Deep Learning Notes

Riccardo Cappi

February 2024

0.1 Disclaimer

These are just my notes that I used to prepare for the exam. So, probably, there will be both spelling and conceptual errors. Feel free to contact me at riccardo.cappi@studenti.unipd.it if you find any errors. This is the github repo where you can find the latex files of the notes: `https://github.com/riccardocappi/Computer-Science-notes`

Contents

Chapter 1

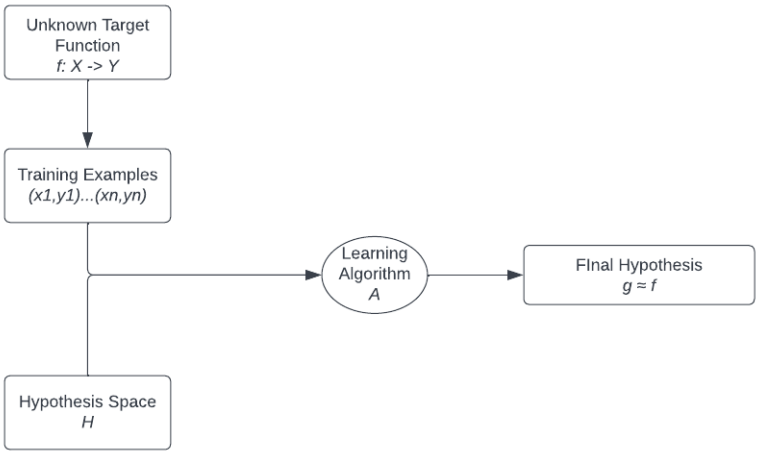
Lec 02-03 Machine Learning Basics

1.1 What is Machine Learning

A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E. Basically, Machine Learning is the field of study that gives computers the ability to learn without being explicitly programmed. In fact, we use Machine Learning when its impossible to **exactly formalise** the problem (and so to give an algorithmic solution) or when formulating a solution it's very complex and cannot be done manually.

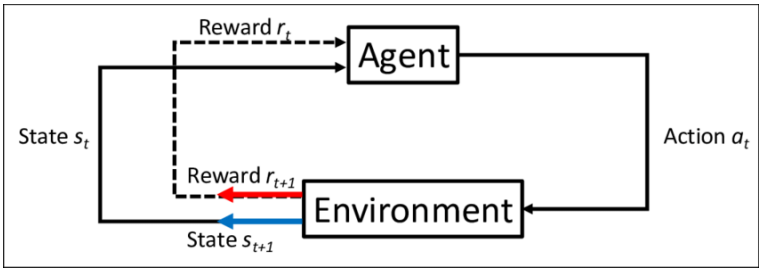
1.1.1 Main Learning Paradigms

- **Supervised Learning:**
  - **Goal:** give the *right answer* for each example in the data.
  - Given a training set $\{(x^{(i)}, y^{(i)})\}$ we look for a function $h(\cdot)$ which is able to map in a predictive way $x^{(i)}$'s to $y^{(i)}$'s. It's called supervised learning because there is an expert that provides a *supervision* assigning a label $y^{(i)}$ to each input $x^{(i)}$
  - **Output:** Classification, regression.
  - Use cases: Object recognition, Predicting pandemic, ...



The *Training examples* are generated according to the *Target function* f (unknown). Once we have this set of pairs, we choose the *hypothesis space*. The *learning algorithm* (e.g a Neural Network) searches in the hypothesis space for a function g that approximates the target function f.

- **Unsupervised Learning:**
  - **Goal:** Find regularities / patterns on the data
  - Given examples $\{x^{(i)}\}$, discover regularities on the whole input domain.
  - There is no supervision.
  - Use cases: Community detection in social media, user profiling, market analysis ...
- **Reinforcement Learning:** An **Agent** operates in an environment e, which in response to action a (given by the agent) in the state s returns the next state and a reward r (which can be positive, negative or neutral). The goal of the Agent is to maximize a reward function.
  - Use cases: Robotics, Games, ...



- **Other Learning Strategies:**
  - Active Learning
  - Online Learning, Incremental & Continual Learning
  - Weak Supervised Learning
  - Self-supervised Learning
  - Deep Learning and Representation Learning
  - Federated Learning

1.1.2 Supervised Learning Keywords

- **Input/Instance space $\mathbf{x} \in X$:** Representation of model's input (e.g. you can choose a Vector as a representation for your input). It contains all the possible inputs for a model. Suppose the model takes in a vector, $input = [x1, x2], x1, x2 \in [1, 10]$, then we have 10^2 possible inputs.
- **Output space $y \in Y$:** In supervised learning we want to perform a prediction based on the input. This prediction can be in the form of:
  - Binary Classification $y \equiv \{-1, +1\}$
  - Multi-Class Classification $y \equiv \{1, ..., m\}$
  - Regression $y \equiv \mathbb{R}$
- **Oracle/Nature:** It determines how examples are generated. We can have two cases of Oracle

  - Target function $f : X \to Y$ It's deterministic and given an object of the input space returns an object of the output space. This function is ideal and **unknown**.
  - Probability distribution $P(\mathbf{x}), P(y \mid \mathbf{x})$ The *selection* of y occurs from a probability distribution. This distribution is still unknown

- **Training set:** Set of pairs $\{(\mathbf{x}_1, y_1), ..., (\mathbf{x}_n, y_n)\}$ where each pair is composed by an instance of the input space and it's corresponding label.

$\mathbf{x}$	y
000	0
001	1
010	1
.	.
.	.
.	.

  Data are typically:

  - Independent: Given two pairs A, B $P(A \mid B) = P(A)$. The choice of one pair is independent from the choice of other pairs.
  - Identically distributed: All pairs are generated by the same probability distribution (the Oracle) $P(\mathbf{x}, y) = P(\mathbf{x})P(y \mid \mathbf{x})$. *Concept drift* is when data aren't identically distributed

- **Hypothesis space:** A **predefined** set of hypothesis/functions $H \equiv \{h \mid h : X \to Y\}$

- **Empirical error/risk:** Discrepancy between the target function f and my approximation of that function $g \in H$ (chosen from the hypothesis space) **on training data**. For example, in a binary classification problem we can compute empirical error as follows:

$$\frac{1}{n}\sum_{i=1}^{n}[\![y_i \neq g(\mathbf{x}_i)]\!]$$

  where $[\![\cdot]\!]$ is a function that is 1 if $\cdot$ is true and 0 otherwise.

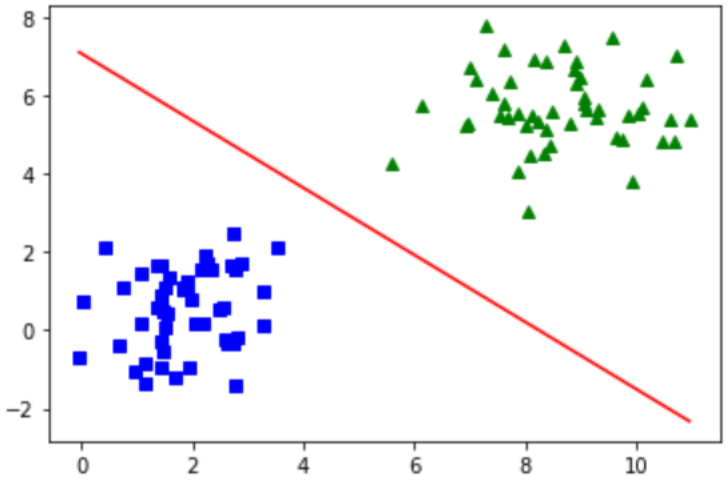
- **Ideal error:** The **expected** error of hypothesis h with respect to target concept c and distribution D is the probability that h will misclassify an instance drawn according to D. This can only be estimated. One way to estimate this quantity is testing the model over new examples that are not in the training set (test set)

- **Inductive bias:** Since the hypothesis space can't contain all possible functions, we must make assumptions about the type of the unknown target function. The inductive bias consists of:

  - The hypothesis space: how H is defined
  - The learning algorithm: how H is explored

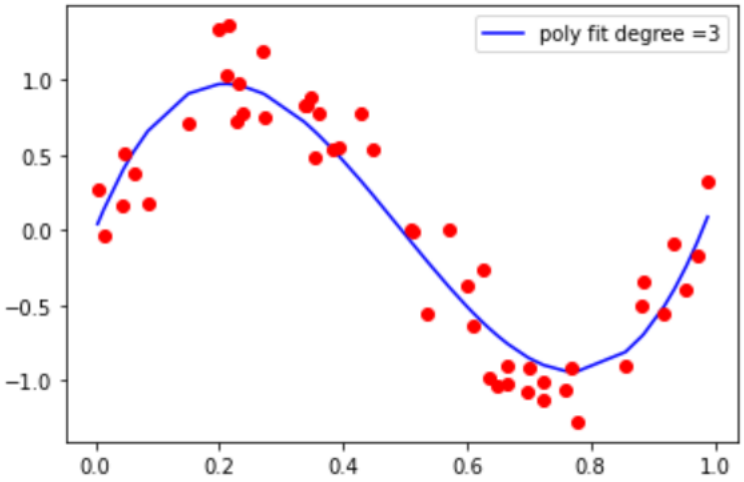
  Examples of Inductive bias:

  - **Hyperplanes in $\mathbb{R}^2$**: We chose as input space points in the plane $X = \{y \mid y \in \mathbb{R}^2\}$, and as hypothesis space the dichotomies induced by hyperplanes in $\mathbb{R}^2$, that is, $H = \{f_{w,b}(y) = sign(\mathbf{w} \cdot y + b), \mathbf{w} \in \mathbb{R}^2, b \in \mathbb{R}\}$.



In this case the assumption is that examples are linearly separable

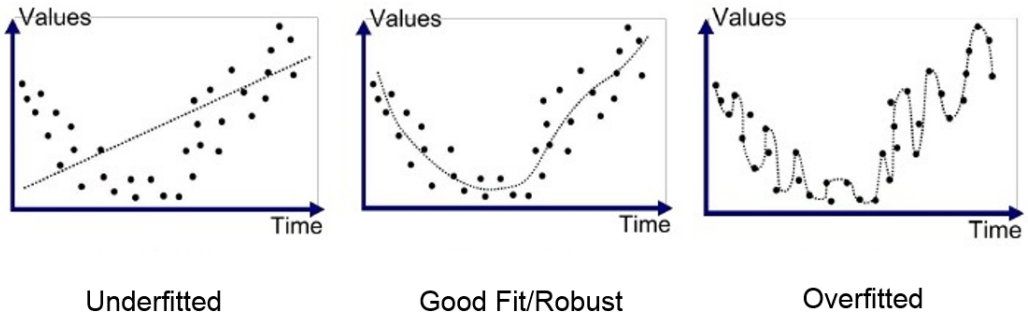
– **Polynomial functions**: Given a training set $S = \{(x_1, y_1), ..., (x_n, y_n)\}, x \in \mathbb{R}, y \in \mathbb{R}$, the hypothesis space is the one containing functions of type: $h_w(x) = w_0 + w_1 x + w_2 x^2 + ... + w_p x^p, p \in \mathbb{N}$. The assumption is on the degree p of the polynomial function.



Bias-Variance Tradeoff: The learning goal is to find the best tradeoff between bias and variance.

– The **bias** error is produced by weak assumptions in the learning algorithm. High bias can cause an algorithm to miss relevant relations between features and target outputs (**underfitting**).

– The **variance** is an error produced by an over-sensitivity to small fluctuations in the training set. High variance can cause an algorithm to model the random noise in the training data, rather than the intended outputs (**overfitting**).



1.2 Example of Learning Algorithm: Perceptron

Consider the space of hyperplanes in $\mathbb{R}^n$, where n is the dimension of the input.

$$H = \{f_{(\mathbf{w},b)}(\mathbf{x}) = sign(\mathbf{w} \cdot \mathbf{x} + b) : \mathbf{w},\mathbf{x} \in \mathbb{R}^n, b \in \mathbb{R}\}$$

where $\mathbf{w}$ is a vector of weights and b is the **bias** term.

We can redefine H as:

$$H = \{f_{\mathbf{w}'}(\mathbf{x}') = sign(\mathbf{w}' \cdot \mathbf{x}') : \mathbf{w}', \mathbf{x}' \in \mathbb{R}^{n+1}\}$$

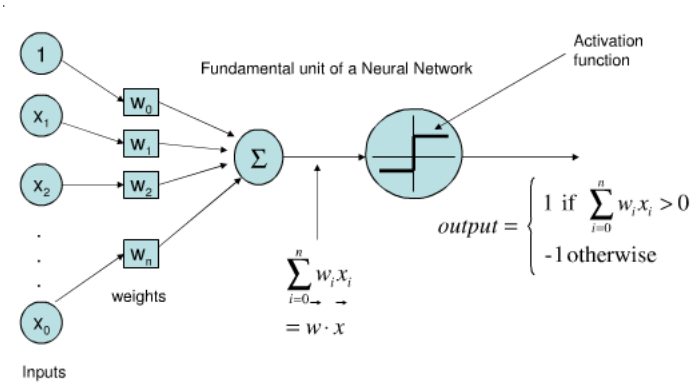
after the following change of variables:

$$\mathbf{w}' = [b, \mathbf{w}], \quad \mathbf{x}' = [1, \mathbf{x}]$$

it follows that:

$$\mathbf{w}' \cdot \mathbf{x}' = b + \sum_{i=1}^{n} \mathbf{w}_i \mathbf{x}_i = \mathbf{w} \cdot \mathbf{x} + b$$

Basically, we add a dimension to $\mathbf{w}$ and $\mathbf{x}$ just to simplify the notation of H.



The model described in the image above is called **Perceptron**. It first computes the dot-product between the weights $\mathbf{w}$ and the input $\mathbf{x}$. The result of this computation is usually called *net*

$$net = \sum_{i=0}^{n} w_i x_i$$

The final output is obtained by applying the **step function** to the *net*.

$$o = \sigma(net) = sign(net)$$

We will refer to this neuron (and associated learning algorithm) as Perceptron.

Since the hypothesis space of the Perceptron is defined as the hyperplanes in $\mathbb{R}^n$, it converges only if the examples in $\mathbb{R}^n$ are **linearly separable**. Otherwise, it will never *find* a hyperplane that separates them.

1.2.1 Perceptron: learning algorithm

Assume to have training examples in $\mathbb{R}^n$ that are linearly separable:

Input: Training set $S = \{(\mathbf{x}, t), \mathbf{x} \in \mathbb{R}^{n+1}, t \in \{-1, +1\}, \eta \geq 0\}$

1. Initialize the value of the weights $\mathbf{w}$ randomly;
2. Repeat (N epochs)
   (a) Select (randomly) one of the training examples $(\mathbf{x}, t)$
   (b) if $o = sign(\mathbf{w} \cdot \mathbf{x}) \neq t$ then
   $$\mathbf{w} \leftarrow \mathbf{w} + \eta(t - o)\mathbf{x}$$

A small value of the learning rate η can make the learning process slow but more stable, that is, it prevents sharp changes in the weights vector. If the training set is linearly separable, it can be shown that the Perceptron training algorithm terminates in a finite number of steps.

Let R be the radius of the **smallest** hyper-sphere centered in the origin enclosing all the instances (how the instances are *spread out*). Let γ be the maximal value such that $t_i net_i = t_i(\mathbf{w} \cdot \mathbf{x}_i) \geq \gamma > 0$ (how much the instances are *separated*). Then, it can be shown that the number of steps of the Perceptron algorithm is bounded from above by the quantity R^2/γ^2. Basically, bigger *distance* between instances means a smaller number of steps to converge and vice versa.

1.3 Model Selection

Model selection is the process used to compare different models and select the optimal one. In particular, model selection can be performed with respect to different values of the hyper-parameters of a fixed model.

There are several methods used to implement it. A first example can be the so called Hold-out procedure. The idea is to obtain a validation set (or hold-out set) Va by splitting the training set Tr. Then, the fixed model is trained using examples in $Tr - Va$, trying different values of the hyper-parameters, and tested against the validation set. This procedure allows you to get an estimate of the error of the model on new unseen data.

Another approach for model selection (and evaluation) is the K-fold cross-validation:

1. The training set is partitioned in k disjointed validation sets $Va_1, ..., Va_k$. For each classifier $h_1, ..., h_k$, we apply the hold-out method on the k-th pair, that is, we train h_i using examples in $Tr - Va_i$ and we test it against V_i.
2. Final error is obtained by individually computing the errors of $h_1, ..., h_k$ on the corresponding validation set and averaging the results.

The above procedure is repeated for different values of the hyper-parameters and the predictor with the smallest final error is selected. The special case where $k = |Tr|$ (the validation sets are made of only one example) is called **leave-one-out** cross-validation.

Chapter 2

Lec 04-05 - Learning With Gradient Descent

2.1 Gradient-based optimization

In machine learning we want to maximize/minimize an **objective function**. This is done by minimizing an **error function**, which is a measure of the committed error. This optimization problem can be solved exploiting the concepts of derivative and **gradient**.

Derivative:
The derivative f' tells us the slope of a function f at any point x.

$$f'(x) = lim_{\epsilon \to 0} \frac{f(x+\epsilon) - f(x)}{\epsilon} \quad \epsilon > 0$$

Given a point:

- Positive derivative means that the function increases at that point
- Negative derivative means that the function decreases at that point.
- Null derivative means that there is a stationary point (minimum, maximum or saddle point).

In order to minimize a function in 1 variable, we have to move in the direction **opposite** to the function derivative.

Gradient is the generalization of derivative with respect to a vector of input variables. In vector calculus, the gradient of a scalar-valued differentiable function $f : \mathbb{R}^n \to \mathbb{R}$ is a vector-valued function $\nabla f : \mathbb{R}^n \to \mathbb{R}^n$ whose value at point p is the vector whose components are the partial derivatives of f at p.

The gradient vector can be interpreted as the direction and rate of fastest increase of the function. The gradient is the zero vector at a point if and only if the point is a stationary point.

2.1.1 Multivariate calculus

Let's consider a function $f : \mathbb{R}^2 \to \mathbb{R}$. It is a function of 2 variables $x_1,\ x_2$ where $x_1(t),\ x_2(t)$ are 2 functions themselves. The derivative of f with respect to t is defined as follows:

$$\frac{d\,f}{d\,t} = \frac{\partial f}{\partial x_1}\frac{\partial x_1}{\partial t} + \frac{\partial f}{\partial x_2}\frac{\partial x_2}{\partial t}$$

We can write the formula above in a more convenient way using vectors multiplications:

$$\begin{bmatrix} \frac{\partial f}{\partial x_1}, & \frac{\partial f}{\partial x_2} \end{bmatrix} \cdot \begin{bmatrix} \frac{\partial x_1(t)}{\partial t} \\ \frac{\partial x_2(t)}{\partial t} \end{bmatrix}$$

Note that the first term is the gradient of f $\nabla_{\vec{x}} f$. Note also that $\frac{\partial x_1(t)}{\partial t}$ is $\nabla_t x_1$, and $\frac{\partial x_2(t)}{\partial t}$ is $\nabla_t x_2$. We can define $X : \mathbb{R} \to \mathbb{R}^2$ as the following vector:

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

where $x_1 : \mathbb{R} \to \mathbb{R}$ and $x_2 : \mathbb{R} \to \mathbb{R}$

Example:
$f(x_1, x_2) = x_1^2 + 2x_2$ where $x_1 = sin(t),\ x_2 = cos(t)$

$$X(t) = \begin{bmatrix} sin(t) \\ cos(t) \end{bmatrix}$$

- $\frac{\partial f}{\partial x_1} = 2x_1$
- $\frac{\partial f}{\partial x_2} = 2$
- $\frac{\partial x_1}{\partial t} = cos(t)$
- $\frac{\partial x_2}{\partial t} = -sin(t)$

$$\begin{aligned} \frac{d\,f}{d\,t} &= 2x_1\, cos(t) + 2\,(-sin(t)) \\ &= 2\, sin(t)cos(t) - 2\, sin(t) \end{aligned}$$

We can do the same thing in matrix notation:

$$\frac{d\,f}{d\,t} = \begin{bmatrix} 2x_1, & 2 \end{bmatrix} \cdot \begin{bmatrix} cos(t) \\ -sin(t) \end{bmatrix} = 2\, sin(t)cos(t) - 2\, sin(t)$$

Example with more than 1 variable:
Given a function $f(x_1, x_2) = x_1^2 + 2x_2$, let $g(s,t)$ be the following function:

$$g(s,t) = \begin{bmatrix} s \cdot sin(t) \\ s \cdot cos(t) \end{bmatrix}$$

where $g_1 = s \cdot sin(t), g_2 = s \cdot cos(t)$. Let h be the composition between f and g:

$$h = f \circ g$$

We want to compute $\frac{d\,h}{d\,(s,t)}$

$$\frac{d\,h}{d\,(s,t)} = \frac{\partial f}{\partial \vec{y}} \cdot \frac{\partial \vec{y}}{\partial (s,t)} = \begin{bmatrix} 2x_1, & 2 \end{bmatrix} \cdot \begin{bmatrix} \frac{\partial g_1}{\partial s}, & \frac{\partial g_1}{\partial t} \\ \\ \frac{\partial g_2}{\partial s}, & \frac{\partial g_2}{\partial t} \end{bmatrix}$$

where $\vec{y} = g(s,t)$

$$\frac{d\,h}{d\,(s,t)} = \begin{bmatrix} 2x_1, & 2 \end{bmatrix} \cdot \begin{bmatrix} sin(t), & s \cdot cos(t) \\ \\ cos(t), & -s \cdot sin(t) \end{bmatrix}$$

The collection of all first-order partial derivatives of a vector-valued function $f : \mathbb{R}^n \to \mathbb{R}^m$ (function g in this case) is called the **Jacobian Matrix**.

2.1.2 Second-order methods

So far, we have discussed gradients, i.e., first-order derivatives. Sometimes, we are interested in derivatives of higher order. Consider a function $f : \mathbb{R}^2 \to \mathbb{R}$ of two variables x, y.We use the following notation for higher-order partial derivatives (and for gradients):

- $\frac{\partial^2 f}{\partial x^2}$ is the second partial derivative of f with respect to x.
- $\frac{\partial^2 f}{\partial x \partial y}$ is the partial derivative obtained by first partial differentiating with respect to x and then with respect to y.

The **Hessian** is the collection of all second-order partial derivative. Generally, for $\mathbf{x} \in \mathbb{R}^n$ and $f : \mathbb{R}^n \to \mathbb{R}$, the Hessian is an $n \times n$ matrix. The Hessian measures the curvature of the function locally around (x, y). If $f : \mathbb{R}^n \to \mathbb{R}^m$ is a vector field, the Hessian is a $(m \times n \times n)$-tensor. In general, Second order derivatives measure the curvature of a function (concave, convex).

$$f: \mathbb{R}^n \to \mathbb{R} \quad , H \in \mathbb{R}^{n \times n}, H_{ij} = \frac{\partial^2}{\partial x_i \partial x_j} f$$

$$H = \begin{array}{|c|c|c|} \hline \frac{\partial^2}{\partial x_1^2} f(\boldsymbol{x}) & \cdots & \frac{\partial^2}{\partial x_1 \partial x_n} f(x) \\ \hline \cdots & & \\ \hline \frac{\partial^2}{\partial x_n x_1} f(\boldsymbol{x}) & & \frac{\partial^2}{\partial x_n^2} f(\boldsymbol{x}) \\ \hline \end{array}$$

The **condition number** is the ratio of the maximum and minimum nonzero eigenvalues of the Hessian matrix. It gives information about the curvature of a function in the different dimensions. Poorly conditioned problems are long, thin valleys (very curved in one direction, very flat in the other).

Second-order optimization algorithm (like Newton's Method) can be used to solve poorly conditioned problems exploiting the Hessian. However, these methods are slow and therefore they are not widely used for deep learning.

2.2 Gradient descent

Let $f(\mathbf{x}) : \mathbb{R}^n \to \mathbb{R}$ be a vector-valued function, find the vector $\theta \in \mathbb{R}^n$ that minimizes the function f:

$$\theta = argmin_{\mathbf{x}} f(\mathbf{x})$$

This minimization problem can be solved using gradient descent. Starting from a random configuration of θ, each parameter is updated in the following way:

$$\theta_{k+1} = \theta_k - \eta \nabla f(\theta_k)$$

where:

- $\nabla f(\theta_k)$ is the partial derivative of the function in θ_k.
- The parameter $\eta > 0$ is known as the *learning rate*.

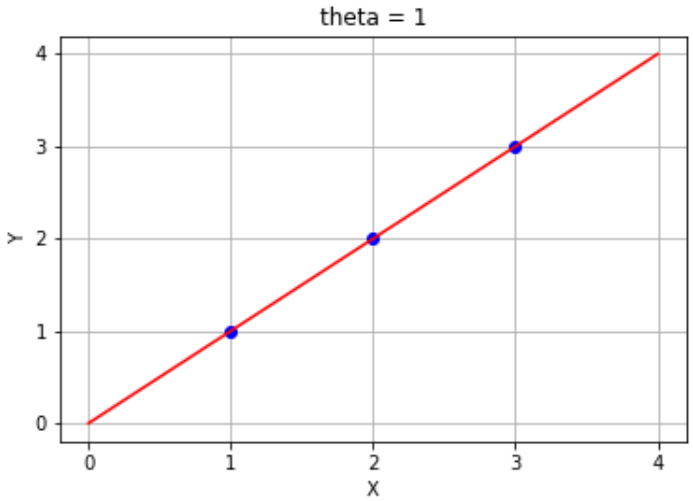
The derivative term $\frac{\partial}{\partial \theta_k} f(\theta_k)$ can be:

- ≥ 0 it means that the function is increasing, so we are decreasing θ_k in the *right direction*.
- ≤ 0 it means that the function is decreasing, so we are increasing θ_k in the *right direction*

If η is too small, gradient descent can be slow. Anyway, if it is too large, it can overshoot the minimum (fail to converge).

2.3 Example

Let's consider a simple example of linear regression with a function $h_\theta(x) = \theta_1 x$ and a cost function $J(\theta_1) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2$.



As you can see from the graph above, our training set is composed by 3 points $\{(1,1),(2,2),(3,3)\}$. In this simple example the function $h_\theta(x)$ that perfectly fits the training set is the one with the parameter $\theta_1 = 1$. In fact, if we compute the cost function $J(\theta_1)$ with respect to $\theta_1 = 1$, the result is 0 (minimized).

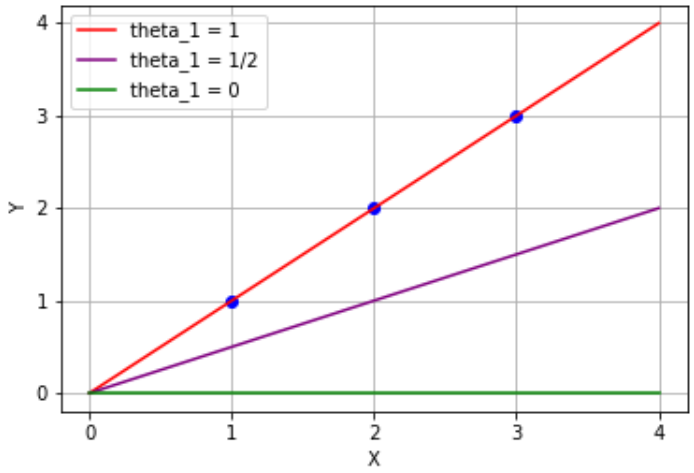
$$J(\theta_1) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2$$

$$= \frac{1}{2m}\sum_{i=1}^{m}(\theta_1 x^{(i)} - y^{(i)})^2$$

$$= \frac{1}{2m}(0+0+0)^2 = 0$$

But let's see what happens for different values of θ_1.

- $\theta_1 = \frac{1}{2}$ $J(\theta_1) \approx 0.67$
- $\theta_1 = 0$ $J(\theta_1) \approx 2.33$
- $\theta_1 = -\frac{1}{2}$ $J(\theta_1) \approx 5.25$



Now we can plot the values of θ_1 on the **X** axis and the values of $J(\theta_1)$ on the **Y** axis. The shape of $J(\theta_1)$ will be the following:

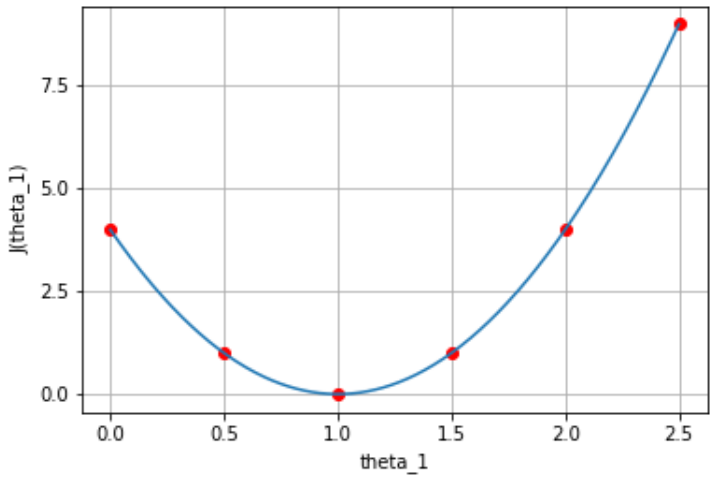
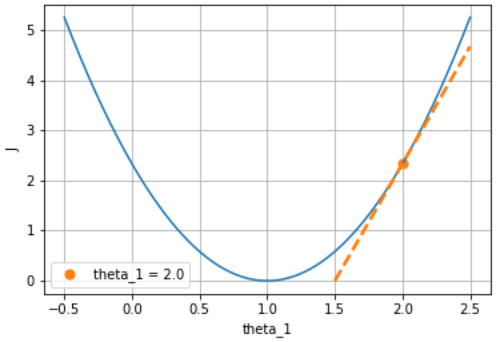
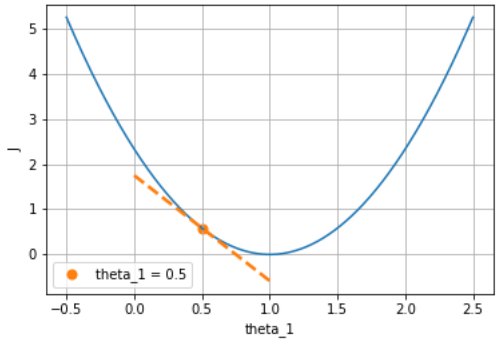
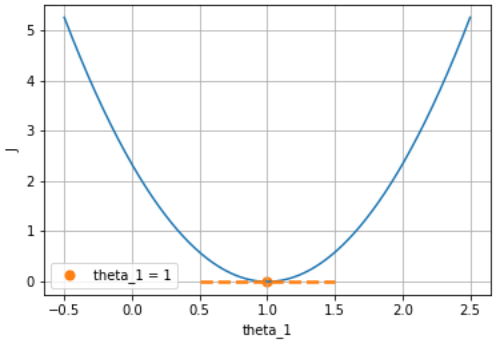


Figure 2.1: Cost function J

We obtain this **convex** function that has its minimum, for this specific $h_{\theta_1}(x^{(i)})$ and y^i, in 0. This principle is also valid for n-dimensional functions but the graphic representation is not that easy.
We can use **gradient descent** technique in order to find θ_1 in an automatic way.







Example of derivation:

We want to minimize $J[\mathbf{w}]$ with respect to the parameters $\mathbf{w}$ using gradient descent. Let's start with the derivation of the loss function $J[\mathbf{w}]$:

$$\frac{\partial J}{\partial w_i} = \frac{\partial}{\partial w_i}\left[\frac{1}{2N}\sum_{s=1}^{N}(t^{(s)} - o^{(s)})^2\right]$$

$$= \frac{1}{2N}\sum_{s=1}^{N}\frac{\partial}{\partial w_i}\left[(t^{(s)} - o^{(s)})^2\right]$$

$$= \frac{1}{2N}\sum_{s=1}^{N}2(t^{(s)} - o^{(s)})\frac{\partial}{\partial w_i}\left[t^{(s)} - o^{(s)}\right]$$

Note that $t^{(s)}$, which is the target value, is a constant that does not depend on w_i. So, it can be removed from the derivation.

$$= \frac{1}{N}\sum_{s=1}^{N}(t^{(s)} - o^{(s)})\left(-\frac{\partial}{\partial w_i}\left[\mathbf{w}\cdot\mathbf{x}^{(s)}\right]\right)$$

$\mathbf{w}\cdot\mathbf{x}^{(s)} = w_1x_1^{(s)} + w_2x_2^{(s)} + ... + w_ix_i^{(s)} + ...$ The partial derivative $-\frac{\partial}{\partial w_i}\left[\mathbf{w}\cdot\mathbf{x}^{(s)}\right]$ with respect to w_i is equal to $x_i^{(s)}$ because all the other terms are constant. Therefore, the derivation becomes:

$$= -\frac{1}{N}\sum_{s=1}^{N}(t^{(s)} - o^{(s)})x_i^{(s)}$$

Chapter 3

Lec 06 - Probability

3.1 Probability - terminology

- **Random variable:** a variable that can take different values randomly

- **Probability distribution:** a description of how likely a random variable x (or a set of random variables) is to take each of its possible states.

  Discrete variables: Probability distribution is described by a **Probability mass function**

  - The domain of P is the set of all possible states of x (k different values).
  - $\forall\, x \in \mathrm{x}\, 0 \leq P(\mathrm{x} = x) \leq 1$
  - $\sum_{x \in \mathrm{x}} P(x) = 1$

  E.g. Uniform distribution $\forall\, x \in \mathrm{x}\, P(\mathrm{x} = x) = \frac{1}{k}$

  Continuous variables: Probability distribution is described by a **Probability Density Function** (PDF)

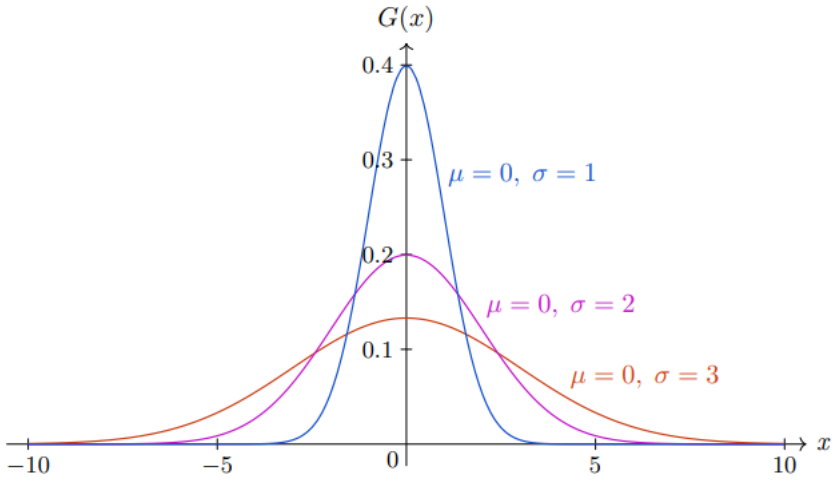
  - The domain of p is the set of all possible states of x
  - $\forall\, x \in \mathrm{x}\, p(x) \geq 0$
  - $\int p(x)dx = 1$

  E.g. Gaussian distribution

- **Joint probability distribution:** Probability distribution over 2 or more variables $P(\mathrm{x} = x, \mathrm{y} = y)$

- **Gaussian distribution**: The one-dimensional Gaussian with mean μ and standard deviation σ has the following shape:

$$G(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$



- **Shannon Entropy (discrete variable):** We can apply information theory to calculate the amount of information there is in an event. This is called *self-information* and can be calculated for a **discrete** event x as follows:

$$I(x) = -log\,(P(x))$$

where $log()$ is the base-2 logarithm and $P(x)$ is the probability of the event x [1].

The choice of the base-2 logarithm means that the units of the information measure is in bits (binary digits). This can be directly interpreted as the number of bits required to represent an event. If the probability that an event occurs is 0.5, we need 1 bit two represent it (0 fail, 1 success); if the event occurs with probability 0.125, we need 3 bits to represent it (remember $y = log_a\,(x) \iff x = a^y$).

