Deep Learning/Double Descent

Gilbert Strang

MIT

October, 2019
Fit training data by a Learning function $F$

We are given training data: Inputs $v$, outputs $w$

**Example**  Each $v$ is an image of a number $w = 0, 1, \ldots, 9$

The vector $v$ describes each pixel in the image

We want to create a learning function so that $F(v) \approx w$

[math.mit.edu/learningfromdata](http://math.mit.edu/learningfromdata)
Deep Neural Networks

1. Key operation: Composition \( F = F_3(F_2(F_1(x, v_0))) \)
2. Key rule: Chain rule for \( x \)-derivatives of \( F \)
3. Key algorithm: Stochastic gradient descent to find \( x \)
4. Key subroutine: Backpropagation to compute \( \text{grad } F \)
5. Key nonlinearity: \( \text{ReLU}(y) = \max(y, 0) = \text{ramp function} \)
Deep Neural Networks

1. Key operation: Composition $F = F_3(F_2(F_1(x, v_0)))$
2. Key rule: Chain rule for $x$-derivatives of $F$
3. Key algorithm: Stochastic gradient descent to find $x$
4. Key subroutine: Backpropagation to compute grad $F$
5. Key nonlinearity: ReLU ($y = \max(y, 0)$ = ramp function

Layer $k$ \quad $v_k = F_k(v_{k-1}) = \text{ReLU} \left( A_k v_{k-1} + b_k \right)$
Deep Neural Networks

Key operation: Composition \( F = F_3(F_2(F_1(x, v_0))) \)

Key rule: Chain rule for \( x \)-derivatives of \( F \)

Key algorithm: Stochastic gradient descent to find \( x \)

Key subroutine: Backpropagation to compute grad \( F \)

Key nonlinearity: ReLU (\( y \)) = \( \max(y, 0) \) = ramp function

Layer \( k \) \( v_k = F_k(v_{k-1}) = \text{ReLU} (A_k v_{k-1} + b_k) \)

Weights for layer \( k \) \( A_k = \text{matrix} \) and \( b_k = \text{offset vector} \)

\( v_0 = \text{training data} / v_1, \ldots, v_{\ell-1} \text{ hidden layers} / v_{\ell} = \text{output} \)
Feature vector $v_0$
Three components for each training sample

$y_1 = A_1 v_0 + b_1$
$y_1$ at layer 1
Four components of $y_1$ and $v_1$

$v_1 = \text{ReLU} (y_1)$
$v_1$ at layer 1

Output $w = v_2$
$v_2 = A_2 v_1$
Key computation: Weights $\mathbf{x}$ minimize overall loss $L(\mathbf{x})$

\[ L(\mathbf{x}) = \frac{1}{N} \sum_{j=1}^{N} \text{loss } \ell(\mathbf{x}, \mathbf{v}_0^j) \text{ on sample } j \]
**Key computation**: Weights $\mathbf{x}$ minimize overall loss $L(\mathbf{x})$

$$L(\mathbf{x}) = \frac{1}{N} \sum_{j=1}^{N} \text{loss } \ell(\mathbf{x}, \mathbf{v}_0^j) \text{ on sample } j$$

“Square loss” = error $\ell(\mathbf{x}, \mathbf{v}_0^j) = \| F(\mathbf{x}, \mathbf{v}_0^j) - \text{true} \|^2$

Cross-entropy loss, hinge loss, ...

Classification problem: true = 1 or $-1$

Regression problem: true = vector

Gradient descent $\mathbf{x}_{k+1} = \text{arg min} \| \mathbf{x}_k - s_k \nabla L(\mathbf{x}_k) \|$

Stochastic descent $\mathbf{x}_{k+1} = \text{arg min} \| \mathbf{x}_k - s_k \nabla \ell(\mathbf{x}_k, \mathbf{v}_0^k) \|$
Key Questions

1. Optimization of the weights $x = A_k$ and $b_k$
Key Questions

1. Optimization of the weights $x = A_k$ and $b_k$

2. Convergence rate of descent and accelerated descent (when $x_{k+1}$ depends on $x_k$ and $x_{k-1}$: momentum added)
Key Questions

1. Optimization of the weights $x = A_k$ and $b_k$

2. Convergence rate of descent and accelerated descent (when $x_{k+1}$ depends on $x_k$ and $x_{k-1}$: momentum added)

3. Do the weights $A_1, b_1 \ldots$ generalize to unseen test data? (Early stopping / Do not overfit the data)
1. Stochastic gradient descent optimizes weights $A_k, b_k$
1. Stochastic gradient descent optimizes weights $A_k, b_k$

2. Backpropagation in the computational graph computes derivatives with respect to weights $x = A_1, b_1, \ldots, A_\ell, b_\ell$
1. Stochastic gradient descent optimizes weights $A_k, b_k$

