Neural networks

COMS 4771
1. Logistic regression
Logistic regression

Suppose $\mathcal{X} = \mathbb{R}^d$ and $\mathcal{Y} = \{0, 1\}$. A logistic regression model is a statistical model where the conditional probability function has a particular form:

$$Y \mid X = x \sim \text{Bern}(\text{logistic}(x^T w)), \quad x \in \mathbb{R}^d,$$

with

$$\text{logistic}(z) := \frac{1}{1 + e^{-z}} = \frac{e^z}{1 + e^z}, \quad z \in \mathbb{R}.$$

- Parameters: $w = (w_1, \ldots, w_d) \in \mathbb{R}^d$.
- Conditional probability function: $\eta_w(x) = \text{logistic}(x^T w)$. 

![Logistic function graph](image.png)
Logistic regression

Network diagram for $\eta_w$:

\[ v := g(z), \quad z := \sum_{j=1}^{d} w_j x_j, \quad (g = \text{logistic}). \]

Here, $g$ is called the \textit{link function}. 
Learning $w$ from data

Training data $((x_i, y_i))_{i=1}^n$ from $\mathbb{R}^d \times \{0, 1\}$.

- Could use MLE to learn $w$ from data.
Learning $\mathbf{w}$ from data

Training data $((\mathbf{x}_i, y_i))_{i=1}^n$ from $\mathbb{R}^d \times \{0, 1\}$.

- Could use MLE to learn $\mathbf{w}$ from data.
- Another option: Squared loss ERM (with link function $g$)

$$
\min_{\mathbf{w} \in \mathbb{R}^d} \frac{1}{n} \sum_{i=1}^n (g(\mathbf{x}_i^T \mathbf{w}) - y_i)^2.
$$
Learning $\mathbf{w}$ from data

Training data $((\mathbf{x}_i, y_i))_{i=1}^n$ from $\mathbb{R}^d \times \{0, 1\}$.

- Could use MLE to learn $\mathbf{w}$ from data.
- Another option: Squared loss ERM (with link function $g$)

$$\min_{\mathbf{w} \in \mathbb{R}^d} \frac{1}{n} \sum_{i=1}^n (g(\mathbf{x}_i^T \mathbf{w}) - y_i)^2.$$ 

- Observe that for any $(\mathbf{X}, Y) \sim P$ (not necessarily logistic regression),

$$\mathbb{E} \left[ (g(\mathbf{x}^T \mathbf{w}) - Y)^2 \mid \mathbf{X} = \mathbf{x} \right] = \left( g(\mathbf{x}^T \mathbf{w}) - \eta(\mathbf{x}) \right)^2 + \operatorname{var}(Y \mid \mathbf{X} = \mathbf{x})$$

where $\eta(\mathbf{x}) = \mathbb{P}(Y = 1 \mid \mathbf{X} = \mathbf{x})$. 
Learning $\mathbf{w}$ from data

Training data $((\mathbf{x}_i, y_i))_{i=1}^n$ from $\mathbb{R}^d \times \{0, 1\}$.

- Could use MLE to learn $\mathbf{w}$ from data.
- Another option: Squared loss ERM (with link function $g$)

$$
\min_{\mathbf{w} \in \mathbb{R}^d} \frac{1}{n} \sum_{i=1}^n (g(\mathbf{x}_i^T \mathbf{w}) - y_i)^2.
$$

- Observe that for any $(\mathbf{X}, Y) \sim P$ (not necessarily logistic regression),

$$
\mathbb{E} \left[ \left( g(\mathbf{x}^T \mathbf{w}) - Y \right)^2 \mid \mathbf{X} = \mathbf{x} \right] = \left( g(\mathbf{x}^T \mathbf{w}) - \eta(\mathbf{x}) \right)^2 + \text{var}(Y \mid \mathbf{X} = \mathbf{x})
$$

where $\eta(\mathbf{x}) = \mathbb{P}(Y = 1 \mid \mathbf{X} = \mathbf{x})$.

- Algorithm for Squared loss ERM with link function $g$?
Stochastic gradient method

\[ \nabla_w \left\{ (g(x^Tw) - y)^2 \right\} = 2(g(x^Tw) - y) \cdot g'(x^Tw) \cdot x. \]
Stochastic gradient method

\[ \nabla_w \left\{ (g(x^T w) - y)^2 \right\} = 2(g(x^T w) - y) \cdot g'(x^T w) \cdot x. \]

---

**Stochastic gradient method for squared loss ERM with link function \( g \)**

1: Start with some initial \( w^{(1)} \in \mathbb{R}^d \).
2: **for** \( t = 1, 2, \ldots \) until some stopping condition is satisfied **do**
   3: Pick \((X^{(t)}, Y^{(t)})\) uniformly at random from \((x_1, y_1), \ldots, (x_n, y_n)\).
   4: Update:
      \[ w^{(t+1)} := w^{(t)} - 2\eta_t \cdot (g(\langle X^{(t)}, w^{(t)} \rangle) - Y^{(t)}) \cdot g'(\langle X^{(t)}, w^{(t)} \rangle) \cdot X^{(t)}. \]
5: **end for**
Extensions

- Other loss functions (e.g., $y \ln \frac{1}{p} + (1 - y) \ln \frac{1}{1-p}$).
- Other link functions (e.g., $g(z) = \text{some polynomial in } z$).
Extensions

- Other loss functions (e.g., $y \ln \frac{1}{p} + (1 - y) \ln \frac{1}{1-p}$).
- Other link functions (e.g., $g(z) = \text{some polynomial in } z$).
- Is the overall objective function convex?
Extensions

- Other loss functions (e.g., $y \ln \frac{1}{p} + (1 - y) \ln \frac{1}{1-p}$).
- Other link functions (e.g., $g(z) = \text{some polynomial in } z$).
- Is the overall objective function convex?
  Somtimes, but not always.
Extensions

- Other loss functions (e.g., $y \ln \frac{1}{p} + (1 - y) \ln \frac{1}{1-p}$).
- Other link functions (e.g., $g(z) =$ some polynomial in $z$).
- Is the overall objective function convex?
  Somtimes, but not always.

