Introduction to Deep Learning a.k.a. “Neural” Networks

Mário A. T. Figueiredo

(based on slides also by André Martins and others)

15th Lisbon Machine Learning Summer School, LxMLS 2025

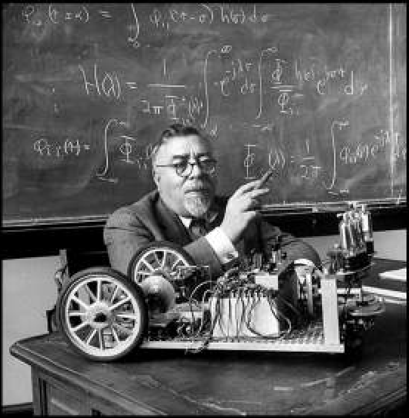
Outline

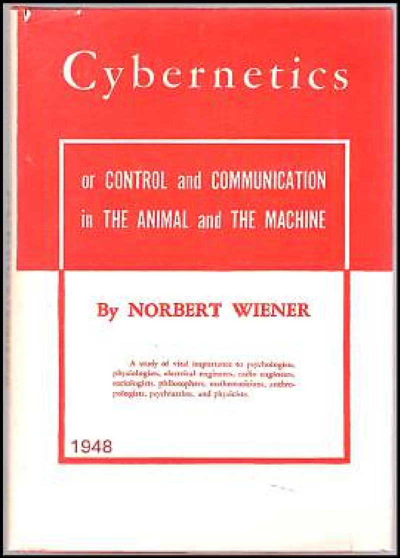
1. **Brief History of Deep Learning (Before LLMs)**
2. **From models of neurons to artificial neural networks**
3. **Deep Learning via Empirical Risk Minimization**
   - Gradient Descent and Stochastic Gradient Descent
   - Gradient Backpropagation and Autodiff
   - Better optimization: momentum, AdaGrad, RMSProp, Adam
4. **Convolutional Neural Networks**

Outline

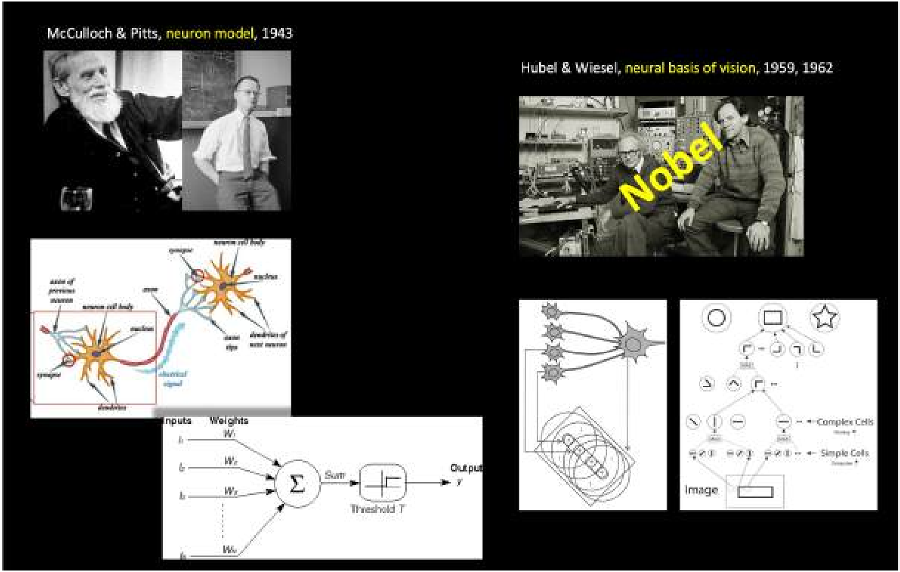
1. **Brief History of Deep Learning (Before LLMs)**
2. **From models of neurons to artificial neural networks**
3. **Deep Learning via Empirical Risk Minimization**
   - Gradient Descent and Stochastic Gradient Descent
   - Gradient Backpropagation and Autodiff
   - Better optimization: momentum, AdaGrad, RMSProp, Adam
4. **Convolutional Neural Networks**

Deep roots

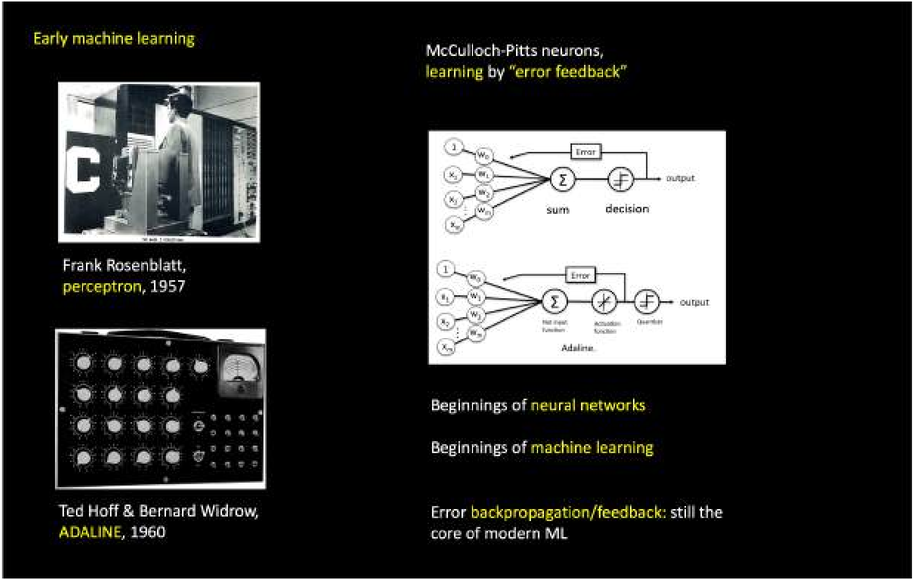



Cybernetics
or CONTROL and COMMUNICATION
in THE ANIMAL and THE MACHINE
By NORBERT WIENER
1948

Early work on neural networks

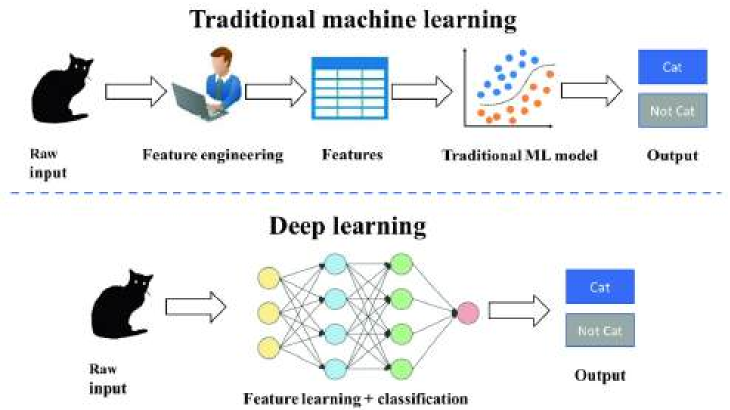


Early machine learning: the Perceptron

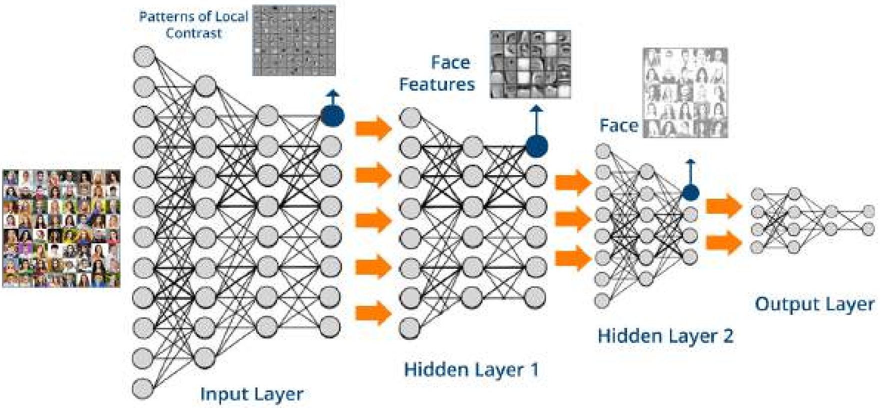


Four decades of evolution

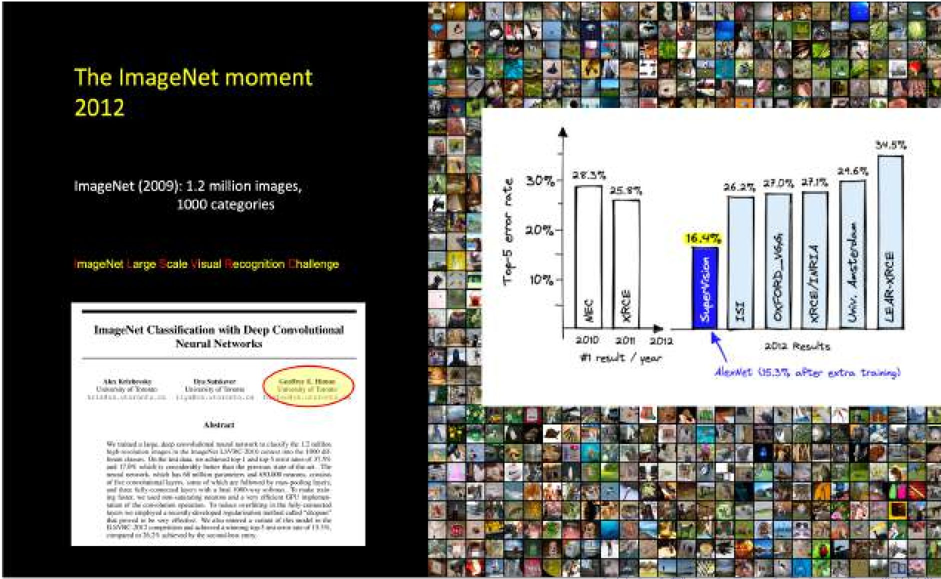
End-to-end learning



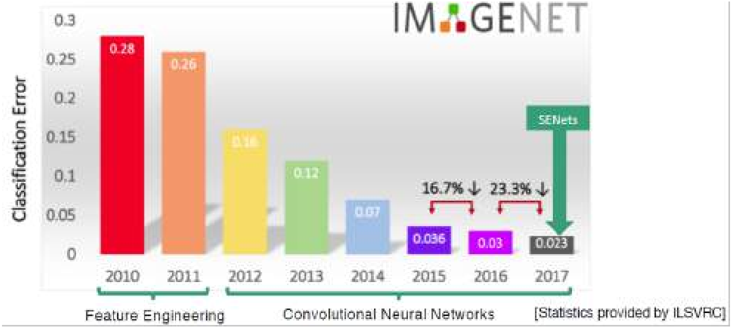
Deep networks: hierarchy of features



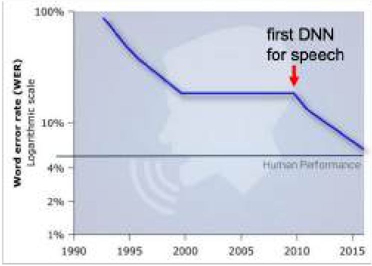
The ImageNet moment

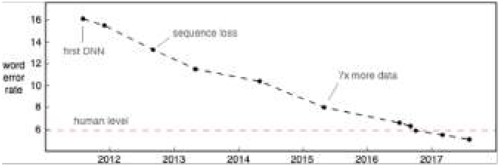


The following years

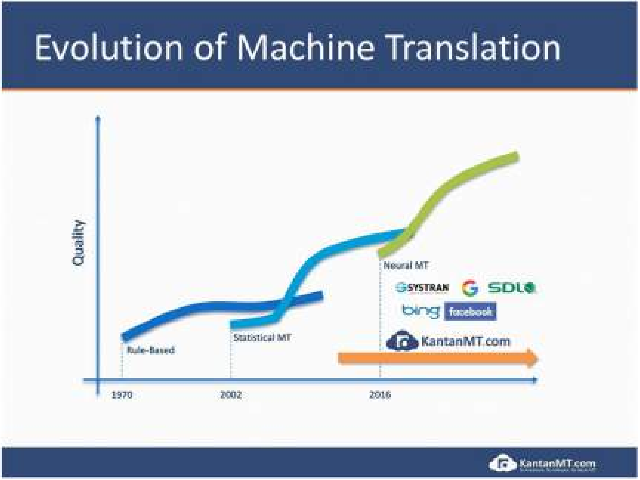


Also in speech recognition...

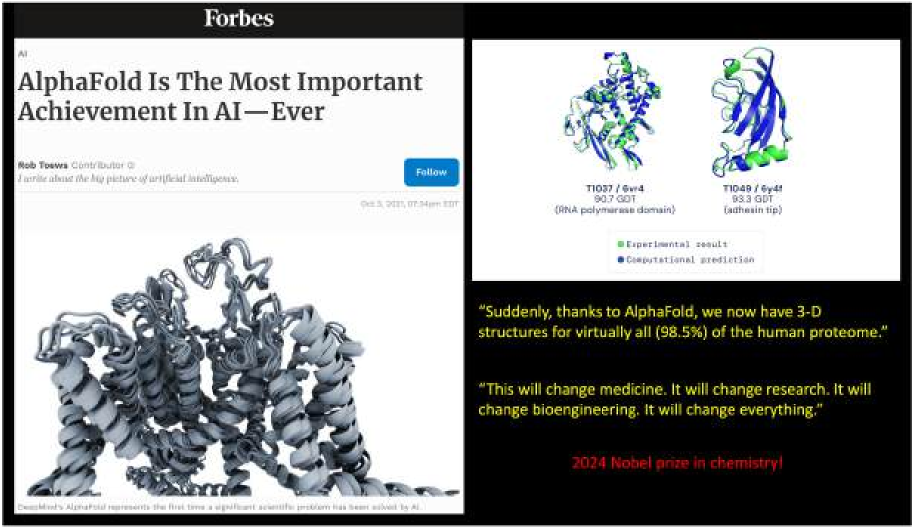




... machine translation,

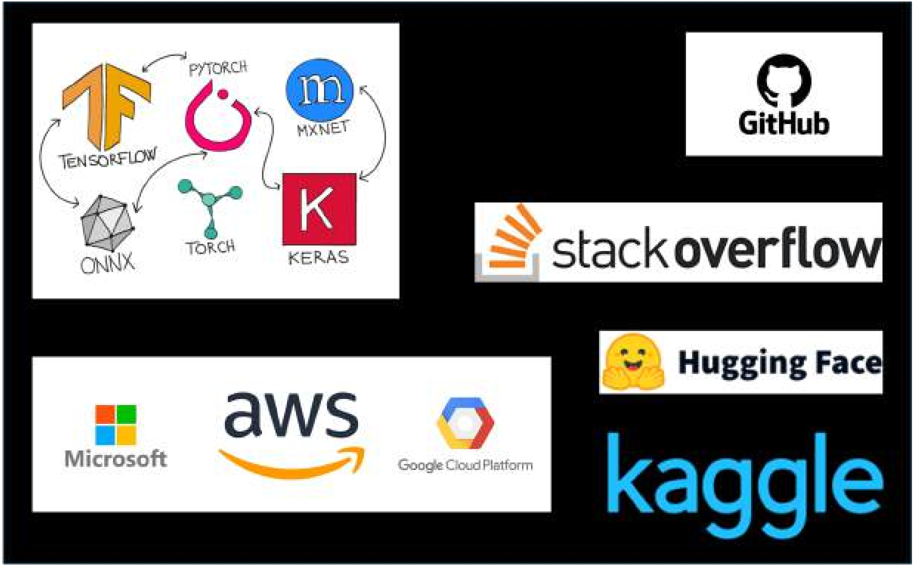


... and biology



Forbes
AI
AlphaFold Is The Most Important Achievement In AI—Ever
Rob Toews Contributor
I write about the big picture of artificial intelligence.
Follow
T1037 / 6vr4
90.7 GDT
(RNA polymerase domain)
T1049 / 6y4f
93.3 GDT
(adhesin tip)
Experimental result
Computational prediction
"Suddenly, thanks to AlphaFold, we now have 3-D structures for virtually all (98.5%) of the human proteome."
"This will change medicine. It will change research. It will change bioengineering. It will change everything."
2024 Nobel prize in chemistry!

Why now? Frictionless reproducibility (Donoho, 2023)



Outline

Neuron model (McCulloch and Pitts, 1943)

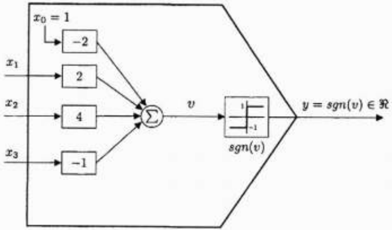


Figure 3.6 Example 3.2: a threshold neural logic for $y = x_2(x_1 + \bar{x}_3)$.

Table 3.6 Truth table for Example 3.2

Neural Inputs x_1	x_2	x_3	$v = w_a^T x_a = -2 + 2x_1 + 4x_2 - x_3$	$y = sgn(v) = sgn(w_a^T x_a)$
−1	−1	−1	−7	−1
−1	−1	1	−9	−1
−1	1	−1	1	1
−1	1	1	−1	−1
1	−1	−1	−3	−1
1	−1	1	−5	−1
1	1	−1	5	1
1	1	1	3	1

Neuron model (McCulloch and Pitts, 1943)

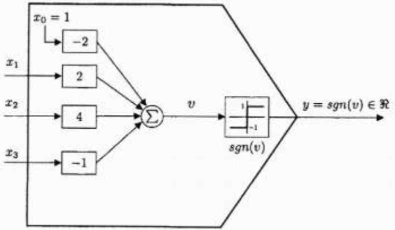


Figure 3.6 Example 3.2: a threshold neural logic for $y = x_2(x_1 + \bar{x}_3)$.

Table 3.6 **Truth table for Example 3.2**

Neural Inputs x_1	x_2	x_3	$v = \boldsymbol{w}_a^T \boldsymbol{x}_a = -2 + 2x_1 + 4x_2 - x_3$	$y = sgn(v) = sgn(\boldsymbol{w}_a^T \boldsymbol{x}_a)$
−1	−1	−1	−7	−1
−1	−1	1	−9	−1
−1	1	−1	1	1
−1	1	1	−1	−1
1	−1	−1	−3	−1
1	−1	1	−5	−1
1	1	−1	5	1
1	1	1	3	1

- Biological neurons are hugely more complex.

Neuron model (McCulloch and Pitts, 1943)

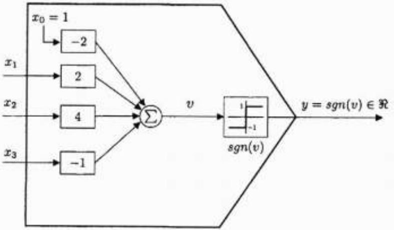


Figure 3.6 Example 3.2: a threshold neural logic for $y = x_2(x_1 + \bar{x}_3)$.

