Introduction to (Convolutional) Neural Networks

Philipp Grohs

Summer School DL and Vis, Sept 2018
1 Motivation and Definition
2 Universal Approximation
3 Backpropagation
4 Stochastic Gradient Descent
5 The Basic Recipe
6 Going Deep
7 Convolutional Neural Networks
8 What I didn’t tell you
1 Motivation and Definition
Which Method to Choose?

We have seen linear Regression, kernel regression, regularization, K-PCA, K-SVM, ... and there exist a zillion other methods.
Which Method to Choose?

We have seen linear Regression, kernel regression, regularization, K-PCA, K-SVM, ... and there exist a zillion other methods.

Is there a universally best method?
No Free Lunch Theorem

"No Free Lunch" Theorem [Wolpert(1996)], Informal Version

Of course not!!

"Proof" of the "Theorem"

If \( \rho \) is completely arbitrary and nothing is known, we cannot possibly infer anything about \( \rho \) from samples \((x_i, y_i)\) for \( m_i = 1 \)...

Every algorithm will have a specific preference, for example as specified through the hypothesis class \( H \) – all "categories" are artificial!

We want our algorithm to reproduce the artificial categories produced by our brain – so let's build a hypothesis class that mimicks our thinking!
“No Free Lunch” Theorem [Wolpert(1996)], Informal Version

Of course not!!
"No Free Lunch" Theorem [Wolpert(1996)], Informal Version

Of course not!!

"Proof" of the "Theorem"

If \( \rho \) is completely arbitrary and nothing is known, we cannot possibly infer anything about \( \rho \) from samples \( ((x_i, y_i))_{i=1}^{m} \)…
"No Free Lunch" Theorem [Wolpert(1996)], Informal Version

Of course not!!

"Proof" of the "Theorem"

If $\rho$ is completely arbitrary and nothing is known, we cannot possibly infer anything about $\rho$ from samples $((x_i, y_i))_{i=1}^m$...

Every algorithm will have a specific preference, for example as specified through the hypothesis class $\mathcal{H}$ – all "categories" are artificial!
“No Free Lunch” Theorem [Wolpert(1996)], Informal Version

Of course not!!

“Proof” of the “Theorem”

If $\rho$ is completely arbitrary and nothing is known, we cannot possibly infer anything about $\rho$ from samples $((x_i, y_i))_{i=1}^{m}$. . .

Every algorithm will have a specific preference, for example as specified through the hypothesis class $\mathcal{H}$ – all “categories” are artificial!

💡 We want our algorithm to reproduce the artificial categories produced by our brain – so let’s build a hypothesis class that mimicks our thinking!
In neuroscience, a biological neural network is a series of interconnected neurons whose activation defines a recognizable linear pathway. The interface through which neurons interact with their neighbors usually consists of several axon terminals connected via synapses to dendrites on other neurons. If the sum of the input signals into one neuron surpasses a certain threshold, the neuron sends an action potential (AP) at the axon hillock and transmits this electrical signal along the axon.
The Brain as Biological Neural Network

“In neuroscience, a biological neural network is a series of interconnected neurons whose activation defines a recognizable linear pathway. The interface through which neurons interact with their neighbors usually consists of several axon terminals connected via synapses to dendrites on other neurons. If the sum of the input signals into one neuron surpasses a certain threshold, the neuron sends an action potential (AP) at the axon hillock and transmits this electrical signal along the axon.”

Neurons

"If the sum of the input signals into one neuron surpasses a certain threshold, the neuron transmits this signal."
recall: “If the sum of the input signals into one neuron surpasses a certain threshold, [...] the neuron transmits this [...] signal [...].”
Artificial Neurons

An artificial neuron with weights $w_1, \ldots, w_s$, bias $b$ and activation function $\sigma: \mathbb{R} \rightarrow \mathbb{R}$ is defined as the function $f(x_1, \ldots, x_s) = \sigma(\sum_{i=1}^s x_i w_i - b)$.

![Diagram of an artificial neuron with inputs, weights, threshold, sum, activation function, and output.](image)
Artificial Neurons

An artificial neuron with weights $w_1, \ldots, w_s$, bias $b$ and activation function $\sigma : \mathbb{R} \rightarrow \mathbb{R}$ is defined as the function

$$f(x_1, \ldots, x_s) = \sigma \left( \sum_{i=1}^{s} x_i w_i - b \right).$$
Activation Functions

Figure: Heaviside activation function (as in biological motivation)

\[ \sigma(x) = \frac{1}{1 + e^{-x}} \]
Activation Functions

**Figure:** Heaviside activation function (as in biological motivation)

\[
\sigma(x) = \frac{1}{1+e^{-x}}
\]

**Figure:** Sigmoid activation function \( \sigma(x) = \frac{1}{1+e^{-x}} \)
Artificial Neural Networks

Artificial neural networks consist of a graph, connecting artificial neurons. Dynamics are difficult to model, due to loops, etc...
Artificial neural networks consist of a graph, connecting artificial neurons!
Artificial neural networks consist of a graph, connecting artificial neurons!

- Dynamics difficult to model, due to loops, etc...
Artificial Feedforward Neural Networks

Use directed, acyclic graph!
Artificial Feedforward Neural Networks

Use directed, acyclic graph!
Artificial Feedforward Neural Networks

**Definition**

Let $L, d, N_1, \ldots, N_L \in \mathbb{N}$. A map $\Phi : \mathbb{R}^d \to \mathbb{R}^{N_L}$ given by

$$\Phi(x) = A_L \sigma (A_{L-1} \sigma (\ldots \sigma (A_1(x))))$$

is called a *neural network*. It is composed of affine linear maps $A_\ell : \mathbb{R}^{N_{\ell-1}} \to \mathbb{R}^{N_\ell}$, $1 \leq \ell \leq L$ (where $N_0 = d$), and non-linear functions—often referred to as *activation function*—$\sigma$ acting component-wise. Here, $d$ is the *dimension of the input layer*, $L$ denotes the *number of layers*, $N_1, \ldots, N_{L-1}$ stands for the *dimensions of the $L-1$ hidden layers*, and $N_L$ is the *dimension of the output layer*. 

An affine map $A_\ell : \mathbb{R}^{N_{\ell-1}} \to \mathbb{R}^{N_\ell}$ is given by $x \mapsto Wx + b$ with weight matrix $W \in \mathbb{R}^{N_{\ell-1} \times N_\ell}$ and bias vector $b \in \mathbb{R}^{N_\ell}$. 
Artificial Feedforward Neural Networks

**Definition**

Let $L, d, N_1, \ldots, N_L \in \mathbb{N}$. A map $\Phi : \mathbb{R}^d \to \mathbb{R}^{N_L}$ given by

$$\Phi(x) = A_L \sigma (A_{L-1} \sigma (\ldots \sigma (A_1(x))))), \quad x \in \mathbb{R}^d,$$

is called a *neural network*. It is composed of affine linear maps $A_\ell : \mathbb{R}^{N_{\ell-1}} \to \mathbb{R}^{N_\ell}, 1 \leq \ell \leq L$ (where $N_0 = d$), and non-linear functions—often referred to as *activation function*—$\sigma$ acting component-wise. Here, $d$ is the *dimension of the input layer*, $L$ denotes the *number of layers*, $N_1, \ldots, N_{L-1}$ stands for the *dimensions of the $L - 1$ hidden layers*, and $N_L$ is the *dimension of the output layer*.

An affine map $A : \mathbb{R}^{N_{\ell-1}} \to \mathbb{R}^{N_\ell}$ is given by $x \mapsto Wx + b$ with *weight matrix* $W \in \mathbb{R}^{N_{\ell-1} \times N_\ell}$ and *bias vector* $b \in \mathbb{R}^{N_\ell}$.
Artificial Feedforward Neural Networks

Output layer

Hidden layer

Input layer

\[ W_2 = \begin{pmatrix} (W_2)_{1,1} & (W_2)_{1,2} & 0 \\ 0 & 0 & (W_2)_{2,3} \end{pmatrix} \]

\[ W_1 = \begin{pmatrix} (W_1)_{1,1} & (W_1)_{1,2} & 0 \\ 0 & 0 & (W_1)_{2,3} \end{pmatrix} \]
Artificial (feedforward) neural networks should not be confused with a model for our brain: neurons are more complicated than simply weighted linear combinations. Our brain is not “feedforward.” Biological neural networks evolve with time, implying neuronal plasticity. Artificial feedforward neural networks constitute a mathematically and computationally convenient but very simplistic mathematical construct which is inspired by our understanding of how the brain works.
Artificial (feedforward) neural networks should not be confused with a model for our brain:
Artificial (feedforward) neural networks should not be confused with a model for our brain:

- Neurons are more complicated than simply weighted linear combinations.
On the Biological Motivation

Artificial (feedforward) neural networks should not be confused with a model for our brain:

- Neurons are more complicated than simply weighted linear combinations
- Our brain is not “feedforward”
Artificial (feedforward) neural networks should not be confused with a model for our brain:

- Neurons are more complicated than simply weighted linear combinations
- Our brain is not “feedforward”
- Biological neural networks evolve with time $\implies$ neuronal plasticity
Artificial (feedforward) neural networks should not be confused with a model for our brain:

- Neurons are more complicated than simply weighted linear combinations
- Our brain is not “feedforward”
- Biological neural networks evolve with time \(\leadsto\) neuronal plasticity
- ...
Artificial (feedforward) neural networks should not be confused with a model for our brain:

- Neurons are more complicated than simply weighted linear combinations
- Our brain is not “feedforward”
- Biological neural networks evolve with time $\rightsquigarrow$ neuronal plasticity
- ...

Artificial feedforward neural networks constitute a mathematically and computationally convenient but very simplistic mathematical construct which is inspired by our understanding of how the brain works.
Terminology

"Neural Network Learning":
Use neural networks of a fixed "topology" as hypothesis class for regression or classification tasks. This requires optimizing the weights and bias parameters.

"Deep Learning":
Neural network learning with neural networks consisting of many (e.g., ≥ 3) layers.
“Neural Network Learning”: Use neural networks of a fixed "topology" as hypothesis class for regression or classification tasks.
Terminology

- **“Neural Network Learning”:** Use neural networks of a fixed “topology” as hypothesis class for regression or classification tasks. This requires optimizing the weights and bias parameters.
Terminology

- **“Neural Network Learning”**: Use neural networks of a fixed “topology” as hypothesis class for regression or classification tasks. This requires optimizing the weights and bias parameters.

- **“Deep Learning”**: Neural network learning with neural networks consisting of many (e.g., $\geq 3$) layers.
2 Universal Approximation
Approximation Question

Main Approximation Problem

(continuous, or measurable) function $f : \mathbb{R}^d \to \mathbb{R}^{N_L}$ be arbitrarily well approximated by a neural network, provided that we choose $N_1, \ldots, N_{L-1}, L$ large enough?
Approximation Question

Main Approximation Problem

(continuous, or measurable) function $f : \mathbb{R}^d \rightarrow \mathbb{R}^{N_L}$ be arbitrarily well approximated by a neural network, provided that we choose $N_1, \ldots, N_{L-1}, L$ large enough?

Surely not!
Approximation Question

Main Approximation Problem

(continuous, or measurable) function $f : \mathbb{R}^d \rightarrow \mathbb{R}^{N_L}$ be arbitrarily well approximated by a neural network, provided that we choose $N_1, \ldots, N_{L-1}, L$ large enough?

Surely not! Suppose that $\sigma$ is a polynomial of degree $r$. Then $\sigma(Ax)$ is a polynomial of degree $\leq r$ for all affine maps $A$ and therefore any neural network with activation function $\sigma$ will be a polynomial of degree $\leq r$. 
Approximation Question

Main Approximation Problem

Under which conditions on the activation function $\sigma$ can every (continuous, or measurable) function $f : \mathbb{R}^d \to \mathbb{R}^{N_L}$ be arbitrarily well approximated by a neural network, provided that we choose $N_1, \ldots, N_{L-1}, L$ large enough?
Theorem

Suppose that $\sigma : \mathbb{R} \to \mathbb{R}$ continuous is not a polynomial and fix $d \geq 1$, $L \geq 2$, $N_L \geq 1 \in \mathbb{N}$ and a compact subset $K \subset \mathbb{R}^d$. Then for any continuous $f : \mathbb{R}^d \to \mathbb{R}^{N_L}$ and any $\varepsilon > 0$ there exist $N_1, \ldots, N_{L-1} \in \mathbb{N}$ and affine linear maps $A_\ell : \mathbb{R}^{N_{\ell-1}} \to \mathbb{R}^{N_{\ell}}$, $1 \leq \ell \leq L$ such that the neural network

$$
\Phi(x) = A_L \sigma (A_{L-1} \sigma (\ldots \sigma (A_1(x))))), \quad x \in \mathbb{R}^d,
$$

approximates $f$ to within accuracy $\varepsilon$, i.e.,

$$
\sup_{x \in K} |\Phi(x) - f(x)| \leq \varepsilon.
$$
Universal Approximation Theorem

Theorem

Suppose that $\sigma : \mathbb{R} \to \mathbb{R}$ continuous is not a polynomial and fix $d \geq 1, L \geq 2, N_L \geq 1 \in \mathbb{N}$ and a compact subset $K \subset \mathbb{R}^d$. Then for any continuous $f : \mathbb{R}^d \to \mathbb{R}^N$ and any $\varepsilon > 0$ there exist $N_1, \ldots, N_{L-1} \in \mathbb{N}$ and affine linear maps $A_\ell : \mathbb{R}^{N_{\ell-1}} \to \mathbb{R}^{N_\ell}$, $1 \leq \ell \leq L$ such that the neural network

$$
\Phi(x) = A_L \sigma (A_{L-1} \sigma (\ldots \sigma (A_1(x))))), \quad x \in \mathbb{R}^d,
$$

approximates $f$ to within accuracy $\varepsilon$, i.e.,

$$
\sup_{x \in K} |\Phi(x) - \Phi(x)| \leq \varepsilon.
$$

Neural networks are “universal approximators” and one hidden layer is enough if the number of nodes is sufficient!
Proof of the Universal Approximation Theorem

For simplicity we only the case of one hidden layer, e.g., $L = 2$ and one output neuron, e.g., $N_L = 1$:

$$\phi(x) = \sum_{i=1}^{c} c_i \sigma(w_i \cdot x - b_i), \quad w_i \in \mathbb{R}^d, \quad c_i, b_i \in \mathbb{R}.$$
Proof of the Universal Approximation Theorem

For simplicity we only the case of one hidden layer, e.g., $L = 2$ and one output neuron, e.g., $N_L = 1$:

$$
\Phi(x) = \sum_{i=1}^{N_1} c_i \sigma(w_i \cdot x - b_i), \quad w_i \in \mathbb{R}^d, \ c_i, b_i \in \mathbb{R}.
$$
Proof of the Universal Approximation Theorem

We will show the following.

