Introduction to Machine Learning (67577)

Shai Shalev-Shwartz

School of CS and Engineering,
The Hebrew University of Jerusalem

Deep Learning

Outline

Gradient-Based Learning

- Consider a hypothesis class which is parameterized by a vector $\theta \in \mathbb{R}^d$

Gradient-Based Learning

- Consider a hypothesis class which is parameterized by a vector $\theta \in \mathbb{R}^d$
- Loss function of h_θ on example (x, y) is denoted $\ell(\theta; (x, y))$

Gradient-Based Learning

- Consider a hypothesis class which is parameterized by a vector $\theta \in \mathbb{R}^d$
- Loss function of h_θ on example (x, y) is denoted $\ell(\theta; (x, y))$
- The true and empirical risks are

$$L_{\mathcal{D}}(\theta) = \mathop{\mathbb{E}}_{(x,y)\sim\mathcal{D}}[\ell(\theta; (x, y))] \quad , \quad L_S(\theta) = \frac{1}{m}\sum_{i=1}^{m} \ell(\theta; (x_i, y_i))$$

Gradient-Based Learning

- Consider a hypothesis class which is parameterized by a vector $\theta \in \mathbb{R}^d$
- Loss function of h_θ on example (x, y) is denoted $\ell(\theta; (x, y))$
- The true and empirical risks are

$$L_{\mathcal{D}}(\theta) = \mathop{\mathbb{E}}_{(x,y)\sim\mathcal{D}}[\ell(\theta; (x, y))] \quad , \quad L_S(\theta) = \frac{1}{m}\sum_{i=1}^{m} \ell(\theta; (x_i, y_i))$$

- Assumption: ℓ is differentiable w.r.t. θ and we can calculate $\nabla \ell(\theta; (x, y))$ efficiently

Gradient-Based Learning

- Consider a hypothesis class which is parameterized by a vector $\theta \in \mathbb{R}^d$
- Loss function of h_θ on example (x, y) is denoted $\ell(\theta; (x, y))$
- The true and empirical risks are

$$L_{\mathcal{D}}(\theta) \;=\; \mathop{\mathbb{E}}_{(x,y)\sim\mathcal{D}}[\ell(\theta; (x, y))] \quad , \qquad L_S(\theta) = \frac{1}{m}\sum_{i=1}^{m} \ell(\theta; (x_i, y_i))$$

- Assumption: ℓ is differentiable w.r.t. θ and we can calculate $\nabla\ell(\theta; (x, y))$ efficiently
- Minimize $L_{\mathcal{D}}$ or L_S with Stochastic Gradient Descent (SGD): Start with $\theta^{(0)}$ and update $\theta^{(t+1)} = \theta^{(t)} - \eta_t \nabla\ell(\theta^{(t)}; (x, y))$

Gradient-Based Learning

- Consider a hypothesis class which is parameterized by a vector $\theta \in \mathbb{R}^d$
- Loss function of h_θ on example (x, y) is denoted $\ell(\theta; (x, y))$
- The true and empirical risks are

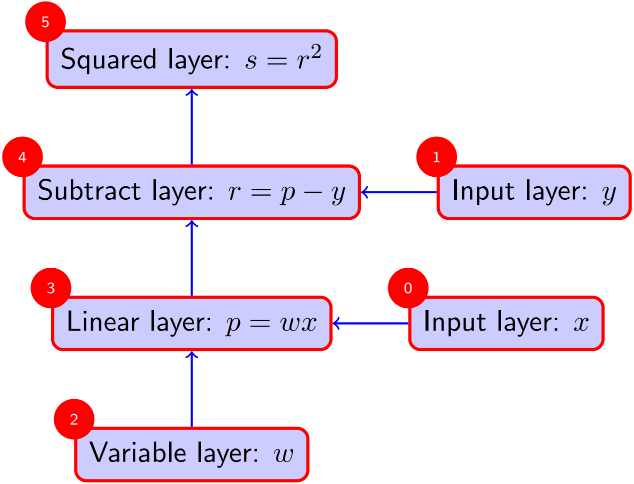
$$L_{\mathcal{D}}(\theta) \; = \; \mathop{\mathbb{E}}_{(x,y)\sim\mathcal{D}}[\ell(\theta;(x,y))] \quad , \quad L_S(\theta) = \frac{1}{m}\sum_{i=1}^{m} \ell(\theta;(x_i,y_i))$$

- Assumption: ℓ is differentiable w.r.t. θ and we can calculate $\nabla\ell(\theta;(x,y))$ efficiently
- Minimize $L_{\mathcal{D}}$ or L_S with Stochastic Gradient Descent (SGD): Start with $\theta^{(0)}$ and update $\theta^{(t+1)} = \theta^{(t)} - \eta_t \nabla\ell(\theta^{(t)};(x,y))$
- SGD converges for convex problems. It may work for non-convex problems if we initialize “close enough” to a “good minimum”

Outline

A computation graph for a one dimensional Least Squares

(numbering of nodes corresponds to topological sort):



Gradient Calculation using the Chain Rule

- Fix x, y and write ℓ as a function of w by

$$\ell(w) = s(r_y(p_x(w))) = (s \circ r_y \circ p_x)(w) \ .$$

Gradient Calculation using the Chain Rule

- Fix x, y and write ℓ as a function of w by

$$\ell(w) = s(r_y(p_x(w))) = (s \circ r_y \circ p_x)(w) \ .$$

- Chain rule:

$$\begin{aligned}
\ell'(w) &= (s \circ r_y \circ p_x)'(w) \\
&= s'(r_y(p_x(w))) \cdot (r_y \circ p_x)'(w) \\
&= s'(r_y(p_x(w))) \cdot r_y'(p_x(w)) \cdot p_x'(w)
\end{aligned}$$