Table 3.6 **Truth table for Example 3.2**

Neural Inputs x_1	x_2	x_3	$v = \boldsymbol{w}_a^T \boldsymbol{x}_a = -2 + 2x_1 + 4x_2 - x_3$	$y = sgn(v) = sgn(\boldsymbol{w}_a^T \boldsymbol{x}_a)$
−1	−1	−1	−7	−1
−1	−1	1	−9	−1
−1	1	−1	1	1
−1	1	1	−1	−1
1	−1	−1	−3	−1
1	−1	1	−5	−1
1	1	−1	5	1
1	1	1	3	1

- Biological neurons are hugely more complex.
- Later models replaced the hard threshold by more general activation

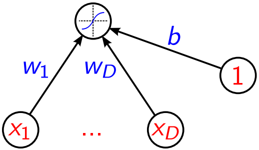
Artificial neuron

- **Pre-activation** (input activation):

$$z(x) = w^T x + b = \sum_{i=1}^{D} w_i x_i + b,$$

w: connection weights

b: bias

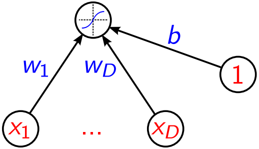


Artificial neuron

- **Pre-activation** (input activation):

$$z(x) = w^T x + b = \sum_{i=1}^{D} w_i x_i + b,$$

w: connection weights
b: bias



- **Activation:**

$$h(x) = g(z(x)) = g(w^T x + b),$$

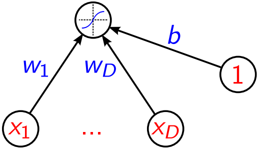
where $g : \mathbb{R} \to \mathbb{R}$ is the activation function.

Artificial neuron

- **Pre-activation** (input activation):

$$z(x) = w^T x + b = \sum_{i=1}^{D} w_i x_i + b,$$

w: connection weights
b: bias



- **Activation:**

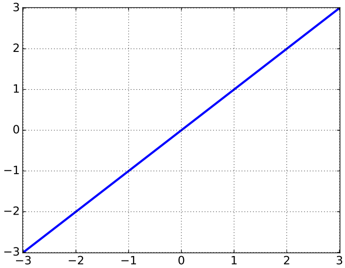
$$h(x) = g(z(x)) = g(w^T x + b),$$

where $g : \mathbb{R} \to \mathbb{R}$ is the activation function.

- Typical activation functions (next): linear (identity); sigmoid (logistic function); hyperbolic tangent (tanh); rectified linear unit (ReLU).

Linear activation

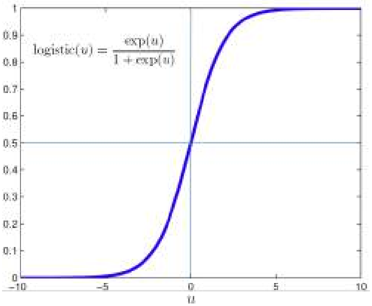
$$\boxed{g(z) = z}$$



- No “squashing” of the input.
- Composing linear layers is equivalent to a single linear layer: no expressive power increase by using multiple layers (but...).

Sigmoid activation

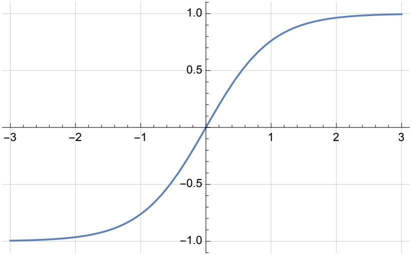
$$g(z) = \sigma(z) = \frac{e^z}{1 + e^z}$$



- Output in $[0, 1]$, can be interpreted as a probability.
- Positive, bounded, strictly increasing.
- Logistic regression corresponds to a network with a single sigmoid unit.
- Combining layers of sigmoid units increases expressiveness (more later).

Hyperbolic tangent activation

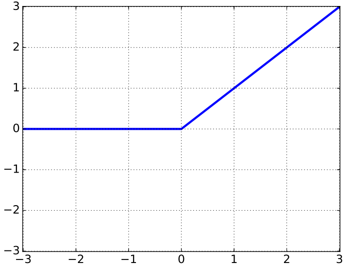
$$g(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$



- "Squashes" the neuron pre-activation to $[-1, +1]$.
- Related to the sigmoid via $\sigma(z) = \frac{1+\tanh(z/2)}{2}$.
- Bounded, strictly increasing.
- Combining layers of tanh units increases expressiveness (more later).

Rectified linear unit

$$\boxed{g(z) = \text{relu}(z) = \max\{0, z\}}$$



- Non-negative, increasing, but not upper bounded.
- Not differentiable at 0.
- Leads to neurons with sparse activities (arguably closer to biology).
- Very cheap to compute.

Multi-layer network

- Key idea: use intermediate (hidden) layers between input and output.

Multi-layer network

- Key idea: use intermediate (hidden) layers between input and output.

- Each hidden layer computes a representation of the input and propagates it forward.

Multi-layer network

- Key idea: use intermediate (hidden) layers between input and output.
- Each hidden layer computes a representation of the input and propagates it forward.
- This increases the expressive power of the network, yielding more complex, non-linear, functions/classifiers

Multi-layer network

- Key idea: use intermediate (hidden) layers between input and output.
- Each hidden layer computes a representation of the input and propagates it forward.
- This increases the expressive power of the network, yielding more complex, non-linear, functions/classifiers
- Also called feed-forward "neural" network

Multi-layer network

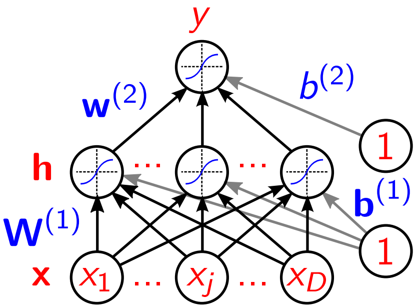
- Key idea: use intermediate (hidden) layers between input and output.
- Each hidden layer computes a representation of the input and propagates it forward.
- This increases the expressive power of the network, yielding more complex, non-linear, functions/classifiers
- Also called feed-forward "neural" network
- Learning the parameters is much harder than in linear models.

Single hidden layer

- Starting simple:
  - ✓ several inputs ($\boldsymbol{x} \in \mathbb{R}^D$);
  - ✓ single output (e.g. $y \in \mathbb{R}$ or $y \in [0, 1]$)

Single hidden layer

- Starting simple:
  - ✓ several inputs ($\boldsymbol{x} \in \mathbb{R}^D$);
  - ✓ single output (e.g. $y \in \mathbb{R}$ or $y \in [0, 1]$)
- Intermediate, hidden, layer of K hidden units ($\boldsymbol{h} \in \mathbb{R}^K$)

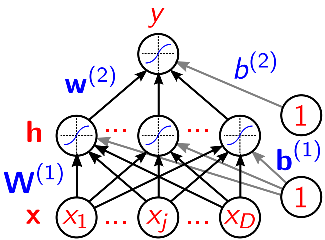


Single hidden layer

- **Hidden layer pre-activation:**

$$\boldsymbol{z}(\boldsymbol{x}) = \boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)},$$

with $\boldsymbol{W}^{(1)} \in \mathbb{R}^{K\times D}$ and $\boldsymbol{b}^{(1)} \in \mathbb{R}^{K}$.



Single hidden layer

- **Hidden layer pre-activation:**

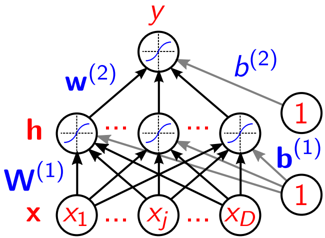
$$\boldsymbol{z}(\boldsymbol{x}) = \boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)},$$

with $\boldsymbol{W}^{(1)} \in \mathbb{R}^{K\times D}$ and $\boldsymbol{b}^{(1)} \in \mathbb{R}^{K}$.

- **Hidden layer activation:**

$$\boldsymbol{h}(\boldsymbol{x}) = \boldsymbol{g}(\boldsymbol{z}(\boldsymbol{x})),$$

where $\boldsymbol{g} : \mathbb{R}^K \rightarrow \mathbb{R}^K$ is applied component-by-component.



Single hidden layer

- **Hidden layer pre-activation:**

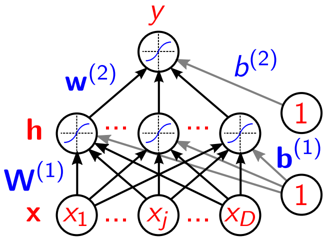
$$\boldsymbol{z}(\boldsymbol{x}) = \boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)},$$

  with $\boldsymbol{W}^{(1)} \in \mathbb{R}^{K\times D}$ and $\boldsymbol{b}^{(1)} \in \mathbb{R}^{K}$.

- **Hidden layer activation:**

$$\boldsymbol{h}(\boldsymbol{x}) = \boldsymbol{g}(\boldsymbol{z}(\boldsymbol{x})),$$

  where $\boldsymbol{g} : \mathbb{R}^K \to \mathbb{R}^K$ is applied component-by-component.



- **Output layer activation:** $f(\boldsymbol{x}) = o(\boldsymbol{h}(\boldsymbol{x})^T\boldsymbol{w}^{(2)} + b^{(2)})$, where $\boldsymbol{w}^{(2)} \in \mathbb{R}^K$ and $o : \mathbb{R} \to \mathbb{R}$ is the output activation function.

Single hidden layer, single output

- Overall,

$$\begin{aligned} f(\boldsymbol{x}) &= o(\boldsymbol{h}(\boldsymbol{x})^T \boldsymbol{w}^{(2)} + b^{(2)}) \\ &= o({\boldsymbol{w}^{(2)}}^T \boldsymbol{g}(\boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)}) + b^{(2)}) \end{aligned}$$

Single hidden layer, single output

- Overall,

$$\begin{aligned} f(\boldsymbol{x}) &= o(\boldsymbol{h}(\boldsymbol{x})^T \boldsymbol{w}^{(2)} + b^{(2)}) \\ &= o({\boldsymbol{w}^{(2)}}^T \boldsymbol{g}(\boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)}) + b^{(2)}) \end{aligned}$$

- Examples:
  - ✓ $o(u) = u$, for regression ($y \in \mathbb{R}$)
  - ✓ $o(u) = \sigma(u)$ for binary classification ($y \in \{\pm 1\}$, $f(\boldsymbol{x}) = \mathbb{P}(y = 1 \mid \boldsymbol{x})$)

Single hidden layer, single output

- Overall,

$$\begin{aligned} f(\boldsymbol{x}) &= o(\boldsymbol{h}(\boldsymbol{x})^T \boldsymbol{w}^{(2)} + b^{(2)}) \\ &= o({\boldsymbol{w}^{(2)}}^T \boldsymbol{g}(\boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)}) + b^{(2)}) \end{aligned}$$

- Examples:
  - ✓ $o(u) = u$, for regression ($y \in \mathbb{R}$)
  - ✓ $o(u) = \sigma(u)$ for binary classification ($y \in \{\pm 1\}$, $f(\boldsymbol{x}) = \mathbb{P}(y = 1 \mid \boldsymbol{x})$)

Single hidden layer, single output

- Overall,

$$\begin{aligned} f(\boldsymbol{x}) &= o(\boldsymbol{h}(\boldsymbol{x})^T \boldsymbol{w}^{(2)} + b^{(2)}) \\ &= o({\boldsymbol{w}^{(2)}}^T \boldsymbol{g}(\boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)}) + b^{(2)}) \end{aligned}$$

- Examples:
  - ✓ $o(u) = u$, for regression ($y \in \mathbb{R}$)
  - ✓ $o(u) = \sigma(u)$ for binary classification ($y \in \{\pm 1\}$, $f(\boldsymbol{x}) = \mathbb{P}(y = 1 \mid \boldsymbol{x})$)

- Non-linear in $\boldsymbol{x}$ and non-linear in $\boldsymbol{W}^{(1)}$ and $\boldsymbol{b}^{(1)}$

Single hidden layer, single output

- Overall,

$$\begin{aligned} f(\boldsymbol{x}) &= o(\boldsymbol{h}(\boldsymbol{x})^T \boldsymbol{w}^{(2)} + b^{(2)}) \\ &= o({\boldsymbol{w}^{(2)}}^T \boldsymbol{g}(\boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)}) + b^{(2)}) \end{aligned}$$

- Examples:
  - ✓ $o(u) = u$, for regression ($y \in \mathbb{R}$)
  - ✓ $o(u) = \sigma(u)$ for binary classification ($y \in \{\pm 1\}$, $f(\boldsymbol{x}) = \mathbb{P}(y = 1 \mid \boldsymbol{x})$)

- Non-linear in $\boldsymbol{x}$ and non-linear in $\boldsymbol{W}^{(1)}$ and $\boldsymbol{b}^{(1)}$

- $\boldsymbol{h}(\boldsymbol{x})$ is a learned internal representation (not manually engineered)

Single hidden layer, multiple outputs

- Overall,

$$\begin{aligned} f(\boldsymbol{x}) &= \boldsymbol{o}(\boldsymbol{h}(\boldsymbol{x})^T \boldsymbol{w}^{(2)} + b^{(2)}) \\ &= \boldsymbol{o}({\boldsymbol{w}^{(2)}}^T \boldsymbol{g}(\boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)}) + b^{(2)}) \end{aligned}$$

Single hidden layer, multiple outputs

- Overall,

$$\begin{aligned} f(\boldsymbol{x}) &= \boldsymbol{o}(\boldsymbol{h}(\boldsymbol{x})^T \boldsymbol{w}^{(2)} + b^{(2)}) \\ &= \boldsymbol{o}({\boldsymbol{w}^{(2)}}^T \boldsymbol{g}(\boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)}) + b^{(2)}) \end{aligned}$$

- Examples:
  - ✓ $\boldsymbol{o}(\boldsymbol{u}) = \boldsymbol{o}$, for multiple regression ($y \in \mathbb{R}$)
  - ✓ $\boldsymbol{o}(\boldsymbol{u}) = \mathbf{softmax}(\boldsymbol{u})$ for classification (with C classes)

$$\mathbf{softmax}(\boldsymbol{u}) = \left[\frac{\exp(u_1)}{\sum_c \exp(u_c)}, \ldots, \frac{\exp(u_C)}{\sum_c \exp(u_c)}\right]$$

Single hidden layer, multiple outputs

- Overall,

$$\begin{aligned} f(\boldsymbol{x}) &= \boldsymbol{o}(\boldsymbol{h}(\boldsymbol{x})^T \boldsymbol{w}^{(2)} + b^{(2)}) \\ &= \boldsymbol{o}({\boldsymbol{w}^{(2)}}^T \boldsymbol{g}(\boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)}) + b^{(2)}) \end{aligned}$$

- Examples:
  - ✓ $\boldsymbol{o}(\boldsymbol{u}) = \boldsymbol{o}$, for multiple regression ($y \in \mathbb{R}$)
  - ✓ $\boldsymbol{o}(\boldsymbol{u}) = \textbf{softmax}(\boldsymbol{u})$ for classification (with C classes)

$$\textbf{softmax}(\boldsymbol{u}) = \left[\frac{\exp(u_1)}{\sum_c \exp(u_c)}, \ldots, \frac{\exp(u_C)}{\sum_c \exp(u_c)}\right]$$

- Non-linear in $\boldsymbol{x}$ and non-linear in $\boldsymbol{W}^{(1)}$ and $\boldsymbol{b}^{(1)}$

Single hidden layer, multiple outputs

- Overall,

$$\begin{aligned} f(\boldsymbol{x}) &= \boldsymbol{o}(\boldsymbol{h}(\boldsymbol{x})^T \boldsymbol{w}^{(2)} + b^{(2)}) \\ &= \boldsymbol{o}({\boldsymbol{w}^{(2)}}^T \boldsymbol{g}(\boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)}) + b^{(2)}) \end{aligned}$$

- Examples:
  - ✓ $\boldsymbol{o}(\boldsymbol{u}) = \boldsymbol{o}$, for multiple regression ($y \in \mathbb{R}$)
  - ✓ $\boldsymbol{o}(\boldsymbol{u}) = \mathbf{softmax}(\boldsymbol{u})$ for classification (with C classes)

$$\mathbf{softmax}(\boldsymbol{u}) = \left[\frac{\exp(u_1)}{\sum_c \exp(u_c)}, \ldots, \frac{\exp(u_C)}{\sum_c \exp(u_c)}\right]$$

- Non-linear in $\boldsymbol{x}$ and non-linear in $\boldsymbol{W}^{(1)}$ and $\boldsymbol{b}^{(1)}$

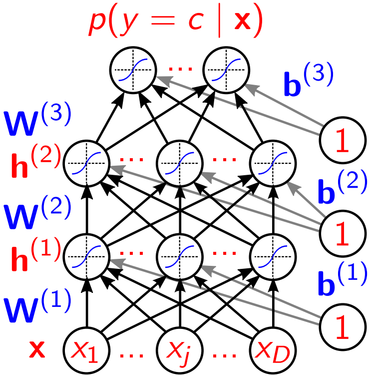
- $\boldsymbol{h}(\boldsymbol{x})$ is a learned internal representation (not manually engineered)

Multiple ($L \geq 1$) hidden layers

- **Hidden layer pre-activation** (define $\boldsymbol{h}^{(0)} = \boldsymbol{x}$ for convenience):

$$\boldsymbol{z}^{(\ell)}(\boldsymbol{x}) = \boldsymbol{W}^{(\ell)} \boldsymbol{h}^{(\ell-1)}(\boldsymbol{x}) + \boldsymbol{b}^{(\ell)},$$

with $\boldsymbol{W}^{(\ell)} \in \mathbb{R}^{K_\ell \times K_{\ell-1}}$; $\boldsymbol{b}^{(\ell)} \in \mathbb{R}^{K_\ell}$



Multiple ($L \geq 1$) hidden layers

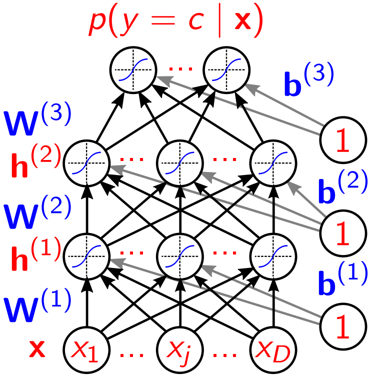
- **Hidden layer pre-activation** (define $\boldsymbol{h}^{(0)} = \boldsymbol{x}$ for convenience):

$$\boldsymbol{z}^{(\ell)}(\boldsymbol{x}) = \boldsymbol{W}^{(\ell)}\boldsymbol{h}^{(\ell-1)}(\boldsymbol{x}) + \boldsymbol{b}^{(\ell)},$$

with $\boldsymbol{W}^{(\ell)} \in \mathbb{R}^{K_\ell \times K_{\ell-1}}$; $\boldsymbol{b}^{(\ell)} \in \mathbb{R}^{K_\ell}$

- **Hidden layer activation:**

$$\boldsymbol{h}^{(\ell)}(\boldsymbol{x}) = \boldsymbol{g}\big(\boldsymbol{z}^{(\ell)}(\boldsymbol{x})\big)$$



Multiple ($L \geq 1$) hidden layers

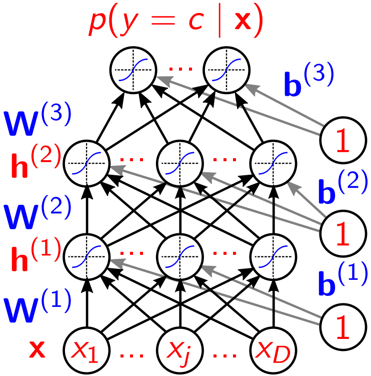
- **Hidden layer pre-activation** (define $\boldsymbol{h}^{(0)} = \boldsymbol{x}$ for convenience):

$$\boldsymbol{z}^{(\ell)}(\boldsymbol{x}) = \boldsymbol{W}^{(\ell)}\boldsymbol{h}^{(\ell-1)}(\boldsymbol{x}) + \boldsymbol{b}^{(\ell)},$$

with $\boldsymbol{W}^{(\ell)} \in \mathbb{R}^{K_\ell \times K_{\ell-1}}$; $\boldsymbol{b}^{(\ell)} \in \mathbb{R}^{K_\ell}$

- **Hidden layer activation:**

$$\boldsymbol{h}^{(\ell)}(\boldsymbol{x}) = \boldsymbol{g}\big(\boldsymbol{z}^{(\ell)}(\boldsymbol{x})\big)$$



- **Output layer activation:**

$$\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{o}(\boldsymbol{z}^{(L+1)}(\boldsymbol{x})) = \boldsymbol{o}(\boldsymbol{W}^{(L+1)}\boldsymbol{h}^{(L)}(\boldsymbol{x}) + \boldsymbol{b}^{(L+1)}).$$

Universal approximation theorem

Theorem

An NN with one hidden layer and a linear output can approximate arbitrarily well any continuous function, given enough hidden units.

