Introduction to Machine Learning

placeholder

1 Regression

1.1 Supervised Learning

Supervised Learning is the task of 'learning' a function relationship, based on a given set of inputs/outputs.

Some terminology:

$x \in \mathbb{R}^d$	Inputs (Attributes/Covariates)
$\phi(x) \in \mathbb{R}^p$	Features
$y \in \mathbb{R}$	Outputs (Targets/Labels)
$D = \{(x_i, y_i)\}_{i=1}^n$	Training Set
D'	Test Set
$f : \mathbb{R}^p \to \mathbb{R}$	Predictor (Model)
$l(f(x), y)$	Loss

Machine Learning Pipelines can often be classified using:

F	Function Class
$L(f)$	Training Loss
	Optimization Method

The function class F is a set of parametrized functions. We are looking for the $f \in F$ that minimizes $L(f)$.

D. **Training Loss**

$$L(f) := \frac{1}{n} \sum_{i=1}^{n} l\big(f(x_i), y\big)$$

1.2 Multiple Linear Regression

Multiple Linear Regression directly uses the $x \in \mathbb{R}^d$.
Here, $F_{\text{affine}} = \big\{f(x) = w^\top x + w_0 \big| w \in \mathbb{R}^d, w_0 \in \mathbb{R}\big\}$.

Rmk. Why are we using linear functions instead?
Any estimator $f \in F_{\text{affine}}$ can be rewritten as $f\big((x,1)\big) = (w, w_0)^\top \cdot (x, 1)$, thus we can augment the inpurs $x \mapsto (x, 1)$ and instead search in $F_{\text{linear}} = \{f(x) = \hat{w}^\top x | \hat{w} \in \mathbb{R}^{d+1}\}$

Loss Functions

D. **Squared Loss** $\quad l\big(f(x), y\big) := \big(f(x) - y\big)^2$
Most common Loss Function, but sensitive to outliers.

D. **Absolute Loss** $\quad l_{\text{abs}}\big(f(x), y\big) := |f(x) - y|$
Less sensitive to outliers, but not differentiable.

D. **Huber Loss**

$$l_{\text{huber}}\big(f(x), y\big) := \begin{cases} \frac{1}{2}\big(f(x) - y\big)^2 & |f(x) - y| \le \delta \\ \delta\big(|f(x) - y| - \frac{1}{2}\delta\big) & |f(x) - y| > \delta \end{cases}$$

Using parameter δ, the penalization of outliers can be controlled

Assymetric Loss: In some cases it is desirable to penalize overestimation harder than underestimation, or vice versa.

D. **Quantile Loss**

$$l_\tau\big(f(x), y\big) := \tau \max\Big\{y - f(x), 0\Big\} + (1-\tau) \max\Big\{f(x) - y, 0\Big\}$$

Using parameter τ, over/underestimation can be penalized

Linear Regression

To find $\hat{f} := \underset{f \in F_{\text{linear}}}{\arg\min} L(f)$ we just look for $w \in \mathbb{R}^d$.

$$\hat{w} := \underset{w \in \mathbb{R}^d}{\arg\min} L(f_w) = \frac{1}{n} \sum_{i=1}^{n} \underbrace{\Big(y_i - w^\top x_i\Big)^2}_{l\left(f(x_i), y_i\right)}$$

A natural abuse of notation here is $L(w) := L(f_w)$.

This can be rewritten in matrix notation:

$$\sum_{i=1}^{n} \Big(y_i - w^\top x_i\Big)^2 = \big\|y - Xw\big\|^2$$

The factor $\frac{1}{n}$ is irrelevant for Optimization, it doesn't depend on w

So we find the usual problem:

$$\hat{w} = \underset{w \in \mathbb{R}^d}{\arg\min} \big\|y - Xw\big\|^2$$

The solution is a stationary point, so:

$$\nabla_w \big\|y - Xw\big\|^2 = 2X^\top (X\hat{w} - y) = 0$$

Which yields the **Normal Equation** from linear algebra.

$$X^\top X \hat{w} = X^\top y$$

2 Classification

In regression, we search an $\hat{f} : \mathbb{R}^d \to \mathbb{R}$, i.e. $y, \hat{y} \in \mathbb{R}$.
In classification, we want $\hat{y} \in \mathcal{Y} \subset \mathbb{R}$, s.t. $\mathcal{Y}$ is discrete.

2.1 Binary Classification

We generally use $\mathcal{Y} = \{+1, -1\}$ and set $\hat{y} = \text{sgn}\big(\hat{f}(x)\big)$.
So, a linear classifier where $\hat{f}(x) = w^\top x$ takes the form:

$$x \mapsto \begin{cases} 1 & w^\top x > 0 \\ -1 & w^\top x < 0 \end{cases}$$

D. **Decision Boundary** $\left\{ x \in \mathbb{R}^d \;\middle|\; \hat{f}(x) = 0 \right\}$

Like in regression, using features is again possible.

2.2 Surrogate Loss

We'd like to reuse the loss minimization from regression.
A natural metric for accuracy is simply checking if $\hat{y} = y$.

D. **Zero-One Loss**

$$l_{0-1}(\hat{y}, y) := \mathbb{I}_{\hat{y} \neq y} = \begin{cases} 1 & \hat{y} \neq y \\ 0 & \hat{y} = y \end{cases}$$

We could try minimizing this:

$$\sum_{(x,y)\in\mathcal{D}} l_{0-1}\big(\hat{y}, y\big) = \sum_{(x,y)\in\mathcal{D}} \mathbb{I}_{f_w(x)\cdot y < 0}$$

Unfortunately, l_{0-1} is non-continuous and non-convex.
We introduce *surrogate loss* to still apply GD.

Note how $\mathbb{I}_{\hat{y}\neq y} = \mathbb{I}_{\hat{y}\cdot y<0}$, so l_{0-1} only depends on $z := \hat{y} \cdot y$.
We thus define losses over z, that are cont. and convex.

D. **Surrogate Loss**

$$l_{\text{exp}} = e^{-z} \qquad l_{\log} = \log(1 + e^{-z})$$

A notable difference is that l'_{exp} is unbounded,
while $l'_{\log} = \frac{1}{1+e^z} \in (-\frac{1}{2}, -1)$ for $z < 0$.
This is better for outliers, thus $l_{\log}$ is usually preferred.