**Theorem**

For $d \in \mathbb{N}$ and $\sigma : \mathbb{R} \to \mathbb{R}$ continuous consider

$$R(\sigma, d) := \text{span} \left\{ \sigma(w \cdot x - b) : w \in \mathbb{R}^d, \ b \in \mathbb{R} \right\}.$$

Then $R(\sigma, d)$ is dense in $C(\mathbb{R}^d)$ if and only if $\sigma$ is not a polynomial.
Proof for $d = 1$ and $\sigma$ smooth

if $\sigma$ is not a polynomial, there exists $x_0 \in \mathbb{R}$ with $\sigma(\lambda x - x_0) \neq 0$ for all $\lambda \in \mathbb{N}$. 

constant functions can be approximated because $\sigma(hx - x_0) \to \sigma(-x_0)$, $h \to 0$.

linear functions can be approximated because $\frac{1}{h}(\sigma((\lambda + h)x - x_0) - \sigma(\lambda x - x_0)) \to x\sigma'(x_0)$, $h, \lambda \to 0$.

the same argument $\Rightarrow$ polynomials in $x$ can be approximated.

Stone-Weierstrass Theorem yields the result.
Proof for $d = 1$ and $\sigma$ smooth

- if $\sigma$ is not a polynomial, there exists $x_0 \in \mathbb{R}$ with $\sigma^{(k)}(-x_0) \neq 0$ for all $k \in \mathbb{N}$. 

linear functions can be approximated because

$$
1_h \left( \sigma\left(\lambda+h(x-x_0)\right) - \sigma(\lambda x-x_0) \right) \to x\sigma'(x_0), \quad h,\lambda \to 0.
$$

The same argument $\Rightarrow$ polynomials in $x$ can be approximated.

Stone-Weierstrass Theorem yields the result.
Proof for $d = 1$ and $\sigma$ smooth

- if $\sigma$ is not a polynomial, there exists $x_0 \in \mathbb{R}$ with $\sigma^{(k)}(-x_0) \neq 0$ for all $k \in \mathbb{N}$.
- constant functions can be approximated because

$$\sigma(hx - x_0) \to \sigma(-x_0) \neq 0, \quad h \to 0.$$
Proof for $d = 1$ and $\sigma$ smooth

- if $\sigma$ is not a polynomial, there exists $x_0 \in \mathbb{R}$ with $\sigma^{(k)}(-x_0) \neq 0$ for all $k \in \mathbb{N}$.
- constant functions can be approximated because
  \[ \sigma(hx - x_0) \rightarrow \sigma(-x_0) \neq 0, \quad h \rightarrow 0. \]
- linear functions can be approximated because
  \[ \frac{1}{h}(\sigma((\lambda + h)x - x_0) - \sigma(\lambda x - x_0)) \rightarrow x\sigma'(-x_0), \quad h, \lambda \rightarrow 0. \]
Proof for $d = 1$ and $\sigma$ smooth

- if $\sigma$ is not a polynomial, there exists $x_0 \in \mathbb{R}$ with $\sigma^{(k)}(-x_0) \neq 0$ for all $k \in \mathbb{N}$.

- constant functions can be approximated because

$$\sigma(hx - x_0) \to \sigma(-x_0) \neq 0, \quad h \to 0.$$ 

- linear functions can be approximated because

$$\frac{1}{h} (\sigma((\lambda + h)x - x_0) - \sigma(\lambda x - x_0)) \to x\sigma'(-x_0), \quad h, \lambda \to 0.$$ 

- same argument $\Rightarrow$ polynomials in $x$ can be approximated.
Proof for $d = 1$ and $\sigma$ smooth

- If $\sigma$ is not a polynomial, there exists $x_0 \in \mathbb{R}$ with $\sigma^{(k)}(-x_0) \neq 0$ for all $k \in \mathbb{N}$.

- Constant functions can be approximated because

$$\sigma(hx - x_0) \to \sigma(-x_0) \neq 0, \quad h \to 0.$$

- Linear functions can be approximated because

$$\frac{1}{h} (\sigma((\lambda + h)x - x_0) - \sigma(\lambda x - x_0)) \to x\sigma'(-x_0), \quad h, \lambda \to 0.$$

- Same argument $\rightsquigarrow$ polynomials in $x$ can be approximated.

- Stone-Weierstrass Theorem yields the result.
Note that the functions \( \{ g(\mathbf{w} \cdot \mathbf{x} - \mathbf{b}) : \mathbf{w} \in \mathbb{R}^d, \mathbf{b} \in \mathbb{R}, g \in C(\mathbb{R}) \} \) are dense in \( C(\mathbb{R}^d) \) (just take \( g \) as \( \sin(\mathbf{w} \cdot \mathbf{x}) \), \( \cos(\mathbf{w} \cdot \mathbf{x}) \) just as in the Fourier series case).

First approximate \( f \in C(\mathbb{R}^d) \) by \( \sum_{i=1}^{N} d_i g_i(\mathbf{v}_i \cdot \mathbf{x} - e_i) \), \( \mathbf{v}_i \in \mathbb{R}^d \), \( d_i, e_i \in \mathbb{R} \), \( g_i \in C(\mathbb{R}) \).

Then apply our univariate result to approximate the univariate functions \( t \mapsto g_i(t - e_i) \) using neural networks.
Note that the functions

$$\text{span}\{g(w \cdot x - b) : w \in \mathbb{R}^d, b \in \mathbb{R}, g \in C(\mathbb{R}) \text{ arbitrary}\},$$

are dense in $C(\mathbb{R}^d)$ (just take $g$ as $\sin(w \cdot x), \cos(w \cdot x)$ just as in the Fourier series case).
Note that the functions
\[ \text{span}\{g(w \cdot x - b) : w \in \mathbb{R}^d, \ b \in \mathbb{R}, \ g \in C(\mathbb{R}) \ \text{arbitrary}\}, \]
are dense in \( C(\mathbb{R}^d) \) (just take \( g \) as \( \sin(w \cdot x), \cos(w \cdot x) \) just as in the Fourier series case).

First approximate \( f \in C(\mathbb{R}^d) \) by
\[
\sum_{i=1}^{N} d_i g_i(v_i \cdot x - e_i), \quad v_i \in \mathbb{R}^d, \ d_i, e_i \in \mathbb{R}, \ g_i \in C(\mathbb{R}).
\]
Note that the functions

$$\text{span}\{g(w \cdot x - b) : w \in \mathbb{R}^d, b \in \mathbb{R}, g \in C(\mathbb{R}) \text{ arbitrary}\},$$

are dense in $C(\mathbb{R}^d)$ (just take $g$ as $\sin(w \cdot x), \cos(w \cdot x)$ just as in the Fourier series case).

First approximate $f \in C(\mathbb{R}^d)$ by

$$\sum_{i=1}^{N} d_i g_i(v_i \cdot x - e_i), \quad v_i \in \mathbb{R}^d, \; d_i, e_i \in \mathbb{R}, \; g_i \in C(\mathbb{R}).$$

Then apply our univariate result to approximate the univariate functions $t \mapsto g_i(t - e_i)$ using neural networks.
The case that $\sigma$ is nonsmooth
The case that $\sigma$ is nonsmooth

pick family $\left( g_\varepsilon \right)_{\varepsilon > 0}$ of mollifiers, i.e.

$$\lim_{\varepsilon \to 0} \sigma * g_\varepsilon \to \sigma$$

uniformly on compacta.
The case that $\sigma$ is nonsmooth

pick family $(g_\epsilon)_{\epsilon>0}$ of mollifiers, i.e.

$$\lim_{\epsilon \to 0} \sigma \ast g_\epsilon \to \sigma$$

uniformly on compacta.

Apply previous result to the smooth function $\sigma \ast g_\epsilon$ and let $\epsilon \to 0$: 
The case that $\sigma$ is nonsmooth

pick family $(g_\varepsilon)_{\varepsilon > 0}$ of mollifiers, i.e.

$$\lim_{\varepsilon \to 0} \sigma \ast g_\varepsilon \to \sigma$$

uniformly on compacta.

Apply previous result to the smooth function $\sigma \ast g_\varepsilon$ and let $\varepsilon \to 0$: 
3 Backpropagation
Regression/Classification with Neural Networks

Neural Network Hypothesis Class

Given \( d, L, N_1, \ldots, N_L \) and \( \sigma \) define the associated hypothesis class

\[
\mathcal{H}_{[d, N_1, \ldots, N_L], \sigma} := \{ A_L \sigma (A_{L-1} \sigma (\ldots \sigma (A_1(x)))) : A_\ell : \mathbb{R}^{N_{\ell-1}} \to \mathbb{R}^{N_\ell} \text{ affine linear} \}.
\]
Regression/Classification with Neural Networks

Neural Network Hypothesis Class

Given $d, L, N_1, \ldots, N_L$ and $\sigma$ define the associated hypothesis class

$$\mathcal{H}_{[d,N_1,\ldots,N_L],\sigma} := \{ A_L \sigma (A_{L-1} \sigma (\ldots \sigma (A_1(x)))) : A_\ell : \mathbb{R}^{N_{\ell-1}} \to \mathbb{R}^{N_\ell} \text{ affine linear} \}.$$

Typical Regression/Classification Task

Given data $z = ((x_i, y_i))_{i=1}^m \subset \mathbb{R}^d \times \mathbb{R}^{N_L}$, find the empirical regression function

$$f_z \in \operatorname{argmin}_{f \in \mathcal{H}_{[d,N_1,\ldots,N_L],\sigma}} \sum_{i=1}^m \mathcal{L}(f, x_i, y_i),$$

where $\mathcal{L} : C(\mathbb{R}^d) \times \mathbb{R}^d \times \mathbb{R}^{N_L} \to \mathbb{R}_+$ is the loss function (in least squares problems we have $\mathcal{L}(f, x, y) = |f(x) - y|^2$).
Example: Handwritten Digits

MNIST Database for handwritten digit recognition
http://yann.lecun.com/exdb/mnist/
Example: Handwritten Digits

- Every image is given as a $28 \times 28$ matrix
  
  \[
  x \in \mathbb{R}^{28 \times 28} \sim \mathbb{R}^{784}.
  \]

MNIST Database for handwritten digit recognition

http://yann.lecun.com/exdb/mnist/
Example: Handwritten Digits

- Every image is given as a $28 \times 28$ matrix
  
  $x \in \mathbb{R}^{28 \times 28} \sim \mathbb{R}^{784}$.

MNIST Database for handwritten digit recognition

http://yann.lecun.com/exdb/mnist/
Example: Handwritten Digits

- Every image is given as a $28 \times 28$ matrix
  
  $$x \in \mathbb{R}^{28 \times 28} \sim \mathbb{R}^{784}.$$ 

- Every label is given as a 10-dim vector $y \in \mathbb{R}^{10}$
  describing the ‘probability’ of each digit

MNIST Database for handwritten digit recognition
http://yann.lecun.com/exdb/mnist/
Example: Handwritten Digits

- Every image is given as a $28 \times 28$ matrix $x \in \mathbb{R}^{28 \times 28} \sim \mathbb{R}^{784}$.

- Every label is given as a 10-dim vector $y \in \mathbb{R}^{10}$ describing the ‘probability’ of each digit.

MNIST Database for handwritten digit recognition
http://yann.lecun.com/exdb/mnist/
Example: Handwritten Digits

■ Every image is given as a $28 \times 28$ matrix
  \[ x \in \mathbb{R}^{28\times28} \sim \mathbb{R}^{784}. \]

■ Every label is given as a 10-dim vector $y \in \mathbb{R}^{10}$
  describing the ‘probability’ of each digit.
Example: Handwritten Digits

- Every image is given as a $28 \times 28$ matrix
  
  $x \in \mathbb{R}^{28 \times 28} \sim \mathbb{R}^{784}$.

- Every label is given as a 10-dim vector $y \in \mathbb{R}^{10}$ describing the ‘probability’ of each digit

- Given labeled training data
  
  $(x_i, y_i)_{i=1}^m \subset \mathbb{R}^{784} \times \mathbb{R}^{10}$. 

- Non-linear, non-convex
Example: Handwritten Digits

- Every image is given as a $28 \times 28$ matrix $x \in \mathbb{R}^{28 \times 28} \sim \mathbb{R}^{784}$.

- Every label is given as a 10-dim vector $y \in \mathbb{R}^{10}$ describing the ‘probability’ of each digit.

- Given labeled *training data* $(x_i, y_i)_{i=1}^{m} \subset \mathbb{R}^{784} \times \mathbb{R}^{10}$.

- Fix network topology, e.g., number of layers (for example $L = 3$) and numbers of neurons ($N_1 = 20$, $N_2 = 20$).
Example: Handwritten Digits

- Every image is given as a $28 \times 28$ matrix $x \in \mathbb{R}^{28 \times 28} \sim \mathbb{R}^{784}$.

- Every label is given as a 10-dim vector $y \in \mathbb{R}^{10}$ describing the ‘probability’ of each digit.

- Given labeled training data $(x_i, y_i)_{i=1}^{m} \subset \mathbb{R}^{784} \times \mathbb{R}^{10}$.

- Fix network topology, e.g., number of layers (for example $L = 3$) and numbers of neurons ($N_1 = 20$, $N_2 = 20$).