Gradient Calculation using the Chain Rule

- Fix x, y and write ℓ as a function of w by

$$\ell(w) = s(r_y(p_x(w))) = (s \circ r_y \circ p_x)(w) \ .$$

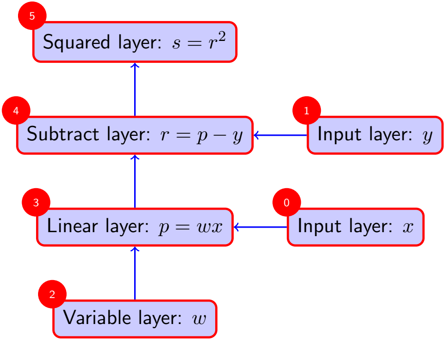
- Chain rule:

$$\begin{aligned}\ell'(w) &= (s \circ r_y \circ p_x)'(w) \\ &= s'(r_y(p_x(w))) \cdot (r_y \circ p_x)'(w) \\ &= s'(r_y(p_x(w))) \cdot r_y'(p_x(w)) \cdot p_x'(w)\end{aligned}$$

- Backpropagation: Calculate by a Forward-Backward pass over the graph

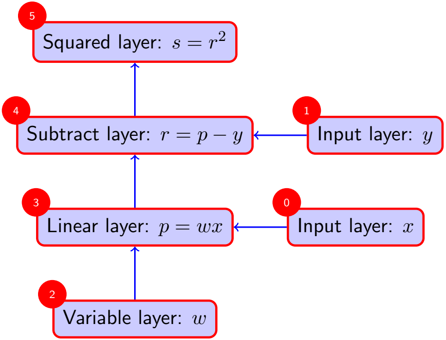
Computation Graph — Forward

- For $t = 0, 1, \ldots, T-1$
  - `Layer[t]->output = Layer[t]->function(Layer[t]->inputs)`



Computation Graph — Backward

- Recall: $\ell'(w) = s'(r_y(p_x(w))) \cdot r'_y(p_x(w)) \cdot p'_x(w)$
- `Layer[T-1]->delta = 1`
- For $t = T-1, T-2, \ldots, 0$
  - For `i in Layer[t]->inputs`:
    - `i->delta = Layer[t]->delta * Layer[t]->derivative(i,Layer[t]->inputs)`



Layers

- Nodes in the computation graph are often called layers

Layers

- Nodes in the computation graph are often called layers
- Each layer is a simple differentiable function

Layers

- Nodes in the computation graph are often called layers
- Each layer is a simple differentiable function
- Layers can implement multivariate functions

Layers

- Nodes in the computation graph are often called layers
- Each layer is a simple differentiable function
- Layers can implement multivariate functions
- Example of popular layers:

Layers

- Nodes in the computation graph are often called layers
- Each layer is a simple differentiable function
- Layers can implement multivariate functions
- Example of popular layers:
  - Affine layer: $O = WX + b\,1^\top$ where $W \in \mathbb{R}^{m,n}, x \in \mathbb{R}^{n,c}, b \in \mathbb{R}^m$

Layers

- Nodes in the computation graph are often called layers
- Each layer is a simple differentiable function
- Layers can implement multivariate functions
- Example of popular layers:
  - Affine layer: $O = WX + b\,1^\top$ where $W \in \mathbb{R}^{m,n}, x \in \mathbb{R}^{n,c}, b \in \mathbb{R}^m$
  - Unary layer: $\forall i,\ o_i = f(x_i)$ for some $f : \mathbb{R} \to \mathbb{R}$ e.g.

Layers

- Nodes in the computation graph are often called layers
- Each layer is a simple differentiable function
- Layers can implement multivariate functions
- Example of popular layers:
  - Affine layer: $O = WX + b\,1^\top$ where $W \in \mathbb{R}^{m,n}, x \in \mathbb{R}^{n,c}, b \in \mathbb{R}^m$
  - Unary layer: $\forall i,\ o_i = f(x_i)$ for some $f : \mathbb{R} \to \mathbb{R}$ e.g.
    - Sigmoid: $f(x) = (1 + \exp(-x))^{-1}$

Layers

- Nodes in the computation graph are often called layers
- Each layer is a simple differentiable function
- Layers can implement multivariate functions
- Example of popular layers:
  - Affine layer: $O = WX + b\,1^\top$ where $W \in \mathbb{R}^{m,n}, x \in \mathbb{R}^{n,c}, b \in \mathbb{R}^m$
  - Unary layer: $\forall i,\ o_i = f(x_i)$ for some $f : \mathbb{R} \to \mathbb{R}$ e.g.
    - Sigmoid: $f(x) = (1 + \exp(-x))^{-1}$
    - Rectified Linear Unit (ReLU): $f(x) = \max\{0, x\}$ (discuss: derivative?)

Layers

- Nodes in the computation graph are often called layers
- Each layer is a simple differentiable function
- Layers can implement multivariate functions
- Example of popular layers:
  - Affine layer: $O = WX + b\,1^\top$ where $W \in \mathbb{R}^{m,n}, x \in \mathbb{R}^{n,c}, b \in \mathbb{R}^m$
  - Unary layer: $\forall i,\ o_i = f(x_i)$ for some $f : \mathbb{R} \to \mathbb{R}$ e.g.
    - Sigmoid: $f(x) = (1 + \exp(-x))^{-1}$
    - Rectified Linear Unit (ReLU): $f(x) = \max\{0, x\}$ (discuss: derivative?)
  - Binary layer: $\forall i,\ o_i = f(x_i, y_i)$ for some $f : \mathbb{R}^2 \to \mathbb{R}$ e.g.