2.3 Logistic Regression

We assume $w_0 = 0$

We try to minimize $l_{\log} = \log(1 + e^{-z})$, so:

$$L(w) = \frac{1}{n}\sum_{i=1}^{n} l_{\log}(z_i) = \frac{1}{n}\sum_{i=1}^{n} \log\Big(1 + e^{-\overbrace{y_i \cdot w^\top x_i}^{z_i}}\Big)$$

Assume $\{x_i, y_i\}_{i=1}^n$ is linearly seperable, i.e.

$$\exists w \in \mathbb{R}^d : \quad \underbrace{y_i \cdot w^\top x_i}_{z_i} > 0 \quad \forall i \leq n$$

Then there are multiple valid decision boundaries. Additionally, $L(w)$ is then convex.

the distance x_0 to the decision boundary is: $\|x_0\|_2 \cdot |\cos(\theta)|$.
θ between $w, x_0 \in \mathbb{R}^d$

$$\|x_0\|_2 \cdot |\cos(\theta)| = \|x_0\|_2 \cdot \frac{|w^\top x_0|}{\|w\|_2 \cdot \|x_0\|_2} = \frac{|w^\top x_0|}{\|w\|_2}$$

Note if w is a unit-vector, this is just $|w^\top x_0|$

D. **Margin** $\text{margin}(w) := \min\limits_{1\leq i\leq n} y_i \cdot w^\top x_i$

2.4 Solutions

D. **Maximum Margin Solution**

$$w_{\text{MM}} := \max_{\|w\|_2=1} \min_{1\leq i\leq n} \Big(y_i \cdot w^\top x_i\Big)$$

If $\mathcal{D}$ is linearly seperable, this is convex.

Rmk. Also called *hard-margin SVM*, assuming lin.-sep. data

D. **Support Vector Machine**

$$w_{\text{SVM}} := \min_{w\in\mathbb{R}^d} \|w\|_2 \quad \text{s.t.} \quad y_i \cdot w^\top x_i \geq 1 \quad \forall i \leq n$$

Solving these problems is actually equivalent, up to scaling:

L. $\dfrac{w_{\text{SVM}}}{\|w_{\text{SVM}}\|_2} = w_{\text{MM}}$ (This also holds for the case $w_0 \neq 0$)

By relaxing the SVM constraints, we can use the SVM problem on lin.-insep. data too:

$$y_i \cdot w^\top x_i \leq 1 \qquad \to \qquad y_i \cdot w^\top x_i \leq 1 - \zeta$$

$\zeta = (\zeta_1, \ldots, \zeta_n)$ s.t. $\zeta_i \geq 0$

D. **Soft-margin SVM**

$$w_{\text{SM}} = \min_{w\in\mathbb{R}^d, \zeta\in\mathbb{R}^n} \left(\|w\|^2 + \lambda \sum_{i=1}^{n} \zeta_i \right) \text{ s.t. } \begin{cases} y_i \cdot w^\top x_i \geq 1 - \zeta_i \\ \zeta_i \geq 0 \quad \forall i \leq n \end{cases}$$

λ (hyperparam.) intuitively controls how much a violation is penalized

Another perspective: The optimal ζ_i are:

$$\zeta_i = \begin{cases} 1 - y_i \cdot w^\top x_i & \text{if } y_i \cdot w^\top x_i \leq 1 \\ 0 & \text{else} \end{cases}$$

So the problem can be formulated without ζ too:

D. **l_2-penalized Hinge Loss Optimization**

$$\min_{w\in\mathbb{R}^d} \left(\|w\|^2 + \lambda \underbrace{\sum_{i=1}^{n} \max(0, 1 - y_i \cdot w^\top x_i)}_{\text{Hinge Loss}} \right)$$

2.5 Gradient Descent for Classification

In practice, instead of explicitly solving w_{SVM} or w_{MM}, GD is usually applied on a diff.-able convex surrogate loss.

2.5.1 On linearly seperable data

Assuming $\{x_i, y_i\}_{i=1}^n$ is lin. seperable, $L(w) = \frac{1}{n}\sum_{i=1}^n l_{\log}(z_i)$ is convex, but no global optimum exists. Using GD, $L(w)$ will approach 0, but the iterates $\{w^t \mid t \in \mathbb{N}\}$ diverge. However, w^t may converge *in direction*, and interestingly:

Th. **GD converges to w_{MM} for lin.-sep. data** (log. loss)

$$\lim_{t\to\infty} \frac{w^t}{\|w^t\|} = w_{\text{MM}}$$

On $L(w)$ (logistic regression), $\mu = 1$

2.5.2 On linearly inseperable data

Assuming $\{x_i, y_i\}_{i=1}^n$ is strictly lin. inseperable, i.e.

$$\forall w \neq 0: \quad \exists i \leq n: \quad y_i \cdot w^\top x_i < 0$$

Th. **GD converges on lin.-insep. data** (log. loss)

$$\exists \hat{w} \in \mathbb{R}^d: \quad \lim_{t\to\infty} w^t = \hat{w}$$

On $L(w)$ (logistic regression), $\mu = \frac{4}{\sigma^2_{\max}(X)}$

Rmk. Only holds for $l_{\log}$. In general, this is a hard problem.

2.6 Multiclass Classification

What if $|\mathcal{Y}| > 2$? E.g. $\mathcal{Y} = \{\text{cat}, \text{dog}, \text{fish}\}$

Idea: Train $K := |\mathcal{Y}|$ classifiers $\hat{f}_1, \dots, \hat{f}_K \in F$.

Why not one $\hat{f}$? E.g. discretize further: $1 \mapsto$ cat, $2 \mapsto$ dog, $3 \mapsto$ fish.
Problem: this assignment suggests cats are closer to dogs than fish.

We can then predict the class using these $\hat{f}_k$:

$$\hat{y}(x) = \arg\max_{1\leq k\leq K} \hat{f}_k(x)$$

This leads to one decision boundary per class.

D. **One-vs-Rest Training**

Train each model seperately by relabeling for each $\hat{f}_k$:

1. Define $\mathcal{D}_k = \{x_i, \tilde{y_i}\}$ where $\tilde{y_i} := \begin{cases} -1 & y_i = k \\ 1 & y_i \neq k \end{cases}$
2. Run binary classification on $\mathcal{D}_k$ to get $\hat{f}_k$

This leads to K classification problems, which might be slow

Another way to reuse the existing methodology is to use a new loss:

D. **Cross-Entropy Loss**

$$l_{\text{ce}}\Big(\hat{f}_1(x), \dots, \hat{f}_K(x),\ y\Big) = -\log\left(\frac{\exp\big(\hat{f}_y(x)\big)}{\sum_{k=1}^K \exp\big(\hat{f}_k(x)\big)}\right)$$

$y \in \{1, \dots, K\}, \quad \hat{f}_i(x) \in \mathbb{R}$

l_{ce} encourages the *true class* $\hat{f}_{y_i}(x_i)$ to be the largest $\hat{f}_k(x_i)$.

