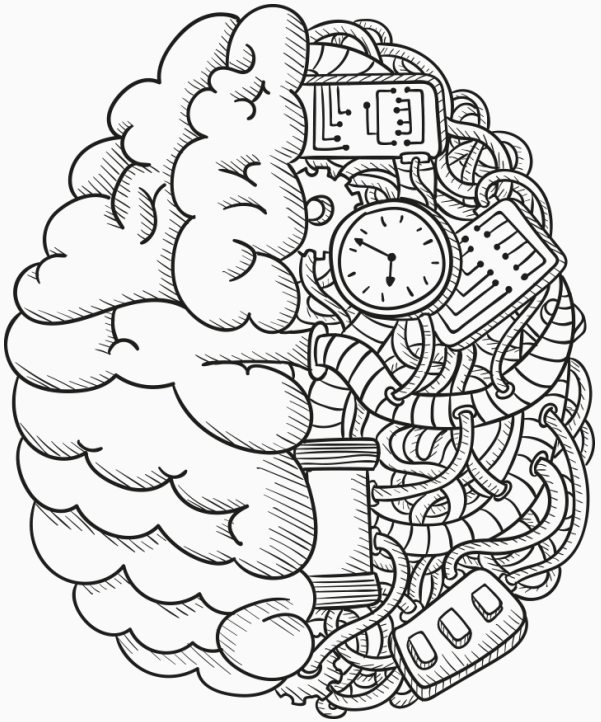
ECE 285

Machine Learning for Image Processing

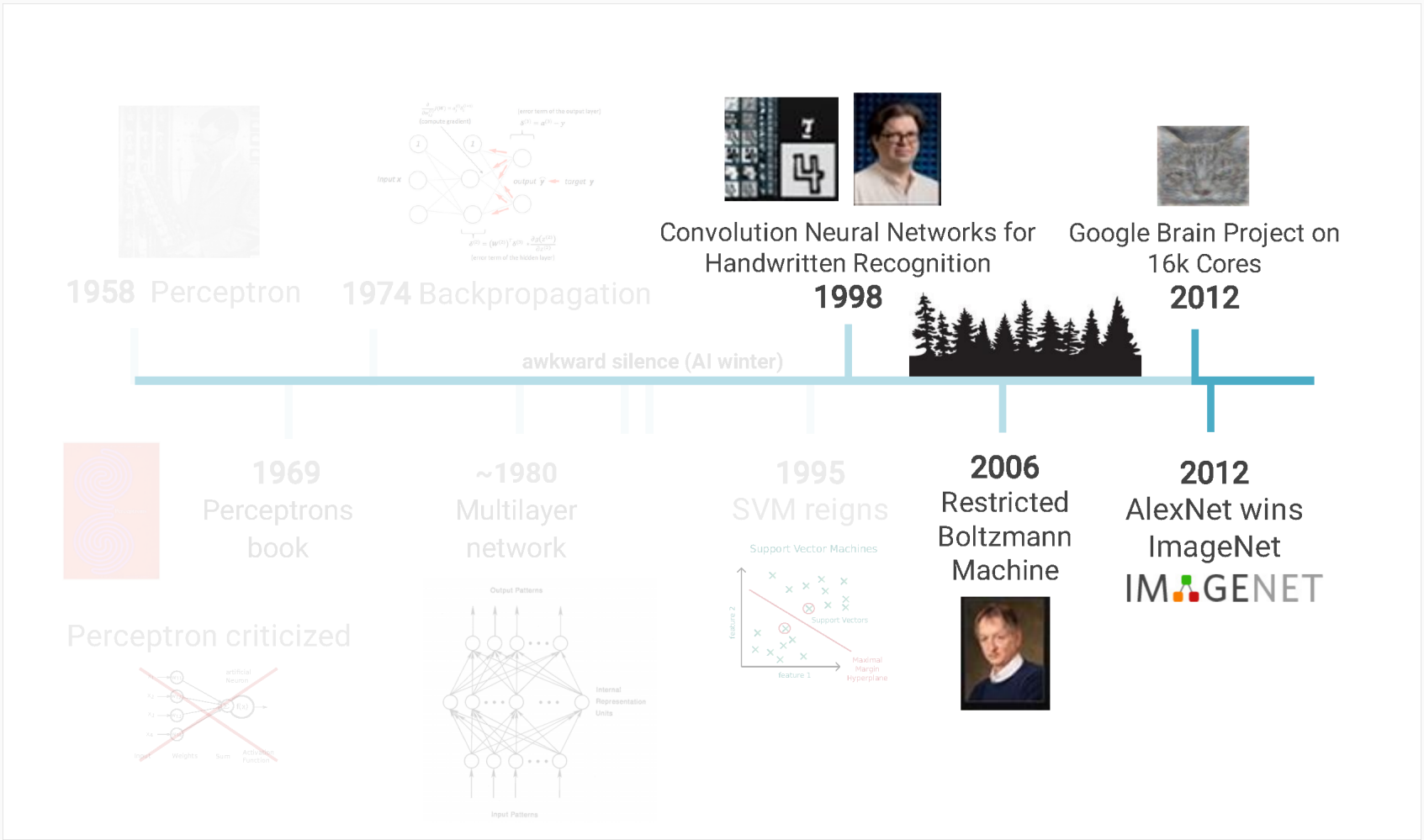
Chapter III – Introduction to deep learning

Charles Deledalle

September 23, 2019



Deep learning



(Source: Lucas Masuch & Vincent Lepetit)

Brief history

- Popularized by Hinton in 2006 with Restricted Boltzmann Machines

Geoffrey Hinton: University of Toronto & Google

- Developed by different actors:

Yann LeCun: New York University & Facebook

Andrew Ng: Stanford & Baidu

Yoshua Bengio: University of Montreal

Jürgen Schmidhuber: Swiss AI Lab & NNAISENSE

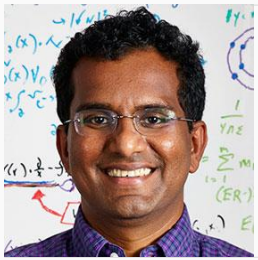
and many others...

Deep learning – Hype or reality?

Quotes

I have worked all my life in Machine Learning, and I've never seen one algorithm knock over benchmarks like Deep Learning
– Andrew Ng (Stanford & Baidu)

Deep Learning is an algorithm which has no theoretical limitations of what it can learn; the more data you give and the more computational time you provide, the better it is *– Geoffrey Hinton (Google)*



Human-level artificial intelligence has the potential to help humanity thrive more than any invention that has come before it *– Dileep George (Co-Founder Vicarious)*

For a very long time it will be a complementary tool that human scientists and human experts can use to help them with the things that humans are not naturally good *– Demis Hassabis (Co-Founder* DeepMind*)*

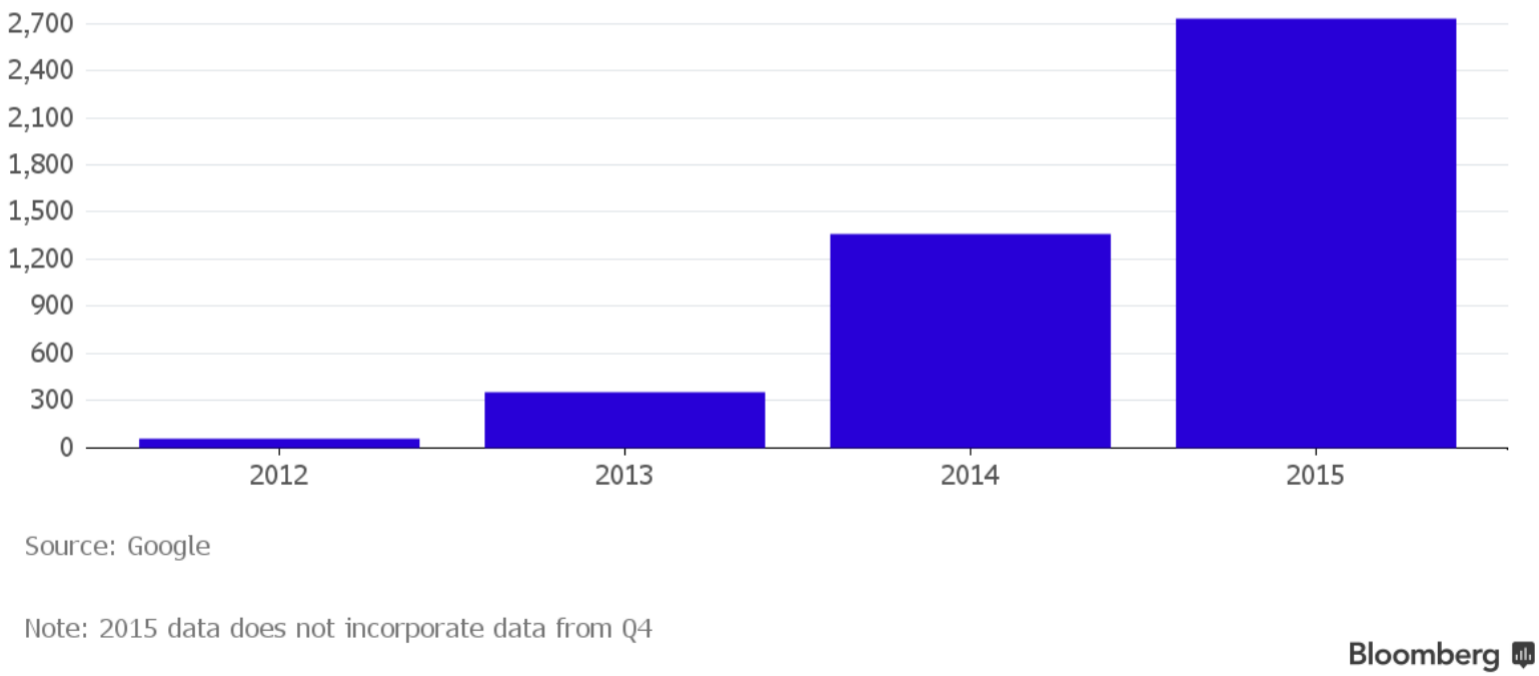
(Source: Lucas Masuch)

Actors and applications

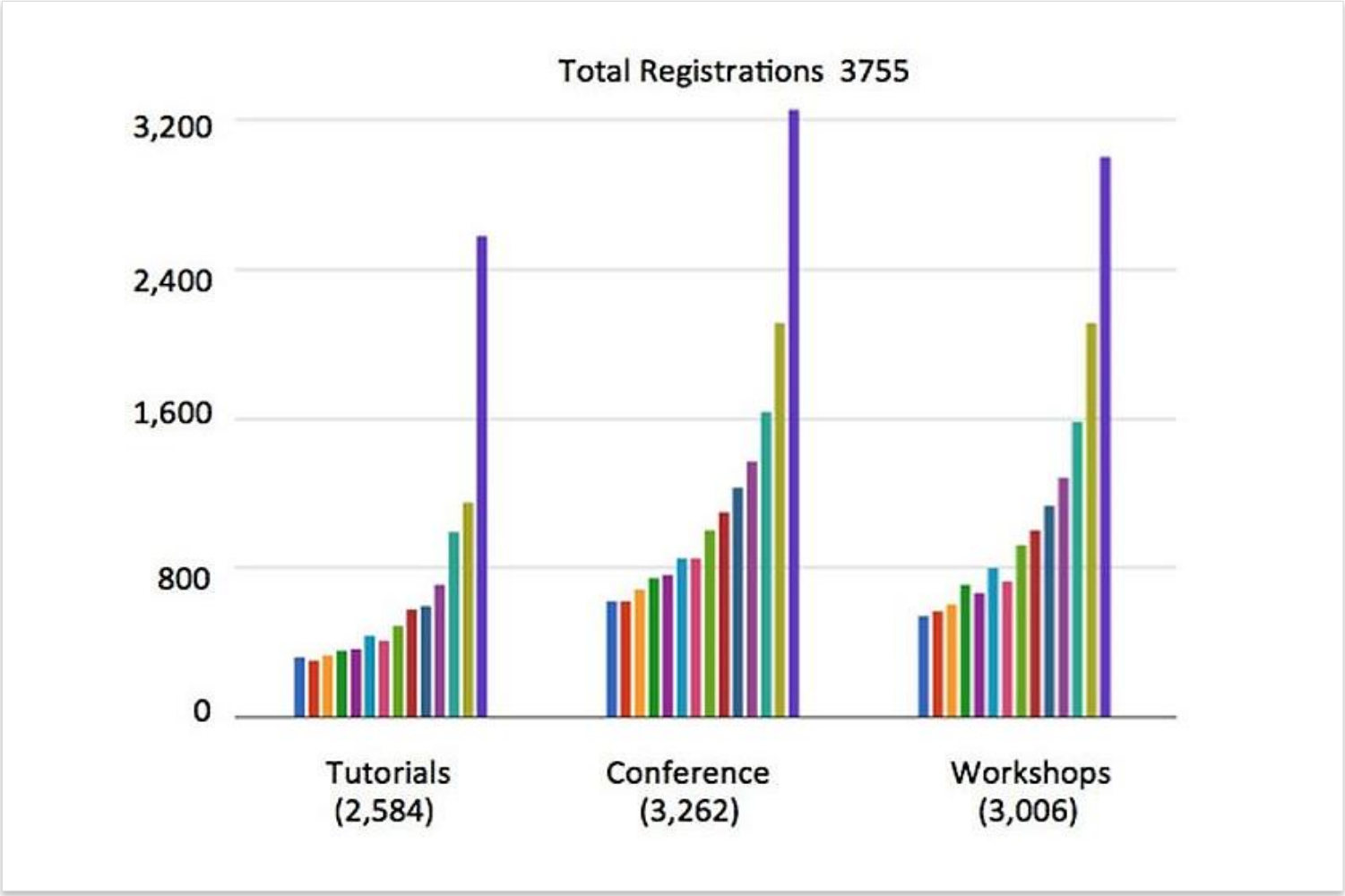
- Very active technology adopted by big actors

- Success story for many different academic problems
  - Image processing
  - Computer vision
  - Speech recognition
  - Natural language processing
  - Translation
  - etc

Deep learning at Google

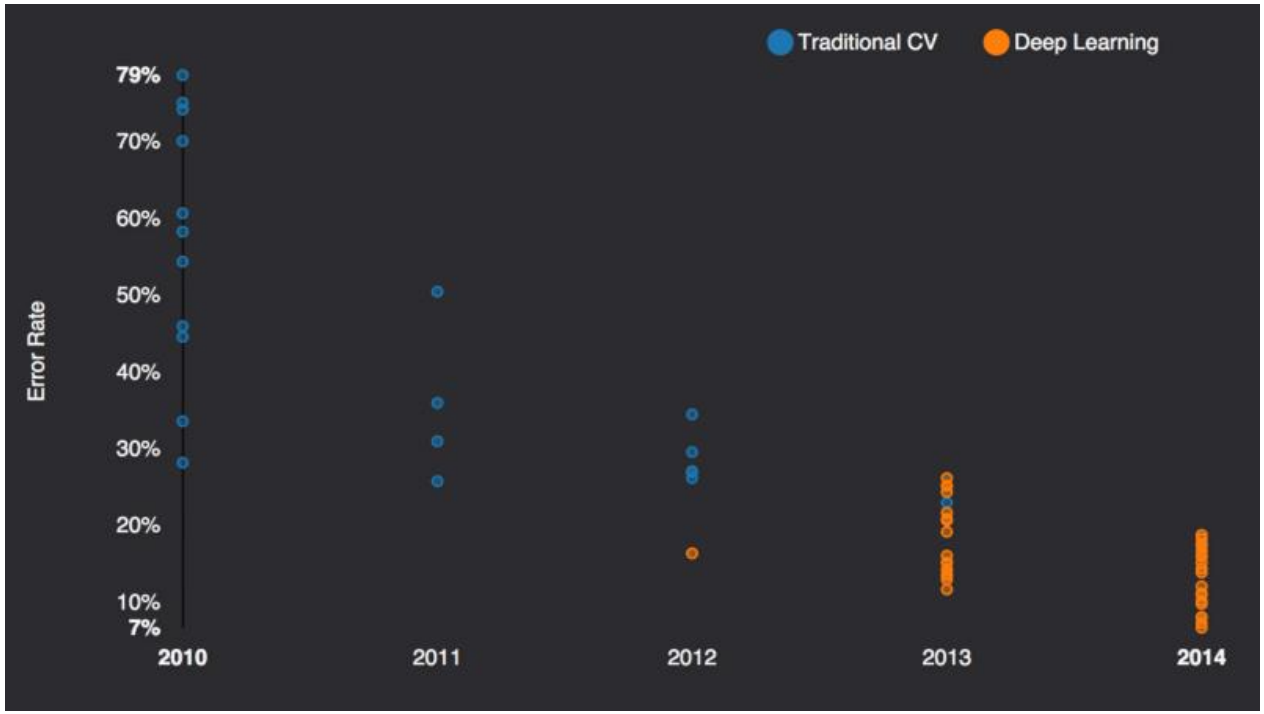


NIPS (Computational Neuroscience Conference) Growth



Success stories

AlexNet wins the ImageNet classification challenge (Krizhevsky, 2012)

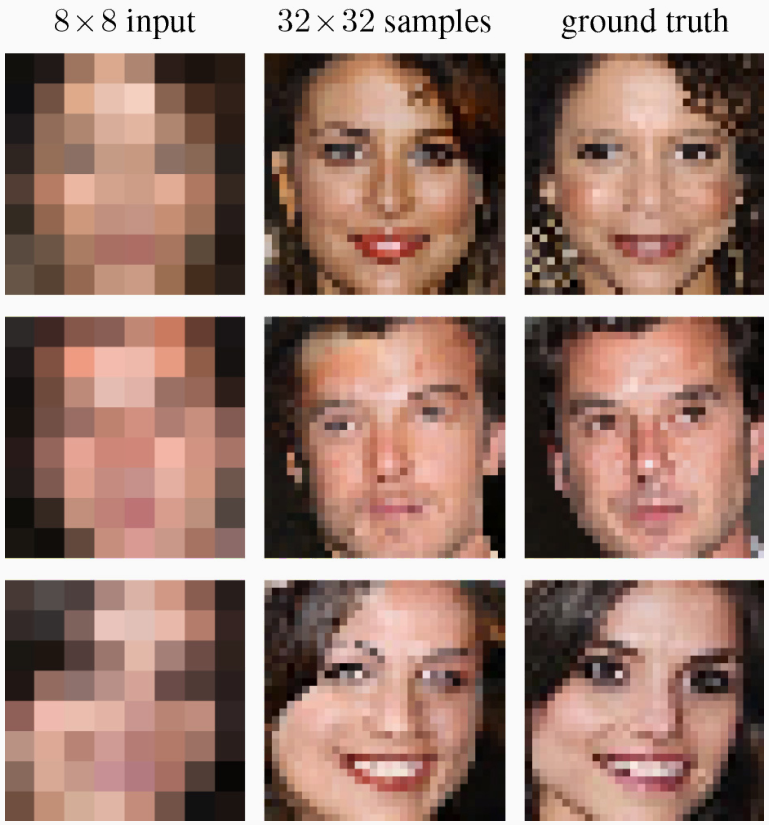


ImageNet: The "computer vision World Cup"

Since, Deep Learning has been the standard and achieved remarkable results.

Success stories

Google Brains's image super-resolution (Dahl *et al.*, 2017).



"Google's neural networks have achieved the dream of CSI viewers", The Guardian.