The Shannon entropy of a **distribution** is the **expected** amount of information in an event drawn from that distribution:

$$H(\mathrm{x}) = -E_{x\sim P(\mathrm{x})}[log(P(x))] = -\sum_i P(x_i) log\, P(x_i)$$

It provides a lower bound on the number of bits needed **on average** to encode a symbol drawn from the distribution.

Distributions that are nearly deterministic (where the outcome is nearly certain) have low entropy; distributions that are closer to uniform have high entropy.

- **Kullback-Leibler divergence and Cross Entropy:** Let's consider two probability distributions $P(x)$ and $Q(x)$. How can we measure how different they are?
  - Kullback-Leiber divergence:

$$D_{KL}(P\,||\,Q) = E_{x\sim P}\left[log\frac{P(x)}{Q(x)}\right] = E_{x\sim P}[log\,P(x) - log\,Q(x)] = \sum_i P(x_i) log\,\left(\frac{P(x_i)}{Q(x_i)}\right)$$

    It is not a true distance because it is not symmetric:

$$D_{KL}(P\,||\,Q) \neq D_{KL}(Q\,||\,P)$$

[1] other bases can be used, e for example

It is the measure of information lost when Q is used to approximate P.

– Cross Entropy:

$$H(P,Q) = H(P) + D_{KL}(P \,||\, Q) = -E_{x\sim P}[log\, Q(x)]$$

where $H(P)$ is the entropy of P. Note that minimizing the Cross Entropy of P with respect to Q is equivalent to minimize KL divergence between P and Q. (if P is given, $H(P)$ and $E_{x\sim P}[log\, P(x)]$ are constants). If P is a fixed distribution minimize KL divergence or minimize CE is the same, but CE is easier to compute.

- **Maximum likelihood estimation:** Is a principled way to derive estimators (models). Consider n examples $Tr = \{\mathbf{x}^1, ..., \mathbf{x}^n\}$ drawn i.i.d. from p_{data} (which is not known in advance). With machine learning we want to estimate this probability p_{data} with some models that depend on a set of parameters θ.

  Let's consider a family of parametric probability distributions (models) $p_{model}(\mathbf{x};\theta)$. It maps a point $\mathbf{x}$ to a real number, estimating $p_{data}(\mathbf{x})$. How can we find θ in such a way that p_{model} and p_{data} are as close as possible ? A possible formalization of this problem is given by the Maximum Likelihood estimation for θ:

$$\theta_{ML} = argmax_\theta\, p_{model}(Tr;\theta) = argmax_\theta \prod_{i=1}^{n} p_{model}(\mathbf{x}^i;\theta)$$

  Basically, we choose the probability distribution which is most likely to have produced our data. Note that we are assuming that all the examples are **independent** each other ($P(x,y) = P(x)P(y)$).

3.2 Maximum likelihood

Maximum likelihood (ML) is a special case of **maximum a posteriori estimation (MAP)**.

$$h_{MAP} = argmax_{h\in H} P(h|D)$$

$$= argmax_{h\in H} \frac{P(D|h)P(h)}{P(D)}$$

where:

- $P(h)$: a priori probability of the hypothesis h
- $P(D)$: a priori probability of training data. It is the probability to observe exactly this training set when we don't know anything about the hypothesis.
- $P(h|D)$: probability of h given D. It is the probability that h is the hypothesis that generates data D.
- $P(D|h)$: probability if D given h. Given a hypothesis h, it is the probability of data D to be generated by h.

Since $P(D)$ does not depend on h, we can consider it as a constant and remove it from the equation.

$$= argmax_{h\in H} P(D|h)P(h)$$

If we assume uniform probabilities on the hypotheses, that is $P(h_i) = P(h_j)$, we can choose the so called **maximum likelihood hypothesis** h_{ML} :

$$h_{ML} = argmax_{h\in H} P(D|h)$$

MAP and maximum likelihood approach make predictions using a single point estimate of θ. The Bayesian approach is to make predictions using a full probability distribution over θ. For example, given a new instance

x, which is the most likely **classification**? The classification given by the most likely hypothesis h_{MAP} is not necessarily the most likely classification. For example, given the following three possible hypothesis:

$$P(h_1|D) = 0.4 \quad P(h_2|D) = 0.3 \quad P(h_3|D) = 0.3$$

We want to classify a new instance **x**:

$$h_1(\mathbf{x}) = + \quad h_2(\mathbf{x}) = - \quad h_1(\mathbf{x}) = -$$

The most likely hypothesis h_1 classifies **x** with the label (+), but the most likely classification is (-). This is because the optimal (Bayes) classification of a certain instance is the class $v_j \in V$ which maximizes the following probability:

$$argmax_{v_j \in V} = \sum_{h_i \in H} P(v_j|h_i)P(h_i|D)$$

where V is the set of possible labels.

However, in real-world problems having the probabilities $P(h_i|D)$ is almost impossible. Therefore, we usually make the assumption of considering the classification made by h_{map} as most probable.

3.2.1 Maximum likelihood estimation

As we said before, the maximum likelihood estimation is defined as follows:

$$\theta_{ML} = argmax_\theta \, p_{model}(Tr; \theta) = argmax_\theta \prod_{i=1}^{n} p_{model}(\mathbf{x}^i; \theta)$$

However, computing the product of many probability is unstable. Therefore, we can apply the *log* and the *argmax* does not change (log-likelihood).

$$\theta_{ML} = log(argmax_\theta \prod_{i=1}^{n} p_{model}(\mathbf{x}^{(i)}; \theta))$$

$$= argmax_\theta \sum_{i=1}^{n} log \, p_{model}(\mathbf{x}^{(i)}, \theta)$$

We can equivalently divide by n to express maximum likelihood as an expectation over training data:

$$\theta_{ML} = argmax_\theta E_{x \sim \hat{p}_{data}}[log \, p_{model}(x; \theta)]$$

where $\hat{p}_{data}$ is an empirical discrete distribution that we get over the examples in the training set. This implies that maximum likelihood minimizes the dissimilarity between $\hat{p}_{data}$ and p_{model}, measured by the KL divergence. It also corresponds to minimize the **cross-entropy** between the two distributions.

3.2.2 Conditional log likelihood

We can use maximum likelihood to estimate a **conditional** probability $P(\mathbf{y}|\mathbf{x}; \theta)$ to predict **y** given **x** (supervised learning).

$$\theta_{ML} = argmax_\theta P(\mathbf{Y}|\mathbf{X}; \theta)$$

If input examples are i.i.d.

$$\theta_{ML} = argmax_\theta \sum_{i=1}^{n} log \, P(\mathbf{y}^{(i)}|\mathbf{x}^{(i)}; \theta)$$

Consider any real-valued target function f and learning examples $\langle \mathbf{x}_i, d_i \rangle$ where d_i has some noise:

- $d_i = f(\mathbf{x}_i) + e_i$
- e_i is a random variable (noise) extracted independently, for each $\mathbf{x}_i$, according to a Gaussian distribution with mean 0.

It can be shown that the maximum likelihood hypothesis is the one that minimizes the mean squared error:

$$\theta_{ML} = argmin_\theta \sum_{i=0}^{m} (d_i - \hat{y}_i)^2$$

maximizing the log-likelihood with respect to θ yields the same estimate of the parameters θ as does minimizing the mean squared error.

Chapter 4

Lec 07 - Neural Networks I

4.1 Neural Networks

An artificial neuron is a unit that computes a non-linear function over the inputs. Its output depends on the input and on the set of **weights**. These weights have to be learned.

An **Artificial Neural Network** is a system consisting of interconnected units that compute nonlinear (numerical) functions. Adjustable weights are associated with connections among units.

Deep Feed Forward Neural Networks, also called multi-layer perceptrons (MLP), approximate a function f^* that maps an input x to a category y. The MLP defines a mapping $\hat{y} = f(x;\theta)$ where θ is the set of parameters. They are typically represented as a composition of many different functions $f(x) = f^{(3)}(f^{(2)}(f^{(1)}(x)))$. The intermediate layers are called **hidden layers**, while the final layer is called **output layer**.

Multiple layers of cascaded units makes a Neural Network able to implement complex non linear functions. The non linearity of the model is given by the activation functions. In fact, without them (linear activation) the result of the model, even if it's very complex, would still be linear.

Example:
Let's try to define a linear model that predicts the XOR function.

$$Tr = \{([0,0],0),([0,1],1),([1,0],1),([1,1],0)\} \quad f(\mathbf{x};\mathbf{w};b) = \mathbf{x}\mathbf{w}^T + b$$

The XOR function **cannot** be learned by any linear classifier. In order to solve this problem, we can define a two-layers Neural Network with the addition of the ReLU activation function $ReLU(y) = max(0,y)$. The networks becomes:

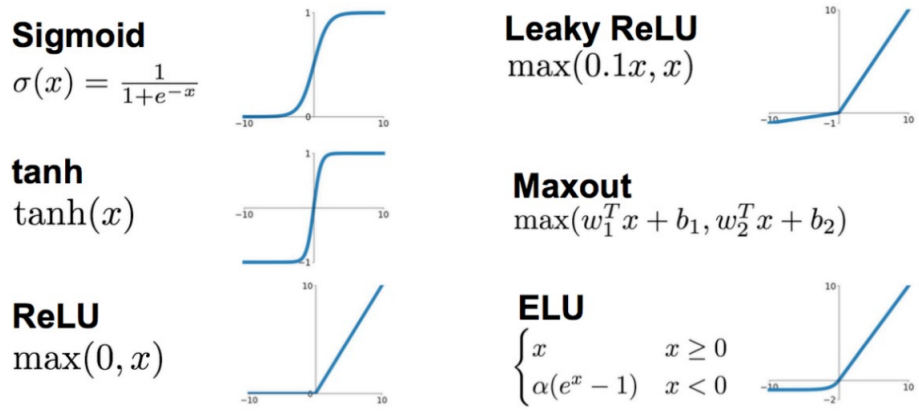
$$f(\mathbf{x};\mathbf{W},\mathbf{c},\mathbf{w},b) = \mathbf{w}^T\, max(0,\mathbf{W}\mathbf{x}^T + \mathbf{c}) + b$$

the following parameters values provide a solution to the XOR problem:

$$\mathbf{W} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix},\ \mathbf{c} = \begin{bmatrix} 0 \\ -1 \end{bmatrix},\ \mathbf{w} = \begin{bmatrix} 1 \\ -2 \end{bmatrix},\ b = 0$$

Note that the first hidden layer has two nodes.

The most common activation functions are the following:



In general, the goal is to have a more complex decision surface. We can achieve this goal by using non-linear activation functions and stacking several hidden layers. In the XOR example above, the initial points are mapped by the first hidden layer into a new space where they are linearly separable. Thanks to this, the output linear layer is able to classify the points correctly.

4.2 Learning in a Neural Network

In general, the problem of how to update the weights of a model in order to minimize the committed error is called **credit assignment problem**. A possible solution is to make a single neuron **derivable** and exploit gradient descent technique to learn the *right* weights. In order to make a neuron derivable, its activation function must be derivable.

Linear models (e.g. SVM) are formulated as **convex models**. However, the non-linearity in neural networks makes the problem **non-convex**. It means that there is no guarantee of achieving the global optimum. Furthermore, the way in which the weight of a NN are initialized has a strong impact on the solution that the model will find.

4.3 Cost Function

A important aspect of the design of a deep neural network is the choice of the cost function. In most cases, the model defines a distribution $p(\mathbf{y}|\mathbf{x};\theta)$ and the cost function is the cross-entropy between training labels and network predictions (negative log-likelihood):

$$J(\theta) = -E_{(x,y)\sim\hat{p}_{data}} log\, p_{model}(\mathbf{y}|\mathbf{x})$$

The specific form of the cost function changes from model to model. The output representation $\mathbf{h} = f(\mathbf{x};\theta)$ determines the form of the cross-entropy function.

- **Linear Regression:** If we assume that the target values are distributed according to a Gaussian distribution G, we can think our output layer as to produce the mean of G.

  $$\hat{\mathbf{y}} = \mathbf{W}^T\mathbf{h} + \mathbf{b}$$

  $$G(\mathbf{y};\hat{\mathbf{y}};\mathbf{I})$$

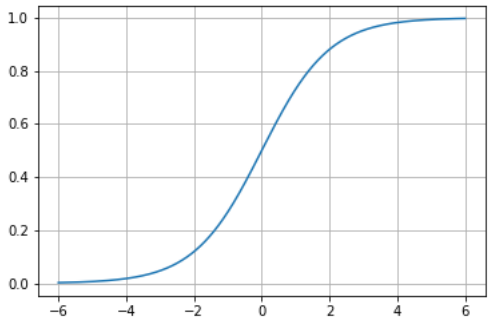
  As we already seen, in this case maximizing the likelihood corresponds to minimize the mean squared error.

  $$J(\theta) = \frac{1}{2}E_{p\sim\hat{p}_{data}}(y - f(\mathbf{x};\theta))^2 + constant$$

Basically, this is a motivation of why the mean squared error cost function is suitable for linear regression.

- **Binary classification:** In this case we assume that our targets are distributed according to a Bernoulli distribution. This distribution depends on a single parameter $\phi \in [0, 1]$:
  - $P(\mathrm{x} = 1) = \phi$
  - $P(\mathrm{x} = 0) = 1 - \phi$

  In general:
  $$P(\mathrm{x} = x) = \phi^x (1 - \phi)^{1-x}$$
  The NN just needs to predict $P(\mathrm{y} = 1|\mathbf{x})$. Therefore, we want to force this number to lie in $[0, 1]$. For example, we can use the following function:
  $$P(\mathrm{y} = 1|\mathbf{x}) = max(0, min(1, \mathbf{w}^T \mathbf{h} + b))$$
  However, this is not a good choice for gradient descent. In fact, if the output is outside $[0, 1]$ the function is not derivable and the gradient will always be 0. We want to ensure that there is always some gradient when the model is wrong. An alternative way to force the output to lie in $[0, 1]$ is to use an **output** linear layer with a **sigmoid activation function**.
  $$\hat{y} = \sigma(\mathbf{w}^T \mathbf{h} + b)$$
  where σ is a Sigmoid or Logistic function:
  $$\sigma(x) = \frac{1}{1 + e^{-x}}$$



  An output unit is composed of two components:
  - $z = \mathbf{w}^T \mathbf{h} + b$
  - $\sigma(\cdot)$: the activation function to **convert** z **into a probability** (in order to apply maximum likelihood optimization).

  How can we define a probability distribution over y using the value z ? The sigmoid can be motivated by constructing an unnormalized probability distribution $\hat{P}(y)$, which does not sum to 1. We assume that the unnormalized probabilities are linear in y and z.
  $$log\, \hat{P}(y) = yz \quad i.e.\ log\, \hat{P}(y = 1) = z,\ log\, \hat{P}(y = 0) = 0$$

We can exponentiate to obtain the unnormalized probabilities:

$$\hat{P}(y) = e^{yz}$$

Then, we normalize to obtain a proper probablity:

$$P(y) = \frac{e^{yz}}{\sum_{y'=0}^{1} e^{y'z}} = \sigma((2y-1)z)$$

where:

- $P(y=1) = \frac{1}{1+e^{-z}}$
- $P(y=0) = \frac{1}{1+e^{z}}$

As we already seen, the cost function used with maximum likelihood is the negative-log likelihood. Therefore, the loss function for maximum likelihood learning of a Bernoulli parametrized by a sigmoid is:

$$J(\theta) = -log\, P(y|\mathbf{x}) = -log\, \sigma((2y-1)z) = \zeta((1-2y)z)$$

where $\zeta(x) = log(1 - exp(x))$ is called **softplus** function.

This approach to predicting the probabilities in log-space is natural to use with maximum likelihood learning. The log in the cost function undoes the exp of the sigmoid. Without this effect, the saturation of the sigmoid could prevent gradient- based learning from making good progress.

The saturation (i.e. when the gradient is very small) occurs when $y = 1$ and z is very positive or $y = 0$ and z is very negative, that is, when the model has the right answer.

With other cost functions, such as MSE, we'll be able to find a solution but it would **not** be the maximum-likelihood solution.

- **Multi-class classification:** In this case the output has to be a probability distribution over a discrete variable with n possible values. We have to generate a vector $\hat{\mathbf{y}} = [\hat{y}_0, \hat{y}_1, ..., \hat{y}_{n-1}]$ where:
  - $\hat{y}_i = P(y = i|\mathbf{x})$
  - $\forall i,\ 0 \leq \hat{y}_i \leq 1$
  - $\sum_i \hat{y}_i = 1$

  We can use the same approach for the Bernoulli distribution generalized to the **Multinoulli distribution**:

  $$\mathbf{z} = \mathbf{W}^T\mathbf{h} + \mathbf{b} \quad \text{where } z_i = log\, \hat{P}(y = i|x)$$

  In order to represent the probability distribution over n different classes, we can use the **Softmax** function, which is a generalization of sigmoid.

  $$softmax(\mathbf{z})_i = \frac{e^{z_i}}{\sum_j^n e^{z_j}}$$

  By applying the log-likelihood, the cost function is:

  $$log\, softmax(\mathbf{z})_i = z_i - log \sum_j e^{z_j}$$

  z_i pushes the correct labels up and $log \sum_j e^{z_j}$ pushes the uncorrect labels down. When we perform the prediction, we'll choose the argmax of $\hat{\mathbf{y}}$.

4.3.1 Output functions in general

Linear, sigmoid and softmax output units are the most common, but NN can generalize to almost any kind of output layer.

Maximum likelihood provides a guide to design almost any output layer.

1. We define a conditional distribution $p(\mathbf{y}|\mathbf{x};\theta)$
2. As cost function, maximum likelihood suggest to use $-log\, p(\mathbf{y}|\mathbf{x};\theta)$

We can think of the NN as $f(\mathbf{x};\theta) = \omega$ where ω are the parameters of a distribution over $\mathbf{y}$ and the cost function is $-log\, p(\mathbf{y};\omega(\mathbf{x}))$

4.4 Hidden units

An hidden unit can be described as accepting an input x, computing $z = W^T x + b$, and applying an element-wise nonlinear function $g(z)$. The design of hidden units does not have many definitive guiding theoretical principle. What we can do is to evaluate hidden units performance on a validation set. To select the most suitable activation function we can rely on some basic intuitions motivating each type of hidden unit.

4.4.1 Hidden units: ReLU

An hidden unit with ReLU activation function is defined as follows:

$$g(z) = max(0, z) \quad z = f(\mathbf{W}^T\mathbf{x} + \mathbf{b})$$

Such hidden units are similar to linear units and therefore easier to optimize. However, ReLU units do not learn via gradient-based methods on examples for which their activation is zero. In order to solve this problem, generalizations of ReLU have been defined (e.g. leakyReLU) $g(z,\alpha)_i = max(0, z_i) + \alpha_i\, min(0, z_i)$.

4.4.2 Hidden units: Tanh and sigmoid

Other common activation functions are Tanh and sigmoid:

- Logistic sigmoid: $g(z) = \sigma(z)$
- Hyperbolic Tangent: $g(z) = tanh(z) = 2\sigma(2z) - 1$

Actually, these two functions are not a very good choice for the hidden layers since they saturate across most of their domain. In particular, they saturate to high value when z is very positive, and to low value when z is very negative. This is good for output units, as we seen before, but not for hidden units, because in hidden layers we just want to keep learning without necessarily having a value between 0 and 1. Tanh typically performs better than the logistic sigmoid.

In general, many differentiable functions are reasonable (e.g. $cos(x)$), but they show no significant advantage over common ones.

4.5 Architecture

The architecture of a NN is its overall structure. Neural networks are generally organised in layers where each layer is a function of the preceding one:

$$\mathbf{h}^{(1)} = g^{(1)}(\mathbf{W}^{(1)T}\mathbf{x} + \mathbf{b}^{(1)})$$

$$\mathbf{h}^{(i)} = g^{(i)}(\mathbf{W}^{(i)T}\mathbf{h}^{(i-1)} + \mathbf{b}^{(i)})$$

The main architectural considerations are:

- The depth of the network
- The width of each layer

Deeper networks tend to generalize better, but they are harder to train.

All these hyper-parameters should be validated on a validation set.

Universal approximation Theorem

Given a feed-forward NN with just one hiddel layer, any continuous function $f : \mathbb{R}^n \to \mathbb{R}$ and an arbitrarily small $\epsilon > 0$, then, for a large class of activation functions, there always exists an integer M such that the function $g : \mathbb{R}^n \to \mathbb{R}$ computed by the net using at least M hidden units approximates the function f with tolerance ϵ, that is:

$$max_{x \in \Omega} |f(\mathbf{x}) - g(\mathbf{x})| < \epsilon$$

Note that the theorem attests the existence of a NN with M hidden units that approximates any continuous function with the desired tolerance, but it says nothing about how M can be computed and how large this network would be. Furthermore, we are not guaranteed that the training algorithm will be able to learn it (the optimization algorithm may not be able to find the value of the parameters that corresponds to the desired function).

Using deeper models can reduce the number of units required to represent the desired function. Furthermore, greater depth does seem to result (empirically) in better generalization for a wide variety of tasks.

Many neural networks architectures have been developed for specific tasks, e.g. Convolutional Neural Networks (CNN) or Recurrent Neural Network (RNN). In general, the layers need to be connected in a chain:

- Skip connections: Make it easier for the gradient to flow from output layers to layers nearer the input.
- Sparse connections: Each unit in a layer is connected to only a small subset of units in the next layer.

Chapter 5

Lec 8 - Backpropagation

5.1 Learning algorithm

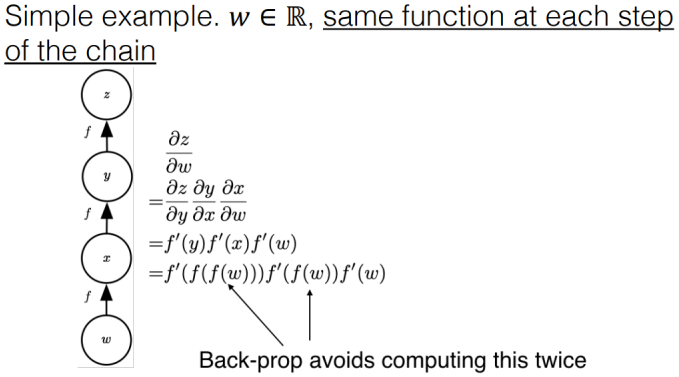
The basic idea of the learning algorithm consists in two phases:

- **Forward phase:** for each example in the training set, present it to the network and compute the output.
- **Backward phase:** Back-propagate the committed error by computing the gradients of the cost function with respect to the network's weights.

Backpropagation is the algorithm used to **compute the gradient** of the loss function with respect to each parameter. The information from the cost function flows backward to compute these gradients. Then, another algorithm performs learning using the gradients (e.g. stochastic gradient descent):

- **Batch gradient descent:** It **cumulates** gradients over all the training examples and then updates the weights.
- **Stochastic gradient descent:** For each example in S, it computes the gradients and update the weights.
- **Mini-batch gradient descent:** It updates the weights considering a subset of examples $Q \subseteq S$.

Basically, Backpropagation is a particular implementation of the chain rule of calculus (more efficient)



5.1.1 Backpropagation for (sigmoid) Perceptron

Let's consider a Perceptron with sigmoid activation function:

$$out(\vec{x}) = \sigma\left(\sum_{i=0}^{n} w_i x_i\right) = \sigma(\vec{w}\cdot\vec{x})$$

where $\sigma(net) = \frac{1}{1+e^{-net}}$. Note that for $\sigma()$ the following relation holds:

$$\sigma'(net) = \frac{d\,\sigma(net)}{d\,net} = \sigma(net)(1-\sigma(net))$$

Given a training set $Tr = \{(\mathbf{x}^{(1)}, y^{(1)}), ..., (\mathbf{x}^{(n)}, y^{(n)})\}$ and MSE cost function $E = \frac{1}{2N}\sum_{d\in Tr}(t^{(d)} - out^{(d)})^2$.

Gradient computation for sigmoidal units:

$$\begin{aligned}
\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i}\frac{1}{2N_{Tr}}\sum_{d\in Tr}(t^{(d)} - out^{(d)})^2 \\
&\text{Consider a single output neuron} \\
&= \frac{1}{2N_{Tr}}\sum_{d\in Tr}\frac{\partial}{\partial w_i}(t^{(d)} - out^{(d)})^2 \\
&\text{Derivative of power} \\
&= \frac{1}{2N_{Tr}}\sum_{d\in Tr} 2(t^{(d)} - out^{(d)})\frac{\partial}{\partial w_i}(t^{(d)} - out^{(d)}) \\
&= \frac{1}{N_{Tr}}\sum_{d\in Tr}(t^{(d)} - out^{(d)})\frac{\partial}{\partial w_i}(t^{(d)} - \sigma(\vec{w}\cdot\vec{x}^{(d)})) \\
&\text{Chain rule} \\
\frac{\partial E}{\partial w_i} &= \frac{1}{N_{Tr}}\sum_{d\in Tr}(t^{(d)} - out^{(d)})(-\frac{\partial\sigma(\vec{w}\cdot\vec{x}^{(d)})}{\partial\vec{w}\cdot\vec{x}^{(d)}}\frac{\partial\vec{w}\cdot\vec{x}^{(d)}}{\partial w_i}) \\
&= -\frac{1}{N_{Tr}}\sum_{d\in Tr}(t^{(d)} - out^{(d)})\sigma(\vec{w}\cdot\vec{x}^{(d)})(1-\sigma(\vec{w}\cdot\vec{x}^{(d)}))x_i^{(d)}
\end{aligned}$$

5.1.2 Gradient descent for Feed-forward Networks

Let's first define some terminology:

- d **input units**: d input data size $\mathbf{x} \equiv (x_1, ..., x_d)$, $d+1$ when including the bias in the weight vector $\mathbf{x}' \equiv (x_0, x_1, ..., x_d)$
- n_H **hidden units** (with output $\mathbf{y} \equiv (y_1, ..., y_{n_H})$)
- c **output units**: $\mathbf{z} \equiv (z_1, ..., z_c)$. The number of desired output is $\mathbf{t} \equiv (t_1, ..., t_c)$
- w_{ji} weight from the i-th input unit to the j-th hidden unit (w_j is the weight vector of the j-th hidden unit)
- w_{kj} weight from the j-th hidden unit to the k-th output unit (w_k is the weight vector of the k-th output unit)

Let's consider, for example, a Neural Network with just one hidden layer and **sigmoid activation functions**. We need to minimize an error function $E[\mathbf{w}]$ that can be defined as follows:

$$E[\mathbf{w}] = \frac{1}{2cN}\sum_{s=1}^{N}\sum_{k=1}^{c}(t_k^{(s)} - z_k^{(s)})^2$$

We need to first compute the gradient for the weights of both output and hidden units.

- **Gradient of the weights of an output unit:**

$$\frac{\partial E}{\partial w_{\hat{k}\hat{j}}} = -\frac{1}{cN}\sum_{s=1}^{N}(t_{\hat{k}}^{(s)} - z_{\hat{k}}^{(s)})z_{\hat{k}}(1-z_{\hat{k}})y_{\hat{j}}^{(s)}$$

  where $y_{\hat{j}}^{(s)}$ is the output of the $\hat{j}$-th hidden unit.

- **Gradient of the weights of a hidden unit:**

$$\frac{\partial E}{\partial w_{\hat{j}\hat{i}}} = -\frac{1}{cN}\sum_{s=1}^{N} y_{\hat{j}}^{(s)}(1-y_{\hat{j}}^{(s)})x_{\hat{i}}^{(s)}\sum_{k=1}^{c}(t_{\hat{k}}^{(s)} - z_{\hat{k}}^{(s)})z_{\hat{k}}(1-z_{\hat{k}})w_{k\hat{j}}$$

Back-propagation (stochastic)

1. Initialize all weights with small random values (e.g. between -0.5 and +0.5)
2. Until the termination condition is satisfied:
   (a) For each $(\mathbf{x}, \mathbf{t}) \in S$:
      i. Present $\mathbf{x}$ to the net and compute the vectors $\mathbf{y}$ and $\mathbf{z}$
      ii. For each output unit k:

$$\delta_k = z_k(1-z_k)(t_k - z_k)$$

$$\Delta w_{kj} = \delta_k y_j$$

      iii. For each hidden unit j:

$$\delta_j = y_j(1-y_j)\sum_{k=1}^{c} w_{kj}\delta_k$$

$$\Delta w_{ji} = \delta_j x_i$$

      iv. Update all weights:

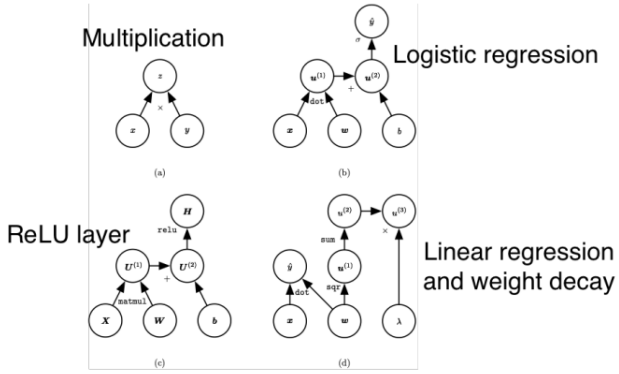
$$w_{sq} \leftarrow w_{sq} + \eta\Delta w_{sq}$$

Note that this algorithm works only for a network with one hidden layer, but can be easily extended. In fact, the algorithm computes the error term δ for each unit of each layer, and then it multiplies this error term for the input of the unit.

5.2 Computational Graph

A computational graph is a way to formalize the structure of a set of computations, such as those involved in mapping inputs and parameters to outputs and loss. Each node in the graph indicates a variable. The variable may be a scalar, vector, matrix, tensor. The edges encode operations, which are simple functions of one or more variables. Without loss of generality, we define an operation to return only a single output variable. This does not lose generality because the output variable can have multiple entries, such as a vector.

If a variable y is computed by applying an operation to a variable x, then we draw a directed edge from x to y.


Multiplication
Logistic regression
ReLU layer
Linear regression
and weight decay

Chapter 6

Lec 09-10 - Regularization

6.1 Regularization

Neural networks are usually *over-parameterized*, that is, they have more parameters than training examples. This implies that the set of parameters can perfectly fit the training data, including the noise (overfitting). In order to limit overfitting, we can perform **regularization**, that is, any modification we make to a learning algorithm intended to reduce its generalization error but not its training error. Basically, regularization aims at reducing the **variance** error (favour simpler hypothesis).

Oldest regularization strategies (adopted for linear/logistic regression) limit the capacity of the models, adding a parameter **norm penalty** to the **objective function**

$$\tilde{J}(\theta; \mathbf{X}; \mathbf{y}) = J(\theta; \mathbf{X}; \mathbf{y}) + \alpha\Omega(\theta)$$

where Ω is the so called **regularization term** that depends on the model's parameters. Larger values of the hyper-parameter α results in more regularization. This is because if α is very high, the optimization will favour the second term more than the first and viceversa. In general, only weights are regularized (bias terms tend to be easy to learn).

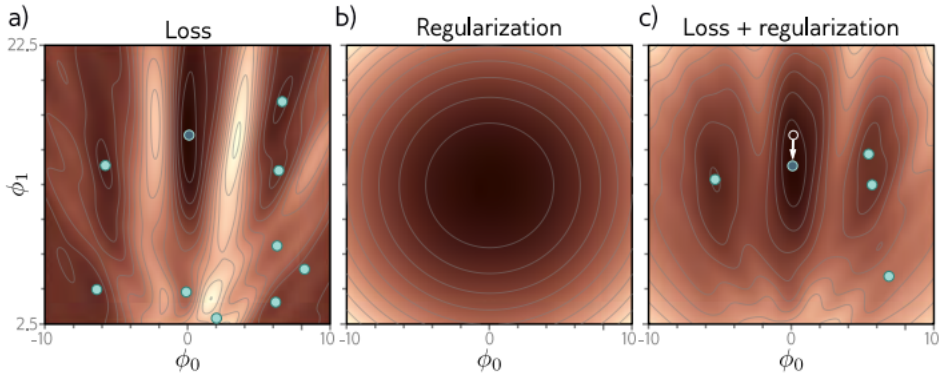


Figure 9.1 Explicit regularization. a) Loss function for Gabor model (see section 6.1.2). Cyan circles represent local minima, green circle represents the global minimum. b) The regularization term favors parameters close to the center of the plot by adding an increasing penalty as we move away from this point. c) The final loss function is the sum of the original loss function plus the regularization term. This surface has fewer local minima, and the global minimum has moved to a different position (arrow shows change).

How can we set this regularization term?

6.2 Weight decay (L^2 norm)

The idea is to drive the weights close to the origin:

$$\Omega(\theta) = \frac{1}{2}||w||_2^2$$

where $||w||_2^2$ is the 2-norm of the weight vector ($w^T w$).

So our new cost function will become:

$$\tilde{J}(\mathbf{w}; \mathbf{X}, \mathbf{y}) = \frac{\alpha}{2}\mathbf{w}^T\mathbf{w} + J(\mathbf{w}; \mathbf{X}, \mathbf{y})$$

Note that, for simplicity, we are assuming no bias.

Then, we have to compute the gradient of the new cost function with respect to $\mathbf{w}$:

$$\nabla_{\mathbf{w}}\tilde{J}(\mathbf{w}; \mathbf{X}, \mathbf{y}) = \alpha\mathbf{w} + \nabla_{\mathbf{w}}J(\mathbf{w}; \mathbf{X}, \mathbf{y})$$

A single step of stochastic gradient descent with learning rate ϵ would be:

$$\mathbf{w} \leftarrow \mathbf{w} - \epsilon(\alpha\mathbf{w} + \nabla_{\mathbf{w}}J(\mathbf{w}; \mathbf{X}, \mathbf{y}))$$

$$\mathbf{w} \leftarrow (1 - \epsilon\alpha)\mathbf{w} - \epsilon\nabla_{\mathbf{w}}J(\mathbf{w}; \mathbf{X}, \mathbf{y})$$

Basically, we are multiplying the weights vector for a value strictly smaller than 1, so it becomes smaller every SGD step. The weights get shrinked on each step (weight decay). This approach is based on the fact that large weights lead to amplifications of the differences between similar inputs, making the network sensitive, and hence overfitting, to its training data. In contrast, small weights reduce the differences between similar inputs, and hence provides improved generalization.

6.2.1 Ordinary Least squares

Regularization is necessary in some ill-posed ML problems:

- When we have less examples than features
- Or more in general when the solution is not unique

Let's consider linear regression:

$$\mathbf{X}\mathbf{w} = \mathbf{y}$$

Let's define MSE loss:

$$J(\mathbf{w}) = \frac{1}{n}||\mathbf{X}\mathbf{w} - \mathbf{y}||^2$$

The gradient of J with respect to $\mathbf{w}$ is:

$$\nabla_{\mathbf{w}} J = \frac{1}{n} 2(\mathbf{X}\mathbf{w} - \mathbf{y})^T \mathbf{X}$$

Now, we can define the closed-form solution equating the derivative to zero:

$$\frac{1}{n} 2(\mathbf{X}\mathbf{w} - \mathbf{y})^T \mathbf{X} = 0$$
$$\mathbf{w} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$$

Note that $(\mathbf{X}^T\mathbf{X})^{-1}$ should be invertible, that is, a square full-rank matrix. $\mathbf{X}^T\mathbf{X}$ is always square, but it is not invertible when the number of variables exceeds the number of data points.

With MSE loss and L^2 norm, the gradients would be:

$$\nabla_{\mathbf{w}} J = 2(\mathbf{X}\mathbf{w} - \mathbf{y})^T \mathbf{X} + 2\alpha \mathbf{w}^T$$

Therefore, the closed-form solution will be:

$$2(\mathbf{X}\mathbf{w} - \mathbf{y})^T \mathbf{X} + 2\alpha \mathbf{w}^T = 0$$
$$\mathbf{w} = (\mathbf{X}^T\mathbf{X} + \alpha\mathbf{I})^{-1}\mathbf{X}^T\mathbf{y}$$

$(\mathbf{X}^T\mathbf{X} + \alpha\mathbf{I})^{-1}$ is a full-rank matrix and therefore always invertible. The diagonal entries of this matrix correspond to the variance of each input feature, and we add α on this diagonal. We can see that L^2 regularization causes the learning algorithm to "perceive" the input $\mathbf{X}$ as having higher variance, which makes it shrink the weights on features whose covariance with the output target is low compared to this added variance.