Layers

- Nodes in the computation graph are often called layers
- Each layer is a simple differentiable function
- Layers can implement multivariate functions
- Example of popular layers:
  - Affine layer: $O = WX + b\,1^\top$ where $W \in \mathbb{R}^{m,n}, x \in \mathbb{R}^{n,c}, b \in \mathbb{R}^m$
  - Unary layer: $\forall i,\ o_i = f(x_i)$ for some $f : \mathbb{R} \to \mathbb{R}$ e.g.
    - Sigmoid: $f(x) = (1 + \exp(-x))^{-1}$
    - Rectified Linear Unit (ReLU): $f(x) = \max\{0, x\}$ (discuss: derivative?)
  - Binary layer: $\forall i,\ o_i = f(x_i, y_i)$ for some $f : \mathbb{R}^2 \to \mathbb{R}$ e.g.
    - Add layer: $f(x, y) = x + y$

Layers

- Nodes in the computation graph are often called layers
- Each layer is a simple differentiable function
- Layers can implement multivariate functions
- Example of popular layers:
  - Affine layer: $O = WX + b\,1^\top$ where $W \in \mathbb{R}^{m,n}, x \in \mathbb{R}^{n,c}, b \in \mathbb{R}^m$
  - Unary layer: $\forall i,\ o_i = f(x_i)$ for some $f : \mathbb{R} \to \mathbb{R}$ e.g.
    - Sigmoid: $f(x) = (1 + \exp(-x))^{-1}$
    - Rectified Linear Unit (ReLU): $f(x) = \max\{0, x\}$ (discuss: derivative?)
  - Binary layer: $\forall i,\ o_i = f(x_i, y_i)$ for some $f : \mathbb{R}^2 \to \mathbb{R}$ e.g.
    - Add layer: $f(x, y) = x + y$
    - Hinge loss: $f(x, y) = [1 - y_i x_i]_+$

Layers

- Nodes in the computation graph are often called layers
- Each layer is a simple differentiable function
- Layers can implement multivariate functions
- Example of popular layers:
  - Affine layer: $O = W X + b\,1^\top$ where $W \in \mathbb{R}^{m,n}, x \in \mathbb{R}^{n,c}, b \in \mathbb{R}^m$
  - Unary layer: $\forall i,\ o_i = f(x_i)$ for some $f : \mathbb{R} \to \mathbb{R}$ e.g.
    - Sigmoid: $f(x) = (1 + \exp(-x))^{-1}$
    - Rectified Linear Unit (ReLU): $f(x) = \max\{0, x\}$ (discuss: derivative?)
  - Binary layer: $\forall i,\ o_i = f(x_i, y_i)$ for some $f : \mathbb{R}^2 \to \mathbb{R}$ e.g.
    - Add layer: $f(x, y) = x + y$
    - Hinge loss: $f(x, y) = [1 - y_i x_i]_+$
    - Logistic loss: $f(x, y) = \log(1 + \exp(-y_i x_i))$

Layers

- Nodes in the computation graph are often called layers
- Each layer is a simple differentiable function
- Layers can implement multivariate functions
- Example of popular layers:
  - Affine layer: $O = W X + b\,1^\top$ where $W \in \mathbb{R}^{m,n}, x \in \mathbb{R}^{n,c}, b \in \mathbb{R}^m$
  - Unary layer: $\forall i,\ o_i = f(x_i)$ for some $f : \mathbb{R} \to \mathbb{R}$ e.g.
    - Sigmoid: $f(x) = (1 + \exp(-x))^{-1}$
    - Rectified Linear Unit (ReLU): $f(x) = \max\{0, x\}$ (discuss: derivative?)
  - Binary layer: $\forall i,\ o_i = f(x_i, y_i)$ for some $f : \mathbb{R}^2 \to \mathbb{R}$ e.g.
    - Add layer: $f(x, y) = x + y$
    - Hinge loss: $f(x, y) = [1 - y_i x_i]_+$
    - Logistic loss: $f(x, y) = \log(1 + \exp(-y_i x_i))$

Layers

- Nodes in the computation graph are often called layers
- Each layer is a simple differentiable function
- Layers can implement multivariate functions
- Example of popular layers:
  - Affine layer: $O = WX + b\,1^\top$ where $W \in \mathbb{R}^{m,n}, x \in \mathbb{R}^{n,c}, b \in \mathbb{R}^m$
  - Unary layer: $\forall i,\ o_i = f(x_i)$ for some $f : \mathbb{R} \to \mathbb{R}$ e.g.
    - Sigmoid: $f(x) = (1 + \exp(-x))^{-1}$
    - Rectified Linear Unit (ReLU): $f(x) = \max\{0, x\}$ (discuss: derivative?)
  - Binary layer: $\forall i,\ o_i = f(x_i, y_i)$ for some $f : \mathbb{R}^2 \to \mathbb{R}$ e.g.
    - Add layer: $f(x, y) = x + y$
    - Hinge loss: $f(x, y) = [1 - y_i x_i]_+$
    - Logistic loss: $f(x, y) = \log(1 + \exp(-y_i x_i))$

Main message

Computation graph enables us to construct very complicated functions from simple building blocks

Backpropgation for multivariate layers

- Recall the backpropagation rule:
  - For `i in Layer[t]->inputs`:
    - `i->delta = Layer[t]->delta * Layer[t]->derivative(i,Layer[t]->inputs)`

Backpropgation for multivariate layers

- Recall the backpropagation rule:
  - For `i in Layer[t]->inputs`:
    - `i->delta = Layer[t]->delta * Layer[t]->derivative(i,Layer[t]->inputs)`
- "delta" is now a vector (same dimension as the output of the layer)

Backpropgation for multivariate layers

- Recall the backpropagation rule:
  - For `i in Layer[t]->inputs`:
    - `i->delta = Layer[t]->delta * Layer[t]->derivative(i,Layer[t]->inputs)`
- "delta" is now a vector (same dimension as the output of the layer)
- "derivative" is the Jacobian matrix:
  The Jacobian of $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$ at $\mathbf{x} \in \mathbb{R}^n$, denoted $J_{\mathbf{x}}(\mathbf{f})$, 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 $\mathbf{x}$.