Rmk. For $K = 2$, if we use $\mathcal{Y} = \{+1, -1\}$ then $l_{\text{ce}} = l_{\log}$.

The parametrized optimization problem then is:

$$\min_{w_1,\dots,w_K \in \mathbb{R}^d} \left(\sum_{i=1}^n l_{\text{ce}}\Big(f_w(x_i), y_i\Big)\right)$$

Here, $w \in \mathbb{R}^{d\times K}$ is a matrix: $w = (w_1, \dots, w_K)$

This then yields $\hat{f}_k = f_{\hat{w}_k}$

Rmk. These methods may lead to very different decision boundaries!

2.7 Generalization

First, a lot of assumptions:

1. Inputs $X \in \mathcal{X}$ come from a prob. distribution $\mathbb{P}_X$
2. Training & Test set sampled i.i.d. from same distribution
   Note that in general, this is rarely true.
3. There exists a ground-truth y^*
4. The observed labels are noisy: $(y \mid x) = \epsilon \cdot y^*(x)$
   A *mutliplicative* noise model, unlike lin.-reg. : $\mathcal{Y} = f^*(X) + \epsilon$
5. ϵ is also from a prob. distribution $\mathbb{P}_\epsilon$
   Not necessarily indep. from x!

Focusing on $y^*(x) \in \{+1, -1\}$, we set $\epsilon \in \{+1, -1\}$.
Intuitvely: ϵ just flips the label

This allows defining a Joint-Distribution

$$\mathbb{P}[x, y] = \mathbb{P}_x[x] \cdot \mathbb{P}[y \mid x]$$

2.7.1 Evaluation

An intuitive metric to check is proximity to y^*:
Which can be done using the 0-1-loss

$$l\Big(\hat{y}(x), y^*(x)\Big) = \mathbb{I}_{\hat{y}(x)\neq y^*(x)}$$

Now, we can define the expected classification error:

$$\mathbb{E}_X\Big[l\big(\hat{y}(x), y^*(x)\big)\Big] = \mathbb{E}_X\Big[\mathbb{I}_{\hat{y}(x)\neq y^*(x)}\Big] = \mathbb{P}\Big[\hat{y}(x) \neq y^*(x)\Big]$$

We can't compute or estimate this, since we don't have y^*. However, we can find an estimate of $\mathbb{E}_{X,Y}\big[l(\hat{y}(X), Y)\big]$ using the observed X, Y.

Why is this useful? It approximates the generalisation error:

D. **Generalisation Error** (0-1-Loss)

$$L\Big(\hat{f}; \mathbb{P}_{X,Y}\Big) = \mathbb{E}_{X,Y}\Big[l\Big(\hat{f}(X), Y\Big)\Big] = \mathbb{P}_{X,Y}\Big(\hat{y} \neq y\Big)$$

We can empirically evaluate this on a test set:

$$\frac{1}{|\mathcal{D}_{\text{test}}|} \sum_{(x,y)\in\mathcal{D}_{\text{test}}} \mathbb{I}_{\hat{y}(x)\neq y^*(x)}$$

2.8 Hypothesis Testing

Classical statistical methods can also be used.

D. Asymmetric Errors

Misclassifications may be weighted differently.

Mistakenly classifying x with $y^*(x) = 1$ as 2, may be worse than 3.

For binary classification, we may label:

$$+1 \mapsto \text{"Positive"} \qquad -1 \mapsto \text{"Negative"}$$

This leads to the notion of confusion matrices.

	$y = -1$	$y = +1$
$\hat{y} = -1$	TN	FN (Type II)
$\hat{y} = +1$	FP (Type I)	TP

D. Empirical Measure

For an event $A \subset \mathcal{X} \times \{+1, -1\}$

$$\mathbb{P}_n[A] := \frac{1}{n} \sum_{i=1}^{n} \mathbb{I}_{(x_i, y_i) \in A}$$

$\mathcal{D} = \{(x_i, y_i) \mid i \leq n\} \subset \mathcal{X} \times \{+1, -1\}$

$\mathbb{P}_n[A]$ is the percentage of $(x, y) \in \mathcal{D}_{\text{train}}$ that belong to A.

L. $\mathbb{P}_n[A]$ is an estimate of $\mathbb{P}_{X,Y}[A]$:

$$\lim_{n \to \infty} \mathbb{P}_n[A] = \mathbb{P}_{X,Y}[A] \qquad \text{(Law of large numbers)}$$

Rmk. Asymmetric Loss in binary lassification

We can now weigh FP, FN differently in the 0-1-error:

$$\frac{c_{\text{FN}}}{|\{x \mid y = +1\}|} \underbrace{\sum_{(x,y), y=1} \mathbb{I}_{\hat{y} \neq -1}}_{\#\text{FN}} + \frac{c_{\text{FP}}}{|\{x \mid y = -1\}|} \underbrace{\sum_{(x,y), y=-1} \mathbb{I}_{\hat{y}(x) = +1}}_{\#\text{FP}}$$

Here $c_{\text{FP}}, c_{\text{FN}}$ are the weights for penalization

Generally, reducing FP errors increases FN errors, and vice versa.

2.9 ROC Curves

Rmk. A side-effect of using $\hat{y}(x) = \text{sign}\hat{f}(x)$ is that the magnitude $|\hat{f}(x)|$ can be interpreted as *confidence*.
We can set:

$$\hat{y}_\tau(x) = \text{sign}\Big(\hat{f}(x) - \tau\Big) = \begin{cases} +1 & \text{if } \hat{f}(x) > \tau \\ -1 & \text{if } \hat{f}(x) < \tau \end{cases}$$

Now τ can be used to penalize FP ($\tau > 0$) or FN ($\tau < 0$).

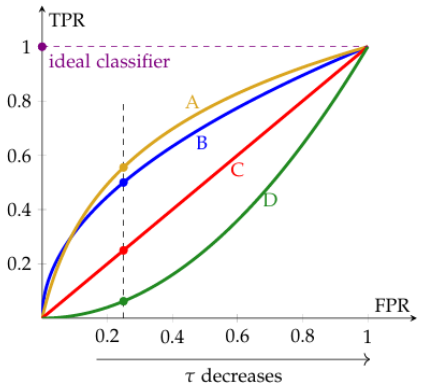
Note how this way, we don't modify the Optimization problem.