6.3 L^1 Regularization

In L^1 regularization, Ω and $\tilde{J}$ are defined as follows:

$$\Omega(\theta) = ||w||_1$$
$$\tilde{J}(\mathbf{w}; \mathbf{X}, \mathbf{y}) = \alpha||w||_1 + J(\mathbf{w}; \mathbf{X}, \mathbf{y})$$

With gradient $\nabla_{\mathbf{w}}\tilde{J}(\mathbf{w}; \mathbf{X}, \mathbf{y}) = \alpha\, sign(\mathbf{w}) + \nabla_{\mathbf{w}} J(\mathbf{w}; \mathbf{X}, \mathbf{y})$ where $sign(\cdot)$ is applied element-wise. This technique forces the solution to be sparse (i.e. lot of weights to 0 and only few of them $\neq 0$). Therefore, it performs **feature selection**.

This sparsity property can be seen looking at the gradients formula:

$$\nabla_{\mathbf{w}}\tilde{J}(\mathbf{w}; \mathbf{X}, \mathbf{y}) = \alpha\, sign(\mathbf{w}) + \nabla_{\mathbf{w}} J(\mathbf{w}; \mathbf{X}, \mathbf{y})$$

If w_i is negative, then the term $\alpha\, sign(w_i)$ will be negative and viceversa. Therefore, when we'll perform the update by subtracting the gradients, w_i will be pushed in the opposite direction of its sign by α.

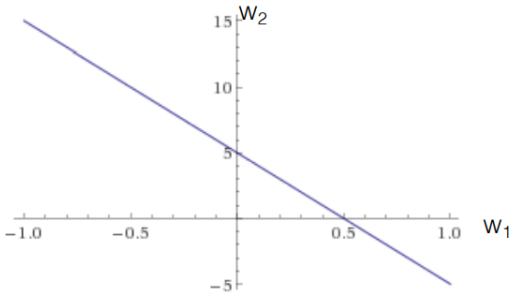
6.3.1 Geometric interpretation

Consider a linear regression:

$$\mathbf{Xw} = \mathbf{y}$$

Given an example in the training set, e.g. $([10, 1], 5)$, we have infinitely many solutions satisfying:

$$10w_1 + w_2 = 5$$
$$w_2 = 5 - 10w_1$$

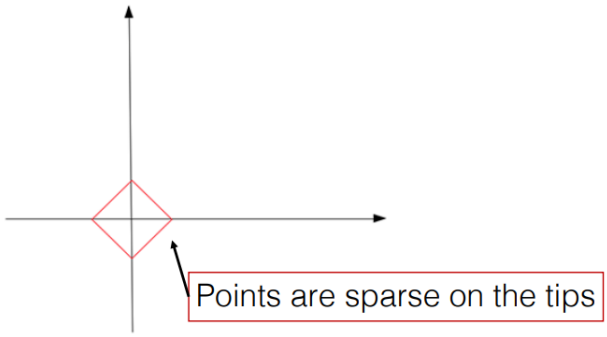


This family of solutions (which are infinite) lies on the blue line in the w_1, w_2 space represented in the figure above.

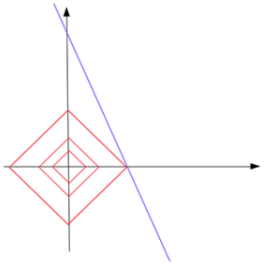
The L^1 norm is defined as the sum of absolute values:

$$||\mathbf{w}||_1 = \sum_i |w_i|$$

In the w_1, w_2 space defined previously, all the points with constant L^1 norm c lies on a square.



If we want to find the solution for $w_2 = 5 - 10w_1$ with minimum L^1 norm, we have to look at the point where it intersects the square.



This intersection will probably be on one of the axis, that is, the solution with minimum L^1 norm will **probably** be sparse. This is not always true, but in high-dimensional spaces it is likely.

In the case of L^2 norm, instead of a square, the points with constant norm will make a circle (in 2-dimensions). L^2 does not enforce sparsity because the solution with minimum norm can be at any point on the circle. Therefore, L^2 causes all the weights to be shrunk towards zero, but none of them are driven to exactly zero.

6.4 Data augmentation

The idea of data augmentation is to generate *fake* training data in order to enrich the training set. For example, in object detection on images, we want the model to be invariant to rotation, translation or scaling; therefore, we can apply some transformations on the training instances in order to achieve this goal.

Another way to perform data augmentation is to inject noise in:

- the input examples (e.g. denoising autoencoders)
- the hidden representations (e.g. droput)
- the target (e.g. label smoothing). Consider a multi-class classification problem. Since the softmax will never give exactly 1 or 0, there will always be a gradient that makes the weights bigger. Label smoothing regularizes a model based on a softmax with k output values by replacing the hard 0 and 1 classification targets with targets of $\frac{\epsilon}{k-1}$ and $1-\epsilon$, respectively.

  This technique is also useful if the dataset has some amount of mistakes in the y labels.

6.5 Semi-supervised learning

Let's say that we have a problem for which it's easy to gather the data but it's very difficult to set labels for those data. In the paradigm of semi-supervised learning, both unlabeled examples from $P(\mathbf{x})$ and labeled examples from $P(\mathbf{x}, \mathbf{y})$ are used to estimate $P(\mathbf{y}|\mathbf{x})$ or predict $\mathbf{y}$ from $\mathbf{x}$.

We learn a representation $h = f(\mathbf{x})$ such that similar examples are close in the new space. A linear classifier in the new space may achieve better generalization. Unsupervised learning can provide useful cues for how to group examples in representation space.

Instead of having separate unsupervised and supervised components in the model, one can construct models in which a generative model of $P(\mathbf{x})$ (autoencoder) shares parameters with a discriminative model of $P(\mathbf{y}|\mathbf{x})$.

6.6 Transfer learning

When training data is limited, other datasets can be exploited to improve performance. Transfer learning means transferring information from previously learned tasks for the learning of new tasks. Basically, we can take a model that was trained on a similar task and adapt it to our problem. In the case of Neural Networks, we can do this by removing the last layer of the network and adding one or more layers in order to produce a suitable output. The main model may be fixed and the new layers trained for the new task, or we may fine-tune the entire model.

Transfer learning can be viewed as initializing most of the weights of the final network in a sensible part of the space that is likely to produce a good solution.

6.7 Multi-Task learning

Multi-task learning (MTL) is a subfield of machine learning in which multiple learning tasks are solved at the same time. Part of the model is shared across different tasks, and is driven to learn a representation that generalizes well.

6.8 Self-supervised learning

When no data from other tasks is available, we can create large amounts of *free* labeled data using self-supervised learning and use this for transfer learning. For example:

- **generative** self-supervised learning: part of each data example is masked and the learning task is to predict the missing part.
- **contrastive** self-supervised learning: two versions of each unlabeled example are presented, when one has been distorted in some way. The system is trained to predict which is the original.

6.9 Early stopping

When training large models with sufficient representational capacity to overfit the task, we often observe that training error decreases steadily over time, but validation set error begins to rise again. This means we can obtain a model with better validation set error (and thus, hopefully, better test set error) by returning the model's parameters at the point in time with the lowest validation set error. When the training algorithm terminates, we return these parameters, rather than the latest parameters. The training algorithm terminates when no parameters have improved over the best recorded validation error for some pre-specified number of iterations.

Early stopping acts as a regularizer because it has the effect of restricting the optimization procedure to a relatively small volume of parameter space in the neighborhood of the initial parameter values.

6.10 Parameter Tying and sharing

Sometimes we might not know precisely what values the parameters should take but we know, from knowledge of the domain and model architecture, that there should be some dependencies between the model parameters.

A common type of dependency that we often want to express is that certain parameters should be close to one another. Consider the following scenario: we have two models performing the same classification task (with the same set of classes) but with somewhat different input distributions. Let us imagine that the tasks are similar enough that we believe the model parameters should be close to each other. We can leverage this information through regularization. Specifically, we can use a parameter norm penalty of the form:

$$\Theta(w^{(A)}, w^{(B)}) = ||w^{(A)} - w(B)||_2^2$$

Basically, it is a regularization term penalizing distant parameters. While a parameter norm penalty is one way to regularize parameters to be close to one another, the more popular way is to use constraints: to force sets of parameters to be equal. This method of regularization is often referred to as **parameter sharing**, because we interpret the various models or model components as sharing a unique set of parameters.

By far the most popular and extensive use of parameter sharing occurs in convolutional neural networks (CNNs) applied to computer vision. The same feature is computed over different locations in the input.

This means that we can find a cat with the same cat detector whether the cat appears at column i or column $i+1$ in the image.

6.11 Sparse representation

We have seen L^1 regularization to induce weight sparsity. We can similarly induce **representation sparsity** by placing a penalty on the activations of the units in a neural network, encouraging their activations to be sparse.

$$\Omega(\mathbf{h}) = ||\mathbf{h}||_1 = \sum_i |h_i|$$

Representational sparsity, on the other hand, describes a representation where many of the elements of the representation are zero (or close to zero).

6.12 Ensemble methods

The idea of ensemble learning is to get predictions from multiple models and aggregate them. In the classification case, an **ensemble** of classifiers (base/weak learners) is a set of classifiers whose individual decisions are combined in some way (e.g. majority voting) to classify new examples. Without sufficient data, many hypotheses can have the same level of accuracy on the training data. By "*averaging*" the classifications of several good classifiers the risk of choosing the wrong classifier is reduced.

Bagging:

1. Create k bootstrap samples:
2. Train a distinct classifier on each sample;
3. Classify new instances by majority voting / average.

For example, considering a linearly separable training set, we can train a perceptron on each bootstrap sample. At the end, we'll obtain a number of perceptrons equals to the number of samples and we can take the majority voting of their predictions.

Ideally, bagging eliminates variance while keeping the bias almost unchanged. Bagging fails when models are very similar (not independent enough). This happens if the learning algorithm is stable, that is, it doesn't change much after changing a few instances. If we have high-bias models and we want to combine them, the result tends to be bad. These methods are more suitable when we want to minimize the variance error.

Boosting:

- Use the training set to train a simple predictor;
- Re-weight the training examples giving more weight to examples that were wrongly classified;
- Repeat n times;
- Combine the simple hypotheses into a single accurate predictor.

Boosting reduces **bias** (zero error on training set) by making each classifier focus on previous mistakes.

6.12.1 Dropout

Dropout can be thought of as an efficient method of performing bagging on neural network. Specifically, dropout trains the ensemble consisting of all sub-networks that can be formed by removing non-output units from an underlying base network. We can effectively remove a unit from a network by multiplying its output value by zero.

To train with dropout, we use a minibatch-based learning algorithm that makes small steps, such as stochastic gradient descent. Each time we load an example into a minibatch, we randomly sample a different binary mask to apply to all of the input and hidden units in the network. The probability of sampling a mask value of one (causing a unit to be included) is a hyperparameter fixed before training begins. Typically, an input unit is included with probability 0.8 and a hidden unit is included with probability 0.5. We then run forward propagation, back-propagation, and the learning update as usual.

More formally, suppose that a mask vector μ specifies which units to include, and $J(\theta, \mu)$ defines the cost of the model defined by parameters θ and mask μ. Then dropout training consists in minimizing $E_\mu J(\theta, \mu)$. We can obtain an unbaiased estimate by sampling μ for each training step.

Dropout training is not quite the same as bagging training. In the case of bagging, the models are all independent. In the case of dropout, the models share parameters, with each model inheriting a different subset of parameters from the parent neural network.

To make a prediction, a bagged ensemble must accumulate votes from all of its members. We refer to this process as **inference** in this context. In the case of bagging, each model i produces a probability distribution $p^{(i)}(y|\mathbf{x})$. The prediction of the ensemble is given by the arithmetic mean of all of these distributions:

$$\frac{1}{k}\sum_{i=1}^{k} p^{(i)}(y|\mathbf{x})$$

In the case of dropout, each sub-model defined by mask vector μ defines a probability distribution $p(y|\mathbf{x}, \mu)$. The arithmetic mean over all masks is given by:

$$\sum_{\mu} p(\mu)p(y|\mathbf{x}, \mu)$$

where $p(\mu)$ is the probability distribution that was used to sample μ at training time.

Because this sum includes an exponential number of terms, it is intractable to evaluate. Instead, we can approximate the inference with sampling, by averaging together the output from many masks. Even 10-20 masks are often sufficient to obtain good performance.

However, there is an even better approach, that allows us to obtain a good approximation to the predictions of the entire ensemble, at the cost of only one forward propagation. To do so, we change to using the geometric mean rather than the arithmetic mean of the ensemble members' predicted distributions. The unnormalized probability distribution defined directly by the geometric mean is given by:

$$\tilde{p}_{ensemble}(y|\mathbf{x}) = \sqrt[2^d]{\prod_{\mu} p(y|\mathbf{x}, \mu)}$$

where d is the number of units that may be dropped. To make predictions we must re-normalize the ensemble:

$$p_{ensemble}(y|\mathbf{x}) = \frac{\tilde{p}_{ensemble}(y|\mathbf{x})}{\sum_{y'} p_{ensemble}(y'|\mathbf{x})}$$

A key insight involved in dropout is that we can approximate $p_{ensemble}$ by evaluating $p(y|\mathbf{x})$ in one model: the model with all units, but with the weights going out of unit i multiplied by the probability of including unit i. The motivation for this modification is to capture the right expected value of the output from that unit. We call this approach the **weight scaling inference rule**. There is not yet any theoretical argument for the accuracy of this approximate inference rule in deep nonlinear networks, but empirically it performs very well.

Because we usually use an inclusion probability of $\frac{1}{2}$, the weight scaling rule usually amounts to dividing the weights by 2 at the end of training, and then using the model as usual.

For many classes of models that do not have nonlinear hidden units, the weight scaling inference rule is exact. For example, consider a softmax regression classifier with n input variables represented by the vector $\mathbf{x}$:

$$P(\mathrm{y} = y|\mathbf{x}) = softmax(\mathbf{W}^T\mathbf{x} + \mathbf{b})_y$$

We can index into the family of sub-models by element-wise multiplication of the input with a binary vector $\mathbf{d}$:

$$P(\mathrm{y} = y|\mathbf{x}) = softmax(\mathbf{W}^T(\mathbf{x} \odot \mathbf{d}) + \mathbf{b})_y$$

It can be proved that the following holds:

$$\tilde{P}_{ensemble}(\mathrm{y} = y|\mathbf{x}) \propto exp(\frac{1}{2}\mathbf{W}^T_{y,:}\mathbf{x} + b_y)$$

6.13 Adversarial training

In order to probe the level of understanding that a network has of the underlying task, we can search for examples that the model misclassifies. Even neural networks that perform at human level accuracy have a nearly 100% error rate on examples that are intentionally constructed to mislead the model. These examples are created by using an optimization procedure which searches for an input $\mathbf{x}'$ near a data point $\mathbf{x}$ such that the model output is very different at $\mathbf{x}'$. In many cases, $\mathbf{x}'$ can be so similar to $\mathbf{x}$ that a human observer cannot tell the difference between the original example and the **adversarial example**, but the network can make highly different predictions. Adversarial examples have many implications, for example, in computer security. They are also interesting in the context of regularization because one can reduce the error rate on the original test set via adversarial training, that is, training on adversarially perturbed example from the training set.

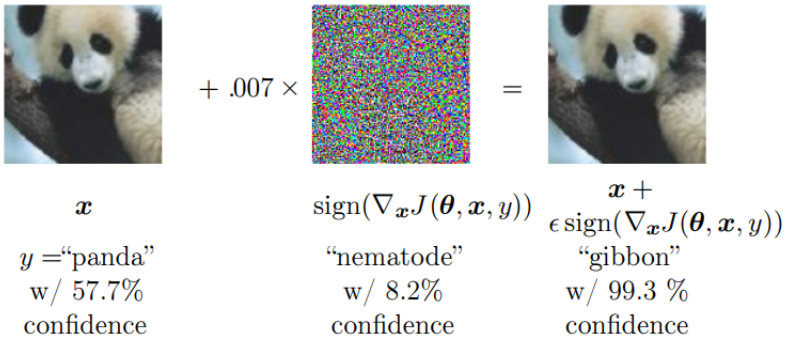


Figure 7.8: A demonstration of adversarial example generation applied to GoogLeNet (Szegedy *et al.*, 2014a) on ImageNet. By adding an imperceptibly small vector whose elements are equal to the sign of the elements of the gradient of the cost function with respect to the input, we can change GoogLeNet's classification of the image. Reproduced with permission from Goodfellow *et al.* (2014b).

Chapter 7

Lec 11 - 12 - Optimization

7.1 Optimization for Neural Networks

Optimization algorithms used for training of deep models differ from traditional optimization algorithms in several ways. In most machine learning scenarios, we care about some performance measure P, that is defined with respect to the test set and may also be intractable. We therefore optimize P only **indirectly**. We reduce a different cost function $J(\theta)$ in the hope that doing so will improve P.

The goal of a machine learning algorithm is to reduce the expected generalization error:

$$J^*(\theta) = E_{(\mathbf{x},y)\sim p_{data}} L(f(\mathbf{x};\theta), y)$$

This quantity is known as the **risk**. If we knew the true distribution $p_{data}(\mathbf{x}, y)$, risk minimization would be an optimization task solvable by an optimization algorithm. However, when we do not know $p_{data}(\mathbf{x}, y)$ but only have a training set of samples, we have a machine learning problem. The simplest way to convert a machine learning problem back into an optimization problem is to minimize the expected loss on the training set. This means replacing the true distribution $p(\mathbf{x}, y)$ with the empirical distribution $\hat{p}(\mathbf{x}, y)$ defined by the training set. We now minimize the **empirical risk**:

$$E_{(\mathbf{x},y)\sim \hat{p}_{data}} L(f(\mathbf{x};\theta), y) = \frac{1}{m}\sum_{i=1}^{m} L(f(\mathbf{x}^{(i)};\theta), y^{(i)})$$

where m is the number of training examples. We optimize the empirical risk, and hope that the risk decreases significantly as well [1]. However, empirical risk minimization is prone to overfitting. Models with high capacity can simply memorize the training set. Furthermore, the most effective modern optimization algorithms are based on gradient descent, but many useful loss functions, such as 0-1 loss, have no useful derivatives.

7.1.1 Surrogate Loss

Sometimes, the loss function we actually care about (e.g. classification error) is not one that can be optimized efficiently. For example, exactly minimizing expected 0-1 loss is typically intractable (exponential in the input dimension), even for a linear classifier. In such situations, one typically optimizes a surrogate loss function instead, which acts as a proxy, but has advantages. For example, the negative log-likelihood of the correct class is typically used as a surrogate for the 0-1 loss.

[1] A variety of theoretical results establish conditions under which the true risk can be expected to decrease by various amounts

A very important difference between optimization in general and optimization as we use it for training algorithms is that training algorithms do not usually halt at a local minimum. A machine learning algorithm usually minimizes a surrogate loss function but halts when a convergence criterion based on early stopping is satisfied (whenever overfitting begins to occur).

7.1.2 Batch and Minibatch algorithms

One aspect of machine learning algorithms that separates them from general optimization algorithms is that the objective function usually decomposes as a sum over the training examples.

Most of the properties of the objective function J used by most of the optimization algorithms are expectations over the training set. For example, the most commonly used property is the gradient:

$$\nabla_\theta J(\theta) = E_{(\mathbf{x},y)\sim\hat{p}_{data}} \nabla_\theta log\ p_{model}(\mathbf{x}, y; \theta)$$

Computing this expectation exactly is very expensive because it requires evaluating the model on every example in the entire dataset. In practice, we can compute these expectations by randomly sampling a small number of examples from the dataset, then taking the average over only those examples.

This is motivated by the fact that the standard error of the mean estimated from n samples is given by $\sigma/\sqrt{n}$, where σ is the true standard deviation of the value of the samples. Note that the denominator of $\sqrt{n}$ shows that there are less than linear returns to using more examples to estimate the gradient. Compare two hypothetical estimates of the gradient, one based on 100 examples and another based on 10,000 examples. The latter requires 100 times more computation than the former, but reduces the standard error of the mean only by a factor of 10.

Most optimization algorithms converge much faster (in terms of total computation, not in terms of number of updates) if they are allowed to rapidly compute approximate estimates of the gradient rather than slowly computing the exact gradient.

Another consideration motivating statistical estimation of the gradient from a small number of samples is redundancy in the training set. In the worst case, all m samples in the training set could be identical copies of each other. A sampling based estimate of the gradient could compute the correct gradient with a single sample, using m times less computation than the naive approach. Furtherore, small batch size have a regularization effect (introduces noise in the gradient). Gradient-based optimization algorithms can be implemented in three ways:

- **(Full, batch) gradient descent**: computes the gradient of the Loss on the whole training set and then updates the weights.
- **(Online) Stochastic gradient descent** computes the gradient of the Loss on a **single** example of the training set and then updates the weights.
- **Mini-batch Stochastic gradient descent** computes the gradient of the Loss on a subset of examples (mini-batch) of the training set and then updates the weights

7.2 Challenges in Neural Network Optimization

7.2.1 Ill- conditioning of Hessian

A very general problem in numerical optimization is ill-conditioning of the Hessian matrix **H**. Ill-conditioning can manifest by causing SGD to get "stuck" because gradient does not carry enough information about the

curvature of the loss function. Even small steps in the gradient direction may result in an increase of the cost function.

The condition number is the ratio between the highest eigenvalues and the lowest eigenvalues. The eigenvalues of the Hessian provides information about the curvature of the loss function in different directions around a point. In particular, the maximum eigenvalue determines the maximum second derivative and the minimum eigenvalue determines the minimum second derivative.

When the Hessian has a poor (large) condition number, it means the curvature of the loss function along some direction is much higher than the curvature along some other direction (in the case of a quadratic cost function, this means having a shape like a thin valley).

Gradient descent is unaware of this change in curvature, so it does not know that it needs to explore preferentially in the direction where the derivative remains negative for longer.

In the case of the quadratic cost function mentioned above, if the step size is not small enough, gradient descent may waste time by repeatedly descending these "canyon walls" since it overshoots the minimum.

Second order methods may solve these problems, but they're not widespread in neural networks training.

7.2.2 Local minima

One of the most prominent features of a convex optimization problem is that it can be reduced to the problem of finding a local minimum. Any local minimum is guaranteed to be a global minimum. When optimizing a convex function, we know that we have reached a good solution if we find a critical point of any kind. With non-convex functions, such as neural nets, it is possible to have many local minima. This could be a problem because they can be associated with a high value loss, being far away from the global minimum. Since the derivative in these points is 0, gradient-based optimization algorithms may be stuck in a sub-optimal solution. This is valid also for other kind of critical points such as saddle points, local maxima and flat regions. However, as we will see, local minima is not necessarily a major problem, since new research suggest that, for large enough nn, local minima have generally low cost.

To check if we reached a critical point, we can plot the norm of the gradient:

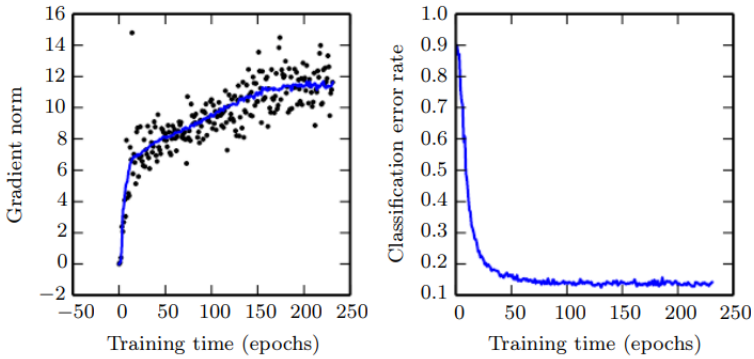


Figure 8.1: Gradient descent often does not arrive at a critical point of any kind. In this example, the gradient norm increases throughout training of a convolutional network used for object detection. *(Left)*A scatterplot showing how the norms of individual gradient evaluations are distributed over time. To improve legibility, only one gradient norm is plotted per epoch. The running average of all gradient norms is plotted as a solid curve. The gradient norm clearly increases over time, rather than decreasing as we would expect if the training process converged to a critical point. *(Right)*Despite the increasing gradient, the training process is reasonably successful. The validation set classification error decreases to a low level.

7.2.3 Saddle points

Many classes of random functions exhibit the following behavior: in low dimensional spaces, local minima are common. In higher dimensional spaces, local minima are rare and saddle points are more common. More formally, For a function $f : \mathbb{R}^n \to \mathbb{R}$ of this type, the expected ratio of the number of saddle points to local minima grows exponentially with n. To understand the intuition behind this behavior, observe that the Hessian matrix at a local minimum has only positive eigenvalues. The Hessian matrix at a saddle point has a mixture of positive and negative eigenvalues. In n-dimensional space, it is exponentially unlikely that all eigenvalues are positive.

Fortunately, the eigenvalues of the Hessian become more likely to be positive as we reach regions of lower cost. This means that local minima are much more likely to have low cost than high cost. This happens also for neural networks.

For first-order optimization algorithms that use only gradient information, the gradient can often become very small near a saddle point. On the other hand, gradient descent empirically seems to be able to escape saddle points in many cases.

For Newton's method, it is clear that saddle points constitute a problem. Gradient descent is designed to move *downhill* and is not explicitly designed to seek a critical point. Newton's method, however, is designed to solve for a point where the gradient is zero. Without appropriate modification, it can jump to a saddle point.

There are other kinds of points with zero gradient besides minima and saddle points. There are also maxima, which are much like saddle points from the perspective of optimization—many algorithms are not attracted to them, but unmodified Newton's method is. Maxima of many classes of random functions become exponentially rare in high dimensional space, just like minima do.

There may also be wide, flat regions of constant value. In these locations, the gradient and also the Hessian are all zero.

7.2.4 Cliffs and exploding gradients

Neural networks with many layers often have extremely steep regions resembling cliffs. On the face of an extremely steep cliff structure, the gradient update step can move the parameters extremely far, usually jumping off of the cliff structure altogether. Fortunately, its most serious consequences can be avoided using the **gradient clipping** heuristic. The basic idea is to reduce the step size to be small enough that it is less likely to go outside the region where the gradient indicates the direction of approximately steepest descent.

7.2.5 Exploding/Vanishing gradient

Another difficulty that neural network optimization algorithms must overcome arises when the computational graph becomes extremely deep. Feedforward networks with many layers have such deep computational graphs. So do **recurrent networks**, which construct very deep computational graphs by repeatedly applying the same operation at each time step of a long temporal sequence. Repeated application of the same parameters gives rise to the **vanishing and exploding gradient problem**. Vanishing gradients make it difficult to know which direction the parameters should move to improve the cost function, while exploding gradients can make learning unstable. The cliff structures described earlier that motivate gradient clipping are an example of the exploding gradient phenomenon.

The vanishing gradient problem can occur also with activation functions that can saturate (e.g. sigmoid) providing gradient close to zero [2].

Recurrent networks use the same matrix $\mathbf{W}$ at each time step, but feedforward networks do not, so even very deep feedforward networks with non-saturating activation functions (e.g. ReLU) can largely avoid the vanishing and exploding gradient problem.

7.2.6 Inexact gradients

Most optimization algorithms are designed with the assumption that we have access to the exact gradient or Hessian matrix. In practice, we usually only have a noisy or even biased estimate of these quantities. Nearly every deep learning algorithm relies on sampling-based estimates using a minibatch of training examples to compute the gradient.

In other cases, the objective function we want to minimize is actually intractable. When the objective function is intractable, typically its gradient is intractable as well. In such cases we can only approximate the gradient.

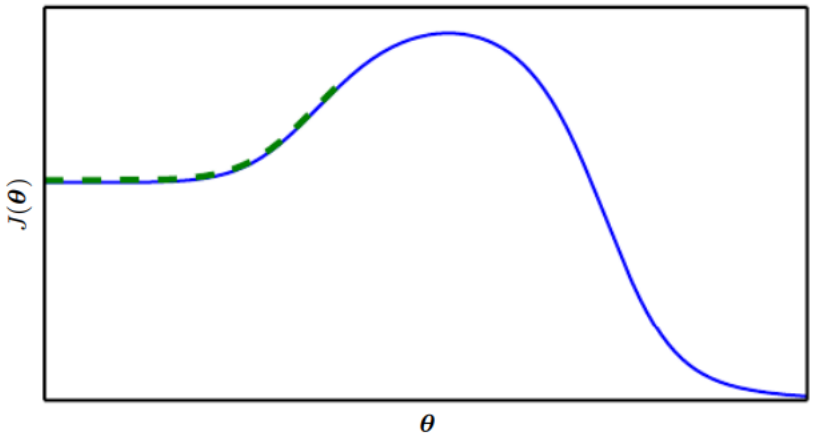
Various neural network optimization algorithms are designed to account for imperfections in the gradient estimate. One can also avoid the problem by choosing a surrogate loss function that is easier to approximate than the true loss.

7.2.7 Local structure not representative of global structure

Many of the problems we have discussed so far correspond to properties of the loss function at a single point—it can be difficult to make a single step if $J(\theta)$ is poorly conditioned at the current point θ, or if θ lies on a cliff, or if θ is a saddle point hiding the opportunity to make progress downhill from the gradient.

[2] basically, weights are no longer updated since the gradient is too small

Optimization based on local downhill moves can fail if the local surface does not point toward the global solution.



Much of research into the difficulties of optimization has focused on whether training arrives at a global minimum, a local minimum, or a saddle point, but in practice neural networks do not arrive at a critical point of any kind. The problem in this case is bad initialization.

Many existing research directions are aimed at finding **good initial points** for problems that have difficult global structure, rather than developing algorithms that use non-local moves. Gradient descent and essentially all learning algorithms that are effective for training neural networks are based on making small, local moves.

7.3 Basic Optimization algorithm

7.3.1 Stochastic Gradient Descent

Stochastic gradient descent (SGD) and its variants are probably the most used optimization algorithms for machine learning in general and for deep learning in particular.

Algorithm 8.1 Stochastic gradient descent (SGD) update at training iteration k

Require: Learning rate ϵ_k.
Require: Initial parameter $\boldsymbol{\theta}$
  while stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient estimate: $\hat{\boldsymbol{g}} \leftarrow +\frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon \hat{\boldsymbol{g}}$
  end while

A crucial parameter for the SGD algorithm is the learning rate. In practice, it is necessary to gradually decrease the learning rate over time, so we now denote the learning rate at iteration k as ϵ_k. This is because the SGD gradient estimator introduces a source of noise (the random sampling of m training examples) that

does not vanish even when we arrive at a minimum. The true gradient approaches 0 in a minimum, the stochastic estimate doesn't, so batch gradient descent can use a fixed learning rate. In practice, it is common to decay the learning rate linearly until iteration τ:

$$\epsilon_k = (1-\alpha)\epsilon_0 + \alpha\epsilon_\tau$$

with $\alpha = \frac{k}{\tau}$. After iteration τ, it is common to leave ϵ_τ constant.

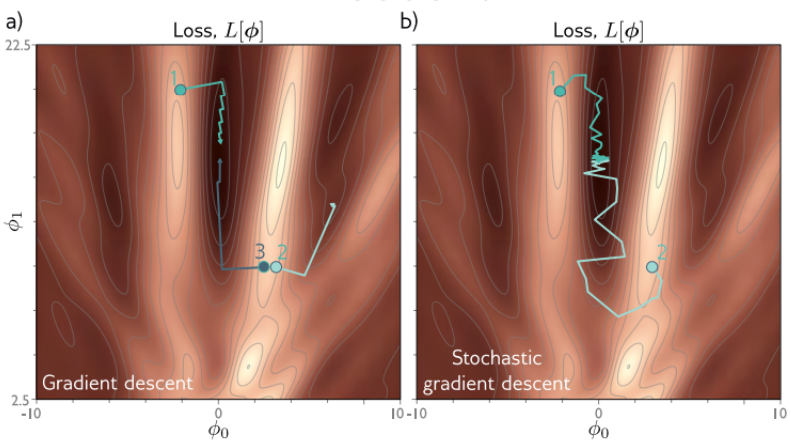
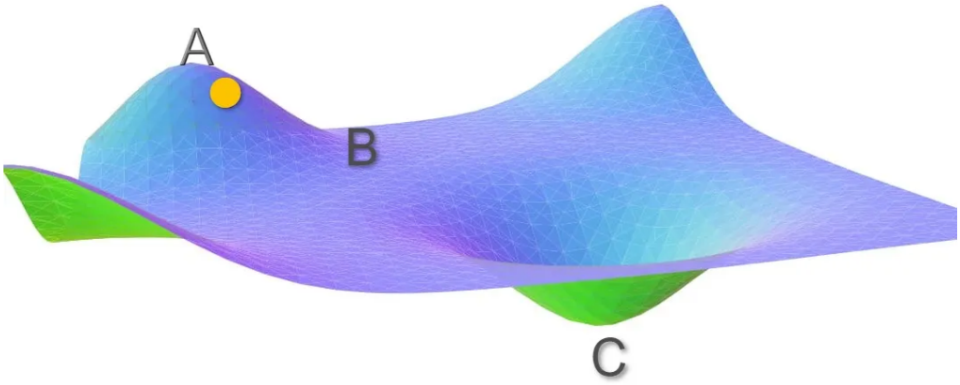


Figure 6.5 Gradient descent vs. stochastic gradient descent. a) Gradient descent. As long as the gradient descent algorithm is initialized in the right "valley" of the loss function (e.g., points 1 and 3), the parameter estimate will move steadily toward the global minimum. However, if it is initialized outside this valley (e.g., point 2) then it will descend toward one of the local minima. b) Stochastic gradient descent adds noise to the optimization process, and so it is possible to escape from the wrong right valley (e.g., point 2) and still reach the global minimum.

7.3.2 Momentum

While stochastic gradient descent remains a very popular optimization strategy, learning with it can sometimes be slow. The method of **momentum** is designed to accelerate learning, especially in the face of high curvature, small but consistent gradients, or noisy gradients. Furthermore, the problem with gradient descent is that the weight update at a moment (t) is governed by the learning rate and gradient at that moment only. It does not take into account the past steps taken while traversing the cost space.



Let's assume the initial weights of the network under consideration correspond to point A. With gradient descent, the Loss function decreases rapidly along the slope AB as the gradient along this slope is high. But as soon as it reaches point B the gradient becomes very low. The weight updates around B is very small. Even after many iterations, the cost moves very slowly before getting stuck at a point where the gradient eventually becomes zero.

Now, Imagine you have a ball rolling from point A. The ball starts rolling down slowly and gathers some momentum across the slope AB. When the ball reaches point B, it has accumulated enough momentum to push itself across the plateau region B and finally following slope BC to land at the global minima C.

The momentum algorithm accumulates an exponentially decaying moving average of past gradients and continues to move in their direction:

$$\mathbf{v}(t) = \alpha \mathbf{v}(t-1) - \epsilon \sigma(t)$$

$$\theta = \theta + \mathbf{v(t)}$$

where:

- $\mathbf{v}(t)$ is called **velocity** vector and determines the new weight update done at iteration t.
- α is a hyperparameter between 0 and 1 which determines how quickly the contributions of previous gradients exponentially decay.
- $\sigma(t) = \nabla_\theta \left(\frac{1}{m} \sum_{i=1}^{m} L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)}) \right)$ is the gradient at iteration t.

Assume that $\mathbf{v}(0) = 0$:

$$\begin{aligned}
&\mathbf{v}(0) = 0 \\
&\mathbf{v}(1) = \alpha\mathbf{v}(0) - \epsilon\sigma(1) \\
&\mathbf{v}(1) = -\epsilon\sigma(1) \\
\\
&\mathbf{v}(2) = \alpha\mathbf{v}(1) - \epsilon\sigma(2) \\
&\mathbf{v}(2) = \alpha(-\epsilon\sigma(1)) - \epsilon\sigma(2) \\
&\mathbf{v}(2) = -\epsilon(\alpha\sigma(1) + \sigma(2)) \\
\\
&\mathbf{v}(3) = \alpha\mathbf{v}(2) - \epsilon\sigma(3) \\
&\mathbf{v}(3) = -\epsilon(\alpha^2\sigma(1) + \alpha\sigma(2) + \sigma(3))
\end{aligned} \tag{7.1}$$

Let's *ignore* the learning rate ϵ and focus on α:

- with $\alpha = 0.1$, at the third iteration the gradient at $t = 3$ will contribute 100% of its value, the gradient at $t = 2$ will contribute 10% of its value, and gradient at t=1 will only contribute 1% of its value (the contribution from earlier gradients decreases rapidly).