- The learning goal is to find the empirical regression function $f_{\mathbf{z}} \in \mathcal{H}[784,20,20,10],\sigma$. 
Example: Handwritten Digits

- Every image is given as a $28 \times 28$ matrix $x \in \mathbb{R}^{28 \times 28} \sim \mathbb{R}^{784}$.

- Every label is given as a 10-dim vector $y \in \mathbb{R}^{10}$ describing the ‘probability’ of each digit.

- Given labeled *training data* $(x_i, y_i)_{i=1}^m \subset \mathbb{R}^{784} \times \mathbb{R}^{10}$.

- Fix network topology, e.g., number of layers (for example $L = 3$) and numbers of neurons ($N_1 = 20$, $N_2 = 20$).

- The learning goal is to find the empirical regression function $f_{\mathbf{z}} \in \mathcal{H}[784, 20, 20, 10], \sigma$.

???how???
Example: Handwritten Digits

- Every image is given as a $28 \times 28$ matrix $x \in \mathbb{R}^{28 \times 28} \sim \mathbb{R}^{784}$:

- Every label is given as a 10-dim vector $y \in \mathbb{R}^{10}$ describing the ‘probability’ of each digit

Given labeled training data $(x_i, y_i)_{i=1}^m \subset \mathbb{R}^{784} \times \mathbb{R}^{10}$.

- Fix network topology, e.g., number of layers (for example $L = 3$) and numbers of neurons ($N_1 = 20$, $N_2 = 20$).

- The learning goal is to find the empirical regression function $f_z \in \mathcal{H}[784, 20, 20, 10], \sigma$.

???how???

Non-linear, non-convex
Gradient Descent: The Simplest Optimization Method

Gradient Descent

Gradient of $F: \mathbb{R}^N \rightarrow \mathbb{R}$ is defined by $\nabla F(u) = \left( \frac{\partial F(u)}{\partial u_1}, \ldots, \frac{\partial F(u)}{\partial u_N} \right)^T$.

Gradient descent with stepsize $\eta > 0$ is defined by $u_{n+1} \leftarrow u_n - \eta \nabla F(u_n)$.

Converges (slowly) to stationary point of $F$. 
Gradient Descent: The Simplest Optimization Method

Gradient Descent

- Gradient of $F : \mathbb{R}^N \rightarrow \mathbb{R}$ is defined by

$$\nabla F(u) = \left( \frac{\partial F(u)}{\partial (u)_1}, \cdots, \frac{\partial F(u)}{\partial (u)_N} \right)^T.$$
Gradient Descent: The Simplest Optimization Method

**Gradient Descent**

- Gradient of $F : \mathbb{R}^N \to \mathbb{R}$ is defined by

$$\nabla F(u) = \left( \frac{\partial F(u)}{\partial (u)_1}, \ldots, \frac{\partial F(u)}{\partial (u)_N} \right)^T.$$

- Gradient descent with *stepsize* $\eta > 0$ is defined by

$$u_{n+1} \leftarrow u_n - \eta \nabla F(u_n).$$

Converges (slowly) to stationary point of $F$. 
Gradient Descent: The Simplest Optimization Method

Gradient Descent

- Gradient of $F : \mathbb{R}^N \rightarrow \mathbb{R}$ is defined by

$$\nabla F(u) = \left( \frac{\partial F(u)}{\partial (u)_1}, \ldots, \frac{\partial F(u)}{\partial (u)_N} \right)^T.$$

- Gradient descent with stepsize $\eta > 0$ is defined by

$$u_{n+1} \leftarrow u_n - \eta \nabla F(u_n).$$

- Converges (slowly) to stationary point of $F$. 
In our problem: $F = \sum_{i=1}^{m} \mathcal{L}(f, x_i, y_i)$ and $u = ((W_\ell, b_\ell))_{\ell=1}^{L}$. 
In our problem: \( F = \sum_{i=1}^{m} \mathcal{L}(f, x_i, y_i) \) and \( u = (W_{\ell}, b_{\ell})_{\ell=1}^{L} \).

Since \( \nabla_{((W_{\ell}, b_{\ell}))_{\ell=1}^{L}} F = \sum_{i=1}^{m} \nabla_{((W_{\ell}, b_{\ell}))_{\ell=1}^{L}} \mathcal{L}(f, x_i, y_i) \), we need to determine (for \( x, y \in \mathbb{R}^d \times \mathbb{R}^{N_L} \) fixed)

\[
\frac{\partial \mathcal{L}(f, x, y)}{\partial (W_{\ell})_{i,j}}, \quad \frac{\partial \mathcal{L}(f, x, y)}{\partial (b_{\ell})_{i}}, \quad \ell = 1, \ldots, L.
\]
In our problem: $F = \sum_{i=1}^{m} \mathcal{L}(f, x_i, y_i)$ and $u = ((W_\ell, b_\ell))_{\ell=1}^{L}$.

Since $\nabla_{((W_\ell,b_\ell))_{\ell=1}^{L}} F = \sum_{i=1}^{m} \nabla_{((W_\ell,b_\ell))_{\ell=1}^{L}} \mathcal{L}(f, x_i, y_i)$, we need to determine (for $x, y \in \mathbb{R}^d \times \mathbb{R}^{N_L}$ fixed)

$$\frac{\partial \mathcal{L}(f, x, y)}{\partial (W_\ell)_{i,j}}, \quad \frac{\partial \mathcal{L}(f, x, y)}{\partial (b_\ell)_i}, \quad \ell = 1, \ldots, L.$$ 

For simplicity suppose that $\mathcal{L}(f, x, y) = (f(x) - y)^2$, so that

$$\frac{\partial \mathcal{L}(f, x, y)}{\partial (W_\ell)_{i,j}} = 2 \cdot (f(x) - y)^T \cdot \frac{\partial f(x)}{\partial (W_\ell)_{i,j}},$$

$$\frac{\partial \mathcal{L}(f, x, y)}{\partial (b_\ell)_i} = 2 \cdot (f(x) - y)^T \cdot \frac{\partial f(x)}{\partial (b_\ell)_i}.$$
\[ x = \begin{pmatrix} (x)_1 \\ (x)_2 \end{pmatrix} \]
\[ x = \begin{pmatrix} (x)_1 \\ (x)_2 \end{pmatrix} \quad a_1 = \sigma(z_1) \]

\[ = \sigma(W_1x + b_1) \]

\[ W_1 = \begin{pmatrix} (W_1)_{1,1} & (W_1)_{1,2} \\ (W_1)_{2,1} & (W_1)_{2,2} \\ (W_1)_{3,1} & (W_1)_{3,2} \end{pmatrix} \]

\[ b_1 = \begin{pmatrix} (b_1)_1 \\ (b_1)_2 \\ (b_1)_3 \end{pmatrix} \]
\[ x = \begin{pmatrix} (x)_1 \\ (x)_2 \end{pmatrix} \]

\[ a_1 = \sigma(\tilde{z}_1) = \sigma(W_1x + b_1) \]

\[ W_1 = \begin{pmatrix} (W_1)_{1,1} & (W_1)_{1,2} \\ (W_1)_{2,1} & (W_1)_{2,2} \\ (W_1)_{3,1} & (W_1)_{3,2} \end{pmatrix} \]

\[ b_1 = \begin{pmatrix} (b_1)_1 \\ (b_1)_2 \\ (b_1)_3 \end{pmatrix} \]

\[ a_2 = \sigma(\tilde{z}_2) = \sigma(W_2a_1 + b_2) \]

\[ W_2 = \begin{pmatrix} (W_2)_{1,1} & (W_2)_{1,2} & (W_2)_{1,3} \\ (W_2)_{2,1} & (W_2)_{2,2} & (W_2)_{2,3} \\ (W_2)_{3,1} & (W_2)_{3,2} & (W_2)_{3,3} \end{pmatrix} \]

\[ b_2 = \begin{pmatrix} (b_2)_1 \\ (b_2)_2 \\ (b_2)_3 \end{pmatrix} \]
\[ W_3 = \begin{pmatrix} (W_3)_{1,1} & (W_3)_{1,2} & (W_3)_{1,3} \\ (W_3)_{2,1} & (W_3)_{2,2} & (W_3)_{2,3} \end{pmatrix} \]

\[ b_3 = \begin{pmatrix} (b_3)_1 \\ (b_3)_2 \end{pmatrix} \]

\[ x = \begin{pmatrix} (x)_1 \\ (x)_2 \end{pmatrix} \]

\[ a_1 = \sigma(z_1) = \sigma(W_1x + b_1) \]

\[ W_1 = \begin{pmatrix} (W_1)_{1,1} & (W_1)_{1,2} \\ (W_1)_{2,1} & (W_1)_{2,2} \\ (W_1)_{3,1} & (W_1)_{3,2} \end{pmatrix} \]

\[ b_1 = \begin{pmatrix} (b_1)_1 \\ (b_1)_2 \\ (b_1)_3 \end{pmatrix} \]

\[ a_2 = \sigma(z_2) = \sigma(W_2a_1 + b_2) \]

\[ W_2 = \begin{pmatrix} (W_2)_{1,1} & (W_2)_{1,2} & (W_2)_{1,3} \\ (W_2)_{2,1} & (W_2)_{2,2} & (W_2)_{2,3} \\ (W_2)_{3,1} & (W_2)_{3,2} & (W_2)_{3,3} \end{pmatrix} \]

\[ b_2 = \begin{pmatrix} (b_2)_1 \\ (b_2)_2 \\ (b_2)_3 \end{pmatrix} \]

\[ \Phi(x) = z_3 = W_3a_2 + b_3 \]
\[
\frac{\partial (z_3)_1}{\partial (W_3)_{1,2}} =
\]

\[
W_3 = \begin{pmatrix}
(W_3)_{1,1} & (W_3)_{1,2} & (W_3)_{1,3} \\
(W_3)_{2,1} & (W_3)_{2,2} & (W_3)_{2,3}
\end{pmatrix}
\]

\[
b_3 = \begin{pmatrix}
(b_3)_{1} \\
(b_3)_{2}
\end{pmatrix}
\]

\[
x = \begin{pmatrix}
(x)_{1} \\
(x)_{2}
\end{pmatrix}
\]

\[
a_1 = \sigma(z_1)
\]

\[
= \sigma(W_1x + b_1)
\]

\[
W_1 = \begin{pmatrix}
(W_1)_{1,1} & (W_1)_{1,2} \\
(W_1)_{2,1} & (W_1)_{2,2} \\
(W_1)_{3,1} & (W_1)_{3,2}
\end{pmatrix}
\]

\[
b_1 = \begin{pmatrix}
(b_1)_{1} \\
(b_1)_{2} \\
(b_1)_{3}
\end{pmatrix}
\]

\[
a_2 = \sigma(z_2)
\]

\[
= \sigma(W_2a_1 + b_2)
\]

\[
W_2 = \begin{pmatrix}
(W_2)_{1,1} & (W_2)_{1,2} & (W_2)_{1,3} \\
(W_2)_{2,1} & (W_2)_{2,2} & (W_2)_{2,3} \\
(W_2)_{3,1} & (W_2)_{3,2} & (W_2)_{3,3}
\end{pmatrix}
\]

\[
b_2 = \begin{pmatrix}
(b_2)_{1} \\
(b_2)_{2} \\
(b_2)_{3}
\end{pmatrix}
\]

\[
\Phi(x) = z_3
\]

\[
= W_3a_2 + b_3
\]
\[
\begin{align*}
\frac{\partial (z_3)_1}{\partial (W_3)_{1,2}} &= \frac{\partial}{\partial (W_3)_{1,2}} ((W_3)_{1,1}(a_2)_1 + (W_3)_{1,2}(a_2)_2 + (W_3)_{1,3}(a_2)_3) \\
W_3 &= \begin{pmatrix}
(W_3)_{1,1} & (W_3)_{1,2} & (W_3)_{1,3} \\
(W_3)_{2,1} & (W_3)_{2,2} & (W_3)_{2,3}
\end{pmatrix} \\
b_3 &= \begin{pmatrix}
(b_3)_1 \\
(b_3)_2
\end{pmatrix}
\end{align*}
\]

\[
\begin{align*}
x &= \begin{pmatrix}
(x)_1 \\
(x)_2
\end{pmatrix} \\
a_1 &= \sigma(z_1) = \sigma(W_1 x + b_1) \\
W_1 &= \begin{pmatrix}
(W_1)_{1,1} & (W_1)_{1,2} \\
(W_1)_{2,1} & (W_1)_{2,2} \\
(W_1)_{3,1} & (W_1)_{3,2}
\end{pmatrix} \\
b_1 &= \begin{pmatrix}
(b_1)_1 \\
(b_1)_2 \\
(b_1)_3
\end{pmatrix}
\end{align*}
\]

\[
\begin{align*}
a_2 &= \sigma(z_2) = \sigma(W_2 a_1 + b_2) \\
W_2 &= \begin{pmatrix}
(W_2)_{1,1} & (W_2)_{1,2} & (W_2)_{1,3} \\
(W_2)_{2,1} & (W_2)_{2,2} & (W_2)_{2,3} \\
(W_2)_{3,1} & (W_2)_{3,2} & (W_2)_{3,3}
\end{pmatrix} \\
b_2 &= \begin{pmatrix}
(b_2)_1 \\
(b_2)_2 \\
(b_2)_3
\end{pmatrix}
\end{align*}
\]

\[
\Phi(x) = z_3 = W_3 a_2 + b_3
\]
\[
\frac{\partial (z_3)_1}{\partial (W_3)_{1,2}} = \frac{\partial}{\partial (W_3)_{1,2}} \left( (W_3)_{1,1} (a_2)_1 + (W_3)_{1,2} (a_2)_2 + (W_3)_{1,3} (a_2)_3 \right) = (a_2)_2
\]

\[
W_3 = \begin{pmatrix}
(W_3)_{1,1} & (W_3)_{1,2} & (W_3)_{1,3} \\
(W_3)_{2,1} & (W_3)_{2,2} & (W_3)_{2,3}
\end{pmatrix}
\]