What if we don't know which FP/TP rate is desired?

Formally: which τ should be used?

D. ROC Curve (Receiver Operating Characteristic)
Plots TP Rate against FP Rate for different τ.

ROC Curve on 4 classifiers



Introduction to Machine Learning (2026), p. 160

Rmk. How to read this? A straight line is equivalent to random guessing, anything above is better. τ isn't directly included in the curve, but it follows from the definition that τ decreases as the FP rate increases.

How can we measure performance independent of τ?

D. AUROC (Area under ROC)
AUROC is 1 for the ideal classifier, and always in $[0, 1]$.

3 Kernels

Motivation: Regression using feature maps $\phi : \mathbb{R}^d \to \mathbb{R}^p$:

$$\min_{w \in \mathbb{R}^p} \frac{1}{n} \sum_{i=1}^{n} l\Big(y_i, w^\top \cdot \phi(x_i)\Big)$$

What if computing/storing $\phi(x)$ is expensive/infeasible?

e.g. if p is large, or infinite

Rmk. To store a poly. $p(x) : \mathbb{R}^d \to \mathbb{R}$ with $\deg(p) = m$ we require $p = \mathcal{O}(d^m)$ features. Storing n data points requires $\mathcal{O}(nd^m)$ memory. Not good.

3.1 Kernelization

By constraining w to $\text{span}(\Phi^\top) \subset \mathbb{R}^p$ we can drastically improve memory usage. Since we know a minimizer exists here, we don't "lose anything".

D. Kernelization

1. **Reparametrization**: We assume $w = \Phi^\top \alpha$ (i)
2. **Loss via Inner Products**: Observe:

$$f(x) = w^\top \phi(x) \overset{\text{(i)}}{=} (\Phi^\top \alpha)^\top \phi(x) = \sum_{i=1}^{n} \alpha_i \Big(\phi(x_i)^\top \phi(x)\Big)$$

   Note: x_i only appears in *inner products* $\phi(x_i)^\top \phi(x_j)$
3. **Replace Inner Products**: We define:

$$k : \begin{cases} \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R} \\ k(x, x') = \phi(x)^\top \phi(x') \end{cases} \qquad K : \begin{cases} K \in \mathbb{R}^{n \times n} \\ K_{ij} = k(x_i, x_j) \end{cases}$$

Now, we can reformulate the optimization problem:

$$\min_{\alpha \in \mathbb{R}^n} \frac{1}{n} \sum_{i=1}^{n} l\left(y_i, \sum_{j=1}^{n} \alpha_j k(x_i, x_j)\right) = \min_{\alpha \in \mathbb{R}^n} \frac{1}{n} \sum_{i=1}^{n} l\Big(y_i, (K\alpha)_i\Big)$$

By storing $K \in \mathbb{R}^{n \times n}$ instead of $\phi(x) \in \mathbb{R}^p$ for $i = 1, \ldots, n$, the memory usage is reduced: $\mathcal{O}(np) \to \mathcal{O}(n^2)$.

3.2 The Kernel Trick

Using k, the computation time is still $\mathcal{O}(n^2 p)$ if

$$k(x_i, x_j) = \phi(x_i)^\top \phi(x_j)$$

So let's replace k with a simple function, which guarantees the existance of some ϕ (which we never calculate).

Rmk. Since we only *implicitly* specify ϕ via k, we can use ϕ s.t. $p = \infty$ now.

D. **Kernel Function** $k : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$

1. k is symmetric: $\forall x, x' : \ k(x, x') = k(x', x)$
2. k is PSD: $\forall n \in \mathbb{N}, \forall (x_1, \dots, x_n) \in \mathbb{R}^d$:

$$K = \begin{bmatrix} k(x_1, x_1) & \cdots & k(x_1, x_n) \\ \vdots & \ddots & \vdots \\ k(x_n, x_1) & \cdots & k(x_n, x_n) \end{bmatrix} \text{ is PSD}$$

Th. **Kernels guarantee existance of ϕ**

If k is a kernel, there exists a Hilbert Space $\left(\mathcal{H}, \langle \cdot, \cdot \rangle_{\mathcal{H}}\right)$ s.t.

$$\exists \phi : \mathbb{R}^d \to \mathcal{H} \text{ s.t. } k(x.x') = \Big\langle \phi(x), \phi(x') \Big\rangle_{\mathcal{H}} \forall x, x' \in \mathbb{R}^d$$

$\mathcal{H}$ may be, for example, $\mathbb{R}^p$ with $\| \cdot \|_2$.

L. **Properties of Kernels**

1. Composed feature maps are Kernels

$$\left.\begin{aligned} \phi &: \mathbb{R}^d \to \mathbb{R}^p \\ \psi &: \mathbb{R}^d \to \mathbb{R}^p \end{aligned}\right\} \quad k(x, x') = \Big\langle \psi\big(\phi(x)\big), \psi\big(\phi(x')\big) \Big\rangle$$

2. Kernels can be added in 2 ways, yielding a kernel

$$\begin{aligned} &\text{(i)} \quad k\Big((x, y), (x', y')\Big) &&= k_1(x, x') + k_2(y, y') \\ &\text{(ii)} \quad k(x, x') &&= k_1(x, x') + k_2(x, x') \end{aligned}$$

3. Kernels can be multiplied in 2 ways, yielding a kernel

4 Neural Networks

Motivation: So far, when looking for $\hat{f}(x)$ the form was $\hat{f}(x) = w^\top x$, or $\hat{f}(x) = w^\top \phi(x)$. Note how the features $x, \phi(x)$ are predetermined. Why not learn them?

New Optimization Problem:

The new join-optimization problem, for w and ϕ:

Θ is a set of parameters for ϕ

$$\hat{w} = \underset{w \in \mathbb{R}^m, \Theta \in \mathbb{R}^{m \times d}}{\arg\min} \left(\frac{1}{n} \sum_{i=1}^{n} l\Big(w^\top \phi(x_i; \Theta), y_i\Big) \right)$$

Where $\phi(x, \Theta) = \Big(\phi_1(x; \theta_1), \dots, \phi_m(x; \theta_m)\Big)$.