- with $\alpha = 0.9$, at the third iteration the gradient at $t = 3$ will contribute 100% of its value, $t = 2$ will contribute 90% of its value, and gradient at $t = 1$ will contribute 81% of its value.

From above, we can deduce that the larger α, the more previous gradients affect the current direction. Note that the actual contribution of each gradient in the weight update will be further subjected to the learning rate.

Common values of α used in practice include .5, .9, and .99. Like the learning rate, α may also be adapted over time.

Algorithm 8.2 Stochastic gradient descent (SGD) with momentum

Require: Learning rate ϵ, momentum parameter α.
Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$.
  while stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient estimate: $\boldsymbol{g} \leftarrow \frac{1}{m}\nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    Compute velocity update: $\boldsymbol{v} \leftarrow \alpha\boldsymbol{v} - \epsilon\boldsymbol{g}$
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$
  end while

When all the past gradients have the same sign we will take large steps while updating the weights. Even if the learning rate is low, all the gradients along the curve will have the same direction, thus increasing the momentum and accelerating the descent.

Another problem that momentum solves is poor conditioning of the Hessian matrix.

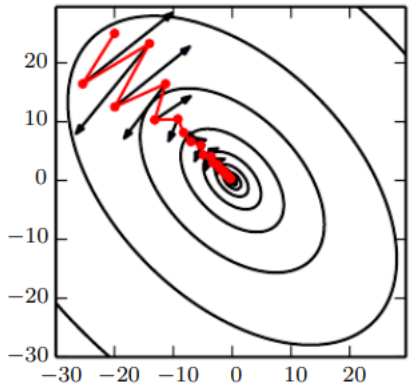


Figure 8.5: Momentum aims primarily to solve two problems: poor conditioning of the Hessian matrix and variance in the stochastic gradient. Here, we illustrate how momentum overcomes the first of these two problems. The contour lines depict a quadratic loss function with a poorly conditioned Hessian matrix. The red path cutting across the contours indicates the path followed by the momentum learning rule as it minimizes this function. At each step along the way, we draw an arrow indicating the step that gradient descent would take at that point. We can see that a poorly conditioned quadratic objective looks like a long, narrow valley or canyon with steep sides. Momentum correctly traverses the canyon lengthwise, while gradient steps waste time moving back and forth across the narrow axis of the canyon. Compare also figure 4.6, which shows the behavior of gradient descent without momentum.

In fact, the gradient in one direction is decreased if the gradient direction repeatedly changes as the terms in the sum cancel out.

The overall effect is a smoother trajectory and reduced oscillatory behavior in valleys

7.3.3 Nesterov momentum

Nesterov momentum introduces a variant of the momentum algorithm. The difference between Nesterov momentum and standard momentum is where the gradient is evaluated. With Nesterov momentum the gradient is evaluated after the current velocity is applied.

Algorithm 8.3 Stochastic gradient descent (SGD) with Nesterov momentum

Require: Learning rate ϵ, momentum parameter α.
Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$.
  while stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding labels $\boldsymbol{y}^{(i)}$.
    Apply interim update: $\tilde{\boldsymbol{\theta}} \leftarrow \boldsymbol{\theta} + \alpha \boldsymbol{v}$
    Compute gradient (at interim point): $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\tilde{\boldsymbol{\theta}}} \sum_i L(f(\boldsymbol{x}^{(i)}; \tilde{\boldsymbol{\theta}}), \boldsymbol{y}^{(i)})$
    Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \boldsymbol{g}$
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$
  end while

7.4 Parameter initialization

Deep Learning training algorithms are iterative and depends on initialization. The initial point can determine whether the algorithm converges at all, with some initial points being so unstable that the algorithm encounters numerical difficulties and fails altogether. The use of an optimization algorithm such as stochastic gradient descent that makes small incremental changes to the weights and tends to halt in areas that are nearer to the initial parameters, expresses a prior that the final parameters should be close to the initial parameters.

Modern initialization strategies are simple and heuristic. Designing improved initialization strategies is a difficult task because neural network optimization is not yet well understood. Perhaps the only property known with complete certainty is that the initial parameters need to "break symmetry" between different units.

If all the weights in a layer are initialized to the same value, each neuron will compute exactly the same function. The gradient will also be the same, so the weights associated to each neuron will evolve having exactly the same values. In other words, the layer will be equivalent to a layer with only one neuron.

We almost always initialize all the weights in the model to values drawn randomly from a Gaussian or uniform distribution. The choice of Gaussian or uniform distribution does not seem to matter very much, but has not been exhaustively studied. The scale of the initial distribution, however, does have a large effect on both the outcome of the optimization procedure and on the ability of the network to generalize.

Larger initial weights will yield a stronger symmetry breaking effect, helping to avoid redundant units. Initial weights that are too large may, however, result in exploding values during forward propagation or back-propagation. Large weights may also result in extreme values that cause the activation function to saturate, causing complete loss of gradient through saturated units.

The perspectives of regularization and optimization can give very different insights into how we should initialize a network. The optimization perspective suggests that the weights should be large enough to propagate information successfully, but some regularization concerns encourage making them smaller.

Some heuristics are available for choosing the initial scale of the weights. One heuristic is to initialize

the weights of a fully connected layer with m inputs and n outputs by sampling each weight from:

$$U(-\frac{1}{\sqrt{m}}, \frac{1}{\sqrt{m}})$$

while others suggest using the normalized initialization:

$$W_{i,j} \sim U\left(-\sqrt{\frac{6}{m+n}}, \sqrt{\frac{6}{m+n}}\right)$$

Typically, we set the biases for each unit to heuristically chosen constants, and initialize only the weights randomly.

7.4.1 Pre-training

We can use pre-trained models as starting points for our models. For example:

- Unsupervised pre-training: Train an unsupervised model on the training data, use the resulting model as initialization.
- Supervised pre-training (transfer learning): Initialize the weights with a model trained on a related task (Especially effective with CNNs).

7.5 Adaptive learning rates

Gradient descent with a fixed step size has an undesirable property: it makes:

- large adjustments to parameters associated with large gradients (where perhaps we should be more cautious).
- 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 makes good progress in both directions and is stable. It can make sense to use a separate learning rate for each parameter, and automatically adapt these learning rates throughout the course of learning.

The delta-bar-delta algorithm is an early heuristic approach to adapting individual learning rates for model parameters during training. The approach is based on a simple idea: if the partial derivative of the loss, with respect to a given model parameter, remains the same sign, then the learning rate should increase. If the partial derivative with respect to that parameter changes sign, then the learning rate should decrease. Of course, this kind of rule can only be applied to full batch optimization.

More recently, a number of incremental (or mini-batch-based) methods have been introduced that adapt the learning rates of model parameters.

7.5.1 AdaGrad

The AdaGrad algorithm individually adapts the learning rates of all model parameters by scaling them inversely proportional to the square root of the sum of all of the historical squared values of the gradients. Parameters with large partial derivatives will se a rapid decrease in their learning rate.

In the context of convex optimization, the AdaGrad algorithm enjoys some desirable theoretical properties. However, empirically it has been found that—for training deep neural network models—the accumulation of squared gradients from the beginning of training can result in a premature and excessive decrease in the effective learning rate.

Algorithm 8.4 The AdaGrad algorithm

Require: Global learning rate ϵ
Require: Initial parameter $\boldsymbol{\theta}$
Require: Small constant δ, perhaps 10^{-7}, for numerical stability
  Initialize gradient accumulation variable $\boldsymbol{r} = \boldsymbol{0}$
  while stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    Accumulate squared gradient: $\boldsymbol{r} \leftarrow \boldsymbol{r} + \boldsymbol{g} \odot \boldsymbol{g}$
    Compute update: $\Delta\boldsymbol{\theta} \leftarrow -\frac{\epsilon}{\delta + \sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$. (Division and square root applied element-wise)
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
  end while

7.5.2 RMSprop

The RMSProp algorithm modifies AdaGrad to perform better in the non-convex setting by changing the gradient accumulation into an exponentially weighted moving average.

Compared to AdaGrad, the use of the moving average introduces a new hyperparameter, ρ, that controls the length scale of the moving average. Empirically, RMSProp has been shown to be an effective and practical optimization algorithm for deep neural networks.

Algorithm 8.5 The RMSProp algorithm

Require: Global learning rate ϵ, decay rate ρ.
Require: Initial parameter $\boldsymbol{\theta}$
Require: Small constant δ, usually 10^{-6}, used to stabilize division by small numbers.
  Initialize accumulation variables $\boldsymbol{r} = 0$
  while stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    Accumulate squared gradient: $\boldsymbol{r} \leftarrow \rho\boldsymbol{r} + (1 - \rho)\boldsymbol{g} \odot \boldsymbol{g}$
    Compute parameter update: $\Delta\boldsymbol{\theta} = -\frac{\epsilon}{\sqrt{\delta + \boldsymbol{r}}} \odot \boldsymbol{g}$. ($\frac{1}{\sqrt{\delta + \boldsymbol{r}}}$ applied element-wise)
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
  end while

7.5.3 Adam

Adam includes bias corrections to the estimates of both the first-order moments (the momentum term) and the (uncentered) second-order moments to account for their initialization at the origin.

Adam is generally regarded as being fairly robust to the choice of hyperparameters, though the learning rate sometimes needs to be changed from the suggested default.

7.6 Second order methods

In contrast to first-order methods, second-order methods make use of second derivatives to improve optimization. The most widely used second-order method is **Newton's method**. Newton's method is an optimization scheme based on using a second-order Taylor series expansion to approximate $J(\theta)$ near some point θ_0, ignoring derivatives of higher order[3]:

$$J(\theta) \approx J(\theta_0) + (\theta - \theta_0)^T \nabla_\theta J(\theta_0) + \frac{1}{2}(\theta - \theta_0)^T \mathbf{H}(\theta - \theta_0)$$

where $\mathbf{H}$ is the Hessian of J with respect to θ evaluated at θ_0. If we then solve for the critical point of this function, we obtain the Newton parameter update rule:

$$\theta^* = \theta_0 - \mathbf{H}^{-1} \nabla_\theta J(\theta_0)$$

If the function is quadratic, Newton's method finds directly the minimum, if not, it can be applied iteratively. However, Computing $\mathbf{H}$ and $\mathbf{H}^{-1}$, is unfeasible for medium-sized networks.

Conjugate gradients is a method to efficiently avoid the calculation of the inverse Hessian by iteratively descending **conjugate directions**. . The inspiration for this approach follows from a careful study of the weakness of the method of steepest descent, where line searches are applied iteratively in the direction associated with the gradient.

In the method of conjugate gradients, we seek to find a search direction that is conjugate to the previous line search direction, i.e. it will not undo progress made in that direction. At training iteration t, the next search direction $\mathbf{d}_t$ takes the form:

$$\mathbf{d}_t = \nabla_\theta J(\theta) + \beta_t \mathbf{d}_{t-1}$$

where β_t is a coefficient whose magnitude controls how much of the direction, $\mathbf{d}_{t-1}$, we should add back to the current search direction.

Two directions, $\mathbf{d}_t$ and $\mathbf{d}_{t-1}$, are defined as conjugate if $\mathbf{d}_t^T \mathbf{H} \mathbf{d}_{t-1} = 0$, where $\mathbf{H}$ is the Hessian matrix. Fortunately, we can compute the correct β_t without computing $\mathbf{H}$.

For quadratic surfaces in k dimensions, conjugate gradients method requires at most k steps to achieve the minimum (it can be adapted for non-quadratic surfaces).

7.7 Batch normalization

LeCun et al. (1998) showed that normalizing the inputs speeds up training. Loffe and Szegedy (2015) proposed Batch Normalization to normalize hidden (pre-)activations:

- each unit's pre-activation is normalized (mean subtraction, standard deviation division).
- during training, mean and standard deviation is computed for each mini-batch.
- backpropagation takes into account the normalization.

[3] the objective function $J(\theta)$ is the empirical risk

- at test time, the global mean and global standard deviation is used. It requires a final phase where, from the first to the last hidden layer:
  - propagate all training data to that layer.
  - compute and store the global mean and global standard deviation for each unit.

Normalize the pre-activation can help to keep the pre-activation in a non-saturating regime.

Chapter 8

Lec 13 - CNNs

8.1 Convolutional Networks

Convolutional networks, also known as convolutional neural networks or CNNs, are a specialized kind of neural network for processing data that has a known, grid-like topology. Examples include time-series data, which can be thought of as a 1D grid taking samples at regular time intervals, and image data, which can be thought of as a 2D grid of pixels. Convolutional networks have been tremendously successful in practical applications. The name "convolutional neural network" indicates that the network employs a mathematical operation called convolution. The idea is to substitute matrix multiplication with convolution.

Images have three properties that suggest the need for specialized architecture:

- they are high-dimensional, e.g. 224×224 RGB values (i.e., 150,528 input dimensions). Hidden layers in fully connected networks are generally larger than the input size, and so even for a shallow network, the number of weights would exceed 150,528! or 22 billion. Obvious practical problems in terms of the required training data, memory, and computation.
- Nearby image pixels are statistically related. Fully connected networks have no notion of "nearby" and treats the relationship between every input equally; if the pixels of the training and test images were randomly permuted in the same way, the network could still be trained with no practical difference.
- The interpretation of an image is stable under geometric transformations. An image of a tree is still an image of a tree if we shift it leftwards by a few pixels. However, this shift changes every input to the network, and so the model would have to learn the patterns of pixels that correspond to a tree separately at every position. This is clearly inefficient.

Convolutional networks are perhaps the greatest success story of biologically inspired artificial intelligence. Neurophysiologists David Hubel and Torsten Wiesel collaborated for several years to determine many of the most basic facts about how the mammalian vision system works. They observed how neurons in the cat's brain responded to images projected in precise locations on a screen in front of the cat. Their great discovery was that neurons in the early visual system responded most strongly to very specific patterns of light, such as precisely oriented bars, but responded hardly at all to other patterns.

8.1.1 Convolution operator

In its most general form, **convolution** is an operation on two functions of a real-valued argument. To motivate the definition of convolution, we start with examples of two functions we might use.

Suppose we are tracking the location of a spaceship with a laser sensor. Our laser sensor provides a single

output $x(t)$, the position of the spaceship at time t. Both x and t are real-valued, i.e., we can get a different reading from the laser sensor at any instant in time.

Now suppose that our laser sensor is somewhat noisy. To obtain a less noisy estimate of the spaceship's position, we would like to average together several measurements. Of course, more recent measurements are more relevant, so we will want this to be a weighted average that gives more weight to recent measurements. We can do this with a weighting function $w(a)$, where a is the age of a measurement. If we apply such a weighted average operation at every moment, we obtain a new function s providing a smoothed estimate of the position of the spaceship:

$$s(t) = \int_{-\infty}^{\infty} x(a)w(t-a)da$$

The convolution operation is typically denoted with an asterisk:

$$s(t) = (x * w)(t)$$

In convolutional network terminology, the first argument (in this example, the function x) to the convolution is often referred to as the **input** and the second argument (in this example, the function w) as the **kernel** (needs to be a valid probability density function). The output is sometimes referred to as the **feature map**.

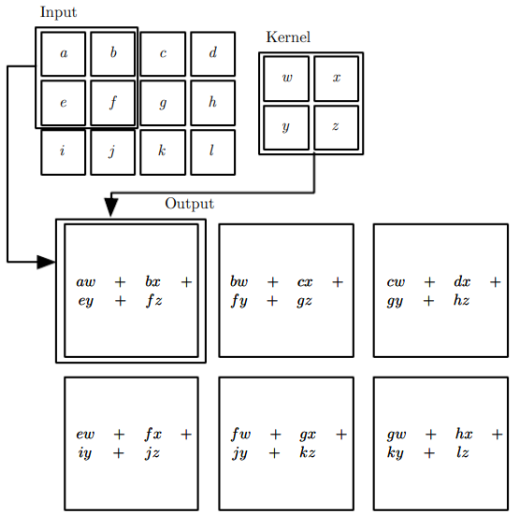
Usually, when we work with data on a computer, time will be discretized:

$$s(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a)$$

In machine learning applications, the input is usually a multidimensional array of data and the kernel is usually a multidimensional array of parameters that are adapted by the learning algorithm.

We often use convolutions over more than one axis at a time. For example, if we use a two-dimensional image I as our input, we probably also want to use a two-dimensional kernel K:

$$S(i,j) = (I * K)(i,j) = \sum_{a=-m}^{m} \sum_{b=-n}^{n} K(a,b)I(i-a, j-b)$$



From the example above we can notice two things:

- The kernel is applied as a sliding window across the image.
- Actually, the operator applied in the example is not convolution, but **cross-correlation:**

$$S(i,j) = \sum_{a=-m}^{m} \sum_{b=-n}^{n} K(a,b)I(i+a, j+b)$$

  In fact, we apply convolution by **flipping** the kernel relative to the input, in the sense that as b increases, the index into the input decreases, but the index into the kernel increases. Many machine learning libraries implement cross-correlation but call it convolution.

The main properties of convolution are:

- Sparse interactions $\rightarrow$ kernel smaller than input, efficiency.
- Parameter sharing
- Equivariant representations
- Works with inputs of variable size

8.1.2 Sparse interactions

Traditional neural network layers use matrix multiplication, having a matrix of parameters with a separate parameter describing the interaction between each input unit and each output unit. Convolutional networks, however, typically have sparse interactions (also referred to as sparse connectivity or sparse weights). This is accomplished by making the kernel smaller than the input. This means that we need to store fewer parameters, which both reduces the memory requirements of the model and improves its statistical efficiency. It also means that computing the output requires fewer operations.

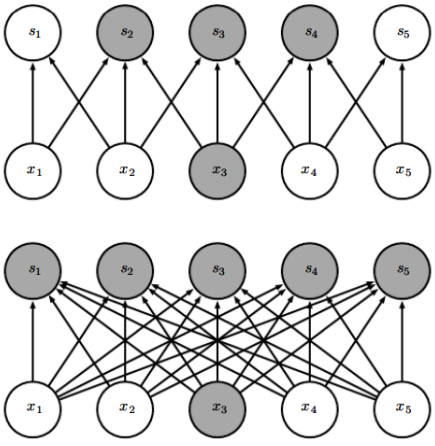
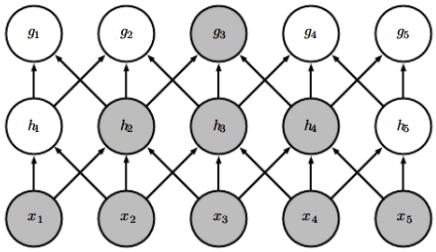


Figure 9.2: *Sparse connectivity, viewed from below:* We highlight one input unit, x_3, and also highlight the output units in $\boldsymbol{s}$ that are affected by this unit. *(Top)*When $\boldsymbol{s}$ is formed by convolution with a kernel of width 3, only three outputs are affected by $\boldsymbol{x}$. *(Bottom)*When $\boldsymbol{s}$ is formed by matrix multiplication, connectivity is no longer sparse, so all of the outputs are affected by x_3.

The receptive field of the units in the deeper layers of a convolutional network is larger than the receptive field of the units in the shallow layers. This means that even though direct connections in a convolutional net are very sparse, units in the deeper layers can be indirectly connected to all or most of the input. This allows the network to efficiently describe complicated interactions between many variables by constructing such interactions from simple building blocks that each describe only sparse interactions.



8.1.3 Parameter sharing

Parameter sharing refers to using the same parameter for more than one function in a model. In a traditional neural net, each element of the weights' matrix is used exactly once when computing the output of a layer. In a convolutional neural net, each member of the kernel is used at every position of the input (except perhaps some of the boundary pixels, depending on the design decisions regarding the boundary). The parameter sharing used by the convolution operation means that rather than learning a separate set of parameters for every location, we learn only one set. It further reduces the storage requirements of the model to k parameters (just the filter), with k several order of magnitude smaller than the input size. Convolution is thus dramatically more efficient than dense matrix multiplication in terms of the memory requirements and statistical efficiency.

Input size: 320 by 280
Kernel size: 2 by 1
Output size: 319 by 280

	Convolution	Dense matrix	Sparse matrix
Stored floats	2	319*280*320*28 0 > 8e9	2*319*280 = 178,640
Float muls or adds	319*280*3 = 267,960	> 16e9	Same as convolution (267,960)

2 multiplications and 1 addition for each position

8.1.4 Equivariance to translation

In the case of convolution, the particular form of parameter sharing causes the layer to have a property called **equivariance** to translation. To say a function is equivariant means that if the input changes, the output changes in the same way. Specifically, a function $f(x)$ is equivariant to a function g if $f(g(x)) = g(f(x))$.

In the case of convolution, if we let g be any function that translates the input, i.e., shifts it, then the convolution function is equivariant to g. For example, with images, convolution creates a 2-D map of where certain features appear in the input. If we move the object in the input, its representation will move the same amount in the output. Basically, due to parameter sharing, the kernel detects a feature regardless of where the feature is in the input image.

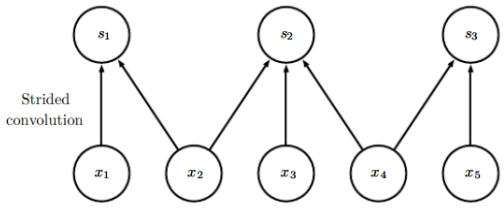
Convolution is not naturally equivariant to some other transformations, such as changes in the scale or rotation of an image. Other mechanisms are necessary for handling these kinds of transformations.

8.1.5 Inputs of variable size

Consider a collection of images, where each image has a different width and height. It is unclear how to model such inputs with a weight matrix of fixed size. Convolution is straightforward to apply; the kernel is simply applied a different number of times depending on the size of the input, and the output of the convolution operation scales accordingly.

8.1.6 Strided Convolution

We may want to skip over some positions of the kernel in order to reduce the computational cost (at the expense of not extracting our features as finely). We can think of this as downsampling the output of the full convolution function. We can do this by sampling only every s pixels in each direction in the output. We refer to s as the **stride** of this downsampled convolution. It is also possible to define a separate stride for each direction of motion.



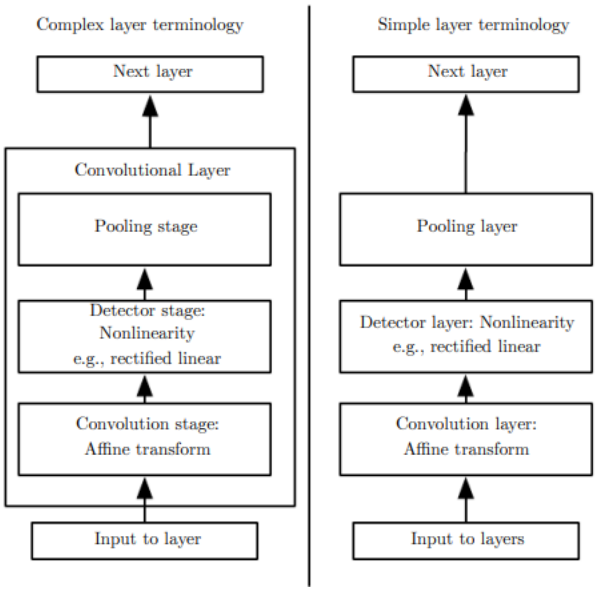
8.1.7 Dilated Convolution

Increasing the kernel size has the disadvantage of requiring more weights. In order to reduce the number of stored weights we can use **dilated convolution**, that is, the kernel values are spaced with zeros. In 1D we can turn a kernel of size five into a dilated kernel of size three by setting the second and fourth elements to zero. We still integrate information from a larger input region but only require three weights to do this. The number of zeros we intersperse between the weights is termed the dilation rate.

8.1.8 Pooling

A typical layer of a convolutional network consists of three stages. In the first stage, the layer performs several convolutions in parallel to produce a set of linear activations. In the second stage, each linear activation is run through a nonlinear activation function, such as the rectified linear activation function. In the third stage, we use a pooling function to modify the output of the layer further.

A pooling function replaces the output of the net at a certain location with a summary statistic of the nearby outputs. For example, the **max pooling** operation reports the maximum output within a rectangular neighborhood. Another popular pooling function is the average of a rectangular neighborhood.



In all cases, pooling helps to make the representation become approximately invariant to small translations of the input. Invariance to translation means that if we translate the input by a small amount, the values of most of the pooled outputs do not change. Furthermore, pooling reduces the size of the input of the subsequent layers, thus it improves efficiency.

For many tasks, pooling is essential for handling inputs of varying size.

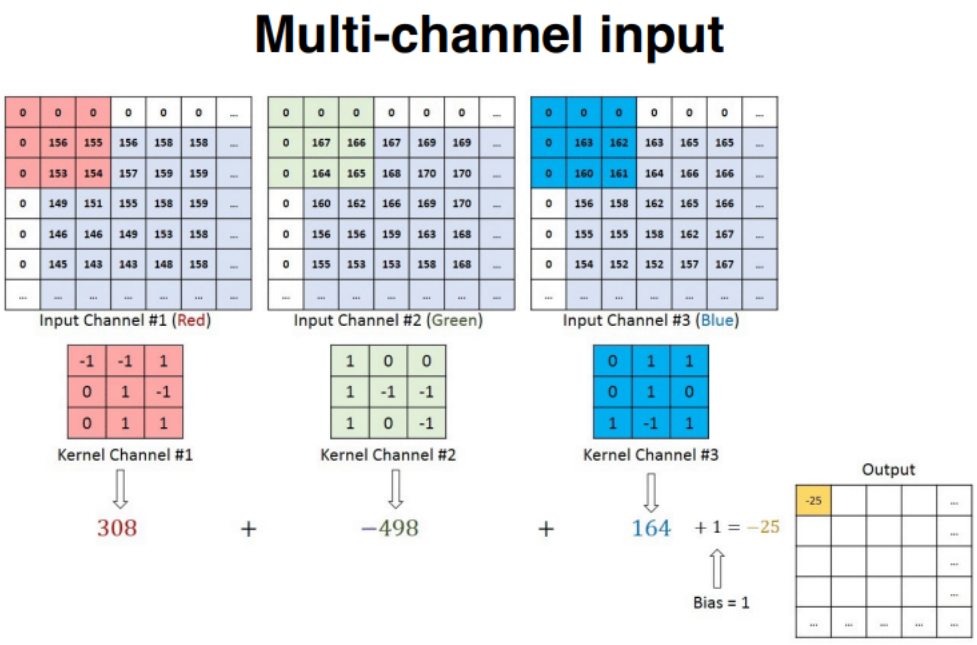
8.1.9 Padding

Given an $n \times n$ kernel, its external row/column coincides with the border of the image when the center of this mask is at distance $\frac{n-1}{2}$ from the edge. If you move further out, part of the window *leaves* the image. This situation can be managed in three different ways:

- limit the movement of the mask, keeping it at a minimum distance of $\frac{n-1}{2}$ from the edges.
- duplicate the external rows/columns of the image
- enlarge the image with rows/columns of zeros

Solution 1 gives reliable results, but produces a different size image from the original. Solutions 2 and 3, on the other hand, give results that are not exactly authentic near the edges, but are often convenient because they allow you to obtain an output image with the same size as the input one.

8.1.10 Multi-channel input



8.1.11 Unshared convolution

In some cases, we do not actually want to use convolution, but rather locally connected layers. This is sometimes also called **unshared convolution**, because it is a similar oper- ation to discrete convolution with a small kernel, but without sharing parameters across locations. Locally connected layers are useful when we know that each feature should be a function of a small part of space, but there is no reason to think that the same feature should occur across all of space.

8.1.12 Structured Outputs

Convolutional networks can be used to output a high-dimensional, structured object, rather than just predicting a class label for a classification task or a real value for a regression task. Typically, this object is a tensor with the same shape of the input, with a class label for each pixel to produce object detection masks.

Chapter 9

Lec 14 - Practical Methodology

9.1 Practical Methodology

During day to day development of machine learning systems, practitioners need to decide whether to gather more data, increase or decrease model capacity, add or remove regularizing features, improve the optimization of a model, improve approximate inference in a model, or debug the software implementation of the model.

We recommend the following practical design process:

- Determine your goals—what error metric to use, and your target value for this error metric. These goals and error metrics should be driven by the problem that the application is intended to solve.
- Establish a working end-to-end pipeline as soon as possible, including the estimation of the appropriate performance metrics.
- Instrument the system well to determine bottlenecks in performance. Diagnose which components are performing worse than expected and whether it is due to overfitting, underfitting, or a defect in the data or software.
- Repeatedly make incremental changes such as gathering new data, adjusting hyperparameters, or changing algorithms, based on specific findings from your instrumentation.

9.1.1 Performance Metrics

Keep in mind that for most applications, it is impossible to achieve absolute zero error. The Bayes error defines the minimum error rate that you can hope to achieve, even if you have infinite training data and can recover the true probability distribution. This is because your input features may not contain complete information about the output variable, or because the system might be intrinsically stochastic. You will also be limited by having a finite amount of training data.

An important aspect is the choice of which metric to use. Several different performance metrics may be used to measure the effectiveness of a complete application that includes machine learning components. It is common to measure the accuracy, or equivalently, the error rate, of a system. However, many applications require more advanced metrics. For example, for an e-mail spam detection system it is much worse to block a legitimate message than to allow a questionable message to pass through. Rather than measuring the error rate of a spam classifier, we may wish to measure some form of total cost, where the cost of blocking legitimate messages is higher than the cost of allowing spam messages.

Sometimes we wish to train a binary classifier that is intended to detect some rare event. Consider a disease that only one in 1M people have. We can easily achieve 99.9999% accuracy on the detection task, by simply hard-coding the classifier to always report that the disease is absent. Clearly, accuracy is a poor way to characterize the performance of such a system. One way to solve this problem is to instead measure **precision** and **recall**. Before introducing these measures, let's define the contingency table:

	Relevant	Not relevant
Retrieved	True Positive (TP)	False Positive (FP)
Not Retrieved	False Negative (FN)	True Negative (TN)

The accuracy α is defined as follows:

$$\alpha = \frac{TP + TN}{TP + TN + FP + FN}$$

If relevance is assumed to be **binary-valued**, effectiveness is typically measured as a combination of:

- **Precision**:
$$\pi = \frac{TP}{TP + FP}$$
is the fraction of detections reported by the model that were correct.

- **Recall:**
$$\rho = \frac{TP}{TP + FN}$$
is the fraction of true events that were detected.

A detector that says no one has the disease would achieve zero precision, and zero recall. A detector that says everyone has the disease would achieve perfect recall, but precision equal to the percentage of people who have the disease (0.0001% in our example of a disease that only one people in a million have).

When using precision and recall, it is common to plot a **PR curve**, with precision on the y-axis and recall on the x-axis. The classifier generates a score that is higher if the event to be detected occurred.

In some cases, it may be necessary to get a trade-off between Precision and Recall. This can be done using the **F-measure**, which is a weighted harmonic mean of the Precision (π) and Recall (ρ):

$$F_\beta = \frac{(1 + \beta^2)\pi\rho}{\beta^2\pi + \rho}$$

If $\beta < 1$ it emphasizes Precision, while $\beta > 1$ emphasizes Recall. When $\beta = 1$, we obtain the so called F1 score:

$$F_1 = 2\frac{\pi\rho}{\pi + \rho}$$

9.1.2 Coverage

In some applications, it is possible for the machine learning system to refuse to make a decision. It is useful when a wrong decision can be harmful and/or if a human operator is able to occasionally take over. A natural performance metric to use in this situation is **coverage**. Coverage is the fraction of examples for which the machine learning system is able to produce a response. It is possible to trade coverage for accuracy. One can always obtain 100% accuracy by refusing to process any example, but this reduces the coverage to 0%.

9.1.3 Baseline models

Baselines are simple models chosen to provide a first estimation of the achievable level of performance. Depending on the complexity of your problem, you may even want to begin without using deep learning. If your problem has a chance of being solved by just choosing a few linear weights correctly, you may want to begin with a simple statistical model like logistic regression.

First, choose the general category of model based on the structure of your data (e.g. if you want to perform supervised learning with fixed-size vectors as input, use a feedforward network with fully connected layers). You should begin by using some kind of piecewise linear unit (ReLUs or their generalizations like Leaky ReLUs, PreLus and maxout). If your input or output is a sequence, use a gated recurrent net (LSTM or GRU).

A reasonable choice of optimization algorithm is SGD with momentum with a decaying learning rate. Another very reasonable alternative is Adam.

While it is reasonable to omit batch normalization from the very first baseline, it should be introduced quickly if optimization appears to be problematic.

Unless your training set contains tens of millions of examples or more, you should include some mild forms of regularization from the start (e.g. early stopping, dropout, etc.).

A common question is whether to begin by using unsupervised learning. This is somewhat domain specific. Some domains, such as natural language processing, are known to benefit tremendously from unsuper- vised learning techniques such as learning unsupervised word embeddings. In other domains, such as computer vision, current unsupervised learning techniques do not bring a benefit.

9.1.4 Determine whether to gather more data

After the first end-to-end system is established, it is time to measure the performance of the algorithm and determine how to improve it. Many machine learning novices are tempted to make improvements by trying out many different algorithms. However, it is often much better to gather more data than to improve the learning algorithm.

First, determine whether the performance on the training set is acceptable. If performance on the training set is poor, the learning algorithm is not using the training data that is already available, so there is no reason to gather more data. Instead, try increasing the size of the model by adding more layers or adding more hidden units to each layer.

Also, try improving the learning algorithm, for example by tuning the learning rate hyperparameter. If large models and carefully tuned optimization algorithms do not work well, then the problem might be the quality of the training data. The data may be too noisy or may not include the right inputs needed to predict the desired outputs. This suggests starting over, collecting cleaner data or collecting a richer set of features.

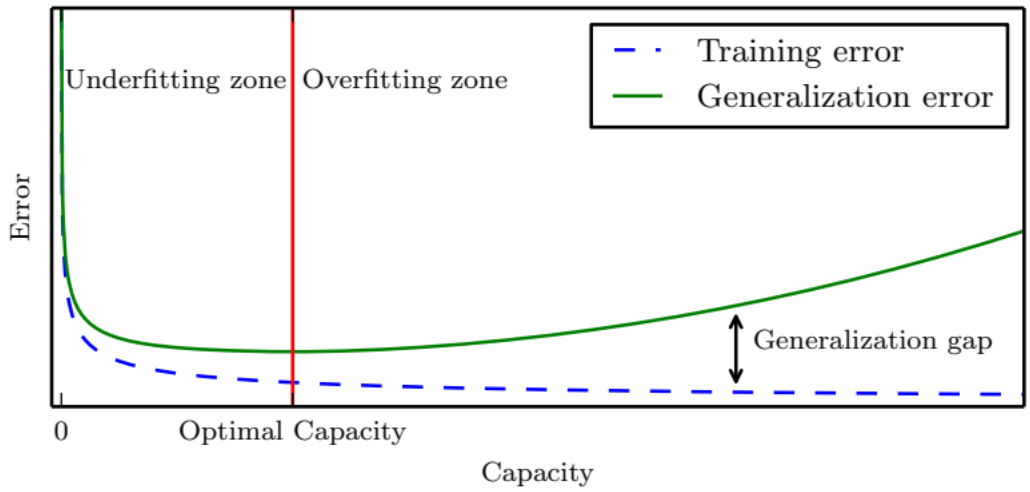
If test set performance is much worse than training set performance, then gathering more data is one of the most effective solutions. If gathering much more data is not feasible, the only other way to improve generalization error is to improve the learning algorithm itself. This becomes the domain of research and not the domain of advice for applied practitioners.

9.1.5 Selecting Hyperparameters

Most deep learning algorithms come with many hyperparameters that control many aspects of the algorithm's behavior. There are two basic approaches to choosing these hyperparameters: choosing them manually and choosing them automatically.