Universal approximation theorem

Theorem

An NN with one hidden layer and a linear output can approximate arbitrarily well any continuous function, given enough hidden units.

- First proved for the sigmoid case by Cybenko (1989);

Universal approximation theorem

Theorem

An NN with one hidden layer and a linear output can approximate arbitrarily well any continuous function, given enough hidden units.

- First proved for the sigmoid case by Cybenko (1989);
- Extended to $\tanh$ and many other activation functions by Hornik, Stinchcombe, and White (1989);

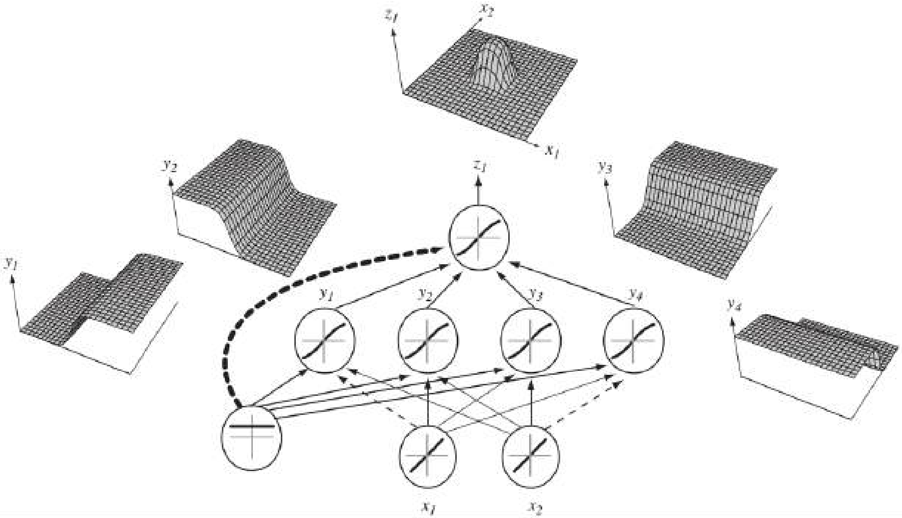
Universal approximation theorem

Theorem

An NN with one hidden layer and a linear output can approximate arbitrarily well any continuous function, given enough hidden units.

- First proved for the sigmoid case by Cybenko (1989);
- Extended to $\tanh$ and many other activation functions by Hornik, Stinchcombe, and White (1989);
- Caveat: may need exponentially many hidden units.

Universal approximation: illustration



Deeper networks

- Deeper networks (more layers) can provide more compact approximations

Theorem

The number of linear regions carved out by a deep neural network with D inputs, depth L, and K hidden units per layer with ReLU activations is

$$O\left(\binom{K}{D}^{D(L-1)} K^D\right)$$

Deeper networks

- Deeper networks (more layers) can provide more compact approximations

Theorem

The number of linear regions carved out by a deep neural network with D inputs, depth L, and K hidden units per layer with ReLU activations is

$$O\left(\binom{K}{D}^{D(L-1)} K^D\right)$$

- For fixed K, deeper networks are exponentially more expressive.

Deeper networks

- Deeper networks (more layers) can provide more compact approximations

Theorem

The number of linear regions carved out by a deep neural network with D inputs, depth L, and K hidden units per layer with ReLU activations is

$$O\left(\binom{K}{D}^{D(L-1)} K^D \right)$$

- For fixed K, deeper networks are exponentially more expressive.

- Proved by Montufar, Pascanu, Cho, and Bengio (2014).

Outline

1. Brief History of Deep Learning (Before LLMs)

2. From models of neurons to artificial neural networks

3. **Deep Learning via Empirical Risk Minimization**
   - Gradient Descent and Stochastic Gradient Descent
   - Gradient Backpropagation and Autodiff
   - Better optimization: momentum, AdaGrad, RMSProp, Adam

4. Convolutional Neural Networks

Empirical risk minimization

- Training/learning: choose parameters $\boldsymbol{\theta} := \{(\boldsymbol{W}^{(\ell)}, \boldsymbol{b}^{(\ell)})\}_{\ell=1}^{L+1}$ by minimizing the empirical risk, maybe plus a regularizer:

$$\mathcal{L}(\boldsymbol{\theta}) = \frac{1}{n} \sum_{i=1}^{n} L(\boldsymbol{f}(\boldsymbol{x}_i; \boldsymbol{\theta}), y_i) + \lambda \Omega(\boldsymbol{\theta})$$

Empirical risk minimization

- Training/learning: choose parameters $\boldsymbol{\theta} := \{(\boldsymbol{W}^{(\ell)}, \boldsymbol{b}^{(\ell)})\}_{\ell=1}^{L+1}$ by minimizing the empirical risk, maybe plus a regularizer:

$$\mathcal{L}(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n} L(\boldsymbol{f}(\boldsymbol{x}_i;\boldsymbol{\theta}), y_i) + \lambda\Omega(\boldsymbol{\theta})$$

✓ $\{(\boldsymbol{x}_i, y_i),\ i = 1, ..., n\}$ is a training set

Empirical risk minimization

- Training/learning: choose parameters $\boldsymbol{\theta} := \{(\boldsymbol{W}^{(\ell)}, \boldsymbol{b}^{(\ell)})\}_{\ell=1}^{L+1}$ by minimizing the empirical risk, maybe plus a regularizer:

$$\mathcal{L}(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n} L(\boldsymbol{f}(\boldsymbol{x}_i;\boldsymbol{\theta}), y_i) + \lambda\Omega(\boldsymbol{\theta})$$

  ✓ $\{(\boldsymbol{x}_i, y_i),\ i = 1, ..., n\}$ is a training set

  ✓ $L(\boldsymbol{f}(\boldsymbol{x}_i;\boldsymbol{\theta}), y_i)$ is a loss function

Empirical risk minimization

- Training/learning: choose parameters $\boldsymbol{\theta} := \{(\boldsymbol{W}^{(\ell)}, \boldsymbol{b}^{(\ell)})\}_{\ell=1}^{L+1}$ by minimizing the empirical risk, maybe plus a regularizer:

$$\mathcal{L}(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n} L(\boldsymbol{f}(\boldsymbol{x}_i;\boldsymbol{\theta}), y_i) + \lambda\Omega(\boldsymbol{\theta})$$

  - ✓ $\{(\boldsymbol{x}_i, y_i),\ i = 1, ..., n\}$ is a training set
  - ✓ $L(\boldsymbol{f}(\boldsymbol{x}_i;\boldsymbol{\theta}), y_i)$ is a loss function
  - ✓ $\Omega(\boldsymbol{\theta})$ is a regularizer

Empirical risk minimization

- Training/learning: choose parameters $\boldsymbol{\theta} := \{(\boldsymbol{W}^{(\ell)}, \boldsymbol{b}^{(\ell)})\}_{\ell=1}^{L+1}$ by minimizing the empirical risk, maybe plus a regularizer:

$$\mathcal{L}(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n} L(\boldsymbol{f}(\boldsymbol{x}_i;\boldsymbol{\theta}), y_i) + \lambda\Omega(\boldsymbol{\theta})$$

  - ✓ $\{(\boldsymbol{x}_i, y_i),\ i = 1, ..., n\}$ is a training set
  - ✓ $L(\boldsymbol{f}(\boldsymbol{x}_i;\boldsymbol{\theta}), y_i)$ is a loss function
  - ✓ $\Omega(\boldsymbol{\theta})$ is a regularizer
  - ✓ λ is the regularization constant (hyperparameter to be tuned)

Outline

Gradient Descent

- Gradient descent algorithm:
  - ✓ Start at some initial point $\boldsymbol{\theta}_0 \in \mathbb{R}^d$
  - ✓ For $t = 1, 2, \ldots,$
    - ▷ choose step-size α_t,
    - ▷ take a step of size α_t in the direction of the negative gradient:

$$\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} - \alpha_t \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}_{t-1})$$

Gradient Descent

- Gradient descent algorithm:
  - ✓ Start at some initial point $\boldsymbol{\theta}_0 \in \mathbb{R}^d$
  - ✓ For $t = 1, 2, ...,$
    - ▷ choose step-size α_t,
    - ▷ take a step of size α_t in the direction of the negative gradient:

$$\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} - \alpha_t \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}_{t-1})$$

Gradient Descent

- Gradient descent algorithm:
  - ✓ Start at some initial point $\boldsymbol{\theta}_0 \in \mathbb{R}^d$
  - ✓ For $t = 1, 2, ...,$
    - ▷ choose step-size α_t,
    - ▷ take a step of size α_t in the direction of the negative gradient:

$$\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} - \alpha_t \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}_{t-1})$$

- Several (many) ways to choose α_t;

Gradient Descent

- Gradient descent algorithm:
  - ✓ Start at some initial point $\boldsymbol{\theta}_0 \in \mathbb{R}^d$
  - ✓ For $t = 1, 2, ...,$
    - ▷ choose step-size α_t,
    - ▷ take a step of size α_t in the direction of the negative gradient:

$$\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} - \alpha_t \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}_{t-1})$$

- Several (many) ways to choose α_t;
- Some stopping criterion is used; *e.g.*, $\|\nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}_t)\| \leq \delta$.

Gradient descent

- The empirical risk minimization (ERM) objective function:

$$\begin{aligned}\mathcal{L}(\boldsymbol{\theta}) &= \lambda\Omega(\boldsymbol{\theta}) + \frac{1}{n}\sum_{i=1}^{n} L(\boldsymbol{f}(\boldsymbol{x}_i;\boldsymbol{\theta}), y_i)\\ &= \frac{1}{n}\sum_{i=1}^{n} \underbrace{\lambda\Omega(\boldsymbol{\theta}) + L(\boldsymbol{f}(\boldsymbol{x}_i;\boldsymbol{\theta}), y_i)}_{\mathcal{L}_i(\boldsymbol{\theta})}\end{aligned}$$

Gradient descent

- The empirical risk minimization (ERM) objective function:

$$\begin{aligned}\mathcal{L}(\boldsymbol{\theta}) &= \lambda\Omega(\boldsymbol{\theta}) + \frac{1}{n}\sum_{i=1}^{n} L(\boldsymbol{f}(\boldsymbol{x}_i;\boldsymbol{\theta}), y_i)\\ &= \frac{1}{n}\sum_{i=1}^{n} \underbrace{\lambda\Omega(\boldsymbol{\theta}) + L(\boldsymbol{f}(\boldsymbol{x}_i;\boldsymbol{\theta}), y_i)}_{\mathcal{L}_i(\boldsymbol{\theta})} \;=\; \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}_i(\boldsymbol{\theta})\end{aligned}$$

Gradient descent

- The empirical risk minimization (ERM) objective function:

$$\begin{aligned}\mathcal{L}(\boldsymbol{\theta}) &= \lambda\Omega(\boldsymbol{\theta}) + \frac{1}{n}\sum_{i=1}^{n} L(\boldsymbol{f}(\boldsymbol{x}_i;\boldsymbol{\theta}), y_i)\\ &= \frac{1}{n}\sum_{i=1}^{n} \underbrace{\lambda\Omega(\boldsymbol{\theta}) + L(\boldsymbol{f}(\boldsymbol{x}_i;\boldsymbol{\theta}), y_i)}_{\mathcal{L}_i(\boldsymbol{\theta})} = \frac{1}{n}\sum_{i=1}^{n}\mathcal{L}_i(\boldsymbol{\theta})\end{aligned}$$

- The gradient:

$$\nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta}) := \frac{1}{n}\sum_{i=1}^{n}\nabla_{\boldsymbol{\theta}}\mathcal{L}_i(\boldsymbol{\theta})$$

Gradient descent

- The empirical risk minimization (ERM) objective function:

$$\begin{aligned}\mathcal{L}(\boldsymbol{\theta}) &= \lambda\Omega(\boldsymbol{\theta}) + \frac{1}{n}\sum_{i=1}^{n} L(\boldsymbol{f}(\boldsymbol{x}_i;\boldsymbol{\theta}), y_i) \\ &= \frac{1}{n}\sum_{i=1}^{n} \underbrace{\lambda\Omega(\boldsymbol{\theta}) + L(\boldsymbol{f}(\boldsymbol{x}_i;\boldsymbol{\theta}), y_i)}_{\mathcal{L}_i(\boldsymbol{\theta})} \;=\; \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}_i(\boldsymbol{\theta})\end{aligned}$$

- The gradient:

$$\nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta}) \;:=\; \frac{1}{n}\sum_{i=1}^{n}\nabla_{\boldsymbol{\theta}}\mathcal{L}_i(\boldsymbol{\theta})$$

- Requires a full pass over the data to update the weights: too slow!

Stochastic gradient descent (SGD)

- Sample one gradient $\nabla_{\boldsymbol{\theta}} \mathcal{L}_i(\boldsymbol{\theta})$ uniformly at random: $j \in \{1, ..., n\}$

Stochastic gradient descent (SGD)

- Sample one gradient $\nabla_{\boldsymbol{\theta}} \mathcal{L}_i(\boldsymbol{\theta})$ uniformly at random: $j \in \{1, ..., n\}$

- This an unbiased estimate of the gradient,

$$\mathbb{E}_j[\nabla_{\boldsymbol{\theta}} \mathcal{L}_j(\boldsymbol{\theta})] = \frac{1}{n} \sum_{i=1}^{n} \nabla_{\boldsymbol{\theta}} \mathcal{L}_i(\boldsymbol{\theta}) = \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta})$$

but may be a noisy (high variance) one.

Stochastic gradient descent (SGD)

- Sample one gradient $\nabla_{\boldsymbol{\theta}} \mathcal{L}_i(\boldsymbol{\theta})$ uniformly at random: $j \in \{1, ..., n\}$

- This an unbiased estimate of the gradient,

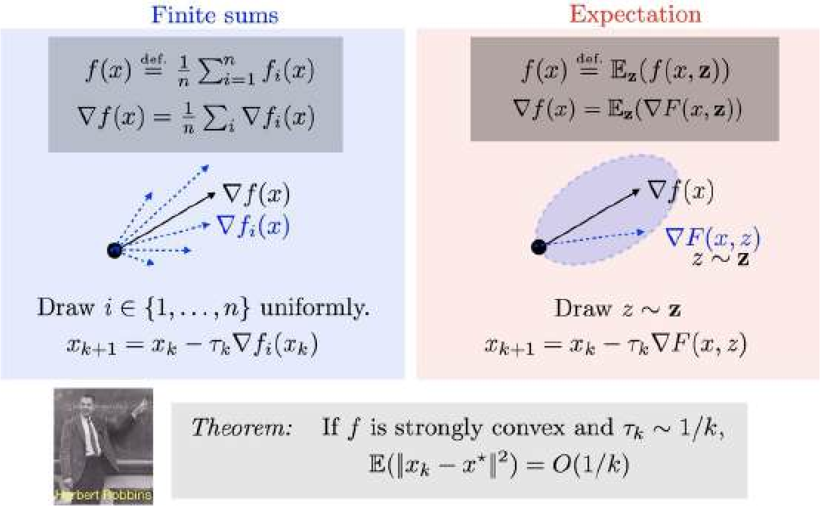
$$\mathbb{E}_j[\nabla_{\boldsymbol{\theta}} \mathcal{L}_j(\boldsymbol{\theta})] = \frac{1}{n} \sum_{i=1}^{n} \nabla_{\boldsymbol{\theta}} \mathcal{L}_i(\boldsymbol{\theta}) = \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta})$$

but may be a noisy (high variance) one.

