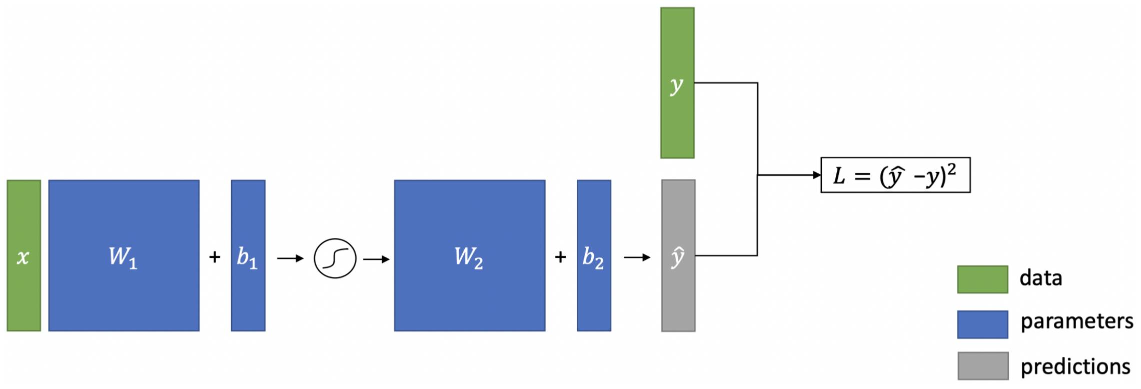
IntroML - Lecture Notes Week 8

Ruben Schenk, ruben.schenk@inf.ethz.ch

July 17, 2022

1 PyTorch Tutorial

Autodiff is an automatic implementation in PyTorch which calculates the gradients automatically given the *forward pass* (i.e. the chain of operations from input to output).



Given this forward pass, PyTorch automatically computes the derivatives (*backward pass*) for us!

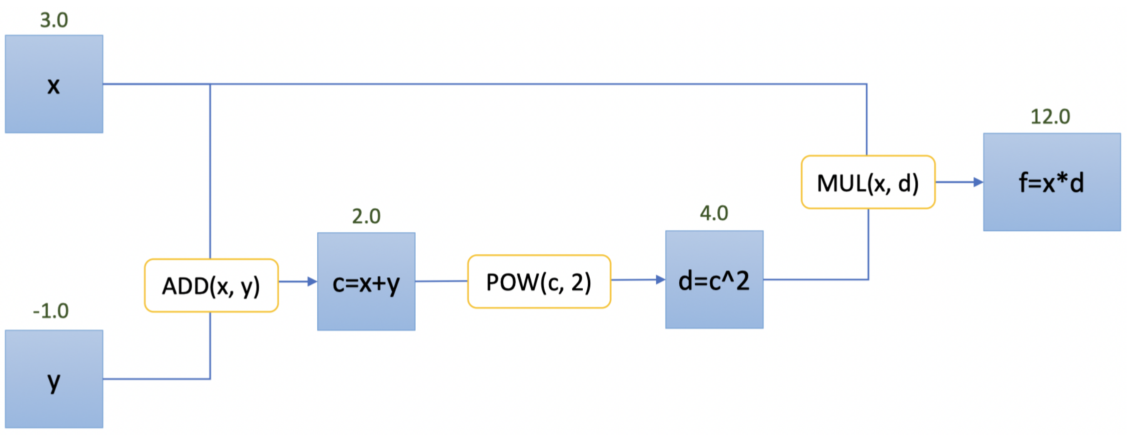
How does Autodiff work? First, as already mentioned, we need to provide the forward pass. Given that, PyTorch constructs the computation graph (DAG) and stores the intermediate computed values at each node of the graph.

Example: Consider the following simple function:

```
def f(x, y):
    c = x + y
    d = c**2
    return x * d

x = torch.tensor([3.0], requires_grad=True)
y = torch.tensor([-1.0], requires_grad=True)
f_val = f(x, y)
```