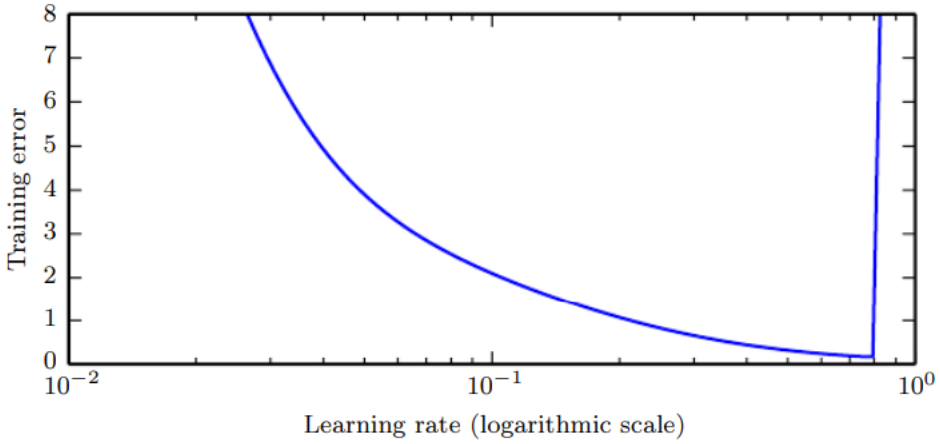
The primary goal of **manual hyperparameter** search is to adjust the effective capacity of the model to match the complexity of the task.

The generalization error typically follows a U-shaped curve



At one extreme, the hyperparameter value corresponds to low capacity, and generalization error is high because training error is high. This is the underfitting regime. At the other extreme, the hyperparameter value corresponds to high capacity, and the generalization error is high because the gap between training and test error is high. Somewhere in the middle lies the optimal model capacity, which achieves the lowest possible generalization error, by adding a medium generalization gap to a medium amount of training error.

The learning rate is perhaps the most important hyperparameter. The effective capacity of the model is highest when the learning rate is correct for the optimization problem, not when the learning rate is especially large or especially small. The learning rate has a U-shaped curve for training error.



Tuning the parameters other than the learning rate requires monitoring both training and test error to diagnose whether your model is overfitting or underfitting, then adjusting its capacity appropriately.

Manual hyperparameter tuning can work very well when the user has a good starting point, such as one determined by others having worked on the same type of application and architecture. However, for many applications, these starting points are not available. In these cases, **automated algorithms** can find useful values of the hyperparameters. It is possible, in principle, to develop hyperparameter optimization algorithms that wrap a learning algorithm and choose its hyperparameters. Unfortunately, hyperparameter optimization algorithms often have their own hyperparameters, such as the range of values that should be explored for each of the learning algorithm's hyperparameters.

When there are three or fewer hyperparameters, the common practice is to perform **grid search**. For each hyperparameter, the user selects a small finite set of values to explore. The grid search algorithm then trains a model for every joint specification of hyperparameter values in the Cartesian product of the set of values for each individual hyperparameter. The experiment that yields the best validation set error is then chosen as having found the best hyperparameters. The obvious problem with grid search is that its computational cost grows exponentially with the number of hyperparameters.

Fortunately, there is an alternative to grid search that is as simple to program, more convenient to use, and converges much faster to good values of the hyperparameters: **random search**

Lastly, there are other approaches for tuning hyperparameters, e.g., **model-based** (Bayesian). In this case the search for good hyperparameters is cast as an optimization problem. The decision variables are the hyperparameters. The cost to be optimized is the validation set error that results from training using these hyperparameters.

9.1.6 Debugging Strategies

Machine learning systems are difficult to debug for a variety of reasons. Some important debugging tests include:

- Visualize the model in action.
- Visualize the worst mistakes.
- Fit a tiny dataset: If you have high error on the training set, determine whether it is due to genuine underfitting or due to a software defect.
- Check the gradient (exploding or vanishing).
- Check the activations: how many neurons fire? ReLU can induce "dead" neurons.

Chapter 10

Lec 15 - Sequence Modeling

10.1 Learning in Sequential Domains

Why learning in sequential domains is different than static domains ? Because successive points in sequential data are **strongly correlated**. Machine learning models and algorithms for sequence learning have to consider that data points are not independent, deal with sequential distortions and/or variations (e.g. In speech, variations in speaking rate) and make use of **contextual information**.

With static data we usually learn:

$$P(\mathbf{o}|\mathbf{x})$$

where $\mathbf{x}$ is a fixed-size tuple of predictive attributes and $\mathbf{o}$ is a classification/regression task.

With **sequential data**, instead, $\mathbf{x}$ is a **sequence** $x^{(1)}, ..., x^{(t)}, ...$ where each $x^{(t)}$ has a static type. $\mathbf{o}$ may be either static (e.g., sequence classification) or a sequence.

Using mathematical induction, a **sequence** is either an external vertex, or an ordered pair (t, h) where the head h is a vertex and the tail t is a sequence.

10.1.1 Sequencial Transductions

Sequence Transduction is a machine learning task that involves converting an input sequence into an output sequence, potentially of different lengths.

Let X and O be the **input** and **output** label spaces. We denote by X^* the set of all sequences with labels in X. We can define a general transduction T as a function

$$T : X^* \to O^*$$

- $T(\cdot)$ has **finite memory** $k \in \mathbb{N}$ if $\forall \mathbf{x} \in X^*$ and $\forall t$, $T(x^{(t)})$ only depends on $\{\mathbf{x}^{(t)}, \mathbf{x}^{(t-1)}, ..., \mathbf{x}^{(t-k)}\}$
- $T(\cdot)$ is **algebraic** if it has 0 finite memory (i.e., no memory at all)
- A transduction $T(\cdot)$ is **causal** if the output at time t does not depend on future inputs (at time $t+1, t+2, ...$)

10.1.2 Learning Sequences

Sequences have variable length but typical machine learning models have a fixed number of inputs. In order to solve this problem we can:

- Limit context to a **fixed-size window**.
- Use **recurrent** models.
- Use **transformers** for non-causal sequences (e.g. text).

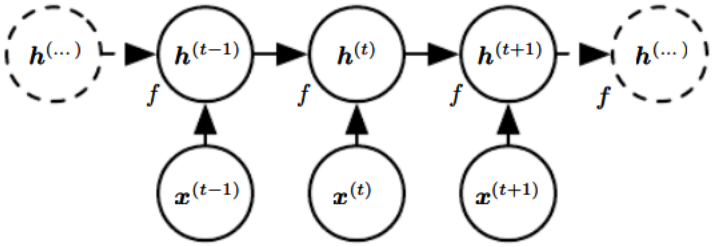
10.2 Recursive State Representation

In order to represent a recursive state we can use the following equations:

$$h^{(t)} = f(h^{(t-1)}, x^{(t)}, t)$$
$$o^{(t)} = g(h^{(t)}, x^{(t)}, t)$$

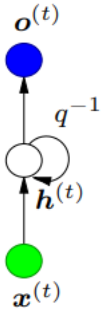
where f is the *state transition function* and g is the *output function*.

$h^{(t)}$ is called the state of the system and it's defined by a **recursive equation**. Indeed, the definition of h at time t refers back to the same definition at time $t-1$. It contains information about the whole past sequence. For a finite number of time steps τ, the graph can be unfolded by applying the definition $\tau - 1$ times. **Unfolding** the equation by repeatedly applying the definition yields an expression that does not involve recurrence. Such an expression can now be represented by a traditional directed acyclic computational graph.



The state transition function can be represented using the time shift operator q^{-1}:

$$q^{-1}h^{(t)} = h^{(t-1)}$$



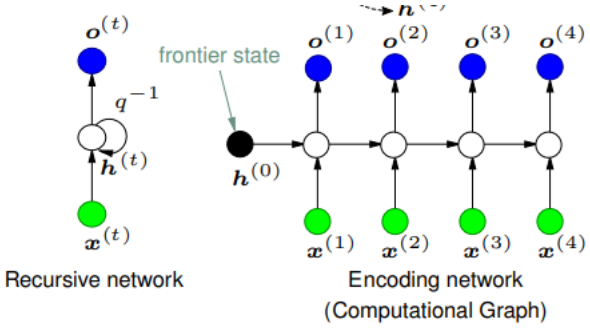
The unfolding process thus introduces two major advantages:

1. Regardless of the sequence length, the learned model always has the same input size, because it is specified in terms of transition from one state to another state, rather than specified in terms of a variable-length history of states.

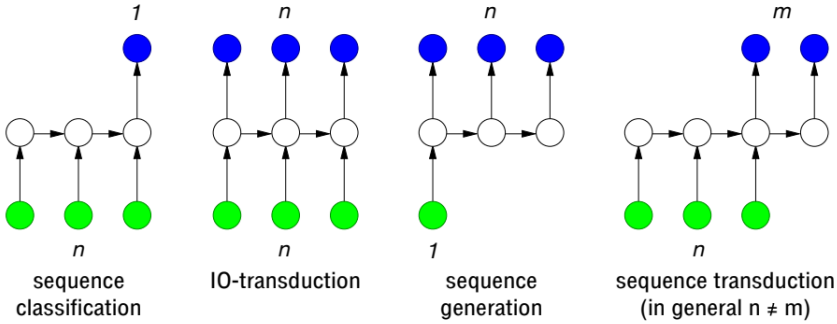
2. It is possible to use the same transition function f with the same parameters at every time step.

These two factors make it possible to learn a single model that operates on all time steps and all sequence lengths. Learning a single, shared model allows generalization to sequence lengths that did not appear in the training set, and allows the model to be estimated with far fewer training examples than would be required without parameter sharing.

Given a sequence $s \in X^*$ and a recursive transduction T, the *encoding network* associated to s and T is formed by unrolling (time unfolding) the recursive network of T through the input sequence s.



Examples of Sequential Transductions



T is **stationary** if $f(\cdot)$ and $g(\cdot)$ do not depend on t

There are different ways in which we can implement $f(\cdot)$ and $g(\cdot)$. There are two general families of models:

- Linear:
  - Kalman Filter
  - Hidden Markov Models
  - Linear Dynamical Systems
  - ...
- Nonlinear
  - Recurrent Neural Networks
  - ...

10.3 Shallow Recurrent Neural Networks

Armed with the graph unrolling and parameter sharing ideas, we can design a wide variety of recurrent neural networks. In general we have:

$$\mathbf{h}^{(t)} = f(\mathbf{U}\mathbf{x}^{(t)} + \mathbf{W}\mathbf{h}^{(t-1)} + \mathbf{b})$$
$$\mathbf{o}^{(t)} = g(\mathbf{V}\mathbf{h}^{(t)} + \mathbf{c})$$

where $f()$ and $g()$ are non-linear functions (e.g. $tanh()$ and $softmax$), and $h^{(0)} = 0$ (or can be learned jointly with the other parameters). $\mathbf{U}$ and $\mathbf{W}$ are weight matrices which parametrize **input-to-hidden** connections and **hidden-to-hidden** recurrent connections respectively. **Hidden-to-output** connections are parametrized by the weight matrix $\mathbf{V}$.

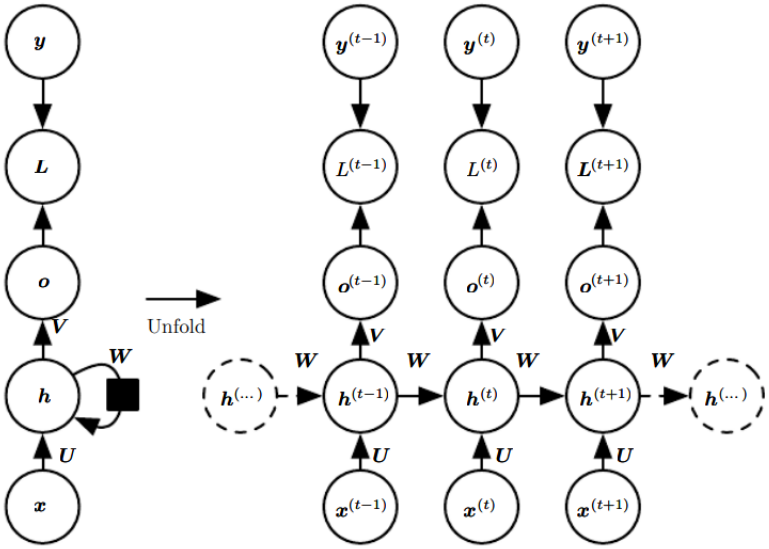
An example of RNN for IO-transduction with discrete outputs:

$$\mathbf{h}^{(t)} = tanh(\mathbf{U}\mathbf{x}^{(t)} + \mathbf{W}\mathbf{h}^{(t-1)} + \mathbf{b})$$
$$\mathbf{o}^{(t)} = \mathbf{V}\mathbf{h}^{(t)} + \mathbf{c}$$
$$\hat{\mathbf{y}} = softmax(\mathbf{o}^{(t)})$$
$$L = \sum_t L^{(t)} = -\sum_t log\, p_{model}(\mathbf{y}^{(t)}|\{\mathbf{x}^1, ..., \mathbf{x}^{(t)}\})$$

where:

- $o^{(t)}$ is the unnormalized log probabilities a time t
- $\mathbf{y}^{(t)}$ is the target vector a time t
- $p_{model}(\mathbf{y}^{(t)}|\{\mathbf{x}^1, ..., \mathbf{x}^{(t)}\}$ is given by reading the entry for $\mathbf{y}^{(t)}$ from the model's output vector $\hat{\mathbf{y}}^{(t)}$, that is, the loss L **internally computes** $\hat{\mathbf{y}}$.
- L is the loss function

The corresponding computation graph is the following



Some examples of important design patterns for recurrent neural networks include the following:

- Recurrent networks that produce an output at each time step and have recurrent connections between hidden units (IO-transduction).

- Recurrent networks that produce an output at each time step and have recurrent connections only from the output at one time step to the hidden units at the next time step

- Recurrent networks with recurrent connections between hidden units, that read an entire sequence and then produce a single output (e.g. for classification).

There are a lot of possible additional architectural features, such as short-cut connections, higher-order states, feedback from output, teacher forcing, bidirectional RNN, etc[1]. All these architectural features (and others...) are orthogonal, i.e. they can be combined together.

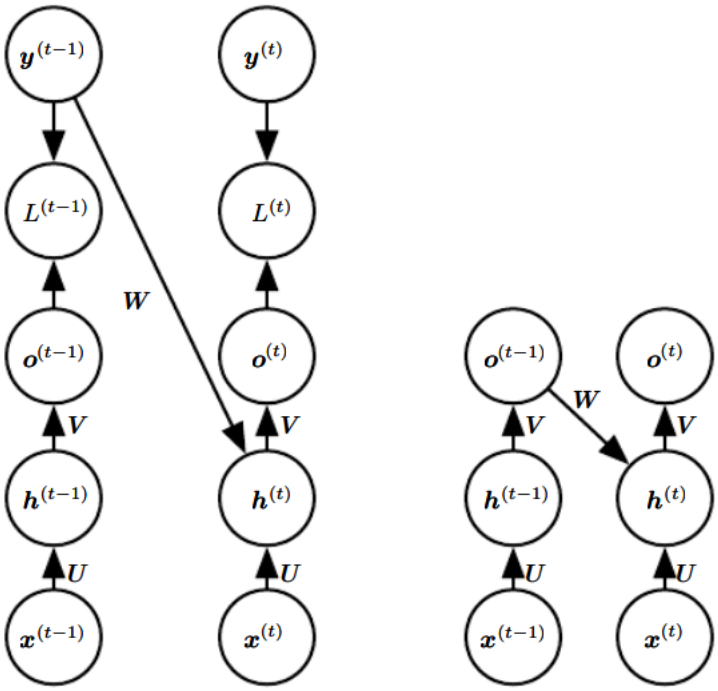
10.3.1 Teacher Forcing

The network with recurrent connections only from the output at one time step to the hidden units at the next time step is strictly less powerful because it lacks hidden-to-hidden recurrent connections. Therefore, it requires that the output units capture all of the information about the past that the network will use to predict the future. Because the output units are explicitly trained to match the training set targets, they are unlikely to capture the necessary information about the past history of the input.

For this reason, models that have recurrent connections from their outputs leading back into the model may be trained with **teacher forcing**. Teacher forcing is a procedure in which during training the model receives the ground truth output $\mathbf{y}^{(t)}$ as input at time $t+1$. When the model is deployed, the true output is generally not known. In this case, we approximate the correct output $\mathbf{y}^{(t)}$ with the model's output $\mathbf{o}^{(t)}$, and feed the output back into the model.

$$
\begin{aligned}
\mathbf{h}^{(t)} &= tanh(\mathbf{U}\mathbf{x}^{(t)} + \mathbf{W}\mathbf{y}^{(t-1)} + \mathbf{b}) \\
\mathbf{o}^{(t)} &= \mathbf{V}\mathbf{h}^{(t)} + \mathbf{c} \\
\hat{\mathbf{y}} &= softmax(\mathbf{o}^{(t)}) \\
L &= -\sum_{t} log\, p_{model}(\mathbf{y}^{(t)} | \mathbf{y}^{(1)}, ..., \mathbf{y}^{(t-1)}, \mathbf{x}^{(1)}, ..., \mathbf{x}^{(t)})
\end{aligned}
$$

[1] See slides for further information

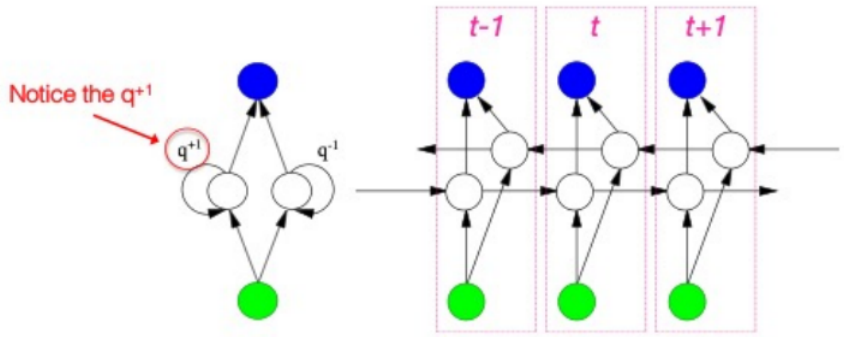


The advantage of eliminating hidden-to-hidden recurrence is that, for any loss function based on comparing the prediction at time t to the training target at time t, all the time steps are decoupled. Training can thus be parallelized, with the gradient for each step t computed in isolation.

10.3.2 Bidirectional RNNs

in many applications we want to output a prediction of $\mathbf{y}^{(t)}$ which may depend on the whole input sequence. For example, if there are two interpretations of the current word that are both plausible, we may have to look far into the future (and the past) to disambiguate them. Bidirectional recurrent neural networks (or bidirectional RNNs) were invented to address that need.

As the name suggests, bidirectional RNNs combine an RNN that moves forward through time, beginning from the start of the sequence, with another RNN that moves backward through time, beginning from the end of the sequence.

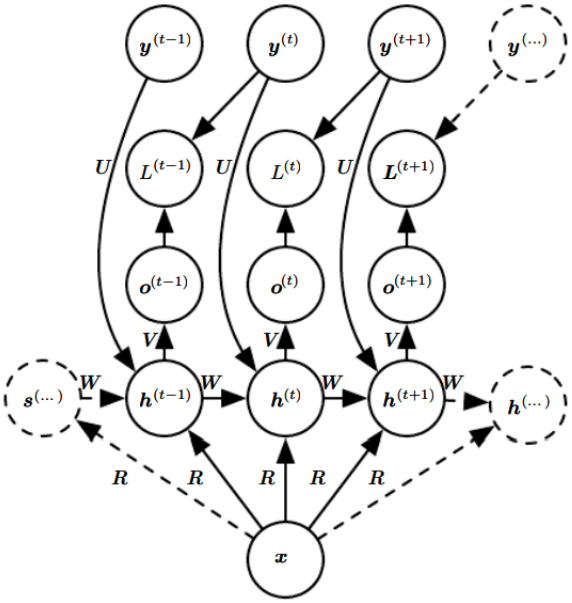


$$\boldsymbol{o}^{(t)} = g(\boldsymbol{V}^{(p)}\mathbf{h}_p^{(t)} + \boldsymbol{V}^{(f)}\mathbf{h}_f^{(t)})$$

$$\underbrace{\mathbf{h}_p^{(t)} = f(\boldsymbol{U}^{(p)}\boldsymbol{x}^{(t)} + \boldsymbol{W}^{(p)}\boldsymbol{h}_p^{(t-1)})}_{\textit{past history}} \qquad \underbrace{\mathbf{h}_f^{(t)} = f(\boldsymbol{U}^{(f)}\boldsymbol{x}^{(t)} + \boldsymbol{W}^{(f)}\boldsymbol{h}_f^{(t+1)})}_{\textit{future history}}$$

10.3.3 1 to n transduction

Previously, we have discussed RNNs that take a sequence of vectors $\mathbf{x}^{(t)}$ for $t = 1, ..., \tau$ as input. Another option is to take only a single vector $\mathbf{x}$ as input. When $\mathbf{x}$ is a fixed-size vector, we can simply make it an extra input of the RNN that generates the $\mathbf{y}$ sequence. The interaction between the input $\mathbf{x}$ and each hidden unit vector $\mathbf{h}^{(t)}$ is parametrized by a newly introduced weight matrix $\mathbf{R}$.



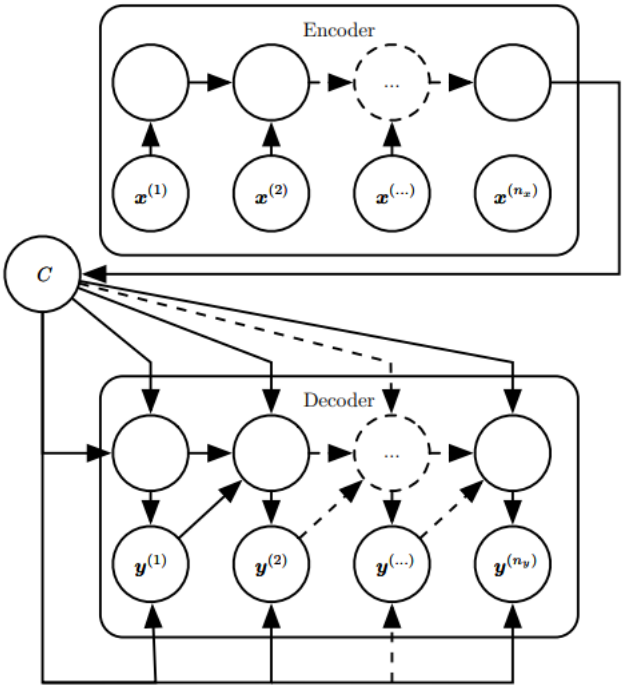
Each element $\mathbf{y}^{(t)}$ of the observed output sequence serves both as input (for the current hidden unit at time t) and, during training, as target (for the previous output unit at time $t-1$).

This RNN is appropriate for tasks such as image captioning, where a single image is used as input to a model that then produces a sequence of words describing the image.

10.3.4 Encoder-Decoder Sequence-to-Sequence Architectures

Here we discuss how an RNN can be trained to map an input sequence to an output sequence which is not necessarily of the same length. This comes up in many applications, such as speech recognition, machine translation, etc.

An encoder-decoder RNN architecture is is composed of an encoder RNN that reads the input sequence and a decoder RNN that generates the output sequence. The final hidden state of the encoder RNN is used to compute a generally fixed-size context variable C which represents a semantic summary of the input sequence and is given as input to the decoder RNN.



If the context C is a vector, then the decoder RNN is simply a vector-to-sequence RNN.

Chapter 11

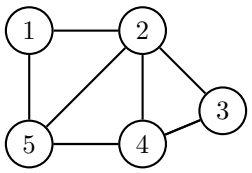
Lec 16 - Graph Neural Networks

11.1 Introduction

Traditional ML approaches have been developed assuming data to be encoded into feature vectors; however, many important real-world applications generate data that are naturally represented by more complex structures, such as graphs. Graphs are particularly suited to represent the relations (arcs) between the components (nodes) constituting an entity. For instance, in social network data, single data "points" (i.e., users) are closely inter-related.

A graph $G = (V, E)$ can be represented using the so called **adjacency matrix**. A $n \times n$ matrix A such that $A[i, j] = 1$ if $edge(i, j) \in E$, 0 otherwise.

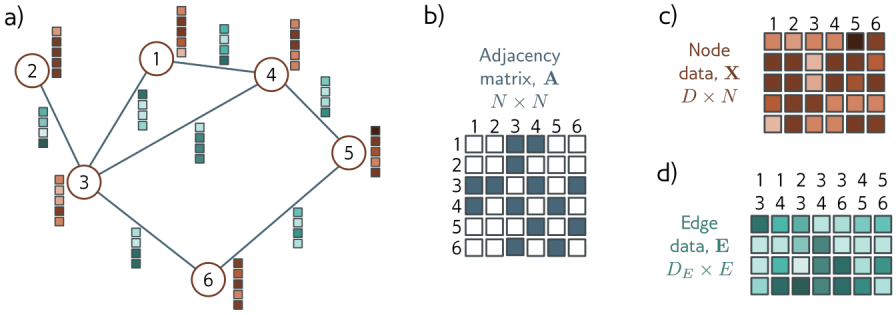
Example:



The Adjacency matrix of the graph above is the following:

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 \end{bmatrix}$$

In undirected graphs this matrix is **symmetric**, while in directed graphs it is **asymmetric**. In case of a weighted graph, each cell of the matrix has either the value of the edge weight w or $-$. Each node and edge is represented by a feature vector:



The position (m, n), of the adjacency matrix contains the number of walks of length one from node m to node n. Position (m, n) of the **squared** adjacency matrix A^2 contains the number of walks of length two from node m to node n.

The main problem settings that can arise when dealing with structured data are the following:

- Predictions over **nodes** in a network: In this setting, the dataset is composed of a single (possibly disconnected) large graph. Each example is a node in the graph, and the learning tasks are defined as predictions over the nodes. Given an unseen node u, the task is to predict the correct target y_u. An example in this setting is the prediction of properties of a social network user based on his or her connections.
- Predictions over **graphs**: In this case, each example is composed of a whole graph, and the learning tasks are predictions of properties of the whole graphs. An example is the prediction of toxicity in humans of chemical compounds represented by their molecular graph.
- Link-prediction tasks: the model predicts whether or not there should be an edge between nodes.

11.2 Learning on graphs is difficult

Let $\mathbf{X}$ be the matrix in which the feature vectors of each node are stored. We can observe that node indexing in graphs is arbitrary. This means that, differently from images, permuting the node indices results in a permutation of the columns of $\mathbf{X}$ and a permutation of both the rows and columns of A. However, the underlying graph is unchanged. This property is called Permutation Invariance. More formally, given a permutation matrix $\mathbf{P}$, we get a different representation of the same graph:

$$\mathbf{X}' = \mathbf{XP}$$
$$\mathbf{A}' = \mathbf{P}^T\mathbf{AP}$$

This property can give the intuition about why learning on graphs is difficult. In fact, determining if two graphs are equal (graphs isomorphism) is a problem for which are not known polynomial-time algorithms. Furthermore, sub-graph isomorphism, which is the problem of determining if a graph is a sub-graph of another graph, is NP-Complete. These problems affect machine learning because a model should be able to predict the same output for isomorphic graphs (which can be represented in different ways). Furthermore, the model we design should capture the similarity between two graphs (sub-graph isomorphism).

In general, the main problems we face when learning on graphs are the following:

1. Same graph can be represented in different ways;

2. How to recognize that a given graph G_2 is a sub-graph of G_1
3. How to represent graphs of different sizes (i.e., different number of nodes) into fixed-size vectors without loosing expressiveness ?
4. How to avoid explosion in the number of parameters with the size of the graphs?

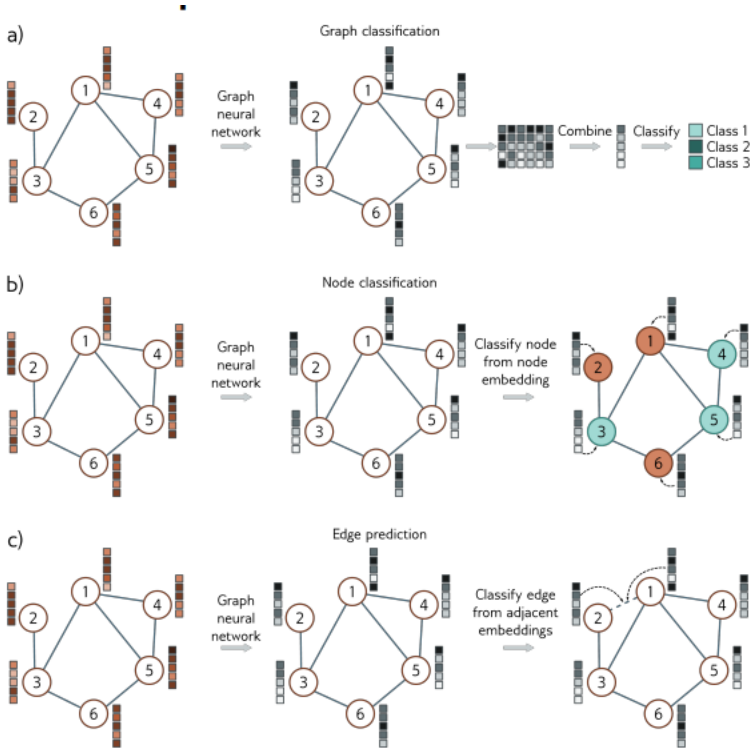
Problem 3 is commonly faced by using recursive models that exploit a causal state space [Sperduti & Starita., TNN 1997], while Problem 4 is commonly faced by exploiting shared parameters.

Regarding Problems 1 and 2, a sound and meaningful representation for graphs can be achieved by using a neural network with **convolution operator**, defined on graphs.

11.3 Graph Neural Networks - General Idea

A Graph Neural Network (GNN) receives in input a graph (adj. matrix and node representations. for simplicity) and passes it through a series of k layers. Each layer computes a hidden representation for each node, with the last layer computing the final nodes' embeddings $\mathbf{H}_k$. Similarly to CNNs, each node representation includes information about the node and its context within the graph.

- For **node-level tasks**, the output is computed from $\mathbf{H}_k$;
- For **graph-level tasks**, the nodes' embeddings are combined (e.g., by averaging), and the resulting vector is mapped via a linear transformation or neural network to a fixed-size vector from which the classification/regression task is performed.
- For **link-prediction tasks**, the embeddings of the two endpoint nodes must be mapped to a single number representing the probability that the edge is present (e.g. dot product of the nodes' embeddings and pass the result through a sigmoid function to create a probability).



11.4 Graph Convolution

The general idea of graph convolution starts from a parallel between graphs and images. GNNs implement convolution in a similar way how CNNs do, that is, learning the features by inspecting neighboring nodes. GNNs generalize the definition of convolution for non-regular structured data.

11.4.1 NN4G by Micheli

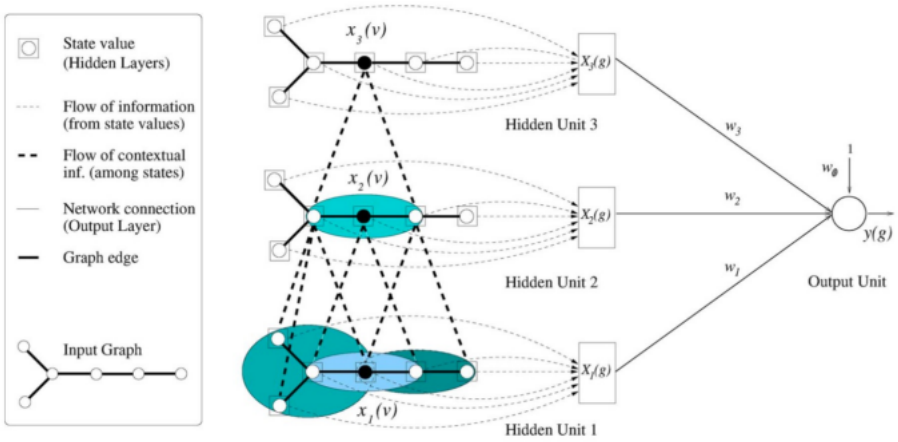
NN4G is an architecture based on a graph convolution that is defined as:

$$\boldsymbol{h}_v^1 = \sigma(\bar{W}^1 \boldsymbol{x}_v)$$
$$\boldsymbol{h}_v^i = \sigma\left(\bar{W}^i \boldsymbol{x}_v + W^i \sum_{u \in N(v)} \boldsymbol{h}_u^{i-1}\right), i > 1$$

where:

- σ is a nonlinear activation function applied element-wise.
- $N(v)$ represent the neighborhood of node v.
- $\mathbf{W}^i$ is a weights' matrix;
- $\mathbf{x}_u$ is the feature vector of node u.

Actually, this is a simplified notation, since the original one uses skip connections (see GNN book chapter).



Note that:

- The first layer ($i = 1$), which has no previous layers, computes the nodes' representations only on the basis of each vertex feature vector.
- Each convolution performed on the i-th hidden units, with $i > 1$, takes as input the neighbors' representations of the previous layer. Basically, it merges the representations of each node with those of its neighbors:
- $X_1(g), X_2(g), X_3(g)$ are scalar values computed by aggregating the representations $\mathbf{x}_i(g)$ for each unit i. In particular they are defined as:

$$X_i(g) = \frac{1}{k} \sum_{v \in Vert(g)} x_i(v)$$

if $k = 1$, this corresponds to a sum. Basically, we compute a representation per-graph per-layer. Then, this 3 representations are parametrized by the weights w_1, w_2, w_3 and used to compute the output for the whole graph (graph-level task).

The convolutional operation presented above can be defined in a compact way using matrix multiplications:

$$H^1 = \sigma(\overline{W}^1 X)$$
$$H^i = \sigma\left(\overline{W}^i X + W^i A H\right), i > 1$$

where **A** is the adjacency matrix. Exploiting matrix multiplications makes the computation really fast. Furthermore, note that, as for images, the receptive field of the layers increases as we stack more layers.

Note also that, since with the convolution operator we are merging neighboring nodes' representations, isomorphic graphs in which the order of the nodes is changed will have the same nodes' representations.

11.4.2 Graph Fourier Transform

The operation described above is graph convolution, but how it is derived? Defining the formal convolution operator on graph is difficult.

Let $x : V \to \mathbb{R}$ be a signal on the nodes V of the graph G, i.e., a function that associates a real value with each node of V. We can represent every signal as a vector $\mathbf{x} \in \mathbb{R}^n$, which from now on we will refer to as signal. In order to set up a convolutional network on G, we need the notion of convolution between a signal $\mathbf{x}$ and a filter signal $\mathbf{f}$.

The key idea is to use a Fourier transform. In the frequency domain, thanks to the **Convolution Theorem**, the (undefined) convolution of two signals becomes the (well-defined) component-wise product of their transforms. So, if we knew how to compute the Fourier transform of a function defined on a graph, we could define the convolution operator.

The Convolution Theorem states that convolution in one domain (time, space) corresponds to pointwise multiplication in frequency domain.

The **graph Fourier transform** is defined starting from the (normalized) Laplacian matrix of the graph, which is defined as:

$$L = I_n - D^{-\frac{1}{2}} A D^{-\frac{1}{2}}$$

where:

- I_n is the identity matrix;
- A is the adjacency matrix;
- D is the degree matrix, that is, the **diagonal** matrix containing the number of edges attached to each vertex;

Then, we can compute the eigendecomposition of L (which is always possible):

$$L = U \Lambda U^T$$

where $\Lambda = diag([\lambda_0, ..., \lambda_{n-1}])$ and U is the Fourier basis of the graph.

Finally, Given a spatial signal $\mathbf{x}$:

- $\hat{\mathbf{x}} = U^T\mathbf{x}$ is its graph Fourier Transform
- $\mathbf{x} = U\hat{\mathbf{x}}$ is the inverse Fourier transform

Therefore, convolution between a parametric filter and a signal can be defined as:

$$y = \mathbf{f}_\theta *_G \mathbf{x} = U\left((U^T\mathbf{f}_\theta) \odot (U^T\mathbf{x})\right)$$

It can be proved that this operator corresponds to the one used for NN4G presented previously (see slides for more details).