Backpropgation for multivariate layers

- Recall the backpropagation rule:
  - For `i in Layer[t]->inputs`:
    - `i->delta = Layer[t]->delta * Layer[t]->derivative(i,Layer[t]->inputs)`
- "delta" is now a vector (same dimension as the output of the layer)
- "derivative" is the Jacobian matrix:
  The Jacobian of $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$ at $\mathbf{x} \in \mathbb{R}^n$, denoted $J_{\mathbf{x}}(\mathbf{f})$, 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 $\mathbf{x}$.
- The multiplication is matrix multiplication

Backpropgation for multivariate layers

- Recall the backpropagation rule:
  - For `i in Layer[t]->inputs`:
    - `i->delta = Layer[t]->delta * Layer[t]->derivative(i,Layer[t]->inputs)`
- "delta" is now a vector (same dimension as the output of the layer)
- "derivative" is the Jacobian matrix:
  The Jacobian of $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$ at $\mathbf{x} \in \mathbb{R}^n$, denoted $J_{\mathbf{x}}(\mathbf{f})$, 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 $\mathbf{x}$.
- The multiplication is matrix multiplication
- The correctness of the algorithm follows from the multivariate chain rule

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

Jacobian — Examples

- If $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^n$ is element-wise application of $\sigma : \mathbb{R} \to \mathbb{R}$ then $J_{\mathbf{x}}(\mathbf{f}) = \mathrm{diag}((\sigma'(x_1), \ldots, \sigma'(x_n)))$.

Jacobian — Examples

- If $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^n$ is element-wise application of $\sigma : \mathbb{R} \to \mathbb{R}$ then $J_{\mathbf{x}}(\mathbf{f}) = \mathrm{diag}((\sigma'(x_1), \ldots, \sigma'(x_n)))$.
- Let $\mathbf{f}(\mathbf{x}, \mathbf{w}, b) = \mathbf{w}^\top \mathbf{x} + b$ for $\mathbf{w}, \mathbf{x} \in \mathbb{R}^n, b \in \mathbb{R}^1$. Then:

$$J_{\mathbf{x}}(\mathbf{f}) = \mathbf{w}^\top \quad , \quad J_{\mathbf{w}}(\mathbf{f}) = \mathbf{x}^\top \quad , \quad J_b(\mathbf{f}) = 1$$

Jacobian — Examples

- If $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^n$ is element-wise application of $\sigma : \mathbb{R} \to \mathbb{R}$ then $J_{\mathbf{x}}(\mathbf{f}) = \mathrm{diag}((\sigma'(x_1), \ldots, \sigma'(x_n)))$.
- Let $\mathbf{f}(\mathbf{x}, \mathbf{w}, b) = \mathbf{w}^\top \mathbf{x} + b$ for $\mathbf{w}, \mathbf{x} \in \mathbb{R}^n, b \in \mathbb{R}^1$. Then:

$$J_{\mathbf{x}}(\mathbf{f}) = \mathbf{w}^\top \quad , \quad J_{\mathbf{w}}(\mathbf{f}) = \mathbf{x}^\top \quad , \quad J_b(\mathbf{f}) = 1$$

- Let $\mathbf{f}(W, \mathbf{x}) = W\mathbf{x}$. Then:

$$J_{\mathbf{x}}(\mathbf{f}) = W \quad , \quad J_W(\mathbf{f}) = \begin{pmatrix} \mathbf{x}^\top & 0 & \cdots & 0 \\ 0 & \mathbf{x}^\top & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \mathbf{x}^\top \end{pmatrix} \quad .$$

Outline

Sample Complexity

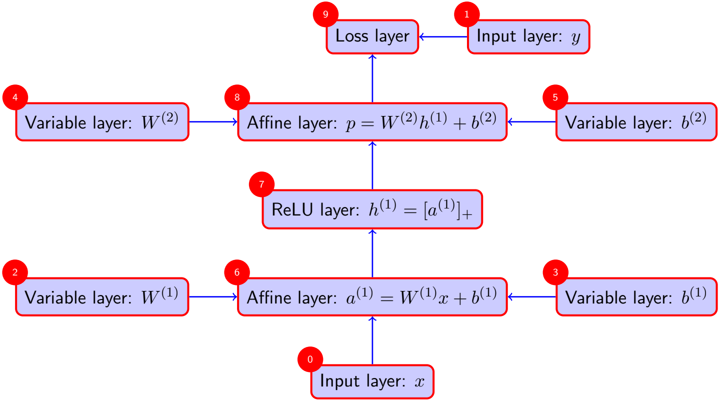
- If we learn d parameters, and each one is stored in, say, 32 bits, then the number of hypotheses in our class is at most 2^{32d}. It follows that the sample complexity is order of d.