Success stories

Image captioning (Karpathy, Fei-Fei, CVPR, 2015).


"girl in pink dress is jumping in air."


"black and white dog jumps over bar."


"young girl in pink shirt is swinging on swing."


"man in blue wetsuit is surfing on wave."


"little girl is eating piece of cake."


"baseball player is throwing ball in game."

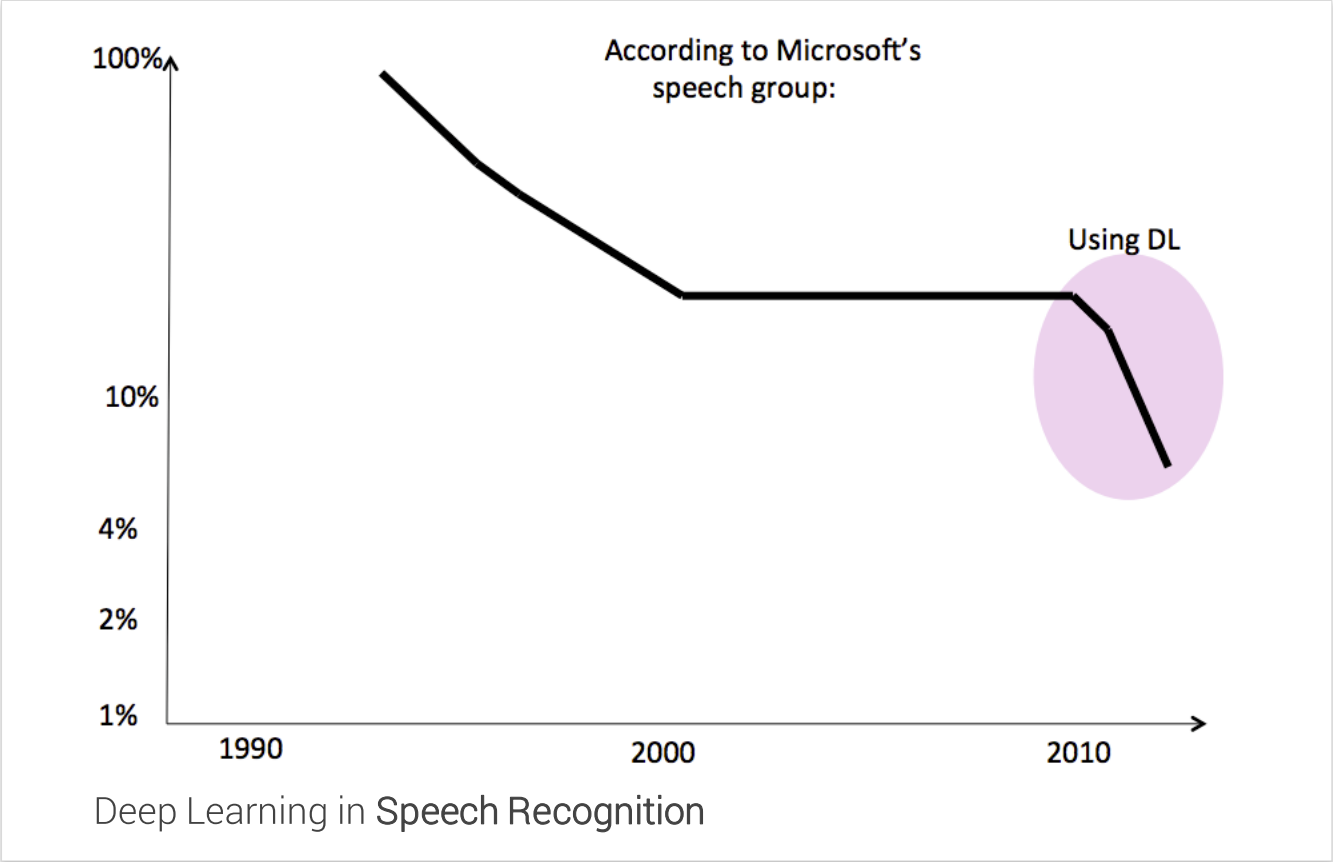

"woman is holding bunch of bananas."


"black cat is sitting on top of suitcase."

"I had never thought AI would arrive at this stage that fast", D. J. Lasiman.

Success stories

Speech recognition

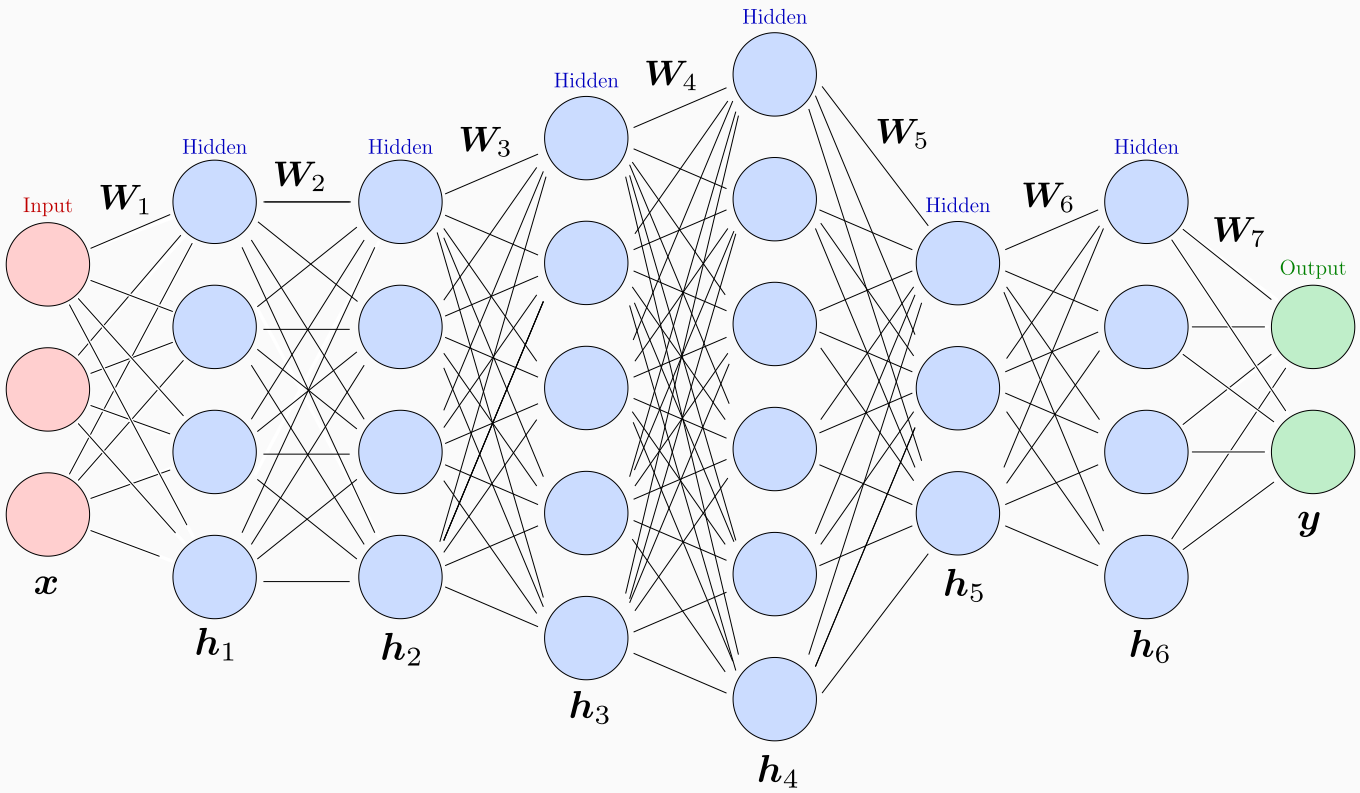


Same improvements in other fields: automatic translation, Go, ...

Deep learning

(Source: Dan Page)

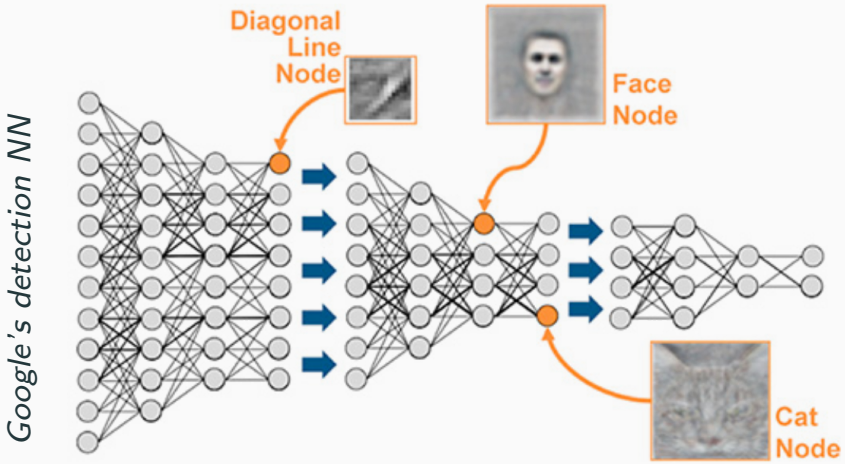
ANNs recap



- Interconnections of neurons (summations and activations),
- Feedforward architecture: input $\rightarrow$ hidden layer(s) $\rightarrow$ output,
- Parameterized by a collection of weights $\boldsymbol{W}_k$ (and biases),
- Backprop: parameters learned from a collection of labeled data.

What is deep learning?

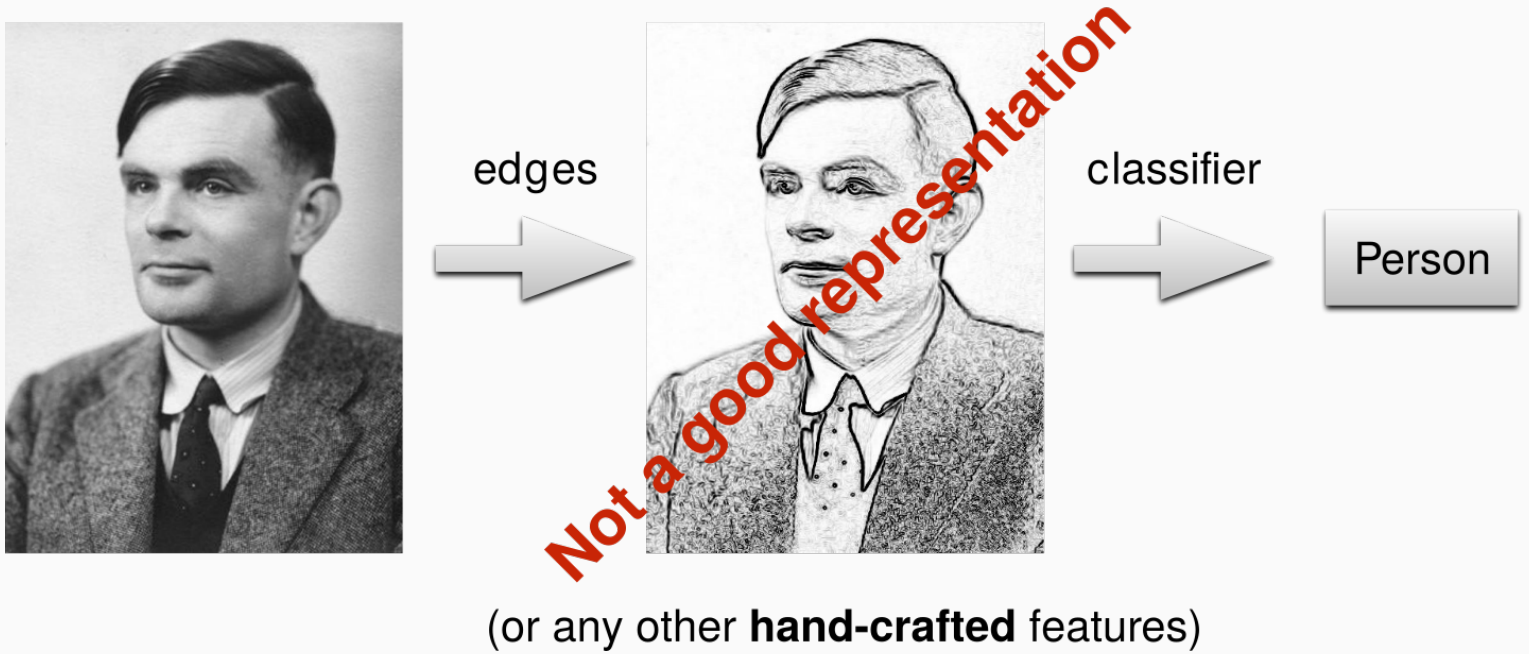
- Representation learning using artificial neural networks
  - → Learning good features automatically from raw data.
  - → Exceptionally effective at learning patterns.
- Learning representations of data with multiple levels of abstraction
  - → hierarchy of layers that mimic the neural networks of our brain,
  - → cascade of non-linear transforms.



- If you provide the system with tons of information, it begins to understand it and responds in useful ways.

(Source: Caner Hazırbaş & Lucas Masuch)

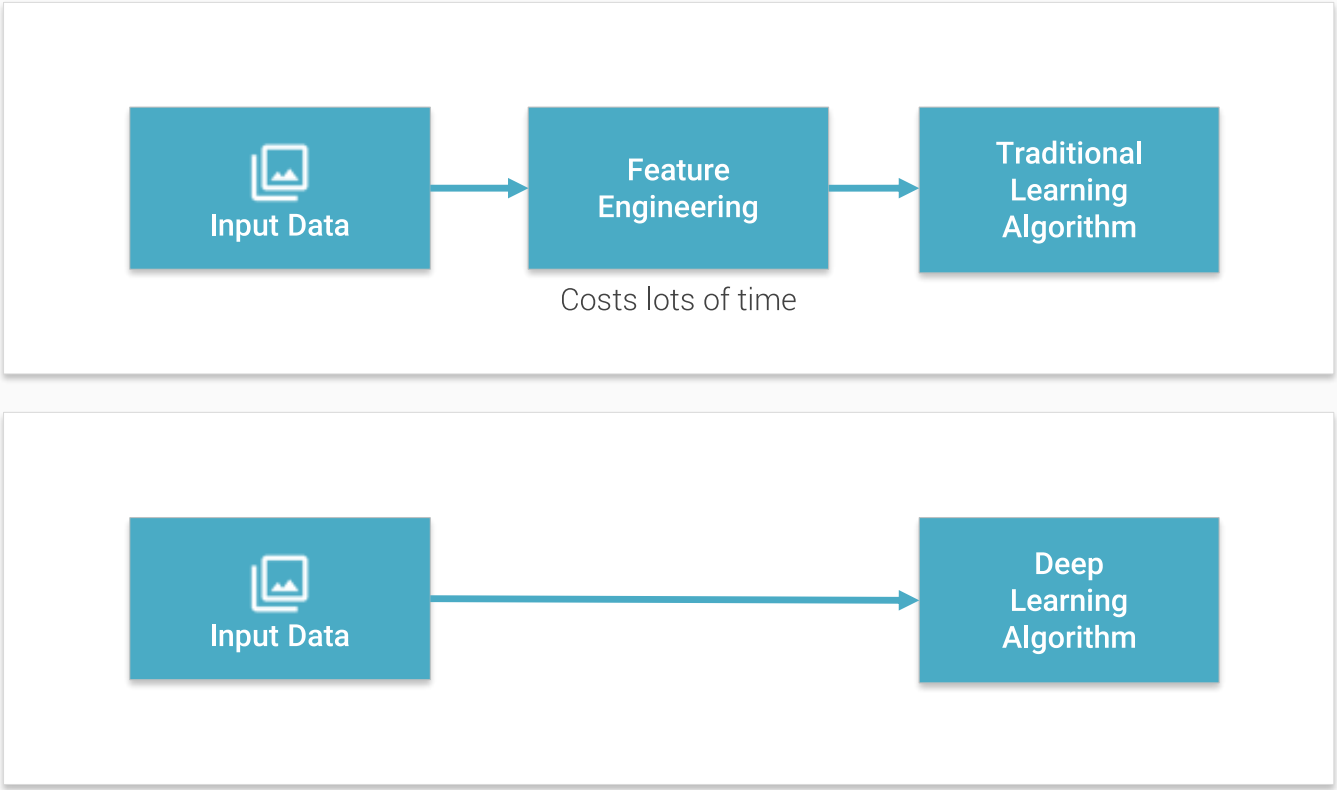
How to teach a machine?



Good representations are often very complex to define.

(Source: Caner Hazırbaş)

Deep learning: No more feature engineering



Learn the latent factors/features behind the data generation

(Source: Lucas Masuch)

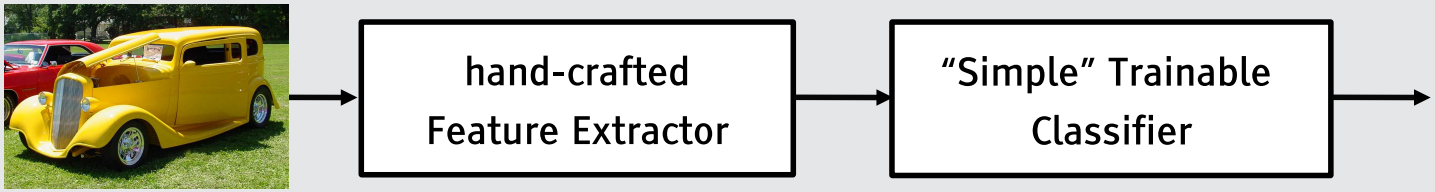
Inspired by the Brain

- The first hierarchy of neurons that receives information in the visual cortex are sensitive to specific edges while brain regions further down the visual pipeline are sensitive to more complex structures such as faces.
- Our brain has lots of neurons connected together and the strength of the connections between neurons represents long term knowledge.
- One learning algorithm hypothesis: all significant mental algorithms are learned except for the learning and reward machinery itself.

