Statistical Machine Learning

https://cvml.ist.ac.at/courses/SML_W20

Christoph Lampert (with material by Andrea Palazzi and others)

IST AUSTRIA

Institute of Science and Technology

Fall Semester 2020/2021

Lecture 9

Overview (tentative)

Date		no.	Topic
Oct 05	Mon	1	A Hands-On Introduction
Oct 07	Wed	2	Bayesian Decision Theory, Generative Probabilistic Models
Oct 12	Mon	3	Discriminative Probabilistic Models
Oct 14	Wed	4	Maximum Margin Classifiers, Generalized Linear Models
Oct 19	Mon	5	Estimators; Overfitting/Underfitting, Regularization, Model Selection
Oct 21	Wed	6	Bias/Fairness, Domain Adaptation
Oct 26	Mon	-	no lecture (public holiday)
Oct 28	Wed	7	Learning Theory I, Concentration of Measure
Nov 02	Mon	8	Learning Theory II
Nov 04	Wed	9	Learning Theory III, Deep Learning I
Nov 09	Mon	10	Deep Learning II
Nov 11	Wed	11	Deep Learning III
Nov 16	Mon	12	project presentations
Nov 18	Wed	13	buffer

Inferring the test loss
from the training loss

Generalization Bound

For every $f \in \mathcal{H}$ it holds:

$$\underbrace{\mathop{\mathbb{E}}_{(x,y)} \ell(y, f(x))}_{\text{generalization loss}} \quad \leq \quad \underbrace{\frac{1}{n} \sum_i \ell(y_i, f(x_i))}_{\text{training loss}} \quad + \quad \text{something}$$

The Power of Randomization

The problem of overfitting emerges mainly because we pick only a single classifier, h, and just by accident it can have $\mathcal{R}(h) \gg \hat{\mathcal{R}}(h)$.

Combining the decisions of many classifiers should lower the chances of overfitting.

Definition (Majority-vote)

Let $\mathcal{Y} = \{\pm 1\}$ (only for convenience of notation). Let $h_1, \ldots, h_T \in \mathcal{H}$ be a set of hypotheses. We define the **uniform majority vote** classifier as

$$h_{\text{majority}}(x) = \operatorname{sign} \frac{1}{T} \sum_{i=1}^{T} h_i(x)$$

Definition (Majority-vote)

More generally, for weights $\alpha_i \in [0,1]$, $\sum_i \alpha_i = 1$, the α-**weighted majority vote classifier** is:

$$h^{\alpha}_{\text{majority}}(x) = \operatorname{sign} \sum_{i=1}^{T} \alpha_i h_i(x) = \mathop{\mathbb{E}}_{i \sim \alpha} [h_i(x)]$$

Weighting make a convenient framework:

- we can use a base set of many (even countably infinite) classifier
- we assign non-zero weights to *good classifiers*, e.g. based on training data
- classical setting is included: set $\alpha_i = \delta_{i=j}$, then $h^{\alpha}_{\text{majority}} = h_j$

Definition (Majority-vote)

More generally, for weights $\alpha_i \in [0,1]$, $\sum_i \alpha_i = 1$, the α-**weighted majority vote classifier** is:

$$h^{\alpha}_{\text{majority}}(x) = \operatorname{sign} \sum_{i=1}^{T} \alpha_i h_i(x) = \mathop{\mathbb{E}}_{i \sim \alpha} [h_i(x)]$$

Weighting make a convenient framework:

- we can use a base set of many (even countably infinite) classifier
- we assign non-zero weights to *good classifiers*, e.g. based on training data
- classical setting is included: set $\alpha_i = \delta_{i=j}$, then $h^{\alpha}_{\text{majority}} = h_j$

Unfortunately, majority vote classifiers are not easy to categorize:

- classical bounds hold equally for *any* $h \in \mathcal{H}$
- if $h^{\alpha}_{\text{majority}} \in \mathcal{H}$, bound no better than for others
- if $h^{\alpha}_{\text{majority}} \notin \mathcal{H}$, no bound at all

Trick: analyze stochastic classifiers

Stochastic Classifiers

Standard scenario:

- $\mathcal{X}$: input set, $\mathcal{Y}$: output set, p probability distribution over $\mathcal{X} \times \mathcal{Y}$
- $\mathcal{H} \subset \{\mathcal{X} \to \mathcal{Y}\}$: hypothesis set, ℓ: loss function
- $\mathcal{D} = \{(x^1, y^1) \ldots, (x^n, y^n)\} \overset{i.i.d.}{\sim} p(x, y)$: training set

Stochastic Classifiers

Standard scenario:

- $\mathcal{X}$: input set, $\mathcal{Y}$: output set, p probability distribution over $\mathcal{X} \times \mathcal{Y}$
- $\mathcal{H} \subset \{\mathcal{X} \to \mathcal{Y}\}$: hypothesis set, ℓ: loss function
- $\mathcal{D} = \{(x^1, y^1) \ldots, (x^n, y^n)\} \overset{i.i.d.}{\sim} p(x, y)$: training set

New:

- Q probability distribution over $\mathcal{H}$

Definition (Gibbs classifier)

For a distribution Q over $\mathcal{H} \subset \{h : \mathcal{X} \to \mathcal{Y}\}$, the Gibbs classifier, h_Q, is defined by the procedure:

- input: $x \in \mathcal{X}$
- sample $h \sim Q$
- output: $h(x)$

The Gibbs classifier is a stochastic classifier, its output is a random variable (wrt Q).

Definition (Gibbs classifier)

For a distribution Q over $\mathcal{H} \subset \{h : \mathcal{X} \to \mathcal{Y}\}$, the **Gibbs classifier**, h_Q, is defined by the procedure:

- input: $x \in \mathcal{X}$
- sample $h \sim Q$
- output: $h(x)$

Because the classifier output is random, so are the risks:

$$\mathcal{R}(h_Q) = \mathop{\mathbb{E}}_{(x,y)\sim p} \ell(y, h_Q(x)) \qquad \hat{\mathcal{R}}(h_Q) = \sum_{i=1}^{n} \ell(y^i, h_Q(x^i))$$

We can study their expected value:

$$\mathcal{R}(Q) = \mathop{\mathbb{E}}_{h\sim Q} \mathcal{R}(h) = \mathop{\mathbb{E}}_{h\sim Q} \mathop{\mathbb{E}}_{(x,y)\sim p} \ell(y, h(x)) \qquad \hat{\mathcal{R}}(Q) = \mathop{\mathbb{E}}_{h\sim Q} \sum_{i=1}^{n} \ell(y^i, h(x^i))$$

- $\mathcal{X}$: input set, $\mathcal{Y}$: output set, p probability distribution over $\mathcal{X} \times \mathcal{Y}$
- $\mathcal{H} \subset \{\mathcal{X} \to \mathcal{Y}\}$: hypothesis set, ℓ: loss function

What's the analog of deterministic learning?