- Stochastic gradient "descent" (SGD):
  - ✓ Start at some initial point $\boldsymbol{\theta}_0 \in \mathbb{R}^d$
  - ✓ For $t = 1, 2, ...,$
    - ▷ sample $i \in \{1, ..., n\}$ at random and choose step-size α_t,
    - ▷ take a step of size α_t in the direction of the negative gradient:

$$\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} - \alpha_t \nabla_{\boldsymbol{\theta}} L(f(\boldsymbol{x}_i; \boldsymbol{\theta}_{t-1}), y_i)$$

Visual summary



(Picture by Gabriel Peyré)

SGD with mini-batches

- Instead of a single sample, use a mini-batch $\{j_1, \ldots, j_B\}$ $(B \ll n)$

SGD with mini-batches

- Instead of a single sample, use a mini-batch $\{j_1, \ldots, j_B\}$ $(B \ll n)$
- Mini-batch SGD (SGD):
  - ✓ Start at some initial point $\boldsymbol{\theta}_0 \in \mathbb{R}^d$
  - ✓ For $t = 1, 2, ...,$
    - ▷ sample $\{j_1, ...j_B\} \subset \{1, ..., n\}$; choose step-size α_t,
    - ▷ take a step of size α_t in the direction of the negative gradient:

$$\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} - \alpha_t \frac{1}{B} \sum_{i=1}^{B} \nabla_{\boldsymbol{\theta}} L(f(\boldsymbol{x}_{j_i}; \boldsymbol{\theta}_{t-1}), y_{j_i})$$

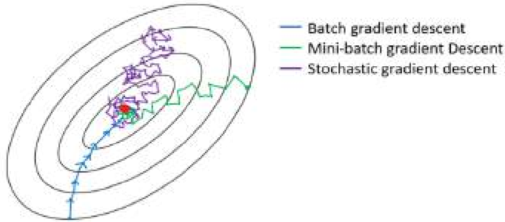
SGD with mini-batches

- Instead of a single sample, use a mini-batch $\{j_1, \ldots, j_B\}$ $(B \ll n)$

- Mini-batch SGD (SGD):
  - ✓ Start at some initial point $\boldsymbol{\theta}_0 \in \mathbb{R}^d$
  - ✓ For $t = 1, 2, ...,$
    - ▷ sample $\{j_1, ...j_B\} \subset \{1, ..., n\}$; choose step-size α_t,
    - ▷ take a step of size α_t in the direction of the negative gradient:

$$\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} - \alpha_t \frac{1}{B} \sum_{i=1}^{B} \nabla_{\boldsymbol{\theta}} L(f(\boldsymbol{x}_{j_i}; \boldsymbol{\theta}_{t-1}), y_{j_i})$$

- Less noisy, still unbiased gradient estimate.



The key Ingredients of SGD

- The loss function $L(f(\boldsymbol{x}_i;\boldsymbol{\theta}), y_i)$;
- A procedure for computing its gradient $\nabla_{\boldsymbol{\theta}} L(f(\boldsymbol{x}_i;\boldsymbol{\theta}), y_i)$;
- The regularizer $\Omega(\boldsymbol{\theta})$;
- ... its gradients, $\nabla_{\boldsymbol{\theta}} \Omega(\boldsymbol{\theta})$

The key Ingredients of SGD

- The loss function $L(f(\boldsymbol{x}_i;\boldsymbol{\theta}), y_i)$;
- A procedure for computing its gradient $\nabla_{\boldsymbol{\theta}} L(f(\boldsymbol{x}_i;\boldsymbol{\theta}), y_i)$;
- The regularizer $\Omega(\boldsymbol{\theta})$;
- ... its gradients, $\nabla_{\boldsymbol{\theta}}\Omega(\boldsymbol{\theta})$

Let's see them one at the time...

Squared error loss

- The common choice for regression/reconstruction problems

Squared error loss

- The common choice for regression/reconstruction problems

- The goal is to have $\widehat{\boldsymbol{y}} = \boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}) \approx \boldsymbol{y}$

Squared error loss

- The common choice for regression/reconstruction problems
- The goal is to have $\widehat{\boldsymbol{y}} = \boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}) \approx \boldsymbol{y}$
- Squared error loss:

$$L(\widehat{\boldsymbol{y}}, \boldsymbol{y}) = \frac{1}{2}\|\widehat{\boldsymbol{y}} - \boldsymbol{y}\|^2$$

Squared error loss

- The common choice for regression/reconstruction problems

- The goal is to have $\widehat{\boldsymbol{y}} = \boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}) \approx \boldsymbol{y}$

- Squared error loss:

$$L(\widehat{\boldsymbol{y}}, \boldsymbol{y}) \;=\; \frac{1}{2}\|\widehat{\boldsymbol{y}} - \boldsymbol{y}\|^2$$

- Loss gradient:

$$\frac{\partial L(\widehat{\boldsymbol{y}}, \boldsymbol{y})}{\partial \widehat{y}_j} = \widehat{y}_j - y_j \quad \Rightarrow \quad \nabla_{\widehat{\boldsymbol{y}}} L(\widehat{\boldsymbol{y}}, \boldsymbol{y}) = \widehat{\boldsymbol{y}} - \boldsymbol{y}$$

Squared error loss

- The common choice for regression/reconstruction problems

- The goal is to have $\widehat{\boldsymbol{y}} = \boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}) \approx \boldsymbol{y}$

- Squared error loss:

$$L(\widehat{\boldsymbol{y}}, \boldsymbol{y}) \;\; = \;\; \frac{1}{2}\|\widehat{\boldsymbol{y}} - \boldsymbol{y}\|^2$$

- Loss gradient:

$$\frac{\partial L(\widehat{\boldsymbol{y}}, \boldsymbol{y})}{\partial \widehat{y}_j} = \widehat{y}_j - y_j \quad \Rightarrow \quad \nabla_{\widehat{\boldsymbol{y}}} L(\widehat{\boldsymbol{y}}, \boldsymbol{y}) = \widehat{\boldsymbol{y}} - \boldsymbol{y}$$

- Notice: this is **not** (yet) $\nabla_{\boldsymbol{\theta}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), \boldsymbol{y})$

Cross-entropy loss (negative log-likelihood)

- The common choice for classification with a softmax output layer

Cross-entropy loss (negative log-likelihood)

- The common choice for classification with a softmax output layer
- NN output: $\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}) = \mathbf{softmax}\big(\boldsymbol{z}(\boldsymbol{x};\boldsymbol{\theta})\big)$ (where $\boldsymbol{z} = \boldsymbol{z}^{(L+1)}$)

Cross-entropy loss (negative log-likelihood)

- The common choice for classification with a softmax output layer
- NN output: $\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}) = \mathbf{softmax}\big(\boldsymbol{z}(\boldsymbol{x};\boldsymbol{\theta})\big)$ (where $\boldsymbol{z} = \boldsymbol{z}^{(L+1)}$)
- Negative log-likelihood, i.e., cross-entropy loss:

$$L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y) = -\sum_{c} 1_{(c=y)} \log f_c(\boldsymbol{x};\boldsymbol{\theta})$$

Cross-entropy loss (negative log-likelihood)

- The common choice for classification with a softmax output layer
- NN output: $\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}) = \mathbf{softmax}\big(\boldsymbol{z}(\boldsymbol{x};\boldsymbol{\theta})\big)$ (where $\boldsymbol{z} = \boldsymbol{z}^{(L+1)}$)
- Negative log-likelihood, i.e., cross-entropy loss:

$$L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y) = -\sum_{c} 1_{(c=y)} \log f_c(\boldsymbol{x};\boldsymbol{\theta}) = -\log\big[\mathbf{softmax}(\boldsymbol{z}(\boldsymbol{x};\boldsymbol{\theta}))\big]_y$$

Cross-entropy loss (negative log-likelihood)

- The common choice for classification with a softmax output layer
- NN output: $\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}) = \mathbf{softmax}\big(\boldsymbol{z}(\boldsymbol{x};\boldsymbol{\theta})\big)$ (where $\boldsymbol{z} = \boldsymbol{z}^{(L+1)}$)
- Negative log-likelihood, i.e., cross-entropy loss:

$$L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y) = -\sum_c 1_{(c=y)} \log f_c(\boldsymbol{x};\boldsymbol{\theta}) = -\log\big[\mathbf{softmax}(\boldsymbol{z}(\boldsymbol{x};\boldsymbol{\theta}))\big]_y$$

- Intuition: reduce loss $\Rightarrow$ increase $\big[\mathbf{softmax}(\boldsymbol{z}(\boldsymbol{x}_i;\boldsymbol{\theta}))\big]_{y_i}$

Cross-entropy loss (negative log-likelihood)

- The common choice for classification with a softmax output layer

- NN output: $\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}) = \mathbf{softmax}\big(\boldsymbol{z}(\boldsymbol{x};\boldsymbol{\theta})\big)$ (where $\boldsymbol{z} = \boldsymbol{z}^{(L+1)}$)

- Negative log-likelihood, i.e., cross-entropy loss:

$$L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y) = -\sum_c 1_{(c=y)} \log f_c(\boldsymbol{x};\boldsymbol{\theta}) = -\log\big[\mathbf{softmax}(\boldsymbol{z}(\boldsymbol{x};\boldsymbol{\theta}))\big]_y$$

- Intuition: reduce loss $\Rightarrow$ increase $\big[\mathbf{softmax}(\boldsymbol{z}(\boldsymbol{x}_i;\boldsymbol{\theta}))\big]_{y_i}$

- Loss gradient with respect to output pre-activation $z_c \equiv \big[\boldsymbol{z}(\boldsymbol{x};\boldsymbol{\theta})\big]_c$

$$\frac{\partial L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}, y))}{\partial z_c} = \big[\mathbf{softmax}(\boldsymbol{z}(\boldsymbol{x}))\big]_c - 1_{(c=y)},$$

Cross-entropy loss (negative log-likelihood)

- The common choice for classification with a softmax output layer
- NN output: $\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}) = \mathbf{softmax}\big(\boldsymbol{z}(\boldsymbol{x};\boldsymbol{\theta})\big)$ (where $\boldsymbol{z} = \boldsymbol{z}^{(L+1)}$)
- Negative log-likelihood, i.e., cross-entropy loss:

$$L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y) = -\sum_c 1_{(c=y)} \log f_c(\boldsymbol{x};\boldsymbol{\theta}) = -\log\big[\mathbf{softmax}(\boldsymbol{z}(\boldsymbol{x};\boldsymbol{\theta}))\big]_y$$

- Intuition: reduce loss $\Rightarrow$ increase $\big[\mathbf{softmax}(\boldsymbol{z}(\boldsymbol{x}_i;\boldsymbol{\theta}))\big]_{y_i}$
- Loss gradient with respect to output pre-activation $z_c \equiv \big[\boldsymbol{z}(\boldsymbol{x};\boldsymbol{\theta})\big]_c$

$$\frac{\partial L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}, y))}{\partial z_c} = \big[\mathbf{softmax}(\boldsymbol{z}(\boldsymbol{x}))\big]_c - 1_{(c=y)},$$

- Intuition: $\partial L / \partial z_c \geq 0$, for $c \neq y$;

Cross-entropy loss (negative log-likelihood)

- The common choice for classification with a softmax output layer
- NN output: $\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}) = \mathbf{softmax}\big(\boldsymbol{z}(\boldsymbol{x};\boldsymbol{\theta})\big)$ (where $\boldsymbol{z} = \boldsymbol{z}^{(L+1)}$)
- Negative log-likelihood, i.e., cross-entropy loss:

$$L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y) = -\sum_c 1_{(c=y)} \log f_c(\boldsymbol{x};\boldsymbol{\theta}) = -\log\big[\mathbf{softmax}(\boldsymbol{z}(\boldsymbol{x};\boldsymbol{\theta}))\big]_y$$

- Intuition: reduce loss $\Rightarrow$ increase $\big[\mathbf{softmax}(\boldsymbol{z}(\boldsymbol{x}_i;\boldsymbol{\theta}))\big]_{y_i}$
- Loss gradient with respect to output pre-activation $z_c \equiv \big[\boldsymbol{z}(\boldsymbol{x};\boldsymbol{\theta})\big]_c$

$$\frac{\partial L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}, y))}{\partial z_c} = \big[\mathbf{softmax}(\boldsymbol{z}(\boldsymbol{x}))\big]_c - 1_{(c=y)},$$

- Intuition: $\partial L/\partial z_c \geq 0$, for $c \neq y$; $\partial L/\partial z_c \leq 0$, for $c = y$ (true class).

Cross-entropy loss (negative log-likelihood)

- The common choice for classification with a softmax output layer
- NN output: $\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}) = \textbf{softmax}\big(\boldsymbol{z}(\boldsymbol{x};\boldsymbol{\theta})\big)$ (where $\boldsymbol{z} = \boldsymbol{z}^{(L+1)}$)
- Negative log-likelihood, i.e., cross-entropy loss:

$$L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y) = -\sum_c 1_{(c=y)} \log f_c(\boldsymbol{x};\boldsymbol{\theta}) = -\log\big[\textbf{softmax}(\boldsymbol{z}(\boldsymbol{x};\boldsymbol{\theta}))\big]_y$$

- Intuition: reduce loss $\Rightarrow$ increase $\big[\textbf{softmax}(\boldsymbol{z}(\boldsymbol{x}_i;\boldsymbol{\theta}))\big]_{y_i}$
- Loss gradient with respect to output pre-activation $z_c \equiv \big[\boldsymbol{z}(\boldsymbol{x};\boldsymbol{\theta})\big]_c$

$$\frac{\partial L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}, y))}{\partial z_c} = \big[\textbf{softmax}(\boldsymbol{z}(\boldsymbol{x}))\big]_c - 1_{(c=y)},$$

- Intuition: $\partial L/\partial z_c \geq 0$, for $c \neq y$; $\partial L/\partial z_c \leq 0$, for $c = y$ (true class).
- Again, this is **not** (yet) $\nabla_{\boldsymbol{\theta}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), \boldsymbol{y})$

The Key Ingredients of SGD

- The loss function $L(f(\boldsymbol{x}_i;\boldsymbol{\theta}), y_i)$; ✓
- A procedure for computing its gradient $\nabla_{\boldsymbol{\theta}} L(f(\boldsymbol{x}_i;\boldsymbol{\theta}), y_i)$; next
- The regularizer $\Omega(\boldsymbol{\theta})$;
- ... its gradients, $\nabla_{\boldsymbol{\theta}}\Omega(\boldsymbol{\theta})$

Outline

Gradient computation

- Recall the goal: compute $\nabla_{\boldsymbol{\theta}} L(f(\boldsymbol{x}_i;\boldsymbol{\theta}), y_i)$,

Gradient computation

- Recall the goal: compute $\nabla_{\boldsymbol{\theta}} L(f(\boldsymbol{x}_i;\boldsymbol{\theta}), y_i)$,

- This will be done with the gradient backpropagation algorithm

Gradient computation

- Recall the goal: compute $\nabla_{\boldsymbol{\theta}} L(f(\boldsymbol{x}_i;\boldsymbol{\theta}), y_i)$,
- This will be done with the gradient backpropagation algorithm
- Key idea: use the chain rule for derivatives!

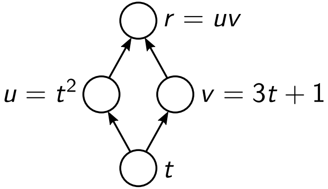
$$h(x) = f\big(g(x)\big) \quad \Rightarrow \quad \frac{d\,h(x)}{d\,x} = \frac{d\,f(u)}{d\,u}\bigg|_{u=g(x)} \frac{d\,g(x)}{d\,x}.$$

Gradient computation

- Recall the goal: compute $\nabla_{\boldsymbol{\theta}} L(f(\boldsymbol{x}_i;\boldsymbol{\theta}), y_i)$,
- This will be done with the gradient backpropagation algorithm
- Key idea: use the chain rule for derivatives!

$$h(x) = f\big(g(x)\big) \quad \Rightarrow \quad \frac{d\,h(x)}{d\,x} = \left.\frac{d\,f(u)}{d\,u}\right|_{u=g(x)} \frac{d\,g(x)}{d\,x}.$$

- Example:



$$\frac{\partial r(t)}{\partial t} \quad =$$

Gradient computation

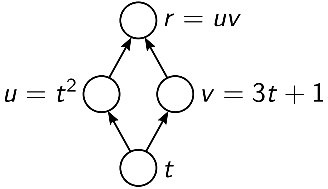
- Recall the goal: compute $\nabla_{\boldsymbol{\theta}} L(f(\boldsymbol{x}_i;\boldsymbol{\theta}), y_i)$,

- This will be done with the gradient backpropagation algorithm

- Key idea: use the chain rule for derivatives!

$$h(x) = f\big(g(x)\big) \quad \Rightarrow \quad \frac{d\,h(x)}{d\,x} = \frac{d\,f(u)}{d\,u}\bigg|_{u=g(x)} \frac{d\,g(x)}{d\,x}.$$

- Example:



$$\frac{\partial r(t)}{\partial t} \quad = \quad \frac{\partial r(u)}{\partial u}\frac{\partial u(t)}{\partial t} + \frac{\partial r(v)}{\partial v}\frac{\partial v(t)}{\partial t}$$

Gradient computation

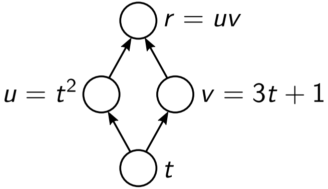
- Recall the goal: compute $\nabla_{\boldsymbol{\theta}} L(f(\boldsymbol{x}_i;\boldsymbol{\theta}), y_i)$,

- This will be done with the gradient backpropagation algorithm

- Key idea: use the chain rule for derivatives!

$$h(x) = f\big(g(x)\big) \quad \Rightarrow \quad \frac{d\,h(x)}{d\,x} = \frac{d\,f(u)}{d\,u}\bigg|_{u=g(x)} \frac{d\,g(x)}{d\,x}.$$

- Example:



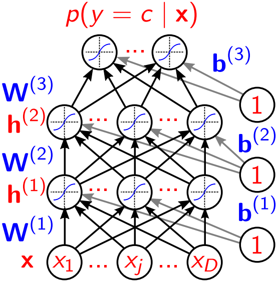
$$\begin{aligned} \frac{\partial r(t)}{\partial t} &= \frac{\partial r(u)}{\partial u}\frac{\partial u(t)}{\partial t} + \frac{\partial r(v)}{\partial v}\frac{\partial v(t)}{\partial t} \\ &= 2tv + 3u \\ &= 2t(3t+1) + 3t^2 = 9t^2 + 2t. \end{aligned}$$

Hidden layer gradient

- Recap: $\boldsymbol{z}^{(\ell+1)} = \boldsymbol{W}^{(\ell+1)}\boldsymbol{h}^{(\ell)} + \boldsymbol{b}^{(\ell+1)}$

$$\frac{\partial L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}),y)}{\partial h_j^{(\ell)}} = \sum_i \frac{\partial L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}),y)}{\partial z_i^{(\ell+1)}} \frac{\partial z_i^{(\ell+1)}}{\partial h_j^{(\ell)}}$$

$$= \sum_i \frac{\partial L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}),y)}{\partial z_i^{(\ell+1)}} \boldsymbol{W}_{i,j}^{(\ell+1)}$$

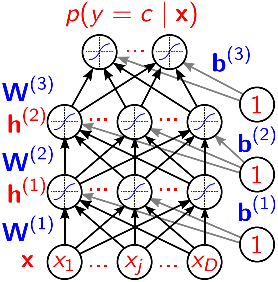


Hidden layer gradient

- Recap: $\boldsymbol{z}^{(\ell+1)} = \boldsymbol{W}^{(\ell+1)}\boldsymbol{h}^{(\ell)} + \boldsymbol{b}^{(\ell+1)}$

$$\frac{\partial L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}),y)}{\partial h_j^{(\ell)}} = \sum_i \frac{\partial L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}),y)}{\partial z_i^{(\ell+1)}} \frac{\partial z_i^{(\ell+1)}}{\partial h_j^{(\ell)}}$$

$$= \sum_i \frac{\partial L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}),y)}{\partial z_i^{(\ell+1)}} \boldsymbol{W}_{i,j}^{(\ell+1)}$$



- Hence

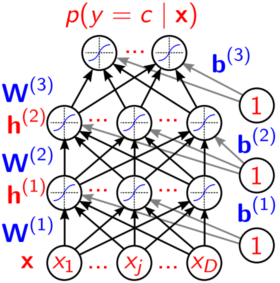
$$\nabla_{\boldsymbol{h}^{(\ell)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}),y) = {\boldsymbol{W}^{(\ell+1)}}^\top \nabla_{\boldsymbol{z}^{(\ell+1)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}),y).$$

Hidden layer gradient (before activation)

- Recap: $h_j^{(\ell)} = g(z_j^{(\ell)})$, where $g : \mathbb{R} \to \mathbb{R}$ is the activation function.

$$\frac{\partial L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y)}{\partial z_j^{(\ell)}} = \frac{\partial L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y)}{\partial h_j^{(\ell)}} \frac{\partial h_j^{(\ell)}}{\partial z_j^{(\ell)}}$$

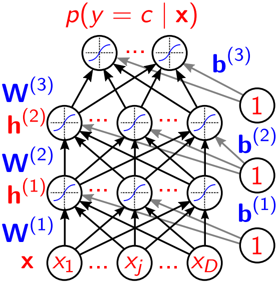
$$= \frac{\partial L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y)}{\partial h_j^{(\ell)}} g'(z_j^{(\ell)})$$



Hidden layer gradient (before activation)

- Recap: $h_j^{(\ell)} = g(z_j^{(\ell)})$, where $g : \mathbb{R} \to \mathbb{R}$ is the activation function.

$$\frac{\partial L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y)}{\partial z_j^{(\ell)}} = \frac{\partial L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y)}{\partial h_j^{(\ell)}} \frac{\partial h_j^{(\ell)}}{\partial z_j^{(\ell)}} = \frac{\partial L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y)}{\partial h_j^{(\ell)}} g'(z_j^{(\ell)})$$

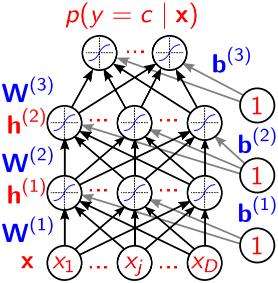


- Hence $\nabla_{\boldsymbol{z}^{(\ell)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y) = \nabla_{\boldsymbol{h}^{(\ell)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y) \odot \boldsymbol{g}'(\boldsymbol{z}^{(\ell)})$.