11.5 Aggregation Layer for graph classification

With Graph Convolution we have a representation for each graph node. How can we map node representations to a graph-level representation? There are some simple solutions, like the sum or the average of nodes' representations as we saw previously, or we can rely on more complex alternatives: Universal readout.

11.6 Graph Recurrent Neural Networks

Scarselli et al. proposed a network architecture where, instead of stacking multiple layers, a single recurrent layer is adopted:

$$\mathbf{h}_v^{t+1} = \sum_{u \in N(v)} f(\mathbf{h}_u^t, \mathbf{x}_v, \mathbf{x}_u)$$

where f is a function (e.g. neural network) with shared parameters across all the nodes and all the time steps. The recurrent system is defined as a contraction mapping, and thus it is guaranteed to converge to a fixed point **h***.

11.6.1 Gated Graph Neural Networks

The idea is to remove the constraint for the recurrent system to be a contraction mapping, and implement this idea by adopting recurrent neural networks to define the recurrence. Specifically, the gated recurrent unit (GRU) is adopted. The recurrent convolution operator is defined as follows:

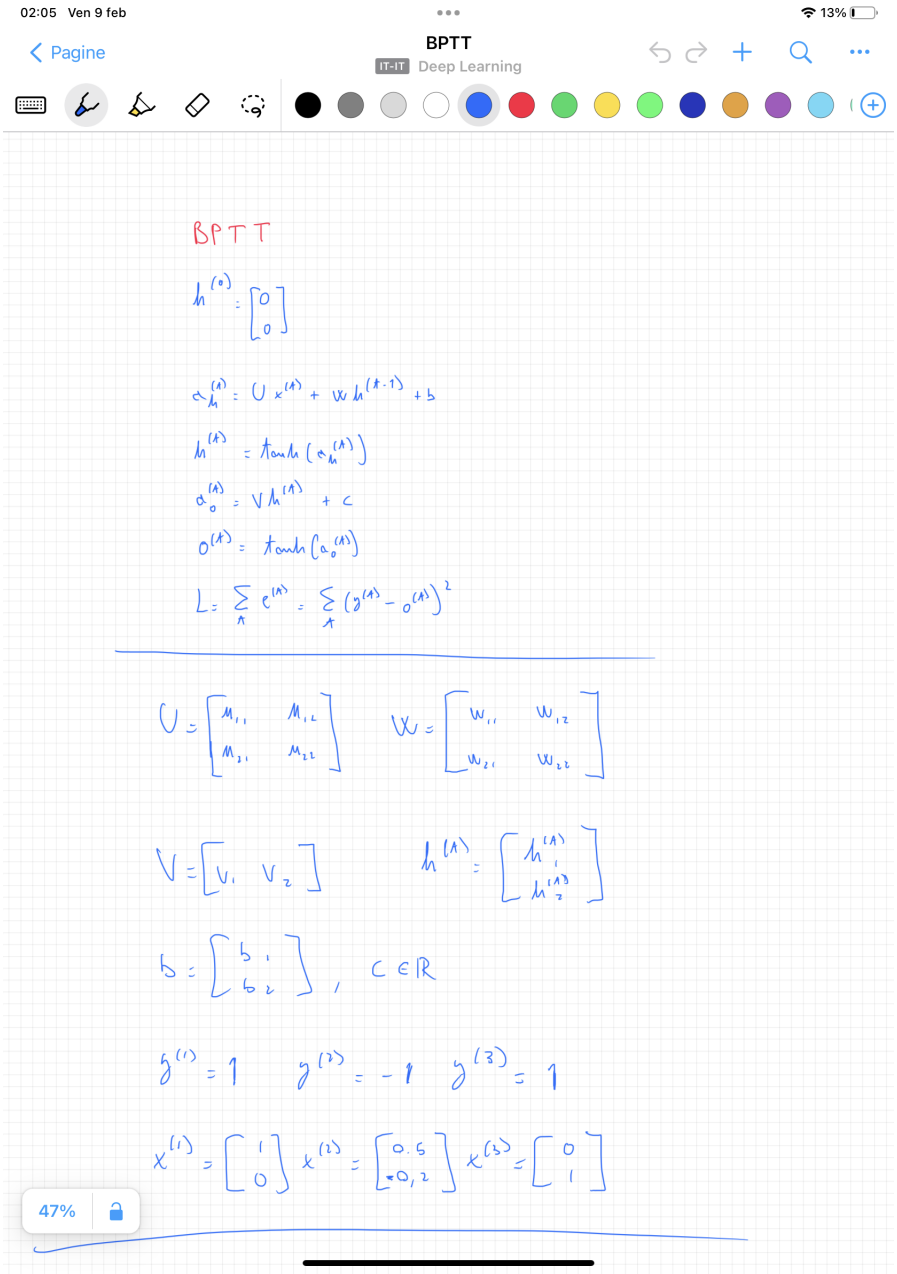
$$\begin{aligned}
\mathbf{h}_v^{(1)} &= [\mathbf{x}_v, \mathbf{0}], \\
\mathbf{a}_v^{(t)} &= \mathbf{A}_v[\mathbf{h}_v^{(t-1)}, \forall v \in V], \\
\mathbf{z}_v^t &= \sigma(\mathbf{W}^z\mathbf{a}_v^{(t)} + \mathbf{U}^z\mathbf{h}_v^{(t-1)}), \\
\mathbf{r}_v^t &= \sigma(\mathbf{W}^r\mathbf{a}_v^{(t)} + \mathbf{U}^r\mathbf{h}_v^{(t-1)}), \\
\mathbf{c}_v^t &= \tanh(\mathbf{W}\mathbf{a}_v^{(t)} + \mathbf{U}(\mathbf{r}_v^t \odot \mathbf{h}_v^{(t-1)})), \\
\mathbf{h}_v^{(t)} &= (1 - \mathbf{z}_v^t) \odot \mathbf{h}_v^{(t-1)} + \mathbf{z}_v^t \odot \mathbf{c}_v^t,
\end{aligned}$$

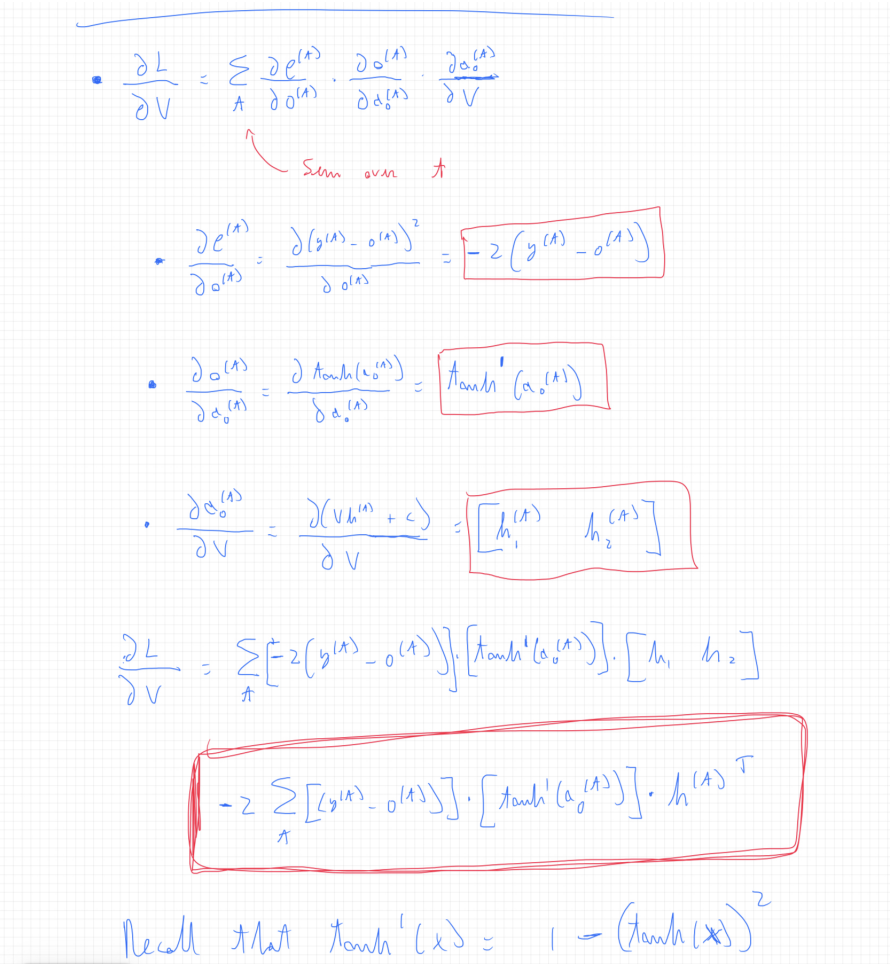
where $\mathbf{A}_v$ is row v of the adjacency matrix $\mathbf{A}$.

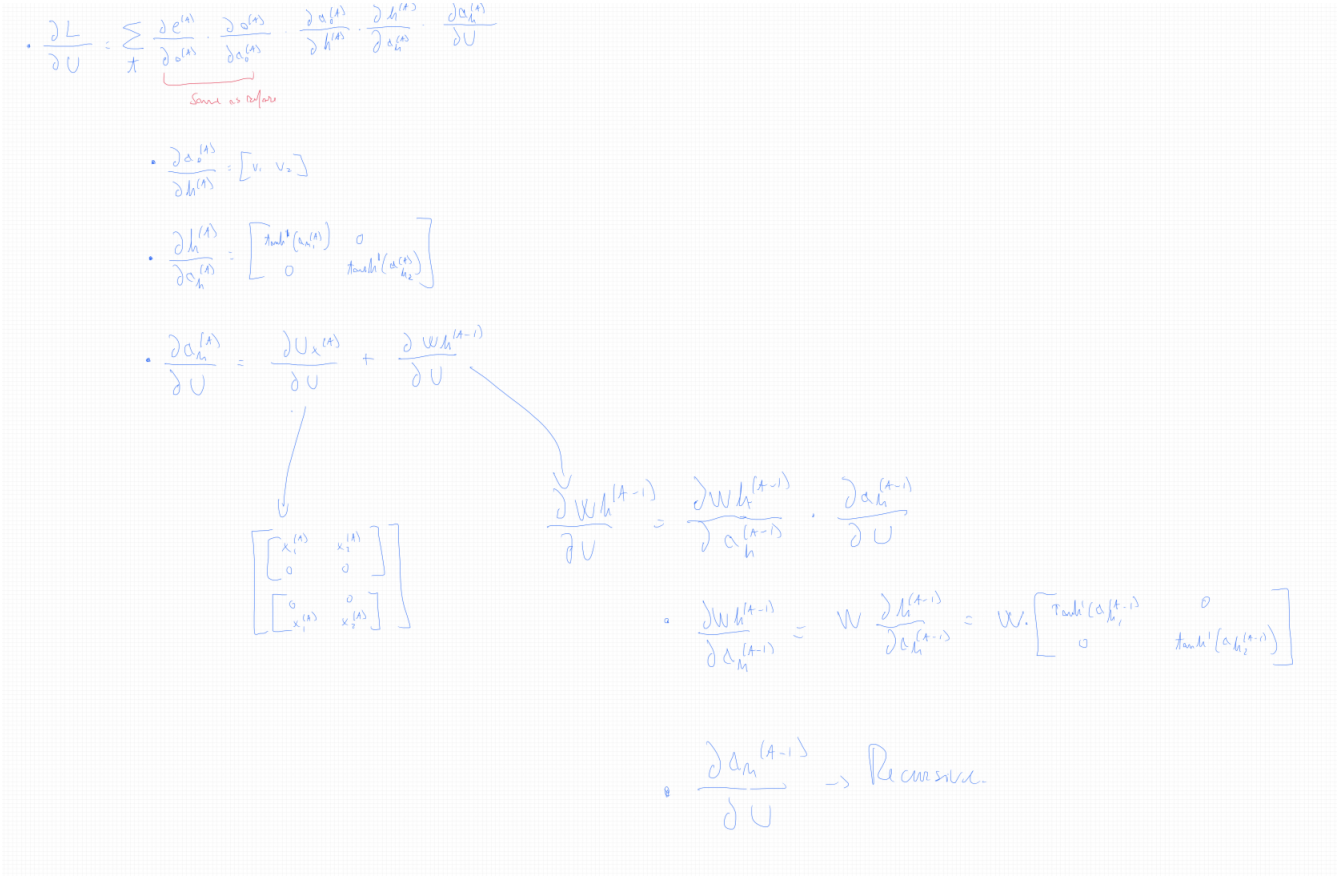
Chapter 12

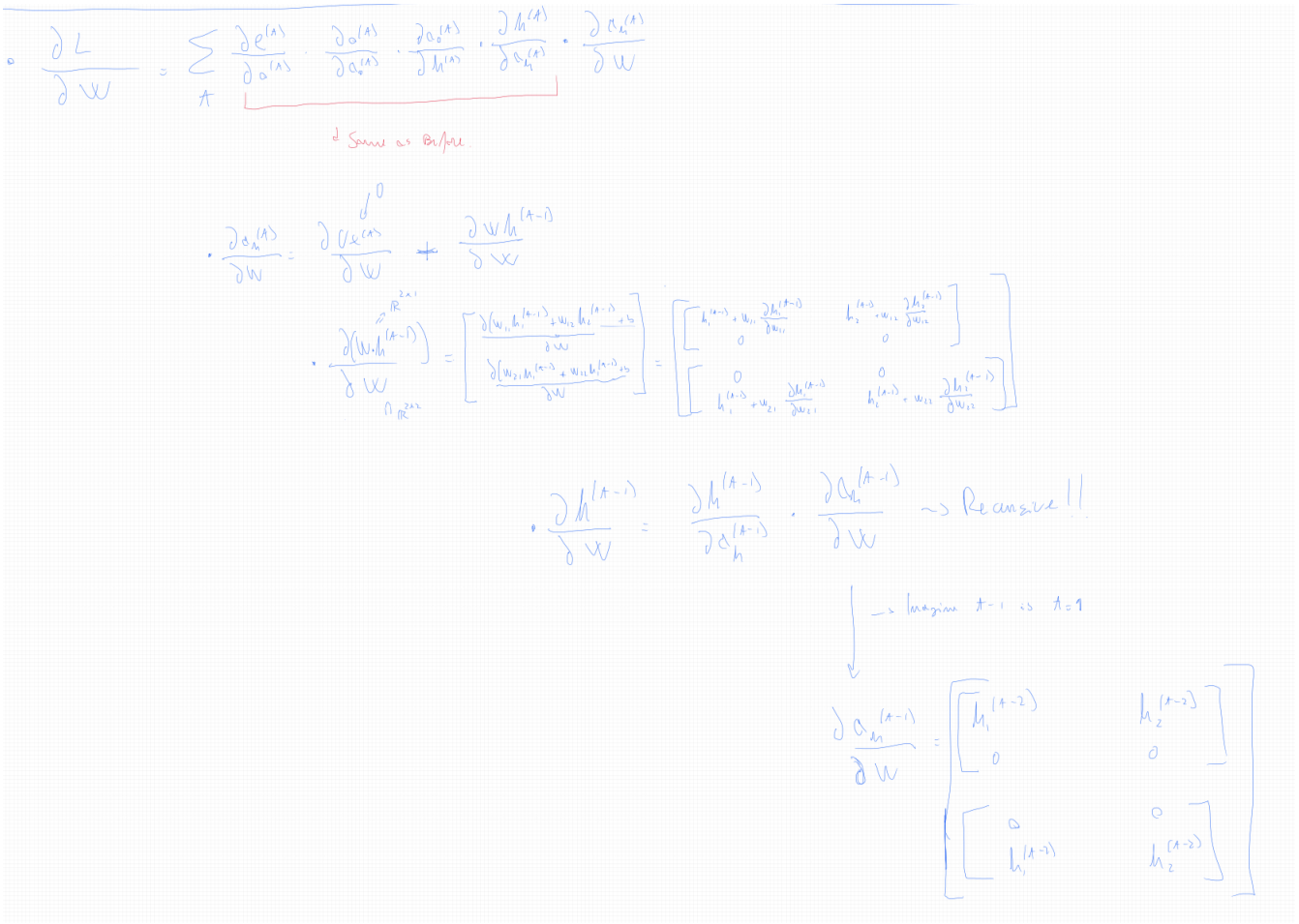
Back-Propagation Through Time

12.1 BPTT


02:05 Ven 9 feb
13%
Pagine
BPTT
IT-IT Deep Learning
BPTT
$h^{(0)} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$
$a_h^{(t)} = U x^{(t)} + W h^{(t-1)} + b$
$h^{(t)} = \tanh(a_h^{(t)})$
$a_o^{(t)} = V h^{(t)} + c$
$o^{(t)} = \tanh(a_o^{(t)})$
$L = \sum_t e^{(t)} = \sum_t (y^{(t)} - o^{(t)})^2$
$U = \begin{bmatrix} u_{11} & u_{12} \\ u_{21} & u_{22} \end{bmatrix}$ $W = \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix}$
$V = \begin{bmatrix} v_1 & v_2 \end{bmatrix}$ $h^{(1)} = \begin{bmatrix} h_1^{(1)} \\ h_2^{(1)} \end{bmatrix}$
$b = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$, $c \in \mathbb{R}$
$y^{(1)} = 1$ $y^{(2)} = -1$ $y^{(3)} = 1$
$x^{(1)} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ $x^{(2)} = \begin{bmatrix} 0.5 \\ -0.2 \end{bmatrix}$ $x^{(3)} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$
47%


$\frac{\partial L}{\partial V} = \sum_t \frac{\partial e^{(t)}}{\partial o^{(t)}} \cdot \frac{\partial o^{(t)}}{\partial a_o^{(t)}} \cdot \frac{\partial a_o^{(t)}}{\partial V}$
Sum over t
$\frac{\partial e^{(t)}}{\partial o^{(t)}} = \frac{\partial (y^{(t)} - o^{(t)})^2}{\partial o^{(t)}} = -2\left(y^{(t)} - o^{(t)}\right)$
$\frac{\partial o^{(t)}}{\partial a_o^{(t)}} = \frac{\partial \tanh(a_o^{(t)})}{\partial a_o^{(t)}} = \tanh'(a_o^{(t)})$
$\frac{\partial a_o^{(t)}}{\partial V} = \frac{\partial (V h^{(t)} + c)}{\partial V} = \left[h_1^{(t)} \quad h_2^{(t)}\right]$
$\frac{\partial L}{\partial V} = \sum_t \left[-2\left(y^{(t)} - o^{(t)}\right)\right] \cdot \left[\tanh'(a_o^{(t)})\right] \cdot \left[h_1 \quad h_2\right]$
$-2 \sum_t \left[\left(y^{(t)} - o^{(t)}\right)\right] \cdot \left[\tanh'(a_o^{(t)})\right] \cdot h^{(t)T}$
Recall that $\tanh'(x) = 1 - (\tanh(x))^2$


Same as before
Recursive


Same as Before
Recursive !!
Imagine t-1 is t=1

Chapter 13

Lec 18 - Long-Term Dependencies

13.1 Learning Long-Term Dependencies

The basic problem of learning long-term dependencies is that gradients propagated over many stages tend to either vanish (most of the time) or explode[1] (rarely, but with much damage to the optimization).

A long-term dependency is when the desired output at time t depends on the input at time $t - \tau$, with $t > \tau >> 1$ (e.g. $\mathbf{x}^{(t-100)} \rightarrow \mathbf{y}^{(t)}$).

This means that, for the Recurrent Neural Network to output the correct desired $\mathbf{y}^{(t)}$, it has to recognize its dependency on $\mathbf{x}^{(t-\tau)}$, and use $\mathbf{x}^{(t-\tau)}$ in the generation of $\mathbf{y}^{(t)}$.

Here are some approaches to try to reduce the vanishing/exploding gradients problem:

- Architectural
  - Long Short-Term Memory or Gated Recurrent units
  - Reservoir Computing: Echo State Networks and Liquid State Machines
- Algorithmic
  - Clipping gradients (avoids exploding gradients)
  - Hessian Free Optimization
  - Smart Initialization: pre-training techniques

13.2 Long Short-Term Memory

Long Short Term Memory networks - usually just called "LSTMs" - are a special kind of RNN, capable of learning long-term dependencies.

They are based on the idea of creating paths through time that have derivatives that neither vanish nor explode.

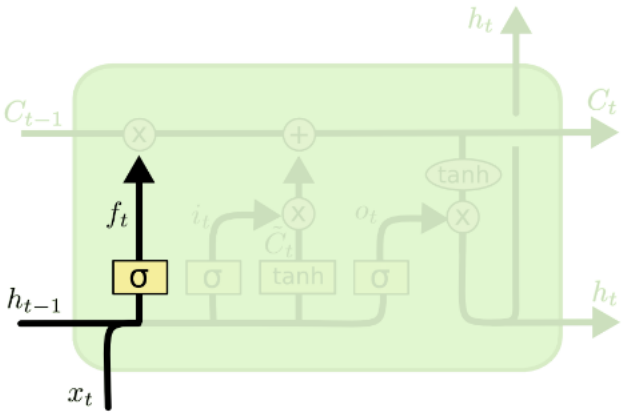
The mechanism allows the networks to "remember" relevant information for a long period of time and to "forget" them when they are no more relevant.

[1] a problem when large error gradients accumulate and result in very large updates to neural network model weights during training

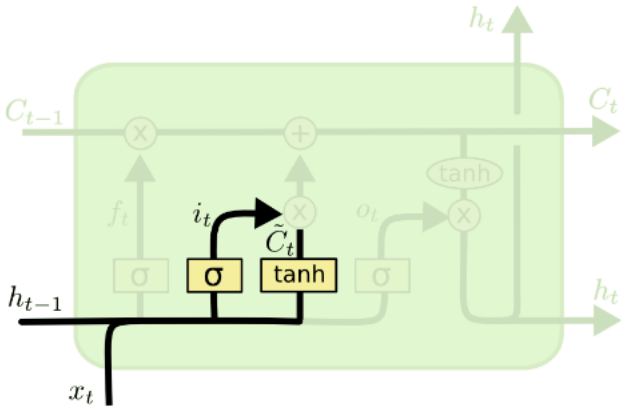
The LSTM does have the ability to remove or add information to the cell state, carefully regulated by structures called gates. Gates are a way to optionally let information through. They are composed out of a sigmoid neural net layer and a pointwise multiplication operation. The sigmoid layer outputs numbers between zero and one, describing how much of each component should be let through. A value of zero means "let nothing through," while a value of one means "let verything through!"

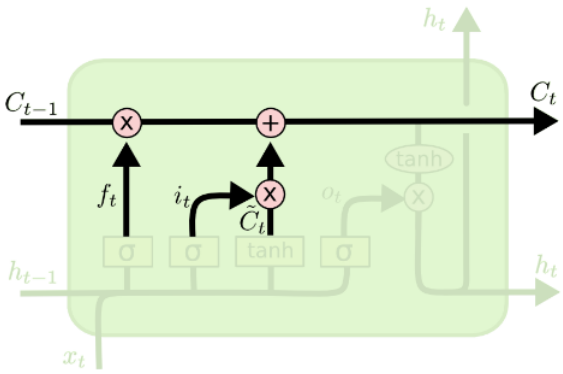
An LSTM has three of these gates, to protect and control the cell state:

1. Forget gate: The sigmoid layer called the "forget gate layer " *decides* what information we're going to throw away from the cell state. It looks at $h^{(t-1)}$ and $x^{(t)}$, and outputs a number between 0 and 1 for each number in the cell state $C^{(t-1)}$. A 1 represents "completely keep this" while a 0 represents "completely get rid of this." Let f_t be its output. The forget gate multiplies the old state by f_t, forgetting the things it decided to forget.



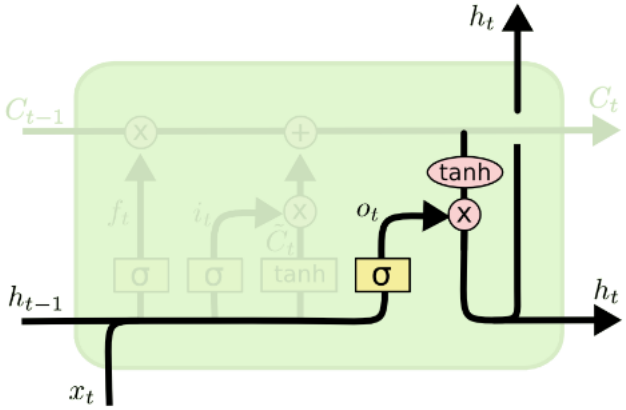
2. Input gate: It decides what new information are going to be stored in the cell state. This has two parts. First, a sigmoid layer called the "input gate layer" decides which values are going to be updated. Let i^t be its output. Next, a tanh layer creates a vector of new candidate values, $\tilde{C}^t$, that could be added to the state. The input gate computes $i^t * \tilde{C}^t$. The output of this gate is added to the output of the forget gate to determine the new cell state.





$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

3. Output gate: It deterines what parts of the cell state are going to be outputted. It puts the cell state through tanh (to push the values to be between -1 and 1). The result is multiplied by the output of a sigmoid layer so that it only outputs the parts it decided to.



There are a lot of variations of the LSTM architecture. One popular variant is adding "peephole connections." This means that we let the gate layers look at the cell state. Other variations are:

- No Input Gate (NIG)
- No Forget Gate (NFG)
- No Output Gate (NOG)
- No Input Activation Function (NIAF)
- No Output Activation Function (NOAF)

However, vanilla LSTM performs reasonably well in general and variations do not significantly improve the performance. Furthermore, the forget gate is crucial for LSTM performance.

13.3 Simplifying LSTM: Gated Recurrent Units

The main difference between GRU and LSTM is that GRU uses a single gating unit that simultaneously controls the forgetting factor and the decision to update the state unit.

$$\boldsymbol{h}^{(t)} = \boldsymbol{z}^{(t)} \odot \boldsymbol{h}^{(t-1)} + (\mathbf{1} - \boldsymbol{z}^{(t)}) \odot \underbrace{\sigma\left(\boldsymbol{U}\boldsymbol{x}^{(t)} + \boldsymbol{W}(\boldsymbol{r}^{(t)} \odot \boldsymbol{h}^{(t-1)})\right)}_{\tilde{\boldsymbol{h}}^{(t)}}$$

The update gate $\mathbf{z}$ selects whether the hidden state need to be updated with a new hidden state $\tilde{\mathbf{h}}$. The reset gate $\mathbf{r}$ decides whether the previous hidden state is ignored.

The values for $\mathbf{z}$ and $\mathbf{r}$ are defined as usual using the sigmoidal layers as in LSTM.

Basically, the idea is that if the $\mathbf{z}$ vector has a value equal to 0 in position i, when we compute the element-wise multiplication between $\mathbf{z}$ and $\mathbf{h}^{(t-1)}$ the i-th value in $\mathbf{h}^{(t-1)}$ will be cancelled. On the other hand, the i-th element in $(1 - \mathbf{z})$ is 1. Therefore, the i-th element of $\mathbf{h}^{(t-1)}$ will be updated with the i-th element of $\tilde{\mathbf{h}}$.

13.4 Reservoir Computing

Reservoir Computing is an umbrella term used to identify a general framework of computation derived from Recurrent Neural Networks (RNN). This technique can be implemented with **Echo State Networks** and **Liquid State Machines**. The idea is to fix the input-to-hidden and hidden-to-hidden connections at random values and only learn the output units connections. The intuition was born from the fact that in training RNNs most of the times the weights showing most change were the ones in the last layer.

The first part of the system, called Reservoir, is an RNN with fixed weights that acts as "black-box" model of a complex system; The second one is known as Readout, a classifier layer of some kind, usually a simple linear one, connected by a set of weights to the Reservoir.

One way to think about these reservoir computing recurrent networks is that they are similar to kernel machines: they map an arbitrary length sequence (the history of inputs up to time t) into a fixed-length vector (the recurrent state $\mathbf{h}^{(t)}$), on which a linear predictor (typically a linear regression) can be applied to solve the problem of interest.

How do we set the input and recurrent weights so that a rich set of histories can be represented in the recurrent neural network state? in order to produce a "rich" set of dynamics, the reservoir should

- be big (hundreds to thousands units).
- be sparsely (hidden weight matrix W up to 20% possibile connections) and randomly (uniform distribution symmetric around zero) connected.
- satisfy the echo state property, i.e., the ability to forget information from the far past (or the effect of $\mathbf{x}^{(t)}$ and $\mathbf{h}^{(t)}$ on the future state should vanish gradually as time passes). This means that the spectral radius $\rho(\mathbf{W}) < 1$, i.e, $\mathbf{W}$ is contractive.
- On the contrary, the input (U) and optional output feedback weight matrices are dense (still random with uniform distribution).

Echo State Networks are composed of standard standard recurrent neurons plus leaky integrators, while **Liquid State Machines** implements spiking integrate-and-fire neurons and dynamic synaptic connection models. A leaky integrator is defined as follows:

$$\mathbf{h}^{(t)} = (1 - a)\mathbf{h}^{(t-1)} + \sigma(\mathbf{U}\mathbf{x}^{(t)} + \mathbf{W}\mathbf{h}^{(t-1)})$$

Basically, it adds a portion of the previous state representation (according to a) to the new state representation.

If the network is too contractive, it will forget too quickly information from the past. In order to overcome this problem we can use the **intrinsic plasticity** approach. The main idea is to exploit the full range of output of the activation function of the hidden units. IP is a computationally efficient online learning rule to adjust threshold and gain of sigmoid reservoir neurons. It drives the neurons' output activities to approximate exponential distributions. The exponential distribution maximizes the entropy of a non-negative random variable with a fixed mean, thus enabling the neurons to transmit maximal information

To evaluate a RC network we use memory capacity, which tells us if the internal state of the network can reproduce input from the far past :

$$\sum_{k=0}^{\infty} r^2(\mathbf{x}^{(t-k)}, \mathbf{o}_k^{(t)})$$

where $r^2(\mathbf{x}^{(t-k)}, \mathbf{o}_k^{(t)})$ is the squared correlation coefficient between the input $\mathbf{x}^{(t-k)}$ with delay k and the corresponding output $\mathbf{o}_k^{(t)}$ generated by the network at time t for delay k.

13.5 Deep Recurrent Networks

The computation in most RNNs can be decomposed into three blocks of parameters and associated transformations:

1. from the input to the hidden state,
2. from the previous hidden state to the next hidden state, and
3. from the hidden state to the output.

With the RNN architecture, each of these three blocks is associated with a single weight matrix. In other words, when the network is unfolded, each of these corresponds to a shallow transformation. By a shallow transformation, we mean a transformation that would be represented by a single layer within a deep MLP. Typically this is a transformation represented by a learned affine transformation followed by a fixed nonlinearity.

Experimental evidence shows a significant advantage if the state of an RNN is decomposed into multiple layers. We can think of the lower layers in the hierarchy as playing a role in transforming the raw input into a representation that is more appropriate at the higher levels of the hidden state.

However, in general, it is easier to optimize shallow architectures and adding depth may hurt learning by making optimization difficult.

Chapter 14

Lec 19 - Transformers

14.1 Transformers

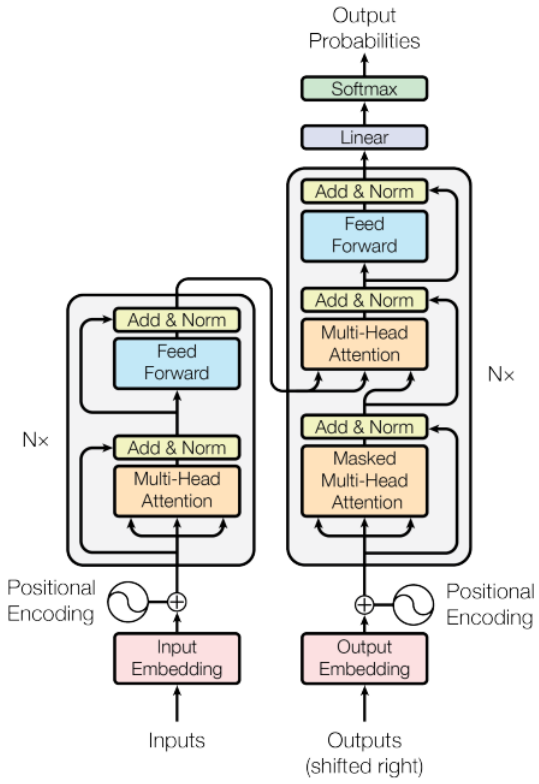
The vanilla Transformer is a sequence-to-sequence model typically used for Machine translation and consists of an encoder and a decoder, each of which is a stack of N identical blocks. The encoder maps an input sequence of symbol representations $(x_1, ..., x_n)$ to a sequence of continuous representations $\mathbf{z} = (z_1, ..., z_n)$. Given $\mathbf{z}$, the decoder then generates an output sequence $(y_1, ..., y_m)$ of symbols one element at a time. At each step the model is auto-regressive, consuming the previously generated symbols as additional input when generating the next. Basically, the encoder first encodes the full sentence, then the decoder decodes one word at time.

- **Encoder:** The encoder is composed of a stack of N[1] identical layers. Each layer has two sub-layers. The first is a multi-head self-attention mechanism, and the second is a simple, positionwise fully connected feed-forward network. For building a deeper model, a residual connection is employed around each module, followed by Layer Normalization module.

- **Decoder**: Compared to the encoder blocks, decoder blocks additionally insert cross-attention modules over the output of the encoder stack between the multi-head self-attention modules and the position-wise FFNs. Furthermore, the self-attention modules in the decoder are adapted to prevent each position from attending to subsequent positions.

Transformer is a model that uses **attention** to boost the speed. More specifically, it uses self-attention. Transformer allows for significantly more parallelization.

[1] $N = 6$ in the original paper



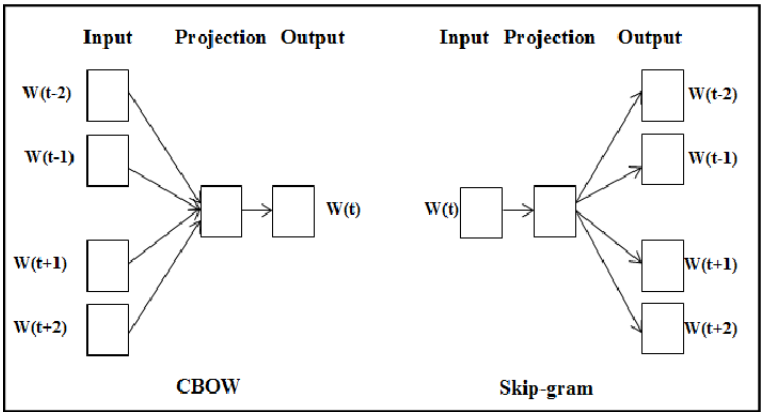
14.2 Input Embedding

An embedding is a vector that **semantically** represents an object/input. In the context of NLP, the goal is to transform a text (set of words) into a vector of numbers such that similar words produce similar vectors. The **word2vec** technique is based on a feed-forward, fully connected architecture. It is similar to an autoencoder, but rather than performing input reconstruction, word2vec trains words according to other words that are neighbors in the input corpus.

Word2vec can learn word embedding in two different ways:

- **CBOW** (Continuous Bag of Words) in which the neural network uses the context to predict a target word.

- **Skip-gram** in which it uses the target word to predict a target context.



14.3 Positional Encoding

Position and order of words are the essential parts of any language. They define the grammar and thus the actual semantics of a sentence. Recurrent Neural Networks (RNNs) inherently take the order of word into account. They parse a sentence word by word in a sequential manner. This will integrate the words' order in the backbone of RNNs.

Since the model contains no recurrence and no convolution, in order for the model to make use of the order of the sequence, we must inject some information about the relative or absolute position of the tokens in the sequence.

One possible solution to give the model some sense of order is the **positional encoding**.

Positional encoding describes the location or position of an entity in a sequence so that each position is assigned a unique representation. There are many reasons why a single number, such as the index value, is not used to represent an item's position in transformer models. For long sequences, the indices can grow large in magnitude. If you normalize the index value to lie between 0 and 1, it can create problems for variable length sequences as they would be normalized differently.