Given a training set, $\mathcal{D} = \{(x^1, y^1) \ldots, (x^n, y^n)\} \overset{i.i.d.}{\sim} p(x, y)$, identify a distribution Q (arbitrary, or from a parametric family), such that $\mathcal{R}(Q)$ is as small as possible.

What would a generalization bound look like?

$$\mathcal{R}(Q) \leq \hat{\mathcal{R}}(Q) + \text{"something"}$$

Majority vote classifier: (now calling weights Q instead of α)

- evaluate all classifiers, $h(x)$ for $h \in \mathcal{H}$
- combine their outputs according to their weights, $\mathbb{E}_{h\sim Q}\, h(x)$
- make one decision based on the result, $\operatorname{sign} \mathbb{E}_{h\sim Q}\, h(x)$
- evaluate the loss of this decision, $\ell(y, \operatorname{sign} \mathbb{E}_{h\sim Q}\, h(x))$

Gibbs classifier:

- evaluate all classifiers, $h(x)$ for $h \in \mathcal{H}$
- evaluate the loss of all their decisions, $\ell(y, h(x))$ for $h \in \mathcal{H}$
- combine their losses according to their weights, $\mathbb{E}_{h\sim Q}\, \ell(y, h(x))$

How are the two situations related?

Lemma

$$\mathcal{R}_{majority}(Q) \leq 2\mathcal{R}_{Gibbs}(Q)$$

Observation:

$$h^Q_{\text{majority}}(x) = \operatorname{sign} \mathop{\mathbb{E}}_{h\sim Q} h(x) = \begin{cases} +1 & \text{if more than 50\% (probability mass) of the individual classifiers say } +1 \\ -1 & \text{otherwise} \end{cases}$$

$$\begin{aligned} \ell(y, h_{\text{majority}}(x)) = 1 \quad &\Rightarrow \quad \Pr_{h\sim Q}\{\ell(y, h(x)) = 1\} \geq 0.5 \\ \ell(y, h_{\text{majority}}(x)) = 1 \quad &\Rightarrow \quad 2 \mathop{\mathbb{E}}_{h\sim Q}[\ell(y, h(x))] \geq 1 \\ 2 \mathop{\mathbb{E}}_{h\sim Q}[\ell(y, h(x))] \quad &\geq \quad \ell(y, h_{\text{majority}}(x)) \\ 2\,\mathcal{R}_{\text{Gibbs}}(Q) \quad &\geq \quad \mathcal{R}_{\text{majority}}(Q) \end{aligned}$$

Generalization bounds for $\mathcal{R}_{\text{Gibbs}}$ also hold for $\mathcal{R}_{\text{majority}}$ (up to factor 2).

Theorem (PAC-Bayesian generalization bound [McAllester, 1999]; many others (also tighter ones) exist**)**

Let the loss, ℓ, be a bounded in $[0,1]$. Let P be a "prior" distribution of $\mathcal{H}$, chosen independently of $\mathcal{D}$. With prob $1-\delta$ over $\mathcal{D} \overset{i.i.d.}{\sim} p^{\otimes n}$, it holds for all "posterior" distributions Q:

$$\mathcal{R}(Q) \leq \hat{\mathcal{R}}(Q) + \frac{1}{\sqrt{n}}\Big(KL(Q\|P) + \frac{1}{8} + \log\frac{1}{\delta}\Big)$$

Called PAC-Bayesian, because it makes a PAC-style statement (different between finite sample and expect error), but for Bayesian-style objects (distributions over predictors).

"Prior" and "posterior" are in quotation marks, because

- the prior is only a technical tool and shows up in the KL term. We don't have to "believe" in it or anything.
- the posterior is not the result of applying Bayes' rule.

Towards a proof:

Theorem (Change of Measure Inequality)

For any distributions P, Q over $\mathcal{H}$ and function $\phi : \mathcal{H} \to \mathbb{R}$:

$$\mathop{\mathbb{E}}_{h\sim Q}[\phi(h)] \leq \frac{1}{\lambda}\Big(KL(Q\|P) + \log \mathop{\mathbb{E}}_{h\sim P} e^{\lambda\phi(h)}\Big)$$

$$\textit{with} \qquad KL(Q\|P) = \mathop{\mathbb{E}}_{h\sim Q}\Big[\log \frac{Q(h)}{P(h)}\Big]$$

We shift from an expectation over Q to an expectation over P.

Very useful, e.g.

- P will be a typically a simple, data-independent, distribution
- Q will depend on a training set $\quad \to$ "trained classifier"

The price we "pay" for this: 1) $\mathrm{KL}(Q\|P)$ and 2) $\mathbb{E}_Q(\cdot)$ turns into $\log \mathbb{E}_P \exp(\cdot)$

Proof sketch, pretending P and Q have densities.

General observation:

$$\mathop{\mathbb{E}}_{h\sim P}[f(h)] = \int_{\mathcal{H}} P(h)f(h)dh = \int_{\mathcal{H}} Q(h)\frac{P(h)}{Q(h)}f(h)dh = \mathop{\mathbb{E}}_{h\sim Q}[\frac{P(h)}{Q(h)}f(h)]$$

$$\begin{aligned}
\log \mathop{\mathbb{E}}_{h\sim P}[e^{\lambda\phi(h)}] &= \log \mathop{\mathbb{E}}_{h\sim Q}\left[e^{\lambda\phi(h)}\frac{P(h)}{Q(h)}\right] \\
&\overset{\text{Jensen's ineq.}}{\geq} \mathop{\mathbb{E}}_{h\sim Q}\left[\log e^{\lambda\phi(h)}\frac{P(h)}{Q(h)}\right] \\
&= \mathop{\mathbb{E}}_{h\sim Q}\left[\lambda\phi(h) - \log\frac{Q(h)}{P(h)}\right] \\
&= \lambda \mathop{\mathbb{E}}_{h\sim Q}[\phi(h)] - \mathsf{KL}(Q\|P)
\end{aligned}$$

$$\overset{\text{rearrange, } \cdot\frac{1}{\lambda}}{\Rightarrow} \quad \mathop{\mathbb{E}}_{h\sim Q}[\phi(h)] \leq \frac{1}{\lambda}\Big(\log \mathop{\mathbb{E}}_{h\sim P}[e^{\lambda\phi(h)}] + \mathsf{KL}(Q\|P)\Big)$$

Theorem (Change of Measure Inequality)

For any distributions P, Q over $\mathcal{H}$ and function $\phi : \mathcal{H} \to \mathbb{R}$:

$$\mathop{\mathbb{E}}_{h\sim Q}[\phi(h)] \leq \frac{1}{\lambda}\Big(KL(Q\|P) + \log \mathop{\mathbb{E}}_{h\sim P} e^{\lambda\phi(h)}\Big)$$

Theorem (PAC-Bayesian generalization bound [McAllester, 1999])

ℓ bounded in $[0,1]$. P independent of $\mathcal{D}$.
With prob $1-\delta$ over $\mathcal{D} \overset{i.i.d.}{\sim} p^{\otimes n}$, it holds for all distributions Q:

$$\mathcal{R}(Q) - \hat{\mathcal{R}}(Q) \leq \frac{1}{\sqrt{n}}\Big(KL(Q\|P) + \frac{1}{8} + \log\frac{1}{\delta}\Big)$$

Proof sketch.