Nevertheless, stochastic gradient method is still often effective at finding approximate local minima.
2. Multilayer neural networks
Two-output network

\[ v_j := g(z_j), \quad z_j := \sum_{i=1}^{d} W_{i,j} x_i, \quad j \in \{1, 2\}. \]
\( k \)-output network

\[ v_j := g(z_j), \quad z_j := \sum_{i=1}^{d} W_{i,j} x_i, \quad j \in \{1, \ldots, k\}. \]
A motivating example: multitask learning

- $k$ binary prediction tasks with a single feature vector (e.g., predicting tags for images).
  Labeled examples are of the form $(x_i, (y_{i,1}, \ldots, y_{i,k})) \in \mathbb{R}^d \times \{0, 1\}^k$. 

Option 1: $k$ independent logistic regression models; learn $w_1, \ldots, w_k$ by minimizing (e.g.)
\[
1/n \sum_{i=1}^n \sum_{j=1}^k (g(x_i^T w_j) - y_{i,j})^2.
\]

Option 2: Do “Option 1”, but also learn to combine predictions of $y_{i,1}, \ldots, y_{i,k}$ to get better predictions for each $y_{i,j}$. 

A motivating example: multitask learning

- $k$ binary prediction tasks with a single feature vector (e.g., predicting tags for images).
  Labeled examples are of the form $(x_i, (y_{i,1}, \ldots, y_{i,k})) \in \mathbb{R}^d \times \{0, 1\}^k$.

- **Option 1**: $k$ independent logistic regression models; learn $\boldsymbol{w}_1, \ldots, \boldsymbol{w}_k$ by minimizing (e.g.)
  \[
  \frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{k} (g(x_i^T \boldsymbol{w}_j) - y_{i,j})^2.
  \]
A motivating example: multitask learning

- $k$ binary prediction tasks with a single feature vector (e.g., predicting tags for images).
  Labeled examples are of the form $(x_i, (y_{i,1}, \ldots, y_{i,k})) \in \mathbb{R}^d \times \{0, 1\}^k$.

- **Option 1**: $k$ independent logistic regression models; learn $w_1, \ldots, w_k$ by minimizing (e.g.)
  \[
  \frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{k} (g(x_i^T w_j) - y_{i,j})^2.
  \]

- Suppose labels $y_{i,1}, \ldots, y_{i,k}$ are not independent.
  E.g., if $y_{i,1} = 1$, then also more likely that $y_{i,2} = 1$. 

---
A motivating example: multitask learning

- $k$ binary prediction tasks with a single feature vector (e.g., predicting tags for images).
  Labeled examples are of the form $(x_i, (y_{i,1}, \ldots, y_{i,k})) \in \mathbb{R}^d \times \{0, 1\}^k$.

- **Option 1**: $k$ independent logistic regression models; learn $w_1, \ldots, w_k$ by minimizing (e.g.)

  $$
  \frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{k} (g(x_i^T w_j) - y_{i,j})^2.
  $$

- Suppose labels $y_{i,1}, \ldots, y_{i,k}$ are *not* independent.
  E.g., if $y_{i,1} = 1$, then also more likely that $y_{i,2} = 1$.

- **Option 2**: Do “Option 1”, but also learn to combine predictions of $y_{i,1}, \ldots, y_{i,k}$ to get better predictions for each $y_{i,j}$. 
Multilayer neural network

- Columns of $W_1 \in \mathbb{R}^{d \times k}$: params. of original logistic regression models.
- Columns of $W_2 \in \mathbb{R}^{k \times k}$: params. of new logistic regression models to combine predictions of original models.
Multilayer neural network

▶ Columns of $W_1 \in \mathbb{R}^{d \times k}$: params. of original logistic regression models.
▶ Columns of $W_2 \in \mathbb{R}^{k \times k}$: params. of new logistic regression models to combine predictions of original models.
▶ Each node is called a *unit*. 
Multilayer neural network

- Columns of $W_1 \in \mathbb{R}^{d \times k}$: params. of original logistic regression models.
- Columns of $W_2 \in \mathbb{R}^{k \times k}$: params. of new logistic regression models to combine predictions of original models.

- Each node is called a *unit*.
- Non-input and non-output units are called *hidden*.
Compositional structure

Suppose we have two functions

\[ f_{W_1} : \mathbb{R}^d \to \mathbb{R}^k, \quad (W_2 \in \mathbb{R}^{d \times k}), \]
\[ f_{W_2} : \mathbb{R}^k \to \mathbb{R}^\ell, \quad (W_2 \in \mathbb{R}^{k \times \ell}), \]

where

\[ f_W(x) := g(W^T x), \]

and \( g \) applies the link function \( g \) coordinate-wise to a vector.
Suppose we have two functions

\[ f_{W_1} : \mathbb{R}^d \to \mathbb{R}^k, \quad (W_2 \in \mathbb{R}^{d \times k}), \]
\[ f_{W_2} : \mathbb{R}^k \to \mathbb{R}^\ell, \quad (W_2 \in \mathbb{R}^{k \times \ell}), \]

where

\[ f_{W}(x) := g(W^T x), \]

and \( g \) applies the link function \( g \) coordinate-wise to a vector.