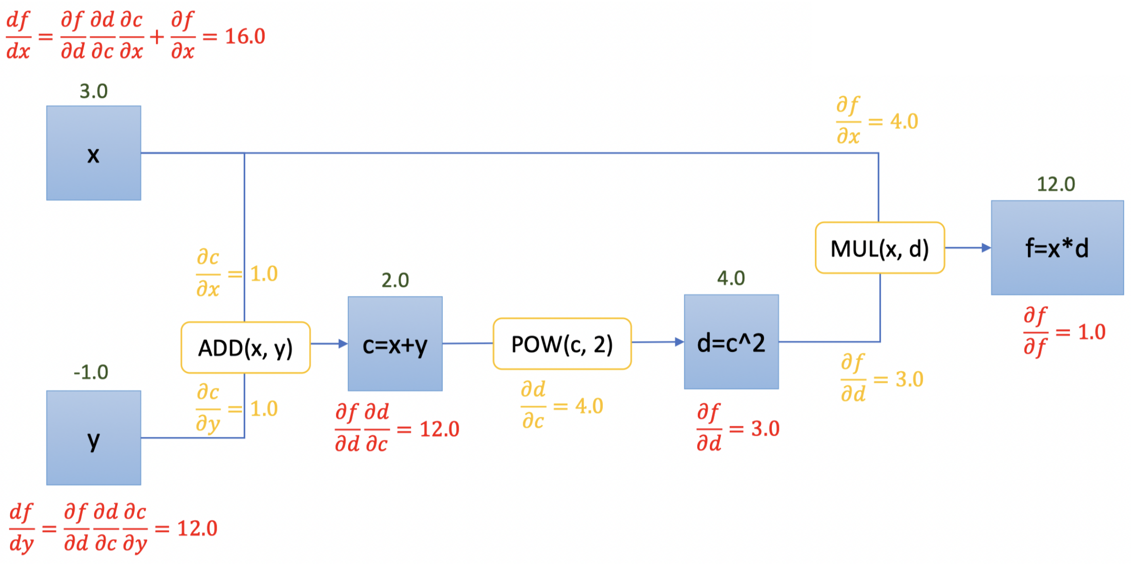
For this function, we get the following forward pass:



It is important to note here that every intermediate result is itself a tensor. Even if we override a variable, the previous tensor is kept alive.

Finally, Autodiff calculates the backward pass the following way:

- Go back from the root (i.e. the loss) to the leaves (i.e. the parameters)
- Each elementary operation has a "backward function" that computes its derivatives
- We apply the chain rule along each path
- Total derivative is the sum of all path derivatives from the root to the leaf


$\frac{df}{dx} = \frac{\partial f}{\partial d}\frac{\partial d}{\partial c}\frac{\partial c}{\partial x} + \frac{\partial f}{\partial x} = 16.0$
3.0
x
$\frac{\partial f}{\partial x} = 4.0$
12.0
f=x*d
MUL(x, d)
$\frac{\partial c}{\partial x} = 1.0$
2.0
4.0
$\frac{\partial f}{\partial f} = 1.0$
ADD(x, y)
c=x+y
POW(c, 2)
d=c^2
-1.0
$\frac{\partial f}{\partial d} = 3.0$
$\frac{\partial c}{\partial y} = 1.0$
$\frac{\partial d}{\partial c} = 4.0$
y
$\frac{\partial f}{\partial d}\frac{\partial d}{\partial c} = 12.0$
$\frac{\partial f}{\partial d} = 3.0$
$\frac{df}{dy} = \frac{\partial f}{\partial d}\frac{\partial d}{\partial c}\frac{\partial c}{\partial y} = 12.0$

2 Unsupervised Learning: Clustering

2.1 Introduction

So far we have concerned ourselfes with *supervised learning*. The idea behind that is that we have some dataset X and their labels Y from which we let the model learn the link between the data and the labels, such that it can label unlabelled data.

The idea behind **unsupervised learning** is that we only provide the dataset X to the model (without the labels) and let the model learn the labels itself, based on some goals (e.g. grouping the data by similarities, understanding meaningful features, etc.).

What is **clustering?** The key idea behind clustering is, given some data points, to group them into clsuters such that:

- *Similar* points are in the same clsuter
- *Dissimilar* points are in different clusters

Points are typically represented either in (high-dimensional) Euclidean space or with distances specified by a metric or (dis-)similarity function. Realted to this approach is *anomal/outlier detection,* i.e. the identification of points that "don't fit well in any of the clusters".

There are some standard approaches to clustering:

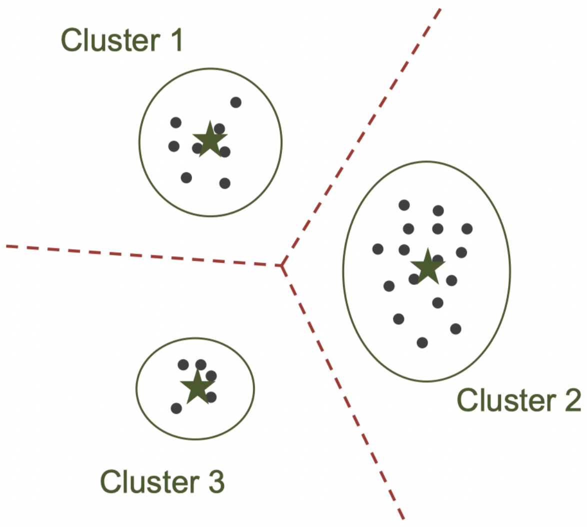
- *Hierarchical clustering:* Build a tree (bottom-up or top-down), representing distances among data points. Examples: single-, average-linkage clustering
- *Partitioning approaches:* Define and optimize a notion of "cost" defined over partitions. Examples: Spectral clustering, graph-cut based approaches
- *Model-based approaches:* Maintain cluster "models" and infer cluster memebership (e.g. assign each point to closest center). Examples: k-means clustering, Gaussian mixture models, etc.