Transformers use a smart positional encoding scheme, where each position/index is mapped to a vector. Hence, the output of the positional encoding layer is a matrix, where each row of the matrix represents an encoded object of the sequence summed with its positional information.

Suppose you have an input sequence of length L. The positional encoding is given by sine and cosine functions of varying frequencies:

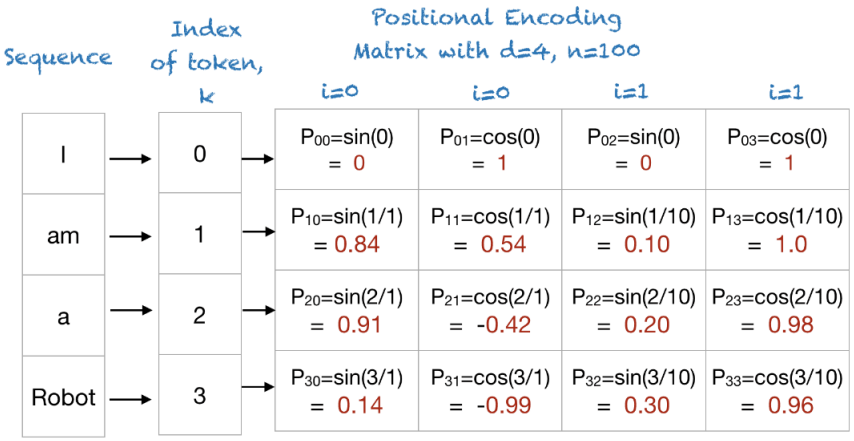
$$PE_{(k,2i)} = sin(k/n^{2i/d})$$
$$PE_{(k,2i+1)} = cos(k/n^{2i/d})$$

where:

- k is the position of an object in the input sequence.
- d is the dimension of the output embedding space.
- $PE_{(k,j)}$ is the position function for mapping a position k in the input sequence to index j of the positional matrix.

- n is a user-defined scalar, set to 10000 by the authors of the original Transformer's paper.
- i is used for mapping to column indices $0 \leq i < d/2$, with a single value of i maps to **both** sine and cosine functions.

You can also imagine the positional embedding as a vector containing pairs of sines and cosines for each frequency.

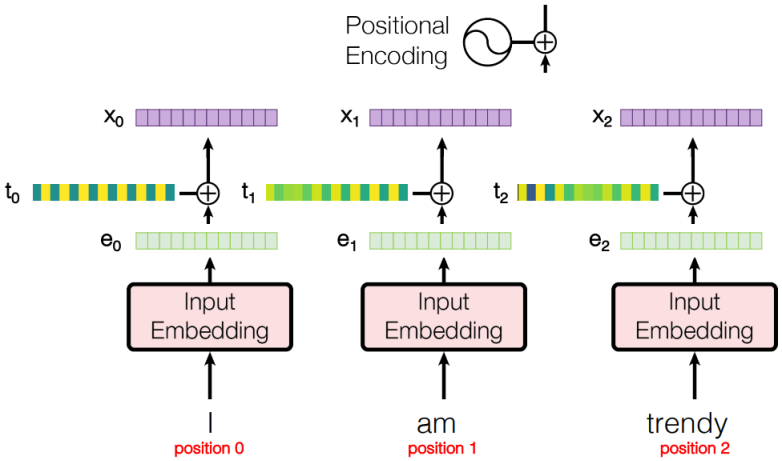


Positional Encoding Matrix for the sequence 'I am a robot'

The scheme for positional encoding has a number of advantages:

- The sine and cosine functions have values in $[-1, 1]$, which keeps the values of the positional encoding matrix in a normalized range.
- As the sinusoid for each position is different, you have a unique way of encoding each position.
- You have a way to add positional information to words embedding in order to encode the relative positions of words (e.g. the words 'brother' and 'sister' will probably have similar embedding representations, but if one word is used in a very different position than the other, they may not be correlated. In order to get different representations, we add positional information).

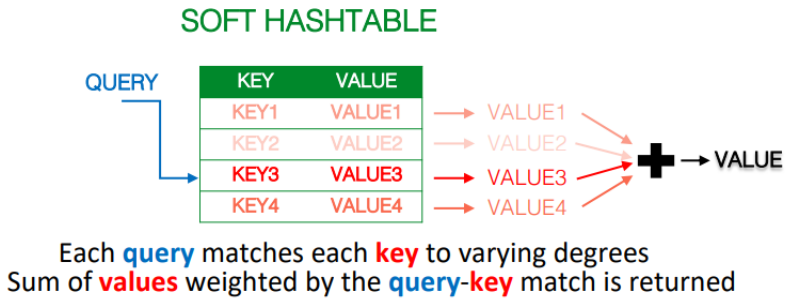
The positional encoding layer sums the positional vector with corresponding word embedding vector.



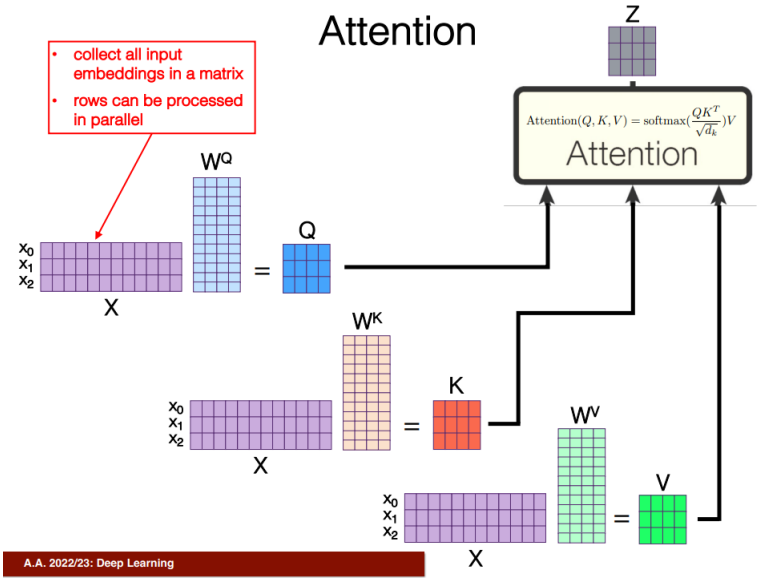
14.4 Attention

In order for the decoding to be precise, it needs to take into account every word of the input, using attention.

An attention function can be described as mapping a query and a set of key-value pairs to an output, where the query, keys, values, and output are all vectors. The output is computed as a weighted sum of the values, where the weight assigned to each value is computed by a compatibility function of the query with the corresponding key.



The first step in calculating attention is to create three vectors from each of the encoder's input vectors. So for each word, we create a **Query vector**, a **Key vector**, and a **Value vector**. These vectors are created by multiplying the embedding by three matrices ($\mathbf{W^Q}, \mathbf{W^K}, \mathbf{W^V}$) that we trained during the training process.



If we collect all the query, keys and value vectors in three matrices ($\mathbf{Q}, \mathbf{K}, \mathbf{V}$), we can define the attention function as follows:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

By taking the dot product between the query matrix and the key matrix, we compute a *score* for each word of the input sentence against the others. The score determines how relevant is a word with respect to the others. The dot-products of queries and keys are divided by $\sqrt{d_k}$ (dimensionality of the embedding vector) to alleviate gradient vanishing problem of the softmax function. The softmax normalizes the scores so they're

all positive and add up to 1. softmax $\left(\frac{QK^T}{\sqrt{d_k}}\right)$ is often called **attention matrix**.

Then, we compute the dot product between the attention matrix and the value matrix $\mathbf{V}$. The intuition here is to keep intact the values of the relevant word(s), and drown-out irrelevant words (by multiplying them by tiny numbers in the attention matrix).

14.4.1 Multi-Head Attention

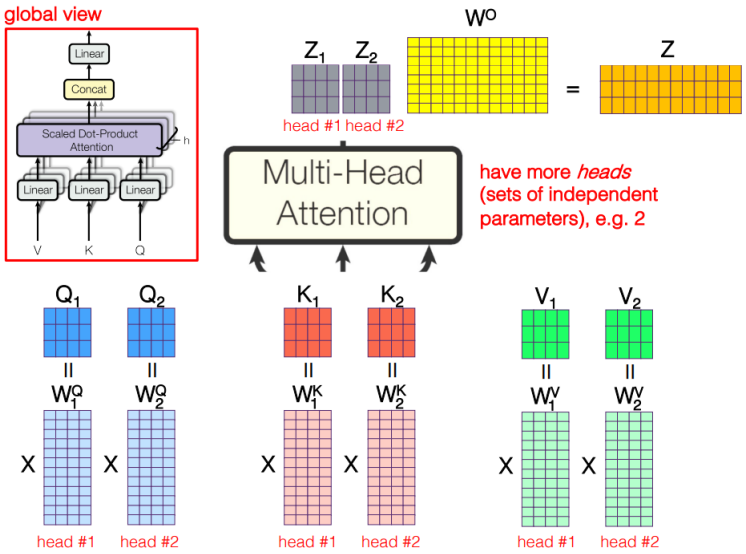
Instead of simply applying a single attention function, Transformer uses **multi-head attention**, where queries keys and values are linearly projected h times with different, learned linear projections to d_k, d_k and d_v dimensions, respectively. On each of these projected versions of queries, keys and values we then perform the attention function in parallel, yielding d_v-dimensional output values. These are concatenated and once again projected, resulting in the final values.

Multi-head attention allows the model to jointly attend to information from different representation subspaces at different positions.

$$\text{MultiHead}(Q, K, V) = \text{Concat}(head_1, ..., head_h)\mathbf{W^O}$$

where

$$head_i = Attention(\mathbf{QW}_i^\mathbf{Q}, \mathbf{KW}_i^\mathbf{K}, \mathbf{VW}_i^\mathbf{V})$$



14.4.2 Applications of Attention in the Model

The Transformer uses multi-head attention in three different ways:

- In "encoder-decoder attention" layers (cross-attention), the queries come from the previous decoder layer, and the memory keys and values come from the output of the encoder. This allows every position in the decoder to attend over all positions in the input sequence.
- The encoder contains **self-attention** layers. In a self-attention layer all of the keys, values and queries come from the same place, in this case, the output of the previous layer in the encoder stack. we set $\mathbf{Q} = \mathbf{K} = \mathbf{V} = \mathbf{Z}$, where $\mathbf{Z}$ is the output of the previous encoder layer.

- Masked Self-attention: We need to prevent leftward information flow in the decoder to preserve the auto-regressive property. Therefore, in the Transformer decoder, the self-attention is restricted such that queries at each position can only attend to all key-value pairs up to and including that position. This is implemented inside of scaled dot-product attention by masking out (setting to $-\infty$) all values in the input of the softmax which correspond to illegal connections.

14.5 Add & Norm and Feed Forward Network

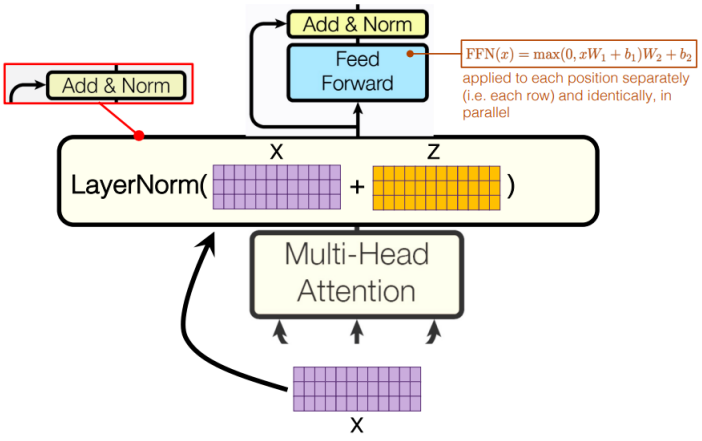
In addition to attention sub-layers, each of the layers in the encoder and decoder contains a fully connected feed-forward network, which is applied to each position separately and identically. This consists of two linear transformations with a ReLU activation in between.

$$FFN(h) = max(0, hW_1 + b_1)W_2 + b_2$$

where h is the output of the previous layer.

In order to build a deep model, Transformer employs a residual connection around each module, followed by Layer Normalization. For instance, each Transformer encoder block may be written as

$$\mathbf{H'} = \text{LayerNorm}(\text{SelfAttention}(\mathbf{X}) + \mathbf{X})$$
$$\mathbf{H} = \text{LayerNorm}(FFN(\mathbf{H'}) + \mathbf{H'})$$



Chapter 15

Lec 20 - Transformers II

15.1 RNNs vs. Transformers

Long-term dependencies:

- RNNs have problems in dealing with long-term dependencies between words that are spread far apart in a long sentence;
- Transformers do not have the above problem, as long as the long-term dependencies are in the range of the maximum allowed input length;

Parallel computation:

- RNNs process the input sequence sequentially one token at a time: before starting the computation for time step t, the computation for time step $t-1$ should be completed; training and inference are slowed down;
- Transformers can process in parallel all the tokens in the input sequence (as well as all sequences in a batch) exploiting matrix multiplication (very efficient in GPUs);

Context fragmentation:

- Attention can only deal with fixed-length sequences, so long sequences should be split into a certain number of segments (chunks) before being fed into a Transformer;
- RNNs do not have the above problem;

Out-of-distribution generalization:

- Transformers are not able to implement recurrent rules (if they exist in data), so in principle they do not generalize well to sequences longer than the training ones;
- RNNs in principle can learn recurrent rules (if they exist in data);

Attention in Transformers scales quadratically with the length of the input sequence. Let D be the hidden dimension of the transformer (size embeddings times #heads). Let T be the length of the input sequence.

Module	Complexity
self-attention	$O(T^2 \cdot D)$
position-wise FFN	$O(T \cdot D^2)$

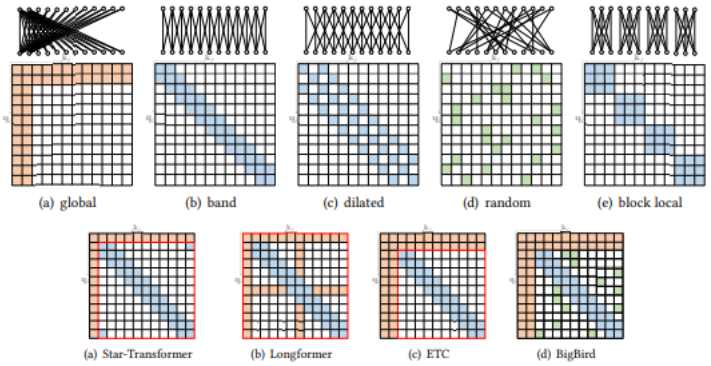
Then, when the input sequences are short, the hidden dimension D dominates the complexity of self-attention and position-wise FFN. The bottleneck of Transformer thus lies in FFN. However, as the input sequences grow longer, the sequence length T gradually dominates the complexity of these modules, in which case self-attention becomes the bottleneck of Transformer.

Furthermore, self-attention does no assume any structural bias over inputs. Even the order information is also needed to be learned from training data. Therefore, Transformer (w/o pre-training) is usually easy to overfit on small or moderate-size data.

15.2 X-formers

X-formers improve the vanilla Transformer from different perspectives:

- **Model Efficiency:** In the standard self-attention mechanism, every token needs to attend to all other tokens. However, it is observed that for the trained Transformers the learned attention matrix $\mathbf{A}$ is often very sparse across most data points. Therefore, it is possible to reduce computation complexity by incorporating structural bias to limit the number of query-key pairs that each query attends to.



  Another effective way of dealing with long sequences is to use **divide-and-conquer** strategy, i.e., to decompose an input sequence into finer segments that can be efficiently processed by Transformer or Transformer modules.

- **Model Generalization:** Transformers can be pre-trained on large-scale unlabeled data.
- **Model Adaptation:** Transformers adapted to specific downstream tasks and applications.

15.3 Model Pre-training

Recent studies suggest that Transformer models that are pre-trained on large corpora can learn universal language representations that are beneficial for downstream tasks. The models are pre-trained using various self-supervised objectives:

- **Decoder only.**
- **Encoder only.**
- **Encoder-Decoder.**

After pre-training a model, one can simply fine-tune it on downstream datasets, instead of training a model from scratch.

15.3.1 Language model via a Decoder

The idea is to train a decoder to predict the next word/token in the sentence. Then, we fine-tune the resulting model to perform the learning task.

Generative Pre-Trained Transformer (GPT)

Given a corpus of unsupervised tokens $U = \{u_1, ..., u_n\}$, we first train a multi-layer transformer Decoder to maximize:

$$L_1(U) = \sum_i log\, P(u_i|u_{i-k}, ..., u_{i-1}, \Theta)$$

with i number of layers

$$h_0 = UW_e + W_p$$
$$h_l = \texttt{transformer\_block}(h_{l-1}) \forall i \in [1, n]$$
$$P(u) = \texttt{softmax}(h_n W_e^T)$$

Then, after the pre-training phase, we add on top of the decoder a softmax linear classifier and fine-tune the whole model (decoder + linear classifier) on the main learning task.

The first version of GPT was a Transformer decoder with 12 layers, 768-dimensional hidden states, 3072-dimensional feed-forward hidden layers; Byte-pair encoding with 40,000 merges; Trained on BooksCorpus: over 7000 unique books.

GPT-2 is a Larger version of GPT (1.5 billion parameters) trained on more data, was shown to produce relatively convincing samples of natural language.

GPT-3 is a Huge model: 175 billion parameters.

15.3.2 Masked Language Model via encoder

Decoders only take left context, while encoders are bidirectional (both left and right context), so how to train an encoder to perform Language Modeling ? We can perform **Mask Language Modeling** (MLM). Basically, we replace some fraction of words in the input with a special [MASK] token, and the self-supervised learning task is to predict these masked words.

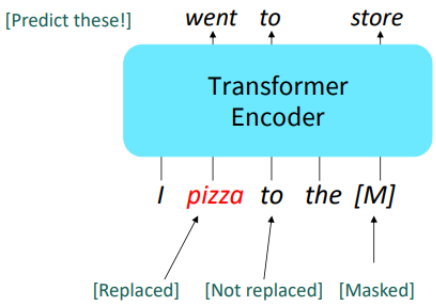
For example:

- original sentence: I went to the store.
- masked sentence: I [M] to the [M].

BERT: Bidirectional Encoder Representations from Transformers

A model based on MLM is BERT. The idea is to predict a random 15% of (sub)word tokens. In particular, the words are *masked* in 3 different ways:

- Replace the input word with [MASK] 80% of the time;
- Replace the input word with a random token 10% of the time;
- Leave the input word unchanged 10% of the time (but still predict it!)



This is done in order to make the model able to learn strong representations of non-masked words too.

Two models were released:

- BERT-base: 12 layers, 768-dim hidden states, 12 attention heads, 110 million params.
- BERT-large: 24 layers, 1024-dim hidden states, 16 attention heads, 340 million params.

Overall pre-training and fine-tuning procedures for BERT. Apart from output layers, the same architectures are used in both pre-training and fine-tuning.

15.3.3 Language Model via encoder-decoder

Encoders don't naturally lead to effective autoregressive (1-word-at-a-time) generation methods as decoders do. The idea is to use an encoder-decoder architecture in order to make the model able to perform both natural language understanding and generation.

In order to do this, we can replace different-length spans from the input with unique placeholders, and decode out the spans that were removed.

Chapter 16

Lec 21 - Autoencoders

16.1 Autoencoder - General Idea

Autoencoders (AEs) are an unsupervised learning technique based on feed forward neural networks. The aim of autoencoders is to learn a representation (often called encoding or code) for a set of data, typically for the purpose of dimensionality reduction.

In particular, an autoencoder is a neural network that is trained to attempt to copy its input to its output. Internally, it has a hidden layer $\mathbf{h}$ that describes a code used to represent the input. The network may be viewed as consisting of two parts:

- An **encoder** function $\mathbf{h} = f(\mathbf{x})$
- A **decoder** function that produces a reconstruction $\mathbf{r} = g(\mathbf{h})$

Since it is not useful to learn the identity function on the whole input domain (hidden code dimension equal to input dimension), autoencoders are trained to learn $\mathbf{x} = g(f(\mathbf{x}))$ with constraints:

- on the architecture of the network (**undercomplete autoencoder**).
- adding a regularizing term to the loss (**overcomplete autoencoder**)

Usually they are restricted in ways that allow them to copy only approximately. Because the model is forced to prioritize which aspects of the input should be copied, it often learns **useful properties** of the data.

Modern autoencoders have generalized the idea of an encoder and a decoder beyond deterministic functions to stochastic mappings $p_{encoder}(\mathbf{h}|\mathbf{x})$ and $p_{decoder}(\mathbf{x}|\mathbf{h})$. we may think of the decoder as providing a conditional distribution $p_{decoder}(\mathbf{x}|\mathbf{h})$. We may then train the autoencoder by minimizing $-log\, p_{decoder}(\mathbf{x}|\mathbf{h})$.

16.2 Undercomplete Autoencoders

One way to obtain useful features from the autoencoder is to constrain $\mathbf{h}$ to have smaller dimension than $\mathbf{x}$. An autoencoder whose code dimension is less than the input dimension is called **undercomplete**. Learning an undercomplete representation forces the autoencoder to capture the most salient features of the training data.

The learning process is described simply as minimizing a loss function;

$$L(\mathbf{x}, g(f(\mathbf{x}))$$

where L is a loss function penalizing $g(f(\mathbf{x}))$ for being dissimilar from $\mathbf{x}$, such as the mean squared error.

When the decoder is linear and L is the mean squared error, an undercomplete autoencoder learns to span the same subspace as **PCA** (SVD). Autoencoders with nonlinear encoder functions f and nonlinear decoder functions g can thus learn a more powerful nonlinear generalization of PCA. However, if the encoder and decoder are allowed too much capacity, the autoencoder can learn to perform the copying task without extracting useful information about the distribution of the data.

Autoencoders are often trained with only a single layer encoder and a single layer decoder. However, this is not a requirement. In fact, using deep encoders and decoders offers many advantages. Depth can exponentially reduce the computational cost of representing some functions. Depth can also exponentially decrease the amount of training data needed to learn some functions. Experimentally, **deep autoencoders** yield much better compression than corresponding shallow or linear autoencoders.

16.2.1 Singular Value Decomposition (SVD)

Singular Value Decomposition (SVD) is a standard linear dimensionality reduction method which combines the features of the original high-dimensional dataset and project them into a lower-dimensional space, ideally retaing most of their intrinsic properties.

Given a matrix X, the SVD decomposes it into the product of two unitary matrices, V and U, and a rectangular diagonal matrix of singular values S:

$$X = V \cdot S \cdot U^T$$

The values in S are called singular values. We can choose to keep only the first k singular values in order to reduce the dimensionality of the input while minimizing the information loss.

16.3 Overcomplete Autoencoders

Overcomplete autoencoders have the hidden code dimension greater than the input. However, autoencoders may fail to learn useful properties of data if the dimension of the code $\mathbf{h}$ is greater or equal to the input dimension. Therefore, rather than limiting the model capacity by keeping the encoder and decoder shallow and the code size small, **regularized autoencoders** use a loss function that encourages the model to have other properties besides the ability to copy its input to its output.

There are different types of regularized autoencoders:

- Sparse autoencoders
- Denoising autoencoders
- Contractive autoencoders
- Autoencoders with Dropout on the hidden layer

16.4 Sparse Autoencoders

A sparse autoencoder limits the capacity of the model by adding a sparsity penalty $\Omega(\mathbf{h})$ on the code layer $\mathbf{h}$, to the cost function:

$$L(\mathbf{x}, g(f(\mathbf{x}))) + \Omega(\mathbf{h})$$

The sparsity penalty makes the model able to perform feature selection. In this way, a sparse autoencoder does not learn just the identity function, but it can learn useful features of the input.

Rather than thinking of the sparsity penalty as a regularizer for the copying task, we can think of the entire sparse autoencoder framework as approximating maximum likelihood training of a generative model that has latent variables (see slides for more details).

16.5 Denoising Autoencoders

Traditionally, autoencoders minimize some function:

$$L(\mathbf{x}, g(f(\mathbf{x})))$$

A **denoising autoencoder** or DAE instead minimizes:

$$L(\mathbf{x}, g(f(\tilde{\mathbf{x}})))$$

where $\tilde{\mathbf{x}}$ is is a copy of $\mathbf{x}$ that has been corrupted by some form of noise. Basically, DAE are forced to reconstruct a corrupted representation of the input. Denoising autoencoders must therefore undo this corruption rather than simply copying their input.

Like many other machine learning algorithms, autoencoders exploit the idea that data concentrates around a low-dimensional **manifold**. Autoencoders aim to learn the structure of the manifold.

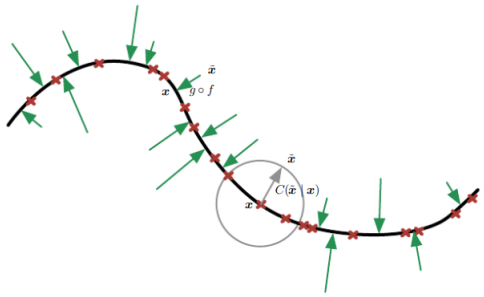


Figure 14.4: A denoising autoencoder is trained to map a corrupted data point $\tilde{\boldsymbol{x}}$ back to the original data point $\boldsymbol{x}$. We illustrate training examples $\boldsymbol{x}$ as red crosses lying near a low-dimensional manifold illustrated with the bold black line. We illustrate the corruption process $C(\tilde{\boldsymbol{x}} \mid \boldsymbol{x})$ with a gray circle of equiprobable corruptions. A gray arrow demonstrates how one training example is transformed into one sample from this corruption process. When the denoising autoencoder is trained to minimize the average of squared errors $||g(f(\tilde{\boldsymbol{x}})) - \boldsymbol{x}||^2$, the reconstruction $g(f(\tilde{\boldsymbol{x}}))$ estimates $\mathbb{E}_{\mathbf{x},\tilde{\mathbf{x}} \sim p_{\text{data}}(\mathbf{x}) C(\tilde{\mathbf{x}} \mid \mathbf{x})}[\mathbf{x} \mid \tilde{\boldsymbol{x}}]$. The vector $g(f(\tilde{\boldsymbol{x}})) - \tilde{\boldsymbol{x}}$ points approximately towards the nearest point on the manifold, since $g(f(\tilde{\boldsymbol{x}}))$ estimates the center of mass of the clean points $\boldsymbol{x}$ which could have given rise to $\tilde{\boldsymbol{x}}$. The autoencoder thus learns a vector field $g(f(\boldsymbol{x})) - \boldsymbol{x}$ indicated by the green arrows. This vector field estimates the score $\nabla_{\boldsymbol{x}} \log p_{\text{data}}(\boldsymbol{x})$ up to a multiplicative factor that is the average root mean square reconstruction error.

16.6 Contractive Autoencoders

The **contractive autoencoder** introduces an explicit regularizer on the code $\mathbf{h} = f(\mathbf{x})$, encouraging the derivatives of f to be as small as possible:

$$\Omega(\mathbf{h}) = \lambda || \frac{\partial f(\mathbf{x})}{\partial \mathbf{x}} ||_F^2$$

The penalty $\Omega(\mathbf{h})$ is the squared Frobenius norm (sum of squared elements) of the Jacobian matrix of partial derivatives associated with the encoder function.

This forces the model to learn a function that does not change much when $\mathbf{x}$ changes slightly.

Chapter 17

Lec 22 - Structured Probabilistic Models

17.1 Joint probability recall

Any joint probability distribution over many random variables may be decomposed into conditional distributions:

$$P(\mathrm{x}^{(1)}, \ldots, \mathrm{x}^{(n)}) = P(\mathrm{x}^{(1)}) \Pi_{i=2}^{n} P(\mathrm{x}^{(i)} \mid \mathrm{x}^{(1)}, \ldots, \mathrm{x}^{(i-1)}).$$

This observation is known as the chain rule or **product rule** of probability.

Two random variables a and b are conditionally independent given a random variable c if the conditional probability distribution over a and b factorizes in this way for every value of c:

$$P(a|b, c) = P(a|c)$$

Equivalently, conditional independence may be stated as:

$$P(a, b|c) = p(a|c)p(b|c)$$

Proof of the equivalent definition [edit]

$P(A, B \mid C) = P(A \mid C)P(B \mid C)$

iff $\frac{P(A, B, C)}{P(C)} = \left(\frac{P(A, C)}{P(C)}\right)\left(\frac{P(B, C)}{P(C)}\right)$ (definition of conditional probability)

iff $P(A, B, C) = \frac{P(A, C)P(B, C)}{P(C)}$ (multiply both sides by $P(C)$)

iff $\frac{P(A, B, C)}{P(B, C)} = \frac{P(A, C)}{P(C)}$ (divide both sides by $P(B, C)$)

iff $P(A \mid B, C) = P(A \mid C)$ (definition of conditional probability) $\therefore$

17.2 Structured Probabilistic Models

Deep learning draws upon many modeling formalisms that researchers can use to guide their design efforts and describe their algorithms. One of these formalisms is the idea of structured probabilistic models.

Machine learning algorithms often involve probability distributions over a very large number of random variables. Often, these probability distributions involve **direct interactions between relatively few variables**. Therefore, Using a single function to describe the entire joint probability distribution can be very inefficient (both computationally and statistically).

Modeling a rich distribution over thousands or millions of random variables is a challenging task, both computationally and statistically. Suppose we only wanted to model binary variables. This is the simplest possible case, and yet already it seems overwhelming. For a small 32×32 pixel color (RGB) image, there are 2^{3072} possible binary images of this form.

In general, if we wish to model a distribution over a random vector $\mathbf{x}$ containing n discrete variables capable of taking on k values each, then the naive approach of representing $P(\mathbf{x})$ by storing a lookup table with one probability value per possible outcome requires k^n parameters!

This tabular approach is Infeasible for several reasons:

- Memory: the cost of storing the representation is too high;
- Runtime: inference and sampling are too slow;
- Statistical efficiency: As the number of parameters in a model increases, so does the amount of training data needed to choose the values of those parameters using a statistical estimator.

Instead of using a single function to represent a probability distribution, we can split a probability distribution into many factors that we multiply together. For example, suppose we have three random variables: a, b and c. Suppose that a influences the value of b and b influences the value of c, but that a and c are independent given b. We can represent the probability distribution over all three variables as a product of probability distributions over two variables:

$$p(a, b, c) = p(a)p(b|a)p(c|b)$$

We can greatly reduce the cost of representing a distribution if we are able to find a factorization into distributions over fewer variables. We can describe these kinds of factorizations using graphs. Because the structure of the model is defined by a graph, these models are often also referred to as **graphical models**.

Tasks for structured probabilistic models:

- Density estimation
- Denoising
- Sample generation
- Missing value imputation
- Sampling

17.3 Graphical Models

Structured probabilistic models provide a formal framework for modeling interactions between random variables. Graphical models use graphs to represent these interactions. Each node represents a random variable. Each edge represents a direct interaction. Paths represent indirect interactions.

Graphical models can be largely divided into two categories: models based on **directed acyclic graphs**, and models based on **undirected graphs**.

17.3.1 Directed Models

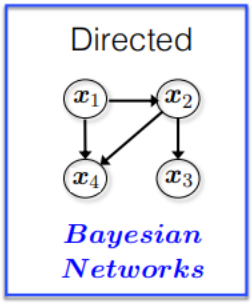
One kind of structured probabilistic model is the directed graphical model, also known as **Bayesian Networks**. Directed graphical models are called "directed" because their edges are directed, that is, they point from one vertex to another. The direction of the arrow indicates which variable's probability distribution is defined in terms of the other's. Basically, drawing an arrow from a to b means that the distribution over b depends on the value of a. Directed models work best when influence clearly flows in one direction (causal relation).

Given a vector of n random variables $\mathbf{x}$, The probability distribution over $\mathbf{x}$ is given by:

$$P(x_1, ..., x_n) = \prod_{i=1}^{n} P(x_i | pa_G(x_i))$$

where $pa_G(x_i)$ represents the parents of x_i in the graph.

For example, the following graph:



Corresponds to the following factorization:

$$P(x_1, x_2, x_3, x_4) = P(x_4|x_1, x_2)P(x_3|x_2)P(x_2|x_1)P(x_1)$$

x_3 is conditional independent from x_1, given x_2, while x_4 is conditional independent from x_3, given x_1, x_2.

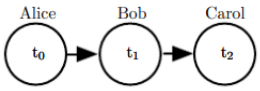


Figure 16.2: A directed graphical model depicting the relay race example. Alice's finishing time t_0 influences Bob's finishing time t_1, because Bob does not get to start running until Alice finishes. Likewise, Carol only gets to start running after Bob finishes, so Bob's finishing time t_1 directly influences Carol's finishing time t_2.

Directed models can drastically reduce the cost of representing a distribution. Let's consider the following example:

Let t_0, t_1, t_2 be discrete variables each with 100 possible values. If we want to represent $P(t_0, t_1, t_2)$ with a table, it would need to store 999,999 values ($100^3 - 1$, since the probability of one of the configurations is made redundant by the constraint that the sum of the probabilities be 1). Now we make the **assumption** that t_2 is conditionally independent from t_0, given t_1:

$$P(t_0, t_1, t_2) = P(t_0)P(t_1|t_0)P(t_2|t_1)$$

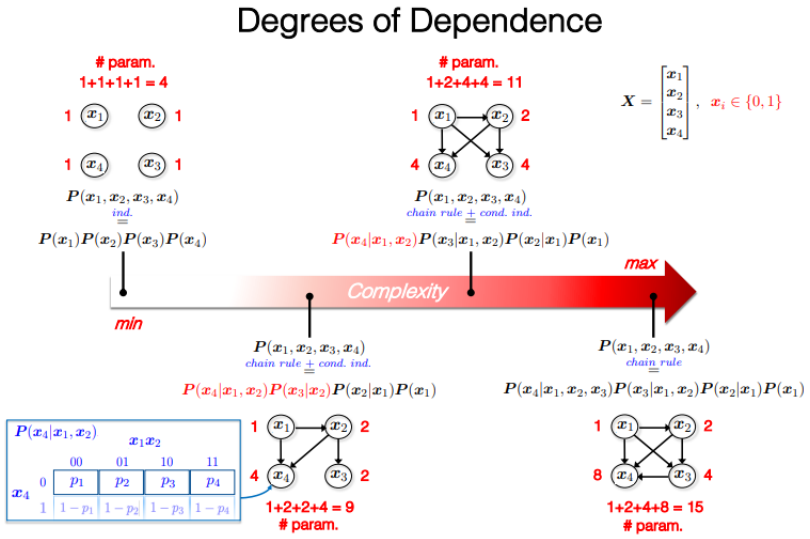
We can only make a table for each of the conditional probability distributions, then the distribution over t_0 requires 99 values, the table defining t_1 given t_0 requires 9900 values, and so does the table defining t_2 given t_1. This comes to a total of 19,899 values. This means that using the directed graphical model reduced our number of parameters by a factor of more than 50!

In general, the number of values required to represent a distribution over a variable x is given by:

$$(k-1) * \prod_{x_p \in pa_G} k_{xp}$$

where k is the number of values that x can take, and k_{xp} is the number of values that the parent x_p of x can take.

Basically, the greater the **degree of dependence** of the graph, the more values are required to represent the distribution.



It is important to realize what kinds of information can and cannot be encoded in the graph. The graph, in this case, encodes only simplifying assumptions about which variables are conditionally independent from each other, but it is also possible to make other kinds of simplifying assumptions.

17.3.2 Undirected Models

Undirected models are used when the influence between variables has no clear direction or is best modeled as flowing in both directions.

As an example of such a situation, suppose we want to model a distribution over three binary variables: whether or not you are sick, whether or not your coworker is sick, and whether or not your roommate is sick. Assuming that your coworker and your roommate do not know each other, it is very unlikely that one of them will give the other an infection such as a cold directly. However, it is reasonably likely that either of them could give you a cold, and that you could pass it on to the other. We can model the indirect transmission of a cold with the following undirected graph:

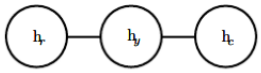


Figure 16.3: An undirected graph representing how your roommate's health h_r, your health h_y, and your work colleague's health h_c affect each other. You and your roommate might infect each other with a cold, and you and your work colleague might do the same, but assuming that your roommate and your colleague do not know each other, they can only infect each other indirectly via you.

