Introduction to Machine Learning (67577)

Shai Shalev-Shwartz

School of CS and Engineering,
The Hebrew University of Jerusalem

Deep Learning
Outline

1. Gradient-Based Learning
2. Computation Graph and Backpropagation
3. Expressiveness and Sample Complexity
4. Computational Complexity
5. Convolutional Networks
6. Solving MNIST with LeNet using Tensorflow
7. Tips and Tricks
Consider a hypothesis class which is parameterized by a vector $\theta \in \mathbb{R}^d$.
Consider a hypothesis class which is parameterized by a vector $\theta \in \mathbb{R}^d$

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

Loss function of $h_\theta$ on example $(x, y)$ is denoted $\ell(\theta; (x, y))$

The true and empirical risks are

$$L_D(\theta) = \mathbb{E}_{(x,y) \sim \mathcal{D}}[\ell(\theta; (x, y))] \quad , \quad L_S(\theta) = \frac{1}{m} \sum_{i=1}^{m} \ell(\theta; (x_i, y_i))$$
Consider a hypothesis class which is parameterized by a vector $\theta \in \mathbb{R}^d$.

Loss function of $h_\theta$ on example $(x, y)$ is denoted $\ell(\theta; (x, y))$.

The true and empirical risks are

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

Assumption: $\ell$ is differentiable w.r.t. $\theta$ and we can calculate $\nabla \ell(\theta; (x, y))$ efficiently.
Consider a hypothesis class which is parameterized by a vector $\theta \in \mathbb{R}^d$

- Loss function of $h_\theta$ on example $(x, y)$ is denoted $\ell(\theta; (x, y))$

- The true and empirical risks are

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

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

- Minimize $L_D$ or $L_S$ with Stochastic Gradient Descent (SGD):
  Start with $\theta^{(0)}$ and update $\theta^{(t+1)} = \theta^{(t)} - \eta_t \nabla \ell(\theta^{(t)}; (x, y))$
Consider a hypothesis class which is parameterized by a vector $\theta \in \mathbb{R}^d$.

Loss function of $h_\theta$ on example $(x, y)$ is denoted $\ell(\theta; (x, y))$.

The true and empirical risks are

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

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

Minimize $L_D$ 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

1. Gradient-Based Learning
2. Computation Graph and Backpropagation
3. Expressiveness and Sample Complexity
4. Computational Complexity
5. Convolutional Networks
6. Solving MNIST with LeNet using Tensorflow
7. Tips and Tricks
A computation graph for a one dimensional Least Squares

(numbering of nodes corresponds to topological sort):

1. Input layer: $y$
2. Variable layer: $w$
3. Linear layer: $p = wx$
4. Subtract layer: $r = p - y$
5. Squared layer: $s = r^2$
Gradient Calculation using the Chain Rule

- Fix $x, y$ and write $\ell$ 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 $\ell$ as a function of $w$ by

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

- Chain rule:

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

$$= s'(r_y(p_x(w))) \cdot (r_y \circ p_x)'(w)$$

$$= s'(r_y(p_x(w))) \cdot r'_y(p_x(w)) \cdot p'_x(w)$$
Gradient Calculation using the Chain Rule

- **Fix** \( x, y \) and write \( \ell \) as a function of \( w \) by

\[
\ell(w) = s(r_y(p_x(w))) = (s \circ r_y \circ p_x)(w).
\]

- **Chain rule:**

\[
\ell'(w) = (s \circ r_y \circ p_x)'(w) = s'(r_y(p_x(w))) \cdot (r_y \circ p_x)'(w) = s'(r_y(p_x(w))) \cdot r_y'(p_x(w)) \cdot p_x'(w).
\]

- **Backpropagation:** Calculate by a Forward-Backward pass over the graph
For \( t = 0, 1, \ldots, T - 1 \)

- \( \text{Layer}[t] \to \text{output} = \text{Layer}[t] \to \text{function}(\text{Layer}[t] \to \text{inputs}) \)
Recall: \[ \ell'(w) = s'(r_y(p_x(w))) \cdot r'_y(p_x(w)) \cdot p'_x(w) \]

Layer\[T-1]\rightarrow \text{delta} = 1

For \( t = T - 1, T - 2, \ldots, 0 \)

For \( i \) in Layer\[t]\rightarrow \text{inputs}:

\( i \rightarrow \text{delta} = \text{Layer}[t]\rightarrow \text{delta} \times \text{Layer}[t]\rightarrow \text{derivative}(i, \text{Layer}[t]\rightarrow \text{inputs}) \)
Nodes in the computation graph are often called **layers**
Nodes in the computation graph are often called layers.