Composition: \( f_{W_1, W_2} := f_{W_2} \circ f_{W_1} \) is defined by

\[ f_{W_1, W_2}(x) := f_{W_2}(f_{W_1}(x)). \]
Suppose we have two functions

\[ f_{W_1} : \mathbb{R}^d \to \mathbb{R}^k, \quad (W_2 \in \mathbb{R}^{d \times k}), \]
\[ f_{W_2} : \mathbb{R}^k \to \mathbb{R}^\ell, \quad (W_2 \in \mathbb{R}^{k \times \ell}), \]

where

\[ f_W(x) := g(W^T x), \]

and \( g \) applies the link function \( g \) coordinate-wise to a vector.

**Composition**: \( f_{W_1, W_2} := f_{W_2} \circ f_{W_1} \) is defined by

\[ f_{W_1, W_2}(x) := f_{W_2}(f_{W_1}(x)). \]

This is a **two-layer neural network**.
Necessity of multiple layers

One-layer neural network with a monotonic link function is a linear (or affine) classifier.

Cannot represent XOR function (Minsky and Papert, 1969).

(a) $x_1$ and $x_2$

(b) $x_1$ or $x_2$

(c) $x_1$ xor $x_2$

(Figure from Stuart Russell.)
“**Theorem**” (Cybenko, 1989; Hornik, 1991; Barron, 1993).

*Any continuous function* $f$ *can be approximated arbitrarily well by a two-layer neural network*

$$f \approx f_{W_2} \circ f_{W_1}.$$  

\[ \mathbb{R}^k \rightarrow \mathbb{R} \quad \mathbb{R}^d \rightarrow \mathbb{R}^k \]

*However:* may need a very large number of hidden units.
Approximation power of multilayer neural networks

▶ "Theorem" (Cybenko, 1989; Hornik, 1991; Barron, 1993).

*Any continuous function* \( f \) *can be approximated arbitrarily well by a two-layer neural network*

\[
f \approx f_{W_2} \circ f_{W_1}.
\]

\[\mathbb{R}^k \to \mathbb{R} \quad \mathbb{R}^d \to \mathbb{R}^k\]

*However:* may need a very large number of hidden units.

▶ "Theorem" (Telgarsky, 2015; Eldan and Shamir, 2015).  

*Some functions* can be approximated with *exponentially fewer hidden units* by using more than two layers.
Approximation power of multilayer neural networks

Any continuous function $f$ can be approximated arbitrarily well by a two-layer neural network

$$f \approx f_{W_2} \circ f_{W_1}.$$

$$\mathbb{R}^k \rightarrow \mathbb{R} \quad \mathbb{R}^d \rightarrow \mathbb{R}^k$$

*However:* may need a very large number of hidden units.

▶ “Theorem” (Telgarsky, 2015; Eldan and Shamir, 2015).
Some functions can be approximated with exponentially fewer hidden units by using more than two layers.

Note: none of this speaks directly to *learning* neural networks from data.
3. Computation and learning with neural networks
General structure of neural network

Neural network for $f : \mathbb{R}^d \rightarrow \mathbb{R}$. (Easy to generalize to $f : \mathbb{R}^d \rightarrow \mathbb{R}^k$.)
General structure of neural network

Neural network for $f : \mathbb{R}^d \rightarrow \mathbb{R}$. (Easy to generalize to $f : \mathbb{R}^d \rightarrow \mathbb{R}^k$.)

- Directed acyclic graph $G = (V, E)$; vertices regarded as formal variables.

\[ \hat{y} \]
\[ x_1 \]
\[ x_2 \]
\[ \cdots \]
\[ x_d \]
General structure of neural network

Neural network for $f : \mathbb{R}^d \rightarrow \mathbb{R}$. (Easy to generalize to $f : \mathbb{R}^d \rightarrow \mathbb{R}^k$.)

- Directed acyclic graph $G = (V, E)$; vertices regarded as formal variables.
- $d$ source vertices, one per input variable, called $x_1, \ldots, x_d$. 

![Diagram of a directed acyclic graph with $d$ source vertices and one sink vertex $\hat{y}$]
General structure of neural network

Neural network for \( f : \mathbb{R}^d \rightarrow \mathbb{R} \). (Easy to generalize to \( f : \mathbb{R}^d \rightarrow \mathbb{R}^k \).)

- Directed acyclic graph \( G = (V, E) \); vertices regarded as formal variables.
- \( d \) source vertices, one per input variable, called \( x_1, \ldots, x_d \).
- Single sink vertex, called \( \hat{y} \).
Neural network for $f: \mathbb{R}^d \rightarrow \mathbb{R}$. (Easy to generalize to $f: \mathbb{R}^d \rightarrow \mathbb{R}^k$.)

- Directed acyclic graph $G = (V, E)$; vertices regarded as formal variables.
- $d$ source vertices, one per input variable, called $x_1, \ldots, x_d$.
- Single sink vertex, called $\hat{y}$.
- Each edge $(u, v) \in E$ has a weight $w_{u,v} \in \mathbb{R}$. 

\[ x_1 \quad x_2 \quad \cdots \quad x_d \]

\[ \hat{y} \quad u \quad v \]

\[ w_{u,v} \]

\[ g(z_v) = \sum_{u \in \pi_G(v)} w_{u,v} \cdot u. \] (g is link function, e.g., logistic function.)
General structure of neural network

Neural network for $f : \mathbb{R}^d \rightarrow \mathbb{R}$. (Easy to generalize to $f : \mathbb{R}^d \rightarrow \mathbb{R}^k$.)