Sample Complexity

- If we learn d parameters, and each one is stored in, say, 32 bits, then the number of hypotheses in our class is at most 2^{32d}. It follows that the sample complexity is order of d.
- Other ways to improve generalization is all sort of regularization

Expressiveness

- So far in the course we considered hypotheses of the form $x \mapsto w^\top x + b$
- Now, consider the following computation graph, known as "one hidden layer network":



Expressiveness of "One Hidden Layer Network"

- Claim: Every Boolean function $f : \{\pm 1\}^n \to \{\pm 1\}$ can be expressed by a one hidden layer network.

Expressiveness of "One Hidden Layer Network"

- Claim: Every Boolean function $f : \{\pm 1\}^n \to \{\pm 1\}$ can be expressed by a one hidden layer network.
- Proof:

Expressiveness of "One Hidden Layer Network"

- Claim: Every Boolean function $f : \{\pm 1\}^n \to \{\pm 1\}$ can be expressed by a one hidden layer network.
- Proof:
  - Show that for integer x we have $\mathrm{sign}(x) = 2([x+1]_+ - [x]_+) - 1$

Expressiveness of "One Hidden Layer Network"

- Claim: Every Boolean function $f : \{\pm 1\}^n \to \{\pm 1\}$ can be expressed by a one hidden layer network.
- Proof:
  - Show that for integer x we have $\mathrm{sign}(x) = 2([x+1]_+ - [x]_+) - 1$
  - Show that any f can be written as $f(x) = \vee_i (x == u_i)$ for some vectors $u_1, \ldots, u_k$

Expressiveness of "One Hidden Layer Network"

- Claim: Every Boolean function $f : \{\pm 1\}^n \to \{\pm 1\}$ can be expressed by a one hidden layer network.
- Proof:
  - Show that for integer x we have $\mathrm{sign}(x) = 2([x+1]_+ - [x]_+) - 1$
  - Show that any f can be written as $f(x) = \vee_i (x == u_i)$ for some vectors $u_1, \ldots, u_k$
  - Show that $\mathrm{sign}(x^\top u_i - (n-1))$ is an indicator to $(x == u_i)$

Expressiveness of "One Hidden Layer Network"

- Claim: Every Boolean function $f : \{\pm 1\}^n \to \{\pm 1\}$ can be expressed by a one hidden layer network.
- Proof:
  - Show that for integer x we have $\mathrm{sign}(x) = 2([x+1]_+ - [x]_+) - 1$
  - Show that any f can be written as $f(x) = \vee_i (x == u_i)$ for some vectors $u_1, \ldots, u_k$
  - Show that $\mathrm{sign}(x^\top u_i - (n-1))$ is an indicator to $(x == u_i)$
  - Conclude that we can adjust the weights so that $yp(x) \geq 1$ for all examples (x, y)

Expressiveness of "One Hidden Layer Network"

- Claim: Every Boolean function $f : \{\pm 1\}^n \to \{\pm 1\}$ can be expressed by a one hidden layer network.
- Proof:
  - Show that for integer x we have $\mathrm{sign}(x) = 2([x+1]_+ - [x]_+) - 1$
  - Show that any f can be written as $f(x) = \vee_i (x == u_i)$ for some vectors $u_1, \ldots, u_k$
  - Show that $\mathrm{sign}(x^\top u_i - (n-1))$ is an indicator to $(x == u_i)$
  - Conclude that we can adjust the weights so that $yp(x) \geq 1$ for all examples (x, y)
- Theorem: For every n, let $s(n)$ be the minimal integer such that there exists a one hidden layer network with $s(n)$ hidden neurons that implements all functions from $\{0,1\}^n$ to $\{0,1\}$. Then, $s(n)$ is exponential in n.

Expressiveness of "One Hidden Layer Network"

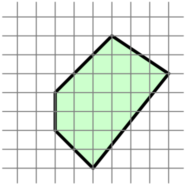
- Claim: Every Boolean function $f : \{\pm 1\}^n \to \{\pm 1\}$ can be expressed by a one hidden layer network.
- Proof:
  - Show that for integer x we have $\mathrm{sign}(x) = 2([x+1]_+ - [x]_+) - 1$
  - Show that any f can be written as $f(x) = \vee_i (x == u_i)$ for some vectors $u_1, \ldots, u_k$
  - Show that $\mathrm{sign}(x^\top u_i - (n-1))$ is an indicator to $(x == u_i)$
  - Conclude that we can adjust the weights so that $yp(x) \geq 1$ for all examples (x, y)
- Theorem: For every n, let $s(n)$ be the minimal integer such that there exists a one hidden layer network with $s(n)$ hidden neurons that implements all functions from $\{0,1\}^n$ to $\{0,1\}$. Then, $s(n)$ is exponential in n.
- Proof: Think on the VC dimension ...

Expressiveness of "One Hidden Layer Network"

- Claim: Every Boolean function $f : \{\pm 1\}^n \to \{\pm 1\}$ can be expressed by a one hidden layer network.
- Proof:
  - Show that for integer x we have $\text{sign}(x) = 2([x+1]_+ - [x]_+) - 1$
  - Show that any f can be written as $f(x) = \vee_i (x == u_i)$ for some vectors $u_1, \ldots, u_k$
  - Show that $\text{sign}(x^\top u_i - (n-1))$ is an indicator to $(x == u_i)$
  - Conclude that we can adjust the weights so that $yp(x) \geq 1$ for all examples (x, y)
- Theorem: For every n, let $s(n)$ be the minimal integer such that there exists a one hidden layer network with $s(n)$ hidden neurons that implements all functions from $\{0,1\}^n$ to $\{0,1\}$. Then, $s(n)$ is exponential in n.
- Proof: Think on the VC dimension ...
- What type of functions can be implemented by small size networks?