\[
b_3 = \begin{pmatrix}
(b_3)_1 \\
(b_3)_2
\end{pmatrix}
\]

\[
x = \begin{pmatrix}
(x)_1 \\
(x)_2
\end{pmatrix}
\]

\[
a_1 = \sigma(z_1)
\]

\[
= \sigma(W_1 x + b_1)
\]

\[
W_1 = \begin{pmatrix}
(W_1)_{1,1} & (W_1)_{1,2} \\
(W_1)_{2,1} & (W_1)_{2,2} \\
(W_1)_{3,1} & (W_1)_{3,2}
\end{pmatrix}
\]

\[
b_1 = \begin{pmatrix}
(b_1)_1 \\
(b_1)_2 \\
(b_1)_3
\end{pmatrix}
\]

\[
a_2 = \sigma(z_2)
\]

\[
= \sigma(W_2 a_1 + b_2)
\]

\[
W_2 = \begin{pmatrix}
(W_2)_{1,1} & (W_2)_{1,2} & (W_2)_{1,3} \\
(W_2)_{2,1} & (W_2)_{2,2} & (W_2)_{2,3} \\
(W_2)_{3,1} & (W_2)_{3,2} & (W_2)_{3,3}
\end{pmatrix}
\]

\[
b_2 = \begin{pmatrix}
(b_2)_1 \\
(b_2)_2 \\
(b_2)_3
\end{pmatrix}
\]

\[
\Phi(x) = z_3
\]

\[
= W_3 a_2 + b_3
\]
\[
\frac{\partial (z_3)^2}{\partial (W_3)_{1,2}} = W_3 = \begin{pmatrix}
(W_3)_{1,1} & (W_3)_{1,2} & (W_3)_{1,3} \\
(W_3)_{2,1} & (W_3)_{2,2} & (W_3)_{2,3}
\end{pmatrix}
\]

\[
b_3 = \begin{pmatrix}
(b_3)_1 \\
(b_3)_2
\end{pmatrix}
\]

\[
x = \begin{pmatrix}
(x)_1 \\
(x)_2
\end{pmatrix}
\]

\[
a_1 = \sigma(z_1) = \sigma(W_1x + b_1)
\]

\[
W_1 = \begin{pmatrix}
(W_1)_{1,1} & (W_1)_{1,2} \\
(W_1)_{2,1} & (W_1)_{2,2} \\
(W_1)_{3,1} & (W_1)_{3,2}
\end{pmatrix}
\]

\[
b_1 = \begin{pmatrix}
(b_1)_1 \\
(b_1)_2 \\
(b_1)_3
\end{pmatrix}
\]

\[
a_2 = \sigma(z_2) = \sigma(W_2a_1 + b_2)
\]

\[
W_2 = \begin{pmatrix}
(W_2)_{1,1} & (W_2)_{1,2} & (W_2)_{1,3} \\
(W_2)_{2,1} & (W_2)_{2,2} & (W_2)_{2,3} \\
(W_2)_{3,1} & (W_2)_{3,2} & (W_2)_{3,3}
\end{pmatrix}
\]

\[
b_2 = \begin{pmatrix}
(b_2)_1 \\
(b_2)_2 \\
(b_2)_3
\end{pmatrix}
\]

\[
\Phi(x) = z_3 = W_3a_2 + b_3
\]
\[ \frac{\partial (z_3)^2}{\partial (W_3)_{1,2}} = \frac{\partial}{\partial (W_3)_{1,2}} ((W_3)_{2,1}(a_2)_1 + (W_3)_{2,2}(a_2)_2 + (W_3)_{2,3}(a_2)_3) \]

\[
W_3 = \begin{pmatrix}
(W_3)_{1,1} & (W_3)_{1,2} & (W_3)_{1,3} \\
(W_3)_{2,1} & (W_3)_{2,2} & (W_3)_{2,3}
\end{pmatrix}, \\
b_3 = \begin{pmatrix}
(b_3)_1 \\
(b_3)_2
\end{pmatrix}
\]

\[ x = \begin{pmatrix}
(x)_1 \\
(x)_2
\end{pmatrix}, \quad a_1 = \sigma(z_1) = \sigma(W_1 x + b_1) \\
W_1 = \begin{pmatrix}
(W_1)_{1,1} & (W_1)_{1,2} \\
(W_1)_{2,1} & (W_1)_{2,2} \\
(W_1)_{3,1} & (W_1)_{3,2}
\end{pmatrix}, \quad b_1 = \begin{pmatrix}
(b_1)_1 \\
(b_1)_2 \\
(b_1)_3
\end{pmatrix}
\]

\[ a_2 = \sigma(z_2) = \sigma(W_2 a_1 + b_2) \\
W_2 = \begin{pmatrix}
(W_2)_{1,1} & (W_2)_{1,2} & (W_2)_{1,3} \\
(W_2)_{2,1} & (W_2)_{2,2} & (W_2)_{2,3} \\
(W_2)_{3,1} & (W_2)_{3,2} & (W_2)_{3,3}
\end{pmatrix}, \quad b_2 = \begin{pmatrix}
(b_2)_1 \\
(b_2)_2 \\
(b_2)_3
\end{pmatrix}
\]

\[ \Phi(x) = z_3 = W_3 a_2 + b_3 \]
\[
\begin{align*}
\frac{\partial(z_3)_2}{\partial(W_3)_{1,2}} &= \\
\frac{\partial}{\partial(W_3)_{1,2}} ((W_3)_{2,1}(a_2)_1 + (W_3)_{2,2}(a_2)_2 + (W_3)_{2,3}(a_2)_3) \\
&= 0
\end{align*}
\]

\[
W_3 = \left(\begin{array}{ccc}
(W_3)_{1,1}(W_3)_{1,2}(W_3)_{1,3} \\
(W_3)_{2,1}(W_3)_{2,2}(W_3)_{2,3}
\end{array}\right)
\]

\[
b_3 = \left(\begin{array}{c}
(b_3)_1 \\
(b_3)_2
\end{array}\right)
\]

\[
\begin{align*}
x &= \left(\begin{array}{c}
x_1 \\
x_2
\end{array}\right) \\
a_1 &= \sigma(z_1) \\
&= \sigma(W_1 x + b_1) \\
W_1 &= \left(\begin{array}{ccc}
(W_1)_{1,1}(W_1)_{1,2} \\
(W_1)_{2,1}(W_1)_{2,2} \\
(W_1)_{3,1}(W_1)_{3,2}
\end{array}\right) \\
b_1 &= \left(\begin{array}{c}
(b_1)_1 \\
(b_1)_2 \\
(b_1)_3
\end{array}\right)
\end{align*}
\]

\[
\begin{align*}
a_2 &= \sigma(z_2) \\
&= \sigma(W_2 a_1 + b_2) \\
W_2 &= \left(\begin{array}{ccc}
(W_2)_{1,1}(W_2)_{1,2}(W_2)_{1,3} \\
(W_2)_{2,1}(W_2)_{2,2}(W_2)_{2,3} \\
(W_2)_{3,1}(W_2)_{3,2}(W_2)_{3,3}
\end{array}\right) \\
b_2 &= \left(\begin{array}{c}
(b_2)_1 \\
(b_2)_2 \\
(b_2)_3
\end{array}\right)
\end{align*}
\]

\[
\Phi(x) = z_3 = W_3 a_2 + b_3
\]
\[
\frac{\partial(z_3)_k}{\partial(W_3)_{i,j}} = \begin{cases} 
(a_2)_j & i = k \\
0 & i \neq k 
\end{cases}
\]

\[
W_3 = \begin{pmatrix}
(W_3)_{1,1} & (W_3)_{1,2} & (W_3)_{1,3} \\
(W_3)_{2,1} & (W_3)_{2,2} & (W_3)_{2,3} 
\end{pmatrix}
\]

\[
b_3 = \begin{pmatrix}
(b_3)_1 \\
(b_3)_2 
\end{pmatrix}
\]

\[
x = \begin{pmatrix}
(x)_1 \\
(x)_2 
\end{pmatrix}
\]

\[
a_1 = \sigma(z_1)
\]

\[
a_2 = \sigma(z_2)
\]

\[
\Phi(x) = z_3
\]

\[
x = \sigma(W_1 x + b_1)
\]

\[
W_1 = \begin{pmatrix}
(W_1)_{1,1} & (W_1)_{1,2} \\
(W_1)_{2,1} & (W_1)_{2,2} \\
(W_1)_{3,1} & (W_1)_{3,2} 
\end{pmatrix}
\]

\[
W_2 = \begin{pmatrix}
(W_2)_{1,1} & (W_2)_{1,2} & (W_2)_{1,3} \\
(W_2)_{2,1} & (W_2)_{2,2} & (W_2)_{2,3} \\
(W_2)_{3,1} & (W_2)_{3,2} & (W_2)_{3,3} 
\end{pmatrix}
\]

\[
b_1 = \begin{pmatrix}
(b_1)_1 \\
(b_1)_2 \\
(b_1)_3 
\end{pmatrix}
\]

\[
b_2 = \begin{pmatrix}
(b_2)_1 \\
(b_2)_2 \\
(b_2)_3 
\end{pmatrix}
\]
\[
\frac{\partial (z_3)_k}{\partial (W_3)_{i,j}} = \begin{cases} 
(a_2)_j & i = k \\
0 & i \neq k 
\end{cases}
\]

\[
\frac{\partial (z_3)_k}{\partial (b_3)_i} = \begin{cases} 
1 & i = k \\
0 & i \neq k 
\end{cases}
\]

\[
x = \begin{pmatrix} (x)_1 \\ (x)_2 \end{pmatrix} \\
a_1 = \sigma(z_1) \\
= \sigma(W_1 x + b_1) \\
W_1 = \begin{pmatrix} (W_1)_{1,1} & (W_1)_{1,2} \\ (W_1)_{2,1} & (W_1)_{2,2} \\ (W_1)_{3,1} & (W_1)_{3,2} \end{pmatrix} \\
b_1 = \begin{pmatrix} (b_1)_1 \\ (b_1)_2 \\ (b_1)_3 \end{pmatrix}
\]

\[
a_2 = \sigma(z_2) \\
= \sigma(W_2 a_1 + b_2) \\
W_2 = \begin{pmatrix} (W_2)_{1,1} & (W_2)_{1,2} & (W_2)_{1,3} \\ (W_2)_{2,1} & (W_2)_{2,2} & (W_2)_{2,3} \\ (W_2)_{3,1} & (W_2)_{3,2} & (W_2)_{3,3} \end{pmatrix} \\
b_2 = \begin{pmatrix} (b_2)_1 \\ (b_2)_2 \\ (b_2)_3 \end{pmatrix}
\]

\[
\Phi(x) = z_3 \\
= W_3 a_2 + b_3
\]

\[
W_3 = \begin{pmatrix} (W_3)_{1,1} & (W_3)_{1,2} & (W_3)_{1,3} \\ (W_3)_{2,1} & (W_3)_{2,2} & (W_3)_{2,3} \end{pmatrix} \\
b_3 = \begin{pmatrix} (b_3)_1 \\ (b_3)_2 \end{pmatrix}
\]
\[
\frac{\partial \Phi(x)}{\partial W_3} = \begin{pmatrix}
\begin{pmatrix} (a_2)_1 \\ 0 \\ 0 \\ (a_2)_1 \\ 0 \\ 0 \\ (a_2)_2 \\ 0 \\ (a_2)_3 \\ 0 
\end{pmatrix}
\end{pmatrix}
\]

\[
\frac{\partial \Phi(x)}{\partial b_3} = \begin{pmatrix}
1 \\
0 \\
0 
\end{pmatrix}
\]

\[
x = \begin{pmatrix} (x)_1 \\ (x)_2 \end{pmatrix}
\]

\[
a_1 = \sigma(z_1) = \sigma(W_1 x + b_1) = \begin{pmatrix} (W_1)_{1,1} (W_1)_{1,2} \\ (W_1)_{2,1} (W_1)_{2,2} \\ (W_1)_{3,1} (W_1)_{3,2} \end{pmatrix}
\]

\[
a_2 = \sigma(z_2) = \sigma(W_2 a_1 + b_2) = \begin{pmatrix} (W_2)_{1,1} (W_2)_{1,2} (W_2)_{1,3} \\ (W_2)_{2,1} (W_2)_{2,2} (W_2)_{2,3} \\ (W_2)_{3,1} (W_2)_{3,2} (W_2)_{3,3} \end{pmatrix}
\]

\[
\Phi(x) = z_3 = W_3 a_2 + b_3
\]
Backprop: Last Layer

\[
\frac{\partial L(f, x, y)}{\partial W_L} \equiv 2 \left( f(x) - y \right)^T \frac{\partial f(x)}{\partial W_L} \equiv 2 \left( f(x) - y \right)^T \frac{\partial f(x)}{\partial b_L}.
\]

Let \( f(x) = W_L \sigma(W_L - 1(\ldots) + b_L - 1) + b_L \). It follows that

\[
\frac{\partial f(x)}{\partial W_L} \equiv (0, \ldots, \sigma(W_L - 1(\ldots) + b_L - 1))_j \equiv (0, \ldots, 1) \frac{\partial f(x)}{\partial b_L} \equiv (0, \ldots, 0)^T.
\]

In matrix notation:

\[
\frac{\partial L(f, x, y)}{\partial W_L} \equiv 2 \left( f(x) - y \right)^T \delta L(\sigma(z_L - 1(\ldots) + b_L - 1)) \equiv 2 \left( f(x) - y \right)^T a_L - 1.
\]

\[
\frac{\partial L(f, x, y)}{\partial b_L} \equiv 2 \left( f(x) - y \right) \equiv 2 \left( f(x) - y \right).
\]
Backprop: Last Layer

\[
\frac{\partial L(f,x,y)}{\partial (W_L)_{i,j}} = 2 \cdot (f(x) - y)^T \cdot \frac{\partial f(x)}{\partial (W_L)_{i,j}},
\]
\[
\frac{\partial L(f,x,y)}{\partial (b_L)_i} = 2 \cdot (f(x) - y)^T \cdot \frac{\partial f(x)}{\partial (b_L)_i}.
\]
**Backprop: Last Layer**

\[
\frac{\partial L(f,x,y)}{\partial (W_L)_{i,j}} = 2 \cdot (f(x) - y)^T \cdot \frac{\partial f(x)}{\partial (W_L)_{i,j}},
\]

\[
\frac{\partial L(f,x,y)}{\partial (b_L)_i} = 2 \cdot (f(x) - y)^T \cdot \frac{\partial f(x)}{\partial (b_L)_i}.
\]