- Change of measure inequality:

$$\mathop{\mathbb{E}}_{h\sim Q}[\phi(h)] \leq \frac{1}{\lambda}\Big(\mathsf{KL}(Q\|P) + \log \mathop{\mathbb{E}}_{h\sim P} e^{\lambda\phi(h)}\Big)$$

- apply with prior P, posterior Q and $\phi(h) = \mathcal{R}(h) - \hat{\mathcal{R}}(h)$:

$$\mathcal{R}(Q) - \hat{\mathcal{R}}(Q) \leq \frac{1}{\lambda}\Big(\mathsf{KL}(Q\|P) + \log \mathop{\mathbb{E}}_{h\sim P} e^{\lambda[\mathcal{R}(h)-\hat{\mathcal{R}}(h)]}\Big)$$

- P and ϕ are independent (in contrast to Q), so with prob. $\geq 1-\delta$

$$\log \mathop{\mathbb{E}}_{h\sim P} e^{\lambda[\mathcal{R}(h)-\hat{\mathcal{R}}(h)]} \overset{\text{Hoeffing's lemma, Markov ineq.}}{\leq} \frac{\lambda^2 n}{8} + \log(1/\delta)$$

- theorem follows by setting $\lambda = \frac{1}{\sqrt{n}}$.

□

Example: reproving a bound for finite hypothesis sets

- $\mathcal{H} = \{h_1, \ldots, h_T\}$ finite
- $P(h) = (\frac{1}{T}, \ldots, \frac{1}{T})$ uniform distribution
- $Q(h) = \delta_{h=h_k}(h)$ indicator on one hypothesis (can depend on $\mathcal{D}$)
- $\mathsf{KL}(Q\|P) = \sum_t Q(t) \log \frac{Q(t)}{P(t)} = \log \frac{1}{P(h_k)} = \log T$

Example: reproving a bound for finite hypothesis sets

- $\mathcal{H} = \{h_1, \ldots, h_T\}$ finite
- $P(h) = (\frac{1}{T}, \ldots, \frac{1}{T})$ uniform distribution
- $Q(h) = \delta_{h=h_k}(h)$ indicator on one hypothesis (can depend on $\mathcal{D}$)
- $\mathsf{KL}(Q\|P) = \sum_t Q(t) \log \frac{Q(t)}{P(t)} = \log \frac{1}{P(h_k)} = \log T$

The PAC-Bayesian statement for Gibbs classifiers:

$$\text{For every dist. } Q: \qquad \mathcal{R}(Q) \leq \hat{\mathcal{R}}(Q) + \frac{1}{\sqrt{n}}\Big(\mathsf{KL}(Q\|P) + \frac{1}{8} + \log\frac{1}{\delta}\Big)$$

translates into a bound for a ordinary (deterministic) classifiers:

$$\text{For every } h \in \mathcal{H}: \qquad \mathcal{R}(h) \leq \hat{\mathcal{R}}(h) + \frac{1}{\sqrt{n}}\Big(\log T + \frac{1}{8} + \log\frac{1}{\delta}\Big)$$

which is similar to the previous bound for finite hypotheses sets.

Example: weighted finite hypothesis set bound

New feature: we can freely chose the prior, it does not have to be uniform.

- $\mathcal{H} = \{h_1, \ldots, h_T\}$ finite (or countable infinite)
- $P(h) = (\pi_1, \ldots, \pi_T)$ arbitrary prior distribution (fixed before seeing $\mathcal{D}$)
- $Q(h) = \delta_{h=h_k}(h)$ indicator on one hypothesis (can depend on $\mathcal{D}$)
- $\mathrm{KL}(Q\|P) = \sum_t Q(t) \log \frac{Q(t)}{P(t)} = \log \frac{1}{\pi_k}$

For every $h_k \in \mathcal{H}$:

$$\mathcal{R}(h_k) \leq \hat{\mathcal{R}}(h_k) + \frac{1}{\sqrt{n}}\Big(\log \frac{1}{\pi_k} + \frac{1}{8} + \log \frac{1}{\delta}\Big)$$

Tighter bound, if well-working hypotheses are (a priori) more likely.

Popular example: "Occam razor bound"

- $P(h) \propto$ *"simplicity"*(h), e.g. length of an encoding

Example: justifying L^2-regularization

- $\mathcal{H} = \{h_w(x) : \mathcal{X} \to \mathcal{Y},\ w \in \mathbb{R}^d\}$ parameterized by $w \in \mathbb{R}^d$
- $P(w) \propto e^{-\lambda \|w\|^2}$ prior: Gaussian around 0
- $Q(w) \propto e^{-\lambda \|w-v\|^2}$ posterior: Gaussian around v
- $\mathrm{KL}(Q\|P) = \lambda \|v\|^2$

$$\mathcal{R}(Q) \leq \hat{\mathcal{R}}(Q) + \frac{1}{\sqrt{n}}\Big(\lambda \|v\|^2 + \frac{1}{8} + \log \frac{1}{\delta}\Big)$$

- most promising classifier: minimize right hand side w.r.t v
  $\to$ "regularizer" $\|v\|^2$ appears naturally in the objective

Example: justifying L^2-regularization

- $\mathcal{H} = \{h_w(x) : \mathcal{X} \to \mathcal{Y},\ w \in \mathbb{R}^d\}$ parameterized by $w \in \mathbb{R}^d$
- $P(w) \propto e^{-\lambda \|w\|^2}$ prior: Gaussian around 0
- $Q(w) \propto e^{-\lambda \|w-v\|^2}$ posterior: Gaussian around v
- $\mathrm{KL}(Q\|P) = \lambda \|v\|^2$

$$\mathcal{R}(Q) \leq \hat{\mathcal{R}}(Q) + \frac{1}{\sqrt{n}}\Big(\lambda \|v\|^2 + \frac{1}{8} + \log \frac{1}{\delta}\Big)$$

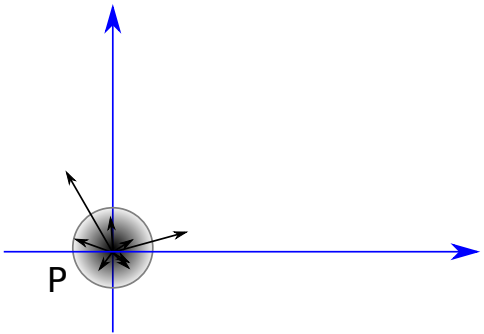
- most promising classifier: minimize right hand side w.r.t v
  $\to$ "regularizer" $\|v\|^2$ appears naturally in the objective

Caveat: $\|\cdot\|^2$ appears because we put it into the exponents of P and Q. Other distributions (which are our choice) yield other bounds/regularizers.