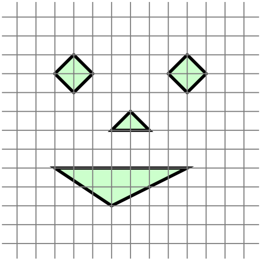
Geometric Intuition

- One hidden layer networks can express intersection of halfspaces



Geometric Intuition

- Two hidden layer networks can express unions of intersection of halfspaces



What can we express with T-depth 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 network of depth at most T and size at most $cT(n)^2 + b$ such that it implements all functions in $\mathcal{F}_n$.

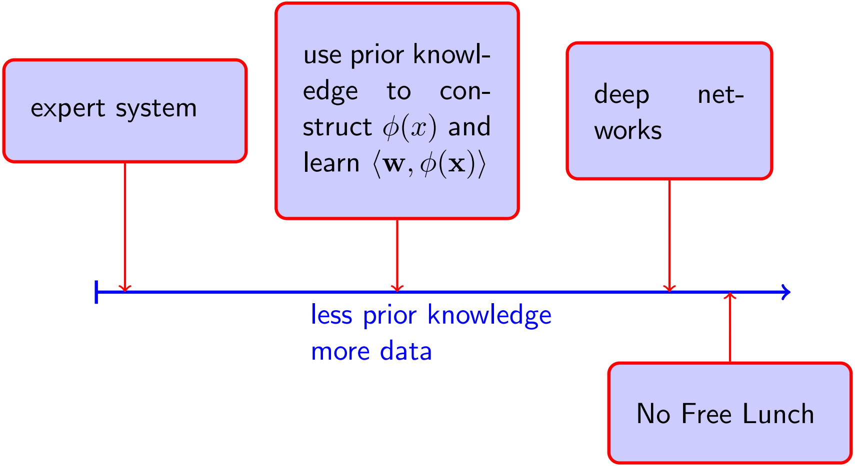
What can we express with T-depth networks ?

- 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 network of depth at most T and size at most $cT(n)^2 + b$ such that it implements all functions in $\mathcal{F}_n$.
- Sample complexity is order of number of variables (in our case polynomial in T)

What can we express with T-depth 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 network of depth at most T and size at most $cT(n)^2 + b$ such that it implements all functions in $\mathcal{F}_n$.
- Sample complexity is order of number of variables (in our case polynomial in T)
- 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 depth T and size $O(T(n)^2)$, and the sample complexity is also bounded by polynomial in $T(n)$!

The ultimate hypothesis class



Outline

Runtime of learning neural networks

- Theorem: It is NP hard to implement the ERM rule even for one hidden layer networks with just 4 neurons in the hidden layer.

Runtime of learning neural networks

- Theorem: It is NP hard to implement the ERM rule even for one hidden layer networks with just 4 neurons in the hidden layer.
- But, maybe ERM is hard but some improper algorithm works ?

Runtime of learning neural networks

- Theorem: It is NP hard to implement the ERM rule even for one hidden layer networks with 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 one hidden layer networks with $\omega(\log(d))$ hidden neurons 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 programs 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 programs that can be implemented in code length of b bits ?
- Main technique: Gradient-based learning (using 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 programs that can be implemented in code length of b bits ?
- Main technique: Gradient-based learning (using 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 programs that can be implemented in code length of b bits ?
- Main technique: Gradient-based learning (using SGD)
- Not convex, no guarantees, can take a long time, but:
  - Often (but not always) 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 programs that can be implemented in code length of b bits ?
- Main technique: Gradient-based learning (using SGD)
- Not convex, no guarantees, can take a long time, but:
  - Often (but not always) still works fine, finds a good solution
  - Easier than optimizing over Python programs ...

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 programs that can be implemented in code length of b bits ?
- Main technique: Gradient-based learning (using SGD)
- Not convex, no guarantees, can take a long time, but:
  - Often (but not always) still works fine, finds a good solution
  - Easier than optimizing over Python programs ...
  - Need to apply some tricks (initialization, learning rate, mini-batching, architecture), and need some luck

Outline

The MNIST dataset

- The task: Handwritten digits recognition

The MNIST dataset

- The task: Handwritten digits recognition
  - Input space: $\mathcal{X} = \{0, 1, \ldots, 255\}^{28 \times 28}$

The MNIST dataset

- The task: Handwritten digits recognition
  - Input space: $\mathcal{X} = \{0, 1, \ldots, 255\}^{28 \times 28}$
  - Output space: $\mathcal{Y} = \{0, 1, \ldots, 9\}$

The MNIST dataset

- The task: Handwritten digits recognition
  - Input space: $\mathcal{X} = \{0, 1, \ldots, 255\}^{28\times 28}$
  - Output space: $\mathcal{Y} = \{0, 1, \ldots, 9\}$
- Multiclass categorization:

The MNIST dataset

- The task: Handwritten digits recognition
  - Input space: $\mathcal{X} = \{0, 1, \ldots, 255\}^{28 \times 28}$
  - Output space: $\mathcal{Y} = \{0, 1, \ldots, 9\}$
- Multiclass categorization:
  - We take hypotheses of the form $h : \mathcal{X} \to \mathbb{R}^{|\mathcal{Y}|}$

The MNIST dataset

- The task: Handwritten digits recognition
  - Input space: $\mathcal{X} = \{0, 1, \ldots, 255\}^{28\times 28}$
  - Output space: $\mathcal{Y} = \{0, 1, \ldots, 9\}$
- Multiclass categorization:
  - We take hypotheses of the form $h : \mathcal{X} \to \mathbb{R}^{|\mathcal{Y}|}$
  - We interpret $h(x)$ as a vector that gives scores for all the labels