2. Backpropagation in the computational graph computes derivatives with respect to weights $x = A_1, b_1, \ldots, A_\ell, b_\ell$

3. The learning function $F(x, v_0) = \ldots F_3(F_2(F_1(x, v)))$
1. Stochastic gradient descent optimizes weights $A_k, b_k$

2. Backpropagation in the computational graph computes derivatives with respect to weights $x = A_1, b_1, \ldots, A_\ell, b_\ell$

3. The learning function $F(x, v_0) = \ldots F_3(F_2(F_1(x, v)))$

$$F_1(v_0) = \max (A_1v_0 + b_1, 0) = \text{ReLU} \circ \text{affine map}$$

$F(v)$ is continuous piecewise linear: how many pieces?

This measures the “expressivity” of the network

Assume 1 hidden layer with $N$ neurons
$v_0$ has $m$ components / $v_1$ has $N$ components / $N$ ReLU’s
\( v_0 \) has \( m \) components / \( v_1 \) has \( N \) components / \( N \) ReLU’s

The number of flat regions in \( \mathbb{R}^m \) bounded by the \( N \) hyperplanes

\[
r(N, m) = \sum_{i=0}^{m} \binom{N}{i} = \binom{N}{0} + \binom{N}{1} + \cdots + \binom{N}{m}
\]
\( v_0 \) has \( m \) components / \( v_1 \) has \( N \) components / \( N \) ReLU’s

The number of flat regions in \( \mathbb{R}^m \) bounded by the \( N \) hyperplanes

\[
 r(N, m) = \sum_{i=0}^{m} \binom{N}{i} = \binom{N}{0} + \binom{N}{1} + \cdots + \binom{N}{m} 
\]

\( N = 3 \) folds in a plane will produce \( 1 + 3 + 3 = 7 \) pieces
$v_0$ has $m$ components / $v_1$ has $N$ components / $N$ ReLU’s

The number of flat regions in $\mathbb{R}^m$ bounded by the $N$ hyperplanes

$$r(N, m) = \sum_{i=0}^{m} \binom{N}{i} = \binom{N}{0} + \binom{N}{1} + \cdots + \binom{N}{m}$$

$N = 3$ folds in a plane will produce $1 + 3 + 3 = 7$ pieces

Start with 2 folds
$\leftarrow r(2, 2) = 4$
Add new fold
$\leftarrow r(2, 1) = 3$
\( v_0 \) has \( m \) components / \( v_1 \) has \( N \) components / \( N \) ReLU’s

The number of flat regions in \( \mathbb{R}^m \) bounded by the \( N \) hyperplanes

\[
r(N, m) = \sum_{i=0}^{m} \binom{N}{i} = \binom{N}{0} + \binom{N}{1} + \cdots + \binom{N}{m}
\]

\( N = 3 \) folds in a plane will produce \( 1 + 3 + 3 = 7 \) pieces

Start with 2 folds
\[ \leftarrow r(2, 2) = 4 \]
Add new fold
\[ \leftarrow r(2, 1) = 3 \]

Recursion \( r(N, m) = r(N - 1, m) + r(N - 1, m - 1) \)
\[ F(x) = F_2(F_1(x)) \] is continuous piecewise linear

One hidden layer of neurons: deep networks have many more
\[ F(x) = F_2(F_1(x)) \] is continuous piecewise linear

One hidden layer of neurons: deep networks have many more

Overfitting is not desirable! Gradient descent stops early!
\[ F(x) = F_2(F_1(x)) \] is continuous piecewise linear

One hidden layer of neurons: deep networks have many more

Overfitting is not desirable! Gradient descent stops early!

“Generalization” measured by success on unseen test data
\[ F(x) = F_2(F_1(x)) \] is continuous piecewise linear

One hidden layer of neurons: deep networks have many more

Overfitting is not desirable! Gradient descent stops early!

“Generalization” measured by success on unseen test data

Big problems are underdetermined [\# weights > \# samples]
$F(x) = F_2(F_1(x))$ is continuous piecewise linear

One hidden layer of neurons: deep networks have many more

Overfitting is not desirable! Gradient descent stops early!