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

- Each layer is a simple differentiable function
- Layers can implement multivariate functions
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:
$$O = WX + b$$
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 $f: \mathbb{R} \to \mathbb{R}$.
Examples:
- 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 $f: \mathbb{R}^2 \to \mathbb{R}$
Examples:
- Add layer: $f(x, y) = x + y$
- Hinge loss: $f(x, y) = [1 - y_i x_i]_+$
- Logistic loss: $f(x, y) = \log(1 + \exp(-y_i x_i))$
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 \( W \in \mathbb{R}^{m,n}, x \in \mathbb{R}^{n,c}, b \in \mathbb{R}^m \)
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:** \( O = WX + b \) \( \top \) 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 \( f : \mathbb{R} \rightarrow \mathbb{R} \) e.g.
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**: $f(x) = (1 + \exp(-x))^{-1}$
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**: \( O = WX + b \) 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**: \( f(x) = \frac{1}{1 + \exp(-x)}^{-1} \)
  - **Rectified Linear Unit (ReLU)**: \( f(x) = \max\{0, x\} \) (discuss: derivative?)
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.
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: \( f(x, y) = x + y \)
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 \( f : \mathbb{R} \rightarrow \mathbb{R} \) 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 \( f : \mathbb{R}^2 \rightarrow \mathbb{R} \) e.g.
  - **Add layer:** \( f(x, y) = x + y \)
  - **Hinge loss:** \( f(x, y) = [1 - y_i x_i]_+ \)
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: \( f(x, y) = \log(1 + \exp(-y_i x_i)) \)
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:** \( f(x, y) = \log(1 + \exp(-y_i x_i)) \)
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 \( f : \mathbb{R} \rightarrow \mathbb{R} \) 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 \( f : \mathbb{R}^2 \rightarrow \mathbb{R} \) 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
Backpropagation 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 $f: \mathbb{R}^n \rightarrow \mathbb{R}^m$ at $x \in \mathbb{R}^n$, denoted $J_x(f)$, is the $m \times n$ matrix whose $i,j$ element is the partial derivative of $f$ w.r.t. its $j$'th variable at $x$.

The multiplication is matrix multiplication.

The correctness of the algorithm follows from the multivariate chain rule:

$J_w(f \circ g) = J_g(w)(f) \cdot J_w(g)$.
Recall the backpropagation rule:

For i in Layer[t]->inputs:

- \( i \rightarrow \text{delta} = \text{Layer}[t] \rightarrow \text{delta} \times \text{Layer}[t] \rightarrow \text{derivative}(i, \text{Layer}[t] \rightarrow \text{inputs}) \)

“delta” is now a vector (same dimension as the output of the layer)
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 \( f : \mathbb{R}^n \rightarrow \mathbb{R}^m \) at \( x \in \mathbb{R}^n \), denoted \( J_x(f) \), is the \( m \times n \) matrix whose \( i, j \) element is the partial derivative of \( f_i : \mathbb{R}^n \rightarrow \mathbb{R} \) w.r.t. its \( j \)'th variable at \( x \).
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 $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ at $x \in \mathbb{R}^n$, denoted $J_x(f)$, is the $m \times n$ matrix whose $i, j$ element is the partial derivative of $f_i : \mathbb{R}^n \rightarrow \mathbb{R}$ w.r.t. its $j$'th variable at $x$.

The multiplication is matrix multiplication
Backpropagation for multivariate layers

- Recall the backpropagation rule:
  - For \( i \) in \( \text{Layer}[t]->inputs \):
    - \( i->\text{delta} = \text{Layer}[t]->\text{delta} \times \text{Layer}[t]->\text{derivative}(i, \text{Layer}[t]->\text{inputs}) \)
  - “\( \text{delta} \)” is now a vector (same dimension as the output of the layer)
  - “\( \text{derivative} \)” is the Jacobian matrix:

    The Jacobian of \( f : \mathbb{R}^n \to \mathbb{R}^m \) at \( x \in \mathbb{R}^n \), denoted \( J_x(f) \), 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 \( x \).