- Directed acyclic graph $G = (V, E)$; vertices regarded as formal variables.
- $d$ source vertices, one per input variable, called $x_1, \ldots, x_d$.
- Single sink vertex, called $\hat{y}$.
- Each edge $(u, v) \in E$ has a weight $w_{u,v} \in \mathbb{R}$.

Value of vertex $v$ given values of parents $\pi_G(v) := \{u \in V : (u, v) \in E\}$ is

$$v := g(z_v), \quad z_v := \sum_{u \in \pi_G(v)} w_{u,v} \cdot u.$$  

($g$ is link function, e.g., logistic function.)
Organizing and evaluating a neural network

Vertices in $V$ partitioned into layers $V_0, V_1, \ldots$. 

$\text{V}_0 := \{x_1, \ldots, x_d\}$, just the input variables.

Put $v$ in $V_l$ if longest path in $G$ from some $x_i$ to $v$ has $l$ edges.

Final layer only consists of sink vertex, $\hat{y}$.

Given input values $x = (x_1, \ldots, x_d) \in \mathbb{R}^d$, how to compute $f(x)$?

1. Compute values of all vertices in $V_1$, given values of vertices in $V_0$ (i.e., input variables).

$v := g(z_v)$, $z_v := \sum_{u \in \pi_G(v)} w_{u,v} \cdot u$.

(All parents of $v \in V_1$ are in $V_0$.)

2. Compute values of all vertices in $V_2$, given values of vertices in $V_0 \cup V_1$.

(All parents of $v \in V_2$ are in $V_0 \cup V_1$.)

3. Etc., until $\hat{y} = f(x)$ is computed.

This is called forward propagation.
Organizing and evaluating a neural network

Vertices in $V$ partitioned into layers $V_0, V_1, \ldots$.

- $V_0 := \{x_1, \ldots, x_d\}$, just the input variables.

Given input values $x = (x_1, \ldots, x_d) \in \mathbb{R}^d$, how to compute $f(x)$?

1. Compute values of all vertices in $V_1$, given values of vertices in $V_0$ (i.e., input variables).
   
   $v := g(z_v)$, $z_v := \sum_{u \in \pi_G(v)} w_{u,v} \cdot u$.
   
   (All parents of $v \in V_1$ are in $V_0$.)

2. Compute values of all vertices in $V_2$, given values of vertices in $V_0 \cup V_1$.
   
   (All parents of $v \in V_2$ are in $V_0 \cup V_1$.)

3. Etc., until $\hat{y} = f(x)$ is computed.

This is called forward propagation.
Organizing and evaluating a neural network

Vertices in $V$ partitioned into layers $V_0, V_1, \ldots$

- $V_0 := \{x_1, \ldots, x_d\}$, just the input variables.
- Put $v$ in $V_l$ if longest path in $G$ from some $x_i$ to $v$ has $l$ edges.

Given input values $x = (x_1, \ldots, x_d) \in \mathbb{R}^d$, how to compute $f(x)$?

1. Compute values of all vertices in $V_1$, given values of vertices in $V_0$ (i.e., input variables).
   $$v := g(z_v)$$
   $$z_v := \sum_{u \in \pi_G(v)} w_{u,v} \cdot u.$$ (All parents of $v \in V_1$ are in $V_0$.)

2. Compute values of all vertices in $V_2$, given values of vertices in $V_0 \cup V_1$.
   (All parents of $v \in V_2$ are in $V_0 \cup V_1$.)

3. Etc., until $\hat{y} = f(x)$ is computed.

This is called forward propagation.
Organizing and evaluating a neural network

Vertices in $V$ partitioned into layers $V_0, V_1, \ldots$.

- $V_0 := \{x_1, \ldots, x_d\}$, just the input variables.
- Put $v$ in $V_l$ if longest path in $G$ from some $x_i$ to $v$ has $l$ edges.
- Final layer only consists of sink vertex, $\hat{y}$.

Given input values $x = (x_1, \ldots, x_d) \in \mathbb{R}^d$, how to compute $f(x)$?

1. Compute values of all vertices in $V_1$, given values of vertices in $V_0$ (i.e., input variables).

   $v := g(z_v)$, $z_v := \sum_{u \in \pi_G(v)} w_{u,v} \cdot u$.

   (All parents of $v \in V_1$ are in $V_0$.)

2. Compute values of all vertices in $V_2$, given values of vertices in $V_0 \cup V_1$.

   (All parents of $v \in V_2$ are in $V_0 \cup V_1$.)

3. Etc., until $\hat{y} = f(x)$ is computed.

This is called forward propagation.
Organizing and evaluating a neural network

Vertices in $V$ partitioned into layers $V_0, V_1, \ldots$.

- $V_0 := \{x_1, \ldots, x_d\}$, just the input variables.
- Put $v$ in $V_l$ if longest path in $G$ from some $x_i$ to $v$ has $l$ edges.
- Final layer only consists of sink vertex, $\hat{y}$.

Given input values $\mathbf{x} = (x_1, \ldots, x_d) \in \mathbb{R}^d$, how to compute $f(\mathbf{x})$?
Organizing and evaluating a neural network

Vertices in $V$ partitioned into layers $V_0, V_1, \ldots$.

- $V_0 := \{x_1, \ldots, x_d\}$, just the input variables.
- Put $v$ in $V_l$ if longest path in $G$ from some $x_i$ to $v$ has $l$ edges.
- Final layer only consists of sink vertex, $\hat{y}$.

Given input values $\mathbf{x} = (x_1, \ldots, x_d) \in \mathbb{R}^d$, how to compute $f(\mathbf{x})$?