“Generalization” measured by success on unseen test data

Big problems are underdetermined [\# weights > \# samples]

Stochastic Gradient Descent finds weights that generalize well
Convolutional Neural Nets (CNN)

\[
A = \begin{bmatrix}
  x_1 & x_0 & x_{-1} & 0 & 0 & 0 \\
  0 & x_1 & x_0 & x_{-1} & 0 & 0 \\
  0 & 0 & x_1 & x_0 & x_{-1} & 0 \\
  0 & 0 & 0 & x_1 & x_0 & x_{-1}
\end{bmatrix}
\]

\[N + 2\] inputs and \(N\) outputs
Convolutional Neural Nets (CNN)

\[
A = \begin{bmatrix}
  x_1 & x_0 & x_{-1} & 0 & 0 & 0 \\
  0 & x_1 & x_0 & x_{-1} & 0 & 0 \\
  0 & 0 & x_1 & x_0 & x_{-1} & 0 \\
  0 & 0 & 0 & x_1 & x_0 & x_{-1}
\end{bmatrix}
\]  
\[N + 2 \text{ inputs and } N \text{ outputs}\]

Each shift has a diagonal of 1’s

\[A = x_1 L + x_0 C + x_{-1} R\]
Convolutional Neural Nets (CNN)

\[ A = \begin{bmatrix}
  x_1 & x_0 & x_{-1} & 0 & 0 & 0 \\
  0 & x_1 & x_0 & x_{-1} & 0 & 0 \\
  0 & 0 & x_1 & x_0 & x_{-1} & 0 \\
  0 & 0 & 0 & x_1 & x_0 & x_{-1} \\
\end{bmatrix} \quad N + 2 \text{ inputs and } N \text{ outputs} \]

Each shift has a diagonal of 1’s

\[ A = x_1 L + x_0 C + x_{-1} R \]

\[
\frac{\partial y}{\partial x_1} = L v \\
\frac{\partial y}{\partial x_0} = C v \\
\frac{\partial y}{\partial x_{-1}} = R v
\]
Convolutions in Two Dimensions

Weights

\[
\begin{bmatrix}
  x_{11} & x_{01} & x_{-11} \\
  x_{10} & x_{00} & x_{-10} \\
  x_{1-1} & x_{0-1} & x_{-1-1}
\end{bmatrix}
\]

Input image \( v_{ij} \) \( i, j \) from \( (0, 0) \) to \( (N+1, N+1) \)

Output image \( y_{ij} \) \( i, j \) from \( (1, 1) \) to \( (N, N) \)

Shifts L, C, R, U, D = Left, Center, Right, Up, Down
Convolutions in Two Dimensions

Weights

\[
\begin{bmatrix}
  x_{11} & x_{01} & x_{-11} \\
  x_{10} & x_{00} & x_{-10} \\
  x_{1-1} & x_{0-1} & x_{-1-1}
\end{bmatrix}
\]

Input image \( v_{ij} \) \( i, j \) from \((0, 0)\) to \((N+1, N+1)\)

Output image \( y_{ij} \) \( i, j \) from \((1, 1)\) to \((N, N)\)

Shifts \( L, C, R, U, D = \text{Left, Center, Right, Up, Down} \)

A convolution is a combination of shift matrices = filter = Toeplitz matrix

The coefficients in the combination will be the “weights” to be learned.
Convolutions in Two Dimensions

Weights

\[
\begin{bmatrix}
x_{11} & x_{01} & x_{-11} \\
x_{10} & x_{00} & x_{-10} \\
x_{1-1} & x_{0-1} & x_{-1-1}
\end{bmatrix}
\]

Input image \( v_{ij} \) \( i, j \) from \((0, 0)\) to \((N + 1, N + 1)\)

Output image \( y_{ij} \) \( i, j \) from \((1, 1)\) to \((N, N)\)

Shifts L, C, R, U, D = Left, Center, Right, Up, Down

A convolution is a combination of shift matrices = filter = Toeplitz matrix

The coefficients in the combination will be the “weights” to be learned.

9 weights instead of \( n^2 \) weights in \( A \)

Allows the neural net to have more width and depth
Computing the weights $x = \text{matrices } A_k, \text{ bias vectors } b_k$

Choose a loss function $\ell$ to measure $F(x, v) - \text{ true output}$
Computing the weights $x = \text{matrices } A_k$, bias vectors $b_k$

Choose a loss function $\ell$ to measure $F(x, v) - \text{true output}$

**Total loss** $L = \frac{1}{N}$ (sum of losses for all $N$ samples)
Computing the weights $\mathbf{x} = \mathbf{A}_k$, bias vectors $\mathbf{b}_k$

Choose a loss function $\ell$ to measure $F(x, v)$—true output

**Total loss** $L = \frac{1}{N}$ (sum of losses for all $N$ samples)

Compute weights $\mathbf{x}$ to minimize the total loss $L$
Here are three loss functions—Cross-entropy is a favorite loss function for neural nets
Here are three loss functions—Cross-entropy is a favorite loss function for neural nets.