θ_i is the ith row of Θ, i.e. $\theta_i := (\Theta)_{i,:}$

More compact, in terms of Θ, which combines w, ϕ:

$$\Theta^* := \arg\min_{\Theta} \Big(L(\Theta; \mathcal{D}) \Big) = \arg\min_{\Theta} \left(\frac{1}{n} \sum_{i=1}^{n} l\Big(\Theta; x_i, y_i\Big) \right)$$

Θ may also encapsulate w, ϕ for multiple layers, depending on definition

4.1 Definitions

D. **Activation Function**

We set $\phi_i(x; \theta_i) = \psi(\theta_i^\top x)$, ψ is the activation function.

$\theta_i \in \mathbb{R}^d, \quad \psi : \mathbb{R} \to \mathbb{R}$

Notation More concisely, $\phi(x; \Theta) = \psi(\Theta x)$

Activation Function	Definition
Identity	$\psi(z) = z$
Sigmoid	$\psi(z) = \frac{1}{1+e^{-z}}$
Hyperbolic tangent	$\psi(z) = \tanh(z)$
Rectified Linear Unit (ReLU)	$\psi(z) = \max(0, z)$

D. **Artificial Neural Network**

The output functions of the above problem take the form:

$$f(x; w, \theta) = \sum_{j=1}^{m} w_j \psi(\theta_j^\top x)$$

Rmk. Also called Multi-Layer Perceptron (MLP)

What is happening here?

Explaining the calculation steps for such an f naturally leads to the common pictorial depiction of neural networks.

$$\begin{aligned} &\text{(i)} \quad x &&= (x_1, \dots, x_n) \in \mathbb{R}^d && \text{(Input Vector)} \\ &\text{(ii)} \quad z &&= \Theta x && \text{(Linear transformation)} \\ &\text{(iii)} \quad h_i &&= \psi(z_i) && \text{(Activation function)} \\ &\text{(iv)} \quad f(x) &&= \sum_{j=1}^{m} w_j h_j && \text{(Output)} \end{aligned}$$

D. **Hidden Layer** $h = \psi(z)$

D. **Bias Term** $b \in \mathbb{R}^m$

Needed, as f might not pass through origin. Similar to using F_{lin} in regression, these can also be added by augmenting the input & hidden layers.

Does this work at all?

Yes, for most functions this does work.

D. **Sigmoidal Function**

$$\sigma(t) \text{ s.t. } \begin{cases} \sigma : \mathbb{R} \to \mathbb{R} \\ \lim_{t \to \infty} = 1 \text{ and } \lim_{t \to \infty} \end{cases}$$

Th. **Universal Approximation Theorem**

$\hat{f}$, that uniformly approximates f, exists and takes this form:

$$\hat{f}(x) = \mathbf{W}^{(2)} \psi\Big(\mathbf{W}^{(1)} x + b\Big)$$

$f : [0, 1]^d \to \mathbb{R}$ continuous, $\quad \psi$ sigmoidal

$\mathbf{W}^{(1)} \in \mathbb{R}^{m \times d}, \quad \mathbf{W}^{(2)} \in \mathbb{R}^{1 \times m}, \quad m \in \mathbb{N}$

Note how m could be very large.

m can intuitively be understood as the "width" of the ANN

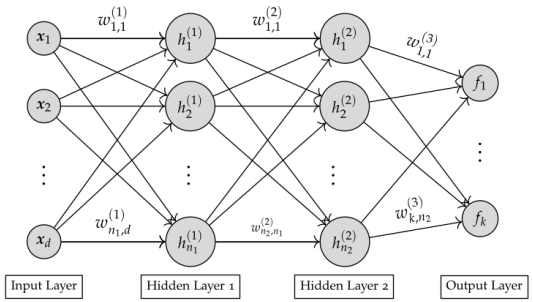
D. Fully Connected Neural Network

More complex ANNs might have:

1. More hidden layers
2. Multiple outputs
3. Differen activation functions across layers

These are called *fully connected*, since every node in a layer is connected to every node in the adjacent layers.

There are also more complex architectures.



Introduction to Machine Learning (2026), p. 183

Notation Weights: $\mathbf{W}^{(i)} := \left[w_{k,l}^{(i)}\right]$, Biases: $b_k^{(i)}$ and $\Theta = \left(\mathbf{W}^{(1)}, \dots, \mathbf{W}^{(L)}, b^{(1)}, \dots, b^{(L)}\right)$ (All parameters)

$w_{k,l}^{(i)}$: "Weight at layer i to node k from node l"

4.2 Forward Propagation

How can we make predictions, i.e. how can $\hat{f}$ be evaluated?

D. Forward Propagation

This is just the computation for 1-layer ANN generalized for L layers

Algorithm 1: Forward Propagation

$h^{(0)} \leftarrow x$;
for $l = 1, \dots, L$ **do**
  $z^{(l)} = \mathbf{W}^{(l)} h^{(l-1)} + b^{(l)}$
  $h^{(l)} = \psi(z^{(l)})$
end
$f \leftarrow \mathbf{W}^{(L)} h^{(L-1)} + b^{(L)}$
return f

4.3 Backwards Propagation

How can we get all gradients needed for model training?

D. Backwards Propagation

Intuition: An efficient way to get the gradients is to reuse results from forward prop. and previous steps. This works best when starting at the back, at $\nabla_{\mathbf{W}^{(L)}} l$.

Goal: $\nabla_{\mathbf{W}^{(1)}} l, \dots, \nabla_{\mathbf{W}^{(L)}} l, \nabla_{b^{(1)}} l, \dots, \nabla_{b^{(L)}} l$

Step 1: Calculate $\nabla_{\mathbf{W}^{(L)}} l$, i.e. start from the back.

$$\begin{aligned}
\nabla_{\mathbf{W}^{(L)}} l &= \frac{\partial l}{\partial \mathbf{W}^L} \\
&= \frac{\partial l}{\partial f} \cdot \frac{\partial f}{\partial \mathbf{W}^{(L)}} && \text{(Chain Rule)} \\
&= \frac{\partial l}{\partial f} \cdot \begin{bmatrix} \left(h^{(L-1)}\right)^\top \\ \vdots \\ \left(h^{(L-1)}\right)^\top \end{bmatrix} && \left(f = \mathbf{W}^{(L)} h^{(L-1)} + b^{(L)}\right) \\
&= \nabla_f l \cdot \begin{bmatrix} \left(h^{(L-1)}\right)^\top \\ \vdots \\ \left(h^{(L-1)}\right)^\top \end{bmatrix} && \left(\frac{\partial l}{\partial f} = \nabla_f l\right)
\end{aligned}$$

Notice how $h^{(L-1)}$ was computed during forward prop.