1. Compute values of all vertices in $V_1$, given values of vertices in $V_0$ (i.e., input variables).

$$v := g(z_v), \quad z_v := \sum_{u \in \pi_G(v)} w_{u,v} \cdot u.$$  

(All parents of $v \in V_1$ are in $V_0$.)
Organizing and evaluating a neural network

Vertices in $V$ partitioned into layers $V_0, V_1, \ldots$.

- $V_0 := \{x_1, \ldots, x_d\}$, just the input variables.
- Put $v$ in $V_l$ if longest path in $G$ from some $x_i$ to $v$ has $l$ edges.
- Final layer only consists of sink vertex, $\hat{y}$.

Given input values $\mathbf{x} = (x_1, \ldots, x_d) \in \mathbb{R}^d$, how to compute $f(x)$?

1. Compute values of all vertices in $V_1$, given values of vertices in $V_0$ (i.e., input variables).

   \[ v := g(z_v), \quad z_v := \sum_{u \in \pi_G(v)} w_{u,v} \cdot u. \]

   (All parents of $v \in V_1$ are in $V_0$.)

2. Compute values of all vertices in $V_2$, given values of vertices in $V_0 \cup V_1$. (All parents of $v \in V_2$ are in $V_0 \cup V_1$.)
Organizing and evaluating a neural network

Vertices in $V$ partitioned into layers $V_0, V_1, \ldots$.

- $V_0 := \{x_1, \ldots, x_d\}$, just the input variables.
- Put $v$ in $V_l$ if longest path in $G$ from some $x_i$ to $v$ has $l$ edges.
- Final layer only consists of sink vertex, $\hat{y}$.

Given input values $x = (x_1, \ldots, x_d) \in \mathbb{R}^d$, how to compute $f(x)$?

1. Compute values of all vertices in $V_1$, given values of vertices in $V_0$ (i.e., input variables).

$$v := g(z_v), \quad z_v := \sum_{u \in \pi_G(v)} w_{u,v} \cdot u.$$  

(All parents of $v \in V_1$ are in $V_0$.)

2. Compute values of all vertices in $V_2$, given values of vertices in $V_0 \cup V_1$. (All parents of $v \in V_2$ are in $V_0 \cup V_1$.)

3. Etc., until $\hat{y} = f(x)$ is computed.
Organizing and evaluating a neural network

Vertices in $V$ partitioned into layers $V_0, V_1, \ldots$.

- $V_0 := \{x_1, \ldots, x_d\}$, just the input variables.
- Put $v$ in $V_l$ if longest path in $G$ from some $x_i$ to $v$ has $l$ edges.
- Final layer only consists of sink vertex, $\hat{y}$.

Given input values $x = (x_1, \ldots, x_d) \in \mathbb{R}^d$, how to compute $f(x)$?

1. Compute values of all vertices in $V_1$, given values of vertices in $V_0$ (i.e., input variables).
   
   $$v := g(z_v), \quad z_v := \sum_{u \in \pi_G(v)} w_{u,v} \cdot u.$$  
   (All parents of $v \in V_1$ are in $V_0$.)

2. Compute values of all vertices in $V_2$, given values of vertices in $V_0 \cup V_1$.  
   (All parents of $v \in V_2$ are in $V_0 \cup V_1$.)

3. Etc., until $\hat{y} = f(x)$ is computed.

This is called forward propagation.
How to fit a neural network to data?

Use stochastic gradient method!

Basic computational problem: compute partial derivative of loss on a labeled example with respect to a parameter.

Mid-to-late 20th century researchers discovered how to use chain rule to organize gradient computation: backpropagation algorithm.

Given:
- labeled example \((x, y) \in \mathbb{R}^d \times \mathbb{R}\);
- current weights \(w_{u,v} \in \mathbb{R}\) for all \((u,v) \in E\);
- values \(v\) and \(z\_v\) for all non-source \(v \in V\) on input \(x\).

(Can first run forward propagation to get \(v\)'s and \(z\_v\)'s.)

Let \(\ell\) denote loss of prediction \(\hat{y} = f(x)\) (e.g., \(\ell := (\hat{y} - y)^2\)).

Goal: Compute \(\frac{\partial \ell}{\partial w_{u,v}}\), \((u,v) \in E\).
How to fit a neural network to data? Use stochastic gradient method!
Training a neural network

How to fit a neural network to data? Use stochastic gradient method!

**Basic computational problem**: compute *partial derivative* of loss on a labeled example with respect to a parameter.
How to fit a neural network to data? Use stochastic gradient method!

**Basic computational problem**: compute *partial derivative* of loss on a labeled example with respect to a parameter.

Mid-to-late 20th century researchers discovered how to use chain rule to organize gradient computation: *backpropagation algorithm.*
How to fit a neural network to data? Use stochastic gradient method!

**Basic computational problem:** compute *partial derivative* of loss on a labeled example with respect to a parameter.

Mid-to-late 20th century researchers discovered how to use chain rule to organize gradient computation: *backpropagation algorithm.*

- **Given:** labeled example \((x, y) \in \mathbb{R}^d \times \mathbb{R};\)
  - current weights \(w_{u,v} \in \mathbb{R}\) for all \((u, v) \in E;\)
How to fit a neural network to data? Use stochastic gradient method!

**Basic computational problem:** compute *partial derivative* of loss on a labeled example with respect to a parameter.

Mid-to-late 20th century researchers discovered how to use chain rule to organize gradient computation: *backpropagation algorithm*.

- **Given:** labeled example \((x, y) \in \mathbb{R}^d \times \mathbb{R}\); current weights \(w_{u,v} \in \mathbb{R}\) for all \((u, v) \in E\); values \(v\) and \(z_v\) for all non-source \(v \in V\) on input \(x\).