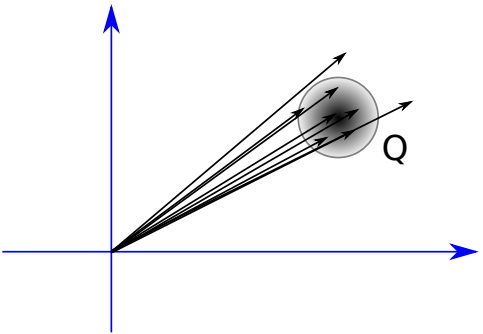
"PAC-Bayes is a bound-generation machine."

Example: SVM-style bound

- $\mathcal{H} = \{h(x) = \text{sign}\langle w, x\rangle,\ w \in \mathbb{R}^d\}$ linear classifiers
- $P(w) \propto e^{-\|w\|^2}$ prior: Gaussian around 0
- $Q(w) \propto e^{-\|w-v\|^2}$ posterior: Gaussian around v



prior: uniform w.r.t. direction



posterior: not uniform, some preferred directions

Example: SVM-style bound

- $\mathcal{H} = \{h(x) = \operatorname{sign}\langle w, x\rangle,\ w \in \mathbb{R}^d\}$ linear classifiers
- $P(w) \propto e^{-\|w\|^2}$ prior: Gaussian around 0
- $Q(w) \propto e^{-\|w-v\|^2}$ posterior shifted by v (non-uniform)

$$\mathcal{R}(Q) \le \hat{\mathcal{R}}(Q) + \frac{1}{\sqrt{n}}\Big(\|v\|^2 + \frac{1}{8} + \log\frac{1}{\delta}\Big)$$

Example: SVM-style bound

- $\mathcal{H} = \{h(x) = \operatorname{sign}\langle w, x\rangle,\ w \in \mathbb{R}^d\}$ linear classifiers
- $P(w) \propto e^{-\|w\|^2}$ prior: Gaussian around 0
- $Q(w) \propto e^{-\|w-v\|^2}$ posterior shifted by v (non-uniform)

$$\mathcal{R}(Q) \le \hat{\mathcal{R}}(Q) + \frac{1}{\sqrt{n}}\Big(\|v\|^2 + \frac{1}{8} + \log\frac{1}{\delta}\Big)$$

When ℓ is 0-1 loss:

- deterministic classifier $\operatorname{sign}\langle v, x\rangle$ is identical to majority vote of Q
- we can relate $\hat{\mathcal{R}}(Q)$ to $\hat{\mathcal{R}}(v)$:

$$\hat{\mathcal{R}}(Q) = \frac{1}{n}\sum_{i=1}^{n} \bar{\Phi}\Big(\frac{y_i\langle v, x_i\rangle}{\|x_i\|}\Big) \text{ for } \bar{\Phi}(t) = \frac{1}{2}\Big(1 - \operatorname{erf}\Big(\frac{t}{\sqrt{2}}\Big)\Big),$$

Together:

$$\frac{1}{2}\mathcal{R}(v) \le \frac{1}{n}\sum_{i=1}^{n} \bar{\Phi}\Big(\frac{y_i\langle v, x_i\rangle}{\|x_i\|}\Big) + \frac{1}{\sqrt{n}}\|v\|^2 + \frac{\frac{1}{8} + \log\frac{1}{\delta}}{\sqrt{n}}$$

Example: Transfer bound

- $\mathcal{H} = \{h_w(x) : \mathcal{X} \to \mathcal{Y},\ w \in \mathbb{R}^d\}$ parameterized by $w \in \mathbb{R}^d$
- $P(w) \propto e^{-\|w - v_0\|^2}$ prior: Gaussian around v_0
- $Q(w) \propto e^{-\|w - v\|^2}$ posterior: Gaussian around v
- $\mathrm{KL}(Q\|P) = \|v - v_0\|^2$

$$\mathcal{R}(Q) \leq \hat{\mathcal{R}}(Q) + \frac{1}{\sqrt{n}}\Big(\|v - v_0\|^2 + \frac{1}{8} + \log\frac{1}{\delta}\Big)$$

Typical situation for fine-tuning:

- inititalize classifier parameters as v_0
- train on $\mathcal{D}$ using (stochastic) gradient descent

Good generalization guarantees, if parameters stay close to initialization.

- "dropout rate" $\alpha \in [0, 1]$
- set of posterior distributions: $Q_{\theta,\alpha}$:

$$\text{for each weight:} \quad w_i = \begin{cases} 0 & \text{with prob. } \alpha \\ \theta_i + \epsilon_i & \text{otherwise, for } \epsilon_i \sim \mathcal{N}(0, 1) \end{cases}$$

- prior distribution: $P = Q_{0,\alpha}$
- $\mathrm{KL}(Q\|P) = \frac{1-\alpha}{2}\|\theta\|^2$

Zero-ing out weights reduces complexity by factor $\frac{1-\alpha}{2}$:

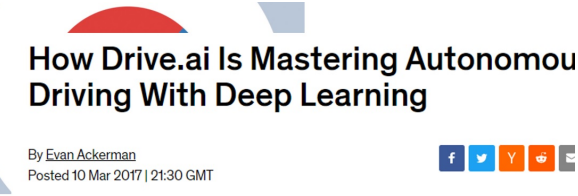
$$\mathcal{R}(Q_{\theta,\alpha}) \leq \hat{\mathcal{R}}(Q_{\theta,\alpha}) + \frac{1}{\sqrt{n}}\Big(\frac{1-\alpha}{2}\|\theta\|^2 + \frac{1}{8} + \log\frac{1}{\delta}\Big)$$