- The multiplication is matrix multiplication
- The correctness of the algorithm follows from the multivariate chain rule

\[
J_w(f \circ g) = J_{g(w)}(f) J_w(g)
\]
If $f : \mathbb{R}^n \rightarrow \mathbb{R}^n$ is element-wise application of $\sigma : \mathbb{R} \rightarrow \mathbb{R}$ then $J_x(f) = \text{diag}((\sigma'(x_1), \ldots, \sigma'(x_n))).$
Jacobian — Examples

- If \( f : \mathbb{R}^n \to \mathbb{R}^n \) is element-wise application of \( \sigma : \mathbb{R} \to \mathbb{R} \) then
  \[ J_x(f) = \text{diag}((\sigma'(x_1), \ldots, \sigma'(x_n))). \]

- Let \( f(x, w, b) = w^\top x + b \) for \( w, x \in \mathbb{R}^n, b \in \mathbb{R}^1 \). Then:
  \[ J_x(f) = w^\top, \quad J_w(f) = x^\top, \quad J_b(f) = 1. \]
Jacobian — Examples

- If $f : \mathbb{R}^n \rightarrow \mathbb{R}^n$ is element-wise application of $\sigma : \mathbb{R} \rightarrow \mathbb{R}$ then $J_x(f) = \text{diag}(\sigma'(x_1), \ldots, \sigma'(x_n))$.

- Let $f(x, w, b) = w^\top x + b$ for $w, x \in \mathbb{R}^n, b \in \mathbb{R}^1$. Then:
  
  $$J_x(f) = w^\top, \quad J_w(f) = x^\top, \quad J_b(f) = 1$$

- Let $f(W, x) = Wx$. Then:
  
  $$J_x(f) = W, \quad J_W(f) = \begin{pmatrix} x^\top & 0 & \cdots & 0 \\ 0 & x^\top & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & x^\top \end{pmatrix}.$$
1. Gradient-Based Learning
2. Computation Graph and Backpropagation
3. Expressiveness and Sample Complexity
4. Computational Complexity
5. Convolutional Networks
6. Solving MNIST with LeNet using Tensorflow
7. Tips and Tricks
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.
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”:

- Input layer: \( x \)
- Affine layer: \( a^{(1)} = W^{(1)} x + b^{(1)} \)
- ReLU layer: \( h^{(1)} = [a^{(1)}]^+ \)
- Affine layer: \( p = W^{(2)} h^{(1)} + b^{(2)} \)
- Variable layer: \( W^{(2)} \)
- Variable layer: \( b^{(2)} \)
- Loss layer
**Claim:** Every Boolean function \( f : \{\pm 1\}^n \rightarrow \{\pm 1\} \) can be expressed by a one hidden layer network.

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

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

Proof:
- Show that for integer $x$ we have $\text{sign}(x) = 2([x + 1]+ - [x]+) - 1$
Claim: Every Boolean function \( f : \{\pm 1\}^n \rightarrow \{\pm 1\} \) 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 \( f(x) = \lor_i (x == u_i) \) for some vectors \( u_1, \ldots, u_k \)

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...
Claim: Every Boolean function \( f : \{\pm 1\}^n \rightarrow \{\pm 1\} \) 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 \( f(x) = \bigvee_i (x == u_i) \) for some vectors \( u_1, \ldots, u_k \)
- Show that \( \text{sign}(x^\top u_i - (n - 1)) \) is an indicator to \( (x == u_i) \)
Claim: Every Boolean function $f : \{-1, 1\}^n \rightarrow \{-1, 1\}$ 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 $f(x) = \lor_i (x = u_i)$ 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 $y_p(x) \geq 1$ for all examples $(x, y)$
Claim: Every Boolean function \( f : \{\pm 1\}^n \rightarrow \{\pm 1\} \) 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 \( f(x) = \bigvee_i (x == u_i) \) 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 \( y_p(x) \geq 1 \) 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 $f : \{\pm 1\}^n \rightarrow \{\pm 1\}$ 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 $f(x) = \lor_i (x == u_i)$ 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 $y_p(x) \geq 1$ 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 ...
Claim: Every Boolean function $f : \{\pm 1\}^n \rightarrow \{\pm 1\}$ can be expressed by a one hidden layer network.