1. **Square loss**  
   \[ L(x) = \frac{1}{N} \sum_{1}^{N} \| F(x, v_i) - \text{true} \|^2 \]  
   : sum over samples \( v_i \)

2. **Hinge loss**  
   \[ L(x) = \frac{1}{N} \sum_{1}^{N} \max(0, 1 - t F(x)) \]  
   for classification

3. **Cross-entropy loss**  
   \[ L(x) = -\frac{1}{N} \sum_{1}^{N} [y_i \log \hat{y}_i + (1 - y_i) \log (1 - \hat{y}_i)] \]
Steepest Descent = Gradient Descent \[ f = x^2 + by^2 \]

- Steepest direction \( \nabla f \) up and down the bowl
- Flat direction \((\nabla f)^\perp\) along the ellipse \( x^2 + by^2 = \text{constant} \)

Steepest direction is perpendicular to the flat direction

Steepest direction is not aimed at the minimum

Steepest descent moves in the gradient direction \[ \begin{bmatrix} -2x \\ -2by \end{bmatrix} . \]

\[
\begin{align*}
x_k &= b \left( \frac{b - 1}{b + 1} \right)^k \\
y_k &= \left( \frac{1 - b}{1 + b} \right)^k \\
f(x_k, y_k) &= \left( \frac{1 - b}{1 + b} \right)^{2k} f(x_0, y_0)
\end{align*}
\]
Descent formula \( x_{k+1} = x_k - s_k \nabla F(x) \)  Stepsize \( s_k \) = Learning rate

The first descent step starts out perpendicular to the level set. As it crosses through lower level sets, the function \( f(x, y) \) is decreasing. **Eventually its path is tangent to a level set \( L \).**

Slow convergence on a zig-zag path to the minimum of \( f = x^2 + by^2 \).
Momentum and the Path of a Heavy Ball

Descent with momentum

\[ x_{k+1} = x_k - s z_k \]

with \( z_k = \nabla f(x_k) + \beta z_{k-1} \)

Descent with momentum

\[ x_{k+1} = x_k - s z_k \]

\[ z_{k+1} - \nabla f(x_{k+1}) = \beta z_k \]

It seems a miracle that this problem has a beautiful solution. The optimal \( s \) and \( \beta \) are

\[ s = \left( \frac{2}{\sqrt{\lambda_{\text{max}}} + \sqrt{\lambda_{\text{min}}}} \right)^2 \]

and

\[ \beta = \left( \frac{\sqrt{\lambda_{\text{max}}} - \sqrt{\lambda_{\text{min}}}}{\sqrt{\lambda_{\text{max}}} + \sqrt{\lambda_{\text{min}}}} \right)^2 \]
Momentum and the Path of a Heavy Ball

Descent with momentum
\[ x_{k+1} = x_k - s z_k \]
with \( z_k = \nabla f(x_k) + \beta z_{k-1} \)

Descent with momentum
\[ x_{k+1} = x_k - s z_k \]
\[ z_{k+1} - \nabla f(x_{k+1}) = \beta z_k \]

Following the eigenvector \( \boldsymbol{q} \)
\[-\lambda c_{k+1} + d_{k+1} = \beta d_k \]
\[ \begin{bmatrix} 1 & 0 \\ -\lambda & 1 \end{bmatrix} \begin{bmatrix} c_{k+1} \\ d_{k+1} \end{bmatrix} = \begin{bmatrix} 1 & -s \\ 0 & \beta \end{bmatrix} \begin{bmatrix} c_k \\ d_k \end{bmatrix} \]
Momentum and the Path of a Heavy Ball

Descent with momentum

\[ x_{k+1} = x_k - s z_k \]

with \( z_k = \nabla f(x_k) + \beta z_{k-1} \)

Descent with momentum

\[ x_{k+1} = x_k - s z_k \]

\[ z_{k+1} - \nabla f(x_{k+1}) = \beta z_k \]

Following the eigenvector \( q \)

\[ c_{k+1} = c_k - s d_k \]

\[ -\lambda c_{k+1} + d_{k+1} = \beta d_k \]

\[
\begin{bmatrix}
1 & 0 \\
-\lambda & 1
\end{bmatrix}
\begin{bmatrix}
c_{k+1} \\
d_{k+1}
\end{bmatrix}
= \begin{bmatrix}
1 \\
0
\end{bmatrix}
- s \begin{bmatrix}
c_k \\
d_k
\end{bmatrix}
\]

It seems a miracle that this problem has a beautiful solution.

