L is the output of the neural network.

Gradient Descent The gradient descent update rule for the parameter w_i is

$$w_i \leftarrow w_i - \eta \cdot \nabla_{w_i} \mathcal{L}(y, f_L)$$

where η is the learning rate. For instance, we have

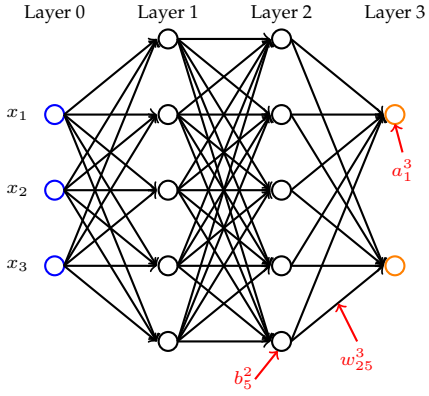
$$
\begin{aligned}
\frac{\partial \mathcal{L}}{\partial w_1} &= \frac{\partial f_1}{\partial w_1}\frac{\partial \mathcal{L}}{\partial f_1} \\
\frac{\partial f_1}{\partial w_1} &= \left[1 - \tanh^2(xw_1)\right] x = \left(1 - f_1^2\right) x \\
\frac{\partial \mathcal{L}}{\partial f_1} &= \frac{\partial \mathcal{L}}{\partial f_2}\frac{\partial f_2}{\partial f_1} = \frac{\partial \mathcal{L}}{\partial f_2}\left(1 - f_2^2\right) w_2.
\end{aligned}
$$

Thus, when we back-propagate, we get

$$\frac{\partial \mathcal{L}}{\partial w_1} = x\left(1 - f_1^2\right)\cdots\left(1 - f_L^2\right) w_2 \cdots w_L \cdot 2\,(f_L - y)\,.$$

Note that the above derivation applies to $\tanh$ activation.

Backpropagation Consider the L-layer neural network below.



We have the following notations:

- b_j^ℓ is the bias of the j^{th} neuron in the ℓ^{th} layer.
- a_j^ℓ is the activation of the j^{th} neuron in the ℓ^{th} layer.
- w_{jk}^ℓ is the weight for the connection from the k^{th} neuron in the $(\ell-1)^{\text{th}}$ layer to the j^{th} neuron in the ℓ^{th} layer.

If the activation function is f and the loss function we are minimizing is C, then the equations describing the network are:

$$a_j^\ell = f\left(\sum_k w_{jk}^\ell a_k^{\ell-1} + b_j^\ell\right)$$

$$\text{Loss} = C\left(a^L\right)$$

Let the weighted inputs to the d neurons in layer ℓ be defined as

$$z^\ell \equiv w^\ell a^{\ell-1} + b^\ell, \quad \text{where } z^\ell \in \mathbb{R}^d.$$

Then, the activation of layer ℓ is also written as $a^\ell \equiv f(z^\ell)$. Also, let $\delta_j^\ell \equiv \frac{\partial C}{\partial z_j^\ell}$ denote the *error* of neuron j in layer ℓ. Then, $\delta^\ell \in \mathbb{R}^d$ denotes the full vector of errors associated with layer ℓ.

Equations of Backpropagation

$$
\begin{aligned}
\delta^L &= \nabla_a C \odot f'\left(z^L\right) \\
\delta^\ell &= \left[\left(w^{\ell+1}\right)^\intercal \delta^{\ell+1}\right] \odot f'\left(z^\ell\right) \\
\frac{\partial C}{\partial b_j^\ell} &= \delta_j^\ell \\
\frac{\partial C}{\partial w_{jk}^\ell} &= a_k^{\ell-1}\delta_j^\ell
\end{aligned}
$$

The symbol $\odot$ represents the Hadamard product.

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \odot \begin{pmatrix} e & f \\ g & h \end{pmatrix} = \begin{pmatrix} ae & bf \\ cg & dh \end{pmatrix}.$$

Recurrent Neural Networks

Temporal/Sequence Problems

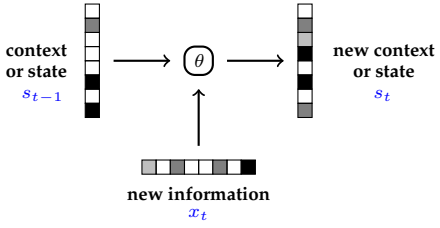
- Sequence prediction problems can be recast in a form amenable to feedforward neural networks.
- We have to engineer how *history* is mapped to a vector (representation). This vector is then fed into, e.g., a neural network.
- We would like to learn how to encode the *history* into a vector.