Let \( f(x) = W_L \sigma(W_{L-1}(\ldots) + b_{L-1}) + b_L \). It follows that

\[
\frac{\partial f(x)}{\partial (W_L)_{i,j}} = (0, \ldots, \sigma(W_{L-1}(\ldots) + b_{L-1})_j, \ldots, 0)^T
\]

\[
\frac{\partial f(x)}{\partial (b_L)_i} = (0, \ldots, 1_i, \ldots, 0)^T
\]
Backprop: Last Layer

\[
\frac{\partial L(f,x,y)}{\partial (W_L)_{i,j}} = 2 \cdot (f(x) - y)^T \cdot \frac{\partial f(x)}{\partial (W_L)_{i,j}},
\]
\[
\frac{\partial L(f,x,y)}{\partial (b_L)_i} = 2 \cdot (f(x) - y)^T \cdot \frac{\partial f(x)}{\partial (b_L)_i}.
\]

Let \( f(x) = W_L \sigma(W_{L-1}(\ldots) + b_{L-1}) + b_L \). It follows that

\[
\frac{\partial f(x)}{\partial (W_L)_{i,j}} = (0, \ldots, \sigma(W_{L-1}(\ldots) + b_{L-1})_j, \ldots, 0)^T
\]
\[
\frac{\partial f(x)}{\partial (b_L)_i} = (0, \ldots, 1^i, \ldots, 0)^T
\]
\[
2(f(x) - y)^T \frac{\partial f(x)}{\partial (W_L)_{i,j}} = 2(f(x) - y)_i \sigma(W_{L-1}(\ldots) + b_{L-1})_j,
\]
\[
2(f(x) - y)^T \frac{\partial f(x)}{\partial (b_L)_i} = 2(f(x) - y)_i
\]
Backprop: Last Layer

\[
\frac{\partial L(f, x, y)}{\partial (W_L)_{i,j}} = 2 \cdot (f(x) - y)^T \cdot \frac{\partial f(x)}{\partial (W_L)_{i,j}},
\]

\[
\frac{\partial L(f, x, y)}{\partial (b_L)_i} = 2 \cdot (f(x) - y)^T \cdot \frac{\partial f(x)}{\partial (b_L)_i}.
\]

Let \( f(x) = W_L \sigma(W_{L-1}(\ldots) + b_{L-1}) + b_L \). It follows that

\[
\frac{\partial f(x)}{\partial (W_L)_{i,j}} = (0, \ldots, \sigma(W_{L-1}(\ldots) + b_{L-1})_j, \ldots, 0)^T
\]

\[
\frac{\partial f(x)}{\partial (b_L)_i} = (0, \ldots, 1_i, \ldots, 0)^T
\]

\[
2(f(x) - y)^T \frac{\partial f(x)}{\partial (W_L)_{i,j}} = 2(f(x) - y)_i \sigma(W_{L-1}(\ldots) + b_{L-1})_j,
\]

\[
2(f(x) - y)^T \frac{\partial f(x)}{\partial (b_L)_i} = 2(f(x) - y)_i
\]

In matrix notation:

\[
\frac{\partial L(f, x, y)}{\partial W_L} = 2(f(x) - y)(\sigma(W_{L-1}(\ldots) + b_{L-1}))^T,
\]

\[
\frac{\partial L(f, x, y)}{\partial b_L} = 2(f(x) - y).
\]
Define $a_{\ell+1} = \sigma(z_{\ell+1})$ where $z_{\ell+1} = W_{\ell+1}a_{\ell} + b_{\ell+1}$, $a_0 = x$, $f(x) = z_L$. 
Backprop: Second-to-last Layer

- Define $a_{\ell+1} = \sigma(z_{\ell+1})$ where $z_{\ell+1} = W_{\ell+1}a_\ell + b_{\ell+1}$, $a_0 = x$, $f(x) = z_L$.
- We have computed $\frac{\mathcal{L}(f,x,y)}{\partial W_L}$, $\frac{\mathcal{L}(f,x,y)}{\partial b_L}$.
Define $a_{\ell+1} = \sigma(z_{\ell+1})$ where $z_{\ell+1} = W_{\ell+1}a_\ell + b_{\ell+1}$, $a_0 = x$, $f(x) = z_L$.

We have computed $\frac{\mathcal{L}(f,x,y)}{\partial W_L}$, $\frac{\mathcal{L}(F,x,y)}{\partial b_L}$.

Then, use chain rule:

$$\frac{\partial \mathcal{L}(f, x, y)}{\partial W_{L-1}} = \frac{\partial \mathcal{L}(f, x, y)}{a_{L-1}} \cdot \frac{\partial a_{L-1}}{\partial W_{L-1}} = 2(f(x) - y)^T \cdot W_L \cdot \frac{\partial a_{L-1}}{\partial W_{L-1}}.$$
Backprop: Second-to-last Layer

- Define \( a_{\ell+1} = \sigma(z_{\ell+1}) \) where \( z_{\ell+1} = W_{\ell+1}a_\ell + b_{\ell+1}, a_0 = x, f(x) = z_L \).
- We have computed \( \frac{\mathcal{L}(f,x,y)}{\partial W_L}, \frac{\mathcal{L}(F,x,y)}{\partial b_L} \).
- Then, use chain rule:

\[
\frac{\partial \mathcal{L}(f, x, y)}{\partial W_{L-1}} = \frac{\partial \mathcal{L}(f, x, y)}{a_{L-1}} \cdot \frac{\partial a_{L-1}}{\partial W_{L-1}} = 2(f(x) - y)^T \cdot W_L \cdot \frac{\partial a_{L-1}}{\partial W_{L-1}}.
\]

\[
= 2(f(x) - y)^T \cdot W_L \cdot \text{diag}(\sigma'(z_{L-1})) \cdot \frac{\partial z_{L-1}}{\partial W_{L-1}}
\]
Define $a_{\ell+1} = \sigma(z_{\ell+1})$ where $z_{\ell+1} = W_{\ell+1}a_\ell + b_{\ell+1}$, $a_0 = x$, $f(x) = z_L$.

We have computed $\frac{\mathcal{L}(f,x,y)}{\partial W_L}, \frac{\mathcal{L}(F,x,y)}{\partial b_L}$.

Then, use chain rule:

$$
\frac{\partial \mathcal{L}(f,x,y)}{\partial W_{L-1}} = \frac{\partial \mathcal{L}(f,x,y)}{a_{L-1}} \cdot \frac{\partial a_{L-1}}{\partial W_{L-1}} = 2(f(x) - y)^T \cdot W_L \cdot \frac{\partial a_{L-1}}{\partial W_{L-1}}.
$$

$$
= 2(f(x) - y)^T \cdot W_L \cdot \text{diag}(\sigma'(z_{L-1})) \cdot \frac{\partial z_{L-1}}{\partial W_{L-1}}.
$$

same as before!
Backprop: Second-to-last Layer

- Define $a_{\ell + 1} = \sigma(z_{\ell + 1})$ where $z_{\ell + 1} = W_{\ell + 1}a_\ell + b_{\ell + 1}$, $a_0 = x$, $f(x) = z_L$.

- We have computed $\frac{\mathcal{L}(f, x, y)}{\partial W_L}$, $\frac{\mathcal{L}(F, x, y)}{\partial b_L}$.

- Then, use chain rule:

$$
\frac{\partial \mathcal{L}(f, x, y)}{\partial W_{L-1}} = \frac{\partial \mathcal{L}(f, x, y)}{a_{L-1}} \cdot \frac{\partial a_{L-1}}{\partial W_{L-1}} = 2(f(x) - y)^T \cdot W_L \cdot \frac{\partial a_{L-1}}{\partial W_{L-1}}.
$$

$$
= 2(f(x) - y)^T \cdot W_L \cdot \text{diag}(\sigma'(z_{L-1})) \cdot \frac{\partial z_{L-1}}{\partial W_{L-1}}
$$

$$
= \text{diag}(\sigma'(z_{L-1})) \cdot W_{L}^T \cdot 2(f(x) - y) \cdot a_{L-2}^T.
$$

$\delta_{L-1}$
Define $a_{\ell+1} = \sigma(z_{\ell+1})$ where $z_{\ell+1} = W_{\ell+1}a_{\ell} + b_{\ell+1}$, $a_0 = x$, $f(x) = z_L$.

We have computed $\frac{\mathcal{L}(f,x,y)}{\partial W_L}$, $\frac{\mathcal{L}(F,x,y)}{\partial b_L}$.

Then, use chain rule:

$$
\frac{\partial \mathcal{L}(f, x, y)}{\partial W_{L-1}} = \frac{\partial \mathcal{L}(f, x, y)}{a_{L-1}} \cdot \frac{\partial a_{L-1}}{\partial W_{L-1}} = 2(f(x) - y)^T \cdot W_L \cdot \frac{\partial a_{L-1}}{\partial W_{L-1}}.
$$

\[
= 2(f(x) - y)^T \cdot W_L \cdot \text{diag}(\sigma'(z_{L-1})) \cdot \frac{\partial z_{L-1}}{\partial W_{L-1}}
\]

\[
= \text{diag}(\sigma'(z_{L-1})) \cdot W_L^T \cdot 2(f(x) - y) \cdot a_{L-2}^T \cdot \delta_{L-1}
\]

Similar arguments yield $\frac{\partial \mathcal{L}(f,x,y)}{\partial b_{L-1}} = \delta_{L-1}$. 
The Backprop Algorithm

1. Calculate $a^\ell = \sigma(z^\ell)$, $z^\ell = A^\ell(a^\ell - 1)$ for $\ell = 1, \ldots, L$ (forward pass).

2. Set $\delta^L = 2( f(x) - y)$.

3. Then $\frac{\partial L(f,x,y)}{\partial b^L} = \delta^L$ and $\frac{\partial L(f,x,y)}{\partial W^L} = \delta^L \cdot a^{T,\ell-1}$.

4. for $\ell$ from $L-1$ to 1 do:
   - $\delta^\ell = \text{diag}(\sigma'(z^\ell)) \cdot W^{T,\ell+1} \cdot \delta^{\ell+1}$
   - Then $\frac{\partial L(f,x,y)}{\partial b^\ell} = \delta^\ell$ and $\frac{\partial L(f,x,y)}{\partial W^\ell} = \delta^\ell \cdot a^{T,\ell-1}$.

5. return $\frac{\partial L(f,x,y)}{\partial b^\ell}, \frac{\partial L(f,x,y)}{\partial W^\ell}, \ell = 1, \ldots, L$. 
The Backprop Algorithm

1. Calculate $a_\ell = \sigma(z_\ell)$, $z_\ell = A_\ell(a_{\ell-1})$ for $\ell = 1, \ldots, L$, $a_0 = x$ (forward pass).
The Backprop Algorithm

1. Calculate $a_\ell = \sigma(z_\ell)$, $z_\ell = A_\ell(a_{\ell-1})$ for $\ell = 1, \ldots, L$, $a_0 = x$ (forward pass).

2. Set $\delta_L = 2(f(x) - y)$
The Backprop Algorithm

1. Calculate $a_\ell = \sigma(z_\ell)$, $z_\ell = A_\ell(a_{\ell-1})$ for $\ell = 1, \ldots, L$, $a_0 = x$ (forward pass).
2. Set $\delta_L = 2(f(x) - y)$
3. Then $\frac{\partial L(f, x, y)}{\partial b_L} = \delta_L$ and $\frac{\partial L(f, x, y)}{\partial W_L} = \delta_L \cdot a_{L-1}^T$. 
The Backprop Algorithm

1. Calculate $a_\ell = \sigma(z_\ell)$, $z_\ell = A_\ell(a_{\ell-1})$ for $\ell = 1, \ldots, L$, $a_0 = x$ (forward pass).

2. Set $\delta_L = 2(f(x) - y)$

3. Then $\frac{\partial L(f,x,y)}{\partial b_L} = \delta_L$ and $\frac{\partial L(f,x,y)}{\partial W_L} = \delta_L \cdot a_{L-1}^T$.

4. for $\ell$ from $L - 1$ to 1 do:
The Backprop Algorithm

1. Calculate $a_\ell = \sigma(z_\ell)$, $z_\ell = A_\ell(a_{\ell-1})$ for $\ell = 1, \ldots, L$, $a_0 = x$ (forward pass).

2. Set $\delta_L = 2(f(x) - y)$

3. Then $\frac{\partial L(f, x, y)}{\partial b_L} = \delta_L$ and $\frac{\partial L(f, x, y)}{\partial W_L} = \delta_L \cdot a_{L-1}^T$.

4. for $\ell$ from $L - 1$ to 1 do:
   - $\delta_\ell = \text{diag}(\sigma'(z_\ell)) \cdot W_{\ell+1}^T \cdot \delta_{\ell+1}$
The Backprop Algorithm

1. Calculate $a_\ell = \sigma(z_\ell)$, $z_\ell = A_\ell(a_{\ell-1})$ for $\ell = 1, \ldots, L$, $a_0 = x$ (forward pass).

2. Set $\delta_L = 2(f(x) - y)$

3. Then $\frac{\partial L(f,x,y)}{\partial b_L} = \delta_L$ and $\frac{\partial L(f,x,y)}{\partial W_L} = \delta_L \cdot a_{L-1}^T$.