  (Can first run *forward propagation* to get \(v\)'s and \(z_v\)'s.)
Training a neural network

How to fit a neural network to data? Use stochastic gradient method!

**Basic computational problem:** compute *partial derivative* of loss on a labeled example with respect to a parameter.

Mid-to-late 20th century researchers discovered how to use chain rule to organize gradient computation: *backpropagation algorithm*.

- **Given:** labeled example \((x, y) \in \mathbb{R}^d \times \mathbb{R};\)
- current weights \(w_{u,v} \in \mathbb{R}\) for all \((u, v) \in E;\)
- values \(v\) and \(z_v\) for all non-source \(v \in V\) on input \(x\).

(Can first run forward propagation to get \(v\)'s and \(z_v\)'s.)

- Let \(\ell\) denote loss of prediction \(\hat{y} = f(x)\) (e.g., \(\ell := (\hat{y} - y)^2\)).
How to fit a neural network to data? Use stochastic gradient method!

**Basic computational problem**: compute *partial derivative* of loss on a labeled example with respect to a parameter.

Mid-to-late 20th century researchers discovered how to use chain rule to organize gradient computation: *backpropagation algorithm*.

- **Given**: labeled example \((x, y) \in \mathbb{R}^d \times \mathbb{R}^d\); current weights \(w_{u,v} \in \mathbb{R}\) for all \((u, v) \in E\); values \(v\) and \(z_v\) for all non-source \(v \in V\) on input \(x\).

  (Can first run forward propagation to get \(v\)'s and \(z_v\)'s.)

- Let \(\ell\) denote loss of prediction \(\hat{y} = f(x)\) (e.g., \(\ell := (\hat{y} - y)^2\)).

- **Goal**: Compute

\[
\frac{\partial \ell}{\partial w_{u,v}}, \quad (u, v) \in E.
\]
Backpropagation: exploiting the chain rule

**Strategy**: use chain rule.

\[
\frac{\partial \ell}{\partial w_{u,v}} = \frac{\partial \ell}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial v} \cdot \frac{\partial v}{\partial w_{u,v}}.
\]

For squared loss \( \ell = (\hat{y} - y)^2 \),

\[\frac{\partial \ell}{\partial \hat{y}} = 2(\hat{y} - y).\]

Easy to compute with other losses as well. (\(\hat{y}\) is computed in forward propagation.)

Since \(v = g(z_v)\) where \(z_v = w_{u,v} \cdot u + \text{terms not involving } w_{u,v}\),

\[
\frac{\partial v}{\partial w_{u,v}} = \frac{\partial v}{\partial z_v} \cdot \frac{\partial z_v}{\partial w_{u,v}} = g'(z_v) \cdot u.
\]

(\(z_v\) and \(u\) are computed in forward propagation.)
Backpropagation: exploiting the chain rule

**Strategy**: use chain rule.

\[
\frac{\partial \ell}{\partial w_{u,v}} = \frac{\partial \ell}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial v} \cdot \frac{\partial v}{\partial w_{u,v}}.
\]

▶ For squared loss \( \ell = (\hat{y} - y)^2 \),

\[
\frac{\partial \ell}{\partial \hat{y}} = 2(\hat{y} - y).
\]

Easy to compute with other losses as well.

(\( \hat{y} \) is computed in forward propagation.)
Backpropagation: exploiting the chain rule

**Strategy:** use chain rule.

\[
\frac{\partial \ell}{\partial w_{u,v}} = \frac{\partial \ell}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial v} \cdot \frac{\partial v}{\partial w_{u,v}}.
\]

- For squared loss \( \ell = (\hat{y} - y)^2 \),

\[
\frac{\partial \ell}{\partial \hat{y}} = 2(\hat{y} - y).
\]

Easy to compute with other losses as well.
(\( \hat{y} \) is computed in forward propagation.)

- Since \( v = g(z_v) \) where \( z_v = w_{u,v} \cdot u + \text{(terms not involving } w_{u,v}) \),

\[
\frac{\partial v}{\partial w_{u,v}} = \frac{\partial v}{\partial z_v} \cdot \frac{\partial z_v}{\partial w_{u,v}} = g'(z_v) \cdot u.
\]

(\( z_v \) and \( u \) are computed in forward propagation.)
Key trick: compute $\frac{\partial \hat{y}}{\partial v}$ for all $v \in V_l$, in decreasing order of layer $l$. 

Since $v_i = g(z_{v_i})$ where $z_{v_i} = w_{v,v_i} \cdot v + (\text{terms not involving } v)$, 

$$\frac{\partial v_i}{\partial v} = g'(z_{v_i}) \cdot w_{v,v_i}.$$ 

(The $z_{v_i}$'s are computed in forward propagation.) 

Since $v_i$ are in a higher layer than $v$, $\frac{\partial \hat{y}}{\partial v_i}$ has already been computed!
Backpropagation: the recursive part

**Key trick:** compute $\frac{\partial \hat{y}}{\partial v}$ for all $v \in V_l$, in decreasing order of layer $l$.

**Strategy:** for $v \neq \hat{y}$, use multivariate chain rule.

Let $k = \text{out-deg}(v)$, $(v, v_1), \ldots, (v, v_k) \in E$:

$$\frac{\partial \hat{y}}{\partial v} = \sum_{i=1}^{k} \frac{\partial \hat{y}}{\partial v_i} \cdot \frac{\partial v_i}{\partial v}.$$
Backpropagation: the recursive part

**Key trick:** compute \( \frac{\partial \hat{y}}{\partial v} \) for all \( v \in V_l \), in decreasing order of layer \( l \).