The optimal \( s \) and \( \beta \) are

\[
s = \left( \frac{2}{\sqrt{\lambda_{\text{max}}} + \sqrt{\lambda_{\text{min}}}} \right)^2
\]

and

\[
\beta = \left( \frac{\sqrt{\lambda_{\text{max}}} - \sqrt{\lambda_{\text{min}}}}{\sqrt{\lambda_{\text{max}}} + \sqrt{\lambda_{\text{min}}}} \right)^2
\]
Key difference: $b$ is replaced by $\sqrt{b}$

<table>
<thead>
<tr>
<th>Ordinary descent factor</th>
<th>$\left(\frac{1 - b}{1 + b}\right)^2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accelerated descent factor</td>
<td>$\left(\frac{1 - \sqrt{b}}{1 + \sqrt{b}}\right)^2$</td>
</tr>
</tbody>
</table>
Key difference: \( b \) is replaced by \( \sqrt{b} \)

<table>
<thead>
<tr>
<th>Ordinary descent factor</th>
<th>( \left( \frac{1 - b}{1 + b} \right)^2 )</th>
<th>Accelerated descent factor</th>
<th>( \left( \frac{1 - \sqrt{b}}{1 + \sqrt{b}} \right)^2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Steepest descent</td>
<td>( \left( \frac{.99}{1.01} \right)^2 = .96 )</td>
<td>Accelerated descent</td>
<td>( \left( \frac{.9}{1.1} \right)^2 = .67 )</td>
</tr>
</tbody>
</table>
**Key difference:** \( b \) is replaced by \( \sqrt{b} \)

<table>
<thead>
<tr>
<th></th>
<th>Ordinary descent factor</th>
<th>Accelerated descent factor</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( \left(\frac{1 - b}{1 + b}\right)^2 )</td>
<td>( \left(\frac{1 - \sqrt{b}}{1 + \sqrt{b}}\right)^2 )</td>
</tr>
<tr>
<td>Steepest descent</td>
<td>( \left(\frac{.99}{1.01}\right)^2 = .96 )</td>
<td>( \left(\frac{.9}{1.1}\right)^2 = .67 )</td>
</tr>
</tbody>
</table>

Notice that \( \lambda_{\text{max}}/\lambda_{\text{min}} = \frac{1}{b} = \kappa \) is the **condition number** of \( S \)
Stochastic Gradient Descent

Stochastic gradient descent uses a “minibatch” of the training data.
Stochastic Gradient Descent

Stochastic gradient descent uses a “minibatch” of the training data. Every step is much faster than using all data.
Stochastic gradient descent uses a “minibatch” of the training data.

Every step is much faster than using all data.

*We don’t want to fit the training data too perfectly* (overfitting)
Stochastic Gradient Descent

Stochastic gradient descent uses a “minibatch” of the training data.

Every step is much faster than using all data.

*We don’t want to fit the training data too perfectly* (overfitting).

*Choosing a polynomial of degree 60 to fit 61 data points.*
Early steps of SGD often converge quickly toward the solution $x^*$.
Stochastic Descent Using One Sample Per Step

Early steps of SGD often converge quickly toward the solution $x^*$

Here we pause to look at semi-convergence: Fast start by stochastic gradient descent

**Convergence at the start changes to large oscillations near the solution**
Early steps of SGD often converge quickly toward the solution $x^*$

Here we pause to look at semi-convergence: Fast start by stochastic gradient descent

Convergence at the start changes to large oscillations near the solution
Early steps of SGD often converge quickly toward the solution $x^*$

Here we pause to look at semi-convergence: Fast start by stochastic gradient descent

Convergence at the start changes to large oscillations near the solution

Kaczmarz for $Ax = b$ with random $i(k)$

$$x_{k+1} = x_k + \frac{b_i - a_i^T x_k}{||a_i||^2} a_i$$
Adaptive Methods Using Earlier Gradients (ADAM)

Adaptive Stochastic Gradient Descent

$$x_{k+1} = x_k - s_k D_k$$
Adaptive Stochastic Gradient Descent

\[ x_{k+1} = x_k - s_k D_k \]

\[
D_k = \delta D_{k-1} + (1 - \delta) \nabla L(x_k) \quad s_k^2 = \beta s_{k-1}^2 + (1 - \beta) \| \nabla L(x_k) \|^2
\]
Adaptive Methods Using Earlier Gradients (ADAM)

Adaptive Stochastic Gradient Descent

\[
x_{k+1} = x_k - s_k D_k
\]

\[
D_k = \delta D_{k-1} + (1 - \delta) \nabla L(x_k)
\]

\[
s_k^2 = \beta s_{k-1}^2 + (1 - \beta) ||\nabla L(x_k)||^2
\]

Why do the weights generalize well to unseen test data?
Computation of $\partial F/\partial x$: Explicit Formulas

$v_L = b_L + A_L v_{L-1}$ or simply $w = b + Av$. 
Computation of $\frac{\partial F}{\partial x}$: Explicit Formulas

$v_L = b_L + A_L v_{L-1}$ or simply $w = b + A v$.