Proof:

1. Show that for integer $x$ we have $\text{sign}(x) = 2([x + 1]_+ - [x]_+) - 1$
2. Show that any $f$ can be written as $f(x) = \lor_i (x == u_i)$ for some vectors $u_1, \ldots, u_k$
3. Show that $\text{sign}(x^\top u_i - (n - 1))$ is an indicator to $(x == u_i)$
4. Conclude that we can adjust the weights so that $y_p(x) \geq 1$ 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
Two hidden layer networks can express unions of intersection of halfspaces
What can we express with $T$-depth networks?

- **Theorem**: Let $T : \mathbb{N} \rightarrow \mathbb{N}$ 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 $T(n)$.
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 $c T(n)^2 + b$ 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 $c T(n)^2 + b$ 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

- expert system
- use prior knowledge to construct $\phi(x)$ and learn $\langle w, \phi(x) \rangle$
- deep networks
- less prior knowledge, more data
- No Free Lunch
Outline

1. Gradient-Based Learning
2. Computation Graph and Backpropagation
3. Expressiveness and Sample Complexity
4. Computational Complexity
5. Convolutional Networks
6. Solving MNIST with LeNet using Tensorflow
7. Tips and Tricks
Theorem: It is NP hard to implement the ERM rule even for one hidden layer networks with just 4 neurons in the hidden layer.
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: 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.
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

1. Gradient-Based Learning
2. Computation Graph and Backpropagation
3. Expressiveness and Sample Complexity
4. Computational Complexity
5. Convolutional Networks
6. Solving MNIST with LeNet using Tensorflow
7. Tips and Tricks
Deep Learning Golden age in Vision

- 2012-2014 Imagenet results:

<table>
<thead>
<tr>
<th>2012 Teams</th>
<th>%error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Supervision (Toronto)</td>
<td>15.3</td>
</tr>
<tr>
<td>ISI (Tokyo)</td>
<td>26.1</td>
</tr>
<tr>
<td>VGG (Oxford)</td>
<td>26.9</td>
</tr>
<tr>
<td>XRCE/INRIA</td>
<td>27.0</td>
</tr>
<tr>
<td>UvA (Amsterdam)</td>
<td>29.6</td>
</tr>
<tr>
<td>INRIA/LEAR</td>
<td>33.4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2013 Teams</th>
<th>%error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clarifai (NYU spinoff)</td>
<td>11.7</td>
</tr>
<tr>
<td>NUS (Singapore)</td>
<td>12.9</td>
</tr>
<tr>
<td>Zeiler-Fergus (NYU)</td>
<td>13.5</td>
</tr>
<tr>
<td>A. Howard</td>
<td>13.5</td>
</tr>
<tr>
<td>OverFeat (NYU)</td>
<td>14.1</td>
</tr>
<tr>
<td>UvA (Amsterdam)</td>
<td>14.2</td>
</tr>
<tr>
<td>Adobe</td>
<td>15.2</td>
</tr>
<tr>
<td>VGG (Oxford)</td>
<td>15.2</td>
</tr>
<tr>
<td>VGG (Oxford)</td>
<td>23.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2014 Teams</th>
<th>%error</th>
</tr>
</thead>
<tbody>
<tr>
<td>GoogLeNet</td>
<td>6.6</td>
</tr>
<tr>
<td>VGG (Oxford)</td>
<td>7.3</td>
</tr>
<tr>
<td>MSRA</td>
<td>8.0</td>
</tr>
<tr>
<td>A. Howard</td>
<td>8.1</td>
</tr>
<tr>
<td>DeeperVision</td>
<td>9.5</td>
</tr>
<tr>
<td>NUS-BST</td>
<td>9.7</td>
</tr>
<tr>
<td>TTIC-ECP</td>
<td>10.2</td>
</tr>
<tr>
<td>XYZ</td>
<td>11.2</td>
</tr>
<tr>
<td>UvA</td>
<td>12.1</td>
</tr>
</tbody>
</table>

