Introduction to Machine Learning (67577)
Lecture 10

Shai Shalev-Shwartz

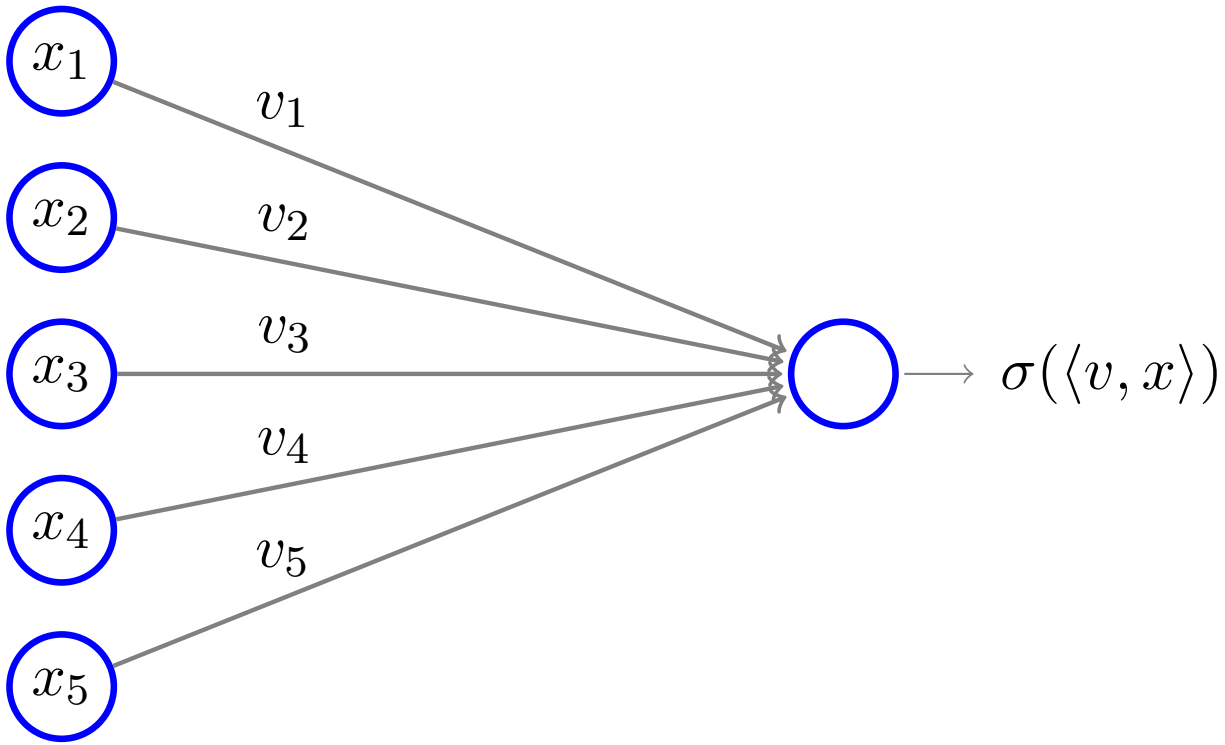
School of CS and Engineering,
The Hebrew University of Jerusalem

Neural Networks

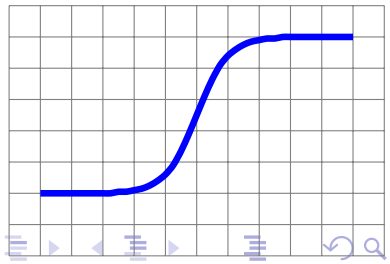
Outline

A Single Artificial Neuron

- A single neuron is a function of the form $\mathbf{x} \mapsto \sigma(\langle \mathbf{v}, \mathbf{x} \rangle)$, where $\sigma : \mathbb{R} \to \mathbb{R}$ is called the activation function of the neuron



- E.g., σ is a sigmoidal function



Neural Networks

- A neural network is obtained by connecting many neurons together

Neural Networks

- A neural network is obtained by connecting many neurons together
- We focus on feedforward networks, formally defined by a directed acyclic graph $G = (V, E)$

Neural Networks

- A neural network is obtained by connecting many neurons together
- We focus on feedforward networks, formally defined by a directed acyclic graph $G = (V, E)$
- Input nodes: nodes with no incoming edges

Neural Networks

- A neural network is obtained by connecting many neurons together
- We focus on feedforward networks, formally defined by a directed acyclic graph $G = (V, E)$
- Input nodes: nodes with no incoming edges
- Output nodes: nodes without out going edges

Neural Networks

- A neural network is obtained by connecting many neurons together
- We focus on feedforward networks, formally defined by a directed acyclic graph $G = (V, E)$
- Input nodes: nodes with no incoming edges
- Output nodes: nodes without out going edges
- weights: $w : E \to \mathbb{R}$

Neural Networks

- A neural network is obtained by connecting many neurons together
- We focus on feedforward networks, formally defined by a directed acyclic graph $G = (V, E)$
- Input nodes: nodes with no incoming edges
- Output nodes: nodes without out going edges
- weights: $w : E \to \mathbb{R}$
- Calculation using breadth-first-search (BFS), where each neuron (node) receives as input:

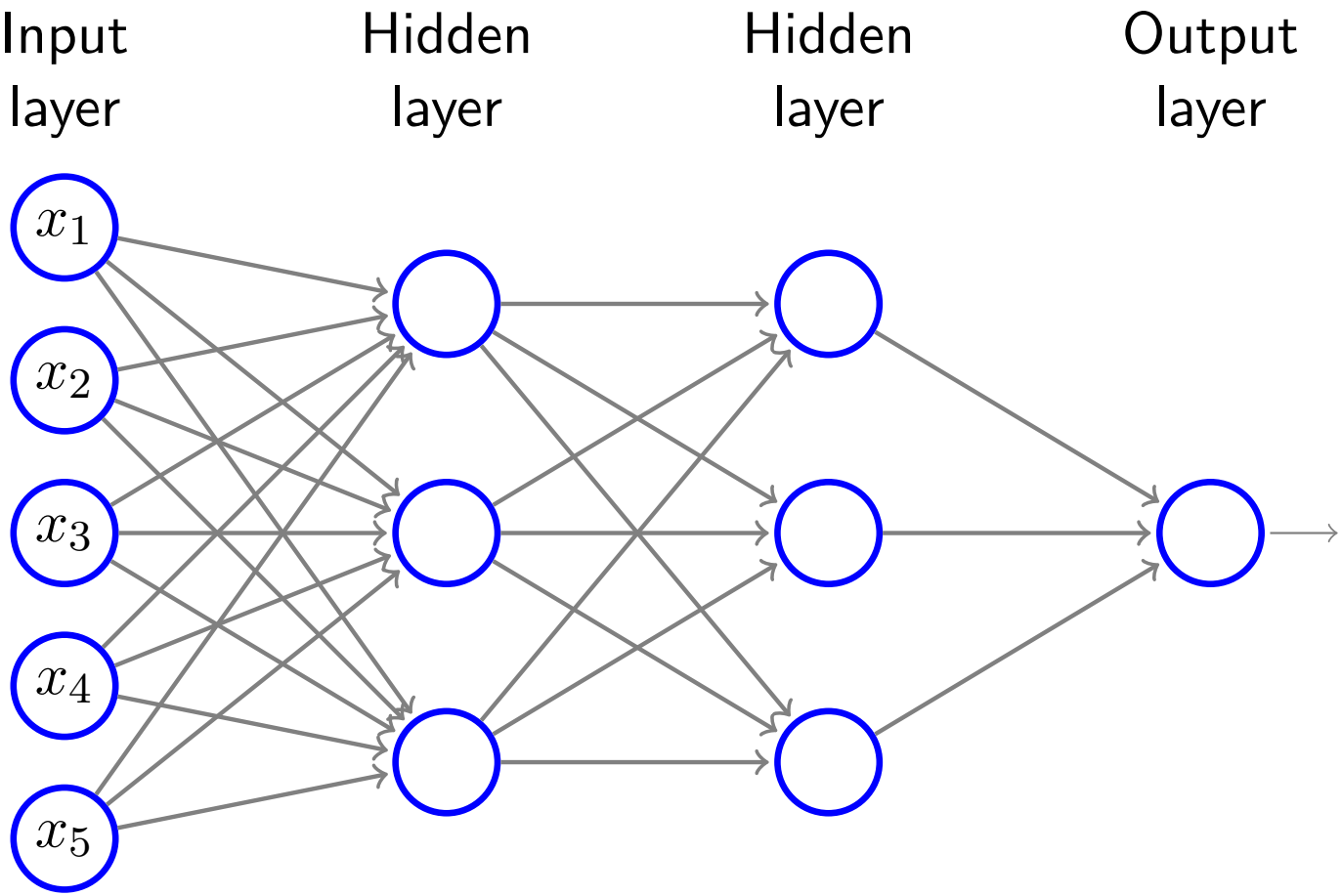
$$a[v] = \sum_{u \to v \in E} w[u \to v] o[u]$$

and output

$$o[v] = \sigma(a[v])$$

Multilayer Neural Networks

- Neurons are organized in layers: $V = \cup_{t=0}^{T} V_t$, and edges are only between adjacent layers
- Example of a multilayer neural network of depth 3 and size 6



Neural Networks as a Hypothesis Class

- Given a neural network (V, E, σ, w), we obtain a hypothesis $h_{V,E,\sigma,w} : \mathbb{R}^{|V_0|-1} \to \mathbb{R}^{|V_T|}$

Neural Networks as a Hypothesis Class

- Given a neural network (V, E, σ, w), we obtain a hypothesis $h_{V,E,\sigma,w} : \mathbb{R}^{|V_0|-1} \to \mathbb{R}^{|V_T|}$
- We refer to (V, E, σ) as the architecture, and it defines a hypothesis class by

$$\mathcal{H}_{V,E,\sigma} = \{h_{V,E,\sigma,w} : w \text{ is a mapping from } E \text{ to } \mathbb{R}\} .$$

Neural Networks as a Hypothesis Class

- Given a neural network (V, E, σ, w), we obtain a hypothesis $h_{V,E,\sigma,w} : \mathbb{R}^{|V_0|-1} \to \mathbb{R}^{|V_T|}$
- We refer to (V, E, σ) as the architecture, and it defines a hypothesis class by

$$\mathcal{H}_{V,E,\sigma} = \{h_{V,E,\sigma,w} : w \text{ is a mapping from } E \text{ to } \mathbb{R}\} .$$

- The architecture is our "Prior knowledge" and the learning task is to find the weight function w

Neural Networks as a Hypothesis Class

- Given a neural network (V, E, σ, w), we obtain a hypothesis $h_{V,E,\sigma,w} : \mathbb{R}^{|V_0|-1} \to \mathbb{R}^{|V_T|}$
- We refer to (V, E, σ) as the architecture, and it defines a hypothesis class by

$$\mathcal{H}_{V,E,\sigma} = \{h_{V,E,\sigma,w} : w \text{ is a mapping from } E \text{ to } \mathbb{R}\} .$$

- The architecture is our "Prior knowledge" and the learning task is to find the weight function w
- We can now study
  estimation error (sample complexity)
  approximation error (expressivenss)
  optimization error (computational complexity)

Outline

Sample Complexity

- Theorem: The VC dimension of $\mathcal{H}_{V,E,\text{sign}}$ is $O(|E| \log(|E|))$.

Sample Complexity

- Theorem: The VC dimension of $\mathcal{H}_{V,E,\text{sign}}$ is $O(|E|\log(|E|))$.
- Theorem: The VC dimension of $\mathcal{H}_{V,E,\sigma}$, for σ being the sigmoidal function, is $\Omega(|E|^2)$.

Sample Complexity

- Theorem: The VC dimension of $\mathcal{H}_{V,E,\text{sign}}$ is $O(|E|\log(|E|))$.
- Theorem: The VC dimension of $\mathcal{H}_{V,E,\sigma}$, for σ being the sigmoidal function, is $\Omega(|E|^2)$.
- Representation trick: In practice, we only care about networks where each weight is represented using $O(1)$ bits, and therefore the VC dimension of such networks is $O(|E|)$, no matter what σ is

Sample Complexity

- Theorem: The VC dimension of $\mathcal{H}_{V,E,\text{sign}}$ is $O(|E|\log(|E|))$.
- Theorem: The VC dimension of $\mathcal{H}_{V,E,\sigma}$, for σ being the sigmoidal function, is $\Omega(|E|^2)$.
- Representation trick: In practice, we only care about networks where each weight is represented using $O(1)$ bits, and therefore the VC dimension of such networks is $O(|E|)$, no matter what σ is
- We can further decrease the sample complexity by many kinds of regularization functions (this is left for an advanced course)

Outline

What can we express with neural networks ?

- For simplicity, lets focus on boolean inputs and sign activation functions.

What can we express with neural networks ?

- For simplicity, lets focus on boolean inputs and sign activation functions.
- What type of functions from $\{\pm 1\}^n$ to $\{\pm 1\}$ can be implemented by $\mathcal{H}_{V,E,\text{sign}}$?