4. for $\ell$ from $L - 1$ to 1 do:
   - $\delta_\ell = \text{diag}(\sigma'(z_\ell)) \cdot W_{\ell+1}^T \cdot \delta_{\ell+1}$
   - Then $\frac{\partial L(f,x,y)}{\partial b_\ell} = \delta_\ell$ and $\frac{\partial L(f,x,y)}{\partial W_\ell} = \delta_\ell \cdot a_{\ell-1}^T$. 

return $\frac{\partial L(f,x,y)}{\partial b_\ell}, \frac{\partial L(f,x,y)}{\partial W_\ell}, \ell = 1, \ldots, L$. 

The Backprop Algorithm

1. Calculate $a_\ell = \sigma(z_\ell)$, $z_\ell = A_\ell(a_{\ell-1})$ for $\ell = 1, \ldots, L$, $a_0 = x$ (forward pass).

2. Set $\delta_L = 2(f(x) - y)$

3. Then $\frac{\partial L(f,x,y)}{\partial b_L} = \delta_L$ and $\frac{\partial L(f,x,y)}{\partial W_L} = \delta_L \cdot a^T_{L-1}$.

4. for $\ell$ from $L - 1$ to 1 do:
   - $\delta_\ell = \text{diag}(\sigma'(z_\ell)) \cdot W_{\ell+1}^T \cdot \delta_{\ell+1}$
   - Then $\frac{\partial L(f,x,y)}{\partial b_\ell} = \delta_\ell$ and $\frac{\partial L(f,x,y)}{\partial W_\ell} = \delta_\ell \cdot a^T_{\ell-1}$.

5. return $\frac{\partial L(f,x,y)}{\partial b_\ell}$, $\frac{\partial L(f,x,y)}{\partial W_\ell}$, $l = 1, \ldots, L$. 
Computational Graphs

(a) $z = x \times y$

(b) $u^{(1)} \rightarrow u^{(2)}$

(c) $H = \text{relu}(U^{(1)} + U^{(2)})$

(d) $\hat{y} = x \text{ dot } w \text{ sqr } \lambda$
Automatic Differentiation
4 Stochastic Gradient Descent
Recall that one gradient descent step requires the calculation of

\[ \sum_{i=1}^{m} \nabla ((W_{\ell}, b_{\ell}))_{\ell=1}^{L} \mathcal{L}(f, x_i, y_i). \]

and each of the summands requires one backpropagation run.
Recall that one gradient descent step requires the calculation of

\[ \sum_{i=1}^{m} \nabla((W_\ell,b_\ell))_{\ell=1}^L \mathcal{L}(f, x_i, y_i). \]

and each of the summands requires one backpropagation run. Thus, the total complexity of one gradient descent step is equal to

\[ m \cdot \text{complexity(backprop)}. \]
Recall that one gradient descent step requires the calculation of

\[ \sum_{i=1}^{m} \nabla ((W_{\ell}, b_{\ell}))_{\ell=1}^{L} \mathcal{L}(f, x_i, y_i). \]

and each of the summands requires one backpropagation run. Thus, the total complexity of one gradient descent step is equal to

\[ m \cdot \text{complexity}(\text{backprop}). \]

The complexity of backprop is asymptotically equal to the number of DOFs of the network:

\[ \text{complexity}(\text{backprop}) \sim \sum_{\ell=1}^{L} N_{\ell-1} \times N_{\ell} + N_{\ell}. \]
An Example

ImageNet database consists of $\sim 1.2 \times 10^7$ images and 1000 categories.

AlexNet, neural network with $\sim 160 \times 10^6$ DOFs is one of the most successful annotation methods.

One step of gradient descent requires $\sim 2 \times 10^{14}$ flops (and memory units)!!
An Example

- ImageNet database consists of $\sim 1.2m$ images and 1000 categories.
An Example

- ImageNet database consists of \( \sim 1.2m \) images and 1000 categories.
- AlexNet, neural network with \( \sim 160m \) DOFs is one of the most successful annotation methods.
ImageNet database consists of $\sim 1.2m$ images and 1000 categories.

AlexNet, neural network with $\sim 160m$ DOFs is one of the most successful annotation methods.

One step of gradient descent requires $\sim 2 \times 10^{14}$ flops (and memory units)!!
Approximate

\[ \sum_{i=1}^{m} \nabla (W_{\ell}, b_{\ell}) \ell_{\ell=1} L(f, x_i, y_i) \]

by

\[ \nabla (W_{\ell}, b_{\ell}) L_{\ell=1} L(f, x_{i^*}, y_{i^*}) \]

for some \( i^* \) chosen uniformly at random from \( \{1, \ldots, m\} \).
Approximate

\[
\sum_{i=1}^{m} \nabla_{\ell=1}^{L} \mathcal{L}(f, x_i, y_i)
\]

by

\[
\nabla_{\ell=1}^{L} \mathcal{L}(f, x_{i^*}, y_{i^*})
\]

for some \( i^* \) chosen uniformly at random from \( \{1, \ldots, m\} \).

In expectation we have

\[
\mathbb{E} \nabla_{\ell=1}^{L} \mathcal{L}(f, x_{i^*}, y_{i^*}) = \frac{1}{m} \sum_{i=1}^{m} \nabla_{\ell=1}^{L} \mathcal{L}(f, x_i, y_i)
\]
The SGD Algorithm

**Goal:** Find stationary point of function \( F = \sum_{i=1}^{m} F_i : \mathbb{R}^N \rightarrow \mathbb{R}. \)
The SGD Algorithm

**Goal:** Find stationary point of function \( F = \sum_{i=1}^{m} F_i : \mathbb{R}^N \rightarrow \mathbb{R} \).

1. Set starting value \( u_0 \) and \( n = 0 \)
The SGD Algorithm

**Goal:** Find stationary point of function $F = \sum_{i=1}^{m} F_i : \mathbb{R}^N \to \mathbb{R}$.

1. Set starting value $u_0$ and $n = 0$
2. **while** (error is large) **do**:
The SGD Algorithm

**Goal:** Find stationary point of function $F = \sum_{i=1}^{m} F_i : \mathbb{R}^N \rightarrow \mathbb{R}$.

1. Set starting value $u_0$ and $n = 0$

2. **while** (error is large) **do**:
   - Pick $i \in \{1, \ldots, m\}$ uniformly at random

   $$u_{n+1} = u_n - \eta \nabla F_i$$

   $n = n + 1$

3. return $u_n$
The SGD Algorithm

Goal: Find stationary point of function $F = \sum_{i=1}^{m} F_i : \mathbb{R}^N \to \mathbb{R}$.

1. Set starting value $u_0$ and $n = 0$
2. while (error is large) do:
   - Pick $i \in \{1, \ldots, m\}$ uniformly at random
   - update $u_{n+1} = u_n - \eta \nabla F_i$
The SGD Algorithm

**Goal:** Find stationary point of function \( F = \sum_{i=1}^{m} F_i : \mathbb{R}^N \rightarrow \mathbb{R} \).

1. Set starting value \( u_0 \) and \( n = 0 \)

2. **while** (error is large) **do:**
   - Pick \( i \in \{1, \ldots, m\} \) uniformly at random
   - update \( u_{n+1} = u_n - \eta \nabla F_i \)
   - \( n = n + 1 \)
The SGD Algorithm

**Goal:** Find stationary point of function $F = \sum_{i=1}^{m} F_i : \mathbb{R}^N \rightarrow \mathbb{R}$.

1. Set starting value $u_0$ and $n = 0$
2. **while** (error is large) **do**:
   - Pick $i \in \{1, \ldots, m\}$ uniformly at random
   - **update** $u_{n+1} = u_n - \eta \nabla F_i^*$
   - $n = n + 1$
3. **return** $u_n$
Typical Behavior

Figure: Comparison btw. GD and SGD. \( m \) steps of SGD are counted as one iteration.
Typical Behavior

Figure: Comparison btw. GD and SGD. \(m\) steps of SGD are counted as one iteration.

Initially very fast convergence, followed by stagnation!
Minibatch SGD

For every \( \{i_1^*, \ldots, i_K^*\} \subset \{1, \ldots, m\} \) chosen uniformly at random, it holds that

\[
\mathbb{E} \frac{1}{K} \sum_{l=1}^{k} \nabla_{(W_l, b_l)} \mathcal{L}(f, x_{i_l^*}, y_{i_l^*}) = \frac{1}{m} \sum_{i=1}^{m} \nabla_{(W_l, b_l)} \mathcal{L}(f, x_i, y_i),
\]

e.g., we have an unbiased estimator for the gradient.
Minibatch SGD

For every \( \{i_1^*, \ldots, i_K^*\} \subset \{1, \ldots, m\} \) chosen uniformly at random, it holds that

\[
\mathbb{E} \frac{1}{K} \sum_{l=1}^{k} \nabla \left( (W_{\ell}, b_{\ell}) \right)_{\ell=1}^{L} \mathcal{L}(f, x_{i_l^*}, y_{i_l^*}) = \frac{1}{m} \sum_{i=1}^{m} \nabla \left( (W_{\ell}, b_{\ell}) \right)_{\ell=1}^{L} \mathcal{L}(f, x_i, y_i),
\]

e.g., we have an \textit{unbiased estimator} for the gradient.

\[ K = 1 \quad \leadsto \quad \text{SGD} \]
For every \( \{i_1^*, \ldots, i_K^*\} \subset \{1, \ldots, m\} \) chosen uniformly at random, it holds that

\[
\mathbb{E} \frac{1}{K} \sum_{l=1}^{k} \nabla \mathcal{L}(f, x_{i_l}^*, y_{i_l}^*) = \frac{1}{m} \sum_{i=1}^{m} \nabla \mathcal{L}(f, x_i, y_i),
\]

e.g., we have an unbiased estimator for the gradient.

- \( K = 1 \rightarrow \text{SGD} \)
- \( K > 1 \rightarrow \text{Minibatch SGD with batchsize } K \).
The sample mean \[ \bar{K} = \frac{1}{\sum k_{l=1}^K} \nabla \left( \left( W_{\ell}, b_{\ell} \right) \right) \] of the loss \[ L_{\ell=1}^L \left( f, x_i^*, y_i^* \right) \] is a random variable with expected value \[ \frac{1}{m} \sum_{i=1}^m \nabla \left( \left( W_{\ell}, b_{\ell} \right) \right) \].

To assess the deviation of the sample mean from its expected value, we compute its standard deviation \[ \sigma / \sqrt{n} \] where \[ \sigma \] is the standard deviation of \[ i \mapsto \nabla \left( \left( W_{\ell}, b_{\ell} \right) \right) \].

Increasing the batch size by a factor of 100 yields an improvement in variance by a factor of 10 while the complexity increases by a factor of 1000.

Common batch size for large models: \[ K = 16, 32 \].
Some Heuristics

- The sample mean \( \frac{1}{K} \sum_{l=1}^{k} \nabla \mathcal{L}(f, x_i^*, y_i^*) \) is itself a random variable that has expected value

\[
\frac{1}{m} \sum_{i=1}^{m} \nabla \mathcal{L}(f, x_i, y_i).
\]
The sample mean $\frac{1}{K} \sum_{l=1}^{k} \nabla ((W_{\ell}, b_{\ell}))_{\ell=1}^{L} \mathcal{L}(f, x_{i_l}^*, y_{i_l}^*)$ is itself a random variable that has expected value $\frac{1}{m} \sum_{i=1}^{m} \nabla ((W_{\ell}, b_{\ell}))_{\ell=1}^{L} \mathcal{L}(f, x_i, y_i)$.

In order to assess the deviation of the sample mean from its expected value we may compute its standard deviation $\sigma/\sqrt{n}$ where $\sigma$ is the standard deviation of $i \mapsto \nabla ((W_{\ell}, b_{\ell}))_{\ell=1}^{L} \mathcal{L}(f, x_i, y_i)$. 

Increasing the batch size by a factor of 100 yields an improvement of the variance by a factor of 10 while the complexity increases by a factor of 100! Common batch size for large models: $K = 16, 32$. 

Some Heuristics
Some Heuristics

- The sample mean $\frac{1}{K} \sum_{l=1}^{K} \nabla ((W_{\ell},b_{\ell}))_{\ell=1}^{L} \mathcal{L}(f, x_{i_l}^*, y_{i_l}^*)$ is itself a random variable that has expected value $\frac{1}{m} \sum_{i=1}^{m} \nabla ((W_{\ell},b_{\ell}))_{\ell=1}^{L} \mathcal{L}(f, x_{i}, y_{i})$.

- In order to assess the deviation of the sample mean from its expected value we may compute its standard deviation $\sigma/\sqrt{n}$ where $\sigma$ is the standard deviation of $i \mapsto \nabla ((W_{\ell},b_{\ell}))_{\ell=1}^{L} \mathcal{L}(f, x_{i}, y_{i})$.

Increasing the batch size by a factor 100 yields an improvement of the variance by a factor 10 while the complexity increases by a factor 100!
Some Heuristics

- The sample mean \( \frac{1}{K} \sum_{l=1}^{k} \nabla \left( (W_{\ell},b_{\ell}) \right)_{\ell=1}^{L} \mathcal{L}(f, x_{i*}, y_{i*}) \) is itself a random variable that has expected value \( \frac{1}{m} \sum_{i=1}^{m} \nabla \left( (W_{\ell},b_{\ell}) \right)_{\ell=1}^{L} \mathcal{L}(f, x_{i}, y_{i}) \).

- In order to assess the deviation of the sample mean from its expected value we may compute its standard deviation \( \sigma / \sqrt{n} \) where \( \sigma \) is the standard deviation of \( i \mapsto \nabla \left( (W_{\ell},b_{\ell}) \right)_{\ell=1}^{L} \mathcal{L}(f, x_{i}, y_{i}) \).

Increasing the batch size by a factor 100 yields an improvement of the variance by a factor 10 while the complexity increases by a factor 100!