Training: optimize $\hat{\mathcal{R}}(Q_{\theta,\alpha}) + \ldots$ via SGD $\rightarrow$ "dropout training"

Prediction: majority vote over many stochastic networks

(Deep) Neural Networks

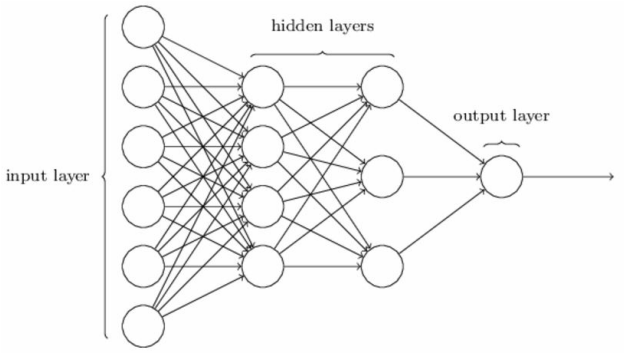
Here's is the Hype



- **Deep Learning** is name used since the mid 2000s for machine learning when the hypothesis set consists of **deep neural networks**.
- **Deep neural networks** are **artificial neural networks** with "many" layers (e.g. ≥ 5).
- **Artificial neural networks** are predictive models inspired by (early) Neuroscience.

Main idea:

- build a complex function out of simple units ("neurons")
- arrange neurons in layers
- any layer's outputs are the next layer's input



Observation:
Despite the current hype on the deep learning (or even "artificial intelligence") revolution, neural networks algorithms are far from a new concept.

This is already the **third time** that neural networks were popular:

- 1940s–1960s: biological inspired learning is proposed, single-neuron models are trained
- 1980s–1990s: neural networks with a couple of hidden layers are trained by means of backpropagation, first systems doing useful tasks
- 2006–now: current wave of research, really taking off since 2012

Mark I Perceptron

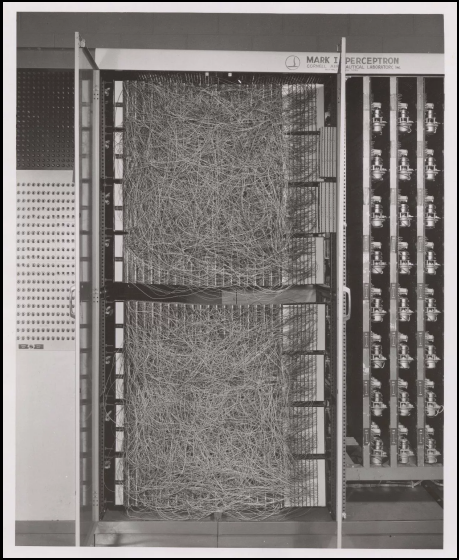


Photo: Cornell University Library

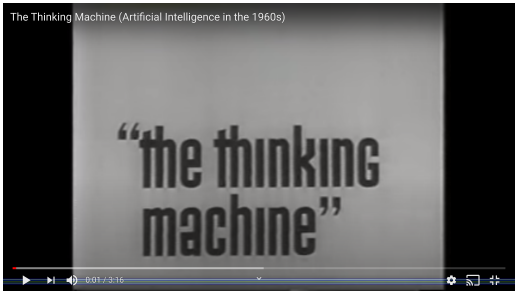
NEW NAVY DEVICE LEARNS BY DOING

Psychologist Shows Embryo of Computer Designed to Read and Grow Wiser

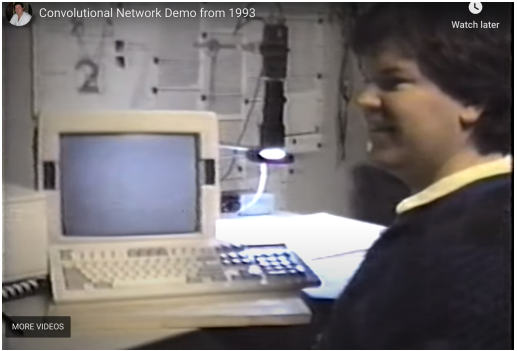
WASHINGTON, July 7 (UPI) —The Navy revealed the embryo of an electronic computer today that it expects will be able to walk, talk, see, write, reproduce itself and be conscious of its existence.

Source: New York Times, 7/7/1958

More Fun Examples



https://youtu.be/aygSMgK3BEM



https://youtu.be/FwFduRA_L6Q

What's different now than it was before?

Today, NNs really do work well, often better than other methods.

This is due to a few complementary factors:

- large labeled datasets
  - digitalization made data readable for computers
  - the Internet made large amounts of data, e.g. images, publicly and freely available
  - crowd-sourcing, e.g. Amazon MTurk, allows collecting large amounts of annotation
- more computational power
  - graphics cards (GPUs) were originally developed exclusively for computer games
  - today, they are heavily used for AI, in particular deep learning
    - e.g., this year, ≈50% of NVIDIA revenue came from data centers
- some methodological progress, as well
  - ReLU activation function
  - batch normalization
  - generative adversarial networks
  - transformer networks

Dataset Sizes

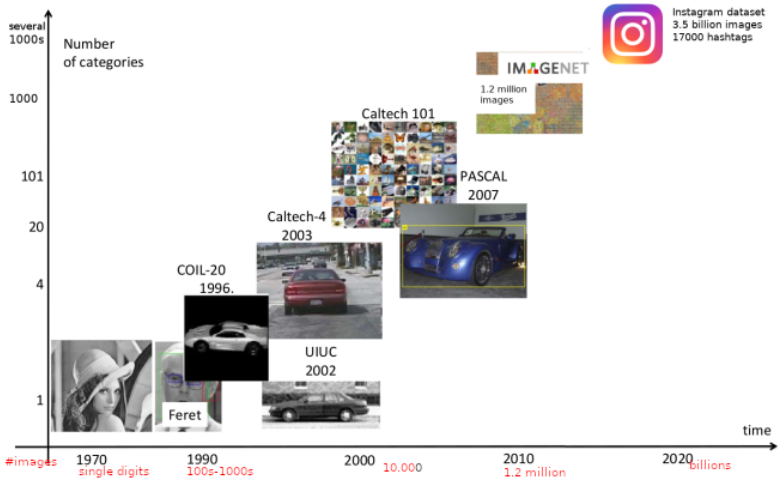


Figure: Size of publicly available datasets has grown tremendously over time.