What can we express with neural networks ?

- For simplicity, lets focus on boolean inputs and sign activation functions.
- What type of functions from $\{\pm 1\}^n$ to $\{\pm 1\}$ can be implemented by $\mathcal{H}_{V,E,\text{sign}}$?
- Theorem: For every n, there exists a graph (V, E) of depth 2, such that $\mathcal{H}_{V,E,\text{sign}}$ contains all functions from $\{\pm 1\}^n$ to $\{\pm 1\}$

What can we express with neural networks ?

- For simplicity, lets focus on boolean inputs and sign activation functions.
- What type of functions from $\{\pm 1\}^n$ to $\{\pm 1\}$ can be implemented by $\mathcal{H}_{V,E,\text{sign}}$?
- Theorem: For every n, there exists a graph (V, E) of depth 2, such that $\mathcal{H}_{V,E,\text{sign}}$ contains all functions from $\{\pm 1\}^n$ to $\{\pm 1\}$
- Theorem: For every n, let $s(n)$ be the minimal integer such that there exists a graph (V, E) with $|V| = s(n)$ such that the hypothesis class $\mathcal{H}_{V,E,\text{sign}}$ contains all the functions from $\{0, 1\}^n$ to $\{0, 1\}$. Then, $s(n)$ is exponential in n.

What can we express with neural networks ?

- For simplicity, lets focus on boolean inputs and sign activation functions.
- What type of functions from $\{\pm 1\}^n$ to $\{\pm 1\}$ can be implemented by $\mathcal{H}_{V,E,\text{sign}}$?
- Theorem: For every n, there exists a graph (V, E) of depth 2, such that $\mathcal{H}_{V,E,\text{sign}}$ contains all functions from $\{\pm 1\}^n$ to $\{\pm 1\}$
- Theorem: For every n, let $s(n)$ be the minimal integer such that there exists a graph (V, E) with $|V| = s(n)$ such that the hypothesis class $\mathcal{H}_{V,E,\text{sign}}$ contains all the functions from $\{0, 1\}^n$ to $\{0, 1\}$. Then, $s(n)$ is exponential in n.
- What type of functions can be implemented by networks of small size ?

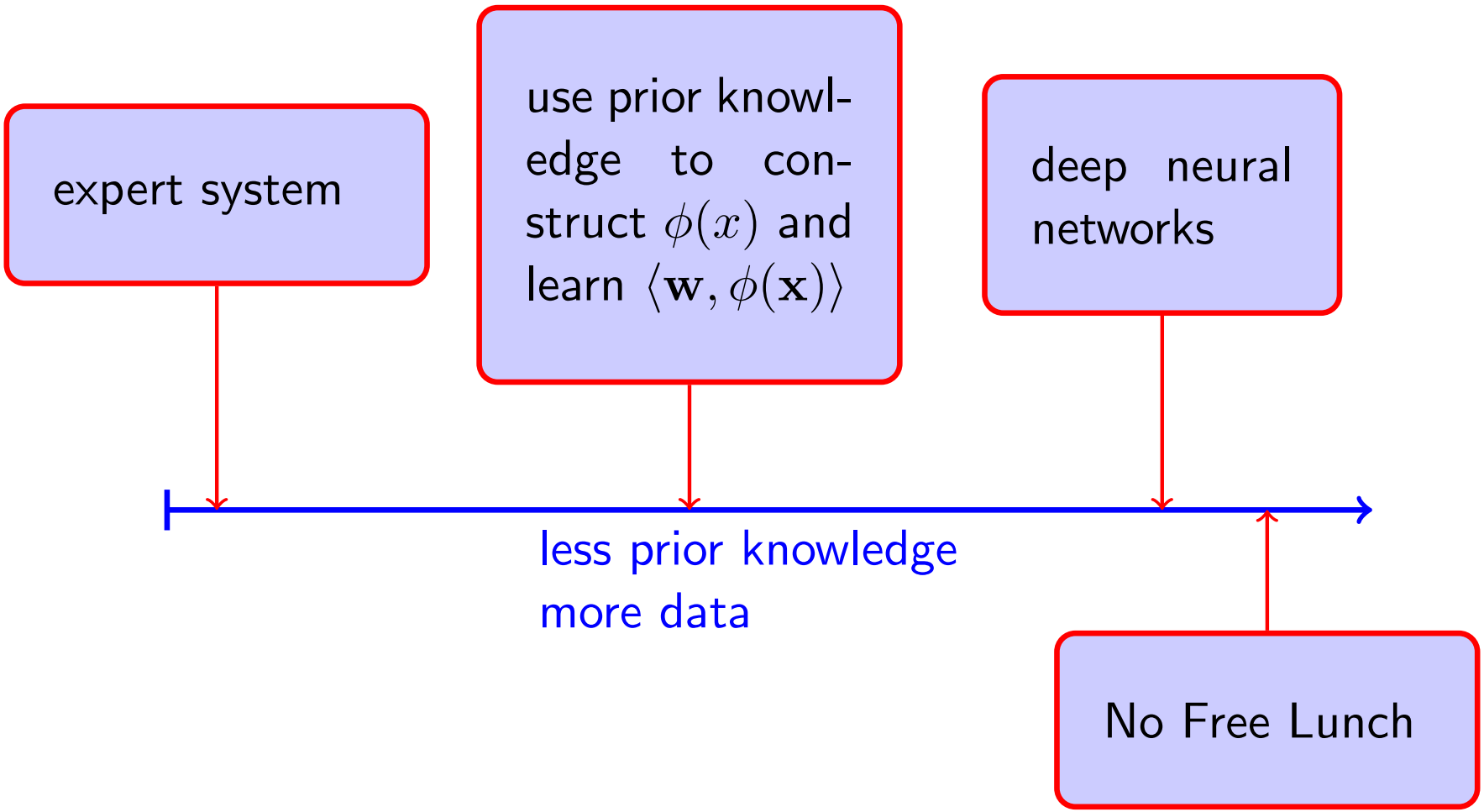
What can we express with neural networks ?

- Theorem: Let $T : \mathbb{N} \to \mathbb{N}$ and for every n, let $\mathcal{F}_n$ be the set of functions that can be implemented using a Turing machine using runtime of at most $T(n)$. Then, there exist constants $b, c \in \mathbb{R}_+$ such that for every n, there is a graph (V_n, E_n) of size at most $c\,T(n)^2 + b$ such that $\mathcal{H}_{V_n,E_n,\text{sign}}$ contains $\mathcal{F}_n$.