Common batchsize for large models: \( K = 16, 32 \).
5 The Basic Recipe
The basic Neural Network Recipe for Learning

Now let's try classifying handwritten digits!
The basic Neural Network Recipe for Learning

1. Neuro-inspired model
The basic Neural Network Recipe for Learning

1. Neuro-inspired model
2. Backprop
The basic Neural Network Recipe for Learning

1. Neuro-inspired model
2. Backprop
3. Minibatch SGD
The basic Neural Network Recipe for Learning

1. Neuro-inspired model
2. Backprop
3. Minibatch SGD

Now let’s try classifying handwritten digits!
MNIST dataset, 30 epochs, learning rate $\eta = 3.0$, minibatch size $K = 10$, training set size $m = 50000$, test set size $= 10000$. Deep learning might not help after all...
Results

MNIST dataset, 30 epochs, learning rate $\eta = 3.0$, minibatch size $K = 10$, training set size $m = 50000$, test set size $= 10000$.

- network size $[784, 30, 10]$.
MNIST dataset, 30 epochs, learning rate $\eta = 3.0$, minibatch size $K = 10$, training set size $m = 50000$, test set size $= 10000$.

- network size $[784, 30, 10]$. Classification accuracy 94.84%.
Results

MNIST dataset, 30 epochs, learning rate $\eta = 3.0$, minibatch size $K = 10$, training set size $m = 50000$, test set size $= 10000$.
- network size $[784, 30, 10]$. Classification accuracy 94.84%.
- network size $[784, 30, 30, 10]$. Deep learning might not help after all...
MNIST dataset, 30 epochs, learning rate $\eta = 3.0$, minibatch size $K = 10$, training set size $m = 50000$, test set size $= 10000$.

- network size $[784, 30, 10]$. Classification accuracy $94.84\%$.
- network size $[784, 30, 30, 10]$. Classification accuracy $95.81\%$.
Results

MNIST dataset, 30 epochs, learning rate $\eta = 3.0$, minibatch size $K = 10$, training set size $m = 50000$, test set size $= 10000$.

- network size $[784, 30, 10]$. Classification accuracy 94.84%.
- network size $[784, 30, 30, 10]$. Classification accuracy 95.81%.
- network size $[784, 30, 30, 30, 10]$. Deep learning might not help after all...
Results

MNIST dataset, 30 epochs, learning rate $\eta = 3.0$, minibatch size $K = 10$, training set size $m = 50000$, test set size $= 10000$.

- network size $[784, 30, 10]$. Classification accuracy 94.84%.
- network size $[784, 30, 30, 10]$. Classification accuracy 95.81%.
- network size $[784, 30, 30, 30, 10]$. Classification accuracy 95.07%.

Deep learning might not help after all...
Results

MNIST dataset, 30 epochs, learning rate $\eta = 3.0$, minibatch size $K = 10$, training set size $m = 50000$, test set size = 10000.

- network size $[784, 30, 10]$. Classification accuracy 94.84%.
- network size $[784, 30, 30, 10]$. Classification accuracy 95.81%.
- network size $[784, 30, 30, 30, 10]$. Classification accuracy 95.07%.

Deep learning might not help after all...
6 Going Deep (?)
Problems with Deep Networks

Overfitting (as usual...)

Vanishing/Exploding Gradient Problem
Problems with Deep Networks

- Overfitting (as usual...)
Problems with Deep Networks

- Overfitting (as usual...)
- Vanishing/Exploding Gradient Problem
Dealing with Overfitting: Regularization

Rather than minimizing

$$\sum_{i=1}^{m} \mathcal{L}(f, x_i, y_i),$$

minimize

$$\sum_{i=1}^{m} \mathcal{L}(f, x_i, y_i) + \lambda \Omega((W_\ell)_{\ell=1}^L),$$

for example

$$\Omega((W_\ell)_{\ell=1}^L) = \sum_{l,i,j} |(W_\ell)_{i,j}|^p.$$
Dealing with Overfitting: Regularization

Rather than minimizing

\[
\sum_{i=1}^{m} \mathcal{L}(f, x_i, y_i),
\]

minimize

\[
\sum_{i=1}^{m} \mathcal{L}(f, x_i, y_i) + \lambda \Omega((W_\ell)_{\ell=1}^L),
\]

for example

\[
\Omega((W_\ell)_{\ell=1}^L) = \sum_{l,i,j} |(W_\ell)_{i,j}|^p.
\]

Gradient update has to be augmented by

\[
\lambda \cdot \frac{\partial}{\partial (W_\ell)_{i,j}} \Omega((W_\ell)_{\ell=1}^L) = \lambda \cdot p \cdot |(W_\ell)_{i,j}|^{p-1} \cdot \text{sgn}((W_\ell)_{i,j})
\]
Since

$$\lim_{p \to 0} \sum_{l,i,j} |(W_\ell)_{i,j}|^p = \#\text{nonzero weights},$$

regularization with $p \leq 1$ promotes sparse connectivity (and hence small memory requirements)!
Since \[
\lim_{p \to 0} \sum_{l,i,j} |(W_\ell)_{i,j}|^p = \# \text{nonzero weights},
\]
regularization with \( p \leq 1 \) promotes sparse connectivity (and hence small memory requirements)!
Dropout

💡 During each feedforward/backprop step drop nodes with probability $p$. After training, multiply all weights with $p$. The final output is "average" over many sparse network models.
During each feedforward/backprop step drop nodes with probability $p$. Final output is “average” over many sparse network models.
During each feedforward/backprop step drop nodes with probability $p$.

After training, multiply all weights with $p$. 
Dropout

- During each feedforward/back-prop step drop nodes with probability $p$.

After training, multiply all weights with $p$.

Final output is “average” over many sparse network models.
Use invariances in dataset to generate more data!
Dataset Augmentation

Use invariances in dataset to generate more data!
Dataset Augmentation

Use invariances in dataset to generate more data!
Use invariances in dataset to generate more data!
**Dataset Augmentation**

Use invariances in dataset to generate more data!

Sometimes also noise is added to the weights to favour ‘robust’ stationary points.
The Vanishing Gradient Problem

If \( \sigma(x) = \frac{1}{1 + e^{-x}} \), it holds that
\[
|\sigma'(x)| \leq 2 \cdot e^{-|x|},
\]
and thus
\[
|\frac{\partial \Phi(x)}{\partial b_1}| \leq 2 \cdot L \prod_{\ell=2}^{L-1} |w_{\ell}| \cdot e^{-\sum_{\ell=1}^{L-1} |z_{\ell}|}.
\]
Bottom layers will learn much slower than top layers and not contribute to learning.

Is depth a nuisance!
The Vanishing Gradient Problem

Figure: “Extremely Deep” Network

If $\sigma(x) = \frac{1}{1 + e^{-x}}$, it holds that $|\sigma'(x)| \leq 2 \cdot e^{-|x|}$, and thus $|\frac{\partial \Phi(x)}{\partial b_1}| \leq 2 \cdot L \prod_{\ell=2}^{L-1} |w_{\ell}| \cdot e^{-\sum_{\ell=1}^{L-1} |z_{\ell}|}$. Bottom layers will learn much slower than top layers and not contribute to learning. Is depth a nuisance!
The Vanishing Gradient Problem

\[ \Phi(x) = w_5 \sigma(w_4 \sigma(w_3 \sigma(w_2 \sigma(w_1 x + b_1) + b_2) + b_3) + b_4) + b_5 \]

**Figure:** “Extremely Deep” Network

If \( \sigma(x) = \frac{1}{1 + e^{-x}} \), it holds that \( |\sigma'(x)| \leq 2 \cdot e^{-|x|} \), and thus

\[ |\frac{\partial \Phi(x)}{\partial b_1}| \leq 2 \cdot L \prod_{\ell=2}^{L-1} |w_{\ell}| \cdot e^{-\sum_{\ell=1}^{L-1} |z_{\ell}|} \]

bottom layers will learn much slower than top layers and not contribute to learning. Is depth a nuisance!
The Vanishing Gradient Problem

Figure: “Extremely Deep” Network

\[ \Phi(x) = w_5 \sigma(w_4 \sigma(w_3 \sigma(w_2 \sigma(w_1 x + b_1) + b_2) + b_3) + b_4) + b_5 \]

\[ \frac{\partial \Phi(x)}{\partial b_1} = \prod_{\ell=2}^{L} w_\ell \cdot \prod_{\ell=1}^{L-1} \sigma'(z_\ell) \]

If \( \sigma(x) = \frac{1}{1+e^{-x}}, \) it holds that \( |\sigma'(x)| \leq 2e^{-|x|}, \) and thus

\[ |\frac{\partial \Phi(x)}{\partial b_1}| \leq 2 \cdot L \prod_{\ell=2}^{L-1} |w_\ell| \cdot e^{-\sum_{\ell=1}^{L-1} |z_\ell|} \]

Bottom layers will learn much slower than top layers and not contribute to learning.

Is depth a nuisance?
The Vanishing Gradient Problem

Figure: “Extremely Deep” Network

\[ \Phi(x) = w_5 \sigma(w_4 \sigma(w_3 \sigma(w_2 \sigma(w_1 x + b_1) + b_2) + b_3) + b_4) + b_5 \]

\[ \frac{\partial \Phi(x)}{\partial b_1} = \prod_{\ell=2}^{L} w_\ell \cdot \prod_{\ell=1}^{L-1} \sigma'(z_\ell) \]

If \( \sigma(x) = \frac{1}{1+e^{-x}} \), it holds that \( |\sigma'(x)| \leq 2 \cdot e^{-|x|} \), and thus

\[ |\frac{\partial \Phi(x)}{\partial b_1}| \leq 2 \cdot \prod_{\ell=2}^{L} |w_\ell| \cdot e^{-\sum_{\ell=1}^{L-1} |z_\ell|} \]
The Vanishing Gradient Problem

Figure: “Extremely Deep” Network

\[ \Phi(x) = w_5 \sigma(w_4 \sigma(w_3 \sigma(w_2 \sigma(w_1 x + b_1) + b_2) + b_3) + b_4) + b_5 \]

\[ \frac{\partial \Phi(x)}{\partial b_1} = \prod_{\ell=2}^{L} w_\ell \cdot \prod_{\ell=1}^{L-1} \sigma'(z_\ell) \]

If \( \sigma(x) = \frac{1}{1+e^{-x}} \), it holds that \( |\sigma'(x)| \leq 2 \cdot e^{-|x|} \), and thus

\[ |\frac{\partial \Phi(x)}{\partial b_1}| \leq 2 \cdot \prod_{\ell=2}^{L} |w_\ell| \cdot e^{-\sum_{\ell=1}^{L-1} |z_\ell|} \]

😢 bottom layers will learn much slower than top layers and not contribute to learning.
The Vanishing Gradient Problem

\[ \Phi(x) = w_5 \sigma(w_4 \sigma(w_3 \sigma(w_2 \sigma(w_1 x + b_1) + b_2) + b_3) + b_4) + b_5 \]

\[ \frac{\partial \Phi(x)}{\partial b_1} = \prod_{\ell=2}^{L} w_{\ell} \cdot \prod_{\ell=1}^{L-1} \sigma'(z_{\ell}) \]

If \( \sigma(x) = \frac{1}{1+e^{-x}} \), it holds that \( |\sigma'(x)| \leq 2 \cdot e^{-|x|} \), and thus

\[ |\frac{\partial \Phi(x)}{\partial b_1}| \leq 2 \cdot \prod_{\ell=2}^{L} |w_{\ell}| \cdot e^{-\sum_{\ell=1}^{L-1} |z_{\ell}|} \]

🤔 bottom layers will learn *much* slower than top layers and not contribute to learning. Is depth a nuisance!?
Dealing with the Vanishing Gradient Problem

Use activation function with 'large' gradient.

ReLU

The Rectified Linear Unit is defined as

\[
\text{ReLU}(x) := \begin{cases} 
  x & \text{if } x > 0 \\
  0 & \text{else}
\end{cases}
\]
Dealing with the Vanishing Gradient Problem

💡 Use activation function with ‘large’ gradient.
Dealing with the Vanishing Gradient Problem

Use activation function with ‘large’ gradient.

ReLU

The Rectified Linear Unit is defined as

\[
\text{ReLU}(x) := \begin{cases} 
  x & x > 0 \\
  0 & \text{else}
\end{cases}
\]
7 Convolutional Neural Networks
Is there a cat in this image?
Is there a cat in this image?
Suppose we have a 'cat-filter'

\[
\begin{pmatrix}
  x_{11} & x_{12} & x_{13} \\
  x_{21} & x_{22} & x_{23} \\
  x_{31} & x_{32} & x_{33}
\end{pmatrix}
\]

(Cat-Selection)

C-S Inequality

For any \( X = \begin{pmatrix} x_{11} & x_{12} & x_{13} \\
 x_{21} & x_{22} & x_{23} \\
 x_{31} & x_{32} & x_{33} \end{pmatrix} \) we have

\[
X \cdot W \leq \left( X \cdot X \right)^{1/2} \left( W \cdot W \right)^{1/2}
\]

with equality if and only if \( X \) is parallel to a cat.

perform cat-test on all \( 3 \times 3 \) image subpatches!
Suppose we have a ‘cat-filter’ $W$
Suppose we have a ‘cat-filter’ $W$
Suppose we have a ‘cat-filter’ $W$

\[
\begin{bmatrix}
  x_{11} & x_{12} & x_{13} \\
  x_{21} & x_{22} & x_{23} \\
  x_{31} & x_{32} & x_{33}
\end{bmatrix}
\]

we have

\[
X \cdot W \leq (X \cdot X)^{1/2} (W \cdot W)^{1/2}
\]

with equality if and only if $X$ is parallel to a cat.

perform cat-test on all $3 \times 3$ image subpatches!
Suppose we have a ‘cat-filter’ $W$

\[
X = \begin{pmatrix}
    x_{11} & x_{12} & x_{13} \\
    x_{21} & x_{22} & x_{23} \\
    x_{31} & x_{32} & x_{33}
\end{pmatrix}
\]

we have

\[
X \cdot W \leq (X \cdot X)^{1/2}(W \cdot W)^{1/2}
\]

with equality if and only if $X$ is parallel to a cat.
Suppose we have a ‘cat-filter’ $W$.