Hidden layer gradient (before activation)

- Recap: $h_j^{(\ell)} = g(z_j^{(\ell)})$, where $g : \mathbb{R} \to \mathbb{R}$ is the activation function.

$$\frac{\partial L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y)}{\partial z_j^{(\ell)}} = \frac{\partial L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y)}{\partial h_j^{(\ell)}} \frac{\partial h_j^{(\ell)}}{\partial z_j^{(\ell)}} = \frac{\partial L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y)}{\partial h_j^{(\ell)}} g'(z_j^{(\ell)})$$



- Hence $\nabla_{\boldsymbol{z}^{(\ell)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y) = \nabla_{\boldsymbol{h}^{(\ell)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y) \odot \boldsymbol{g}'(\boldsymbol{z}^{(\ell)})$.

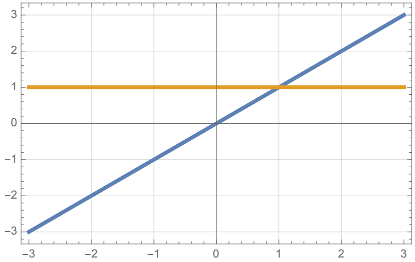
- What are the activation function derivatives $\boldsymbol{g}'(\boldsymbol{z}^{(\ell)})$?

Linear activation

$$\boxed{g(z) = z}$$

Derivative:

$$\boxed{g'(z) = 1}$$

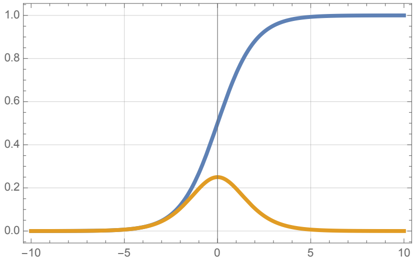


Sigmoid activation

$$g(z) = \sigma(z) = \frac{1}{1 + e^{-z}}$$

Derivative:

$$g'(z) = g(z)(1 - g(z))$$

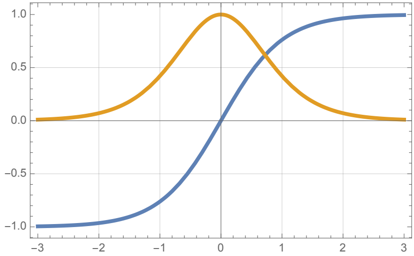


Hyperbolic tangent activation

$$\boxed{g(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}}$$

Derivative:

$$\boxed{g'(z) = 1 - g(z)^2 = \text{sech}^2(x)}$$

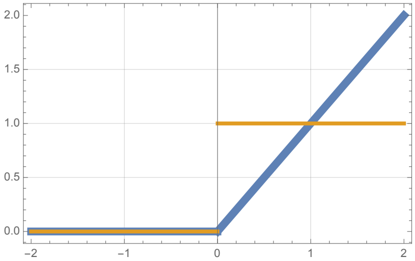


Rectified linear unit activation

$$\boxed{g(z) = \text{relu}(z) = \max\{0, z\}}$$

Derivative (except for $z = 0$):

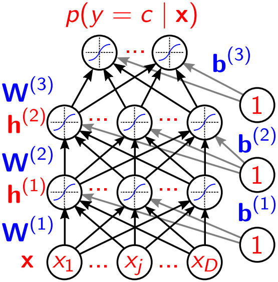
$$\boxed{g'(z) = 1_{z>0}}$$



Parameter gradient

- Recap: $\boldsymbol{z}^{(\ell)} = \boldsymbol{W}^{(\ell)}\boldsymbol{h}^{(\ell-1)} + \boldsymbol{b}^{(\ell)}$.

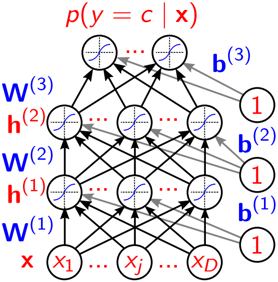
$$\frac{\partial L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y)}{\partial \boldsymbol{W}_{i,j}^{(\ell)}} = \frac{\partial L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y)}{\partial z_i^{(\ell)}} \frac{\partial z_i^{(\ell)}}{\partial \boldsymbol{W}_{i,j}^{(\ell)}} = \frac{\partial L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y)}{\partial z_i^{(\ell)}} h_j^{(\ell-1)}$$



Parameter gradient

- Recap: $\boldsymbol{z}^{(\ell)} = \boldsymbol{W}^{(\ell)}\boldsymbol{h}^{(\ell-1)} + \boldsymbol{b}^{(\ell)}$.

$$\frac{\partial L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}),y)}{\partial \boldsymbol{W}_{i,j}^{(\ell)}} = \frac{\partial L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}),y)}{\partial z_i^{(\ell)}}\frac{\partial z_i^{(\ell)}}{\partial \boldsymbol{W}_{i,j}^{(\ell)}} = \frac{\partial L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}),y)}{\partial z_i^{(\ell)}} h_j^{(\ell-1)}$$

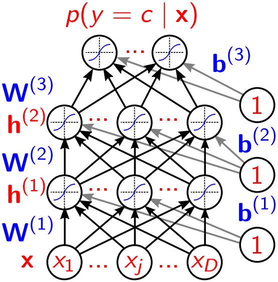


- Hence $\nabla_{\boldsymbol{W}^{(\ell)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}),y) = \nabla_{\boldsymbol{z}^{(\ell)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}),y)\boldsymbol{h}^{(\ell-1)\top}$

Parameter gradient

- Recap: $\boldsymbol{z}^{(\ell)} = \boldsymbol{W}^{(\ell)}\boldsymbol{h}^{(\ell-1)} + \boldsymbol{b}^{(\ell)}$.

$$\frac{\partial L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}),y)}{\partial \boldsymbol{W}_{i,j}^{(\ell)}} = \frac{\partial L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}),y)}{\partial z_i^{(\ell)}}\frac{\partial z_i^{(\ell)}}{\partial \boldsymbol{W}_{i,j}^{(\ell)}} = \frac{\partial L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}),y)}{\partial z_i^{(\ell)}} h_j^{(\ell-1)}$$



- Hence $\nabla_{\boldsymbol{W}^{(\ell)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}),y) = \nabla_{\boldsymbol{z}^{(\ell)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}),y)\boldsymbol{h}^{(\ell-1)^\top}$
- Similarly, $\nabla_{\boldsymbol{b}^{(\ell)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}),y) = \nabla_{\boldsymbol{z}^{(\ell)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}),y)$

Backpropagation

Compute output gradient (before activation):

$$\nabla_{\boldsymbol{z}^{(L+1)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y) = \boldsymbol{f}(\boldsymbol{x}) - \mathbf{1}_y$$

for ℓ from $L+1$ to 1 **do**

Compute gradients of hidden layer parameters:

$$\begin{aligned}\nabla_{\boldsymbol{W}^{(\ell)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y) &= \nabla_{\boldsymbol{z}^{(\ell)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y)\ {\boldsymbol{h}^{(\ell-1)}}^{\top} \\ \nabla_{\boldsymbol{b}^{(\ell)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y) &= \nabla_{\boldsymbol{z}^{(\ell)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y)\end{aligned}$$

Compute gradient of hidden layer below:

$$\nabla_{\boldsymbol{h}^{(\ell-1)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y) = {\boldsymbol{W}^{(\ell)}}^{\top} \nabla_{\boldsymbol{z}^{(\ell)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y)$$

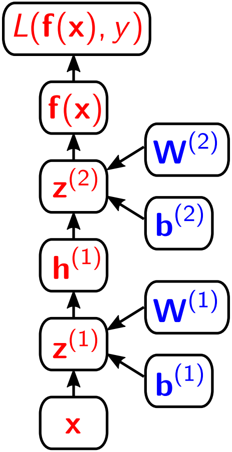
Compute gradient of hidden layer below (before activation):

$$\nabla_{\boldsymbol{z}^{(\ell-1)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y) = \nabla_{\boldsymbol{h}^{(\ell-1)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y) \odot \boldsymbol{g}'(\boldsymbol{z}^{(\ell-1)})$$

end for

The computation graph view

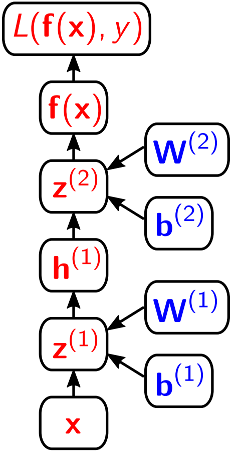
- Forward propagation can be represented as a DAG (directed acyclic graph).



The computation graph view

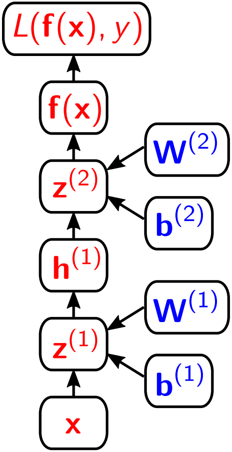
- Forward propagation can be represented as a DAG (directed acyclic graph).

- Allows implementing forward propagation in a modular way.



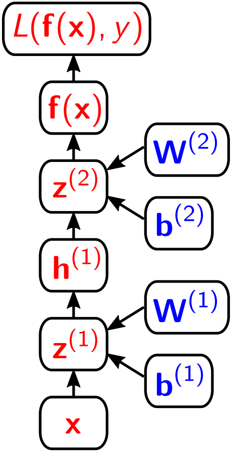
The computation graph view

- Forward propagation can be represented as a DAG (directed acyclic graph).
- Allows implementing forward propagation in a modular way.
- Each box can be an object with a `fprop` method, which computes the output of the box given its inputs.



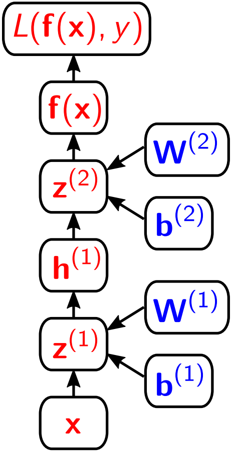
The computation graph view

- Forward propagation can be represented as a DAG (directed acyclic graph).
- Allows implementing forward propagation in a modular way.
- Each box can be an object with a `fprop` method, which computes the output of the box given its inputs.
- Calling the `fprop` method of each box in the right order yields forward propagation.



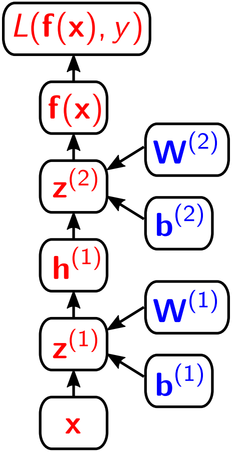
Automatic differentiation (Autodiff)

- Backpropagation is also implementable in a modular way.



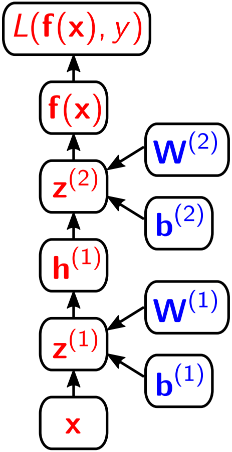
Automatic differentiation (Autodiff)

- Backpropagation is also implementable in a modular way.
- Each box should have a `bprop` method, which computes the loss gradient w.r.t. its parents, given the loss gradient w.r.t. to the output.



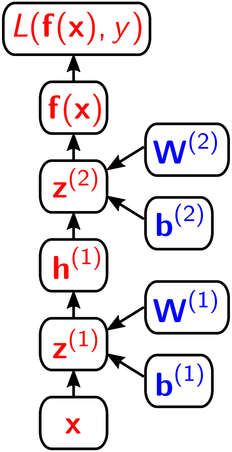
Automatic differentiation (Autodiff)

- Backpropagation is also implementable in a modular way.
- Each box should have a `bprop` method, which computes the loss gradient w.r.t. its parents, given the loss gradient w.r.t. to the output.
- Can make use of cached computation done during the `fprop` method



Automatic differentiation (Autodiff)

- Backpropagation is also implementable in a modular way.
- Each box should have a `bprop` method, which computes the loss gradient w.r.t. its parents, given the loss gradient w.r.t. to the output.
- Can make use of cached computation done during the `fprop` method
- Calling the `bprop` method in reverse order yields backpropagation (only needs to reach the parameters)



Many software toolkits for deep learning

- Tensorflow
- Torch
- Pytorch
- MXNet
- Keras
- JAX
- ...

All implement automatic differentiation.

The key ingredients of SGD

- The loss function $L(f(\boldsymbol{x}_i;\boldsymbol{\theta}), y_i)$; ✓
- A procedure for computing its gradient $\nabla_{\boldsymbol{\theta}} L(f(\boldsymbol{x}_i;\boldsymbol{\theta}), y_i)$; ✓
- The regularizer $\Omega(\boldsymbol{\theta})$; next
- ... its gradients, $\nabla_{\boldsymbol{\theta}}\Omega(\boldsymbol{\theta})$. next

Regularization

- Objective function to be minimized:

$$\mathcal{L}(\boldsymbol{\theta}) := \lambda\Omega(\boldsymbol{\theta}) + \frac{1}{N}\sum_{n=1}^{N} L(\boldsymbol{f}(\boldsymbol{x}_i;\boldsymbol{\theta}), y_i)$$

- We will next focus on the regularizer and its gradient

Regularization

- Objective function to be minimized:

$$\mathcal{L}(\boldsymbol{\theta}) := \lambda\Omega(\boldsymbol{\theta}) + \frac{1}{N}\sum_{n=1}^{N} L(\boldsymbol{f}(\boldsymbol{x}_i;\boldsymbol{\theta}), y_i)$$

- We will next focus on the regularizer and its gradient
- We will study:
  - ✓ ℓ_2 regularization (weight decay);
  - ✓ ℓ_1 regularization (LASSO-type);
  - ✓ dropout regularization, which doesn't have the form above.