What can we express with neural networks ?

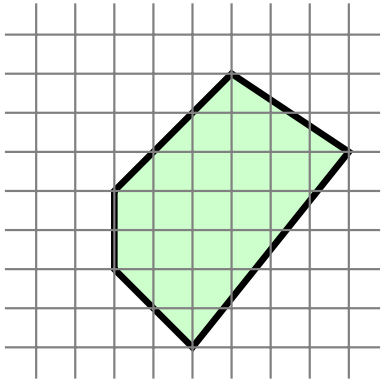
- Theorem: Let $T : \mathbb{N} \to \mathbb{N}$ and for every n, let $\mathcal{F}_n$ be the set of functions that can be implemented using a Turing machine using runtime of at most $T(n)$. Then, there exist constants $b, c \in \mathbb{R}_+$ such that for every n, there is a graph (V_n, E_n) of size at most $cT(n)^2 + b$ such that $\mathcal{H}_{V_n,E_n,\text{sign}}$ contains $\mathcal{F}_n$.
- Conclusion: A very weak notion of prior knowledge suffices — if we only care about functions that can be implemented in time $T(n)$, we can use neural networks of size $O(T(n)^2)$, and the sample complexity is also bounded by $O(T(n)^2)$!

The ultimate hypothesis class



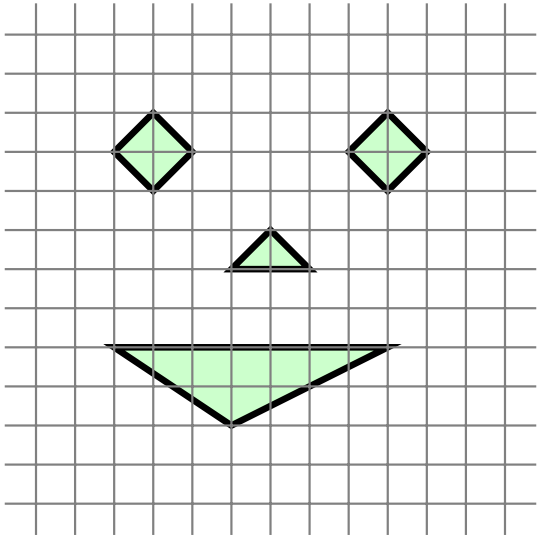
Geometric Intuition

- 2 layer networks can express intersection of halfspaces



Geometric Intuition

- 3 layer networks can express unions of intersection of halfspaces



Outline

Runtime of learning neural networks

ERM problem:

$$\mathrm{ERM}(S) = \operatorname*{argmin}_{h \in \mathcal{H}_{V,E,\sigma}} L_S(h) = \operatorname*{argmin}_{w} L_S(h_{V,E,\sigma,w})$$

- Theorem: It is NP hard to implement the ERM rule with respect to $\mathcal{H}_{V,E,\mathrm{sign}}$ even for networks with a single hidden layer that contain just 4 neurons in the hidden layer.

Runtime of learning neural networks

ERM problem:

$$\mathrm{ERM}(S) = \operatorname*{argmin}_{h \in \mathcal{H}_{V,E,\sigma}} L_S(h) = \operatorname*{argmin}_{w} L_S(h_{V,E,\sigma,w})$$

- Theorem: It is NP hard to implement the ERM rule with respect to $\mathcal{H}_{V,E,\mathrm{sign}}$ even for networks with a single hidden layer that contain just 4 neurons in the hidden layer.
- But, maybe ERM is hard but some improper algorithm works ?

Runtime of learning neural networks

ERM problem:

$$\mathrm{ERM}(S) = \operatorname*{argmin}_{h \in \mathcal{H}_{V,E,\sigma}} L_S(h) = \operatorname*{argmin}_{w} L_S(h_{V,E,\sigma,w})$$

- Theorem: It is NP hard to implement the ERM rule with respect to $\mathcal{H}_{V,E,\mathrm{sign}}$ even for networks with a single hidden layer that contain just 4 neurons in the hidden layer.
- But, maybe ERM is hard but some improper algorithm works ?
- Theorem: Under some average case complexity assumption, it is hard to learn neural networks of depth 2 and size $\omega(\log(d))$ even improperly

How to train neural network ?

- So, neural networks can form an excellent hypothesis class, but it is intractable to train it.

How to train neural network ?

- So, neural networks can form an excellent hypothesis class, but it is intractable to train it.
- How is this different than the class of all Python program that can be implemented in code length of b bits ?

How to train neural network ?

- So, neural networks can form an excellent hypothesis class, but it is intractable to train it.
- How is this different than the class of all Python program that can be implemented in code length of b bits ?
- Main technique: Stochastic Gradient Descent (SGD)

How to train neural network ?

- So, neural networks can form an excellent hypothesis class, but it is intractable to train it.
- How is this different than the class of all Python program that can be implemented in code length of b bits ?
- Main technique: Stochastic Gradient Descent (SGD)
- Not convex, no guarantees, can take a long time, but:

How to train neural network ?

- So, neural networks can form an excellent hypothesis class, but it is intractable to train it.
- How is this different than the class of all Python program that can be implemented in code length of b bits ?
- Main technique: Stochastic Gradient Descent (SGD)
- Not convex, no guarantees, can take a long time, but:
  - Often still works fine, finds a good solution

How to train neural network ?

- So, neural networks can form an excellent hypothesis class, but it is intractable to train it.
- How is this different than the class of all Python program that can be implemented in code length of b bits ?
- Main technique: Stochastic Gradient Descent (SGD)
- Not convex, no guarantees, can take a long time, but:
  - Often still works fine, finds a good solution
  - Easier than optimizing over Python programs ...