Step 2: Calculate $\nabla_{\mathbf{W}^{(L-1)}} l$.

$$\nabla_{\mathbf{W}^{(L-1)}} l = \underbrace{\frac{\partial l}{\partial f}}_{(1)} \cdot \underbrace{\frac{\partial f}{\partial h^{(L-1)}}}_{(2)} \cdot \underbrace{\frac{\partial h^{(L-1)}}{\partial z^{(L-1)}}}_{(3)} \cdot \underbrace{\frac{z^{(L-1)}}{\partial \mathbf{W}^{(L-1)}}}_{(4)} \quad \text{(Chain Rule)}$$

1. Already done in Step 1.
2. Already done in forward propagation, equal to $\mathbf{W}^{(L)}$:
$$f \overset{\text{def}}{=} \mathbf{W}^{(L)} h^{(L-1)} + b^{(L)} \implies \frac{\partial f}{\partial h^{(L-1)}} = \mathbf{W}^{(L)}$$
3. **Not done.** Needs to be calculated:
$$\begin{aligned}
\frac{\partial h^{(L-1)}}{\partial z^{(L-1)}} &= \frac{\partial \psi\left(z^{(L-1)}\right)}{\partial z^{(L-1)}} \\
&= \operatorname{diag}\left(\psi'\left(z^{(L-1)}\right)\right) \\
&= \begin{bmatrix} \psi'\left(z_1^{(L-1)}\right) & 0 & \cdots & 0 \\ 0 & \psi'\left(z_2^{(L-1)}\right) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \psi\left(z_n^{(L-1)}\right) \end{bmatrix}
\end{aligned}$$
4. Already done in forward propagation, analogous to step 1.
$$\frac{\partial z^{(L-1)}}{\partial \mathbf{W}^{(L-1)}} = \begin{bmatrix} \left(h^{(L-2)}\right)^\top \\ \vdots \\ \left(h^{(L-2)}\right)^\top \end{bmatrix}$$

Step $i \leq L$: Calculate $\nabla_{\mathbf{W}^{(L-i)}} l$ Analogoues to step 2.

The biases $\nabla_{b^{(l)}} l$ are analogous.

4.4 Optimization

Problem: How can we train the model, i.e find Θ^*?

$$\Theta^* := \arg\min_{\Theta}\Big(L(\Theta;\mathcal{D})\Big) = \arg\min_{\Theta}\left(\frac{1}{n}\sum_{i=1}^{n} l\Big(\Theta; x_i, y_i\Big)\right)$$

Rmk. $L(\Theta;\mathcal{D})$ is generally not convex.
i.e. local minima, saddle points may exist

Rmk. $\dim(\Theta)$ is the total param. count of NN, may be very large

Solution: Gradient Descent (with optimizations)

- Stochastic Gradient Descent
  (Why? $\dim(\Theta)$ is very large, $\nabla_\Theta l(\Theta; x_i, y_i)$ are expensive)
- Minibatch Gradient Descent
  (Why? $\mathcal{D}$ may be very large, so there are *many* gradients)

The standard GD update for Θ is:

$$\Theta^{t+1} = \Theta^t - \eta_t \cdot \nabla_\Theta L\Big(\Theta;\mathcal{D}\Big)$$

In Minibatch GD, this becomes:
Where $\mathcal{S} \subset \{1,\dots,n\}$

$$\Theta^{t+1} = \Theta^t - \eta_t \cdot \nabla_{\Theta^t}\left(\frac{1}{|\mathcal{S}|}\sum_{i\in\mathcal{S}} l\Big(\Theta^t; x_i, y_i\Big)\right)$$

Rmk. An advantage: If Θ^t approaches a stat. point (which isn't the global minimum), GD will converge, but MB-GD may not converge.

The further subsections go into more details:

1. Preventing vanishing & exploding Gradients
2. Choice of initial w_i
3. Choice of μ_t

4.4.1 Vanishing & Exploding Gradients

$$\nabla_{\Theta^t}\left(\frac{1}{|\mathcal{S}|}\sum_{i\in\mathcal{S}} l\Big(\Theta^t; x_i, y_i\Big)\right) = \frac{1}{|\mathcal{S}|}\sum_{i\in\mathcal{S}}\Big(\nabla_{\Theta^t} l(\Theta^t; x_i, y_i)\Big)$$

The terms $\nabla_{\Theta^t} l\Big(\Theta^t; x_i, y_i\Big)$ each contain $\nabla_{\mathbf{W}^{(l)}} l\Big(\mathbf{W}^{(l)}; x_i, y_i\Big)$.

Problem: Optimization might fail if:

$$\Big\|\nabla_{\mathbf{W}^{(l)}} l\Big\| \to \infty \qquad \text{or} \qquad \Big\|\nabla_{\mathbf{W}^{(l)}} l\Big\| \to 0$$

Solution: $\|\nabla_{\mathbf{W}^{(l)}} l\|$ depends linearly on $\text{diag}\Big(\psi'(z^{(l)})\Big)$, so the choice of ψ (ψ') can be used to constrain $\|\nabla_{\mathbf{W}^{(l)}} l\|$
Generally, the gradient follows the behaviour of ψ'

Which features do we want ψ (ψ') to fullfil?

- ψ' should be fast to calculate
- ψ' should be non-zero (and not get too close)

Rmk. The $\Big\|\nabla_{\mathbf{W}^{(l)}} l\Big\|$ may still vanish for any ψ.

4.4.2 Random Weight Initialization

$$\begin{aligned}\nabla_{\mathbf{W}^{(l)}} l &= \frac{\partial l}{\partial f}\cdot\frac{\partial f}{\partial h^{(l)}}\cdot\frac{\partial h^{(l)}}{\partial z^{(l)}}\cdot\frac{\partial z^{(l)}}{\partial \mathbf{W}^{(l)}} \\ &= \frac{\partial l}{\partial f}\cdot\frac{\partial f}{\partial h^{(l)}}\cdot \text{diag}\Big(\psi'(z^{(l)})\Big)\cdot\begin{bmatrix}(h^{(l-1)})^\top \\ \vdots \\ (h^{(l-1)})^\top\end{bmatrix}\end{aligned}$$

So, the gradient $\|\nabla_{\mathbf{W}^{(l)}} l\|$ also depends on $\|h^{(l-1)}\|$.
For $l \in \{1,\dots,L\}$

Problem: It might be that $\|h^{(l-1)}\| \to \infty$ or $\|h^{(l-1)}\| \to 0$.