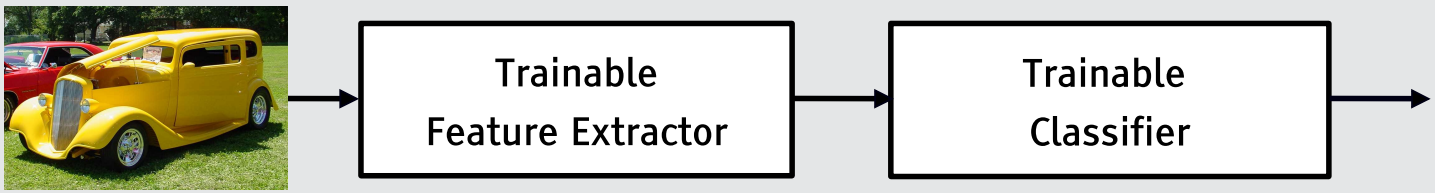
(Source: Lucas Masuch)

Deep learning: No more feature engineering

- The traditional model of pattern recognition (since the late 50's)
  - Fixed/engineered features (or fixed kernel) + trainable classifier



- End-to-end learning / Feature learning / Deep learning
  - Trainable features (or kernel) + trainable classifier

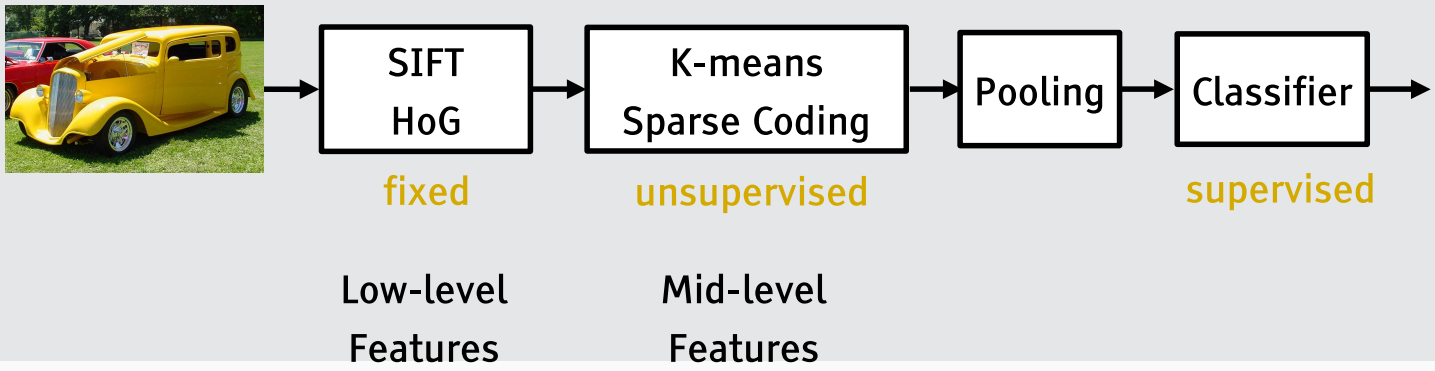


(Source: Yann LeCun & Marc'Aurelio Ranzato)

Deep learning: No more feature engineering

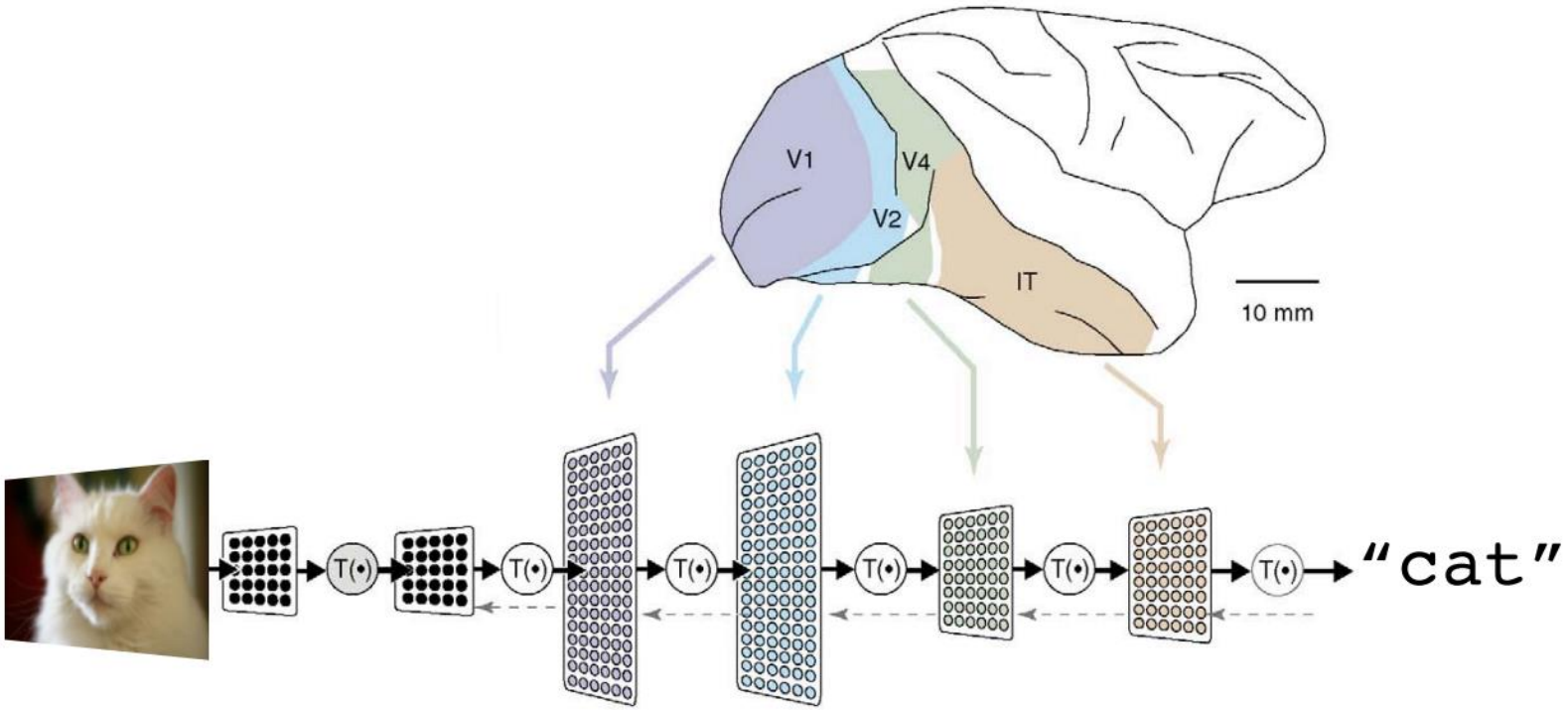
- Non-deep learning architecture for pattern recognition
  - Speech recognition: early 90's – 2011

  - Object Recognition: 2006 – 2012



(Source: Yann LeCun & Marc'Aurelio Ranzato)

Deep learning – Basic architecture



A deep neural network consists of a hierarchy of layers, whereby each layer transforms the input data into more abstract representations (e.g. edge -> nose -> face). The output layer combines those features to make predictions.

(Source: Lucas Masuch)

Trainable feature hierarchy

- Hierarchy of representations with increasing levels of abstraction.
- Each stage is a kind of trainable feature transform.

Image recognition

- Pixel $\rightarrow$ edge $\rightarrow$ texton $\rightarrow$ motif $\rightarrow$ part $\rightarrow$ object

Text

- Character $\rightarrow$ word $\rightarrow$ word group $\rightarrow$ clause $\rightarrow$ sentence $\rightarrow$ story

Speech

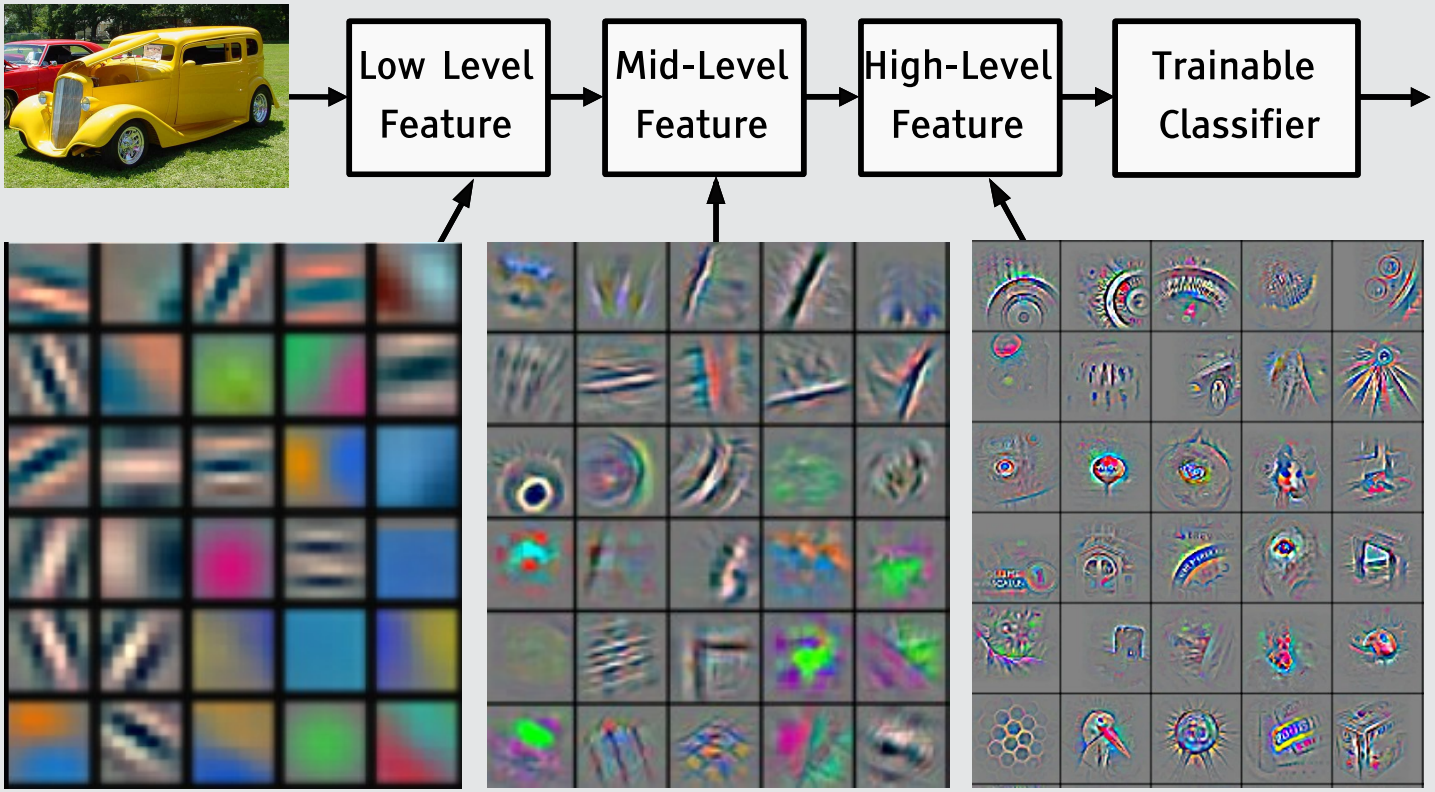
- Sample $\rightarrow$ spectral band $\rightarrow$ sound $\rightarrow$ ... $\rightarrow$ phone $\rightarrow$ phoneme $\rightarrow$ word

Deep Learning addresses the problem of learning hierarchical representations.

(Source: Yann LeCun & Marc'Aurelio Ranzato)

Deep learning – Feature hierarchy

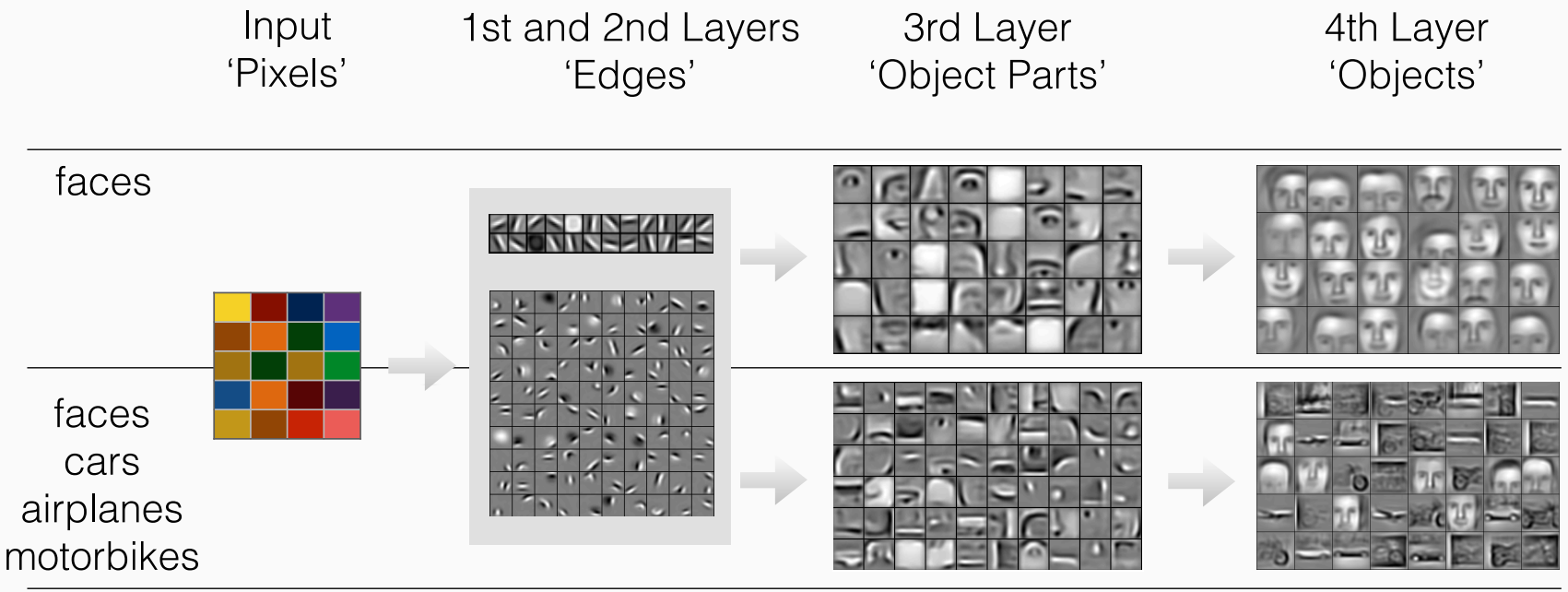
- It's deep if it has more than one stage of non-linear feature transformation



Feature visualization of convolutional net
trained on ImageNet from (Zeiler & Fergus, 2013)

(Source: Yann LeCun & Marc'Aurelio Ranzato)

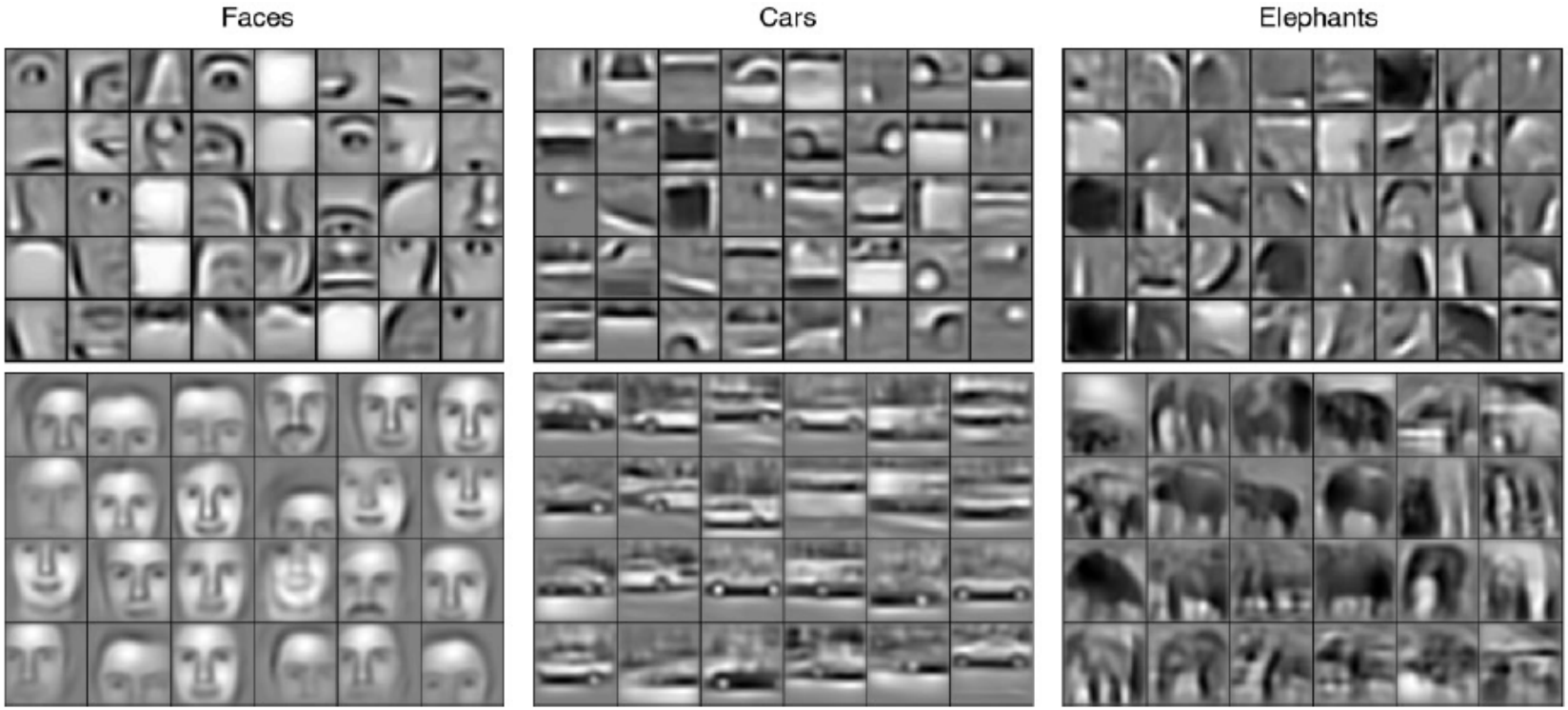
Deep learning – Feature hierarchy



Each layer progressively extracts higher level features of the input until the final layer essentially makes a decision about what the input shows. The more layers the network has, the higher level features it will learn.

(Source: Andrew Ng & Lucas Masuch & Caner Hazırbaş)

Deep learning – Feature hierarchy



- Mid- to high-level features are specific to the given task.
- Low-level representations contain less specific features.
  (can be shared for many different applications)

Deep learning – Training

Today's trend: make it deeper and deeper

- 2012: 8 layers (AlexNet – Krizhevsky *et al.*, 2012)
- 2014: 19 layers (VGG Net – Simonyan & Zisserman, 2014)
- 2014: 22 layers (GoogLeNet – Szegedy *et al.*, 2014)
- 2015: 152 layers (ResNet – He *et al.*, 2015)
- 2016: 201 layers (DenseNet – Huang *et al.*, 2017)

But remember, with back-propagation:

- We got stuck at local optima or saddle points
- The learning time does not scale well
  - it is very slow for deep networks and can be unstable.

How did networks get so deep? First, why does backprop fail?

Deep learning – Gradient vanishing problems

Back-propagation and gradient vanishing problems

$$\text{Update} \quad w_{i,j} \leftarrow w_{i,j} - \gamma \frac{\partial E(\boldsymbol{W})}{\partial w_{i,j}} \quad \text{with} \quad \frac{\partial E(\boldsymbol{W})}{\partial w_{i,j}} = \sum_{(\boldsymbol{x},\boldsymbol{d})\in\mathcal{T}} \delta_i h_j$$

$$\text{where} \quad \delta_i = g'(a_i) \times \begin{cases} y_i - d_i & \text{if } i \text{ is an output node} \\ \sum_l w_{l,i}\delta_l & \text{otherwise} \end{cases}$$

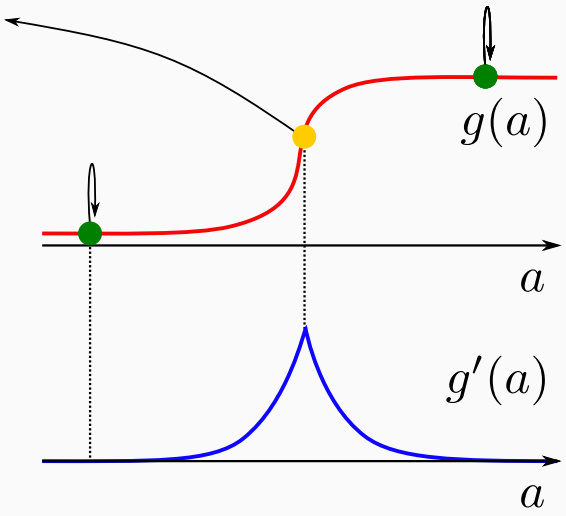
- With deep networks, the gradient vanishes quickly.
- Unfortunately, this arises even though we are far from a solution.
- The updates become insignificant, which leads to slow training rates.
- This strongly depends on the shape of $g'(a)$.

