Simple Neural Nets

CAP 5415, FALL 2019.
Credits

Lecture contains materials from Matt Gormley
  ◦ http://www.cs.cmu.edu/~mgormley/
Consider the first stage of Neural Network

- A $d \times 1$ input feature vector is connected to $L \times 1$ output vector
  - Each element of the output vector $y_i$ is the output of a "neuron".
- The relation between an input vector $x$ and the output of the layer $y$ is given by $Wx = z$.
- This is also called a Fully Connected Layer.
Output Stage

Now consider a second stage added to the network
• The second layer \( z \) is now “hidden” and not observed.
• The output vector \( y \) is related to the hidden layer by the equation \( Fz = y \)

The relation between the input and the output is given by
\[
FQ(Wx) = y
\]
Building a Neural Net

Output

Features

\[ x_1 \quad \cdots \quad x_M \]
Building a Neural Net

Output

Hidden Layer

Input

\[ D = M \]
Building a Neural Net

Output

Hidden Layer

Input

\[ D = M \]
Building a Neural Net

Input

Hidden Layer

Output

\[ D = M \]
Building a Neural Net

Output

Hidden Layer

Input

\[ D < M \]
• 0 hidden layers: linear classifier
  – Hyperplanes

Example from Eric Postma via Jason Eisner
• 1 hidden layer
  – Boundary of convex region (open or closed)

Example from Eric Postma via Jason Eisner
Decision Boundary

- 2 hidden layers
  - Combinations of convex regions

Example from Eric Postma via Jason Eisner
Multi-Class Output

Decision Functions

Output

Hidden Layer

Input

\[ y_1, \ldots, y_K \]
\[ a_1, a_2, \ldots, a_D \]
\[ x_1, x_2, x_3, \ldots, x_M \]
ARCHITECTURES
Neural Network Architectures

Even for a basic Neural Network, there are many design decisions to make:

1. # of hidden layers (depth)
2. # of units per hidden layer (width)
3. Type of activation function (nonlinearity)
4. Form of objective function
Activation Functions

Neural Network with sigmoid activation functions

(A) Input
Given $x_i$, $\forall i$

(B) Hidden (linear)
$a_j = \sum_{i=0}^{M} \alpha_{ji} x_i$, $\forall j$

(C) Hidden (sigmoid)
$z_j = \frac{1}{1+\exp(-a_j)}$, $\forall j$

(D) Output (linear)
$b = \sum_{j=0}^{D} \beta_j z_j$

(E) Output (sigmoid)
$y = \frac{1}{1+\exp(-b)}$

(F) Loss
$J = \frac{1}{2}(y - y^*)^2$
Activation Functions

Neural Network with arbitrary nonlinear activation functions

(A) Input
Given $x_i$, $\forall i$

(B) Hidden (linear)
$a_j = \sum_{i=0}^{M} \alpha_{ji} x_i$, $\forall j$

(C) Hidden (nonlinear)
$z_j = \sigma(a_j)$, $\forall j$

(D) Output (linear)
$b = \sum_{j=0}^{D} \beta_j z_j$

(E) Output (nonlinear)
y = $\sigma(b)$

(F) Loss
$J = \frac{1}{2} (y - y^*)^2$
Activation Functions

So far, we’ve assumed that the activation function (nonlinearity) is always the sigmoid function...

Sigmoid / Logistic Function

\[ \text{logistic}(u) = \frac{1}{1 + e^{-u}} \]
Activation Functions

• A new change: modifying the nonlinearity
  – The logistic is not widely used in modern ANNs

Alternate 1: tanh
Like logistic function but shifted to range [-1, +1]

Slide from William Cohen
Understanding the difficulty of training deep feedforward neural networks

Figure from Glorot & Bentio (2010)
Activation Functions

- A new change: modifying the nonlinearity
  - reLU often used in vision tasks

Alternate 2: rectified linear unit

Linear with a cutoff at zero

(Implementation: clip the gradient when you pass zero)
Activation Functions

- A new change: modifying the nonlinearity
  - reLU often used in vision tasks

Alternate 2: rectified linear unit

Soft version: \( \log(\exp(x)+1) \)

Doesn’t saturate (at one end)
Sparsifies outputs
Helps with vanishing gradient

Slide from William Cohen
Objective Functions for NNs

- **Regression:**
  - Use the same objective as Linear Regression
  - Quadratic loss (i.e. mean squared error)

- **Classification:**
  - Use the same objective as Logistic Regression
  - Cross-entropy (i.e. negative log likelihood)
  - This requires probabilities, so we add an additional “softmax” layer at the end of our network