Solution: Set $h^{(l-1)}$ randomly, bound mean μ and var. σ^2.
There is no generally optimal bound for σ^2, it depends on the NN.

Rmk. Practical distributions for common ψ:
$n_{\text{out}}, n_{\text{out}}$ are the node counts of the layers adjacent to w_i

ψ	**Weights**
tanh	$w_i \sim \mathcal{N}\Big(0, \frac{1}{n_{\text{in}}}\Big)$
tanh	$w_i \sim \mathcal{N}\Big(0, \frac{2}{n_{\text{in}}+n_{\text{out}}}\Big)$
ReLu	$w_i \sim \mathcal{N}\Big(0, \frac{2}{n_{\text{in}}}\Big)$

4.4.3 Learning Rate & Weight Updates

$$\Theta^{t+1} = \Theta^t - \mu_t \cdot \nabla_{\Theta^t}\left(\frac{1}{|\mathcal{S}|}\sum_{i\in\mathcal{S}} l\Big(\Theta^t; x_i, y_i\Big)\right)$$

Problem: How to choose μ? (Learning Rate)

Solution: Heuristics.
There is no generally optimal μ.

Method **Piecewise constant μ_t**

Intuitively, it makes sense to reduce μ_t as optimization progresses, as the algorithm approaches the minimum.
Linear/cosine decay could also be used.

$$\mu_t = \begin{cases} 1 & 0 \geq t < 3 \\ 0.5 & 3 \leq t < 6 \\ 0.25 & 6 \leq t < 9 \\ \dots & \end{cases}$$

Method **Weight update indicator**

In practice, SGD often oscillates in finding the minimum. Then, a monotonic μ_t doesn't make sense.

$$\frac{\Big|\nabla_{\Theta^t} L(\Theta^t;\mathcal{D})\Big|}{\|\Theta^t\|}$$

In general, if this indicator ratio is small, a higher learning rate makes sense (and vice versa).
Intuitively: How strongly is the weight change, relative to weight size.

Method **Momentum**

Combine the update direction with the previous update directions, for some weight $m > 0$, to stabilize.

4.5 Regularization

Problem: How can overfitting be avoided?

A few methods can be applied directly to SGD:

Method **Penalty Term**

Similar to Ridge/LASSO Regression, a penalty term can be used, with some weight $\lambda > 0$.

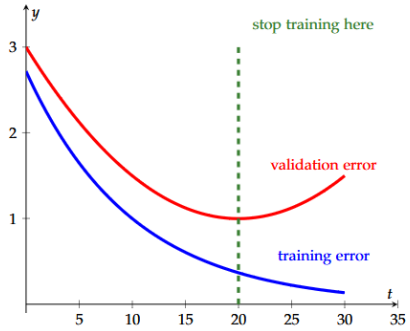
$$\arg\min_{\Theta \in \mathbb{R}^d} \Big(L(\Theta; \mathcal{D}) \Big) \quad \rightarrow \quad \arg\min_{\Theta \in \mathbb{R}^d} \Big(L(\Theta; \mathcal{D}) + \lambda \|\Theta\|^2 \Big)$$

Method **Earlier stop**

Choosing a different stop criterion for SGD, e.g. performance on the test set $\mathcal{D}'$.

Rmk. **Validation & Training Error**

Overfitting occurs when the training error (on $\mathcal{D}$) continues to fall, but the test error increases (on $\mathcal{D}'$).



Introduction to Machine Learning (2026), p. 196

4.5.1 Dropout Regularization

This is a method specific to Neural Networks.

Method **Dropout**

Fix some $p \in (0, 1)$. For each SGD iteration in training: *Drop out* each hidden unit with probability $1 - p$, and skip their optimization for this iteration.

$$z_j^{(l)} = \sum_{i=0}^{n_{i-1}} \Big(w_{j,i}^{(l)} h_i^{(l-1)} \Big) + b_j^{(l)} \qquad \text{(Regular Neuron)}$$

$$z_j^{(l)} = \sum_{i=0}^{n_{i-1}} \Big(w_{j,i}^{(l)} h_i^{(l-1)} \cdot \mathbb{I}_{C_i} \Big) + b_j^{(l)} \qquad \text{(With Dropout)}$$

Where $C_i := \{\text{"Unit } h_i^{(l-1)} \text{ is kept this iter."}\}$, so $\mathbb{P}[C_i] = p$

Problem: For $\mathcal{D}'$, we again want to use all layers.

Solution: Scale all weights with p

For this, we use $\mathbb{E}\Big[z_j^{(l)}\Big]$ instead of $z_j^{(l)}$.

$$\mathbb{E}\Big[z_j^{(l)}\Big] = \Big(p \cdot w_j^{(l)}\Big)^\top \cdot h^{(l-1)} + b_j^{(l)}$$

By using $\mathbb{E}\big[\mathbb{I}_{C_i}\big] = \mathbb{P}[C_i] = p$

4.5.2 Batch Normalization

During SGD, the weight's σ^2 may again explode.

After few iterations, w_i may have changed completely from init.

Problem: Internal covariate shift.

The mean μ deviates from 0 and σ^2 might increase

Solution: Standardize ψ also during training.

Method **Batch Normalization**

In pratice, only batches of ψ are normalized. The core idea is to set $\mu \mapsto 0$ and $\sigma^2 \mapsto 1$ within the batch.

This isn't optimal for all problems, and can be tweaked using β, γ

The algorithm uses the parameters β, γ and buffers $\mu_{\text{EMA}}, \sigma^2_{\text{EMA}}$.

β, γ are learnable and can also be optimized

Normalization Step (Training set)

For a minibatch $\mathcal{S} = \{i_1, \ldots, i_k\}$, the batch is $\{x_{i_1}, \ldots, x_{i_k}\}$.