- The gradient may also explode leading to instabilities:
  $\rightarrow$ gradient exploding problem.

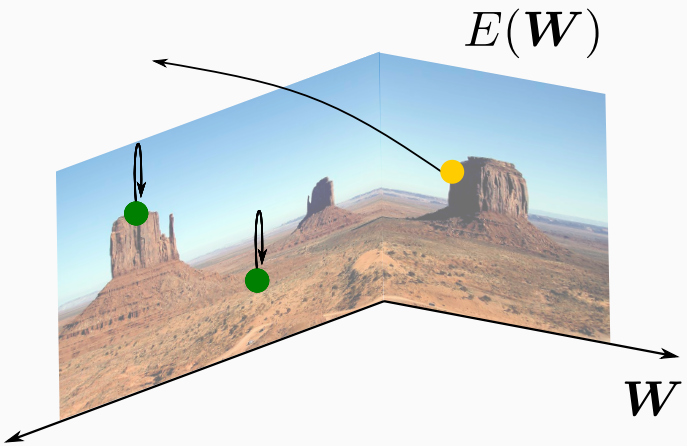
Deep learning – Gradient vanishing problem

As the network gets deeper, the landscape of E becomes:

- very hilly → lots of stationary points,
- with large plateaus → gradient vanishing problem,
- and delimited by cliffs → gradient exploding problem.



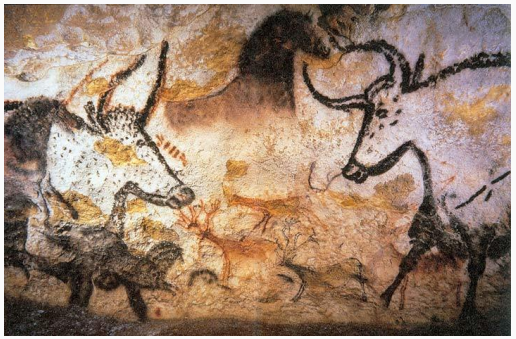
Activation function



Cost landscape

So, what has changed?

Unsupervised pretraining (Prehistory)



Depiction of aurochs, horses and deer (Lascaux, France)

(Greedy layer-wise) Unsupervised pretraining

Idea: Training a shallow network is easier than training a deep one.

1. **Pretraining:** Train several successive layers in an unsupervised way,
2. **Stacking:** Stack them, stick a classifier at the last layer, and
3. **Fine tuning:** do classical back propagation.

- Originally proposed by Hinton 2006 to learn Deep Belief Networks (DBN),
  $\rightarrow$ Considered as deep learning renaissance.
- Greedy because each layer is optimized independently,
- Regularization: decreases test error without decreasing training error.

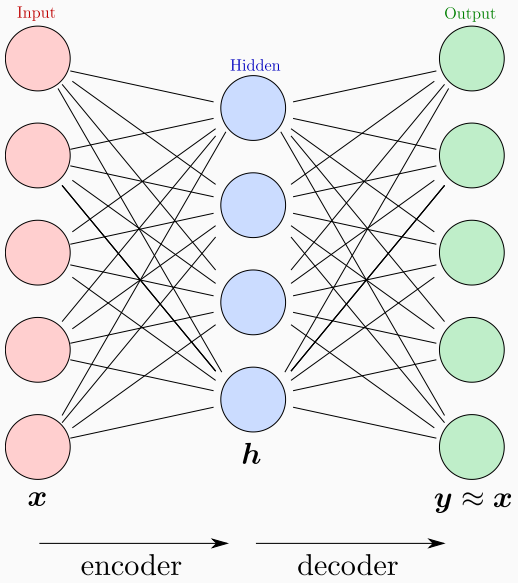
How to train layers/networks in an unsupervised way?

For DBN, Hinton proposed using restricted Boltzmann machine.

But here, we will consider auto-encoders (Bengio *et al.*, 2007).

Auto-encoders (AEs)

- **Unsupervised learning:** unlabeled training data $\mathcal{T} = \{\boldsymbol{x}^1, \boldsymbol{x}^2, \ldots, \boldsymbol{x}^N\}$
- **Applications:** Dimensionality reduction, denoising, feature extraction, ...
- **Model:** feed-forward NNs trained to reproduce the inputs at the outputs,
- Early attempts date back to auto-associative memories (Kohonen 80s),
- Renewed interest since 2006.



Trained with backprop, using the $\boldsymbol{x}$'s both as inputs and desired outputs, *e.g.*:

$$E(\boldsymbol{W}) = \sum_{i=1}^{N} \|\boldsymbol{x}^i - \boldsymbol{y}^i\|^2 \quad \text{where} \quad \boldsymbol{y}^i = f(\boldsymbol{x}^i; \boldsymbol{W})$$

The hidden layer $\boldsymbol{h}$ learns features relevant for the given dataset. Also called representation layer.

Auto-encoders (AEs)

- Will try to reproduce the input, *i.e.*, to learn the identity function.
- In order to avoid this trivial solution: constrain the hidden layer.

Undercomplete representations

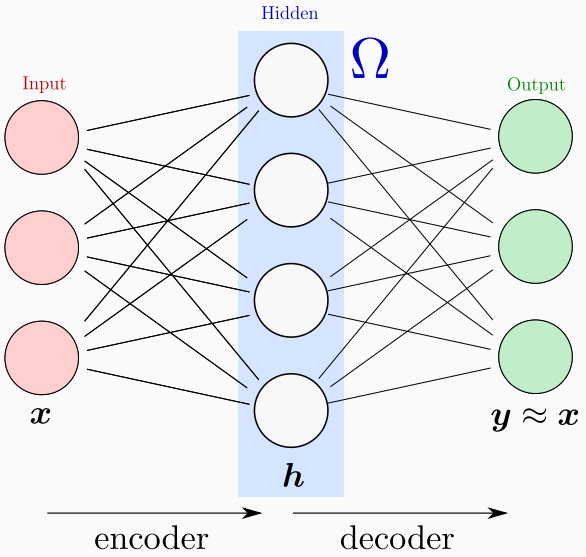
- Classical constraint is to use less hidden units than in the input layer, (like on the previous illustration)
- Dimensionality reduction: learn a compressed representation $\boldsymbol{h}$ of the data by exploiting the correlation among the features,
- If all units are linear and the loss is the MSE: performs a transformation similar to PCA, *i.e.*, it learns a projection that maximizes the variance of the data (without the orthogonal property).

Auto-encoders (AEs)

Overcomplete representations (1/3)

- Hidden layer may be larger than the input layer.
- Add a penalization into the loss function to avoid the identity solution:

$$E(\boldsymbol{W}) = \sum_{i=1}^{N} \|\boldsymbol{x}^i - \boldsymbol{y}^i\|^2 + \Omega(\boldsymbol{h}^i)$$

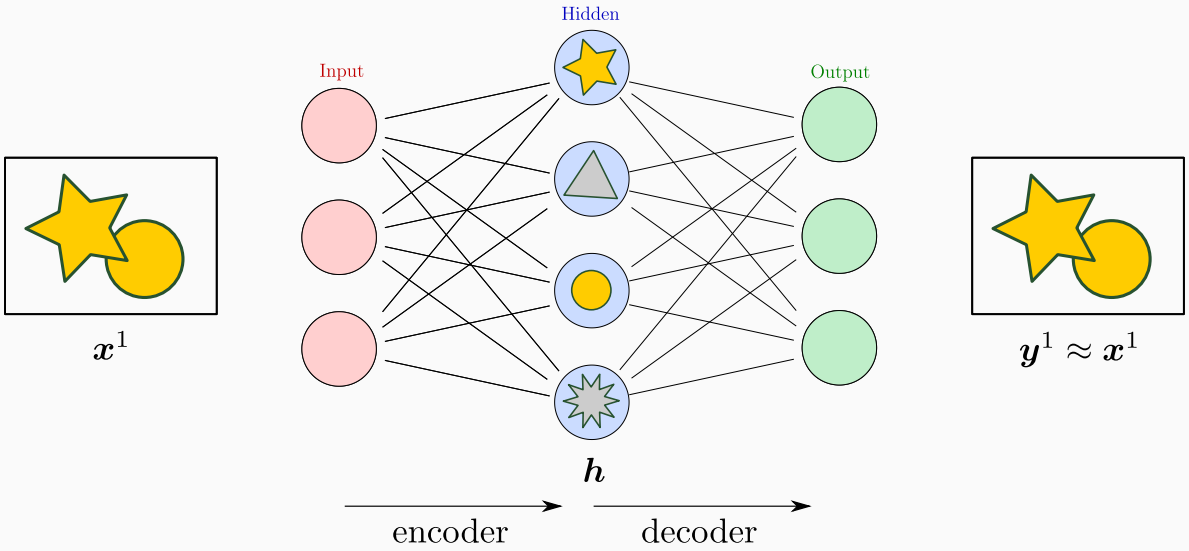


- The penalty prevents hidden units from being freely independent, and $\boldsymbol{h}$ to take arbitrary configurations.
- This requires adapting the backprop algorithm.

Auto-encoders (AEs)

Overcomplete representations (2/3)

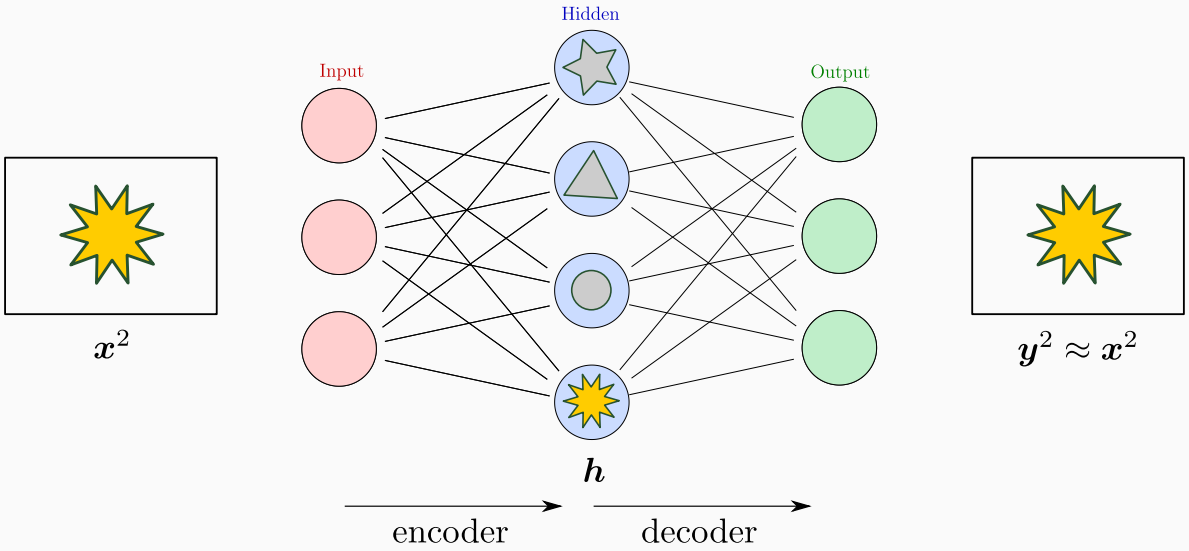
- **Sparse autoencoders:** ℓ_1 penalty: $\Omega(\boldsymbol{h}) = \lambda \|\boldsymbol{h}\|_1 = \lambda \sum |h_k|$
  - promote sparsity: tends to drive some entries of $\boldsymbol{h}$ to be exactly zero,
  - responds differently to different stimuli $\boldsymbol{x}$ (different non-zero coeffs),
  - hidden units are specialized,
  - learns local mappings (union of subspaces/manifold learning).



Auto-encoders (AEs)

Overcomplete representations (2/3)

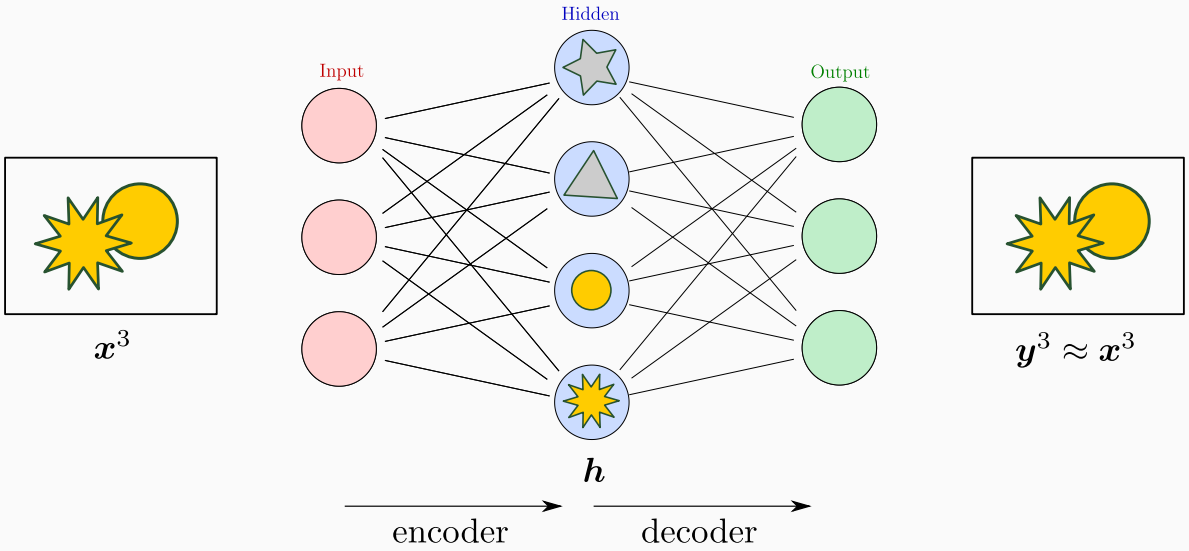
- **Sparse autoencoders:** ℓ_1 penalty: $\Omega(\boldsymbol{h}) = \lambda \|\boldsymbol{h}\|_1 = \lambda \sum |h_k|$
  - promote sparsity: tends to drive some entries of $\boldsymbol{h}$ to be exactly zero,
  - responds differently to different stimuli $\boldsymbol{x}$ (different non-zero coeffs),
  - hidden units are specialized,
  - learns local mappings (union of subspaces/manifold learning).



Auto-encoders (AEs)

Overcomplete representations (2/3)

- **Sparse autoencoders:** ℓ_1 penalty: $\Omega(\boldsymbol{h}) = \lambda \|\boldsymbol{h}\|_1 = \lambda \sum |h_k|$
  - promote sparsity: tends to drive some entries of $\boldsymbol{h}$ to be exactly zero,
  - responds differently to different stimuli $\boldsymbol{x}$ (different non-zero coeffs),
  - hidden units are specialized,
  - learns local mappings (union of subspaces/manifold learning).



Auto-encoders (AEs)

Overcomplete representations (3/3)

- **Contractive autoencoders:**

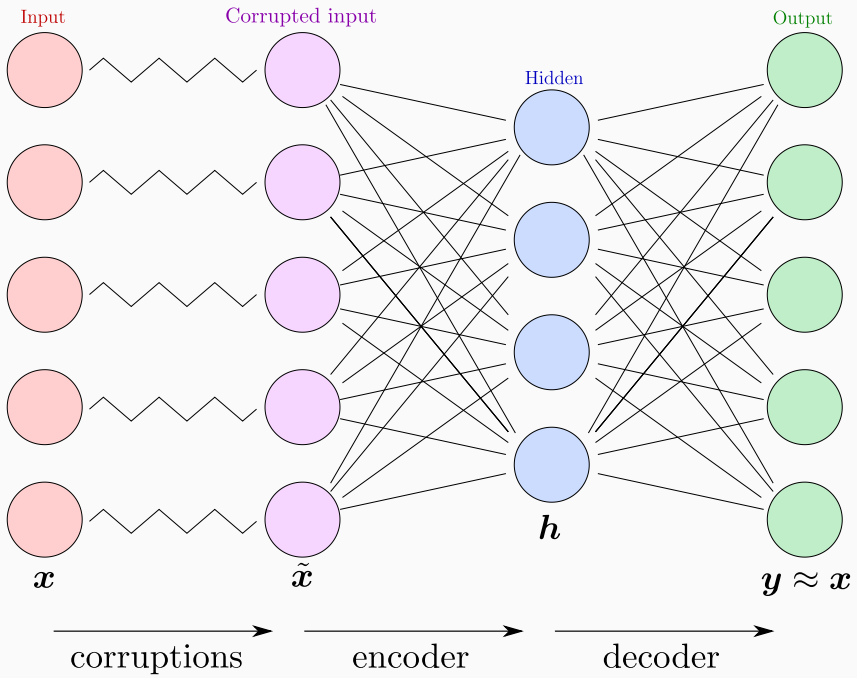
$$\Omega(\boldsymbol{h}) = \lambda \| \frac{\partial \boldsymbol{h}}{\partial \boldsymbol{x}} \|_F^2 = \lambda \sum_k \sum_l \left(\frac{\partial h_k}{\partial x_l} \right)^2$$

  - Encourage $\frac{\partial \boldsymbol{h}}{\partial \boldsymbol{x}}$ to be close to 0,
  - Insensitive to variations in $\boldsymbol{x}$ except in meaningful directions.

- **Prior on the distribution of hidden units**
  - Under a probabilistic interpretation of the NN, priors can play the role of penalties (Bayesian framework).
  - Ex: $\Omega(\boldsymbol{h}) = \lambda \|\boldsymbol{h}\|_1 = -\log p(\boldsymbol{h}) + cst.$ under a Laplacian prior p.

Denoising auto-encoders (DAEs) (Vincent et al., 2008)

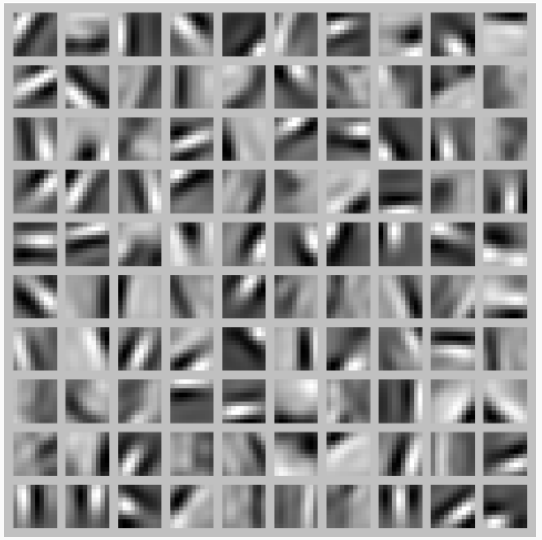
- Train the AE to reconstruct the input from a corrupted version of it:
  - Some inputs randomly set to 0: $\tilde{x}_i = x_i$ with proba P, 0 otherwise,
  - Or Gaussian additive noise: $\tilde{x}_i = x_i + w_i, \quad w_i \sim \mathcal{N}(0, \sigma^2)$.
- Force the hidden layer to discover more robust features,
- Sort of regularization: prevents from simply learning the identity.



Auto-encoders (AEs) – Visualizing hidden units activities

One way to understand what a hidden unit (or any unit) is doing, is to look at its weight vector.