Number of Neural Network Layers

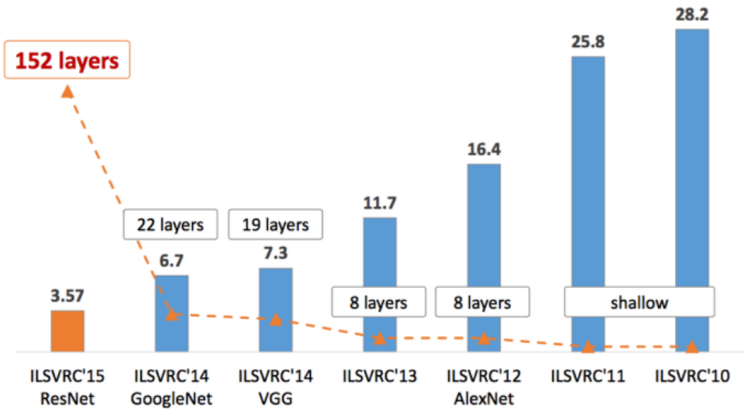


Figure: Size and complexity of models (e.g. number of layers) over time.

Computational Resources

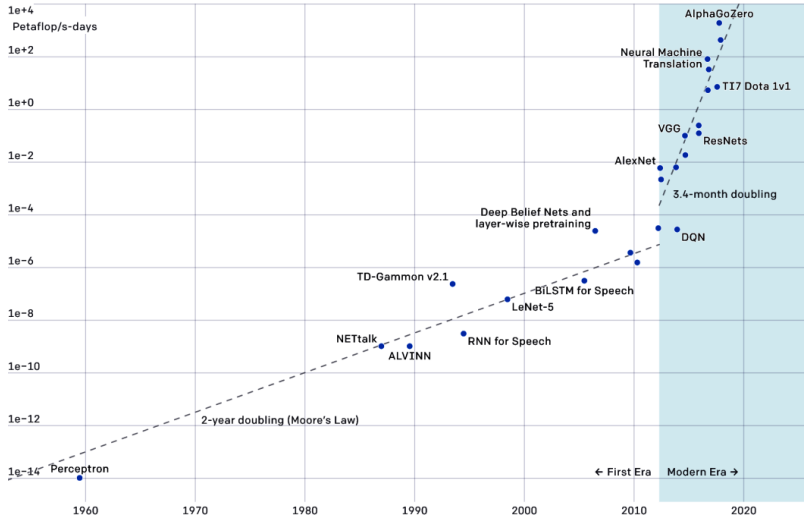


Figure: Amount of compute operations used to train machine learning models. Image: OpenAI

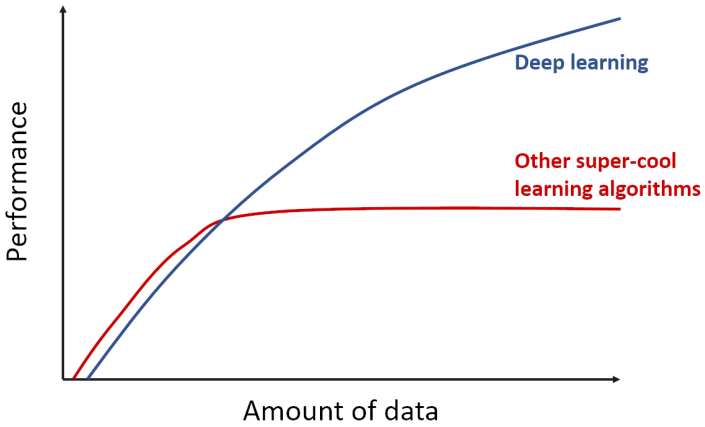
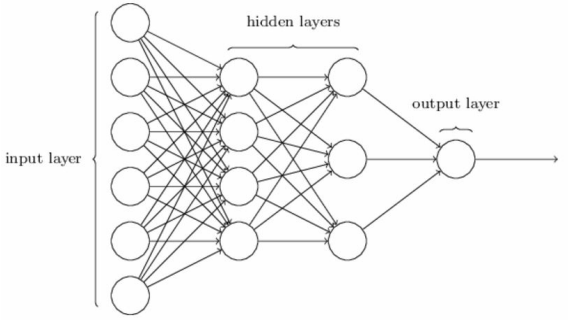


Figure: Andrew Ng. "What data scientists should know about deep learning".



Notation:

- inputs: $x \in \mathcal{X} = \mathbb{R}^d$, outputs: $y \in \mathcal{Y}$, e.g. $\mathcal{Y} = \{1, \ldots, K\}$, or $\mathcal{Y} = \mathbb{R}^K$.
- neural networks consist of layers,
  - first layer has original x as input: input layer,
  - all other layers have output of previous layer as input,
  - last layer has prediction $h(x)$ as output: output layer,
  - layers that are neither input nor output are called hidden layers,

Each such neural network architecture parametrized a set of functions, $h : \mathcal{X} \to \mathbb{R}^K$

- each layer, l computes an output $h^{(l)}(v)$ from its input v, where

$$h^{(l)}(v) = \sigma_l(W_l v + b_l) \qquad \text{for } l = 1, \ldots, L$$

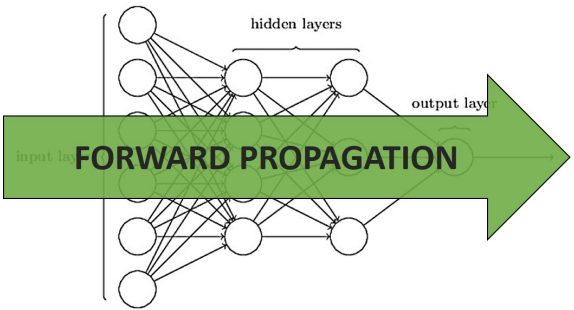
  - W_l is a weight matrix of size (number of layer outputs) $\times$ (number of layer inputs),
  - b_l is a vector of bias terms (as many elements as layer has outputs),
  - σ_l is a non-linear function, called activation function, that is applied componentwise.
    - typically σ_l is the same for all neurons and all layers, except probably the output layer

Overall:

$$h(x) = h^{(L)}(h^{(L-1)}(\ldots h^{(2)}(h^{(1)}(x)))$$

- h is parametrized by $\theta = (W_1, b_1, \ldots, W_L, b_L)$
- the non-linearities, σ_l, usually have no free parameters to learn (but exceptions exist)

The process of computing the network output given its input is also called **forward propagation**.