Step 1: Find current values of μ, σ^2.

$$\mu_{\mathcal{S}} := \frac{1}{|\mathcal{S}|} \sum_{j \in \mathcal{S}} x_j \qquad \text{(minibatch mean)}$$

$$\sigma^2_{\mathcal{S}} := \frac{1}{|\mathcal{S}|} \sum_{j \in \mathcal{S}} \Big(x_i - \mu_{\mathcal{S}} \Big)^2 \qquad \text{(minibatch variance)}$$

Step 2: Update the moving average: $\mu_{\text{EMA}}, \sigma^2_{\text{EMA}}$.

$$\mu_{\text{EMA}} = (1 - \alpha)\mu_{\text{EMA}} + \alpha \cdot \mu_{\mathcal{S}} \qquad \text{(avg. mean update)}$$

$$\sigma^2_{\text{EMA}} = (1 - \alpha)\sigma^2_{\text{EMA}} + \alpha \cdot \sigma^2_{\mathcal{S}} \qquad \text{(avg. variance update)}$$

Step 3: Update the x_j.

$$\hat{x}_j = \frac{x_j - \mu_{\mathcal{S}}}{\sqrt{\sigma^2_{\mathcal{S}} + \epsilon}} \qquad \text{(point normalization)}$$

$$\bar{x}_j = \gamma \cdot \hat{x}_j + \beta \qquad \text{(scale \& shift)}$$

Normalization Step (Test set)

Only apply step 3, now using the moving average values.

$$\hat{x}_j = \frac{x_j - \mu_{\text{EMA}}}{\sqrt{\sigma^2_{\text{EMA}} + \epsilon}} \qquad \text{(point normalization)}$$

$$\bar{x}_j = \gamma \cdot \hat{x}_j + \beta \qquad \text{(scale \& shift)}$$

4.6 Convolutional Neural Networks

In fully connected NNs:

$$h^{(l)} = \psi\Big(\mathbf{W}^{(l)} h^{(l-1)}\Big)$$

Each unit of layer $l-1$ affects each unit of layer l.
In CNNs, this is relaxed: not all nodes (must) interact.

D. **Convolutional Neural Network** (CNN)
Layers are connected via convolutions.

$$h^{(l)} = \psi\Big(w^{(l)} * h^{(l-1)}\Big)$$

Rmk. In CNNs, the weights are also called *filters*.

D. **Convolution** (Discrete, 2D)
$w \in \mathbb{R}^k, \quad x \in \mathbb{R}^d$

$$w * x := \sum_{j=\max\{1,i-d+1\}}^{\min\{i,k\}} \Big(w_j \cdot x_{i-j+1}\Big)$$

Understanding this is easier by example:

Example: $w = (w_1, w_2)^\top, \quad x = (x_1, x_2, x_3)^\top$

$$w * x = \begin{bmatrix} w_1 \cdot x_1 + w_2 \cdot 0 \\ w_1 \cdot x_2 + w_2 \cdot x_1 \\ w_1 \cdot x_3 + w_2 \cdot x_2 \\ w_1 \cdot 0 + w_2 \cdot x_3 \end{bmatrix}$$

Example: a CNN with 3 inputs and 1 hidden layer:

$$\begin{bmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \end{bmatrix} = \underbrace{\begin{bmatrix} w_1 & 0 & 0 \\ w_2 & w_1 & 0 \\ 0 & w_2 & w_1 \\ 0 & 0 & w_2 \end{bmatrix}}_{\mathbf{W}^{(1)} \text{ in CNN}} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = w * x$$

4.6.1 Multidimensional Convolution

TODO add explanation

5 Unsupervised Learning

In **Unsupervised Learning**, $\mathcal{D}$ contains no labels.
Models both define labels & assign inputs to labels.

$$\mathcal{D} = \Big\{x_1, \dots, x_n\Big\} \qquad \text{(Dataset in unsupervised learning)}$$

There are many use-cases:

1. Compression
2. Discovery of latent variables
3. Anomaly detection
4. Exploratory data analysis

5.1 Clustering

D. **Clustering**

The goal here is to group inputs into clusters, based on some definiton of similarity, e.g. l_2 distance for $\mathcal{D} \subset \mathbb{R}^2$.
This can be seen as the unsupervised analogy to classification

Method **Hierarchical Clustering**

A simple method, using the "similarity" measure directly.

1. Each $x \in \mathcal{D}$ starts in its own cluster
2. Iteratively, the 2 "closest" clusters are merged

This results in a tree, thus *hierarchical* clustering.

Method **Partitioning**

In Partitioning methods, a weighted graph is constucted using $\mathcal{D}$ and partitioned using graph theory approaches, i.e. using cuts or spectral analysis.

Rmk. Both Hierarchical and Partitioning do not give a natural way to deduce cluster membership for new datapoints.

5.1.1 k-Means Clustering

In k-means, a cluster is represented by its center: $\mu_j \in \mathbb{R}^d$.
The cluster assignment z_i for $x_i \in \mathcal{D}$:

$$z_i = \underset{j=1,\dots,k}{\arg\min} \Big\|x_i - \mu_j\Big\| \qquad \text{(Closest center)}$$

Rmk. This strategy induces a partition of $\mathbb{R}^d$. (Voronoi Pattern)

Problem: How to find $\mu = (\mu_1, \dots, \mu_k)^\top$?

A new optimization objective:

$$\hat{R}(\mu) = \sum_{i=1}^{n} \min_{j \in \{1,\dots,k\}} \Big\|x_i - \mu_j\Big\|^2 = \sum_{i=1}^{n} \Big\|x_i - \mu_{z_i}\Big\|$$

(minimize the sum of sq. distances between points & their centers)

So we are searching:

$$\arg\min_{\mu} \Big(\hat{R}(\mu)\Big) \qquad \text{(optimal } k\text{-means cluster)}$$

Rmk. This is non-convex & NP-hard.

Method **Lloyd's Heuristic**

This is an iterative method to find the cluster centers.

D. $z^{(t)} = \Big(z_1^{(t)}, \dots, z_n^{(t)}\Big)^\top$ (assignment of x_i at iter. t)

D. $\mu^{(t)} = \Big(\mu_1^{(t)}, \dots, \mu_k^{(t)}\Big)^\top$ (Cluster centers at iter. t)

D. $n_j^{(t)} = \Big|\Big\{i = 1, \dots, n \;\Big|\; z_j^{(t)} = j\Big\}\Big|$ (Size of cluster j at iter. t)

Algorithm 2: Lloyd's Heuristic

$\mu^{(0)} \leftarrow \Big(\mu_1^{(0)}, \dots, \mu_k^{(0)}\Big)$;
repeat
  $z_i^{(t)} \leftarrow \underset{j \in \{1,\dots,k\}}{\arg\min} \Big\|x_i - \mu_j^{(t-1)}\Big\|$ for $i = 1, \dots, n$;
  $\mu_j^{(t)} \leftarrow \frac{1}{n_j^{(t)}} \sum_{i \text{ s.t. } z_i^{(t)} = j} x_i$ for $j = 1, \dots, k$;
  $t \leftarrow t + 1$;
until *convergence*;

Rmk. Each iteration is in $\mathcal{O}(nkd)$.

5.2 Principal Component Analysis