The output $w_i$ is not affected by $b_j$ or $A_{jk}$ if $j \neq i$.
Computation of $\partial F/\partial x$: Explicit Formulas

$v_L = b_L + A_L v_{L-1}$ or simply $w = b + Av$.

The output $w_i$ is not affected by $b_j$ or $A_{jk}$ if $j \neq i$.

**Fully connected layer**

**Independent weights** $A_{jk}$

$$\frac{\partial w_i}{\partial b_j} = \delta_{ij} \quad \text{and} \quad \frac{\partial w_i}{\partial A_{jk}} = \delta_{ij} v_k$$
Computation of $\partial F/\partial x$: Explicit Formulas

$v_L = b_L + A_L v_{L-1}$ or simply $w = b + Av$.

The output $w_i$ is not affected by $b_j$ or $A_{jk}$ if $j \neq i$.

Fully connected layer

Independent weights $A_{jk}$

$\frac{\partial w_i}{\partial b_j} = \delta_{ij}$ and $\frac{\partial w_i}{\partial A_{jk}} = \delta_{ij} v_k$

Example

$\begin{bmatrix} w_1 \\ w_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} + \begin{bmatrix} a_{11} v_1 + a_{12} v_2 \\ a_{21} v_1 + a_{22} v_2 \end{bmatrix}$
Computation of $\partial F/\partial x$: Explicit Formulas

$v_L = b_L + A_L v_{L-1}$ or simply $w = b + Av$.

The output $w_i$ is not affected by $b_j$ or $A_{jk}$ if $j \neq i$

Fully connected layer
Independent weights $A_{jk}$

\[
\frac{\partial w_i}{\partial b_j} = \delta_{ij} \quad \text{and} \quad \frac{\partial w_i}{\partial A_{jk}} = \delta_{ij} v_k
\]

Example

\[
\begin{bmatrix}
  w_1 \\
  w_2
\end{bmatrix} = \begin{bmatrix}
  b_1 \\
  b_2
\end{bmatrix} + \begin{bmatrix}
  a_{11} v_1 + a_{12} v_2 \\
  a_{21} v_1 + a_{22} v_2
\end{bmatrix}
\]

\[
\frac{\partial w_1}{\partial b_1} = 1, \quad \frac{\partial w_1}{\partial b_2} = 0, \quad \frac{\partial w_1}{\partial a_{11}} = v_1, \quad \frac{\partial w_1}{\partial a_{12}} = v_2, \quad \frac{\partial w_1}{\partial a_{21}} = \frac{\partial w_1}{\partial a_{22}} = 0.
\]
Backpropagation and the Chain Rule

$L(x)$ adds up all the losses $\ell (w - \text{true}) = \ell (F(x,v) - \text{true})$
Backpropagation and the Chain Rule

$L(x)$ adds up all the losses $\ell (w - \text{true}) = \ell (F(x, v) - \text{true})$

The partial derivatives of $L$ with respect to the weights $x$ should be zero.
Backpropagation and the Chain Rule

$L(x)$ adds up all the losses $\ell(w - \text{true}) = \ell(F(x, v) - \text{true})$

The partial derivatives of $L$ with respect to the weights $x$ should be zero.

Chain rule \[ \frac{d}{dx} (F_3(F_2(F_1(x)))) = \left( \frac{dF_3}{dF_2}(F_2(F_1(x))) \right) \left( \frac{dF_2}{dF_1}(F_1(x)) \right) \left( \frac{dF_1}{dx}(x) \right) \]
Backpropagation and the Chain Rule

$L(x)$ adds up all the losses $\ell(w - \text{true}) = \ell(F(x, v) - \text{true})$

The partial derivatives of $L$ with respect to the weights $x$ should be zero.

Chain rule
\[
\frac{d}{dx}(F_3(F_2(F_1(x)))) = \left(\frac{dF_3}{dF_2}(F_2(F_1(x)))\right) \left(\frac{dF_2}{dF_1}(F_1(x))\right) \left(\frac{dF_1}{dx}(x)\right)
\]

What is the multivariable chain rule?
Backpropagation and the Chain Rule

\[ L(x) \text{ adds up all the losses } \ell (w - \text{true}) = \ell (F(x, v) - \text{true}) \]

The partial derivatives of \( L \) with respect to the weights \( x \) should be zero.

Chain rule \[ \frac{d}{dx} (F_3(F_2(F_1(x)))) = \left( \frac{dF_3}{dF_2}(F_2(F_1(x))) \right) \left( \frac{dF_2}{dF_1}(F_1(x)) \right) \left( \frac{dF_1}{dx}(x) \right) \]

What is the multivariable chain rule?