The MNIST dataset

- The task: Handwritten digits recognition
  - Input space: $\mathcal{X} = \{0, 1, \ldots, 255\}^{28\times 28}$
  - Output space: $\mathcal{Y} = \{0, 1, \ldots, 9\}$
- Multiclass categorization:
  - We take hypotheses of the form $h : \mathcal{X} \to \mathbb{R}^{|\mathcal{Y}|}$
  - We interpret $h(x)$ as a vector that gives scores for all the labels
  - The actual prediction is the label with the highest score: $\mathrm{argmax}_i \, h_i(x)$

The MNIST dataset

- The task: Handwritten digits recognition
  - Input space: $\mathcal{X} = \{0, 1, \ldots, 255\}^{28\times 28}$
  - Output space: $\mathcal{Y} = \{0, 1, \ldots, 9\}$
- Multiclass categorization:
  - We take hypotheses of the form $h : \mathcal{X} \to \mathbb{R}^{|\mathcal{Y}|}$
  - We interpret $h(x)$ as a vector that gives scores for all the labels
  - The actual prediction is the label with the highest score: $\mathrm{argmax}_i\, h_i(x)$
- Network architecture: $x \to$ Affine(500) $\to$ ReLU $\to$ Affine(10).

The MNIST dataset

- The task: Handwritten digits recognition
  - Input space: $\mathcal{X} = \{0, 1, \ldots, 255\}^{28 \times 28}$
  - Output space: $\mathcal{Y} = \{0, 1, \ldots, 9\}$
- Multiclass categorization:
  - We take hypotheses of the form $h : \mathcal{X} \to \mathbb{R}^{|\mathcal{Y}|}$
  - We interpret $h(x)$ as a vector that gives scores for all the labels
  - The actual prediction is the label with the highest score: $\operatorname{argmax}_i h_i(x)$
- Network architecture: $x \to$ Affine(500) $\to$ ReLU $\to$ Affine(10).
- Logistic loss for multiclass categorization:

The MNIST dataset

- The task: Handwritten digits recognition
  - Input space: $\mathcal{X} = \{0, 1, \ldots, 255\}^{28\times 28}$
  - Output space: $\mathcal{Y} = \{0, 1, \ldots, 9\}$
- Multiclass categorization:
  - We take hypotheses of the form $h : \mathcal{X} \to \mathbb{R}^{|\mathcal{Y}|}$
  - We interpret $h(x)$ as a vector that gives scores for all the labels
  - The actual prediction is the label with the highest score: $\mathrm{argmax}_i \, h_i(x)$
- Network architecture: $x \to$ Affine(500) $\to$ ReLU $\to$ Affine(10).
- Logistic loss for multiclass categorization:
  - SoftMax: $\forall i, \quad p_i = \frac{\exp(h_i(x))}{\sum_j \exp(h_j(x))}$

The MNIST dataset

- The task: Handwritten digits recognition
  - Input space: $\mathcal{X} = \{0, 1, \ldots, 255\}^{28\times 28}$
  - Output space: $\mathcal{Y} = \{0, 1, \ldots, 9\}$
- Multiclass categorization:
  - We take hypotheses of the form $h : \mathcal{X} \to \mathbb{R}^{|\mathcal{Y}|}$
  - We interpret $h(x)$ as a vector that gives scores for all the labels
  - The actual prediction is the label with the highest score: $\mathrm{argmax}_i\, h_i(x)$
- Network architecture: $x \to$ Affine(500) $\to$ ReLU $\to$ Affine(10).
- Logistic loss for multiclass categorization:
  - SoftMax: $\forall i, \quad p_i = \frac{\exp(h_i(x))}{\sum_j \exp(h_j(x))}$
  - LogLoss: If the correct label is y then the loss is $-\log(p_y) = \log\left(\sum_j \exp(h_j(x) - h_i(x))\right)$

Some Training Tricks

- Input normalization: divide each element of x by 255 to make sure it is in $[0, 1]$

Some Training Tricks

- Input normalization: divide each element of x by 255 to make sure it is in $[0, 1]$
- Initialization is important: One trick that works well in practice is to initialize the bias to be zero and initialize the rows of W to be random in $[-1/\sqrt{n}, 1/\sqrt{n}]$

Some Training Tricks

- Input normalization: divide each element of x by 255 to make sure it is in $[0, 1]$
- Initialization is important: One trick that works well in practice is to initialize the bias to be zero and initialize the rows of W to be random in $[-1/\sqrt{n}, 1/\sqrt{n}]$
- Mini-batches: At each iteration of SGD we calculate the average loss on k random examples for $k > 1$. Advantages:

Some Training Tricks

- Input normalization: divide each element of x by 255 to make sure it is in $[0, 1]$
- Initialization is important: One trick that works well in practice is to initialize the bias to be zero and initialize the rows of W to be random in $[-1/\sqrt{n}, 1/\sqrt{n}]$
- Mini-batches: At each iteration of SGD we calculate the average loss on k random examples for $k > 1$. Advantages:
  - Reduces the variance of the update direction (w.r.t. the full gradient), hence converges faster

Some Training Tricks

- Input normalization: divide each element of x by 255 to make sure it is in $[0, 1]$
- Initialization is important: One trick that works well in practice is to initialize the bias to be zero and initialize the rows of W to be random in $[-1/\sqrt{n}, 1/\sqrt{n}]$
- Mini-batches: At each iteration of SGD we calculate the average loss on k random examples for $k > 1$. Advantages:
  - Reduces the variance of the update direction (w.r.t. the full gradient), hence converges faster
  - We don't pay a lot in time because of parallel implementation

Some Training Tricks

- Input normalization: divide each element of x by 255 to make sure it is in $[0, 1]$
- Initialization is important: One trick that works well in practice is to initialize the bias to be zero and initialize the rows of W to be random in $[-1/\sqrt{n}, 1/\sqrt{n}]$
- Mini-batches: At each iteration of SGD we calculate the average loss on k random examples for $k > 1$. Advantages:
  - Reduces the variance of the update direction (w.r.t. the full gradient), hence converges faster
  - We don't pay a lot in time because of parallel implementation
- Learning rate: Choice of learning rate is important. One way is to start with some fixed η and decrease it by $1/2$ whenever the training stops making progress.