- Sparse AE trained on a small set of 10×10 images,
- 100 hidden units (hidden layer size = input layer size),
- Each square represents an input image maximizing the response of one of the hidden units,
- Each hidden unit has learned a feature detector,
- Mostly edge detectors at different angles and positions
  → Low-level features.

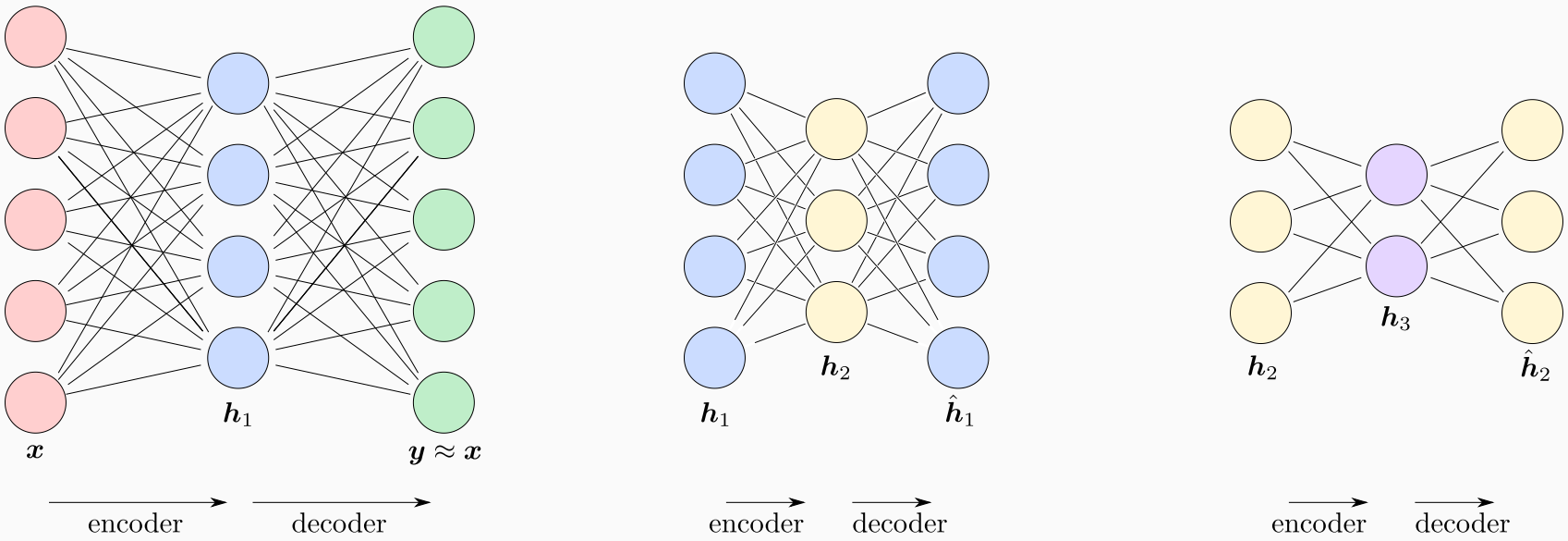


(Source: Andrew Ng)

How to get higher level features?

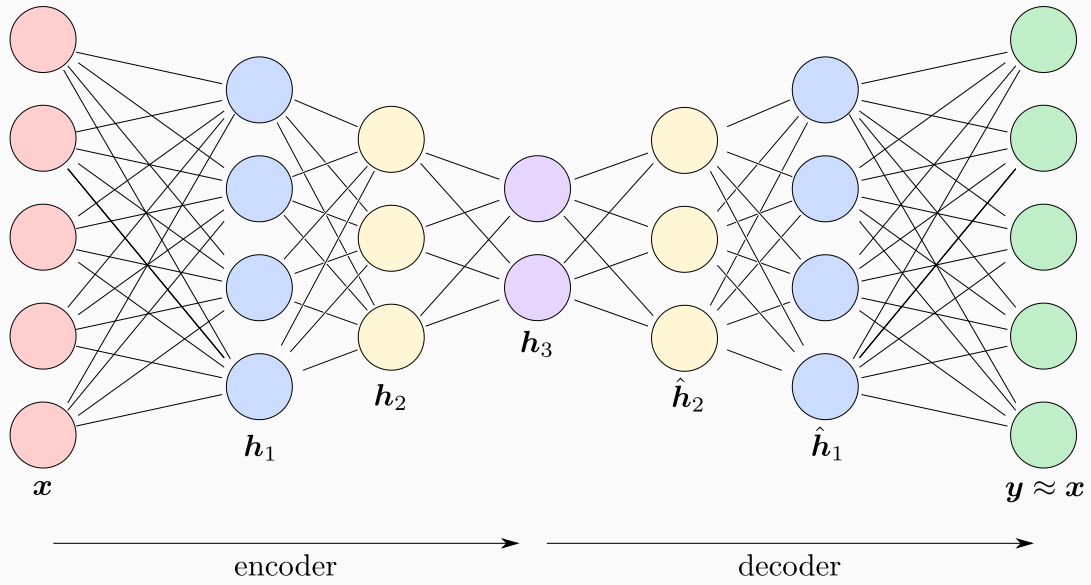
Stacked auto-encoders (Bengio *et al.*, 2007)

- **Idea:** Train successive layers of AEs to learn more complex features.
- **Strategy**: (greedy layer-wise training)
  - Each AE is trained using the representation layer of a previous AE,



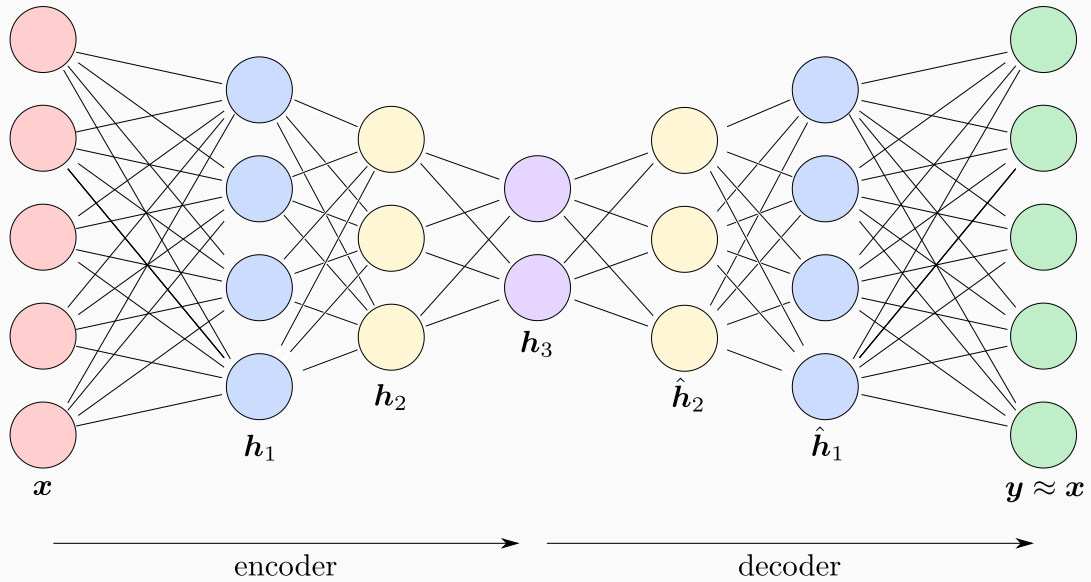
Stacked auto-encoders (Bengio *et al.*, 2007)

- **Idea:** Train successive layers of AEs to learn more complex features.
- **Strategy**: (greedy layer-wise training)
  - Each AE is trained using the representation layer of a previous AE,
  - The encoders and decoders are next stacked together,



Stacked auto-encoders (Bengio *et al.*, 2007)

- **Idea:** Train successive layers of AEs to learn more complex features.
- **Strategy**: (greedy layer-wise training)
  - Each AE is trained using the representation layer of a previous AE,
  - The encoders and decoders are next stacked together,



- Fine tuning can be performed to tweak the deep AE.

Deep auto-encoders

Learning High Level Representations in Videos – Google (Le et al. 2012)

- 10 million images from YouTube videos,
- Images are 200×200 pixels,
- Sparse deep auto-encoder with 10^9 connections,
- Show test images to the network (*e.g.*, faces),
- Look for neurons with maximum response,
- Some neurons respond to high level characteristics
  $\rightarrow$ Faces, cats, silhouettes, ...
- Major breakthrough in machine learning.

Remark: in fact this deep auto-encoder has not been trained using unsupervised pretraining, but with techniques that will be introduced later.

Combining deep auto-encoders with a classifier

Idea:

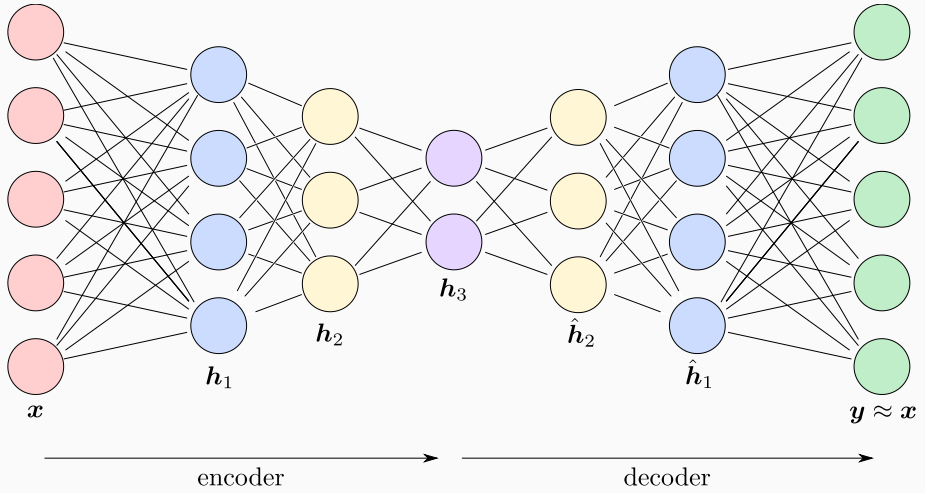
- The learned features can be used as inputs to any other system,
- Instead of using the input features $\boldsymbol{x}$ use $\boldsymbol{h}$ obtained at the deepest layer of the encoder.

Strategy:

1. Given a training set $\{(\boldsymbol{x}^1, d^1), \ldots, (\boldsymbol{x}^N, d^N)\}$,
2. Unsupervisedly train an auto-encoder from $\{\boldsymbol{x}^1, \ldots, \boldsymbol{x}^N\}$,
3. Get the feature codes $\{\boldsymbol{h}^1, \ldots, \boldsymbol{h}^N\}$ (from the deepest layer),
4. Use these codes to train a classifier on $\{(\boldsymbol{h}^1, d^1), \ldots, (\boldsymbol{h}^N, d^N)\}$,
5. Plug the encoder and the classifier together,
6. Perform fine tuning (end-to-end training with backprop).

Combining deep auto-encoders with a classifier

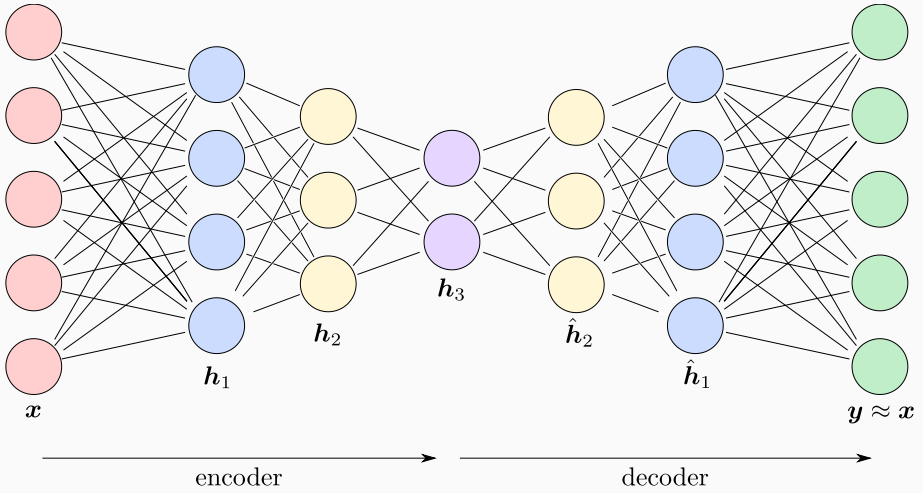
1. Train auto-encoder:



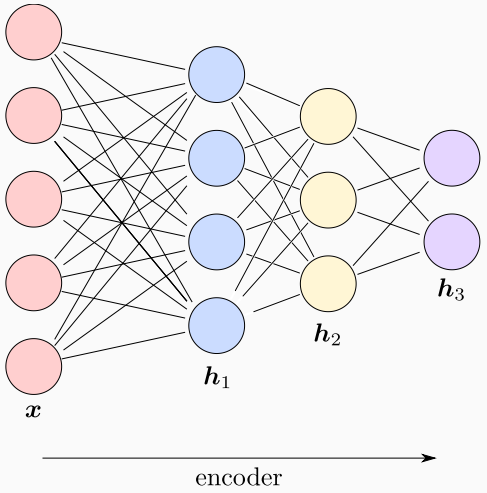
2. Plug classifier:

Combining deep auto-encoders with a classifier

1. Train auto-encoder:

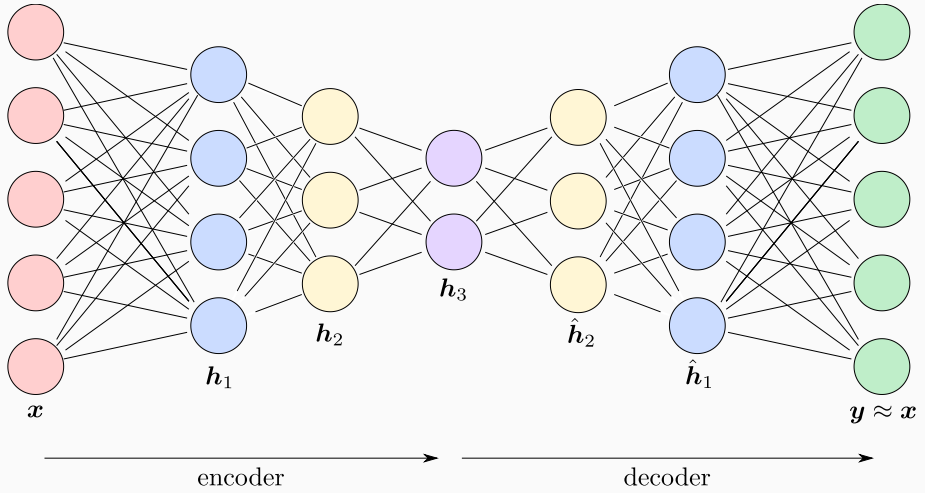


2. Plug classifier:

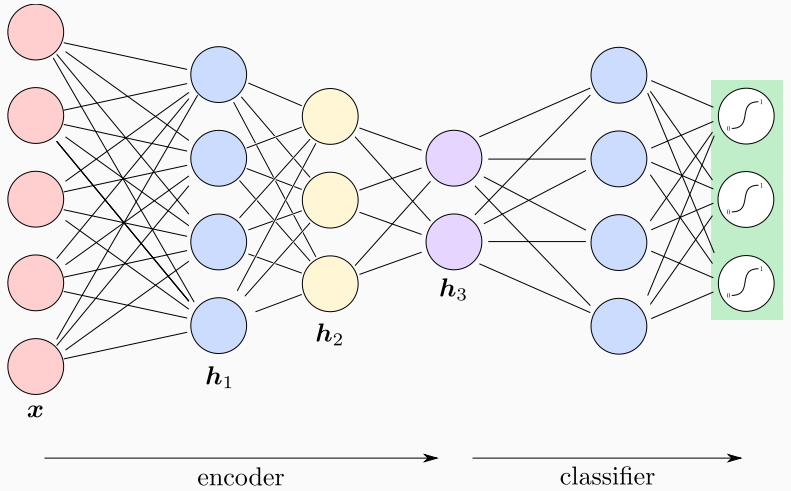


Combining deep auto-encoders with a classifier

1. Train auto-encoder:



2. Plug classifier:



Unsupervised pretraining – Overview

Why not learn all the layers at once?

- The gradients propagated backwards diminish rapidly in magnitude so that the first layers will not learn much (vanishing gradient problem),
- Non convex form of the loss function: many local minima.

Why use auto-encoders?

- Unsupervised learning
  - Often easy to get unlabeled data,
  - Extract "meaningful" features,
  - Can be used as preprocessing for other datasets (transfer learning).
- Supervised fine tuning
  - Can be used when labels are available for other tasks.

Today, there exist procedures for training deep NN architectures from scratch, but the unsupervised pretraining approach was the first method to succeed.

Modern deep learning – Recipes

What has changed again?

Modern deep learning makes use of several tools and heuristics for training large architectures: type of units, normalization, optimization...

1. Fight vanishing/exploding gradients

   $\longrightarrow$ Rectifiers, Gradient clipping.

2. Improve accuracy by using tons of data thanks to
   - Stochastic gradient descent (and variants),
   - GPUs, parallelization, distributed computing, ...

3. Fight poor solutions (saddle points) and over-fitting

   $\longrightarrow$ Early stopping, Dropout, Batch normalization, Regularization, ...

4. Multitude of open source solutions.

Disclaimers

The field evolves so quickly that parts of what follows are possibly already outdated.

And if they are not outdated yet, they will be in the next few months.

Many recent tools, sometimes less than a year old, become the new standard just because they work better.

Why do they work better? The answer is usually based on intuition, and rigorous answers are often lacking.

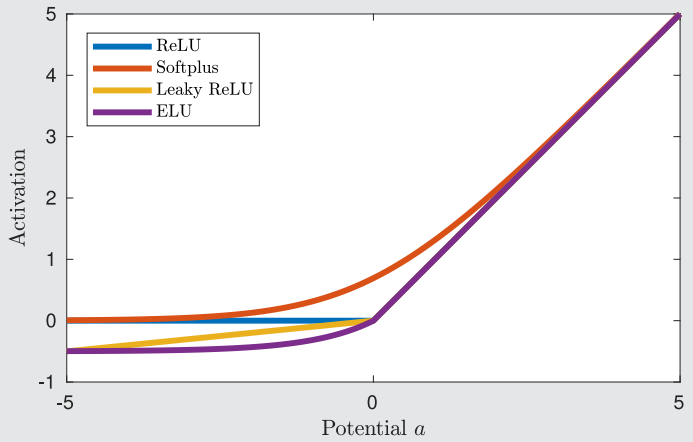
Rectifier (Glorot, Bordes and Bengio, 2011)

- **Goal:** Avoid the gradient vanishing problem.
- **How:** make sure $g'(a_j)$ is not close to zero for all training points, this was often arising with hyperbolic tangents or sigmoids.

- Rectifier linear unit (ReLU):

$$g(a) = \max(a, 0)$$

- Softplus: $g(a) = \log(1 + \exp a)$
- Leaky ReLU: $g(a) = \max(a, 0.1a)$
- Exponential LU (ELU): $g(a) = \begin{cases} a & \text{if} \quad a \geqslant 0 \\ \alpha(e^a - 1) & \text{otherwise} \end{cases}$



In a randomly initialized network, about 50% of hidden units are activated.
It also promotes sparsity of hidden units (structured/organized features).

Gradient clipping (Mikolov, 2012; Pascanu et al., 2012)

- **Goal:** Avoid the gradient exploding problem.
- **How:** make sure $\nabla E(\boldsymbol{W})$ does not get too large.