Key Concepts

- **Encoding** – e.g., mapping a sequence to a vector
- **Decoding** – e.g., mapping a vector to, e.g., a sequence

Example: Encoding Sentences

- Introduce adjustable *lego pieces* and optimize them for end-to-end performance.



$$s_t = \tanh\left(W^{s,s}s_{t-1} + W^{s,x}x_t\right)$$

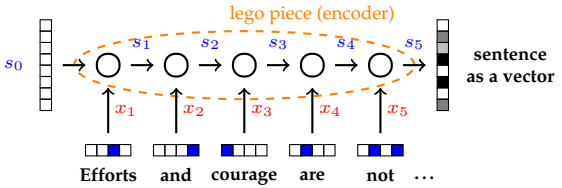
- Let's say we want to encode the incomplete sentence **Efforts and courage are not**. First, we have to represent the first word as a vector (say, a one-hot vector). This will be x_1. Then,

$$s_1 = \tanh\left(W^{s,x}x_1\right).$$

The second word will be x_2, and we compute for s_2.

$$s_2 = \tanh\left(W^{s,s}s_1 + W^{s,x}x_t\right).$$

We continue this process until we've encoded all the words in the sentence. We can visualize this as follows:



Differences from standard feedforward architecture

- Input is received at each layer (per word), not just at the beginning as in a typical feedforward network.
- The number of layers varies and depends on the length of the sentence.
- Parameters of each layer (representing an application of an RNN) are shared (same RNN at each step).

Basic RNN

$$s_t = \tanh\left(W^{s,s}s_{t-1} + W^{s,x}x_t\right)$$

Simple Gated RNN

$$
\begin{aligned}
g_t &= \text{sigmoid}\left(W^{g,s}s_{t-1} + W^{g,x}x_t\right) \\
s_t &= (1 - g_t) \odot s_{t-1} + g_t \odot \tanh\left(W^{s,s}s_{t-1} + W^{s,x}x_t\right)
\end{aligned}
$$

Long Short-Term Memory (LSTM)

$$
\begin{aligned}
f_t &= \text{sigmoid}\left(W^{f,h}h_{t-1} + W^{f,x}x_t\right) && \text{forget gate} \\
i_t &= \text{sigmoid}\left(W^{i,h}h_{t-1} + W^{i,x}x_t\right) && \text{input gate} \\
o_t &= \text{sigmoid}\left(W^{o,h}h_{t-1} + W^{o,x}x_t\right) && \text{output gate} \\
c_t &= f_t \odot c_{t-1} + i_t \odot \tanh\left(W^{c,h}h_{t-1} + W^{c,x}x_t\right) && \text{memory cell} \\
h_t &= o_t \odot \tanh(c_t) && \text{visible state}
\end{aligned}
$$

Markov Language Models Let $w \in V$ denote the set of possible words/symbols that includes

- an **UNK** symbol for any unknown word (out of vocabulary)
- ⟨**beg**⟩ symbol for specifying the start of a sentence
- ⟨**end**⟩ symbol for specifying the end of the sentence

First-order Markov Model In a first-order Markov model (**bigram model**), the next symbol only depends on the previous one. Each symbol (except ⟨beg⟩) in the sequence is predicted using the same condition probability table until an ⟨end⟩ symbol is seen. The probability associated to the is

$$\prod_{i=1} \mathbb{P}\left(w_i | w_{i-1}\right).$$

Maximum Likelihood Estimation The goal is to maximize the probability that the model can generate all the observed sentences (corpus S)

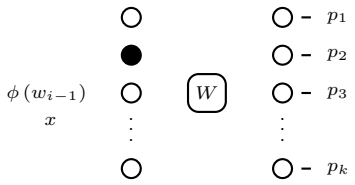
$$s \in S, s = \left\{w_1^s, w_2^s, \ldots, w_{|s|}^s\right\}$$

$$\ell = \log\left\{\prod_{s \in S}\left[\prod_{i=1}^{|s|} \mathbb{P}\left(w_i^s | w_{i-1}^s\right)\right]\right\}$$

The maximum likelihood estimate is obtained as normalized counts of successive word occurrences (matching statistics)

$$\widehat{\mathbb{P}}\left(w' | w\right) = \frac{\text{count}\left(w', w\right)}{\sum_{\tilde{w}} \text{count}\left(w, \tilde{w}\right)}$$

Feature-based Markov Model We can also represent the Markov model as a feedforward neural network (very extendable). We define a one-hot vector, $\phi\left(w_{i-1}\right)$, corresponding to the previous word. This will be an input to the feedforward neural network.