For any $X = \begin{pmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{pmatrix}$ we have

$$X \cdot W \leq (X \cdot X)^{1/2}(W \cdot W)^{1/2}$$

with equality if and only if $X$ is parallel to a cat.
Suppose we have a ‘cat-filter’ \( W \)

For any

\[
X = \begin{pmatrix}
x_{11} & x_{12} & x_{13} \\
x_{21} & x_{22} & x_{23} \\
x_{31} & x_{32} & x_{33}
\end{pmatrix}
\]

we have

\[
X \cdot W \leq (X \cdot X)^{1/2}(W \cdot W)^{1/2}
\]

with equality if and only if \( X \) is parallel to a cat.

perform cat-test on all \( 3 \times 3 \) image subpatches!
Convolution

Definition

Suppose that $X, Y \in \mathbb{R}^{n \times n}$. Then $Z = X * Y \in \mathbb{R}^{n \times n}$ is defined as

$$Z[i, j] = \sum_{k,l=0}^{n-1} X[i - k, j - l]Y[k, l],$$

where periodization or zero-padding of $X, Y$ is used if $i - k$ or $j - l$ is not in $\{0, \ldots, n - 1\}$. 

Efficient computation possible via FFT (or directly if $X$ or $Y$ are sparse)!
Convolution

Definition

Suppose that $X, Y \in \mathbb{R}^{n \times n}$. Then $Z = X \ast Y \in \mathbb{R}^{n \times n}$ is defined as

$$Z[i, j] = \sum_{k, l=0}^{n-1} X[i - k, j - l]Y[k, l],$$

where periodization or zero-padding of $X, Y$ is used if $i - k$ or $j - l$ is not in $\{0, \ldots, n - 1\}$.

Efficient computation possible via FFT (or directly if $X$ or $Y$ are sparse)!
Example: Detecting Vertical Edges

Given ‘vertical-edge-detection-filter’ $W = \begin{pmatrix} -1 & 2 & -1 \\ -1 & 2 & -1 \\ -1 & 2 & -1 \end{pmatrix}$
Example: Detecting Vertical Edges

Given ‘vertical-edge-detection-filter’ \( W = \begin{pmatrix} -1 & 2 & -1 \\ -1 & 2 & -1 \\ -1 & 2 & -1 \end{pmatrix} \)
Example: Detecting Vertical Edges
Example: Detecting Vertical Edges
Example: Detecting Vertical Edges
Example: Detecting Vertical Edges
Introducing Convolutional Nodes

- A convolutional node accepts as input a stack of images, e.g. $X \in \mathbb{R}^{n_1 \times n_2 \times S}$. 
Introducing Convolutional Nodes

- A convolutional node accepts as input a stack of images, e.g. $X \in \mathbb{R}^{n_1 \times n_2 \times S}$. 

![Diagram showing red, green, and blue color channels](image)
Introducing Convolutional Nodes

A convolutional node accepts as input a stack of images, e.g. $X \in \mathbb{R}^{n_1 \times n_2 \times S}$.

Given a filter $W \in \mathbb{R}^{F \times F \times S}$, where $F$ is the spatial extent and a bias $b \in \mathbb{R}$, it computes a matrix

$$Z = W \ast_{12} X := \sum_{i=1}^{S} X[:, :, i] \ast W[:, :, i] + b.$$
Introducing Convolutional Layers

- A convolutional node accepts as input a stack of images, e.g. $X \in \mathbb{R}^{n_1 \times n_2 \times S}$.

- Given a filter $W \in \mathbb{R}^{F \times F \times S}$, where $F$ is the *spatial extent* and a *bias* $b \in \mathbb{R}$, it computes a matrix

$$Z = W *_{12} X := \sum_{i=1}^{S} X[:, :, i] * W[:, :, i] + b.$$  

- A convolutional layer consists of $K$ convolutional nodes $((W_i, b_i))_{i=1}^{K} \subset \mathbb{R}^{F \times F \times S} \times \mathbb{R}$ and produces as output a stack $Z \in \mathbb{R}^{n_1 \times n_2 \times K}$ via

$$Z[:, :, i] = W_i *_{12} X + b_i.$$
Introducing Convolutional Layers

- A convolutional node accepts as input a stack of images, e.g. $X \in \mathbb{R}^{n_1 \times n_2 \times S}$.
- Given a filter $W \in \mathbb{R}^{F \times F \times S}$, where $F$ is the spatial extent and a bias $b \in \mathbb{R}$, it computes a matrix
  \[ Z = W *_{12} X := \sum_{i=1}^{S} X[:, :, i] * W[:, :, i] + b. \]

- A convolutional layer consists of $K$ convolutional nodes $((W_i, b_i))_{i=1}^{K} \subset \mathbb{R}^{F \times F \times S} \times \mathbb{R}$ and produces as output a stack $Z \in \mathbb{R}^{n_1 \times n_2 \times K}$ via
  \[ Z[:, :, i] = W_i *_{12} X + b_i. \]

💡 A convolutional layer can be written as a conventional neural network layer!
The activation layer is defined in the same way as before, e.g., $Z \in \mathbb{R}^{n_1 \times n_2 \times K}$ is mapped to

$$A = \text{ReLU}(Z)$$

where ReLU is applied component-wise.
Pooling Layers

💡 Reduce dimensionality after filtering.
Pooling Layers

💡 Reduce dimensionality after filtering.

**Definition**

A *pooling operator* $\mathbf{R}$ acts layer-wise on a tensor $X \in \mathbb{R}^{n_1 \times n_2 \times S}$ to result in a tensor $\mathbf{R}(X) \in \mathbb{R}^{m_1 \times m_2 \times S}$, where $m_1 < n_1$ and $m_2 < n_2$. 
Pooling Layers

💡 Reduce dimensionality after filtering.

**Definition**

A *pooling operator* $\mathbf{R}$ acts layer-wise on a tensor $X \in \mathbb{R}^{n_1 \times n_2 \times S}$ to result in a tensor $\mathbf{R}(X) \in \mathbb{R}^{m_1 \times m_2 \times S}$, where $m_1 < n_1$ and $m_2 < n_2$.

---

Downsampling
Pooling Layers

Reduce dimensionality after filtering.

**Definition**

A *pooling operator* $\mathbf{R}$ acts layer-wise on a tensor $X \in \mathbb{R}^{n_1 \times n_2 \times S}$ to result in a tensor $\mathbf{R}(X) \in \mathbb{R}^{m_1 \times m_2 \times S}$, where $m_1 < n_1$ and $m_2 < n_2$. 

**Downsampling**

**Max-pooling**
### Convolutional Neural Networks (CNNs)

**Definition**

A CNN with $L$ layers consists of $L$ iterative applications of a convolutional layer, followed by an activation layer, (possibly) followed by a pooling layer.
Convolutional Neural Networks (CNNs)

**Definition**
A CNN with $L$ layers consists of $L$ iterative applications of a convolutional layer, followed by an activation layer, (possibly) followed by a pooling layer.

💡 Typical architectures consist of a CNN (as a *feature extractor*), followed by a fully connected NN (as a *classifier*).
Convolutional Neural Networks (CNNs)

Definition
A CNN with $L$ layers consists of $L$ iterative applications of a convolutional layer, followed by an activation layer, (possibly) followed by a pooling layer.

💡 Typical architectures consist of a CNN (as a feature extractor), followed by a fully connected NN (as a classifier)

Figure: LeNet (1998, LeCun et al): the first successful CNN architecture, used for reading handwritten digits
Feature Extractor vs. Classifier
Feature Extractor vs. Classifier
Feature Extractor vs. Classifier

D. Trump

B. Sanders

A. Merkel

B. Johnson
```python
x_image = tf.reshape(x, [-1, 28, 28, 1])

# First convolutional layer - maps one grayscale image to 32 feature maps.
W_conv1 = weight_variable([5, 5, 1, 32])
b_conv1 = bias_variable([32])
h_conv1 = tf.nn.relu(conv2d(x_image, W_conv1) + b_conv1)

# Pooling layer - downsamples by 2X.
h_pool1 = max_pool_2x2(h_conv1)

# Second convolutional layer -- maps 32 feature maps to 64.
W_conv2 = weight_variable([5, 5, 32, 64])
b_conv2 = bias_variable([64])
h_conv2 = tf.nn.relu(conv2d(h_pool1, W_conv2) + b_conv2)

# Second pooling layer.
h_pool2 = max_pool_2x2(h_conv2)

# Fully connected layer 1 -- after 2 round of downsampling, our 28x28 image
W_fc1 = weight_variable([7 * 7 * 64, 1024])
b_fc1 = bias_variable([1024])
h_pool2_flat = tf.reshape(h_pool2, [-1, 7*7*64])
h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)

# Map the 1024 features to 10 classes, one for each digit
W_fc2 = weight_variable([1024, 10])
b_fc2 = bias_variable([10])
y_conv = tf.matmul(h_fc1, W_fc2) + b_fc2
```
x_image = tf.reshape(x, [-1, 28, 28, 1])
# First convolutional layer - maps one grayscale image to 32 feature maps.
W_conv1 = weight_variable([5, 5, 1, 32])
b_conv1 = bias_variable([32])
h_conv1 = tf.nn.relu(conv2d(x_image, W_conv1) + b_conv1)
# Pooling layer - downsamples by 2X.
h_pool1 = max_pool_2x2(h_conv1)
# Second convolutional layer -- maps 32 feature maps to 64.
W_conv2 = weight_variable([5, 5, 32, 64])
b_conv2 = bias_variable([64])
h_conv2 = tf.nn.relu(conv2d(h_pool1, W_conv2) + b_conv2)
# Second pooling layer.
h_pool2 = max_pool_2x2(h_conv2)
# Fully connected layer 1 -- after flatten:
W_fc1 = weight_variable([7 * 7 * 64, 1024])
b_fc1 = bias_variable([1024])
h_pool2_flat = tf.reshape(h_pool2, [-1, 7*7*64])
h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)
# Map the 1024 features to 10 classes, one for each digit
W_fc2 = weight_variable([1024, 10])
b_fc2 = bias_variable([10])
y_conv = tf.matmul(h_fc1, W_fc2) + b_fc2

```python
x_image = tf.reshape(x, [-1, 28, 28, 1])

# First Convolutional Layer - maps one grayscale image to 32 feature maps.
W_conv1 = tf.get_variable('W_conv1',
                           initializer=tf.truncated_normal([5, 5, 1, 32]))

b_conv1 = tf.get_variable('b_conv1',
                           initializer=tf.constant(0.1, shape=[32]))

h_conv1 = tf.nn.relu(tf.nn.bias_add(tf.nn.conv2d(x_image, W_conv1) + b_conv1)
# Pooling Layer - reduces size of the image maps by 2X.

h_pool1 = tf.nn.max_pool(h_conv1, ksize=[1, 2, 2, 1],
                        strides=[1, 2, 2, 1],
                        padding='SAME')

# Second Convolutional Layer -- maps 32 feature maps to 64.
W_conv2 = tf.get_variable('W_conv2',
                           initializer=tf.truncated_normal([5, 5, 32, 64]))

b_conv2 = tf.get_variable('b_conv2',
                           initializer=tf.constant(0.1, shape=[64]))

h_conv2 = tf.nn.relu(tf.nn.bias_add(tf.nn.conv2d(h_pool1, W_conv2) + b_conv2)
# Full-Connected Layer -- after flatten, a 784 x 1024 layer.
W_fc1 = tf.get_variable('W_fc1',
                        initializer=tf.truncated_normal([7 * 7 * 64, 1024]))

b_fc1 = tf.get_variable('b_fc1',
                        initializer=tf.constant(0.1, shape=[1024]))

h_fc1 = tf.nn.relu(tf.matmul(h_pool2, [-1, 7*7*64])

# Maps to 10 classes, one for each digit.
W_fc2 = tf.get_variable('W_fc2',
                        initializer=tf.truncated_normal([1024, 10]))

b_fc2 = tf.get_variable('b_fc2',
                        initializer=tf.constant(0.1, shape=[10]))

y_conv = tf.nn.softmax(tf.matmul(h_fc1, W_fc2) + b_fc2)
```

Test accuracy: 0.9934
8 What I didn’t tell you
Data structures & algorithms for efficient deep learning (computational graphs, automatic differentiation, adaptive learning rate, hardware, ...)

Things to do besides regression or classification

Finetuning (choice of activation function, choice of loss function, ...)

More sophisticated training procedures for feature extractor layers (Autoencoder, Restricted Boltzmann Machines, ...)

Recurrent Neural Networks
1 Data structures & algorithms for efficient deep learning
(computational graphs, automatic differentiation, adaptive learning rate, hardware, ...)

2 Things to do besides regression or classification

3 Finetuning (choice of activation function, choice of loss function, ...)

4 More sophisticated training procedures for feature extractor layers (Autoencoder, Restricted Boltzmann Machines, ...)

5 Recurrent Neural Networks
1. Data structures & algorithms for efficient deep learning (computational graphs, automatic differentiation, adaptive learning rate, hardware, ...)

2. Things to do besides regression or classification
1. Data structures & algorithms for efficient deep learning (computational graphs, automatic differentiation, adaptive learning rate, hardware, ...)

2. Things to do besides regression or classification

3. Finetuning (choice of activation function, choice of loss function, ...)

4. More sophisticated training procedures for feature extractor layers (Autoencoder, Restricted Boltzmann Machines, ...)

5. Recurrent Neural Networks
1. Data structures & algorithms for efficient deep learning (computational graphs, automatic differentiation, adaptive learning rate, hardware, ...)

2. Things to do besides regression or classification

3. Finetuning (choice of activation function, choice of loss function, ...)

4. More sophisticated training procedures for feature extractor layers (Autoencoder, Restricted Boltzmann Machines, ...)

5. Recurrent Neural Networks
1. Data structures & algorithms for efficient deep learning (computational graphs, automatic differentiation, adaptive learning rate, hardware, ...)

2. Things to do besides regression or classification

3. Finetuning (choice of activation function, choice of loss function, ...)

4. More sophisticated training procedures for feature extractor layers (Autoencoder, Restricted Boltzmann Machines, ...)

5. Recurrent Neural Networks
Questions?