- Cliffs cause the learning process to overshoot a desirable minimum.
- Preserve the direction but prevent extreme values:

$$\text{If } \|\nabla E(\boldsymbol{W}^t)\| \leqslant \tau: \quad \boldsymbol{W}^{t+1} \leftarrow \boldsymbol{W}^t - \gamma \nabla E(\boldsymbol{W}^t)$$

$$\text{Else:} \quad \boldsymbol{W}^{t+1} \leftarrow \boldsymbol{W}^t - \gamma \tau \frac{\nabla E(\boldsymbol{W}^t)}{\|\nabla E(\boldsymbol{W}^t)\|}$$

- Pick threshold τ by averaging the norm over a large number of updates.
- New solution $\boldsymbol{W}^{t+1}$ stays in a ball of radius τ around the current solution.

Stochastic gradient descent

Reminder: (Batch) gradient descent with backprop

1. For each training sample
   - Perform backprop,
   - Sum their gradients.
2. Update the weights (based on the gradient over the whole *batch*).

Go to Step 1

→ Convergence to a stationary point under some technical assumptions.

- Requires the loading of the whole dataset into memory (big data sets),
- Can't be efficiently parallelized or used online,
- Scanning the whole training set is slow (*epoch*),
- *Epoch:* a scan in which each training sample has been visited at least once.
- After 10 epochs, the weights have been updated only 10 times,
- Designed to fall into local minima.

Stochastic gradient descent

Stochastic gradient descent (SGD) **(Robbins and Monro, 1951)**

1. For each training sample (shuffled randomly – thus "stochastic")
   - Perform backprop,
   - Evaluate its gradient,
   - Update the weights (based only on that sample).

$\rightarrow$ Convergence under technical assumptions with high probability.

- After 10 epochs, the weights have been updated $10 \times N$ times (N size of the training set),
- Faster if the gradient of each sample is a good proxy for the total gradient,
- Can get out of local minima as it explores the space in a random fashion,
- It allows online learning,
- But it is noisy, unstable and can still get trapped into local minima.

- Origin: ADALINE (Widrow & Hoff, 1960).

Stochastic gradient descent

Mini-batch gradient descent

1. Shuffle: create a random partition of subsets of size b (*mini-batches*)
2. For each mini-batch
   - Perform backprop and accumulate the gradients,
   - Update the weights (based on the gradient of the *mini-batch*)

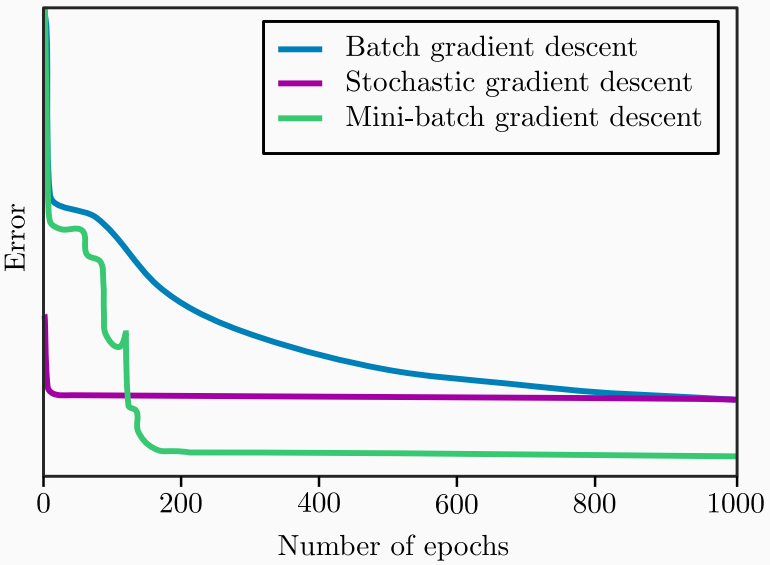
Go to Step 1

$\rightarrow$ Convergence under technical assumptions with high probability.

- Usually, a mini-batch contains around 50 to 256 data points,
- After 10 epochs, the weights have been updated $10 \times N/b$ times
- Smoothes the noise, then provides a better proxy for the total gradient,
- Can still get out of some local minima,
- Trade-off between batch GD and SGD (stability vs. speed)
- Enjoys highly optimized matrix manipulation tools.

Remark: Nowadays, SGD often referred to as mini-batch gradient descent.

Stochastic gradient descent



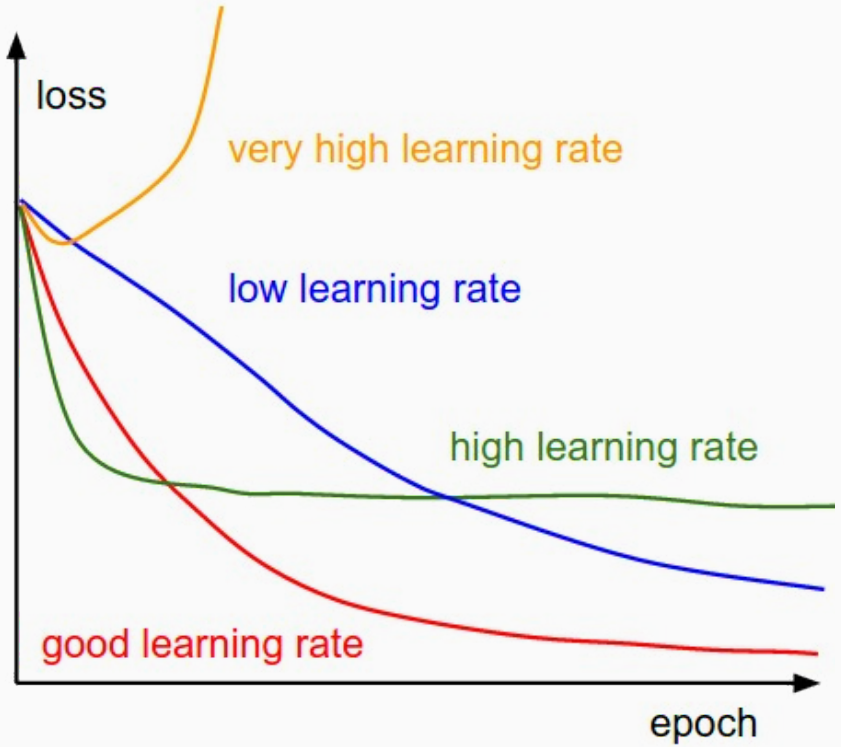
Mini-batch gradient descent:

- Not only faster,
- Usually finds better solutions.

Because randomness helps explore new configurations,
and mini-batches favor more relevant configurations.

But make sure to adjust the learning rates.

Learning rate (reminder)



$$\boldsymbol{W}^{t+1} \leftarrow \boldsymbol{W}^{t} - \gamma \nabla E(\boldsymbol{W}^{t})$$

Learning rate schedules

Learning rate schedules: reduce γ at each epoch t

$$\boldsymbol{W}^{t+1} \leftarrow \boldsymbol{W}^t - \gamma^t \nabla E(\boldsymbol{W}^t) \quad \text{with} \quad \lim_{t\to\infty} \gamma^t = 0$$

- Time-based decay: $\gamma^t = \frac{1}{1+\kappa t}\gamma^0, \kappa > 0$ (*aka*, search-then-converge)
- Step decay: $\gamma^t = \kappa^t \gamma^0, 0 < \kappa < 1$ (*aka*, drop decay)
- Exponential decay: $\gamma^t = \exp(-\kappa t)\gamma^0, \kappa > 0$

- Guaranteed to converge (avoid SGD from oscillating forever),
- But may stop too early (too fast),
- Connections with simulated annealing (γ^t seen as temperature),
  - $\rightarrow$ Hence, often referred to as annealing.
    - Large γ^t: explore the space (allows uphill jump),
    - Small γ^t: move downhill towards the best local solution.
- Variant: restart schedule from the best iterate (achieving smaller loss).

Adaptive learning rates

Adaptive learning rates: adapt γ during the iteration t independently for each connection w_{ij}

$$w_{ij}^{t+1} \leftarrow w_{ij}^{t} - \gamma_{ij}^{t} \frac{\partial E(\boldsymbol{W}^t)}{\partial w_{i,j}^{t}}$$

AdaGrad (Duchi *et al.*, 2011): $$\gamma_{i,j}^{t} = \frac{\gamma^0}{\sqrt{\sum_{k=1}^{t} \left| \frac{\partial E(\boldsymbol{W}^k)}{\partial w_{i,j}^{k}} \right|^2 + \varepsilon}}$$

Use larger rates for directions that remain flat, smaller if they keep changing.

Other strategies: RMSProp (Hinton, 2012) and Adam (Kingma & Ba, 2015).

new standard

Main advantage: Less sensitive to the initial choice of $\gamma^0 > 0$.

Further reading: `http://ruder.io/optimizing-gradient-descent/`

Momentum and Nesterov accelerated gradient (NAG)

Idea: denoise gradients (in SGD) by using inertia, memory of velocity

(Stochastic) gradient descent (for each mini-batch)

$$\boldsymbol{W}^{t+1} \leftarrow \boldsymbol{W}^t - \gamma \nabla E(\boldsymbol{W}^t)$$

Momentum – introduce velocity $\boldsymbol{V}^t$ (parameter $\rho \approx .9$)

$$\boldsymbol{W}^{t+1} \leftarrow \boldsymbol{W}^t - \underbrace{\left(\gamma \nabla E(\boldsymbol{W}^t) + \rho \boldsymbol{V}^t\right)}_{=\boldsymbol{V}^{t+1}}$$

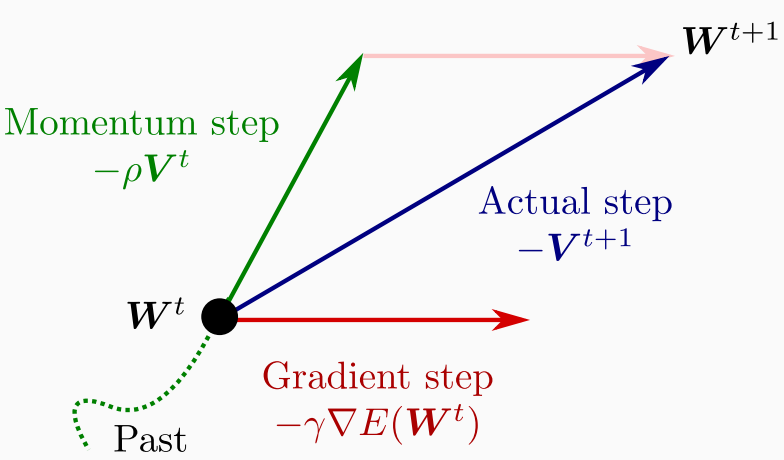
Nesterov acceleration (1983) (in deep community: Sutskever *et al.*, 2013)

$$\boldsymbol{W}^{t+1} \leftarrow \boldsymbol{W}^t - \underbrace{\left(\gamma \nabla E(\boldsymbol{W}^t - \rho \boldsymbol{V}^t) + \rho \boldsymbol{V}^t\right)}_{=\boldsymbol{V}^{t+1}}$$

Nice illustrations: `https://distill.pub/2017/momentum/`

Momentum and Nesterov accelerated gradient (NAG)

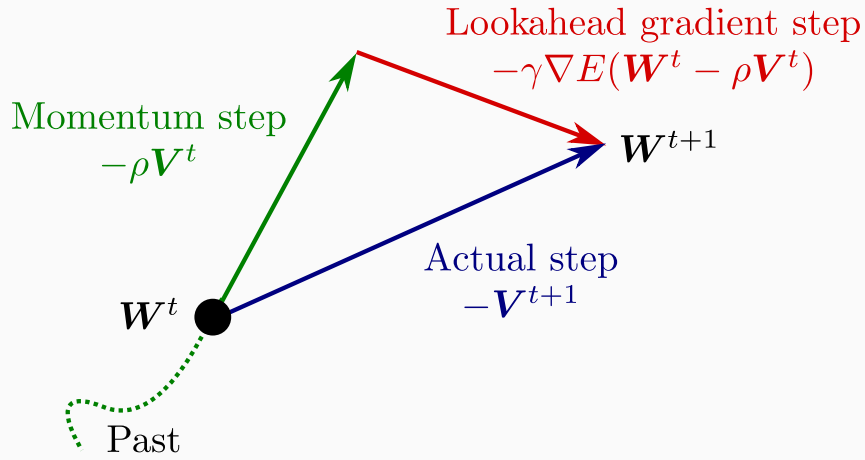
Momentum update



$$W^{t+1} \leftarrow W^t - \underbrace{\left(\gamma \nabla E(W^t) + \rho V^t\right)}_{=V^{t+1}}$$

(Correction of the gradient)

Nesterov acceleration update



$$W^{t+1} \leftarrow W^t - \underbrace{\left(\gamma \nabla E(W^t - \rho V^t) + \rho V^t\right)}_{=V^{t+1}}$$

(Correction of the momentum)

In convex and batch settings, NAG decays in $O(1/t^2)$ instead of $O(1/t)$.
For deep learning, dramatic acceleration observed in practice.
It also improves regularity of SGD.

Weight decay

Goal: prevent from exploding weights, limit the freedom, avoid over-fitting.

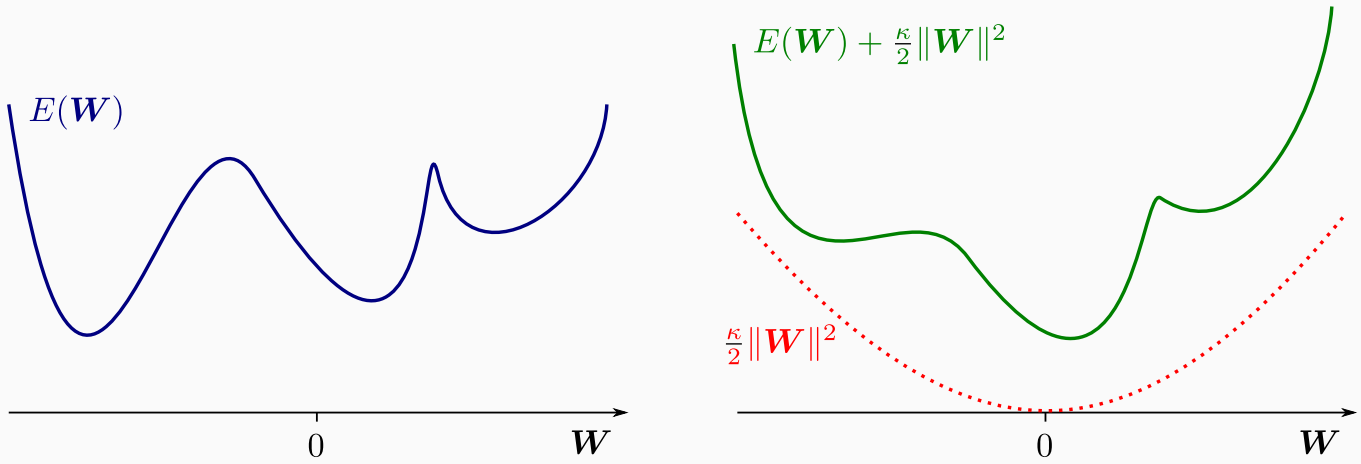
Weight decay:

$$\boldsymbol{W}^{t+1} \leftarrow \boldsymbol{W}^t - \gamma(\nabla E(\boldsymbol{W}^t) + \kappa \boldsymbol{W}^t)$$

It corresponds to adding an ℓ_2 penalty to the loss

$$\min_{\boldsymbol{W}} E(\boldsymbol{W}) + \frac{\kappa}{2}\|\boldsymbol{W}\|_F^2$$

and has the effect of shrinking the weights towards zero.



Gradient noise (Neelakantan, 2015)

If a little bit of noise is good, what about adding noise to the gradient?

$$\boldsymbol{W}^{t+1} \leftarrow \boldsymbol{W}^t - \gamma(\nabla E(\boldsymbol{W}^t) + \mathcal{N}(0, \sigma_t^2)) \quad \text{with} \quad \sigma_t = \frac{\sigma_0}{(1+t)^\kappa}$$

Works surprisingly great in many situations!

All together

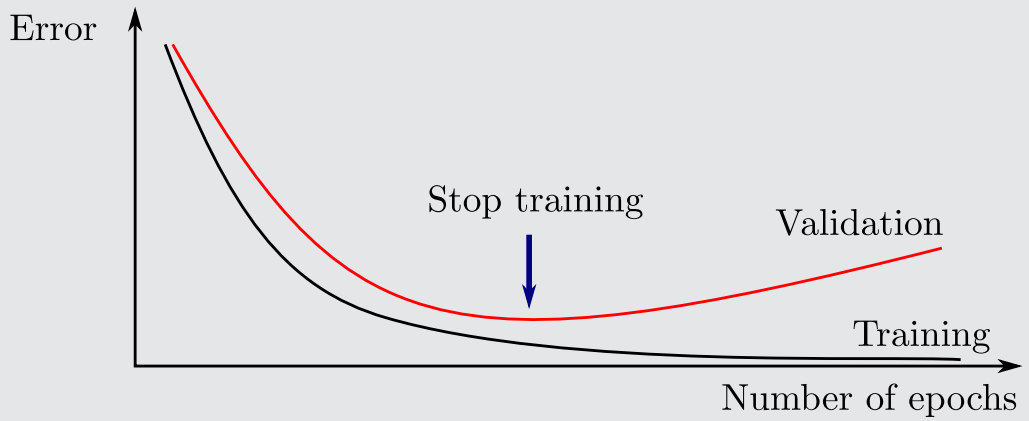
Nesterov + Adaptive learning rates + Weight decay + Gradient noise:

$$\boldsymbol{W}^{t+1} \leftarrow \boldsymbol{W}^t - \underbrace{\left(\gamma^t(\nabla E(\boldsymbol{W}^t - \rho \boldsymbol{V}^t) + \kappa(\boldsymbol{W}^t - \rho \boldsymbol{V}^t) + \mathcal{N}(0, {\sigma_t}^2)) + \rho \boldsymbol{V}^t\right)}_{=\boldsymbol{V}^{t+1}}$$

Good luck to tune all the parameters jointly!
Luckily, many toolboxes offer good default settings.

Early stopping (Girosi, Jones and Poggio, 1995)

- **Goal:** Reduce over-fitting.
- **How:** Stop the iterations of gradient descent before convergence.



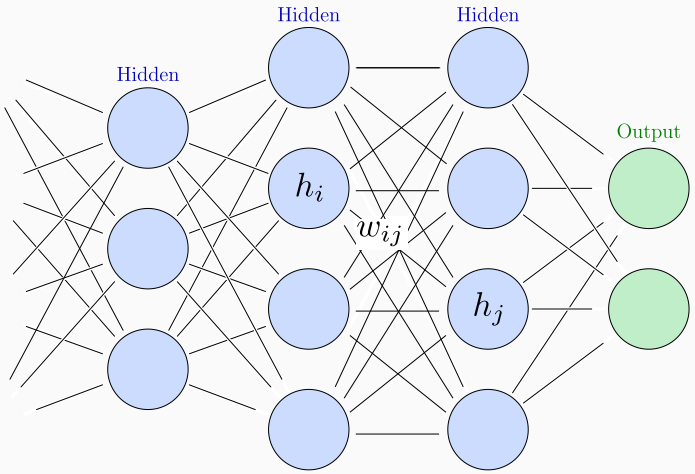
1. Split the training set into a training subset and a validation subset,
2. Train only on the training subset,
3. Evaluate the error on the validation subset every few epochs,
4. Stop training as soon as the error on the validation starts increasing.