SGD for Neural Networks

Main skeleton:

- Random initialization: rule of thumb, $w[u \to v] \sim U[-c, c]$ where $c = \sqrt{3/|\{(u', v) \in E\}|}$

SGD for Neural Networks

Main skeleton:

- Random initialization: rule of thumb, $w[u \to v] \sim U[-c, c]$ where $c = \sqrt{3/|\{(u', v) \in E\}|}$
- Update step with Nesterov's momentum:

$$w_{t+1} = \mu_t w_t - \eta_t \tilde{\nabla} L_{\mathcal{D}}(\theta_t + \mu_t w_t)$$
$$\theta_{t+1} = \theta_t + w_{t+1}$$

where:
μ_t is momentum parameter (e.g. $\mu_t = 0.9$ for all t)
η_t is learning rate (e.g. $\eta_t = 0.01$ for all t)
$\tilde{\nabla} L_{\mathcal{D}}$ is an estimate of the gradient of $L_{\mathcal{D}}$ based on a small set of random examples (often called a "minibatch")

SGD for Neural Networks

Main skeleton:

- Random initialization: rule of thumb, $w[u \to v] \sim U[-c, c]$ where $c = \sqrt{3/|\{(u', v) \in E\}|}$
- Update step with Nesterov's momentum:

$$w_{t+1} = \mu_t w_t - \eta_t \tilde{\nabla} L_{\mathcal{D}}(\theta_t + \mu_t w_t)$$
$$\theta_{t+1} = \theta_t + w_{t+1}$$

  where:
  μ_t is momentum parameter (e.g. $\mu_t = 0.9$ for all t)
  η_t is learning rate (e.g. $\eta_t = 0.01$ for all t)
  $\tilde{\nabla} L_{\mathcal{D}}$ is an estimate of the gradient of $L_{\mathcal{D}}$ based on a small set of random examples (often called a "minibatch")

- It is left to show how to calculate the gradient

Back-Propagation

- The back-propagation algorithm is an efficient way to calculate $\nabla \ell(h_w(x), y)$ using the chain rule

Back-Propagation

- The back-propagation algorithm is an efficient way to calculate $\nabla \ell(h_w(x), y)$ using the chain rule
- Recall: the Jacobian of $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$ at $\mathbf{w} \in \mathbb{R}^n$, denoted $J_{\mathbf{w}}(\mathbf{f})$, is the $m \times n$ matrix whose i, j element is the partial derivative of $f_i : \mathbb{R}^n \to \mathbb{R}$ w.r.t. its j'th variable at $\mathbf{w}$. E.g.

Back-Propagation

- The back-propagation algorithm is an efficient way to calculate $\nabla \ell(h_w(x), y)$ using the chain rule
- Recall: the Jacobian of $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$ at $\mathbf{w} \in \mathbb{R}^n$, denoted $J_{\mathbf{w}}(\mathbf{f})$, is the $m \times n$ matrix whose i, j element is the partial derivative of $f_i : \mathbb{R}^n \to \mathbb{R}$ w.r.t. its j'th variable at $\mathbf{w}$. E.g.
  - If $\mathbf{f}(\mathbf{w}) = A\mathbf{w}$ then $J_{\mathbf{w}}(\mathbf{f}) = A$.

Back-Propagation

- The back-propagation algorithm is an efficient way to calculate $\nabla \ell(h_w(x), y)$ using the chain rule
- Recall: the Jacobian of $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$ at $\mathbf{w} \in \mathbb{R}^n$, denoted $J_{\mathbf{w}}(\mathbf{f})$, is the $m \times n$ matrix whose i, j element is the partial derivative of $f_i : \mathbb{R}^n \to \mathbb{R}$ w.r.t. its j'th variable at $\mathbf{w}$. E.g.
  - If $\mathbf{f}(\mathbf{w}) = A\mathbf{w}$ then $J_{\mathbf{w}}(\mathbf{f}) = A$.
  - If $\boldsymbol{\sigma} : \mathbb{R}^n \to \mathbb{R}^n$ is element-wise application of $\sigma : \mathbb{R} \to \mathbb{R}$ then $J_{\boldsymbol{\theta}}(\boldsymbol{\sigma}) = \mathrm{diag}((\sigma'(\theta_1), \ldots, \sigma'(\theta_n)))$.

Back-Propagation

- The back-propagation algorithm is an efficient way to calculate $\nabla \ell(h_w(x), y)$ using the chain rule
- Recall: the Jacobian of $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$ at $\mathbf{w} \in \mathbb{R}^n$, denoted $J_{\mathbf{w}}(\mathbf{f})$, is the $m \times n$ matrix whose i, j element is the partial derivative of $f_i : \mathbb{R}^n \to \mathbb{R}$ w.r.t. its j'th variable at $\mathbf{w}$. E.g.
  - If $\mathbf{f}(\mathbf{w}) = A\mathbf{w}$ then $J_{\mathbf{w}}(\mathbf{f}) = A$.
  - If $\boldsymbol{\sigma} : \mathbb{R}^n \to \mathbb{R}^n$ is element-wise application of $\sigma : \mathbb{R} \to \mathbb{R}$ then $J_{\boldsymbol{\theta}}(\boldsymbol{\sigma}) = \mathrm{diag}((\sigma'(\theta_1), \ldots, \sigma'(\theta_n)))$.
- Chain rule:

$$J_{\mathbf{w}}(\mathbf{f} \circ \mathbf{g}) = J_{g(\mathbf{w})}(\mathbf{f}) J_{\mathbf{w}}(\mathbf{g})$$