Forward propagation just means evaluating the definition of f step-by-step:

$$h(x) = h^{(L)}(h^{(L-1)}(\dots h^{(2)}(h^{(1)}(x))))$$

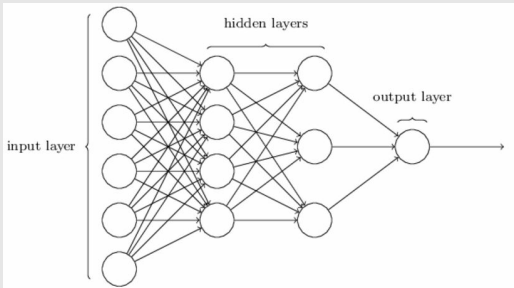
Example

The 4-layer network from the picture encodes the function:

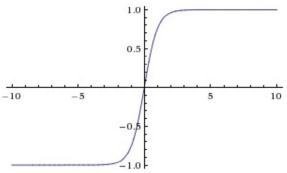
$$h(x) = b_3 + W_3\sigma(b_2 + W_2\sigma(b_1 + W_1 x)) \tag{1}$$

where we have integrate

- σ is the activation function
- $x \in \mathbb{R}^6$ is the input
- $W_1 \in \mathbb{R}^{4\times 6}$ and $b_1 \in \mathbb{R}^4$ are the weight matrix and bias vector of the first layer
- $W_2 \in \mathbb{R}^{3\times 4}$ and $b_2 \in \mathbb{R}^3$ are the weight matrix and bias vector of the second layer
- $W_3 \in \mathbb{R}^{1\times 3}$ are $b_3 \in \mathbb{R}$ are the weight matrix and bias vector of the third layer



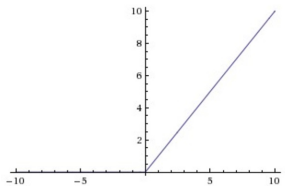
Total number of parameters: $24 + 4 + 12 + 3 + 3 + 1 = 47$



Tanh activation

tanh is a symmetric sigmoid function: $\tanh(t) = \dfrac{e^t - e^{-t}}{e^t + e^{-t}}$.

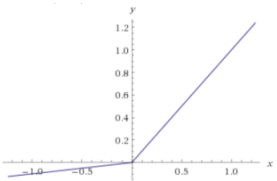
- most popular activation function from classic era of neural networks
- symmetric, differentiable
- costly to implement (several evaluations of trigonometric)
- value and gradient saturate for $t \to \pm\infty$



ReLu activation

ReLU stands for *Rectified Linear Unit*, $\text{ReLU}(t) = \mathbf{max}(0, t)$

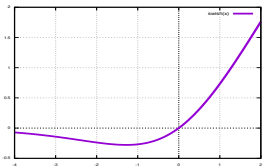
- most popular activation function from deep learning era
- not differentiable, not symmetric, not saturating
- very efficient to implement
- observed to result in networks that are easier to train than, e.g., with $\tanh$



leaky ReLU activation

leaky ReLU is a generalization of ReLU,
$\text{LReLU}(t) = \mathbf{max}(0, t) + \alpha\,\mathbf{min}(0, t)$ for small $\alpha > 0$.

- not differentiable, not symmetric, not saturating
- still very efficient to implement
- avoids problem that ReLU is constant 0 for negative inputs



Swish activation

swish is a "soft" alternative to ReLU: $\text{swish}(t) = \dfrac{t}{1 + e^{-\beta t}}$

- recent competitor to ReLU
- differentiable, not symmetric, not monotonic
- often $\beta = 1$
- β interpolates between linear ($\beta = 0$) and ReLU ($\beta \to \infty$)

... and many more...

Why using non-linear activations at all?

Neural network function (ignoring bias vectors):

$$h(x) = W_L \sigma(\dots \sigma(W_2 \sigma(W_1\, x))$$

Without σ, we'd have

$$h(x) = W_L\, W_{L-1}\, \dots W_2\, W_1\, x \qquad = \tilde{W} x \quad \text{for } \tilde{W} = W_L W_{L-1} \cdots W_2 W_1$$

so $h(x)$ would simply be a linear function, parametrized in a very wasteful way.

(analogously, if σ is linear or affine itself)

Note: linear activation functions are sometimes used as simplifying assumptions in NN theory, $\rightarrow$ "linear networks"

How deep should my network be (i.e. how many layers)?

- Mathematically, two-layer networks are enough to represent any target function.

Theorem (Universal approximation)

For any continuous function, $g : \mathcal{X} \to \mathbb{R}$, and any $\epsilon > 0$, there is a two-layer neural network, f, that approximates g up to precision ϵ in L^∞-norm.

- Practically, such networks would have a huge number of neurons.
- Deeper network allow building complex functions with overall fewer neurons.
- But: deeper network take longer to evaluate

How wide should my network be (i.e. how many neurons in each layer)?

- Wider networks have higher capacity, they can represent more functions.
- Wider networks are easier to train.
- But: wider networks need more memory and computation

"As deep and wide as the available resources allow."

Training (Deep) Neural Networks

Training a deep network for classification typically looks like training a generalized linear model in which the feature map that is also parameterized and learned:

Generalized Linear Model	**Neural Network**
$f_\theta(x) = W\phi(x)$	$f_\theta(x) = W_L\phi(x)$ with $\phi(x) = \sigma(W_{L-1}\sigma(\dots\sigma(W_1x)\,)\,)$
$\theta = W$	$\theta = (W_1, \dots, W_L)$

Training a deep network for classification typically looks like training a generalized linear model in which the feature map that is also parameterized and learned:

Generalized Linear Model	Neural Network
$f_\theta(x) = W\phi(x)$	$f_\theta(x) = W_L\phi(x)$ with $\phi(x) = \sigma(W_{L-1}\sigma(\dots\sigma(W_1 x)))$
$\theta = W$	$\theta = (W_1, \dots, W_L)$

Parameters are learned by (surrogate) risk minimization: $\min_\theta \quad \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}(\, y_i, f_\theta(x_i)\,)$

- **binary classification** $\mathcal{L}(y_i, f_\theta(x_i)) = -\log(1 + e^{-y_i f_\theta(x_i)})$ "log-loss"
- **multi-class classification** $\mathcal{L}(y_i, f_\theta(x_i)) = -\log \frac{e^{f_\theta(x)[y]}}{\sum_{k=1}^{K} e^{f_\theta(x)[k]}}$ "cross-entropy" / "soft-max" loss
- **regression** $\mathcal{L}(y_i, f_\theta(x_i)) = (y_i - f_\theta(x_i)^2$ "squared loss"

In contrast to linear models, the resulting optimization problems are non-convex!

(Non-convex) Numeric Optimization

Numeric Optimization

Numeric optimization of a differentiable function, F, is a rather well understood field.
E.g., the **gradient descent** method will usually converge to a locally optimal solution!