ℓ_2 regularization

- The biases $\boldsymbol{b}^{(1)}, ..., \boldsymbol{b}^{(L+1)}$ are not regularized; only the weights:

$$\Omega(\boldsymbol{\theta}) = \frac{1}{2} \sum_{\ell=1}^{L+1} \|\boldsymbol{W}^{(\ell)}\|_2^2$$

ℓ_2 regularization

- The biases $\boldsymbol{b}^{(1)}, ..., \boldsymbol{b}^{(L+1)}$ are not regularized; only the weights:

$$\Omega(\boldsymbol{\theta}) = \frac{1}{2} \sum_{\ell=1}^{L+1} \|\boldsymbol{W}^{(\ell)}\|_2^2$$

- Equivalent to a Gaussian prior on the weights

ℓ_2 regularization

- The biases $\boldsymbol{b}^{(1)}, ..., \boldsymbol{b}^{(L+1)}$ are not regularized; only the weights:

$$\Omega(\boldsymbol{\theta}) = \frac{1}{2} \sum_{\ell=1}^{L+1} \|\boldsymbol{W}^{(\ell)}\|_2^2$$

- Equivalent to a Gaussian prior on the weights

- Gradient of this regularizer is: $\nabla_{\boldsymbol{W}^{(\ell)}} \Omega(\boldsymbol{\theta}) = \boldsymbol{W}^{(\ell)}$

ℓ_2 regularization

- The biases $\boldsymbol{b}^{(1)}, ..., \boldsymbol{b}^{(L+1)}$ are not regularized; only the weights:
$$\Omega(\boldsymbol{\theta}) = \frac{1}{2} \sum_{\ell=1}^{L+1} \|\boldsymbol{W}^{(\ell)}\|_2^2$$

- Equivalent to a Gaussian prior on the weights

- Gradient of this regularizer is: $\nabla_{\boldsymbol{W}^{(\ell)}} \Omega(\boldsymbol{\theta}) = \boldsymbol{W}^{(\ell)}$

- Weight decay effect
$$\begin{aligned}
\boldsymbol{W}^{(\ell)} & \leftarrow \boldsymbol{W}^{(\ell)} - \eta \nabla_{\boldsymbol{W}^{(\ell)}} \mathcal{L}_i(\boldsymbol{\theta}) \\
& = \boldsymbol{W}^{(\ell)} - \eta(\lambda \nabla_{\boldsymbol{W}^{(\ell)}} \Omega(\boldsymbol{\theta}) + \nabla_{\boldsymbol{W}^{(\ell)}} L(f(\boldsymbol{x}_i; \boldsymbol{\theta}), y_i)) \\
& = \underbrace{(1 - \eta\lambda)}_{<1} \boldsymbol{W}^{(\ell)} - \eta \nabla_{\boldsymbol{W}^{(\ell)}} L(f(\boldsymbol{x}_i; \boldsymbol{\theta}), y_i)
\end{aligned}$$

ℓ_1 regularization

- The biases $\boldsymbol{b}^{(1)}, ..., \boldsymbol{b}^{(L+1)}$ are not regularized; only the weights:

$$\Omega(\boldsymbol{\theta}) = \sum_{\ell} \|\boldsymbol{W}^{(\ell)}\|_1 = \sum_{\ell} \sum_{ij} |W_{ij}^{(\ell)}|$$

ℓ_1 regularization

- The biases $\boldsymbol{b}^{(1)}, ..., \boldsymbol{b}^{(L+1)}$ are not regularized; only the weights:

$$\Omega(\boldsymbol{\theta}) = \sum_{\ell} \|\boldsymbol{W}^{(\ell)}\|_1 = \sum_{\ell} \sum_{ij} |W_{ij}^{(\ell)}|$$

- Equivalent to Laplacian prior on the weights

ℓ_1 regularization

- The biases $\boldsymbol{b}^{(1)}, ..., \boldsymbol{b}^{(L+1)}$ are not regularized; only the weights:

$$\Omega(\boldsymbol{\theta}) = \sum_{\ell} \|\boldsymbol{W}^{(\ell)}\|_1 = \sum_{\ell} \sum_{ij} |W_{ij}^{(\ell)}|$$

- Equivalent to Laplacian prior on the weights

- Gradient is: $\nabla_{\boldsymbol{W}^{(\ell)}} \Omega(\boldsymbol{\theta}) = \operatorname{sign}(\boldsymbol{W}^{(\ell)})$

ℓ_1 regularization

- The biases $\boldsymbol{b}^{(1)}, \ldots, \boldsymbol{b}^{(L+1)}$ are not regularized; only the weights:

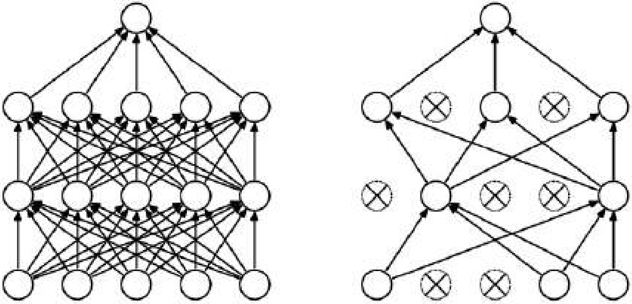
$$\Omega(\boldsymbol{\theta}) = \sum_{\ell} \|\boldsymbol{W}^{(\ell)}\|_1 = \sum_{\ell} \sum_{ij} |W_{ij}^{(\ell)}|$$

- Equivalent to Laplacian prior on the weights

- Gradient is: $\nabla_{\boldsymbol{W}^{(\ell)}} \Omega(\boldsymbol{\theta}) = \operatorname{sign}(\boldsymbol{W}^{(\ell)})$

- Promotes sparsity of the weights

Dropout regularization



(a) Standard Neural Net

(b) After applying dropout.

During training, remove some hidden units, chosen at random

Srivastava, Hinton, Krizhevsky, Sutskever, and Salakhutdinov (2014).

Dropout regularization

- Each hidden unit output is set to 0 with probability p (e.g. $p = 0.3$)

Dropout regularization

- Each hidden unit output is set to 0 with probability p (e.g. $p = 0.3$)
- Prevents hidden units from co-adapting to other units, forcing them to be more generally useful.

Dropout regularization

- Each hidden unit output is set to 0 with probability p (e.g. $p = 0.3$)
- Prevents hidden units from co-adapting to other units, forcing them to be more generally useful.
- Most common choice: inverted dropout: the output of the units that were not dropped is divided by $1 - p$

Dropout regularization

- Each hidden unit output is set to 0 with probability p (e.g. $p = 0.3$)
- Prevents hidden units from co-adapting to other units, forcing them to be more generally useful.
- Most common choice: inverted dropout: the output of the units that were not dropped is divided by $1 - p$
- This ensures that the expected value of the output remains the same during training and inference.

Backpropagation with dropout

Compute output gradient (before activation):

$$\nabla_{\boldsymbol{z}^{(L+1)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y) = -(\mathbf{1}_y - \boldsymbol{f}(\boldsymbol{x}))$$

for ℓ from $L+1$ to 1 **do**

Compute gradients of hidden layer parameters:

$$\nabla_{\boldsymbol{W}^{(\ell)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y) = \nabla_{\boldsymbol{z}^{(\ell)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y) \underbrace{\boldsymbol{h}^{(\ell-1)^\top}}_{\text{includes mask } \boldsymbol{m}^{(\ell-1)}}$$

$$\nabla_{\boldsymbol{b}^{(\ell)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y) = \nabla_{\boldsymbol{z}^{(\ell)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y)$$

Compute gradient of hidden layer below:

$$\nabla_{\boldsymbol{h}^{(\ell-1)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y) = \boldsymbol{W}^{(\ell)^\top} \nabla_{\boldsymbol{z}^{(\ell)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y)$$

Compute gradient of hidden layer below (before activation):

$$\nabla_{\boldsymbol{z}^{(\ell-1)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y) = \nabla_{\boldsymbol{h}^{(\ell-1)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y) \odot \boldsymbol{g}'(\boldsymbol{z}^{(\ell-1)}) \odot \boldsymbol{m}^{(\ell-1)}$$

end for

Tricks of the trade: Initialization

- Biases: initialize at zero

Tricks of the trade: Initialization

- Biases: initialize at zero
- Weights:

Tricks of the trade: Initialization

- Biases: initialize at zero

- Weights:

  ✓ Cannot initialize to zero with $\tanh$ activation (gradients would also be all zero and we would be at saddle point)

Tricks of the trade: Initialization

- Biases: initialize at zero

- Weights:

  ✓ Cannot initialize to zero with $\tanh$ activation (gradients would also be all zero and we would be at saddle point)

  ✓ Cannot initialize the weights to the same value (need to break the symmetry)

Tricks of the trade: Initialization

- Biases: initialize at zero
- Weights:
  - ✓ Cannot initialize to zero with $\tanh$ activation (gradients would also be all zero and we would be at saddle point)
  - ✓ Cannot initialize the weights to the same value (need to break the symmetry)
  - ✓ Random initialization (Gaussian, uniform), sampling around 0 to break symmetry, or 'Glorot initialization" (Glorot and Bengio, 2010)

$$\boldsymbol{W}_{i,j}^{(\ell)} \sim U[-t, t], \text{ with } t = \frac{\sqrt{6}}{\sqrt{K^{(\ell)} + K^{(\ell-1)}}}$$

Tricks of the trade: Initialization

- Biases: initialize at zero

- Weights:

  ✓ Cannot initialize to zero with $\tanh$ activation (gradients would also be all zero and we would be at saddle point)

  ✓ Cannot initialize the weights to the same value (need to break the symmetry)

  ✓ Random initialization (Gaussian, uniform), sampling around 0 to break symmetry, or 'Glorot initialization" (Glorot and Bengio, 2010)

$$\boldsymbol{W}_{i,j}^{(\ell)} \sim U[-t, t], \text{ with } t = \frac{\sqrt{6}}{\sqrt{K^{(\ell)} + K^{(\ell-1)}}}$$

  ✓ For ReLU activations, the mean should be a small positive number

More tricks of the trade

- Hyperparameter tuning (just use Optuna)
- Normalization of the data
- Decaying the learning rate
- Mini-batches size
- Adaptive learning rates
- Gradient checking
- Debug on small datasets

Outline

1. Brief History of Deep Learning (Before LLMs)
2. From models of neurons to artificial neural networks
3. **Deep Learning via Empirical Risk Minimization**
   - Gradient Descent and Stochastic Gradient Descent
   - Gradient Backpropagation and Autodiff
   - Better optimization: momentum, AdaGrad, RMSProp, Adam
4. Convolutional Neural Networks

Momentum

- Momentum: remember the previous step, combine it in the update:

$$\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} - \alpha_t \boldsymbol{g}(\boldsymbol{\theta}_{t-1}) + \gamma_t(\boldsymbol{\theta}_{t-1} - \boldsymbol{\theta}_{t-2});$$

$\boldsymbol{g}(\boldsymbol{\theta}_t)$ is the gradient estimate (batch, single sample, minibatch).

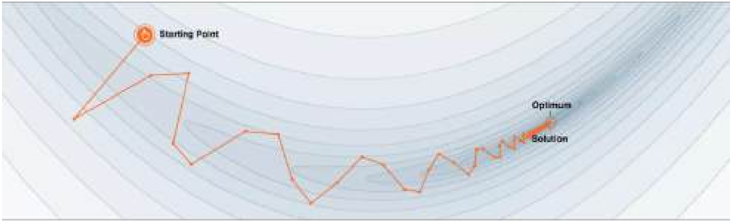
Momentum

- Momentum: remember the previous step, combine it in the update:

$$\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} - \alpha_t \boldsymbol{g}(\boldsymbol{\theta}_{t-1}) + \gamma_t(\boldsymbol{\theta}_{t-1} - \boldsymbol{\theta}_{t-2});$$

$\boldsymbol{g}(\boldsymbol{\theta}_t)$ is the gradient estimate (batch, single sample, minibatch).

- Advantage: reduces the update in directions with changing gradients; increases the update in directions with stable gradient.



Adaptive gradient (AdaGrad)

- AdaGrad[1]: use separate step sizes for each component $\theta_{j,t}$ of $\boldsymbol{\theta}_t$.

[1] J. Duchi, E. Hazan, Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization", Jour. of Machine Learning Research, vo. 12, 2011

Adaptive gradient (AdaGrad)

- AdaGrad[1]: use separate step sizes for each component $\theta_{j,t}$ of $\boldsymbol{\theta}_t$.
- Scale the update of each component (ε for numerical stability)

$$\theta_{j,t} = \theta_{j,t-1} - \frac{\alpha}{\sqrt{G_{j,t-1} + \varepsilon}}\, g_j(\boldsymbol{\theta}_{t-1})$$

[1] J. Duchi, E. Hazan, Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization", Jour. of Machine Learning Research, vo. 12, 2011

Adaptive gradient (AdaGrad)

- AdaGrad[1]: use separate step sizes for each component $\theta_{j,t}$ of $\boldsymbol{\theta}_t$.
- Scale the update of each component (ε for numerical stability)

$$\theta_{j,t} = \theta_{j,t-1} - \frac{\alpha}{\sqrt{G_{j,t-1} + \varepsilon}} \, g_j(\boldsymbol{\theta}_{t-1})$$

- where $G_{j,t}$ accumulates all the squared gradient values in component t

$$G_{j,t} = \sum_{t'=1}^{t} \big(g_j(\boldsymbol{\theta}_{t'})\big)^2 = G_{j,t-1} + \big(g_j(\boldsymbol{\theta}_t)\big)^2$$

[1] J. Duchi, E. Hazan, Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization", Jour. of Machine Learning Research, vo. 12, 2011

Adaptive gradient (AdaGrad)

- AdaGrad[1]: use separate step sizes for each component $\theta_{j,t}$ of $\boldsymbol{\theta}_t$.
- Scale the update of each component (ε for numerical stability)

$$\theta_{j,t} = \theta_{j,t-1} - \frac{\alpha}{\sqrt{G_{j,t-1} + \varepsilon}}\, g_j(\boldsymbol{\theta}_{t-1})$$

- where $G_{j,t}$ accumulates all the squared gradient values in component t

$$G_{j,t} = \sum_{t'=1}^{t} \big(g_j(\boldsymbol{\theta}_{t'})\big)^2 = G_{j,t-1} + \big(g_j(\boldsymbol{\theta}_t)\big)^2$$

- Advantages: robust to choice of α and to different parameter scaling.

[1] J. Duchi, E. Hazan, Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization", Jour. of Machine Learning Research, vo. 12, 2011

Adaptive gradient (AdaGrad)

- AdaGrad[1]: use separate step sizes for each component $\theta_{j,t}$ of $\boldsymbol{\theta}_t$.
- Scale the update of each component (ε for numerical stability)

$$\theta_{j,t} = \theta_{j,t-1} - \frac{\alpha}{\sqrt{G_{j,t-1} + \varepsilon}}\, g_j(\boldsymbol{\theta}_{t-1})$$

- where $G_{j,t}$ accumulates all the squared gradient values in component t

$$G_{j,t} = \sum_{t'=1}^{t} \big(g_j(\boldsymbol{\theta}_{t'})\big)^2 = G_{j,t-1} + \big(g_j(\boldsymbol{\theta}_t)\big)^2$$

- Advantages: robust to choice of α and to different parameter scaling.
- Drawbacks: step size vanishes, because $G_{j,t} \geq G_{j,t-1}$.

[1] J. Duchi, E. Hazan, Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization", Jour. of Machine Learning Research, vo. 12, 2011

Root Mean Square Propagation (RMSProp)

- RMSProp[2] addresses the vanishing learning issue.

[2] Presented by G. Hinton in a Coursera lecture.

Root Mean Square Propagation (RMSProp)

- RMSProp[2] addresses the vanishing learning issue.
- Same scaled update of each component

$$\theta_{j,t} = \theta_{j,t-1} - \frac{\alpha}{\sqrt{G_{j,t-1} + \varepsilon}}\, g_j(\boldsymbol{\theta}_{t-1})$$

[2] Presented by G. Hinton in a Coursera lecture.

Root Mean Square Propagation (RMSProp)

- RMSProp[2] addresses the vanishing learning issue.
- Same scaled update of each component

$$\theta_{j,t} = \theta_{j,t-1} - \frac{\alpha}{\sqrt{G_{j,t-1} + \varepsilon}}\, g_j(\boldsymbol{\theta}_{t-1})$$

- Use a forgetting/decay factor γ (typically 0.9),

$$G_{j,t} = \gamma\, G_{j,t-1} + (1-\gamma)\big(g_j(\boldsymbol{\theta}_t)\big)^2$$

[2] Presented by G. Hinton in a Coursera lecture.

Root Mean Square Propagation (RMSProp)

- RMSProp[2] addresses the vanishing learning issue.
- Same scaled update of each component

$$\theta_{j,t} = \theta_{j,t-1} - \frac{\alpha}{\sqrt{G_{j,t-1} + \varepsilon}}\, g_j(\boldsymbol{\theta}_{t-1})$$

- Use a forgetting/decay factor γ (typically 0.9),

$$G_{j,t} = \gamma\, G_{j,t-1} + (1-\gamma)\big(g_j(\boldsymbol{\theta}_t)\big)^2$$

- Now, $G_{j,t}$ may be smaller than $G_{j,t-1}$.

[2] Presented by G. Hinton in a Coursera lecture.

Root Mean Square Propagation (RMSProp)

- RMSProp[2] addresses the vanishing learning issue.
- Same scaled update of each component

$$\theta_{j,t} = \theta_{j,t-1} - \frac{\alpha}{\sqrt{G_{j,t-1} + \varepsilon}}\, g_j(\boldsymbol{\theta}_{t-1})$$

- Use a forgetting/decay factor γ (typically 0.9),

$$G_{j,t} = \gamma\, G_{j,t-1} + (1-\gamma)\big(g_j(\boldsymbol{\theta}_t)\big)^2$$

- Now, $G_{j,t}$ may be smaller than $G_{j,t-1}$.
- Advantages: robust to choice of α (typically 0.01 or 0.001); robust to different parameter scaling.

[2] Presented by G. Hinton in a Coursera lecture.

Adam: adaptive momentum estimation

- Adam[3]: combines aspects of RMSProp and momentum.

[3] D. Kingma, J. Ba, "Adam: A Method for Stochastic Optimization", *International Conference for Learning Representations*, 2015. (more than 220000 citations)

Adam: adaptive momentum estimation

- Adam[3]: combines aspects of RMSProp and momentum.
- Separate moving averages of gradient and squared gradient.

[3] D. Kingma, J. Ba, "Adam: A Method for Stochastic Optimization", *International Conference for Learning Representations*, 2015. (more than 220000 citations)

Adam: adaptive momentum estimation

- Adam[3]: combines aspects of RMSProp and momentum.
- Separate moving averages of gradient and squared gradient.
- Initial: $\mathbf{m}_t = 0$, $\boldsymbol{v}_t = 0$ (typical $\beta_1 = 0.9, \beta_2 = 0.999, \alpha = 10^{-3}$):

$$\begin{aligned}
\mathbf{m}_t &= \beta_1 \mathbf{m}_{t-1} + (1-\beta_1)\boldsymbol{g}_t \\
\boldsymbol{v}_t &= \beta_2 \boldsymbol{v}_{t-1} + (1-\beta_2)\boldsymbol{g}_t^2 \\
\hat{\mathbf{m}}_t &= \mathbf{m}_t/(1-\beta_1^t) && \text{(bias correction due to } \mathbf{m}_0 = 0\text{)} \\
\hat{\boldsymbol{v}}_t &= \boldsymbol{v}_t/(1-\beta_2^t) && \text{(bias correction due to } \boldsymbol{v}_0 = 0\text{)} \\
\boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \alpha \frac{\hat{\mathbf{m}}_t}{\sqrt{\hat{\boldsymbol{v}}_t + \varepsilon}} && \text{(component-wise)}
\end{aligned}$$

[3] D. Kingma, J. Ba, "Adam: A Method for Stochastic Optimization", *International Conference for Learning Representations*, 2015. (more than 220000 citations)

Adam: adaptive momentum estimation

- Adam[3]: combines aspects of RMSProp and momentum.
- Separate moving averages of gradient and squared gradient.
- Initial: $\mathbf{m}_t = 0$, $\boldsymbol{v}_t = 0$ (typical $\beta_1 = 0.9, \beta_2 = 0.999, \alpha = 10^{-3}$):

$$\begin{aligned}
\mathbf{m}_t &= \beta_1 \mathbf{m}_{t-1} + (1-\beta_1)\boldsymbol{g}_t \\
\boldsymbol{v}_t &= \beta_2 \boldsymbol{v}_{t-1} + (1-\beta_2)\boldsymbol{g}_t^2 \\
\hat{\mathbf{m}}_t &= \mathbf{m}_t/(1-\beta_1^t) && \text{(bias correction due to } \mathbf{m}_0 = 0\text{)} \\
\hat{\boldsymbol{v}}_t &= \boldsymbol{v}_t/(1-\beta_2^t) && \text{(bias correction due to } \boldsymbol{v}_0 = 0\text{)} \\
\boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \alpha \frac{\hat{\mathbf{m}}_t}{\sqrt{\hat{\boldsymbol{v}}_t + \varepsilon}} && \text{(component-wise)}
\end{aligned}$$

- Advantages: Computationally efficient, low memory usage, suitable for large datasets and many parameters.