<table>
<thead>
<tr>
<th>Objective</th>
<th>Forward</th>
<th>Backward</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quadratic</td>
<td>$J = \frac{1}{2}(y - y^*)^2$</td>
<td>$\frac{dJ}{dy} = y - y^*$</td>
</tr>
<tr>
<td>Cross Entropy</td>
<td>$J = y^* \log(y) + (1 - y^*) \log(1 - y)$</td>
<td>$\frac{dJ}{dy} = y^* \frac{1}{y} + (1 - y^*) \frac{1}{y - 1}$</td>
</tr>
</tbody>
</table>
Multi-Class Output
Multi-Class Output

Softmax:

\[ y_k = \frac{\exp(b_k)}{\sum_{l=1}^{K} \exp(b_l)} \]

\[ \text{(A) Input} \quad \text{Given } x_i, \forall i \]

\[ \text{(B) Hidden (linear)} \quad a_j = \sum_{i=0}^{M} \alpha_{ji} x_i, \forall j \]

\[ \text{(C) Hidden (nonlinear)} \quad z_j = \sigma(a_j), \forall j \]

\[ \text{(D) Output (linear)} \quad b_k = \sum_{j=0}^{D} \beta_{kj} z_j \forall k \]

\[ \text{(E) Output (softmax)} \quad y_k = \frac{\exp(b_k)}{\sum_{l=1}^{K} \exp(b_l)} \]

\[ \text{(F) Loss} \quad J = \sum_{k=1}^{K} y_k^* \log(y_k) \]
Figure 5: Cross entropy (black, surface on top) and quadratic (red, bottom surface) cost as a function of two weights (one at each layer) of a network with two layers, $W_1$ respectively on the first layer and $W_2$ on the second, output layer.
A Recipe for Machine Learning

1. Given training data:
\[ \{x_i, y_i\}_{i=1}^{N} \]

2. Choose each of these:
   - Decision function
     \[ \hat{y} = f_\theta(x_i) \]
   - Loss function
     \[ \ell(\hat{y}, y_i) \in \mathbb{R} \]

3. Define goal:
\[ \theta^* = \arg\min_\theta \sum_{i=1}^{N} \ell(f_\theta(x_i), y_i) \]

4. Train with SGD:
   (take small steps opposite the gradient)
\[ \theta^{(t+1)} = \theta^{(t)} - \eta_t \nabla \ell(f_\theta(x_i), y_i) \]
BACKPROPAGATION
1. Given training data:
\[
\{x_i, y_i\}_{i=1}^{N}
\]

2. Choose each of these:
- Decision function
  \[
  \hat{y} = f_\theta(x_i)
  \]
- Loss function
  \[
  \ell(\hat{y}, y_i) \in \mathbb{R}
  \]

3. Define goal:
\[
\theta^* = \arg\min_{\theta} \sum_{i=1}^{N} \ell(f_\theta(x_i), y_i)
\]

4. Train with SGD:
(take small steps opposite the gradient)
\[
\theta^{(t+1)} = \theta^{(t)} - \eta_t \nabla \ell(f_\theta(x_i), y_i)
\]
• **Question 1:** How can we compute the gradients of the parameters of an arbitrary neural network?

• **Question 2:** How can we make the gradient computation efficient?
Given: \( y = g(u) \) and \( u = h(x) \).

Chain Rule:

\[
\frac{dy_i}{dx_k} = \sum_{j=1}^{J} \frac{dy_i}{du_j} \frac{du_j}{dx_k}, \quad \forall i, k
\]
Given: $y = g(u)$ and $u = h(x)$.

**Chain Rule:**

$$\frac{dy_i}{dx_k} = \sum_{j=1}^{J} \frac{dy_i}{du_j} \frac{du_j}{dx_k}, \quad \forall i, k$$

**Backpropagation** is just repeated application of the **chain rule** from Calculus 101.
Chain Rule:

Given: $y = g(u)$ and $u = h(x)$.

Chain Rule:

$$\frac{dy_i}{dx_k} = \sum_{j=1}^{J} \frac{dy_i}{du_j} \frac{du_j}{dx_k}, \quad \forall i, k$$

Backpropagation:

1. **Instantiate the computation as a directed acyclic graph**, where each intermediate quantity is a node.
2. At each node, store (a) the quantity computed in the forward pass and (b) the **partial derivative** of the goal with respect to that node’s intermediate quantity.
3. **Initialize** all partial derivatives to 0.
4. Visit each node in **reverse topological order**. At each node, add its contribution to the partial derivatives of its parents.

This algorithm is also called **automatic differentiation in the reverse-mode**.
**Simple Example:** The goal is to compute $J = \cos(\sin(x^2) + 3x^2)$ on the forward pass and the derivative $\frac{dJ}{dx}$ on the backward pass.