Back-Propagation

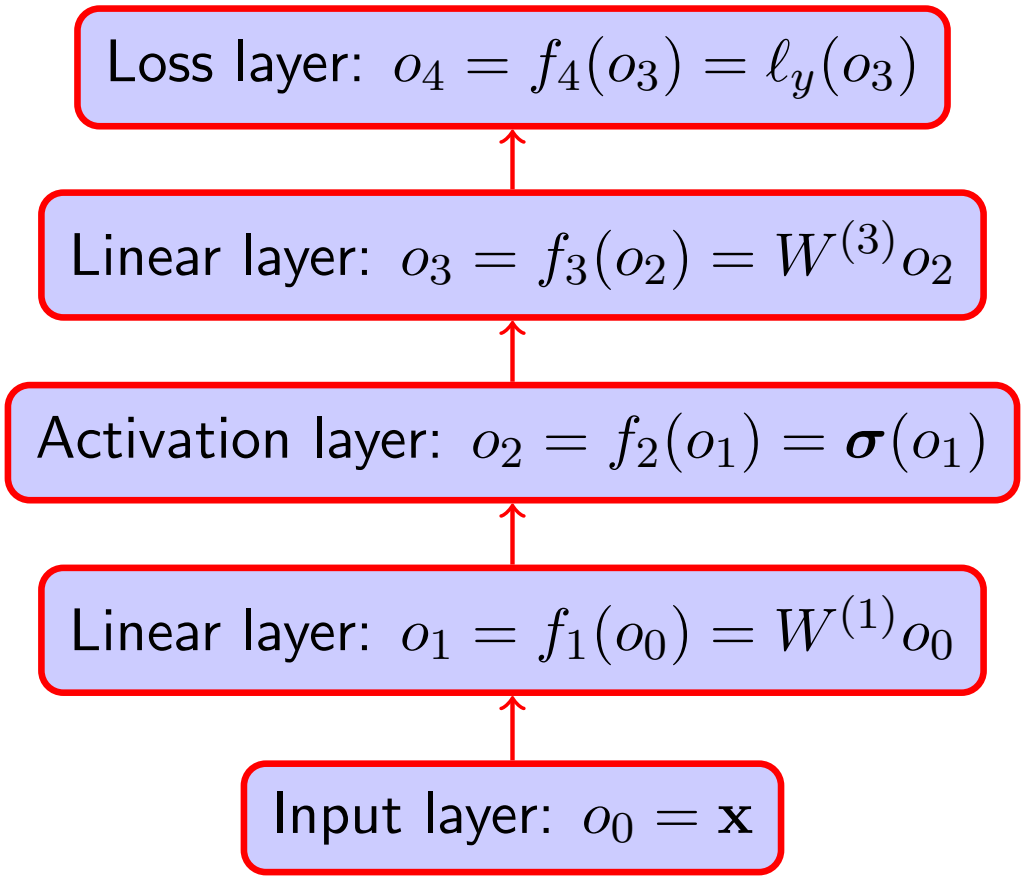
- The back-propagation algorithm is an efficient way to calculate $\nabla \ell(h_w(x), y)$ using the chain rule
- Recall: the Jacobian of $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$ at $\mathbf{w} \in \mathbb{R}^n$, denoted $J_{\mathbf{w}}(\mathbf{f})$, is the $m \times n$ matrix whose i, j element is the partial derivative of $f_i : \mathbb{R}^n \to \mathbb{R}$ w.r.t. its j'th variable at $\mathbf{w}$. E.g.
  - If $\mathbf{f}(\mathbf{w}) = A\mathbf{w}$ then $J_{\mathbf{w}}(\mathbf{f}) = A$.
  - If $\boldsymbol{\sigma} : \mathbb{R}^n \to \mathbb{R}^n$ is element-wise application of $\sigma : \mathbb{R} \to \mathbb{R}$ then $J_{\boldsymbol{\theta}}(\boldsymbol{\sigma}) = \mathrm{diag}((\sigma'(\theta_1), \ldots, \sigma'(\theta_n)))$.
- Chain rule:

$$J_{\mathbf{w}}(\mathbf{f} \circ \mathbf{g}) = J_{g(\mathbf{w})}(\mathbf{f}) J_{\mathbf{w}}(\mathbf{g})$$

- Let $\ell_y : \mathbb{R}^k \to \mathbb{R}$ be the loss function given predictions $\boldsymbol{\theta} \in \mathbb{R}^k$ and label y.

Back-Propagation

It's convenient to describe the network as a sequence of simple layer functions:



Back-Propagation

- Can write $\ell(h_{\mathbf{w}}, (\mathbf{x}, y)) = (f_{T+1} \circ \ldots \circ f_3 \circ f_2 \circ f_1)(\mathbf{x})$
- Denote $F_t = f_{T+1} \circ \ldots \circ f_{t+1}$ and $\delta_t = J_{o_t}(F_t)$, then

$$\begin{aligned}\delta_t &= J_{o_t}(F_t) = J_{o_t}(F_{t-1} \circ f_{t+1}) \\ &= J_{f_{t+1}(o_t)}(F_{t-1}) J_{o_t}(f_{t+1}) = J_{o_{t+1}}(F_{t-1}) J_{o_t}(f_{t+1}) \\ &= \delta_{t+1} J_{o_t}(f_{t+1})\end{aligned}$$

- Note that

$$J_{o_t}(f_{t+1}) = \begin{cases} W^{(t+1)} & \text{for linear layer} \\ \mathrm{diag}(\boldsymbol{\sigma}'(o_t)) & \text{for activation layer} \end{cases}$$

- Using the chain rule again we obtain

$$J_{W^{(t)}}(\ell(h_{\mathbf{w}}, (\mathbf{x}, y))) = \delta_t o_{t-1}^\top$$

Back-Propagation: Pseudo-code

Forward:

- set $o_0 = \mathbf{x}$ and for $t = 1, 2, \ldots, T$ set

$$o_t = f_t(o_{t-1}) = \begin{cases} W^{(t)} o_{t-1} & \text{for linear layer} \\ \boldsymbol{\sigma}(o_{t-1}) & \text{for activation layer} \end{cases}$$

Backward:

- set $\delta_{T+1} = \nabla \ell_y(o_T)$ and for $t = T, T-1, \ldots, 1$ set

$$\delta_t = \delta_{t+1} J_{o_t}(f_{t+1}) = \delta_{t+1} \cdot \begin{cases} W^{(t+1)} & \text{for linear layer} \\ \mathrm{diag}(\boldsymbol{\sigma}'(o_t)) & \text{for activation layer} \end{cases}$$

- For linear layers, set the gradient w.r.t. the weights in $W^{(t)}$ to be the elements of the matrix $\delta_t o_{t-1}^\top$

Outline

Convolutional Networks

- Designed for computer vision problems
- Three main ideas:
  - Convolutional layers: use the same weights on all patches of the image
  - Pooling layers: decrease image resolution (good for translation invariance, for higher level features, and for runtime)
  - Contrast normalization layers: let neurons "compete" with adjacent neurons

Convolutional and Pooling

- Layers are organized with three dimensional arrays, corresponding to width, height, and channel. E.g., in the first layer, if we have an RGB image of width 40 and heigh 80 than we have three channels, each of which is a 40x80 image

Convolutional and Pooling

- Layers are organized with three dimensional arrays, corresponding to width, height, and channel. E.g., in the first layer, if we have an RGB image of width 40 and heigh 80 than we have three channels, each of which is a 40x80 image
- Weight sharing: Each "neuron" maps the previous layer into a new image by convolving the previous layer with a "kernel"

$$o^+(h,w,c) = \sum_{c'} \sum_{i=1}^{k_h} \sum_{j=1}^{k_w} W(c,i,j,c') o(h+i, w+j, c') + b(c)$$