**Strategy:** for \( v \neq \hat{y} \), use multivariate chain rule.

Let \( k = \text{out-deg}(v) \), \((v, v_1), \ldots, (v, v_k) \in E:\)

\[
\frac{\partial \hat{y}}{\partial v} = \sum_{i=1}^{k} \frac{\partial \hat{y}}{\partial v_i} \cdot \frac{\partial v_i}{\partial v}.
\]

- Since \( v_i = g(z_{v_i}) \) where \( z_{v_i} = w_{v,v_i} \cdot v + \text{(terms not involving } v) \),
  
  \[
  \frac{\partial v_i}{\partial v} = g'(z_{v_i}) \cdot w_{v,v_i}.
  \]

  (The \( z_{v_i} \)'s are computed in forward propagation.)
Backpropagation: the recursive part

**Key trick:** compute $\frac{\partial \hat{y}}{\partial v}$ for all $v \in V_l$, in decreasing order of layer $l$.

**Strategy:** for $v \neq \hat{y}$, use multivariate chain rule.

Let $k = \text{out-deg}(v)$, $(v, v_1), \ldots, (v, v_k) \in E$:

$$\frac{\partial \hat{y}}{\partial v} = \sum_{i=1}^{k} \frac{\partial \hat{y}}{\partial v_i} \cdot \frac{\partial v_i}{\partial v}.$$  

- Since $v_i = g(z_{v_i})$ where $z_{v_i} = w_{v,v_i} \cdot v + \text{(terms not involving } v)$,
  $$\frac{\partial v_i}{\partial v} = g'(z_{v_i}) \cdot w_{v,v_i}.$$  
  (The $z_{v_i}$’s are computed in forward propagation.)

- Since $v_i$ are in a *higher layer* than $v$, $\frac{\partial \hat{y}}{\partial v_i}$ has already been computed!
Matrix view of forward/backward propagation

- Recall general neural network function $f : \mathbb{R}^d \rightarrow \mathbb{R}$ can be written as

$$f(x) = g_L(W_L \cdots g_2(W_2 g_1(W_1 x)) \cdots),$$

where $W_i \in \mathbb{R}^{d_i \times d_{i-1}}$ is weight matrix for layer $i$, and $g_i : \mathbb{R}^{d_i} \rightarrow \mathbb{R}^{d_i}$ collects the non-linear link functions for layer $i$. 

Previous backprop equations prove correctness of the following matrix derivative formula:

$$\frac{\partial \ell}{\partial W_i} = \left[ g'_L(z_L) \circ W'_L + \cdots + g'_{L-1}(z_{L-1}) \circ W'_{L-1}(g'_L(z_L) \ell'(f_L(x))) \right] f_{i-1}(x)^T$$

where $\circ$ denotes element-wise product.
Matrix view of forward/backward propagation

- Recall general neural network function $f: \mathbb{R}^d \rightarrow \mathbb{R}$ can be written as
  \[
  f(x) = g_L(W_L \cdots g_2(W_2 g_1(W_1 x)) \cdots),
  \]
  where $W_i \in \mathbb{R}^{d_i \times d_{i-1}}$ is weight matrix for layer $i$, and $g_i : \mathbb{R}^{d_i} \rightarrow \mathbb{R}^{d_i}$ collects the non-linear link functions for layer $i$.

- The first layer thus computes
  \[
  f_1(x) := g_1(z_1) \quad \text{where} \quad z_1 := W_1 x,
  \]
  and $i$-th layer computes
  \[
  f_i(x) := g_i(z_i) \quad \text{where} \quad z_i := W_i f_{i-1}(x).
  \]
  (Convention: $f_0(x) := x$.)
Matrix view of forward/backward propagation

- Recall general neural network function $f : \mathbb{R}^d \rightarrow \mathbb{R}$ can be written as

$$f(x) = g_L(W_L \cdots g_2(W_2 g_1(W_1 x)) \cdots),$$

where $W_i \in \mathbb{R}^{d_i \times d_{i-1}}$ is weight matrix for layer $i$,
and $g_i : \mathbb{R}^{d_i} \rightarrow \mathbb{R}^{d_i}$ collects the non-linear link functions for layer $i$.

- The first layer thus computes

$$f_1(x) := g_1(z_1) \quad \text{where} \quad z_1 := W_1 x,$$

and $i$-th layer computes

$$f_i(x) := g_i(z_i) \quad \text{where} \quad z_i := W_i f_{i-1}(x).$$

(Convention: $f_0(x) := x$.)

- Previous backprop equations prove correctness of the following matrix derivative formula:

$$\frac{\partial \ell}{\partial W_i} = \left[ g'_i(z_i) \odot W_{i+1}^\top \left( \cdots g'_{L-1}(z_{L-1}) \odot W_L^\top \left( g'_L(z_L) \ell'(f_L(x)) \right) \right) \right] f_{i-1}(x)^\top$$

where $\odot$ denotes element-wise product.
Practical tips

- Apply stochastic gradient method to examples in random order.

  Can use several examples to form gradient estimate: *mini-batch.*

\[
\lambda^{(t)} := \frac{1}{b} \sum_{i=(t-1)b+1}^{tb} \left. \frac{\partial \ell_i}{\partial W} \right|_{W=W^{(t)}}
\]

where \( \ell_i \) is loss on \( i \)-th training example.
Practical tips

- Apply stochastic gradient method to examples in random order.