<table>
<thead>
<tr>
<th>Forward</th>
</tr>
</thead>
<tbody>
<tr>
<td>$J = \cos(u)$</td>
</tr>
<tr>
<td>$u = u_1 + u_2$</td>
</tr>
<tr>
<td>$u_1 = \sin(t)$</td>
</tr>
<tr>
<td>$u_2 = 3t$</td>
</tr>
<tr>
<td>$t = x^2$</td>
</tr>
</tbody>
</table>
**Simple Example:** The goal is to compute $J = \cos(\sin(x^2) + 3x^2)$ on the forward pass and the derivative $\frac{dJ}{dx}$ on the backward pass.

<table>
<thead>
<tr>
<th>Forward</th>
<th>Backward</th>
</tr>
</thead>
<tbody>
<tr>
<td>$J = \cos(u)$</td>
<td>$\frac{dJ}{du} = -\sin(u)$</td>
</tr>
<tr>
<td>$u = u_1 + u_2$</td>
<td>$\frac{dJ}{du_1} = \frac{dJ}{du} \frac{du}{du_1}$, $\frac{du}{du_1} = 1$</td>
</tr>
<tr>
<td>$u_1 = \sin(t)$</td>
<td>$\frac{dJ}{dt} = \frac{dJ}{du_1} \frac{du_1}{dt}$, $\frac{du_1}{dt} = \cos(t)$</td>
</tr>
<tr>
<td>$u_2 = 3t$</td>
<td>$\frac{dJ}{du_2} = \frac{dJ}{du} \frac{du}{du_2}$, $\frac{du}{du_2} = 1$</td>
</tr>
<tr>
<td>$t = x^2$</td>
<td>$\frac{dJ}{dx} = \frac{dJ}{dt} \frac{dt}{dx}$, $\frac{dt}{dx} = 2x$</td>
</tr>
</tbody>
</table>
Case 1: Logistic Regression

\[ J = y^* \log y + (1 - y^*) \log(1 - y) \]
\[ y = \frac{1}{1 + \exp(-a)} \]
\[ a = \sum_{j=0}^{D} \theta_j x_j \]

Forward

\[ \frac{dJ}{da} = \frac{y^*}{y} + \frac{(1 - y^*)}{y - 1} \]
\[ \frac{dJ}{d\theta_j} = \frac{dJ}{da} \frac{da}{d\theta_j}, \quad \frac{da}{d\theta_j} = x_j \]
\[ \frac{dJ}{dx_j} = \frac{dJ}{da} \frac{da}{dx_j}, \quad \frac{da}{dx_j} = \theta_j \]

Backward
Backpropagation

(A) Input
Given $x_i$, $\forall i$

(B) Hidden (linear)
$$a_j = \sum_{i=0}^{M} \alpha_{ji} x_i, \forall j$$

(C) Hidden (sigmoid)
$$z_j = \frac{1}{1+\exp(-a_j)}, \forall j$$

(D) Output (linear)
$$b = \sum_{j=0}^{D} \beta_j z_j$$

(E) Output (sigmoid)
$$y = \frac{1}{1+\exp(-b)}$$
Backpropagation

(A) Input
Given $x_i$, $\forall i$

(B) Hidden (linear)
$$a_j = \sum_{i=0}^{M} \alpha_{ji} x_i, \forall j$$

(C) Hidden (sigmoid)
$$z_j = \frac{1}{1 + \exp(-a_j)}, \forall j$$

(D) Output (linear)
$$b = \sum_{j=0}^{D} \beta_j z_j$$

(E) Output (sigmoid)
$$y = \frac{1}{1 + \exp(-b)}$$

(F) Loss
$$J = \frac{1}{2} (y - y^*)^2$$
Case 2: Neural Network

Forward

\[ J = y^* \log y + (1 - y^*) \log (1 - y) \]

\[ y = \frac{1}{1 + \exp(-b)} \]

\[ b = \sum_{j=0}^{D} \beta_j z_j \]

\[ z_j = \frac{1}{1 + \exp(-a_j)} \]

\[ a_j = \sum_{i=0}^{M} \alpha_{ji} x_i \]

Backward

\[ \frac{dJ}{dy} = \frac{y^*}{y} + \frac{(1 - y^*)}{y - 1} \]

\[ \frac{dJ}{db} = \frac{dJ}{dy} \frac{dy}{db} = \frac{\exp(-b)}{(\exp(-b) + 1)^2} \]

\[ \frac{dJ}{d\beta_j} = \frac{dJ}{db} \frac{db}{d\beta_j} = z_j \]

\[ \frac{dJ}{dz_j} = \frac{dJ}{db} \frac{db}{dz_j} = \beta_j \]

\[ \frac{dJ}{da_j} = \frac{dJ}{dz_j} \frac{dz_j}{da_j} = \frac{\exp(-a_j)}{(\exp(-a_j) + 1)^2} \]

\[ \frac{dJ}{d\alpha_{ji}} = \frac{dJ}{da_j} \frac{da_j}{d\alpha_{ji}} = x_i \]

\[ \frac{dJ}{dx_i} = \frac{dJ}{da_j} \frac{da_j}{dx_i} = \sum_{j=0}^{D} \alpha_{ji} \]
Chain Rule:

Given: \( y = g(u) \) and \( u = h(x) \).