- 2015 results: MSRA under **3.5%** error. (using a CNN with 150 layers!)

figures from Yann LeCun’s CVPR’15 plenary
Convolution Layer

- **Input:** \( C \) images

\[
O\left[c', h', w'ight] = b\left[c'ight] + C_{-1} \sum_{c=0}^{k-1} \sum_{h=0}^{k-1} \sum_{w=0}^{k-1} W\left[c', c, h, w\right] X\left[c, h + h', w + w'\right]
\]

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

- Input: $C$ images
- Output: $C'$ images

\[
O_{[c',h',w']} = b(c') + \sum_{k=0}^{H-1} \sum_{h=0}^{W-1} W(c')_{[c,h,w]} X_{[c,h]+h',w'+w']}
\]

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

- Input: $C$ images
- Output: $C'$ images
- Calculation:

$$O[c', h', w'] = b^{(c')} + \sum_{c=0}^{C-1} \sum_{h=0}^{k-1} \sum_{w=0}^{k-1} W^{(c')}[c, h, w] X[c, h + h', w + w']$$
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
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
Im2Col Layer

- Im2Col for $3 \times 3$ convolution

```
 0  1  2  3  4
 5  6  7  8  9
10 11 12 13 14
15 16 17 18 19
20 21 22 23 24
```

```
0  0  1
 0  1  2
 1  2  3
 0  5  6
 5  6  7
 6  7  8
 0 10 11
10 11 12
11 12 13
```
Im2Col Layer

- Im2Col for $3 \times 3$ convolution with 2 input channels
Parameters of Convolutions layer

- Kernel height and kernel width
- Stride height and stride width
- zero padding (True or False)
- Number of output channels
Pooling Layer

- Input: Image of size $H \times W$

Calculation: Divide input image to $k \times k$ windows and for each such window output the maximal value (or average value)

Observe: equivalent to Im2Col + reduce operation

Discuss: how to calculate derivative?
Pooling Layer

- **Input:** Image of size $H \times W$
- **Output:** Image of size $(H/k) \times (W/k)$
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)
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)
- Observe: equivalent to Im2Col + reduce operation
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)
- **Observe:** equivalent to Im2Col + reduce operation
- **Discuss:** how to calculate derivative?
Outline

1. Gradient-Based Learning
2. Computation Graph and Backpropagation
3. Expressiveness and Sample Complexity
4. Computational Complexity
5. Convolutional Networks
6. Solving MNIST with LeNet using Tensorflow
7. Tips and Tricks
Example: LeNet for MNIST

- **The task**: Handwritten digits recognition
Example: LeNet for MNIST

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

- **The task**: Handwritten digits recognition
  - Input space: \( \mathcal{X} = \{0, 1, \ldots, 255\}^{28 \times 28} \)
  - Output space: \( \mathcal{Y} = \{0, 1, \ldots, 9\} \)
Example: LeNet for MNIST

- **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:**

Network architecture:
- \( x \rightarrow \text{Conv}(5x5, 1x1, no-pad, 20) \rightarrow \text{Pool}(2x2) \rightarrow \text{Conv}(5x5, 1x1, no-pad, 50) \rightarrow \text{Pool}(2x2) \rightarrow \text{Affine}(500) \rightarrow \text{ReLU} \rightarrow \text{Affine}(10) \).
Example: LeNet for MNIST

- **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 $h : \mathcal{X} \rightarrow \mathbb{R}^{|\mathcal{Y}|}$
Example: LeNet for MNIST

- **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 $h : \mathcal{X} \rightarrow \mathbb{R}^{|\mathcal{Y}|}$
  - We interpret $h(x)$ as a vector that gives scores for all the labels
Example: LeNet for MNIST

- **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 $h : \mathcal{X} \rightarrow \mathbb{R}^{\left|\mathcal{Y}\right|}$
  - We interpret $h(x)$ as a vector that gives scores for all the labels
  - The actual prediction is the label with the highest score: $\arg\max_i h_i(x)$
Example: LeNet for MNIST

- **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 \( h : \mathcal{X} \rightarrow \mathbb{R}^{|\mathcal{Y}|} \)
  - We interpret \( h(x) \) as a vector that gives scores for all the labels
  - The actual prediction is the label with the highest score: \( \arg\max_i h_i(x) \)