Convolutional and Pooling

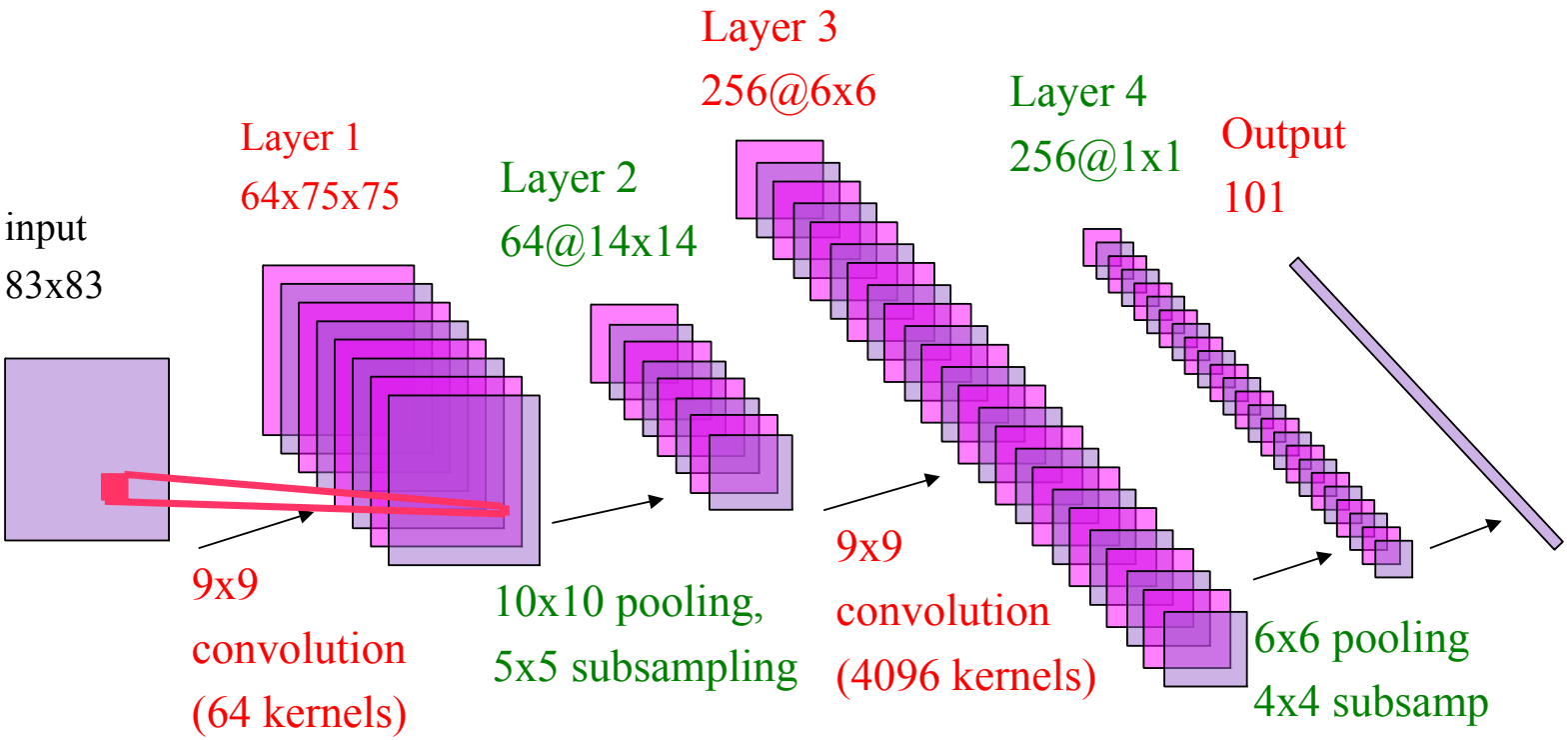
- Layers are organized with three dimensional arrays, corresponding to width, height, and channel. E.g., in the first layer, if we have an RGB image of width 40 and heigh 80 than we have three channels, each of which is a 40x80 image
- Weight sharing: Each "neuron" maps the previous layer into a new image by convolving the previous layer with a "kernel"

$$o^+(h,w,c) = \sum_{c'} \sum_{i=1}^{k_h} \sum_{j=1}^{k_w} W(c,i,j,c') o(h+i, w+j, c') + b(c)$$

- A pooling layer reduces the resolution of each image in the previous layer

Convolutional Network (ConvNet)

Y LeCun
MA Ranzato



Neural Networks as Feature Learning

- "Feature Engineering" approach: expert constructs feature mapping $\phi : \mathcal{X} \to \mathbb{R}^d$. Then, apply machine learning to find a linear predictor on $\phi(\mathbf{x})$.