Chain Rule:

\[
\frac{dy_i}{dx_k} = \sum_{j=1}^{J} \frac{dy_i}{du_j} \frac{du_j}{dx_k}, \quad \forall i, k
\]

Backpropagation:

1. **Instantiate the computation as a directed acyclic graph**, where each intermediate quantity is a node.
2. At each node, store (a) the quantity computed in the forward pass and (b) the **partial derivative** of the goal with respect to that node’s intermediate quantity.
3. **Initialize** all partial derivatives to 0.
4. Visit each node in **reverse topological order**. At each node, add its contribution to the partial derivatives of its parents.

This algorithm is also called **automatic differentiation in the reverse-mode**.
Chain Rule:

Given: \( y = g(u) \) and \( u = h(x) \).

\[
\frac{dy_i}{dx_k} = \sum_{j=1}^{J} \frac{dy_i}{du_j} \cdot \frac{du_j}{dx_k}, \quad \forall i, k
\]

Backpropagation:
1. **Instantiate the computation as a directed acyclic graph**, where each node represents a Tensor.
2. At each node, store (a) the quantity computed in the forward pass and (b) the **partial derivatives** of the goal with respect to that node’s Tensor.
3. **Initialize** all partial derivatives to 0.
4. Visit each node in **reverse topological order**. At each node, add its contribution to the partial derivatives of its parents.

This algorithm is also called **automatic differentiation in the reverse-mode**.
### Case 2: Backpropagation

<table>
<thead>
<tr>
<th>Module</th>
<th>Forward</th>
<th>Backward</th>
</tr>
</thead>
<tbody>
<tr>
<td>Module 5</td>
<td>$J = y^* \log y + (1 - y^*) \log(1 - y)$</td>
<td>$\frac{dJ}{dy} = \frac{y^<em>}{y} + \frac{(1 - y^</em>)}{y - 1}$</td>
</tr>
<tr>
<td>Module 4</td>
<td>$y = \frac{1}{1 + \exp(-b)}$</td>
<td>$\frac{dJ}{db} = \frac{dJ}{dy} \frac{dy}{db} = \frac{\exp(-b)}{(\exp(-b) + 1)^2}$</td>
</tr>
<tr>
<td>Module 3</td>
<td>$b = \sum_{j=0}^{D} \beta_j z_j$</td>
<td>$\frac{dJ}{d\beta_j} = \frac{dJ}{db} \frac{db}{d\beta_j}, \frac{d\beta_j}{db} = z_j$</td>
</tr>
<tr>
<td>Module 2</td>
<td>$z_j = \frac{1}{1 + \exp(-a_j)}$</td>
<td>$\frac{dJ}{da_j} = \frac{dJ}{dz_j} \frac{dz_j}{da_j}, \frac{dz_j}{da_j} = \frac{\exp(-a_j)}{(\exp(-a_j) + 1)^2}$</td>
</tr>
<tr>
<td>Module 1</td>
<td>$a_j = \sum_{i=0}^{M} \alpha_{ji} x_i$</td>
<td>$\frac{dJ}{d\alpha_{ji}} = \frac{dJ}{da_j} \frac{da_j}{d\alpha_{ji}}, \frac{da_j}{d\alpha_{ji}} = x_i$</td>
</tr>
</tbody>
</table>
1. Given training data:
   \[ \{x_i, y_i\}_{i=1}^{N} \]

2. Choose each of these:
   - Decision function
     \[ \hat{y} = f_\theta(x_i) \]
   - Loss function
     \[ \ell(\hat{y}, y_i) \in \mathbb{R} \]

3. Define goal:

4. Train with SGD:
   \[ \theta(t+1) = \theta(t) - \eta_t \nabla \ell(f_\theta(x_i), y_i) \]

**Gradients**

**Backpropagation** can compute this gradient!
And it’s a **special case of a more general algorithm** called reverse-mode automatic differentiation that can compute the gradient of any differentiable function efficiently!
Summary

1. Neural Networks…
   – provide a way of learning features
   – are highly nonlinear prediction functions
   – (can be) a highly parallel network of logistic regression classifiers
   – discover useful hidden representations of the input

2. Backpropagation…
   – provides an efficient way to compute gradients
   – is a special case of reverse-mode automatic differentiation
Credits

• Lecture contains materials from Matt Gormley
  – http://www.cs.cmu.edu/~mgormley/