- **Network architecture:** \( x \rightarrow \text{Conv}(5\times5,1\times1,\text{no-pad},20) \rightarrow \text{Pool}(2\times2) \rightarrow \text{Conv}(5\times5,1\times1,\text{no-pad},50) \rightarrow \text{Pool}(2\times2) \rightarrow \text{Affine}(500) \rightarrow \text{ReLU} \rightarrow \text{Affine}(10) \).
Example: LeNet for MNIST

- **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 $h : \mathcal{X} \rightarrow \mathbb{R}^{|\mathcal{Y}|}$
  - We interpret $h(x)$ as a vector that gives scores for all the labels
  - The actual prediction is the label with the highest score: $\text{argmax}_i h_i(x)$

- **Network architecture:**
  $x \rightarrow \text{Conv}(5 \times 5, 1 \times 1, \text{no-pad}, 20) \rightarrow \text{Pool}(2 \times 2) \rightarrow \text{Conv}(5 \times 5, 1 \times 1, \text{no-pad}, 50) \rightarrow \text{Pool}(2 \times 2) \rightarrow \text{Affine}(500) \rightarrow \text{ReLU} \rightarrow \text{Affine}(10)$.

- **Logistic loss for multiclass categorization:**

\[
\text{SoftMax: } \forall i, p_i = \exp(h_i(x)) \sum_j \exp(h_j(x))
\]

\[
\text{LogLoss: If the correct label is } y \text{ then the loss is } -\log(p_y) = \log(\sum_j \exp(h_j(x)) - h_i(x))
\]
Example: LeNet for MNIST

- **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 $h : \mathcal{X} \rightarrow \mathbb{R}^{|\mathcal{Y}|}$
  - We interpret $h(x)$ as a vector that gives scores for all the labels
  - The actual prediction is the label with the highest score: $\arg\max_i h_i(x)$

- **Network architecture:**
  - $x \rightarrow \text{Conv}(5\times5, 1\times1, \text{no-pad}, 20) \rightarrow \text{Pool}(2\times2) \rightarrow \text{Conv}(5\times5, 1\times1, \text{no-pad}, 50) \rightarrow \text{Pool}(2\times2) \rightarrow \text{Affine}(500) \rightarrow \text{ReLU} \rightarrow \text{Affine}(10)$

- **Logistic loss for multiclass categorization:**
  - SoftMax: $\forall i, \quad p_i = \frac{\exp(h_i(x))}{\sum_j \exp(h_j(x))}$
Example: LeNet for MNIST

- **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: \( \arg\max_i h_i(x) \)

- **Network architecture:** \( x \to \text{Conv}(5 \times 5, 1 \times 1, \text{no-pad}, 20) \to \text{Pool}(2 \times 2) \to \text{Conv}(5 \times 5, 1 \times 1, \text{no-pad}, 50) \to \text{Pool}(2 \times 2) \to \text{Affine}(500) \to \text{ReLU} \to \text{Affine}(10) \).

- **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 \( -\log(p_y) = \log \left( \sum_j \exp(h_j(x) - h_i(x)) \right) \)
Outline

1. Gradient-Based Learning
2. Computation Graph and Backpropagation
3. Expressiveness and Sample Complexity
4. Computational Complexity
5. Convolutional Networks
6. Solving MNIST with LeNet using Tensorflow
7. Tips and Tricks
Reduction Layers

- The complexity of convolutional layers is $C_{in} \times C_{out} \times H \times W$
- A “reduction layer” is a $1 \times 1$ convolution aiming at reducing $C_{in}$
- It can greatly reduce the computational complexity (less time) and sample complexity (fewer parameters)
Inception modules

- Szegedy et al (Google)
- Won the ImageNet 2014 challenge (6.67% error)
Residual Networks

Figure 2. Residual learning: a building block.

- He, Zhang, Ren, Sun (Microsoft)
- Won the ImageNet 2015 challenge with a 152 layers network (3.57% error)
Some Training Tricks

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

- **Initialization**: 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 $\eta$ 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.
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

Learning rate: Choice of learning rate is important. One way is to start with some fixed $\eta$ 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.
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 $\eta$ 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 $\eta$ 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
- ...
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.