Which order (forward or backward along the chain) is faster?
Backward-mode $AD$ is faster for $M_1M_2w$

$(M_1M_2)w$ needs $N^3+N^2$ multiplications  
$M_1(M_2w)$ needs only $N^2+N^2$
Backward-mode $AD$ is faster for $M_1 M_2 w$

$(M_1 M_2)w$ needs $N^3 + N^2$ multiplications \hspace{1cm} $M_1(M_2w)$ needs only $N^2 + N^2$

**Forward** \(((M_1 M_2)M_3)\ldots M_L)w$$ needs $(L - 1)N^3 + N^2$
Backward-mode $AD$ is faster for $M_1M_2w$

$(M_1M_2)w$ needs $N^3 + N^2$ multiplications \[ M_1(M_2w) \] needs only $N^2 + N^2$

**Forward** \[ (((M_1M_2)M_3)\ldots M_L)w \] needs $(L - 1)N^3 + N^2$

**Backward** \[ M_1(M_2(\ldots (M_Lw))) \] needs $LN^2$
The Multivariable Chain Rule

\[
\frac{\partial w}{\partial v} = \begin{bmatrix}
    \frac{\partial w_1}{\partial v_1} & \ldots & \frac{\partial w_1}{\partial v_n} \\
    \ldots & \ldots & \ldots \\
    \frac{\partial w_p}{\partial v_1} & \ldots & \frac{\partial w_p}{\partial v_n}
\end{bmatrix}
\]

\[
\frac{\partial v}{\partial u} = \begin{bmatrix}
    \frac{\partial v_1}{\partial u_1} & \ldots & \frac{\partial v_1}{\partial u_m} \\
    \ldots & \ldots & \ldots \\
    \frac{\partial v_n}{\partial u_1} & \ldots & \frac{\partial v_n}{\partial u_m}
\end{bmatrix}
\]

\[
\frac{\partial w}{\partial u} = \left( \frac{\partial w}{\partial v} \right) \left( \frac{\partial v}{\partial u} \right)
\]
The Multivariable Chain Rule

\[
\frac{\partial w}{\partial v} = \begin{bmatrix}
\frac{\partial w_1}{\partial v_1} & \cdots & \frac{\partial w_1}{\partial v_n} \\
\frac{\partial w_p}{\partial v_1} & \cdots & \frac{\partial w_p}{\partial v_n}
\end{bmatrix}
\]

\[
\frac{\partial v}{\partial u} = \begin{bmatrix}
\frac{\partial v_1}{\partial u_1} & \cdots & \frac{\partial v_1}{\partial u_m} \\
\frac{\partial v_n}{\partial u_1} & \cdots & \frac{\partial v_n}{\partial u_m}
\end{bmatrix}
\]

\[
\frac{\partial w}{\partial u_k} = \frac{\partial w_i}{\partial v_1} \frac{\partial v_1}{\partial u_k} + \cdots + \frac{\partial w_i}{\partial v_n} \frac{\partial v_n}{\partial u_k} = \left( \frac{\partial w_i}{\partial v_1}, \ldots, \frac{\partial w_i}{\partial v_n} \right) \cdot \left( \frac{\partial v_1}{\partial u_k}, \ldots, \frac{\partial v_n}{\partial u_k} \right)
\]

Multivariable chain rule: Multiply matrices!

\[
\frac{\partial w}{\partial u} = \left( \frac{\partial w}{\partial v} \right) \left( \frac{\partial v}{\partial u} \right)
\]
The words **learning rate** are often used in place of stepsize.
The words **learning rate** are often used in place of stepsize. 

- **$s_k$ is too small**  Then gradient descent takes too long to minimize $L(x)$.
The words **learning rate** are often used in place of stepsize

**$s_k$ is too small**  Then gradient descent takes too long to minimize $L(x)$

**$s_k$ is too large**  Overshooting the best choice $x_{k+1}$ in the descent direction
Hyperparameters: The Fateful Decisions

The words **learning rate** are often used in place of stepsize

$s_k$ **is too small** Then gradient descent takes too long to minimize $L(x)$

$s_k$ **is too large** Overshooting the best choice $x_{k+1}$ in the descent direction

**Cross-validation** Divide the available data into $K$ subsets
Regularization = Weight decay: $\ell^2$ or $\ell^1$

Small $\lambda$: increase the variance of the error (overfitting)
Regularization = Weight decay: $\ell^2$ or $\ell^1$

Small $\lambda$: increase the variance of the error (overfitting)

Large $\lambda$: increase the bias (underfitting), $||b - Ax||^2$ is less important
Regularization = Weight decay: $\ell^2$ or $\ell^1$

Small $\lambda$: increase the variance of the error (overfitting)

Large $\lambda$: increase the bias (underfitting), $||b - Ax||^2$ is less important