Some Training Tricks

- Input normalization: divide each element of x by 255 to make sure it is in $[0, 1]$
- Initialization is important: One trick that works well in practice is to initialize the bias to be zero and initialize the rows of W to be random in $[-1/\sqrt{n}, 1/\sqrt{n}]$
- Mini-batches: At each iteration of SGD we calculate the average loss on k random examples for $k > 1$. Advantages:
  - Reduces the variance of the update direction (w.r.t. the full gradient), hence converges faster
  - We don't pay a lot in time because of parallel implementation
- Learning rate: Choice of learning rate is important. One way is to start with some fixed η and decrease it by $1/2$ whenever the training stops making progress.
- Variants of SGD: There are plenty of variants that work better than vanilla SGD.

Failures of Deep Learning

- Parity of more than 30 bits
- Multiplication of large numbers
- Matrix inversion
- ...

Outline

Convolutional Networks

- Convolution layer:

Convolutional Networks

- Convolution layer:
  - Input: C images

Convolutional Networks

- Convolution layer:
  - Input: C images
  - Output: C' images

Convolutional Networks

- Convolution layer:
  - Input: C images
  - Output: C' images
  - Calculation:

$$O[c',h',w'] = b^{(c')} + \sum_{c=0}^{C-1} \sum_{h=0}^{k-1} \sum_{w=0}^{k-1} W^{(c')}[c,h,w]\, X[c,h+h',w+w']$$

Convolutional Networks

- Convolution layer:
  - Input: C images
  - Output: C' images
  - Calculation:

$$O[c',h',w'] = b^{(c')} + \sum_{c=0}^{C-1}\sum_{h=0}^{k-1}\sum_{w=0}^{k-1} W^{(c')}[c,h,w]\,X[c,h+h',w+w']$$

  - Observe: equivalent to an Affine layer with weight sharing

Convolutional Networks

- Convolution layer:
  - Input: C images
  - Output: C' images
  - Calculation:

$$O[c',h',w'] = b^{(c')} + \sum_{c=0}^{C-1}\sum_{h=0}^{k-1}\sum_{w=0}^{k-1} W^{(c')}[c,h,w]\,X[c,h+h',w+w']$$

  - Observe: equivalent to an Affine layer with weight sharing
  - Observe: can be implemented as a combination of Im2Col layer and Affine layer

Convolutional Networks

- Convolution layer:
  - Input: C images
  - Output: C' images
  - Calculation:

$$O[c',h',w'] = b^{(c')} + \sum_{c=0}^{C-1} \sum_{h=0}^{k-1} \sum_{w=0}^{k-1} W^{(c')}[c,h,w]\, X[c,h+h',w+w']$$

  - Observe: equivalent to an Affine layer with weight sharing
  - Observe: can be implemented as a combination of Im2Col layer and Affine layer
- Pooling layer:

Convolutional Networks

- Convolution layer:
  - Input: C images
  - Output: C' images
  - Calculation:

$$O[c',h',w'] = b^{(c')} + \sum_{c=0}^{C-1}\sum_{h=0}^{k-1}\sum_{w=0}^{k-1} W^{(c')}[c,h,w]\, X[c,h+h',w+w']$$

  - Observe: equivalent to an Affine layer with weight sharing
  - Observe: can be implemented as a combination of Im2Col layer and Affine layer
- Pooling layer:
  - Input: Image of size $H \times W$

Convolutional Networks

- Convolution layer:
  - Input: C images
  - Output: C' images
  - Calculation:

$$O[c',h',w'] = b^{(c')} + \sum_{c=0}^{C-1}\sum_{h=0}^{k-1}\sum_{w=0}^{k-1} W^{(c')}[c,h,w]\, X[c,h+h',w+w']$$

  - Observe: equivalent to an Affine layer with weight sharing
  - Observe: can be implemented as a combination of Im2Col layer and Affine layer
- Pooling layer:
  - Input: Image of size $H \times W$
  - Output: Image of size $(H/k) \times (W/k)$

Convolutional Networks

- Convolution layer:
  - Input: C images
  - Output: C' images
  - Calculation:

$$O[c', h', w'] = b^{(c')} + \sum_{c=0}^{C-1} \sum_{h=0}^{k-1} \sum_{w=0}^{k-1} W^{(c')}[c, h, w] \, X[c, h + h', w + w']$$

  - Observe: equivalent to an Affine layer with weight sharing
  - Observe: can be implemented as a combination of Im2Col layer and Affine layer
- Pooling layer:
  - Input: Image of size $H \times W$
  - Output: Image of size $(H/k) \times (W/k)$
  - Calculation: Divide input image to $k \times k$ windows and for each such window output the maximal value (or average value)

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

- Deep Learning can be used to construct the ultimate hypothesis class
- Worst-case complexity is exponential
- . . . but, empirically, it works reasonably well and leads to state-of-the-art on many real world problems