Can use several examples to form gradient estimate: *mini-batch*.

\[ \lambda^{(t)} := \frac{1}{b} \sum_{i=(t-1)b+1}^{tb} \frac{\partial \ell_i}{\partial W} \bigg|_{W=W(t)} \]

where \( \ell_i \) is loss on \( i \)-th training example.

- Standardize inputs (i.e., center and divide by standard deviation).

Doing this at every layer during training: *batch normalization*.

(Must also apply same/similar normalization at test time.)
Practical tips

- Apply stochastic gradient method to examples in random order.

  Can use several examples to form gradient estimate: *mini-batch*.

  $$\lambda^{(t)} := \frac{1}{b} \sum_{i=(t-1)b+1}^{tb} \frac{\partial \ell_i}{\partial W} \bigg|_{W=W^{(t)}}$$

  where $\ell_i$ is loss on $i$-th training example.

- Standardize inputs (i.e., center and divide by standard deviation).

  Doing this at every layer during training: *batch normalization*. (Must also apply same/similar normalization at test time.)

- **Initialization:**

  Take care so initial weights not too large or small.

  E.g., for node with $d$ inputs, draw weights iid from $N(0, 1/d)$.
Modern networks

- Two kinds of linear layers:
  - “dense” / “fully-connected” layers $W_i f_{i-1}(x)$ as before,
  - *convolutional layers*, which have a special sparse representation.
Modern networks

- Two kinds of linear layers:
  - “dense” / “fully-connected” layers $W_i f_{i-1}(x)$ as before,
  - convolutional layers, which have a special sparse representation.

- Common link functions:
  - ReLU $z_i \mapsto \max \{0, z_i\}$ (applied element-wise)
  - SoftMax $z_i \mapsto \frac{e^{z_i}}{\sum_j e^{z_j}}$ (applied element-wise)
  - max-pooling $z \mapsto \max_i z_i$ (after convolution layers; $\mathbb{R}^d \to \mathbb{R}$),
Modern networks

- Two kinds of linear layers:
  - “dense” / “fully-connected” layers $W_i f_{i-1}(x)$ as before,
  - convolutional layers, which have a special sparse representation.

- Common link functions:
  - ReLU $z_i \mapsto \max\{0, z_i\}$ (applied element-wise)
  - SoftMax $z_i \mapsto \frac{e^{z_i}}{\sum_j e^{z_j}}$ (applied element-wise)
  - max-pooling $z \mapsto \max_i z_i$ (after convolution layers; $\mathbb{R}^d \to \mathbb{R}$),

- Trend in architectures (in some domains, like vision and speech):
  - Few convolutional layers then many dense layers (AlexNet)
  - More convolutional layers (VGGNet)
  - Nearly purely convolutional layers in many (100+) layers with variety of identity connections throughout (ResNet, DenseNet).
Modern networks

- Two kinds of linear layers:
  - “dense” / “fully-connected” layers $W_i f_{i-1}(x)$ as before,
  - convolutional layers, which have a special sparse representation.

- Common link functions:
  - ReLU $z_i \mapsto \max\{0, z_i\}$ (applied element-wise)
  - SoftMax $z_i \mapsto \frac{e^{z_i}}{\sum_j e^{z_j}}$ (applied element-wise)
  - max-pooling $z \mapsto \max_i z_i$ (after convolution layers; $\mathbb{R}^d \to \mathbb{R}$),

- Trend in architectures (in some domains, like vision and speech):
  - Few convolutional layers then many dense layers (AlexNet)
  - More convolutional layers (VGGNet)
  - Nearly purely convolutional layers in many (100+) layers with variety of identity connections throughout (ResNet, DenseNet).

- Can use intermediate computation (e.g., $f_i(x)$) as feature expansion!

Indispensable in visual and audio tasks; application attempts are constant in all other disciplines.
Frontier of experimental machine learning research.
Wrap-up

- Frontier of experimental machine learning research.
- Basic ideas same as from the 1980s.
Wrap-up

- Frontier of experimental machine learning research.
- Basic ideas same as from the 1980s.
- What’s new?
  - Fast hardware.

- Fast hardware.
- Tons of data.
- Many-layered networks (made possible through many adjustments).
  - Applications: visual detection and recognition, speech recognition, general function fitting (e.g., learning “reward” functions of different actions of video games), etc.
Wrap-up

- Frontier of experimental machine learning research.
- Basic ideas same as from the 1980s.
- What’s new?
  - Fast hardware.
  - Tons of data.

- Many-layered networks (made possible through many adjustments).
- Applications: visual detection and recognition, speech recognition, general function fitting (e.g., learning “reward” functions of different actions of video games), etc.
- In cutting-edge applications, training is still delicate.
Wrap-up

- Frontier of experimental machine learning research.
- Basic ideas same as from the 1980s.
- What’s new?
  - Fast hardware.
  - Tons of data.
  - Many-layered networks (made possible through many adjustments).
Wrap-up

- Frontier of experimental machine learning research.
- Basic ideas same as from the 1980s.
- What’s new?
  - Fast hardware.
  - Tons of data.
  - Many-layered networks (made possible through many adjustments).
  - Applications: visual detection and recognition, speech recognition, general function fitting (e.g., learning “reward” functions of different actions of video games), etc.
Wrap-up

- Frontier of experimental machine learning research.
- Basic ideas same as from the 1980s.
- What’s new?
  - Fast hardware.
  - Tons of data.
  - Many-layered networks (made possible through many adjustments).
  - Applications: visual detection and recognition, speech recognition, general function fitting (e.g., learning “reward” functions of different actions of video games), etc.
- In cutting-edge applications, training is still delicate.
Key takeaways

1. Structure of neural networks; concept of link functions.
3. Forward and backward propagation algorithms.