[3] D. Kingma, J. Ba, "Adam: A Method for Stochastic Optimization", *International Conference for Learning Representations*, 2015. (more than 220000 citations)

Adam: adaptive momentum estimation

- Adam[3]: combines aspects of RMSProp and momentum.
- Separate moving averages of gradient and squared gradient.
- Initial: $\mathbf{m}_t = 0$, $\boldsymbol{v}_t = 0$ (typical $\beta_1 = 0.9, \beta_2 = 0.999, \alpha = 10^{-3}$):

$$
\begin{aligned}
\mathbf{m}_t &= \beta_1 \mathbf{m}_{t-1} + (1-\beta_1)\boldsymbol{g}_t \\
\boldsymbol{v}_t &= \beta_2 \boldsymbol{v}_{t-1} + (1-\beta_2)\boldsymbol{g}_t^2 \\
\hat{\mathbf{m}}_t &= \mathbf{m}_t / (1-\beta_1^t) && \text{(bias correction due to } \mathbf{m}_0 = 0) \\
\hat{\boldsymbol{v}}_t &= \boldsymbol{v}_t / (1-\beta_2^t) && \text{(bias correction due to } \boldsymbol{v}_0 = 0) \\
\boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \alpha \frac{\hat{\mathbf{m}}_t}{\sqrt{\hat{\boldsymbol{v}}_t + \varepsilon}} && \text{(component-wise)}
\end{aligned}
$$

- Advantages: Computationally efficient, low memory usage, suitable for large datasets and many parameters.
- Drawbacks: Possible convergence issues with noisy gradient estimates.

[3] D. Kingma, J. Ba, "Adam: A Method for Stochastic Optimization", *International Conference for Learning Representations*, 2015. (more than 220000 citations)

Outline

Convolutional networks (CNN)

- How is a convolutional network different from a standard network?

Convolutional networks (CNN)

- How is a convolutional network different from a standard network?

  ...it is just a NN with a special connectivity structure

Convolutional networks (CNN)

- How is a convolutional network different from a standard network?

  ...it is just a NN with a special connectivity structure

- Convolutional networks have convolutional layers

Convolutional networks (CNN)

- How is a convolutional network different from a standard network?

  ...it is just a NN with a special connectivity structure

- Convolutional networks have convolutional layers

- How are they different from a fully connected layers?

Neocognitron (Fukushima, 1982)

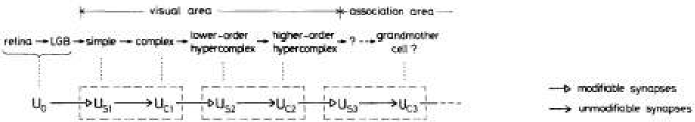


Fig. 1. Correspondence between the hierarchy model by Hubel and Wiesel, and the neural network of the neocognitron

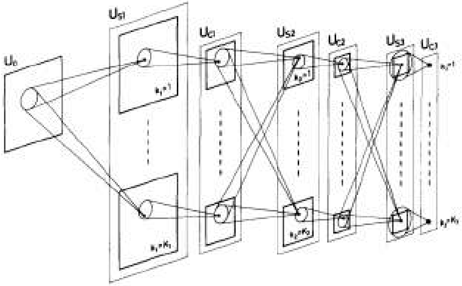
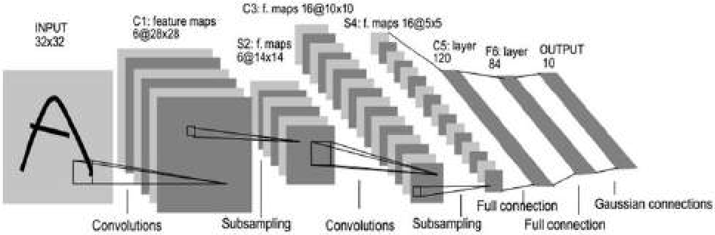
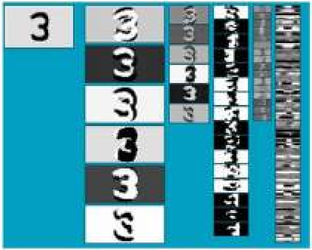


Fig. 2. Schematic diagram illustrating the interconnections between layers in the neocognitron

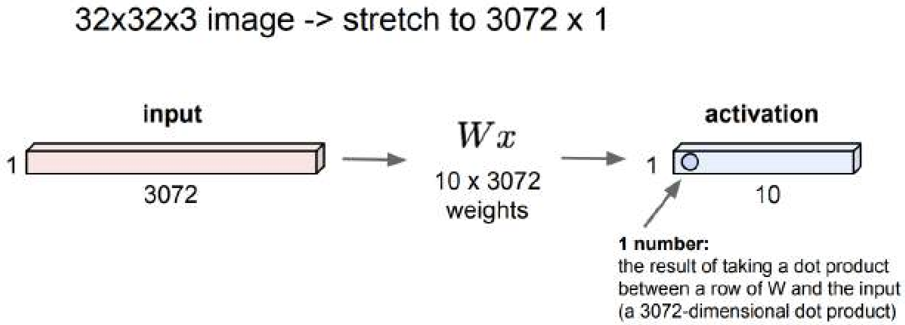
- Inspired by the multi-stage hierarchy of the visual nervous system (Hubel and Wiesel, 1965).

ConvNet (LeNet-5) (LeCun, 1998)





Fully connected layer

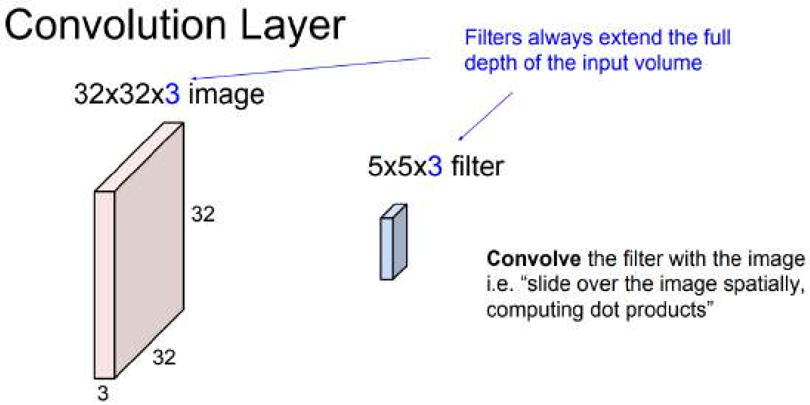


(Credits: Fei-Fei Li, Johnson, Yeung)

- All activations depend on all inputs.

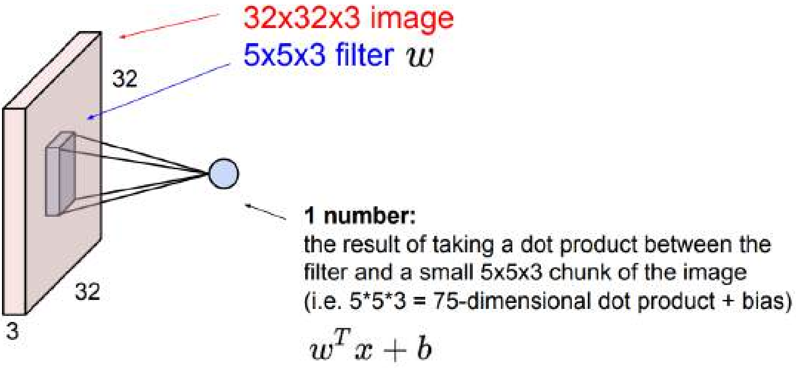
Convolutional layer

- Don't stretch/reshape: preserve the spacial structure!



(Credits: Fei-Fei Li, Johnson, Yeung)

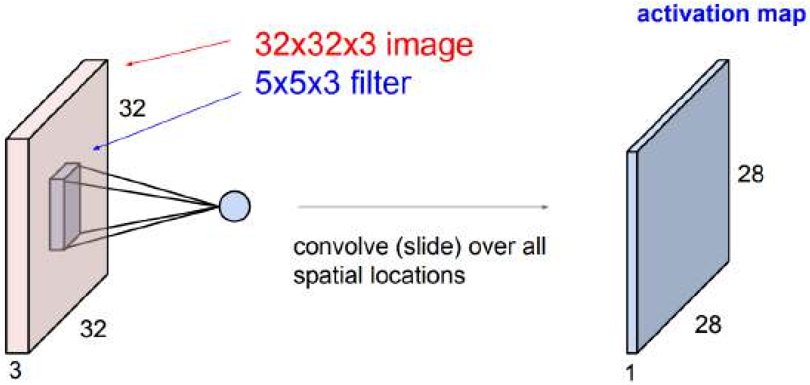
Convolutional layer



(Credits: Fei-Fei Li, Johnson, Yeung)

Convolutional layer

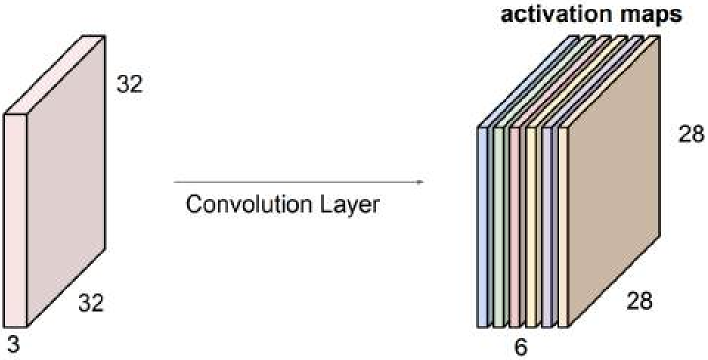
- Apply the same filter to all spatial locations (28x28 times, why?):



(Credits: Fei-Fei Li, Johnson, Yeung)

Convolutional layer

- For example, **6** 5x5x3 filters yield **6** activation maps:



(Credits: Fei-Fei Li, Johnson, Yeung)

- Stack these up to get a new "image" of size 28x28x6!

Image size, filter size, stride, channels

- Stride: shift in pixels between two consecutive windows. In the previous illustrations, stride $= 1$.

Image size, filter size, stride, channels

- Stride: shift in pixels between two consecutive windows. In the previous illustrations, stride $= 1$.

- Number of channels: number of filters used in each layer.

Image size, filter size, stride, channels

- Stride: shift in pixels between two consecutive windows. In the previous illustrations, stride $= 1$.

- Number of channels: number of filters used in each layer.

- Given an $N \times N \times D$ image, $F \times F \times D$ filters, K channels, and stride S, the resulting output will be of size $M \times M \times K$, where

$$M = (N - F)/S + 1$$

Image size, filter size, stride, channels

- Stride: shift in pixels between two consecutive windows. In the previous illustrations, stride $= 1$.

- Number of channels: number of filters used in each layer.

- Given an $N \times N \times D$ image, $F \times F \times D$ filters, K channels, and stride S, the resulting output will be of size $M \times M \times K$, where

$$M = (N - F)/S + 1$$

- Examples:
  - ✓ $N = 32$, $D = 3$, $F = 5$, $K = 6$, $S = 1$ results in an $28 \times 28 \times 6$ output
  - ✓ $N = 32$, $D = 3$, $F = 5$, $K = 6$, $S = 3$ results in an $10 \times 10 \times 6$ output

Image size, filter size, stride, channels

- Stride: shift in pixels between two consecutive windows. In the previous illustrations, stride $= 1$.

- Number of channels: number of filters used in each layer.

- Given an $N \times N \times D$ image, $F \times F \times D$ filters, K channels, and stride S, the resulting output will be of size $M \times M \times K$, where

$$M = (N - F)/S + 1$$

- Examples:
  - ✓ $N = 32$, $D = 3$, $F = 5$, $K = 6$, $S = 1$ results in an $28 \times 28 \times 6$ output
  - ✓ $N = 32$, $D = 3$, $F = 5$, $K = 6$, $S = 3$ results in an $10 \times 10 \times 6$ output

- Padding: append zeros around the images. Common padding size: $(F - 1)/2$, which preserves spatial size: $M = N$.

CNNs and convolutions

- Why is this called "convolutional"?
- The convolution of a signal x and a filter w, denoted $x * w$, is:

$$h[t] = (x * w)[t] = \sum_{a=-\infty}^{\infty} x[t-a]\, w[a].$$

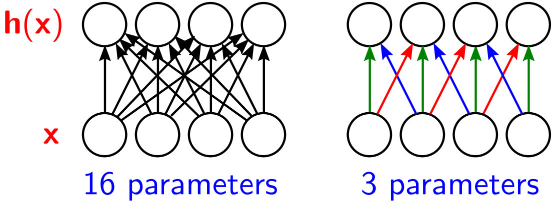
CNNs and convolutions

- Why is this called "convolutional"?

- The convolution of a signal x and a filter w, denoted $x * w$, is:

$$h[t] = (x * w)[t] = \sum_{a=-\infty}^{\infty} x[t-a]\, w[a].$$

- Basic idea: sparse/local connectivity and parameter tying/sharing.



Convolutions with padding

- Expression above is for infinite-support signal x and filter w.

Convolutions with padding

- Expression above is for infinite-support signal x and filter w.

- Finite support: $x = (x[0], ..., x[N-1])$; $w = (w[-E], ..., w[E])$ $(F = 2E + 1)$

$$h[t] = (x * w)[t] = \sum_{a=-E}^{E} w[a]x[t-a], \quad \text{for } t = E, ..., N-1-E$$

The result has support of size $N - 1 - E - E + 1 = N - 2E = N - F + 1$.

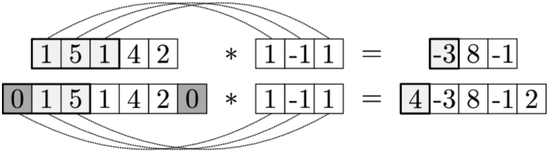
Convolutions with padding

- Expression above is for infinite-support signal x and filter w.
- Finite support: $x = (x[0], ..., x[N-1])$; $w = (w[-E], ..., w[E])$ $(F = 2E + 1)$

$$h[t] = (x * w)[t] = \sum_{a=-E}^{E} w[a]x[t-a], \quad \text{for } t = E, ..., N-1-E$$

The result has support of size $N - 1 - E - E + 1 = N - 2E = N - F + 1$.

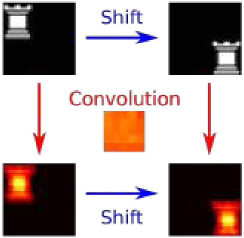
- Padding: append $E = (F-1)/2$ zeros at each side of x.



(Slide credit to Rob Fergus)

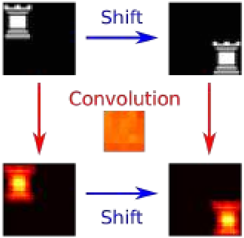
Convolutions and parameter tying

- Leads to translation/shift equivariance (a form of inductive bias)



Convolutions and parameter tying

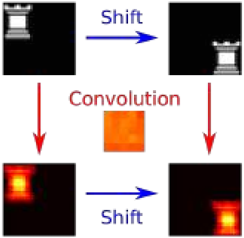
- Leads to translation/shift equivariance (a form of inductive bias)



- Advantages of sharing/tying parameters:
  - ✓ Reduces the number of parameters to be learned.
  - ✓ Allows dealing with arbitrary large, variable-length, inputs.

Convolutions and parameter tying

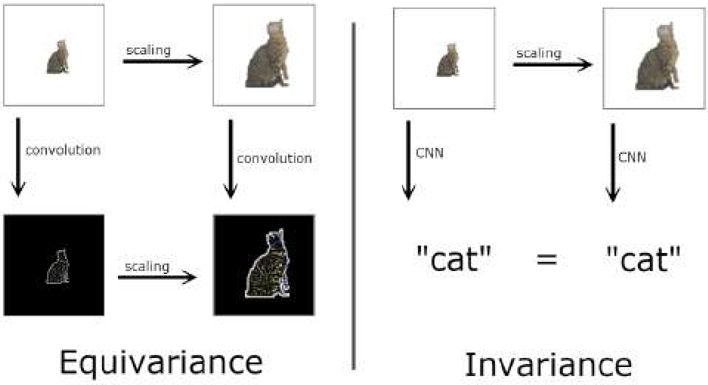
- Leads to translation/shift equivariance (a form of inductive bias)



- Advantages of sharing/tying parameters:
  - ✓ Reduces the number of parameters to be learned.
  - ✓ Allows dealing with arbitrary large, variable-length, inputs.

- Can be used for 1D (signals, text, sequences,...), 2D (images, spatial distributions, ...), 3D (video, point clouds, ...), even graphs.

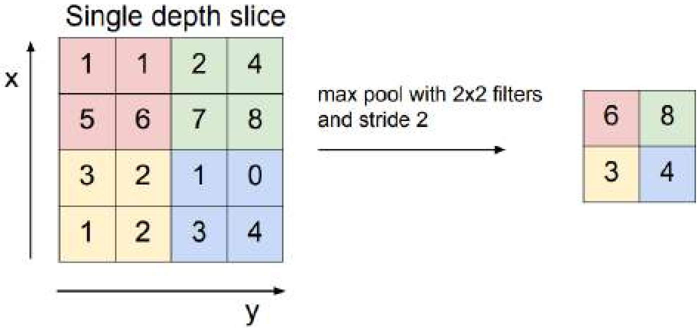
Equivariance and invariance



- Pooling layers provide invariance.