In this case, there is not a clean, uni-directional narrative on which to base the model.

Formally, an undirected graphical model is a structured probabilistic model defined on an undirected graph G. For each clique C in the graph, a factor $\phi(C)$ (also called **clique potential**) measures the affinity of the variables in that clique for being in each of their possible joint states. The factors are constrained to be non-negative.

Then, given a vector of random variables $\mathbf{x}$, the **unnormalized probability distribution** over $\mathbf{x}$ is given by:

$$\tilde{P}(\mathbf{x}) = \prod_{C \in G} \phi(C)$$

While the unnormalized probability distribution is guaranteed to be non-negative everywhere, it is not guaranteed to sum or integrate to 1. To obtain a valid probability distribution, we must use the corresponding normalized probability distribution:

$$P(\mathbf{x}) = \frac{1}{Z}\tilde{P}(\mathbf{x})$$

where the **partition function** Z is the value that results in the probability distribution summing or integrating to 1. Since Z is an integral or sum over all possible joint assignments of the state $\mathbf{x}$ it is often intractable to compute. Due to the intractability of computing Z exactly, we must resort to approximations.

Energy-Based Models

A special case of undirected probabilistic models are **Energy-Based Models**. They enforce $\phi(C) > 0$ by defining the unnormalized probability distribution as:

$$\tilde{P}(\mathbf{x}) = exp(-E(\mathbf{x}))$$

$$\tilde{\boldsymbol{P}}(\boldsymbol{X}) = \exp(-E(\boldsymbol{X})) = \exp(-\sum_{C \in G} E_C(\boldsymbol{X}_C)) = \prod_{C \in G} \exp(-E_C(\boldsymbol{X}_C))$$

where $E(\mathbf{x})$ is known as the **energy function** that has to be learned.

Because $exp(z)$ is positive for all z, this guarantees that no energy function will result in a probability of zero for any state $\mathbf{x}$.

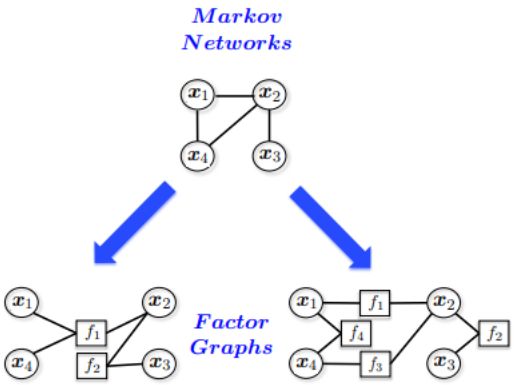
Any distribution of this form is an example of a **Boltzmann distribution**. For this reason, many energy-based models are called **Boltzmann machines**.

Factor Graphs

Factor graphs are another way of drawing undirected models that resolve an ambiguity about graph clique factorization. For example, a clique containing three nodes may correspond to a factor over all three nodes,

or may correspond to three factors that each contain only a pair of the nodes.

Factor graphs resolve this ambiguity by explicitly representing the scope of each ϕ function.



17.4 Sampling

If we are able to represent a probability distribution using graphical models, then we can answer any **probabilistic query**. However, exact inference in graphical models is computational inefficient most of the times. Therefore, we can use sampling for approximate inference. The basic idea is to draw N samples from a sampling distribution S and compute an approximate posterior probability $\hat{\mathbf{P}}$. Then, we expect $\hat{\mathbf{P}}$ and the true probability $\mathbf{P}$ to converge as N increases.

One advantage of *directed* graphical models is that a simple and efficient procedure called **ancestral sampling** can produce a sample from the joint distribution represented by the model.

The basic idea is to sort the variables x_i in the graph into a topological ordering (i.e. from the variables with fewer parents), and sample each node given its parents. In other words, we first sample $x_1 \sim P(x_1)$, the sample $x_2 \sim P(x_2|pa_G(x_2))$, and so on, until we finally sample $x_n \sim P(x_n|pa_G(x_n))$.

So long as each conditional distribution $P(x_i|pa_G(x_i))$ is easy to sample from, then the whole model is easy to sample from. Without the topological sorting, we might attempt to sample a variable before its parents are available.

One drawback of ancestral sampling is that it does not support every conditional sampling operation. When we wish to sample from a subset of the variables in a directed graphical model, given some other variables, we often require that all the conditioning variables come earlier than the variables to be sampled in the ordered graph. Unfortunately, ancestral sampling is applicable only to directed models.

Sampling from an **undirected model** seems to require resolving cyclical dependencies. Every variable interacts with every other variable, so there is no clear beginning point for the sampling process. Unfortunately, drawing samples from an undirected graphical model is an expensive, multi-pass process. The conceptually simplest approach is **Gibbs sampling**.

Chapter 18

Lec 23 - Structured Probabilistic Models II

18.1 Learning about Dependencies

We know that graphs can be used to represent in an efficient way the factorization of joint probability distributions, but how can we find the correct factorization given a specific distribution? There are two main families of approaches:

- Learning Graph Structure
- Use Latent Variables

18.1.1 Learning Graph Structure

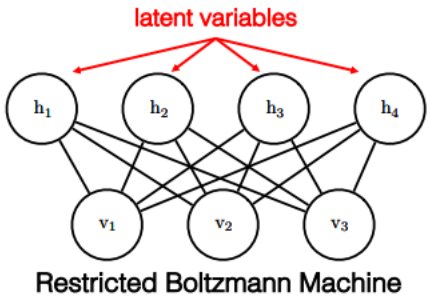
When the model is intended to capture dependencies between visible variables with direct connections, it is usually infeasible to connect all variables, so the graph must be designed to connect those variables that are tightly coupled and omit edges between other variables. An entire field of machine learning called **structure learning** is devoted to this problem.

Most structure learning techniques are a form of greedy search in the space of all possible graphs. A structure is proposed, a model with that structure is trained, then given a score. The score rewards high training set accuracy and penalizes model complexity. Candidate structures with a small number of edges added or removed are then proposed as the next step of the search. The search proceeds to a new structure that is expected to increase the score.

18.1.2 Use Latent Variables

In the context of deep learning, the approach most commonly used to model the dependencies between the observed variables $\mathbf{v}$ is to introduce several latent or "hidden" variables $\mathbf{h}$. The model can then capture dependencies between any pair of variables v_i and v_j indirectly, via direct dependencies between v_i and $\mathbf{h}$, and direct dependencies between $\mathbf{h}$ and v_j. Using latent variables instead of adaptive structure avoids the need to perform discrete searches and multiple rounds of training. Note that in this case we are assuming that the visible variables depend on some latent non-observable variables.

A fixed structure over visible and hidden variables can use direct interactions between visible and hidden units to impose indirect interactions between visible units. Using simple parameter learning techniques we can learn a model with a fixed structure that imputes the right structure on the marginal $P(\mathbf{v})$.



Obviously, we need to make an assumption regarding the number of latent variables we want to fix (just like the number of hidden units in a neural network). Furthermore, we usually set dense connections of latent variables to observed variables.

18.2 Inference and Approximate Inference

With **inference** we mean the task of predicting the value of some variables given other variables, or predicting the probability distribution over some variables given the value of other variables.

For example, in a latent variable model we might want to extract features $E[\mathbf{h}|\mathbf{v}]$ describing the observed variables $\mathbf{v}$.

Unfortunately, for most interesting deep models, these inference problems are intractable, even when we use a structured graphical model to simplify them. The graph structure allows us to represent complicated, high-dimensional distributions with a reasonable number of parameters, but the graphs used for deep learning are usually not restrictive enough to also allow efficient inference. It is straightforward to see that computing the marginal probability of a general graphical model is #P hard.

For this reason, deep learning usually rely on **approximate inference**, in which we approximate the true distribution $p(\mathbf{h}|\mathbf{v})$ by seeking an approximate distribution $q(\mathbf{h}|\mathbf{v})$ that is as close to the true one as possible.

18.3 Restricted Boltzmann Machine

Restricted Boltzmann Machines (RMBs) are undirected probabilistic graphical models containing a layer of observable variables and a single layer of latent variables. They are **unsupervised** models used to learn underlying patterns in the data.

The canonical RBM is an **energy-based** model with binary visible and hidden units. Its energy function is:

$$E(\mathbf{v}, \mathbf{h}) = -\mathbf{b}^T\mathbf{v} - \mathbf{c}^T\mathbf{h} - \mathbf{v}^T\mathbf{W}\mathbf{h}$$

where $\mathbf{b}$, $\mathbf{c}$, and $\mathbf{W}$ are learnable parameters. We can see that the model is divided into two groups of units: $\mathbf{v}$ and $\mathbf{h}$, and the interaction between them is described by a matrix $\mathbf{W}$ (the connections between visible and latent variables are bidirectional). Note that there are no direct interactions between any two visible units or between any two hidden units (hence the "restricted," a general Boltzmann machine may have arbitrary connections).

Based on this, the joint probability distribution represented by the model is given by:

$$P(\mathbf{v} = v, \mathbf{h} = h) = \frac{1}{Z}exp(-E(v, h))$$

Z is known as the partition function:

$$\sum_v \sum_h exp(-E(v,h))$$

Though $P(\mathbf{v})$ is intractable, the bipartite graph structure of the RBM has the very special property that its conditional distributions $P(\mathbf{h}|\mathbf{v})$ and $P(\mathbf{v}|\mathbf{h})$ are factorial and relatively simple to compute and to sample from:

$$\begin{aligned}
P(\boldsymbol{h} \mid \boldsymbol{v}) &= \frac{P(\boldsymbol{h}, \boldsymbol{v})}{P(\boldsymbol{v})} \\
&= \frac{1}{P(\boldsymbol{v})} \frac{1}{Z} \exp\left\{\boldsymbol{b}^\top \boldsymbol{v} + \boldsymbol{c}^\top \boldsymbol{h} + \boldsymbol{v}^\top \boldsymbol{W} \boldsymbol{h}\right\} \\
&= \frac{1}{Z'} \exp\left\{\boldsymbol{c}^\top \boldsymbol{h} + \boldsymbol{v}^\top \boldsymbol{W} \boldsymbol{h}\right\} \\
&= \frac{1}{Z'} \exp\left\{\sum_{j=1}^{n_h} c_j h_j + \sum_{j=1}^{n_h} \boldsymbol{v}^\top \boldsymbol{W}_{:,j} h_j\right\} \\
&= \frac{1}{Z'} \prod_{j=1}^{n_h} \exp\left\{c_j h_j + \boldsymbol{v}^\top \boldsymbol{W}_{:,j} h_j\right\}
\end{aligned}$$

$\boldsymbol{v}$ observed: constant

$\boldsymbol{v}$ observed: constant

h_j separation

From unnormalized to normalized distributions

$$\begin{aligned}
P(h_j = 1 \mid \boldsymbol{v}) &= \frac{\tilde{P}(h_j = 1 \mid \boldsymbol{v})}{\tilde{P}(h_j = 0 \mid \boldsymbol{v}) + \tilde{P}(h_j = 1 \mid \boldsymbol{v})} \\
&= \frac{\exp\left\{c_j + \boldsymbol{v}^\top \boldsymbol{W}_{:,j}\right\}}{\exp\{0\} + \exp\left\{c_j + \boldsymbol{v}^\top \boldsymbol{W}_{:,j}\right\}} \\
&= \sigma\left(c_j + \boldsymbol{v}^\top \boldsymbol{W}_{:,j}\right).
\end{aligned}$$

Recall that: $\sigma(x) = \frac{\exp(x)}{\exp(x) + \exp(0)}$

Finally, for the hidden layer:

Recall that: $1 - \sigma(x) = \sigma(-x)$

$$P(\boldsymbol{h} \mid \boldsymbol{v}) = \prod_{j=1}^{n_h} \sigma\left((2\boldsymbol{h} - 1) \odot (\boldsymbol{c} + \boldsymbol{W}^\top \boldsymbol{v})\right)_j$$

...and with a similar derivation, for the visible layer:

$$P(\boldsymbol{v} \mid \boldsymbol{h}) = \prod_{i=1}^{n_v} \sigma\left((2\boldsymbol{v} - 1) \odot (\boldsymbol{b} + \boldsymbol{W}\boldsymbol{h})\right)_i$$

Obviously, the latent variables' values are not observable from the data, but they are **inferred** by the model during training. Knowing $P(\mathbf{h}|\mathbf{v})$ and $P(\mathbf{v}|\mathbf{h})$ is useful to perform any kind of probabilistic task (e.g. $P(\mathbf{v}|\mathbf{h})$ can be used to generate new samples). The values of $\mathbf{W}$, $\mathbf{c}$ and $\mathbf{b}$ need to be learned during training from data.

Given training data $\mathbf{x}$, we want to maximize the log-likelihood $log\, p(\mathbf{x}; \theta)$. However, its gradient with respect to θ is :

$$\nabla_\theta log\, P(\mathbf{x}; \theta) = \nabla_\theta log\, \tilde{P}(\mathbf{x}; \theta) - \nabla_\theta log\, Z(\theta)$$

where $\tilde{P}(\mathbf{x}; \theta)$ is the unormalized probability distribution defined by the RBM model, that is, $exp(-E(\mathbf{v}, \mathbf{h}))$. Computing this term is not a problem. However, the term $\nabla_\theta log\, Z(\theta)$ is usually intractable.

Under some conditions satisfied by RBMs (as well as most ML models), it can be shown that:

$$\nabla_\theta log\, Z = E_{x \sim P(\mathbf{x})} \nabla_\theta log\, \tilde{P}(\mathbf{x})$$

However, this is not possible to compute exactly since we need to sample $\mathbf{x} \sim P(\mathbf{x})$. Therefore, we must rely on some kind of approximation (Monte Carlo methods)[1].

RBMs are usually trained using gradient ascent and Gibbs sampling to deal with the intractability of the partition function. The basic idea is that the weights are adjusted in order to minimize the difference between the reconstructed data and the original data.

18.4 Monte Carlo Methods

Randomized algorithms fall into two rough categories: Las Vegas algorithms and Monte Carlo algorithms. Las Vegas algorithms always return precisely the correct answer (or report that they failed). These algorithms consume a random amount of resources, usually memory or time. In contrast, **Monte Carlo** algorithms return answers with a random amount of error.

18.4.1 Monte Carlo Sampling

There are many reasons that we may wish to **draw samples** from a probability distribution. Sampling provides a flexible way to approximate many sums and integrals at reduced cost.

When a sum or an integral cannot be computed exactly, it is often possible to approximate it using Monte Carlo sampling. The idea is to view the sum or integral s as if it was an expectation under some distribution p and to approximate the expectation by a corresponding average.

$$s = \sum_{\boldsymbol{x}} p(\boldsymbol{x}) f(\boldsymbol{x}) = E_p[f(\mathbf{x})]$$

$$s = \int p(\boldsymbol{x}) f(\boldsymbol{x}) d\boldsymbol{x} = E_p[f(\mathbf{x})]$$

We can approximate s by drawing n samples $\mathbf{x}^{(1)}, ...\mathbf{x}^{(n)}$ from p and computing the empirical average:

$$\hat{s}_n = \frac{1}{n} \sum_{i=1}^{n} f(x^i)$$

For very large n, the error converges "almost surely" to 0. However, sampling from $p(\mathbf{x})$ is not always possible.

18.4.2 Importance of sampling

An important step in the decomposition of the sum or integral performed by Monte Carlo Sampling techniques is deciding which part of the integral (sum) should play the role of the probability $p(\mathbf{x})$ and which part of the integral (sum) should play the role of the quantity $f(\mathbf{x})$ whose expected value (under that probability distribution) is to be estimated. There is no unique decomposition because $p(\mathbf{x})f(\mathbf{x})$ can always be rewritten as:

$$p(\mathbf{x}) f(\mathbf{x}) = q(\mathbf{x}) \frac{p(\mathbf{x}) f(\mathbf{x})}{q(\mathbf{x})}$$

where we now sample from q and average from $\frac{pf}{q}$. Fortunately, we can easily compute the optimal importance sampling q^* so to reduce variance:

$$q^*(\mathbf{x}) = \frac{p(\mathbf{x}) |f(\mathbf{x})|}{Z}$$

where Z is the normalization constant, chosen so that q^* sums or integrates to 1 as appropriate.

[1] Note that approximating the gradients is not a big problem as we have seen with mini-batch algorithms

18.4.3 Markov Chain

In many cases, we wish to use a Monte Carlo technique but there is no tractable method for drawing exact samples from the distribution $p(\mathbf{x})$ or from a good (low variance) importance sampling distribution $q(\mathbf{x})$. In the context of deep learning, this most often happens when $p(\mathbf{x})$ is represented by an undirected model (such as RBM). In these cases, we introduce a mathematical tool called a Markov chain to approximately sample from $p(\mathbf{x})$. The family of algorithms that use Markov chains to perform Monte Carlo estimates is called Markov chain Monte Carlo methods (MCMC).

The core idea of a Markov chain is to have a state $\mathbf{x}$ that begins as an arbitrary value. Over time, we randomly update $\mathbf{x}$ repeatedly. Eventually $\mathbf{x}$ becomes (very nearly) a fair sample from $p(\mathbf{x})$.

Formally, a Markov chain is defined by a random state $\mathbf{x}$ and a transition distribution $T(\mathbf{x}'|\mathbf{x})$ specifying the probability that a random update will go to state $\mathbf{x}'$ if it starts in state $\mathbf{x}$. Running the Markov chain means repeatedly updating the state $\mathbf{x}$ to a value $\mathbf{x}'$ sampled from $T(\mathbf{x}'|\mathbf{x})$.

Let $q^{(t)}(x)$ the distribution at iteration t from which x is drawn. The goal is for $q^{(t)}(x)$ to converge to $p(x)$.

Let i be an integer that represents a state. Then, we can describe $q(x)$ as a vector $\mathbf{v}$ where $v_i = q(x = i)$. The probability of a single state landing in state x' is given by:

$$q^{(t+1)}(x') = \sum_x q^{(t)}(x)T(x'|x) \tag{18.1}$$

We can represent the effect of the transition operator T using a matrix $\mathbf{A}$. We define $\mathbf{A}$ so that:

$$A_{i,j} = T(x' = i|x = j)$$

$\mathbf{A}$ is called a stochastic matrix, that is, (each of its columns represents a probability distribution.

Then, we can rewrite equation (1) in the following way:

$$\mathbf{v}^{(t)} = \mathbf{A}\mathbf{v}^{(t-1)}$$

It can be shown that the process of applying the Markov Chain update repeatedly eventually converges to a **stationary distribution**, sometimes also called the **equilibrium distribution**:

$$\mathbf{v}' = \mathbf{A}\mathbf{v} = \mathbf{v}$$

If we have chosen T correctly, then the stationary distribution q will be equal to the distribution p we wish to sample from.

18.5 Gibbs sampling

In the context of deep learning, we commonly use Markov chains to draw samples from an energy-based model defining a distribution $p(\mathbf{x})$. In this case, we want the $q(\mathbf{x})$ for the Markov chain to be $p(x)$. To obtain the desired $q(\mathbf{x})$, we must choose an appropriate $T(x'|x)$.

A conceptually simple and effective approach for building a Markov chain that samples from $p(\mathbf{x})$ is to use **Gibbs sampling**, in which sampling from $T(\mathbf{x}'|\mathbf{x})$ is performed by selecting one variable x_i and sampling it from p conditioned on its neighbors in the undirected graph G defining the structure of the energy-based model.

It is also possible to sample several variables at the same time so long as they are conditionally independent given all of their neighbors. All of the hidden units of an RBM may be sampled simultaneously because they are conditionally independent from each other given all of the visible units. The same is valid also for the visible variables. Gibbs sampling approaches that update many variables simultaneously in this way are called **block Gibbs sampling**.

Usually in deep learning we just run the process for n steps, for some n that we think will be big enough, and hope for the best.

Chapter 19

Lec 24 - Generative Models

19.1 Generative Models

Generative models aim at learning a model so that $p_{model}(\mathbf{x})$ is as close as possible to $p_{data}(\mathbf{x})$. There are two main families of approaches:

- **Explicit density estimation**: Models of this type explicit define and solve for $p_{model}(\mathbf{x})$: For example RBMs which approximate $p_{model}(\mathbf{x})$ via Markov Chain (Gibbs sampling), or Variational Autoencoders (VAEs), which approximate the density via a variational approach (ELBO).
- **Implicit density estimation:** Which implies learning models that can sample from $p_{model}(\mathbf{x})$ without explicitly defining it. For example, Generative Adversarial Networks (GANs) use a direct approach based on Game Theory.

Both VAEs and GANs exploit **differentiable generator networks**

19.2 Evidence Lower Bound (ELBO)

Assume we have a probabilistic model consisting of observed variables $\mathbf{v}$ and latent variables $\mathbf{h}$. We would like to compute the log probability of the observed data, $log\, p(\mathbf{v}; \theta)$. However, this term is intractable because we have only a finite number of samples of visible variables. Instead, we can compute a lower bound on $log\, p(\mathbf{v}; \theta)$. This bound is called the **evidence lower bound** (ELBO). Specifically, the evidence lower bound is defined to be:

$$\mathcal{L}(\boldsymbol{v}, \boldsymbol{\theta}, q) = \log p(\boldsymbol{v}; \boldsymbol{\theta}) - D_{\mathrm{KL}}\left(q(\boldsymbol{h} \mid \boldsymbol{v}) \| p(\boldsymbol{h} \mid \boldsymbol{v}; \boldsymbol{\theta})\right)$$

where q is an arbitrary probability distribution over $\mathbf{h}$. The two terms are equal if and only if q is the same distribution as $p(\mathbf{h}|\mathbf{v})$. We can rearrange $\mathcal{L}$ is a more convenient way:

$$\begin{aligned}
\mathcal{L}(\boldsymbol{v}, \boldsymbol{\theta}, q) &= \log p(\boldsymbol{v}; \boldsymbol{\theta}) - D_{\mathrm{KL}}(q(\boldsymbol{h} \mid \boldsymbol{v}) \| p(\boldsymbol{h} \mid \boldsymbol{v}; \boldsymbol{\theta})) \\
&= \log p(\boldsymbol{v}; \boldsymbol{\theta}) - \mathbb{E}_{\mathbf{h} \sim q} \log \frac{q(\boldsymbol{h} \mid \boldsymbol{v})}{p(\boldsymbol{h} \mid \boldsymbol{v})} \\
&= \log p(\boldsymbol{v}; \boldsymbol{\theta}) - \mathbb{E}_{\mathbf{h} \sim q} \log \frac{q(\boldsymbol{h} \mid \boldsymbol{v})}{\frac{p(\boldsymbol{h}, \boldsymbol{v}; \boldsymbol{\theta})}{p(\boldsymbol{v}; \boldsymbol{\theta})}} \\
&= \log p(\boldsymbol{v}; \boldsymbol{\theta}) - \mathbb{E}_{\mathbf{h} \sim q} \left[\log q(\boldsymbol{h} \mid \boldsymbol{v}) - \log p(\boldsymbol{h}, \boldsymbol{v}; \boldsymbol{\theta}) + \log p(\boldsymbol{v}; \boldsymbol{\theta})\right] \\
&= -\mathbb{E}_{\mathbf{h} \sim q} \left[\log q(\boldsymbol{h} \mid \boldsymbol{v}) - \log p(\boldsymbol{h}, \boldsymbol{v}; \boldsymbol{\theta})\right].
\end{aligned}$$

$$\mathcal{L}(\boldsymbol{v}, \boldsymbol{\theta}, q) = \mathbb{E}_{\mathbf{h}\sim q}\left[\log p(\boldsymbol{h}, \boldsymbol{v})\right] + H(q).$$

When $q(\mathbf{h}|\mathbf{v}) = p(\mathbf{h}|\mathbf{v})$, the approximation is perfect. We can thus think of inference as the procedure for finding the q that maximizes $\mathcal{L}$.

19.3 Differentiable Generator Networks

Many generative models are based on the idea of using a **differentiable generator network**. A differentiable generator network transforms samples of latent variables $\mathbf{z}$ to samples $\mathbf{x}$ (direct) or to distributions over samples $\mathbf{x}$ (indirect) using a differentiable function $g(\mathbf{z}; \theta^{(g)})$, which is typically represented by a neural network. Basically, the idea of differentiable generator networks is to define a distribution over latent variables $\mathbf{z}$ (often very simple, like Gaussian distribution), and then learn, via g, how to transform the shape of that distribution into the distribution of our training data.

This model class includes variational autoencoders, which pair the generator net with an inference net, generative adversarial networks, which pair the generator network with a discriminator network, and techniques that train generator networks in isolation.

Generator networks are essentially just parametrized computational procedures for generating samples, where the architecture provides the family of possible distributions to sample from and the parameters select a distribution from within that family.

Example of direct sample generation: As an example, the standard procedure for drawing samples from a normal distribution with mean μ and covariance Σ is to feed samples $\mathbf{z}$ from a normal distribution with zero mean and identity covariance into a very simple generator network. This generator network contains just one affine layer:

$$\mathbf{x} = g(\mathbf{z}) = \mu + \mathbf{L}\mathbf{z}$$

where $\mathbf{L}$ is given by the Cholesky decomposition of Σ.

Example of indirect sample generation (define distribution over samples): Use $g(\cdot)$ with sigmoid outputs to provide the mean parameters of Bernoulli distributions:

$$p(x_i = 1|\mathbf{z}) = g(\mathbf{z})_i$$

we impose a distribution over $\mathbf{x}$ by marginalizing $\mathbf{z}$:

$$p(\mathbf{x}) = \mathbb{E}_{\mathbf{z}} p(\mathbf{x}|\mathbf{z})$$

As the name suggests, a Differentiable Generator Networks must be differentiable in order to perform gradient-based optimization. However, the sampling from the distribution defined over latent variables $\mathbf{z}$

$$z \sim \mathcal{N}(\mu, \sigma^2)$$

is not performed by a function, but it is a stochastic process which changes every time we query it. Therefore, it does not make sense to compute the gradient of z with respect to the parameters of its distribution, μ and σ^2. So, we can rewrite the sampling process as transforming an underlying random value $\epsilon \sim \mathcal{N}(\epsilon, 0, 1)$ to obtain a sample from the desired distribution:

$$z = \mu + \sigma\epsilon$$

We are now able to back-propagate through the sampling operation, by regarding it as a deterministic operation with an extra input ϵ. This technique is known as **reparametrization trick** (this will be more clear when talking about Variational Autoencoders, since in that case μ and σ are parameters that have to be learned).

19.4 Variational Autoencoders (VAEs)

The variational autoencoder is a **directed model** that uses learned approximate inference and can be trained purely with gradient-based methods.

VAEs assume that our training data $\mathcal{T} = \{\mathbf{x}^{(i)}\}_{i=1}^{n}$ is generated by some latent variables $\mathbf{z}$. Let $p_\theta(\mathbf{z})$ the prior distribution over $\mathbf{z}$. The basic idea of VAEs is to set a simple distribution for $p_\theta(\mathbf{z})$, e.g. Gaussian, and sample $\mathbf{z}$ from it. Then, we can use a generator network $g(\mathbf{z})$ to estimate $p_\theta(\mathbf{x}|\mathbf{z})$ and generate new samples. The goal is to estimate the parameters θ of this generative model, that is, how to map $\mathbf{x}$ to $\mathbf{z}$ and $\mathbf{z}$ to $\mathbf{x}$. However, both the terms $p_\theta(\mathbf{x})$ and $p_\theta(\mathbf{z}|\mathbf{x})$ are intractable, so we cannot perform maximum likelihood estimation.

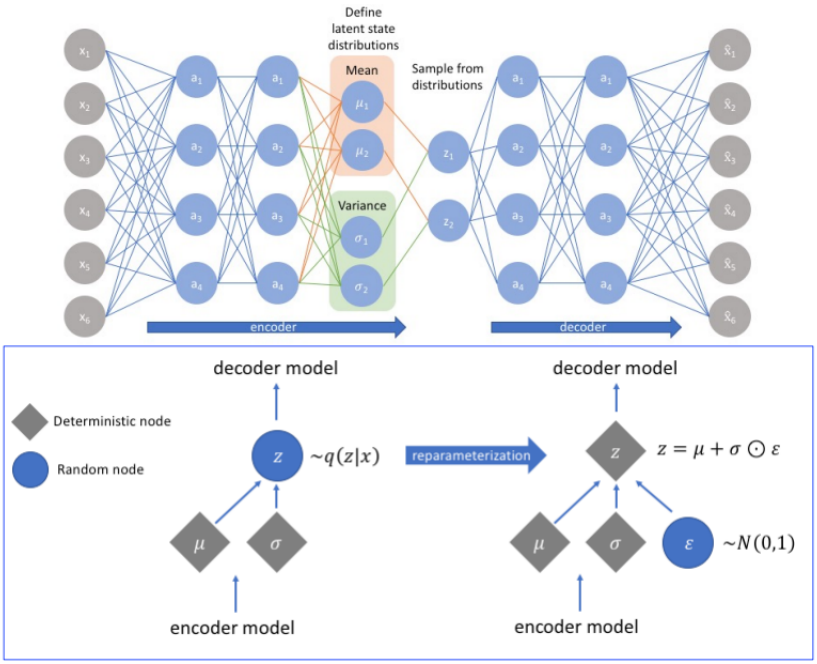
In order to overcome this problem, we can define in addition to the decoder modeling $p_\theta(\mathbf{x}|\mathbf{z})$ an encoder $q_\phi(\mathbf{z}|\mathbf{x})$ that approximates $p_\theta(\mathbf{z}|\mathbf{x})$ using ELBO.

This lower bound $\mathcal{L}(q)$ is defined as follows:

$$
\begin{aligned}
\mathcal{L}(q) &= \mathbb{E}_{\boldsymbol{z}\sim q(\boldsymbol{z}|\boldsymbol{x})} \log p_{\text{model}}(\boldsymbol{z}, \boldsymbol{x}) + \mathcal{H}(q(\mathbf{z} \mid \boldsymbol{x})) \\
&= \mathbb{E}_{\boldsymbol{z}\sim q(\boldsymbol{z}|\boldsymbol{x})} \log p_{\text{model}}(\boldsymbol{x} \mid \boldsymbol{z}) - D_{\text{KL}}(q(\mathbf{z} \mid \boldsymbol{x}) || p_{\text{model}}(\mathbf{z})) \\
&\leq \log p_{\text{model}}(\boldsymbol{x}).
\end{aligned}
$$

The first term $log\, p_{model}(\mathbf{x}|\mathbf{z})$ maximizes the likelihood of the original input being reconstructed. The second term $D_{KL}(q(\mathbf{z}|\mathbf{x})||p_{model}(\mathbf{z}))$ forces the encoder $q_\phi(\mathbf{z}|\mathbf{x})$ to become a Gaussian prior on $\mathbf{z}$ (if we set $p_\theta(\mathbf{z})$ as a Gaussian). The learning process aims at finding the parameters θ^*, ϕ^* for the decoder and encoder that maximize the lower bound.

In other words, the encoder network generates, for each input, two vectors μ, σ representing the mean and the variance of the Gaussian distribution we defined over $\mathbf{z}$ (remember that we assumed that the latent variables were distributed according to a Gaussian distribution of 0 mean and variance 1). Then, the decoder network samples from the distribution defined by μ and σ, using the **reparametrization trick**, and performs the reconstruction (remember that we assumed that our data are generated by the latent variables).



The reparametrization trick, as we said before, allows the model to learn the parameter related to the mean and variance vectors while maintaining the stochasticity of the entire system via epsilon.

To generate a sample from the model, the VAE first draws a sample z from the distribution we defined over the latent variables $\mathbf{z}$ (Gaussian with mean 0 and variance 1). The sample is then run through a differentiable generator network $g(\mathbf{z})$, that is, the decoder. Finally, $\mathbf{x}$ is sampled from a distribution $p_\theta(\mathbf{x}|\mathbf{z})$. We usually choose a Gaussian distribution for $p_\theta(\mathbf{z})$ because is easy to sample from.

The VAE framework is very straightforward to extend to a wide range of model architectures. One particularly sophisticated VAE is the **deep recurrent attention writer** or DRAW model. DRAW uses a recurrent encoder and recurrent decoder combined with an attention mechanism. The generation process for the DRAW model consists of sequentially visiting different small image patches and drawing the values of the pixels at those points.

19.5 Generative Adversarial Networks (GANs)

Generative adversarial networks or GANs are another generative modeling approach based on differentiable generator networks.

Generative adversarial networks are based on a game theoretic scenario in which the generator network must compete against an adversary. The generator network directly produces samples $\mathbf{x} = g(\mathbf{z}; \theta^{(g)})$. Its adversary, the **discriminator network**, attempts to distinguish between samples drawn from the training data and samples drawn from the generator. The discriminator emits a probability value given by $d(\mathbf{x}; \theta^{(d)})$, indicating the probability that $\mathbf{x}$ is a real training example rather than a fake sample drawn from the model.

This drives the discriminator to attempt to learn to correctly classify samples as real or fake. Simultaneously, the generator attempts to fool the classifier into believing its samples are real. At convergence, the generator's samples are indistinguishable from real data, and the discriminator outputs $\frac{1}{2}$ everywhere. The discriminator may then be discarded.

During training, the model samples from a simple distribution, e.g. random noise, and learn a transformation (generator network) to the training distribution. The transformation should be complex enough (e.g. deep neural network).

The objective function is defined as a **minmax objective function**:

$$\min_{\theta_g} \max_{\theta_d} \left[\mathbb{E}_{x \sim p_{data}} \log \underbrace{D_{\theta_d}(x)}_{\substack{\text{Discriminator output} \\ \text{for real data x}}} + \mathbb{E}_{z \sim p(z)} \log(1 - \underbrace{D_{\theta_d}(G_{\theta_g}(z))}_{\substack{\text{Discriminator output for} \\ \text{generated fake data G(z)}}}) \right]$$

The intuition behind this is that the discriminator network tries to maximize the *accuracy* in determining whether an example is real or fake, while the generator network tries to minimize the performance of the discriminator. If the generator fails to fool the discriminator, it will be punished and it will adapt the generation of new samples accordingly.

In particular, the first term of the function is maximized when the discriminator successfully detects a real image, while the second term is maximized if the generator fails to fool the discriminator (remember that the discriminator outputs a value between 0 and 1 determining the likelihood of a real image).

The training procedure alternates a gradient ascent step on the discriminator, and a gradient descent step on the generator.

Algorithm 1 Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator, k, is a hyperparameter. We used $k = 1$, the least expensive option, in our experiments.

for number of training iterations **do**
  for k steps **do**
    • Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
    • Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
    • Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right].$$

  end for
  • Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
  • Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right).$$

end for
The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

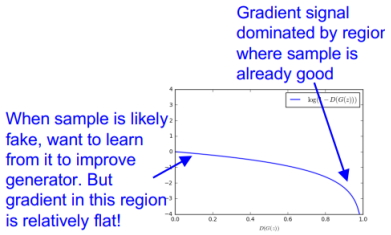
It can be shown that, under some conditions, eventually the generator converges to the distribution of training data.

The shape of the generator's cost function:

$$log\left(1 - D_{\theta_d}(G_{\theta_g}(z))\right)$$

shows that the generator aims at decreasing the log probability that the discriminator makes the correct prediction. However, we can see that when the generator's samples are likely fake the gradient is small in

module, while when the samples fool the discriminator the gradient is large. This is not good for learning, since we want to improve the model when it does not fool the discriminator, and not viceversa.



Therefore, in order to overcome this problem, we can reformulate the objective function such that the generator aims to increase the log probability that the discriminator makes a mistake, rather than aiming to decrease the log probability that the discriminator makes the correct prediction.

19.6 VAEs vs GANs

The main motivation for the design of GANs is that the learning process requires neither approximate inference nor approximation of a partition function gradient. Furthermore, they generate higher quality samples compared to VAEs. However, they can be tricky and unstable to train.

VAEs, instead, provide useful latent representation that allows for inference queries. However, samples from variational autoencoders trained on images tend to be somewhat blurry.