Normalized initialization

Idea: A good weight initialization should be random

$$w_{ij} \sim \underbrace{\mathcal{N}(0, \sigma^2)}_{\text{Gaussian}} \quad \text{or} \quad w_{ij} \sim \underbrace{\mathcal{U}\left(-\sqrt{3}\sigma, \sqrt{3}\sigma\right)}_{\text{Uniform}}$$

with standard deviation σ chosen such that all (hidden) outputs h_j have unit variance (during the first step of backprop).



$$\text{Var}[h_j] = \text{Var}\left[g\left(\sum_i w_{ij} h_i\right)\right] = 1$$

Randomness:
prevents units from learning the same thing.

Normalization: preserves dynamic of the outputs and gradients at each layer:
$\rightarrow$ all units get updated with similar speed.

Normalized initialization

1. Normalize the inputs with about zero-mean and unit variance,
2. Initialize all the biases to zero,
3. For the weights w_{ij} of a unit j with N inputs and activation g:
   - **Xavier initialization** (Glorot & Bengio, 2010): for $g(a) = a$

$$\text{Var}\left[g\left(\sum_i w_{ij} h_i\right)\right] = \sum_i \sigma^2 \underbrace{\text{Var}\left[h_i\right]}_{=1} = 1 \quad \Rightarrow \quad \sigma = \frac{1}{\sqrt{N}}$$

   (Works also for $g(a) = \tanh(a)$.)

   - **He initialization** (He et al., 2015): for ReLU $\quad \rightarrow \quad \sigma = \sqrt{\frac{2}{N}}$

   (First time, a machine surpasses humans on ImageNet.)

   - **Kumar initialization** (Kumar, 2017): for any g differentiable at 0:

$$\sigma^2 = \frac{1}{N g'(0)^2 (1 + g(0)^2)}$$

Normalized initialization

Alternatives:

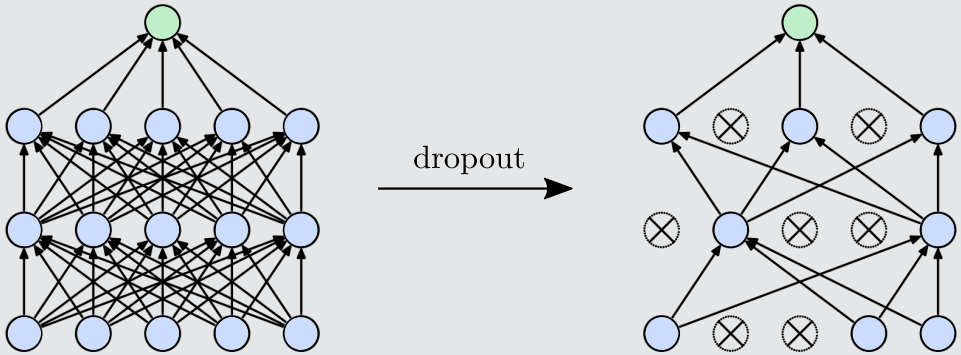
- Force weights of different units to be orthogonal (Saxe *et a.*, 2014)
  - Choose $\boldsymbol{W}$ as one of the orthonormal matrices obtained by taking the SVD of a matrix with random elements: $\boldsymbol{M} = \boldsymbol{U}\boldsymbol{S}\boldsymbol{V}^T$.
  - It promotes units to be specialized for different features.

- Mix orthogonality and equal variance principles (Mishkin & Matas, 2016)
  - Choose $\boldsymbol{W}$ as a random orthonormal matrix,
  - Normalize each unit by its variance estimated on mini-batches,
  - It forces units to be specialized in different tasks,
  - It makes sure the gradients do not vanish in some layers.

Dropout (Hinton *et al.*, 2012)

- **Goal:** Reduce over-fitting.
- **How:** Train smaller networks.

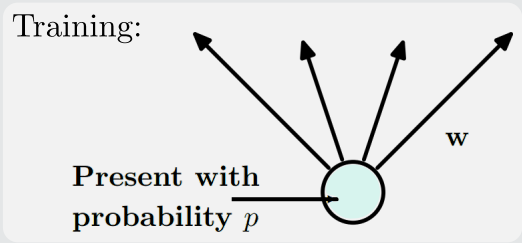
- Build a sub-network for each training sample,

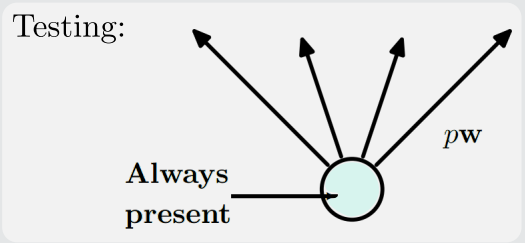


- Drop nodes with probability $1 - p$ or keep them with probability p,
  (in practice use a random binary mask to disable entries of $\boldsymbol{W}$).
- Usually, $p = .5$ for hidden layers, and $p \approx 1$ for the input.

Dropout (Hinton *et al.*, 2012)

- Use back-propagation for each data point and its sub-network,
- Note that each sub-network within a mini-batch shares common weights,
- At test time, dropout is disabled and unit outputs are multiplied by their corresponding p such that all units behave as if trained without dropout.





- Prevents units from co-adapting too much (depend on each other),
- Allows the network to learn more robust and specific features,
- Converges faster, and each epoch costs less.

Batch normalization (Ioffe and Szegedy, 2015)

- **Goal:** Learn faster, learn better.
- **How:** Normalize inputs of each hidden layer.

Internal Covariate Shift

- With backprop, the distribution of the inputs $\boldsymbol{h}$ of a (deep) hidden layer keeps changing during the iterations.
- This is painful as the layers need to continuously adapt to it, then
  - Requires low learning rates,
  - Careful parameter initialization,
  - Proper choices of activation functions,
  - Leads to slow convergence, and poor solutions.

Batch normalization (Ioffe and Szegedy, 2015)

Standardization

- Transform $\boldsymbol{a}$ (recall, $\boldsymbol{a} = \boldsymbol{W}\boldsymbol{h} + \boldsymbol{b}$) to satisfy
  - **Zero-mean:** remove arbitrary shift of your data,
  - **Unit variance:** remove arbitrary spread of your data,

- Standardization dates back to classical ML with hand-crafted features,
- Classically applied to the input features,
- Here features at each (hidden) layer at each epoch are normalized.

Batch normalization (Ioffe and Szegedy, 2015)

Idea: perform for each unit i of a given hidden layer

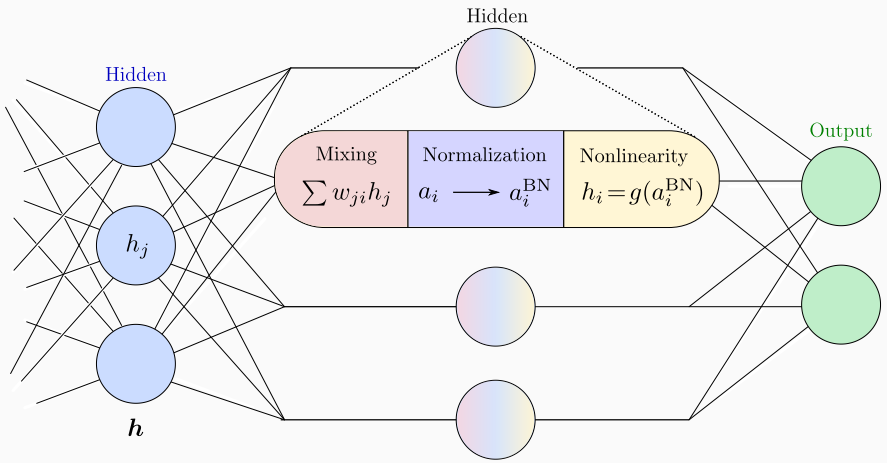
$$a_i^{\text{BN}} \leftarrow \beta_i + \alpha_i \underbrace{\frac{a_i - \mu_i}{\sqrt{\sigma_i^2 + \varepsilon}}}_{\text{standardized input}}$$

- μ_i, σ_i: mean and standard deviation of a_i over the current mini-batch, (obtained during the forward step)
- α_i, β_i: scalar parameters learned by backprop, (updated during the backward step)
- $\varepsilon > 0$: regularization to avoid instabilities.

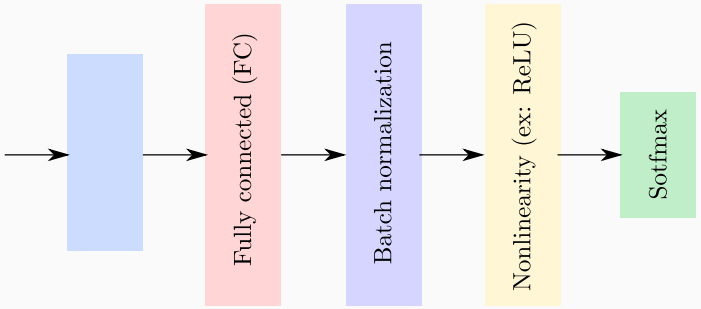
α_i and β_i introduced to preserve the network capacity.
(to make use of non-linearities, otherwise sigmoid $\approx$ linear)

During testing, replace μ_i and σ_i by an average of the last estimates obtained over mini-batches during training.

Batch normalization (Ioffe and Szegedy, 2015)



Old school representation

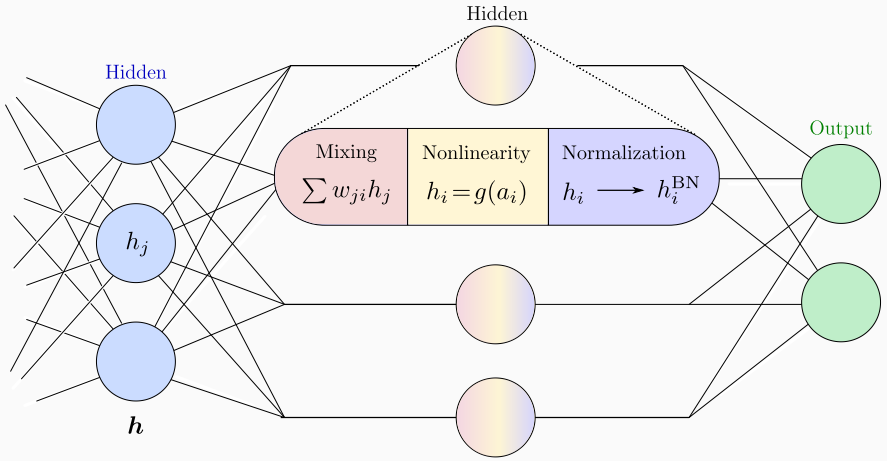


Modern representation

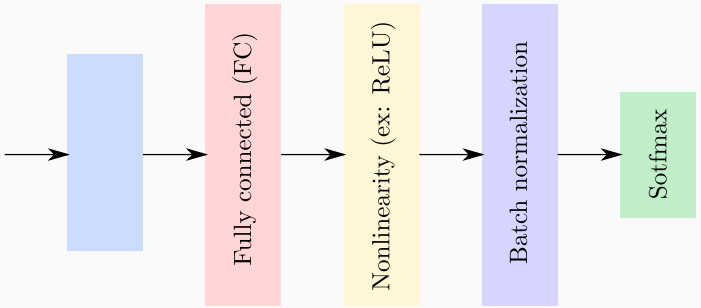
- α_i and β_i encode the range of inputs given to the activation function, (can be seen as parameters of the activation function)
- each layer learns a little bit more by itself independently from other layers,
- allows higher learning rates (larger gradient descent step size γ),
- reduce over-fitting – slight regularization effects.

Batch normalization – variant

- Some implementations normalized h_i instead of a_i,



Old school representation



Modern representation

- In this case α_i and β_i perform a kind of preconditioning,
- What strategy is best is still under debate.

Residual Networks (ResNets) (Microsoft – He *et al.*, 2015).

Up to 2015, deep was about ≈ 20 layers, and using more was harmful.

Goal: Get deeper and deeper while improving accuracy.

Idea: Learn $f : \boldsymbol{x} \to \underbrace{\boldsymbol{d} - \boldsymbol{x}}_{\text{residual}}$ instead of $h : \boldsymbol{x} \to \boldsymbol{d}$ (assuming same size), then give your prediction of $\boldsymbol{d}$ as $\boldsymbol{y} = f(\boldsymbol{x}) + \boldsymbol{x}$

- Tough to learn a mapping close to the identity with non-linear layers:
  - Consider a 1 ReLU unit:
    $$\begin{cases} \text{❶ } y = \underbrace{\max(0, wx + b)}_{h(x)} \\ \text{❷ } y = \underbrace{\max(0, wx + b)}_{f(x)} + x. \end{cases}$$
  - Learn $y = x$: trivial for ❷, impossible for ❶.
  - Of course, possible with more units but less trivial.
- Simpler to learn what the fluctuations around x should be.

Residual Networks (ResNets)

Example (Denoising $x = d + n$ with d the desired clean image)

- The residual is (minus) the noise component: $d - x = -n$,
- subtle difference between the input $x = d + n$ and output d
  $\rightarrow$ harder to learn.
- large difference between the input $x = d + n$ and the residual $-n$
  $\rightarrow$ easier to learn.

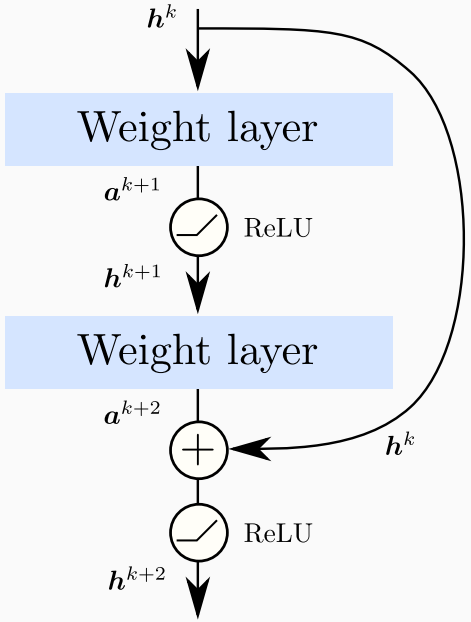
But why does it help me to be deeper?

Residual Networks (ResNets)

- Don't necessarily use this concept only for the input and output layers,
- Instead, use successions of hidden layers with same size,
- Use it every two hidden layers:

$$\begin{aligned} \boldsymbol{a}^{k+1} &= \boldsymbol{W}^k \boldsymbol{h}^k \\ \boldsymbol{h}^{k+1} &= g(\boldsymbol{a}^{k+1}) \\ \boldsymbol{a}^{k+2} &= \boldsymbol{W}^{k+1} \boldsymbol{h}^{k+1} \\ \boldsymbol{h}^{k+2} &= g(\boldsymbol{a}^{k+2} + \boldsymbol{h}^k) \end{aligned}$$

- Use shortcut connections:



- It does not require extra parameters,
- Does not add computational overhead,
- Acts as a kind of preconditioning
  $\longrightarrow$ allowing for huge accelerations,
- Led to a 152 deep NN that won the 2015 ImageNet competition,
  $\longrightarrow$ we will return to this next class.

Regularization

- **Goal:** Avoid over-fitting.
- **How:** Use prior knowledge.

Early stopping, dropout, batch normalization, *etc.*, implicitly promote some regularity. But we can also introduce regularity explicitly given prior knowledge on the problem/application context.

Remember: autoencoders extract relevant features by making use of

- Undercomplete representation (remove nodes):

  information can be compressed/encoded with a small dictionary,

- Overcomplete/sparse representation (penalize node's outputs):

  information can be encoded with a few atoms of a large dictionary.

In both cases, we have used prior knowledge that information must be structured/organized.

Regularization

In general, there exists a lot more of intrinsic regularity in the data that should be injected in the model (all the more for images).

For instance, as for the nodes, depending on the application, some connections can be removed (non-fully connected layer), or their weights can be penalized by adding a regularization term Ω in the loss function:

- $\Omega = \ell_1$ to promote sparsity (sparse layer),
- $\Omega = \ell_2$ to promote small coefficients (weight decay),
- ...

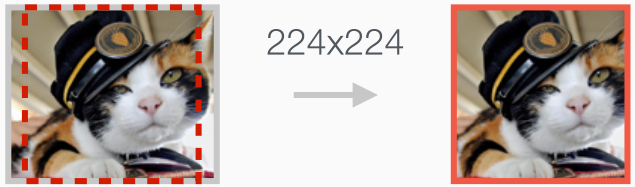
As we will see next class, Convolution Neural Networks (CNN) are a kind of highly constrained architecture very relevant for images.

Regularity is probably the most important ingredient allowing for deep learning.

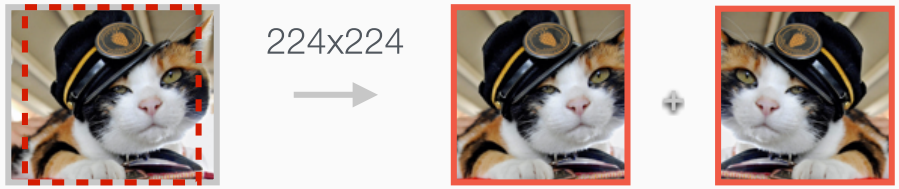
Data augmentation

Idea: generate artificially more data points from the training set itself.

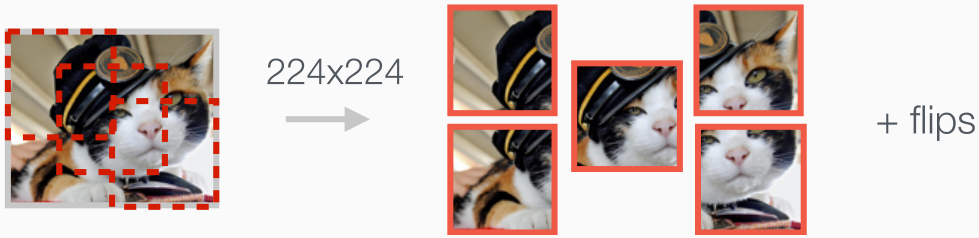
No augmentation (=1 image)



Flip augmentation (=2 images)



Crop + flip augmentation (=10 images)



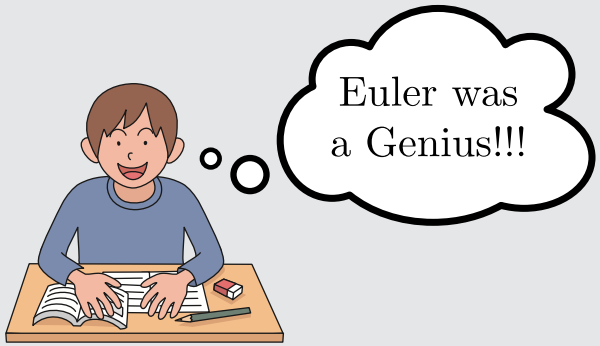
Others: rotations, zooms (rescalings), brightness, noise, ...

(Source: Ken Chatfield)

Curriculum learning (Bengio et al, 2009)

- **Convex criteria:** the order of presentation of samples should not matter to the convergence point, but could influence convergence speed.
- **Non-convex criteria:** the order and selection of samples could yield a better solution.