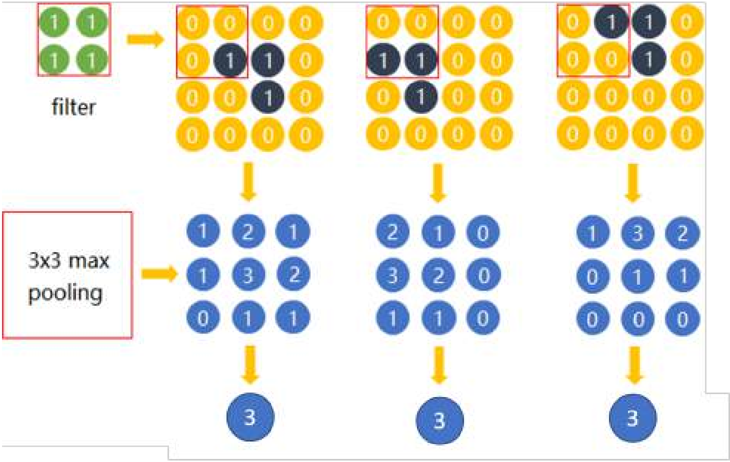
Pooling layer

- Makes the representations smaller, more manageable.
- Operates over each activation map (each channel) independently
- Example: max-pooling:

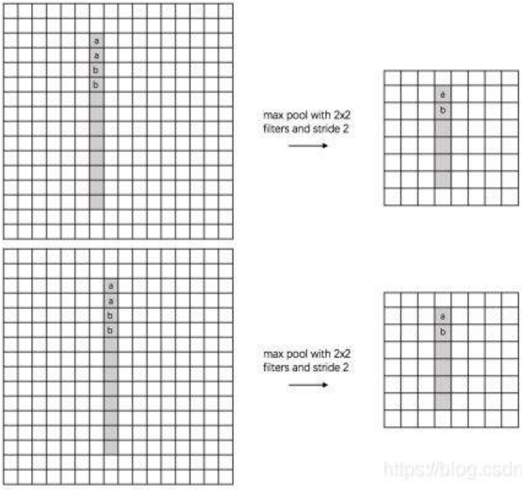


(Credits: Fei-Fei Li, Johnson, Yeung)

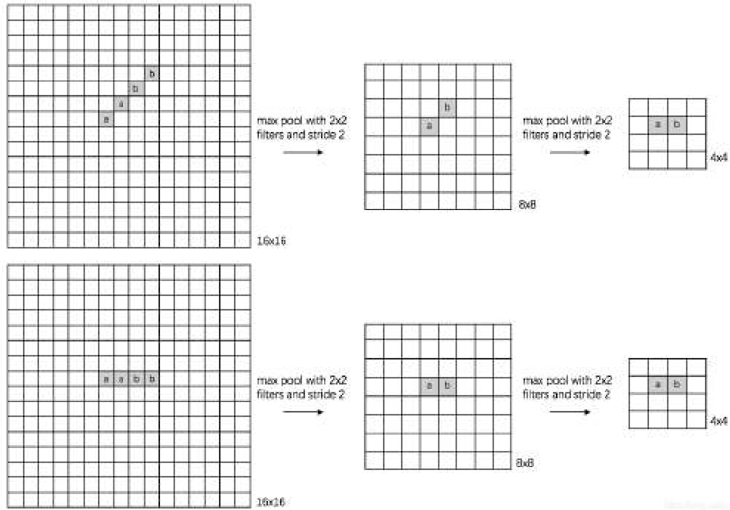
Max pooling: shift invariance



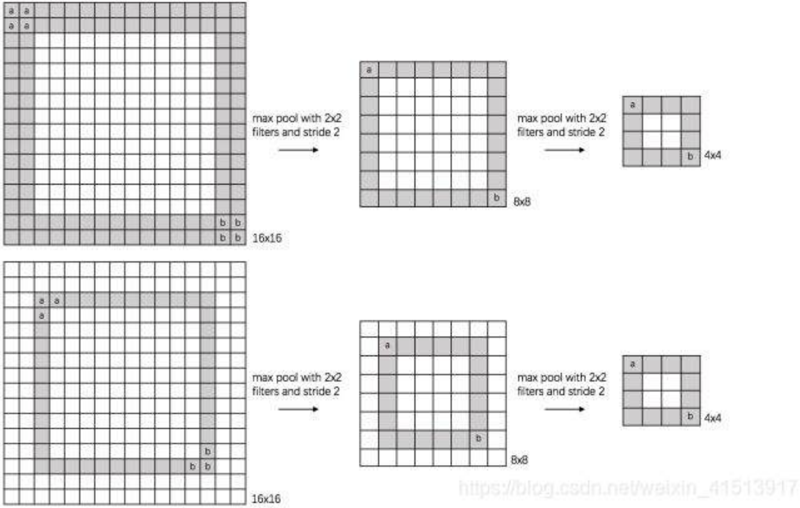
Max pooling: shift invariance (II)



Max pooling: rotation invariance

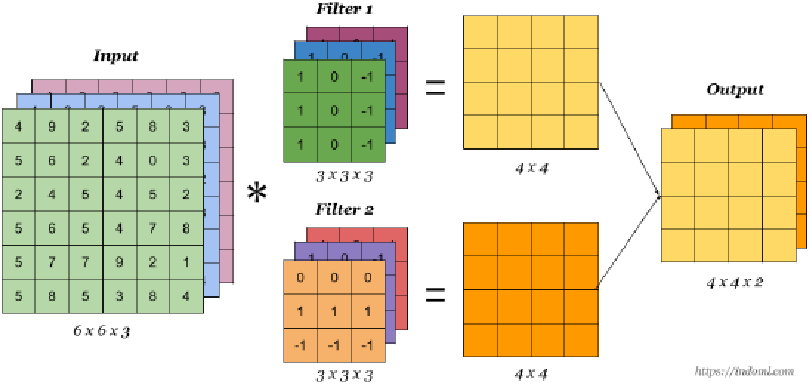


Max pooling: scale invariance



Multiple convolution filters: feature maps

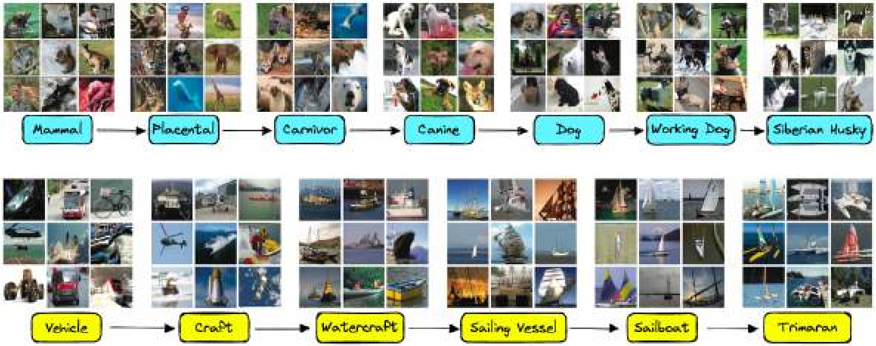
- Different filter for each channel, but keeping spatial invariance:



(Figure credit: Andrew Ng)

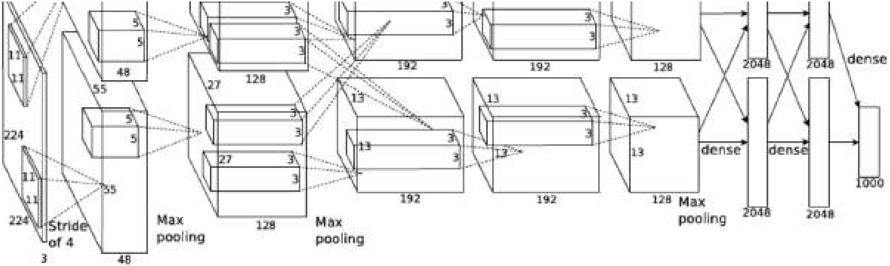
ImageNet dataset

- 14 million labelled images gathered (from the Internet)
- 22000 hierarchical classes
- ImageNet Large Scale Visual
- Recognition Challenge (ILSVRC)
- **Classification:** 1,000 object classes, 1.4M/50K/100K images
- **Detection:** 200 object classes, 400K/20K/40K images



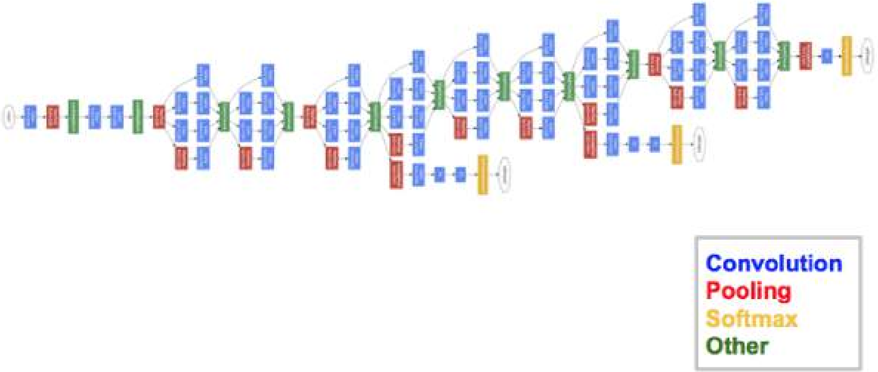
AlexNet (Krizhevsky, Sutskever, Hinton, 2012)

- 54M parameters; 8 layers (5 conv, 3 fully-connected)
- Trained on 1.4M ImageNet images
- Trained on 2 GPUs for a week (50x speed-up over CPU)
- Dropout regularization
- Test error: 16.4% (second best team was 26.2%)



GoogLeNet

- GoogLeNet inception module: very deep convolutional network, fewer (5M) parameters



Residual networks (ResNets)

- Add skip-connections; tends to lead to more stable learning.

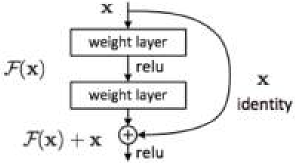


Figure 2. Residual learning: a building block.

(He, Zhang, Ren, Sun, 2016)

Residual networks (ResNets)

- Add skip-connections; tends to lead to more stable learning.

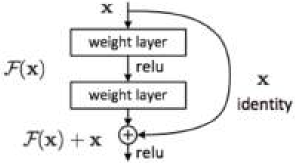


Figure 2. Residual learning: a building block.

(He, Zhang, Ren, Sun, 2016)

- Key (but not the only) motivation: mitigate vanishing gradients.

Residual networks (ResNets)

- Add skip-connections; tends to lead to more stable learning.

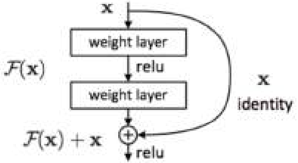


Figure 2. Residual learning: a building block.

(He, Zhang, Ren, Sun, 2016)

- Key (but not the only) motivation: mitigate vanishing gradients.

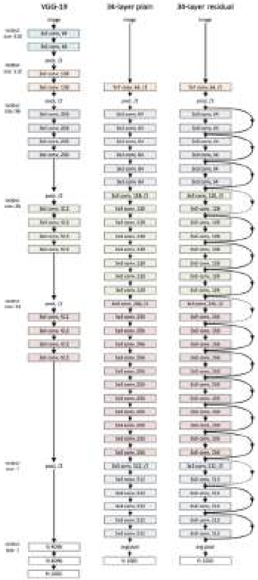
- With $H(\boldsymbol{x}) = \mathcal{F}(\boldsymbol{x}) + \lambda \boldsymbol{x}$, the gradient back-propagation becomes

$$\frac{\partial L}{\partial \boldsymbol{x}} = \frac{\partial L}{\partial H} \frac{\partial H}{\partial \boldsymbol{x}} = \frac{\partial L}{\partial H} \left(\frac{\partial \mathcal{F}}{\partial \boldsymbol{x}} + \lambda \right)$$

Residual networks (ResNets)

- Very deep network (34 layers here, but up to 152 layers!)

- VGG-19 ("Visual Geometry Group") by Simonyan and Zisserman (2014); 19 layers.



Residual networks (ResNets)

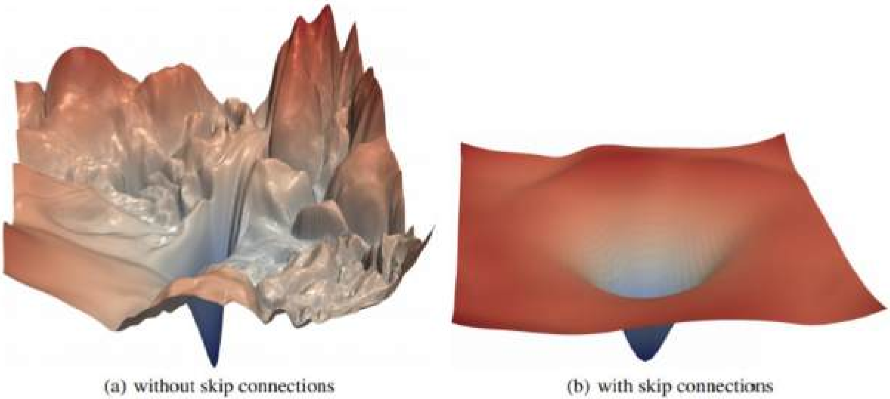


Figure 1: The loss surfaces of ResNet-56 with/without skip connections. The proposed filter normalization scheme is used to enable comparisons of sharpness/flatness between the two figures.

(Li, Xu, Taylor, Studer, Goldstein, 2018)

Beyond NNs and CNNs

- Other architectures have been proposed which offer alternatives to convolutions
- For example: transformers.
- These are somewhat similar to "dynamic convolutions".
- Covered in another lecture.

Visualization

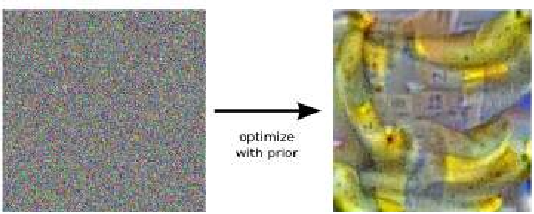
- **Idea:** Optimize input to maximize particular output

Visualization

- **Idea:** Optimize input to maximize particular output
- Depends on the initialization

Visualization

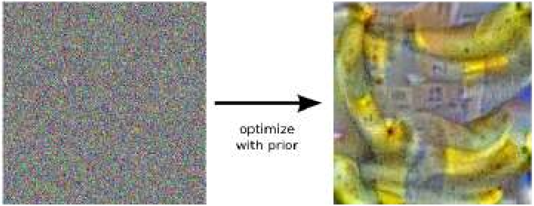
- **Idea:** Optimize input to maximize particular output
- Depends on the initialization
- Google DeepDream, maximizing "banana" output:



(from https://research.googleblog.com/2015/06/inceptionism-going-deeper-into-neural.html)

Visualization

- **Idea:** Optimize input to maximize particular output
- Depends on the initialization
- Google DeepDream, maximizing "banana" output:

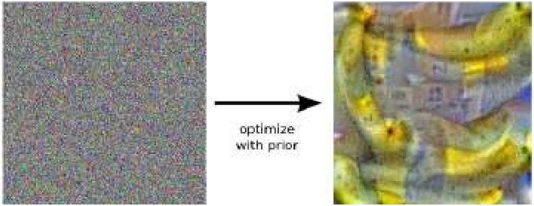


(from https://research.googleblog.com/2015/06/inceptionism-going-deeper-into-neural.html)

- Can also specify an inner layer and tune the input to maximize its activations: useful to see what kind of features it is representing.

Visualization

- **Idea:** Optimize input to maximize particular output
- Depends on the initialization
- Google DeepDream, maximizing "banana" output:



(from https://research.googleblog.com/2015/06/inceptionism-going-deeper-into-neural.html)

- Can also specify an inner layer and tune the input to maximize its activations: useful to see what kind of features it is representing.
- Specifying a higher layer produces more complex representations...

Adversarial attacks

- Can we perturb an input slightly to fool a classifier?
- For example: 1-pixel attacks
- Glass-box model: assumes access to the model
- Backpropagate to the inputs to find pixels which maximize the gradient
- There's also work for black-box adversarial attacks (don't have access to the model, but can query it).

(Credits: Su, Vargas, Sakurai (2018))

Even worse: perturb objects, not images

- Print the model of a turtle in a 3D printer.
- Perturbing the texture fools the model into thinking it's a rifle, regardless of the pose of the object!

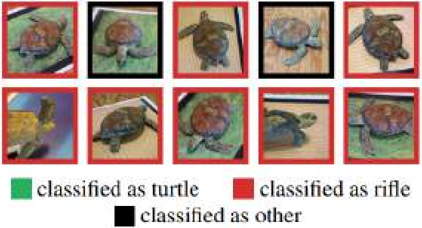
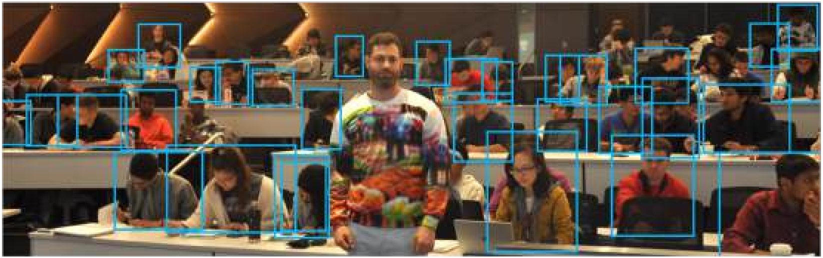


Figure 1. Randomly sampled poses of a 3D-printed turtle adversarially perturbed to classify as a rifle at every viewpoint[2]. An unperturbed model is classified correctly as a turtle nearly 100% of the time.

(Credits: Athalye, Engstrom, Ilyas, Kwok (2018))

- Neural networks may be very brittle!

The anti-detection sweater



Making an Invisibility Cloak: Real World Adversarial Attacks on Object Detectors

Zuxuan Wu[1,2], Ser-Nam Lim[2], Larry S. Davis[1], and Tom Goldstein[1,2]

[1]University of Maryland, College Park [2]Facebook AI

2020

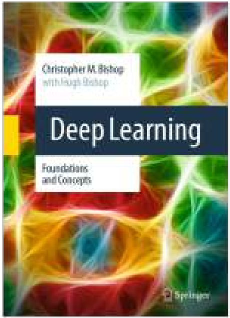
More to come in upcoming lectures...

We covered only the very basics of deep learning, ...

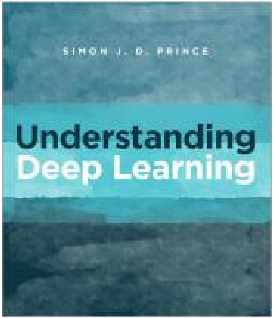
... much more in upcoming lectures:

- Sequence and language models: Noah Smith
- Transformers and large pre-trained models: Sweta Agrawal
- Deep learning for vision and language: Desmond Elliot

Recommended reading

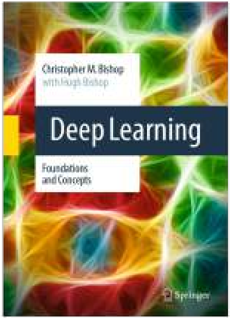


Springer, 2024
https://www.bishopbook.com/

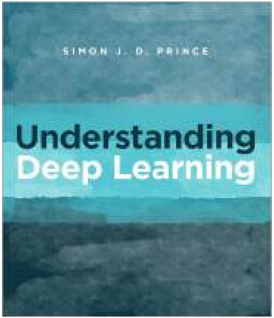


MIT Press, 2023
https://udlbook.github.io/udlbook/

Recommended reading



Springer, 2024
https://www.bishopbook.com/



MIT Press, 2023
https://udlbook.github.io/udlbook/

Thank you! Questions?