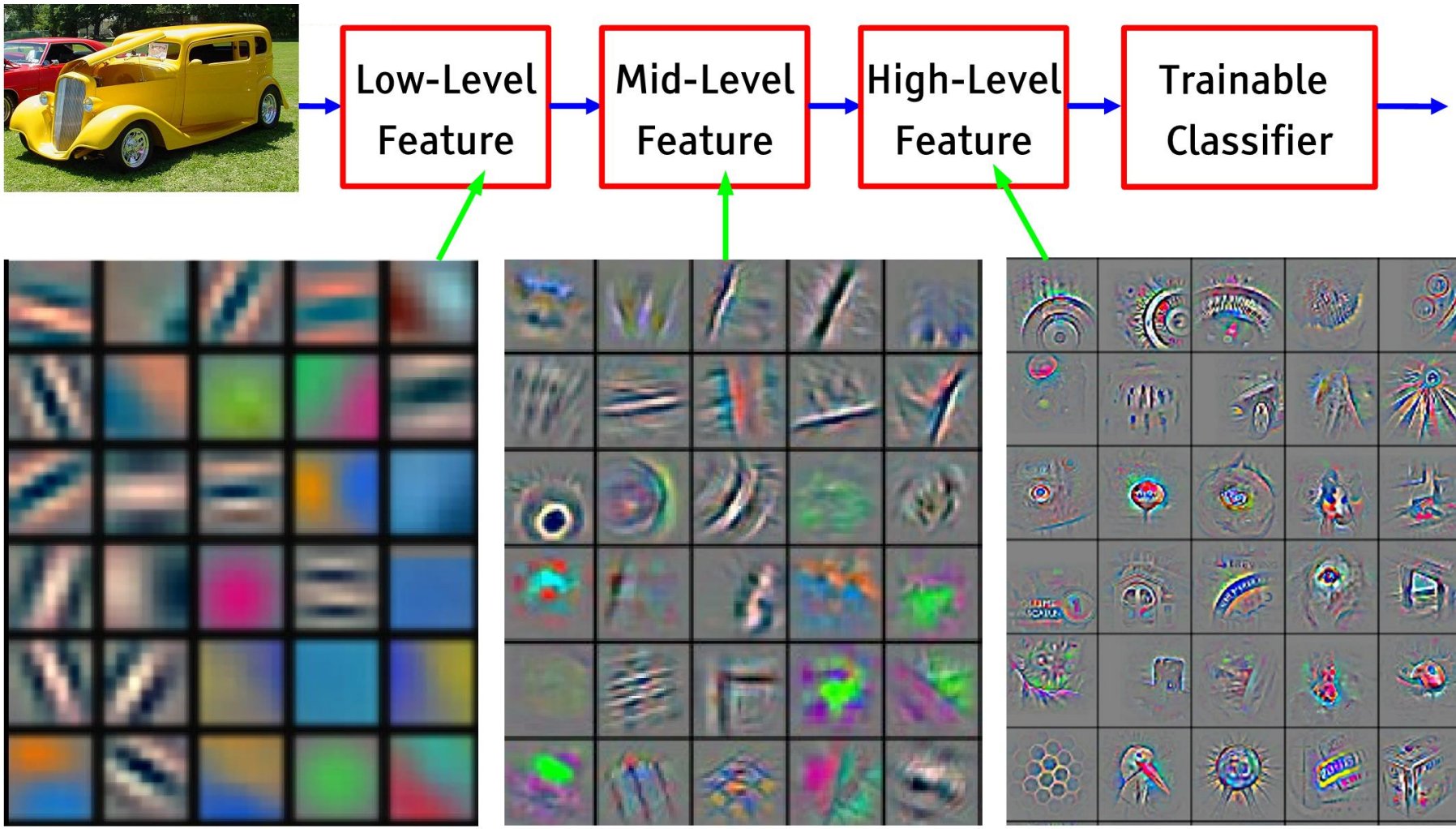
Neural Networks as Feature Learning

- "Feature Engineering" approach: expert constructs feature mapping $\phi : \mathcal{X} \to \mathbb{R}^d$. Then, apply machine learning to find a linear predictor on $\phi(\mathbf{x})$.
- "Deep learning" approach: neurons in hidden layers can be thought of as features that are being learned automatically from the data

Neural Networks as Feature Learning

- "Feature Engineering" approach: expert constructs feature mapping $\phi : \mathcal{X} \to \mathbb{R}^d$. Then, apply machine learning to find a linear predictor on $\phi(\mathbf{x})$.
- "Deep learning" approach: neurons in hidden layers can be thought of as features that are being learned automatically from the data
- Shallow neurons corresponds to low level features while deep neurons correspond to high level features

Neural Networks as Feature Learning



Taken from Yan LeCun's deep learning tutorial

Multiclass/Multitask/Feature Sharing/Representation learning

- Neurons in intermediate layers are shared by different tasks/classes
- Only last layer is specific to task/class
- Sometimes, network is optimized for certain classes, but the intermediate neurons are used as features for a new problem. This is called transfer learning. The last hidden layer can be thought of as a representation of the instance.

Neural Networks: Current Trends

- ReLU activation: $\sigma(a) = \max\{0, a\}$. This helps convergence, but do not hurt expressiveness

Neural Networks: Current Trends

- ReLU activation: $\sigma(a) = \max\{0, a\}$. This helps convergence, but do not hurt expressiveness
- Very large networks: often, the number of parameters is very large, even much larger than the number of examples. This might lead to overfitting, which is (partially) avoided by many types of regularization

Neural Networks: Current Trends

- ReLU activation: $\sigma(a) = \max\{0, a\}$. This helps convergence, but do not hurt expressiveness
- Very large networks: often, the number of parameters is very large, even much larger than the number of examples. This might lead to overfitting, which is (partially) avoided by many types of regularization
- Regularization: besides norm regularization, early stopping of SGD also serves as a regularizer

Neural Networks: Current Trends

- ReLU activation: $\sigma(a) = \max\{0, a\}$. This helps convergence, but do not hurt expressiveness
- Very large networks: often, the number of parameters is very large, even much larger than the number of examples. This might lead to overfitting, which is (partially) avoided by many types of regularization
- Regularization: besides norm regularization, early stopping of SGD also serves as a regularizer
- Dropout: this is another form of regularization, in which some neurons are "muted" at random during training

Neural Networks: Current Trends

- ReLU activation: $\sigma(a) = \max\{0, a\}$. This helps convergence, but do not hurt expressiveness
- Very large networks: often, the number of parameters is very large, even much larger than the number of examples. This might lead to overfitting, which is (partially) avoided by many types of regularization
- Regularization: besides norm regularization, early stopping of SGD also serves as a regularizer
- Dropout: this is another form of regularization, in which some neurons are "muted" at random during training
- Weight sharing (convolutional networks)

Neural Networks: Current Trends

- ReLU activation: $\sigma(a) = \max\{0, a\}$. This helps convergence, but do not hurt expressiveness
- Very large networks: often, the number of parameters is very large, even much larger than the number of examples. This might lead to overfitting, which is (partially) avoided by many types of regularization
- Regularization: besides norm regularization, early stopping of SGD also serves as a regularizer
- Dropout: this is another form of regularization, in which some neurons are "muted" at random during training
- Weight sharing (convolutional networks)
- SGD tricks: momentum, Nesterov's acceleration, other forms of second order approximation

Neural Networks: Current Trends

- ReLU activation: $\sigma(a) = \max\{0, a\}$. This helps convergence, but do not hurt expressiveness
- Very large networks: often, the number of parameters is very large, even much larger than the number of examples. This might lead to overfitting, which is (partially) avoided by many types of regularization
- Regularization: besides norm regularization, early stopping of SGD also serves as a regularizer
- Dropout: this is another form of regularization, in which some neurons are "muted" at random during training
- Weight sharing (convolutional networks)
- SGD tricks: momentum, Nesterov's acceleration, other forms of second order approximation
- Training on GPU

Historical Remarks

- 1940s-70s:
  - Inspired by learning/modeling the brain (Pitts, Hebb, and others)
  - Perceptron Rule (Rosenblatt), Multilayer perceptron (Minksy and Papert)
  - Backpropagation (Werbos 1975)
- 1980s – early 1990s:
  - Practical Back-prop (Rumelhart, Hinton et al 1986) and SGD (Bottou)
  - Initial empirical success
- 1990s-2000s:
  - Lost favor to implicit linear methods: SVM, Boosting
- 2006 –:
  - Regain popularity because of unsupervised pre-training (Hinton, Bengio, LeCun, Ng, and others)
  - Computational advances and several new tricks allow training HUGE networks. Empirical success leads to renewed interest
  - 2012: Krizhevsky, Sustkever, Hinton: significant improvement of state-of-the-art on imagenet dataset (object recognition of 1000 classes), without unsupervised pre-training

Summary

- Neural networks can be used to construct the ultimate hypothesis class
- Computationally, it's impossible to train neural networks
- . . . but, empirically, it works reasonably well
- Leads to state-of-the-art on many real world problems
- Biggest theoretical question: When does it work and why ?