(Steepest) Gradient Descent Minimization

input $\alpha > 0$, step size (=learning rate), $\epsilon > 0$, tolerance (for stopping criterion)

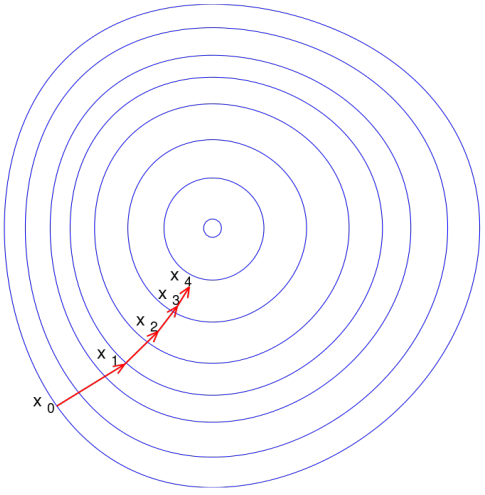
1: initialize θ
2: **repeat**
3: $\quad v \leftarrow \nabla_\theta F(\theta)$
4: $\quad \theta \leftarrow \theta - \alpha v$
5: **until** $\|v\| < \epsilon$

output $\theta \in \mathbb{R}^d$ learned parameter vector

Many variants, to increase generality or efficiency. Some we'll discuss later today:

- stochastic gradient descent
- non-differentiable objectives
- changing stepsize over time (manually or automatically)
- faster convergence through momentum

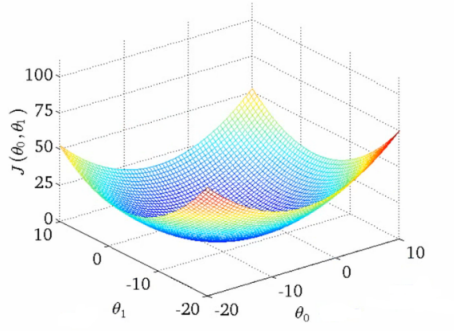
Gradient descent searches a minimum of a differentiable function by iterative steps in the opposite direction of the gradient of the function.



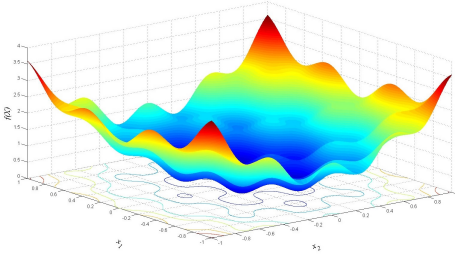
Gradient descent on a series of level sets

If the objective function is convex, e.g. linear logistic regression, gradient descent converges to a **global minimum** (in fact, it still converges to a local minimum, but all local minima are actual global minima)

For neural networks, the objective function is non-convex, so gradient descent might only find a **local minimum**.



Convex Function



Non-Convex Function

In ML, the function we want to minimize is often a sum over many training examples:

$$\min_{\theta \in \mathbb{R}^d} F(\theta) \quad \text{for} \quad F(\theta) = \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(\, y_i, f_\theta(x_i)\,)$$

Every computation of the gradient of F needs at least like $O(nd)$ operations:

- d is the dimensionality of the parameters
- n is the number of training examples.

Both d and n can be big (millions). How to speed this up?

- we'll not get rid of $O(d)$, if we want to change $\theta \in \mathbb{R}^d$,
- but we can get rid of the scaling with $O(n)$ for each update!

$$F(\theta) = \frac{1}{n}\sum_{i=1}^{n} f_i(\theta), \qquad \text{for differentiable functions } f_1, \ldots, f_n.$$

Stochastic Gradient Descent (SGD)

input step sizes $\alpha_1, \alpha_2, \ldots$
input number of iterations, T
1: initialize θ_0
2: **for** $t = 1, \ldots, T$ **do**
3: $\quad i \leftarrow$ random index in $1, 2, \ldots, n$
4: $\quad v \leftarrow \nabla f_i(\theta_{t-1})$
5: $\quad \theta_t \leftarrow \theta_{t-1} - \alpha_t v$
6: **end for**
output θ_T, or average $\frac{1}{T-T_0}\sum_{t=T_0}^{T} \theta_t$

- Time for each iteration is independent of n
- Gradient is "wrong" is each step, but correct in expectation.
- Objective does not decrease in every step,
- In practice, one typically does not pick a random i in each step, but creates a random permutation of indices and goes through it sequentially.
- Each pass through the training set is called an epoch.

$$F(\theta) = \frac{1}{n} \sum_{i=1}^{n} f_i(\theta), \qquad \text{for differentiable functions } f_1, \ldots, f_n.$$

Minibatch SGD

input step sizes $\alpha_1, \alpha_2, \ldots$
input number of iterations, T
input batchsize B
1: initialize θ_0
2: **for** $t = 1, \ldots, T$ **do**
3: $\quad i_1, \ldots, i_B \leftarrow B$ random indices
4: $\quad v \leftarrow \frac{1}{B} \sum_{j=1}^{B} \nabla f_{i_j}(\theta_{t-1})$
5: $\quad \theta_t \leftarrow \theta_{t-1} - \alpha_t v$
6: **end for**
output θ_T, or average $\frac{1}{T-T_0} \sum_{t=T_0}^{T} \theta_t$

- Time for each iteration is proportional to B
- Variance of gradient estimate is reduced by $\frac{1}{B}$
- Optimal batchsize is problem dependent
- The computation of v can be performed in a parallel/distributed way.

In practice, one rarely uses the procedure described above (so called vanilla SGD).

Rather, additional tricks are added, resulting in a number of popular optimizers,e.g.

- momentum
- non-uniform step size: AdaGrad, RMSProp, Adam
- both

Not popular: second order optimization e.g. Newton

In vanilla gradient descent, the update is a negative multiple of the current gradient:

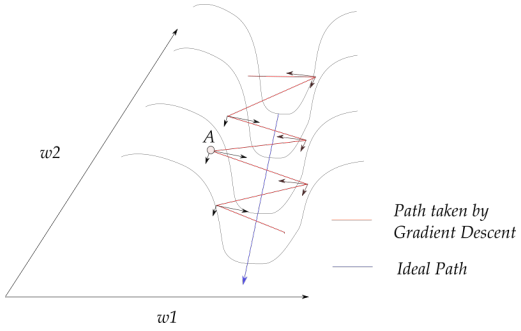
$$v_t \leftarrow \alpha_t \nabla f_i(\theta_{t-1})$$

(Stochastic) Gradient Descent with Momentum

In gradient descent with momentum, part of the previous update direction is preserved for the next step:

$$v_t \leftarrow \eta v_{t-1} + \nabla f_i(\theta_{t-1})$$

η is a decay factor, e.g. $\eta = 0.9$



Main idea: directions that appear consistently in updates get amplified, inconsistent directions do not. This can lead to substantial speedups, especially if the objective has "narrow valleys".