2.2 k-Means Clustering

The idea behind **k-means clustering** is as follows:

1. Represent each cluster by a single point (center)
2. Assing points to the closest center $\mu_i \in \mathbb{R}^d$
3. Induces *Voroni partition*

Our data is given as $D = \{x_1, ..., x_n\}$, $x_j \in \mathbb{R}^d$ and each x_j is assigned a cluster $z_j = \arg\min_i ||x_j - \mu_i||_2$, $z_j \in 1, ..., k$.



To reformulate, we may pose the k-means problem in the the following way:

- Assume points are in Euclidean space $x_i \in \mathbb{R}^d$
- represent clusters as centers $\mu_j \in \mathbb{R}^d$
- Each point is assigned to the closest center

The goal is to pick centers to *minimize the sum of squared distances,* i.e.

$$\text{minimize } \hat{R}(\mu) := \hat{R}(\mu_1, ..., \mu_k) = \sum_{i=1}^{n} \min_{j \in \{1,...,k\}} ||x_i - \mu_j||_2^2.$$

In general this is a non-convex optimization and is NP-hard to solve!

Algorithm (Lloyd's Heuristic): The following represents on of the k-means algorithms:

```
1. Initialize cluster centers $\mu^{(0)} = [\mu_1^{(0)}, ..., \mu_k^{(0)}]$
2. While not converged:
    (a) Assign each point $x_i$ to the closest center
            $z_i \leftarrow \arg \min_{j \in \{1, ..., k\}} ||x_i - \mu_j^{(t-1)}||_2$
    (b) Update the center as the mean of assigned data points
            $\mu_j^{(t)} \leftarrow \frac{1}{n_j} \sum_{i:z_i=j} x_i$
```

K-means is *guaranteed to monotonically decrease* the average squared distance in each iteration, i.e. $\hat{R}(\mu^t) \geq \hat{R}(\mu^{t+1})$. In other words:

- Given μ^t, we obtain $z^z = \arg\min_z \hat{R}(\mu^t, z)$
- Given z^t, we obtain $\mu^t = \arg\min_\mu \hat{R}(\mu, z^t)$

From this, we have:

$$\hat{R}(\mu^t, z^t) \geq \hat{R}(\mu^{t+1}, z^t) \geq \hat{R}(\mu^{t+1}, z^{t+1}).$$

This converges to a *local optimum* and the cost per iteration is $O(n \cdot k \cdot d)$.

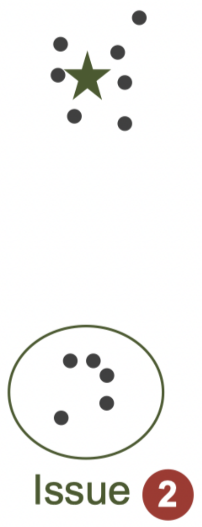
However, k-means also poses some challenges:

- Generally only converges to local optimum, the performance strongly depends on the initialization.
- Number of iterations required can be exponential. In practice however, the algorithm often converges quickly.
- Determining the number of clusters k can be difficult.
- One cannot well model clusters of arbitrary shape.

2.3 k-Means++ Clustering

Lloyd's heuristic does not generally converge to the optimal solution since the performance depends heavily on the initialization.

One option for seeding would be *random seeding,* though one can directly see some problems with this approach:



This is where the **k-means++** algorithm comes into play. It proposes adaptive seeding:

Algorithm: The k-means++ algorithm works in the following way:

1. Startwith a random point as the center

$$\text{Prior } \mu_1^0 = x_1 \text{ where } i \sim Unif(1, ..., n).$$

2. Add centers 2...k randomly, proportionally to the squared distance to the closest selected center

$$\text{Given } \mu_{1:j}^0 \text{ pick } \mu_{1:j+1}^0 = x_i \text{ where } p(i) = \frac{1}{z} \min_{l \in \{1,...,j\}} ||x_i - \mu_l^0||_2^2$$