In the figure,

$$p_k = \mathbb{P}\left(w_i = k | w_{i-1}\right)$$

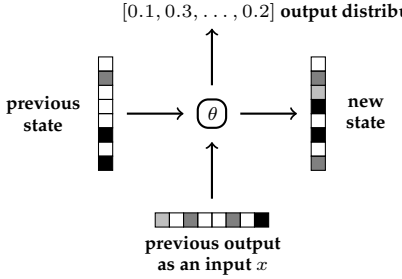
is the probability of the next word, given the previous word. The aggregate input to the k^{th} output unit is

$$z_k = \sum_k x_j W_{jk} + W_{0k}.$$

These input values are not probabilities. A typical transformation is the **softmax transformation**:

$$p_k = \frac{e^{z_k}}{\sum_j e^{z_j}}.$$

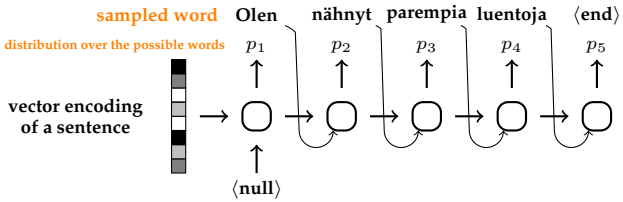
RNNs for Sequences Our RNN now also produces an output (e.g., a word) as well as update its state



$$s_t = \tanh\left(W^{s,s} s_{t-1} + W^{s,x} x_t\right) \quad \text{state}$$

$$p_t = \text{softmax}\left(W^o s_t\right) \quad \text{output distribution}$$

Decoding

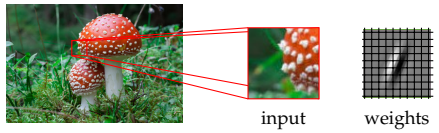


Convolutional Neural Networks

Problem Image classification

- The presence of objects may vary in location across different images.

Patch classifier/filter



The patch classifier goes through the entire image. We can think of the weights as the image that the unit prefers to see.

Convolution The convolution is an operation between two functions f and g:

$$(f * g)(t) \equiv \int_{-\infty}^{+\infty} f(\tau) g(t - \tau) d\tau.$$

Intuitively, convolution *blends* the two functions f and g by expressing the amount of overlap of one function as it is shifted over another function.

Discrete Convolution For discrete functions, we can define the convolution as

$$(f * g)[n] \equiv \sum_{m=-\infty}^{m=+\infty} f[m] g[n - m].$$

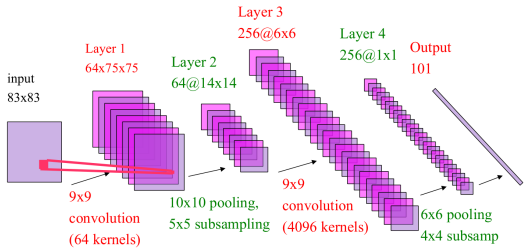
Pooling We wish to know whether a feature was there but not exactly where it was.

Pooling (Max) Pooling region and *stride* may vary.

- Pooling induces translation invariance at the cost of spatial resolution.
- Stride reduces the size of the resulting feature map.

original | feature map | feature map after max pooling

Example of CNN From LeCun (2013)



Unsupervised Learning

Clustering

A random variable associates a value to every possible outcome. It can take discrete or continuous values.

Generative Models

Definition of Variance

Mixture Models; EM Algorithm

Total Expectation Theorem Given a random variable X and events $A_1, \ldots, A_n$, we have

$$\mathbb{E}\left[X\right] = \mathbb{P}(A_1)\mathbb{E}\left[X|A_1\right] + \cdots + \mathbb{P}(A_n)\mathbb{E}\left[X|A_n\right].$$

Reinforcement Learning

Markov Decision Processes

Definition A random variable is continuous if it can be described by a PDF, $f_X(x)$, such that

Bellman Equations

Value Iteration Algorithm

Q-Value Iteration

Recommended Resources

- Introduction to Machine Learning with Python (Müller and Guido)
- Machine Learning with Python – From Linear Models to Deep Learning [Lecture Slides] (`http://www.edx.org`)
- LaTeX File (`github.com/mynameisjanus/186501xStatistics`)

Please share this cheatsheet with friends!