Deep learning with many extra weights and good hyperparameters will find solutions that generalize, without penalty
Regularization = Weight decay: $\ell^2$ or $\ell^1$

Small $\lambda$: increase the variance of the error (overfitting)

Large $\lambda$: increase the bias (underfitting), $||b - Ax||^2$ is less important

Deep learning with many extra weights and good hyperparameters will find solutions that generalize, without penalty
Softmax Outputs for Multiclass Networks

\[ p_j = \frac{1}{S} e^{w_j} \quad \text{where} \quad S = \sum_{k=1}^{n} e^{w_k} \]
Softmax Outputs for Multiclass Networks

\[
\begin{align*}
\text{Softmax} & \quad p_j = \frac{1}{S} e^{w_j} \quad \text{where} \quad S = \sum_{k=1}^{n} e^{w_k}
\end{align*}
\]

Softmax produces the probabilities in [teachyourmachine.com](http://teachyourmachine.com)
Softmax Outputs for Multiclass Networks

$$p_j = \frac{1}{S} e^{w_j} \quad \text{where} \quad S = \sum_{k=1}^{n} e^{w_k}$$

Softmax produces the probabilities in teachyourmachine.com

The World Championship at the Game of Go
Softmax Outputs for Multiclass Networks

\[
\text{Softmax} \quad p_j = \frac{1}{S} e^{w_j} \quad \text{where} \quad S = \sum_{k=1}^{n} e^{w_k}
\]

Softmax produces the probabilities in teachyourmachine.com

The World Championship at the Game of Go

Residual Networks (ResNets) \( \text{“skip connections”} \)
Neural Nets Give Universal Approximation

If $f(v)$ is continuous there exists $x$ so that $|F(x, v) - f(v)| < \epsilon$ for all $v$
If $f(v)$ is continuous there exists $x$ so that $|F(x, v) - f(v)| < \epsilon$ for all $v$

Accuracy of approximation to $f$:

$$\min_x \|F(x, v) - f(v)\| \leq C\|f\|_S$$
Neural Nets Give Universal Approximation

If $f(v)$ is continuous there exists $x$ so that $|F(x, v) - f(v)| < \epsilon$ for all $v$

Accuracy of approximation to $f$\n\[
\min_x ||F(x, v) - f(v)|| \leq C||f||_S
\]

Deep networks give closer approximation than splines or shallow nets
Counting Flat Pieces in the Graph

**Theorem** For $v$ in $\mathbb{R}^m$, suppose the graph of $F(v)$ has folds along $N$ hyperplanes $H_1, \ldots, H_N$. Those come from $N$ linear equations $a_i^T v + b_i = 0$, in other words ReLU at $N$ neurons. $F$ has $r(N, m)$ linear pieces:

$$r(N, m) = \sum_{i=0}^m \binom{N}{i} = \binom{N}{0} + \binom{N}{1} + \cdots + \binom{N}{m}$$
Theorem  For $\mathbf{v}$ in $\mathbb{R}^m$, suppose the graph of $F(\mathbf{v})$ has folds along $N$ hyperplanes $H_1, \ldots, H_N$. Those come from $N$ linear equations $a_i^T \mathbf{v} + b_i = 0$, in other words ReLU at $N$ neurons. $F$ has $r(N, m)$ linear pieces:

$$r(N, m) = \sum_{i=0}^{m} \binom{N}{i} = \binom{N}{0} + \binom{N}{1} + \cdots + \binom{N}{m}$$

$$r(N, m) = r(N - 1, m) + r(N - 1, m - 1)$$
Counting Flat Pieces in the Graph

**Theorem**  For \( v \) in \( \mathbb{R}^m \), suppose the graph of \( F(v) \) has folds along \( N \) hyperplanes \( H_1, \ldots, H_N \). Those come from \( N \) linear equations \( a_i^T v + b_i = 0 \), in other words ReLU at \( N \) neurons. \( F \) has \( r(N, m) \) linear pieces:

\[
r(N, m) = \sum_{i=0}^{m} \binom{N}{i} = \binom{N}{0} + \binom{N}{1} + \cdots + \binom{N}{m}
\]

\[
r(N, m) = r(N - 1, m) + r(N - 1, m - 1)
\]

Start with 2 planes

\[\leftarrow r(2, 2) = 4\]

Add new plane \( H \)

\[\leftarrow r(2, 1) = 3\]
Continuous Piecewise Linear Function

How many linear pieces with more layers?
How many linear pieces with more layers?

Now ReLU is folding piecewise linear functions
How many linear pieces with more layers?

Now ReLU is folding **piecewise linear** functions

Hanin-Rolnick: Still $r(N, m) \approx cN^m$ pieces from $N$ neurons