Neural networks
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:

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

with

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

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

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 $\mathbf{w}$ 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.
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),

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

where $\eta(\mathbf{x}) = \mathbb{P}(Y = 1 \mid \mathbf{X} = \mathbf{x})$. 
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.
- Another option: Squared loss ERM (with link function $g$)

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

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

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

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

- Algorithm for Squared loss ERM with link function $g$?
\[ \nabla_w \left\{ (g(x^T w) - y)^2 \right\} = 2(g(x^T w) - y) \cdot g'(x^T w) \cdot x. \]
Stochastic gradient method

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

---

**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?
  Sometimes, 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., \( g(z) = \text{some polynomial in } z \)).
- **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

\[ v_j := g(z_j), \quad z_j := \sum_{i=1}^{d} W_{i,j} x_i, \quad j \in \{1, 2\}. \]
$v_j := g(z_j), \quad z_j := \sum_{i=1}^{d} W_{i,j} x_i, \quad j \in \{1, \ldots, k\}.$
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 $(x_i, (y_{i,1}, \ldots, y_{i,k})) \in \mathbb{R}^d \times \{0, 1\}^k$. 
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.)
\[
\frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{k} (g(x_i^T w_j) - y_{i,j})^2.
\]
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.)
  \[
  \frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{k} (g(x_i^Tw_j) - y_{i,j})^2.
  \]

- **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 $y_{i,j}$.

- 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 $\mathbf{W}^{(1)} \in \mathbb{R}^{d \times k}$: params. of original logistic regression models.
- Columns of $\mathbf{W}^{(2)} \in \mathbb{R}^{k \times k}$: 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 $\mathbf{W}^{(1)} \in \mathbb{R}^{d \times k}$: params. of original logistic regression models.
- Columns of $\mathbf{W}^{(2)} \in \mathbb{R}^{k \times k}$: 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*.
Suppose we have two functions

\[ f_{W(1)} : \mathbb{R}^d \to \mathbb{R}^k, \quad (W^{(1)} \in \mathbb{R}^{d \times k}), \]
\[ f_{W(2)} : \mathbb{R}^k \to \mathbb{R}^\ell, \quad (W^{(2)} \in \mathbb{R}^{k \times \ell}), \]

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

\[ f_{\mathbf{W}(1)} : \mathbb{R}^d \to \mathbb{R}^k, \quad (\mathbf{W}(1) \in \mathbb{R}^{d \times k}), \]
\[ f_{\mathbf{W}(2)} : \mathbb{R}^k \to \mathbb{R}^\ell, \quad (\mathbf{W}(2) \in \mathbb{R}^{k \times \ell}), \]

where

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

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

**Composition:** \( f_{\mathbf{W}(1),\mathbf{W}(2)} := f_{\mathbf{W}(2)} \circ f_{\mathbf{W}(1)} \) is defined by

\[ f_{\mathbf{W}(1),\mathbf{W}(2)}(x) := f_{\mathbf{W}(2)}(f_{\mathbf{W}(1)}(x)). \]
Compositional structure

- Suppose we have two functions
  \[ f_{W(1)} : \mathbb{R}^d \to \mathbb{R}^k, \quad (W^{(1)} \in \mathbb{R}^{d \times k}), \]
  \[ f_{W(2)} : \mathbb{R}^k \to \mathbb{R}^\ell, \quad (W^{(2)} \in \mathbb{R}^{k \times \ell}), \]

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

\[ f_{W(1),W(2)}(x) := f_{W(2)} \left( f_{W(1)}(x) \right). \]

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).

(Figure from Stuart Russell.)
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
  
  $$f \approx f_{W^{(2)}} \circ f_{W^{(1)}}.$$
  
  \[ \mathbb{R}^k \rightarrow \mathbb{R} \quad \mathbb{R}^d \rightarrow \mathbb{R}^k \]

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

Note: none of this speaks directly to learning neural networks from data.
Approximation power of multilayer neural networks


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

\[
f \approx f_{W^{(2)}} \circ f_{W^{(1)}}.
\]

\[
\begin{align*}
\mathbb{R}^k &\rightarrow \mathbb{R} \\
\mathbb{R}^d &\rightarrow \mathbb{R}^k
\end{align*}
\]

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.
“**Theorem**” (Cybenko, 1989; Hornik, 1991; Barron, 1993).

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

$$f \approx f_{W^{(2)}} \circ f_{W^{(1)}}.$$

*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.

\[
\text{Value of vertex } v \text{ given values of parents } \pi_G(v) := \{ u \in V : (u, v) \in E \} \text{ is } v := g(z_v), \quad z_v := \sum_{u \in \pi_G(v)} w_{u,v} u.
\]

\( g \) 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 $x_1, \ldots, x_d$. 

![Diagram of a neural network with directed edges and a single sink vertex \( \hat{y} \).]
General structure of neural network

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

- 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 \( \hat{y} \).
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 \( w_{u,v} \in \mathbb{R}\).
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 $V_0, V_1, \ldots$.
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.
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.

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

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

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, $\hat{y}$.

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*. 
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 $f(x)$?
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).

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

   (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 $x = (x_1, \ldots, x_d) \in \mathbb{R}^d$, 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 $\mathbf{x} = (x_1, \ldots, x_d) \in \mathbb{R}^d$, 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 $V_0$.)

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 $\hat{y} = f(\mathbf{x})$ is computed.

This is called \textit{forward propagation}.
Training a neural network

How to fit a neural network to data?
How to fit a neural network to data? Use stochastic gradient method!
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 \((u, v) \in E;\)
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.)
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

\[
\frac{\partial \ell}{\partial w_{u,v}}, \quad (u, v) \in E.
\]
**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.)
**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.)
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 \( z_v = w_{u,v} \cdot u + \text{(terms not involving } u) \),

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

\( z_v \) and \( u \) are computed in forward propagation.
Backpropagation: the recursive part

**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 $v \in V_i$, 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!
Practical tips

- Apply stochastic gradient method to examples in random order. (Totally unclear if fancier methods, like “Adam”, work any better.)
- Standardize inputs (i.e., center and divide by standard deviation).
- Random initialization: Take care so weights are not too large or small. E.g., for node with \(d\) inputs, draw weights iid from \(N(0, 1/d)\).
Practical tips

▶ Apply stochastic gradient method to examples in random order.
  (Totally unclear if fancier methods, like “Adam”, work any better.)
▶ Standardize inputs (i.e., center and divide by standard deviation).
Practical tips

- Apply stochastic gradient method to examples in random order. (Totally unclear if fancier methods, like “Adam”, work any better.)
- Standardize inputs (i.e., center and divide by standard deviation).
- Random initialization: Take care so weights are not too large or small. E.g., for node with $d$ inputs, draw weights iid from $\mathcal{N}(0, 1/d)$. 
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?
  - Designing complex “architectures” that work well for specific problems.
Wrap-up

- Frontier of experimental machine learning research.
- Basic ideas same as from the 1980s.
- What’s new?
  - Designing complex “architectures” that work well for specific problems.
  - Fast algorithms / hardware for optimizing neural network weights on large data sets (e.g., using stream-based processing, distributed systems).
Wrap-up

- Frontier of experimental machine learning research.
- Basic ideas same as from the 1980s.
- What’s new?
  - Designing complex “architectures” that work well for specific problems.
  - Fast algorithms / hardware for optimizing neural network weights on large data sets (e.g., using stream-based processing, distributed systems).
  - Applications: visual detection and recognition, speech recognition, general function fitting (e.g., learning “reward” functions of different actions of video games), etc.
- ...
Key takeaways

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