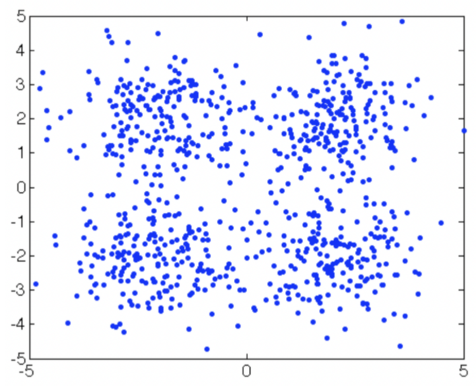
3. We can show the the expected cost is $O(\log k)$ times that of the optimal k-means solution

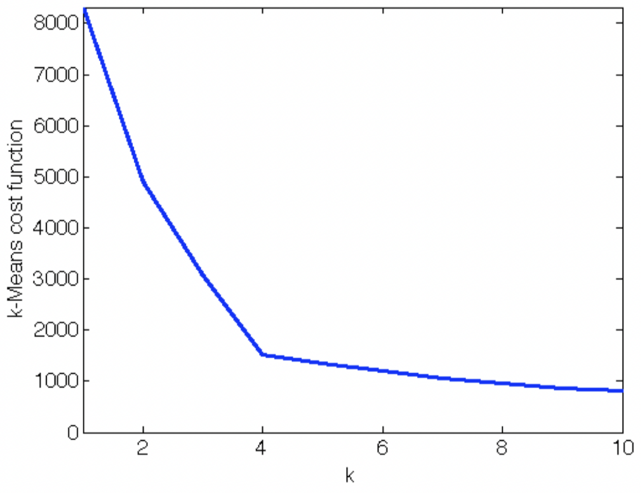
$$\text{Given } \hat{R}(\mu_{\text{k-means++}}) \leq O(\log k) \min_{\mu} \hat{R}(\mu)$$

2.4 Model Selection

In general, **model selection** (e.g. determining the number of clsuters) is difficult. There are several different approaches:

- Heuristic quality measures
- Regularization, i.e. favoring simple models with few parameters by penalizing complex models
- Information theoretic basis (tradeoff between robustness (stability) and informativeness)

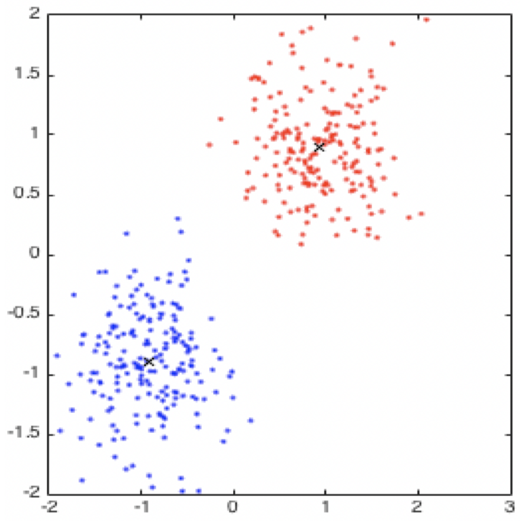


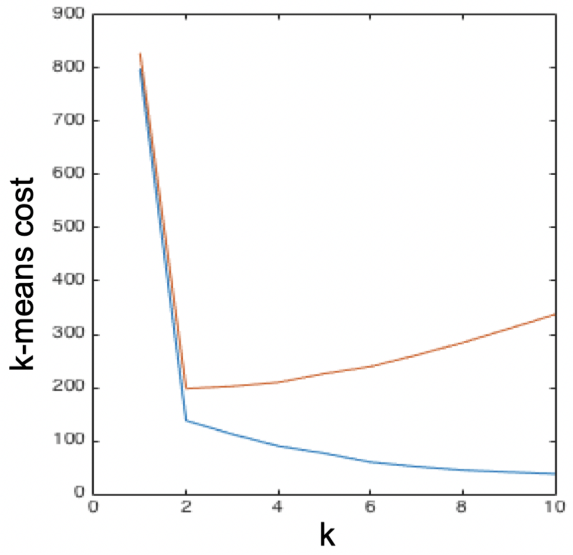


On **heuristic for determining** k might be to find the k that decreases our loss function. We then pick k such that increasing k leads to negligible decrease in loss.

Another idea is to use **regularization** and penalize models with higher k, i.e.:

$$\min_{k,\,\mu_{1:k}} \hat{R}(\mu_{1:k}) + \lambda \cdot k.$$





This approach poses another problem since we now have to choose a "correct" λ, however, this is usually easier than choosing the "correct" k.

2.5 Challenges with k-Means

To summarize, we again list some of the challenges encountered when working with k-means:

1. Generally only converges to local optimum
2. Number of iterations required can be exponential
3. Cannot well model clusters of arbitrary shape
4. Determining the number of clusters k is difficult

Problem 1 and 2 are practically not a big issue. Problem 3 can be fixed via kernel-k-means which we'll introduce later. Problem 4 is unsolved and also a practical problem.