1. Integers
2. Real numbers
3. Complex numbers
4. $e^{i\pi} = -1$

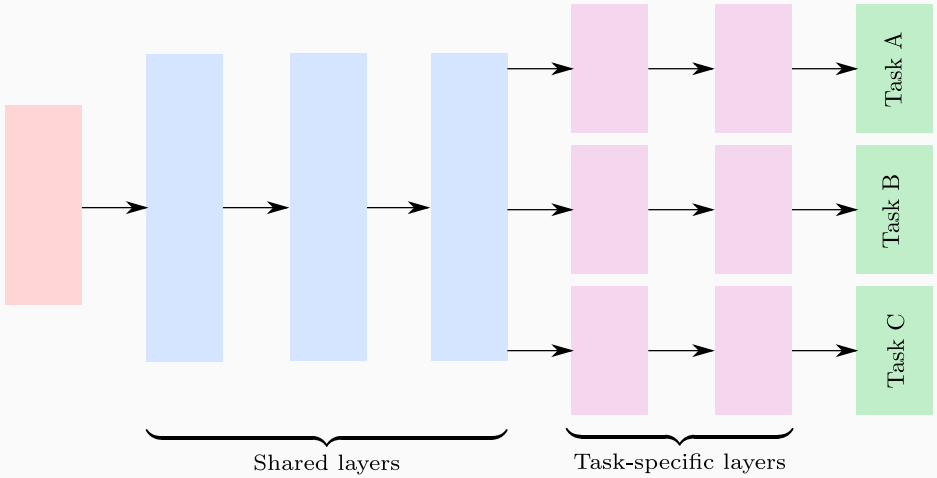


(Inspiration: Pawan Kumar)

- Start learning on easy samples, and increase the difficulty step by step.
- Improves but requires to sort the training samples by difficulty.
- When you cannot or do not know how, shuffling randomly is safer.

Multitask learning (Caruana, 1997)

- Learning jointly how to solve several tasks improves performance,
- Effect of regularization (avoid over-fitting),
- Useful in context of insufficiently labeled data.



Share first hidden layers, use specific ones last.

Related to:

- **unsupervised pretraining** (Hinton, 2006): train first in an unsupervised way, fine-tune to solve a given task.
- **transfer learning** (Pratt, 1993, 1997): train first on a given large dataset (detect red cars), fine-tune to a smaller one (detect tumors).

Open source solutions

Most popular

- **TensorFlow** – Google Brain – Python, C/C++
- **Torch** – Facebook – Lua, Python
- **Theano** – University of Montreal – Python (ended Nov 2017)
- **Keras** high level interface for TensorFlow, Theano
- **Caffe** – Berkeley Vision and Learning Center - Python, MATLAB
- **MxNet** – Apache – Python, Matlab, Julia, C++, ...
- **MatConvNet** – University of Oxford – Matlab

Main features

- Implementations of most classical DL models, tools, recipes...
- Pretrained models,
- GPU/CUDA support,
- Flexible interfaces,
- Automatic differentiation.

(Multi) GPUs architectures

- Neural Networks are inherently parallel,
- GPUs are good at matrix/tensor operations,
- Much faster than CPU, lower power, lower cost,
- New architectures are optimized for deep learning:

NVIDIA Tesla V100: World's first GPU to break the 100 teraflops barrier of deep learning performance. New feature: 640 tensor cores. Relatively cheap ($\approx$ \$8,000). Released June 2017.

- Relatively easy to manipulate thanks to CUDA,
- Many deep learning toolboxes have GPU support,
- Can be integrated in clusters/datacenters for distributed computing.

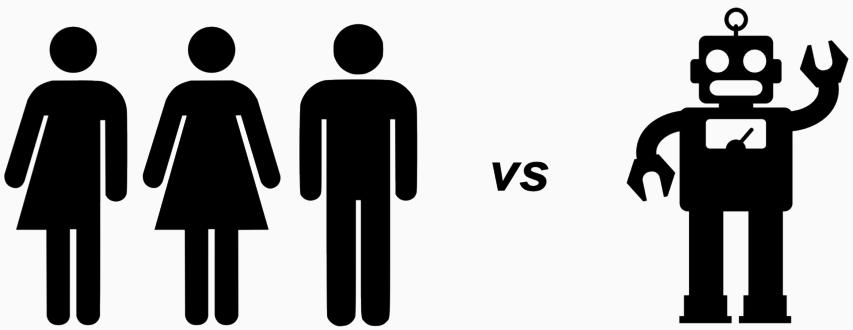
Best practices

- Check/clean your data: remove corrupted ones, perform normalization, ...
- Shuffle the training samples,
- Split your data into training (70%) and testing (30%),
- Use early stopping: take another 30% of training for validation,
- Never train on test data,
- Start with an existing network and adapt it to your problem,
- Start smallish, keep adding layers and nodes until you overfit too much,
- Check that you can achieve zero loss on a tiny subset (20 examples),
- Track the loss during training,
- Track the first-layer features,
- Try the successive rates: $\gamma = 0.3, 0.1, 0.03, 0.01, 0.003, 0.001$.

Recommended reading:

`https://karpathy.github.io/2019/04/25/recipe/`

Weaknesses and criticisms



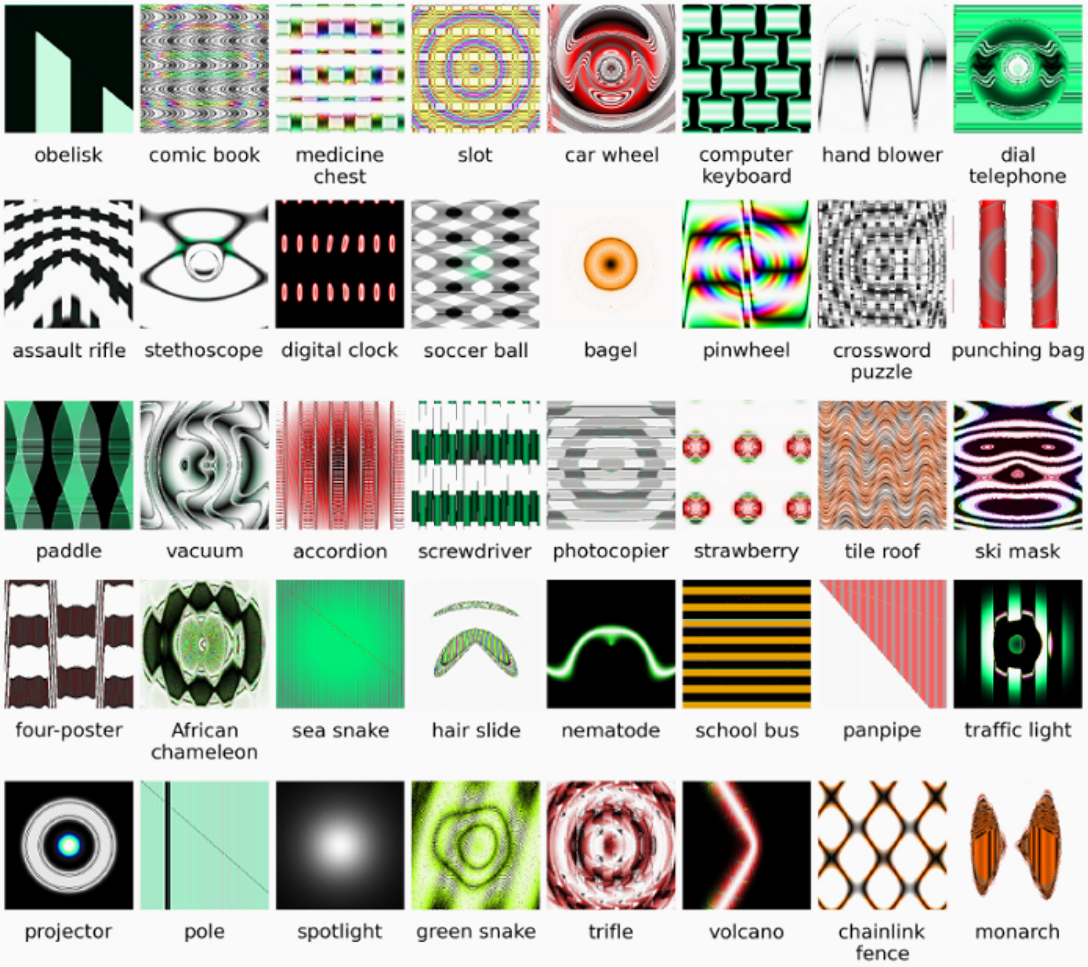
Differences from the brain

- The way artificial neurons fire is fundamentally different than biological neurons.
- A human brain has 100 billion neurons and 100 trillion connections (synapses) and operates on 20 watts (enough to run a dim light bulb) - in comparison the 2012 Google Brain project had 10 million neurons and 1 billion connections on 16,000 CPUs (about 3 million watts).
- The brain is limited to 5 types of input data from the 5 senses.
- Children do not learn what a cow is by reviewing 100,000 pictures labeled "cow" and "not cow", but this is how deep learning works.

(Source: Devin Didericksen)

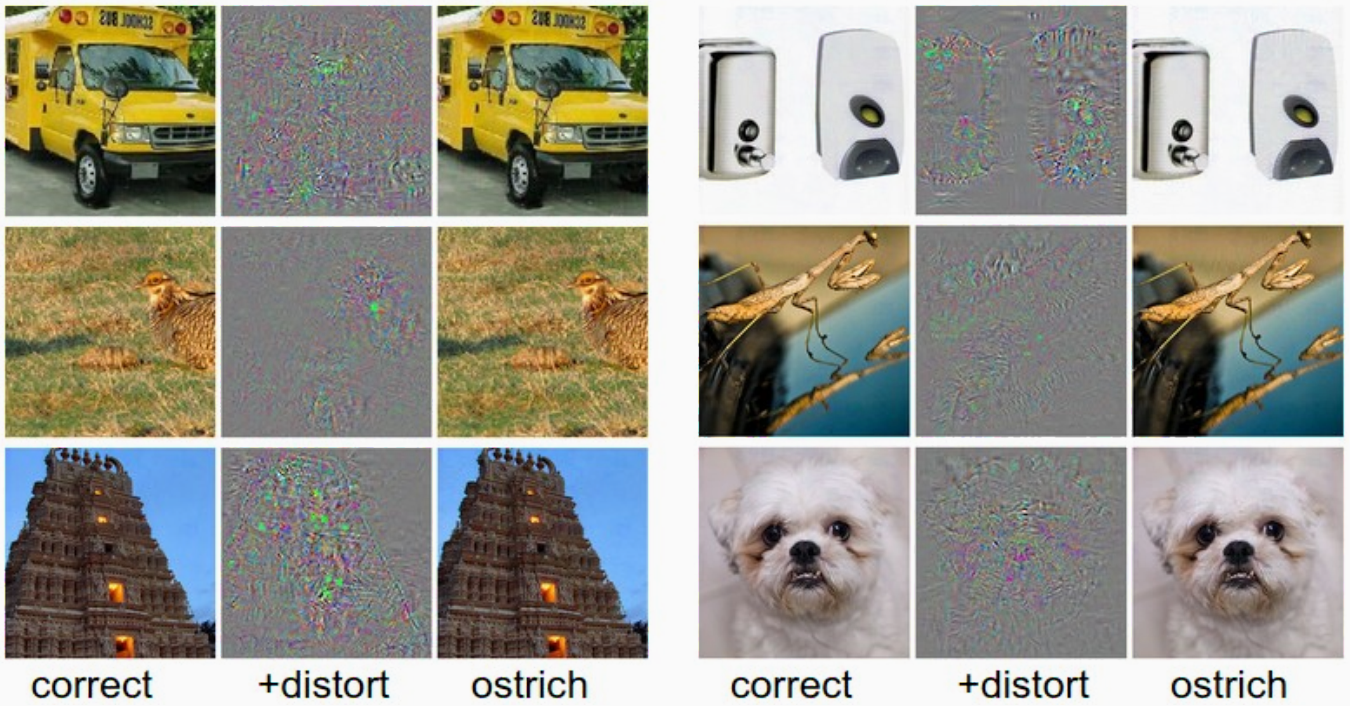
Weaknesses and Criticisms

Deep neural networks are easily fooled (Nguyen, CVPR, *et al*, 2014)



- Images are unrecognizable by humans,
- System has 99.1% certainty that the image is that type of thing,
- Unclear what it learns, clearly it does not organize concepts as we do.

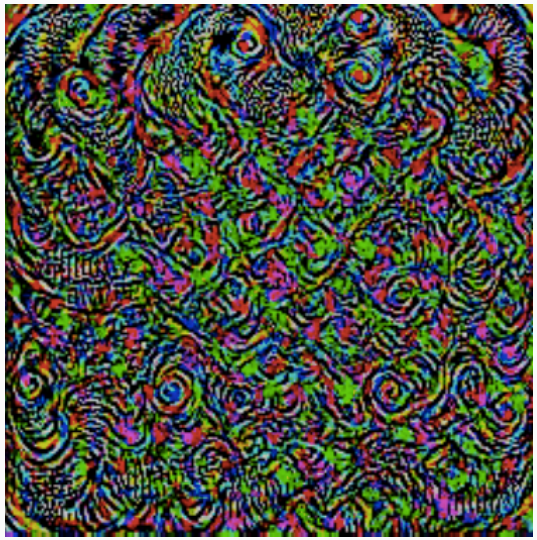
Intriguing properties of neural networks (Szegedy *et al.*, 2013)



For each sample, there exists a small (non random) perturbation that can completely fool the network, even though humans are not fooled.

Weaknesses and criticisms

Universal Adversarial Perturbations (Moosavi-Dezfooli et al., ICCV, 2017)



Universal perturbation $\boldsymbol{r}$ for GoogLeNet

Indian elephant

Macaw

Look for a small perturbation $\boldsymbol{r}$ such that

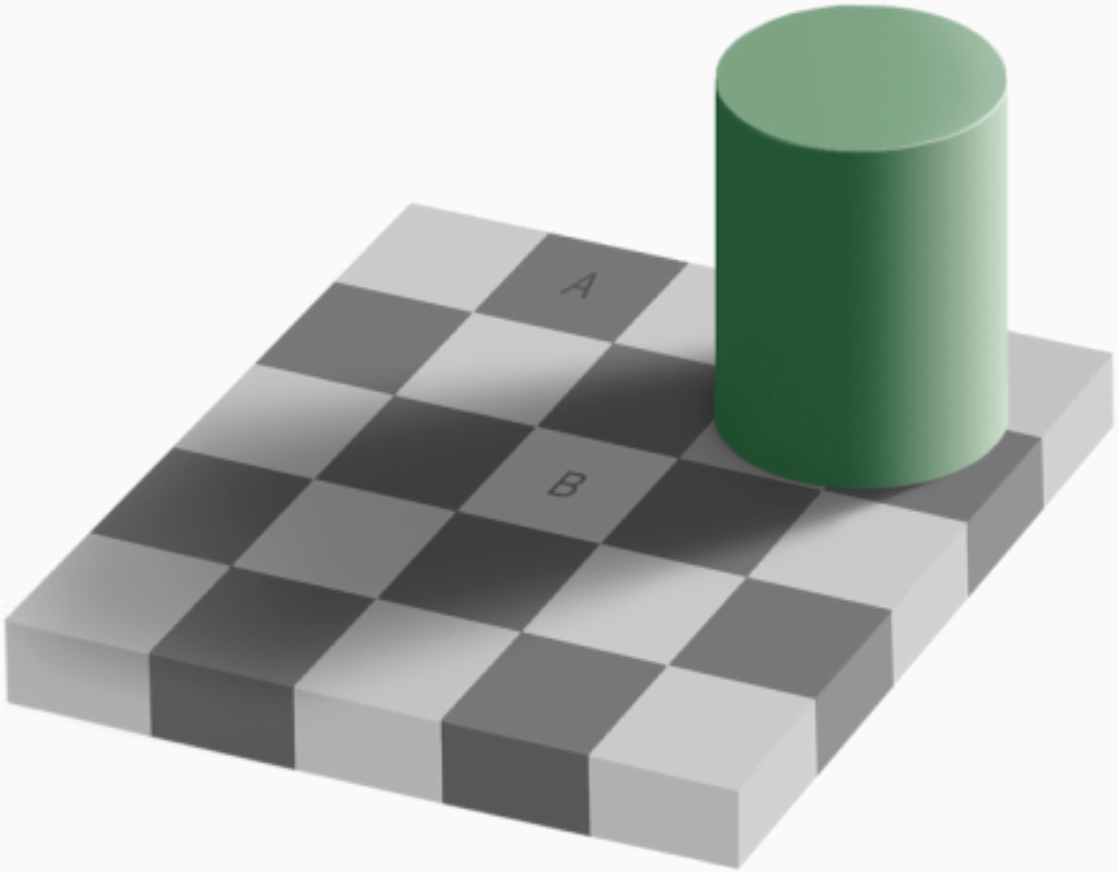
$$\underset{\boldsymbol{r}}{\operatorname{argmin}} \|\boldsymbol{r}\|_2 \quad \text{subject to} \quad \forall (\boldsymbol{x}, d) \in \mathcal{T}, \quad y = f(\boldsymbol{x} + \boldsymbol{r}) \neq d$$

Worst, it can be the same perturbation for all training samples.

$\longrightarrow$ raises serious security issues in critical applications.

Humans can be fooled too

Which square is darker? A or B?



Overview

Pros & Cons

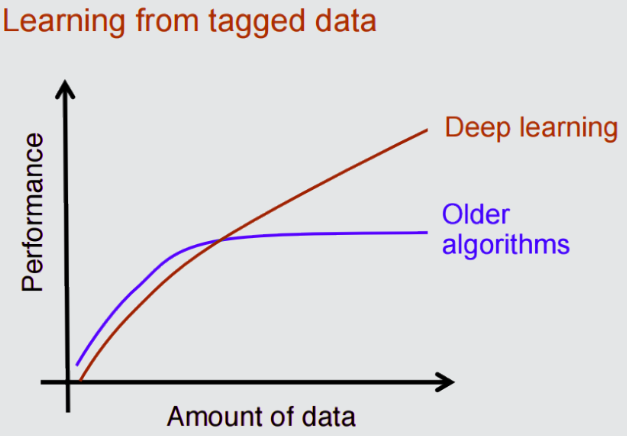
- Best performing method in many Computer Vision tasks,
- No need for hand-crafted features,
- Robustness to natural variations in the data is automatically learned,
- Can be reused for many different applications and data types,
- Applicable to large-scale problems, e.g., classification with 1000 classes,
- Performance improves with more data,
- Easy parallelization on GPUs.

- Need of huge amount of training data,
- Hard to design and tune,
- Difficult to analyze and understand: many features are not really human-understandable,
- Other methods like SVMs are easily deployed even by novices and can usually get you good results,
- Tend to learn everything. It's better to encode prior knowledge about structure of images (or audio or text) if large datasets are not available.

(Source: Caner Hazırbaş)

When to use Deep Learning?

- You have a large amount of data with good quality,



- You are modeling image/audio/language/time-series data,
- Your data contains high-level abstractions:

  Excels in tasks where the basic unit (pixel, word) has very little meaning in itself, but their combination has a useful meaning.

- You need a model that is less reliant on handmade features, and instead can learn important features from the data.

(Source: Lucas Masuch, Andrew Ng)

Questions?

Next class: Image classification and CNNs

Sources, images courtesy and acknowledgment

K. Chatfield
P. Gallinari,
C. Hazırbaş
A. Horodniceanu
Y. LeCun
V. Lepetit
L. Masuch
A. Ng
M. Ranzato