Binary Logistic Regression

+ Multinomial Logistic Regression
Reminders

• **Midterm Exam 1**
  – Tue, Feb. 18, 7:00pm – 9:00pm

• **Homework 4: Logistic Regression**
  – Out: Wed, Feb. 19
  – Due: Fri, Feb. 28 at 11:59pm

• **Today’s In-Class Poll**

• *Reading on Probabilistic Learning is reused later in the course for MLE/MAP*
MLE

Suppose we have data $\mathcal{D} = \{x^{(i)}\}_{i=1}^{N}$

**Principle of Maximum Likelihood Estimation:**
Choose the parameters that maximize the likelihood of the data.

$$\theta^{\text{MLE}} = \arg\max_{\theta} \prod_{i=1}^{N} p(x^{(i)}|\theta)$$

Maximum Likelihood Estimate (MLE)
MLE

What does maximizing likelihood accomplish?

• There is only a finite amount of probability mass (i.e. sum-to-one constraint)

• MLE tries to allocate *as much* probability mass *as possible* to the things we have observed...

...at the expense of the things we have *not* observed
MOTIVATION:
LOGISTIC REGRESSION
Example: Image Classification

• ImageNet LSVRC-2010 contest:
  – **Dataset:** 1.2 million labeled images, 1000 classes
  – **Task:** Given a new image, label it with the correct class
  – **Multiclass** classification problem
• Examples from http://image-net.org/
Bird

Warm-blooded egg-laying vertebrates characterised by feathers and forelimbs modified as wings

<table>
<thead>
<tr>
<th>Category</th>
<th>Subcategory</th>
<th>Example Images</th>
</tr>
</thead>
<tbody>
<tr>
<td>Marine animal</td>
<td>Marine creature</td>
<td>Sea animal, sea creature</td>
</tr>
<tr>
<td>Frog</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bird</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parrot</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Duck</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Penguin</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Crane</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vulture</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Penguin</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2125 images, 92.85% accuracy
German iris, *Iris kochii*

Irises of northern Italy having dark blue-purple flowers, similar in size but smaller than *Iris germanica*. 

<table>
<thead>
<tr>
<th>Common Name</th>
<th>Scientific Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>German iris</td>
<td>Iris kochii</td>
</tr>
</tbody>
</table>

Related Terms:
- Iris
- Flag, flower
- Water iris
- German iris
- Habitant of Italy
- Iris germanica
- Iris kochii
- Iris flag, water flag
- Habitat of Italy
- Iris species

Images of the species.
Court, courtyard

An area wholly or partly surrounded by walls of buildings. "The house was built around an inner court."
Figure 2: An illustration of the architecture of our CNN, explicitly showing the delineation of responsibilities between the two GPUs. One GPU runs the layer-parts at the top of the figure while the other runs the layer-parts at the bottom. The GPUs communicate only at certain layers. The network's input is 150,528-dimensional, and the number of neurons in the network's remaining layers is given by 253,440–186,624–64,896–64,896–43,264–4096–4096–1000.

The second convolutional layer takes as input the (response-normalized and pooled) output of the first convolutional layer and filters it with 256 kernels of size $5 \times 5 \times 48$. The third, fourth, and fifth convolutional layers are connected to one another without any intervening pooling or normalization layers. The third convolutional layer has 384 kernels of size $3 \times 3 \times 256$ connected to the (normalized, pooled) outputs of the second convolutional layer. The fourth convolutional layer has 384 kernels of size $3 \times 3 \times 192$, and the fifth convolutional layer has 256 kernels of size $3 \times 3 \times 192$. The fully-connected layers have 4096 neurons each.

4 Reducing Overfitting

Our neural network architecture has 60 million parameters. Although the 1000 classes of ILSVRC make each training example impose 10 bits of constraint on the mapping from image to label, this turns out to be insufficient to learn so many parameters without considerable overfitting. Below, we describe the two primary ways in which we combat overfitting.

4.1 Data Augmentation

The easiest and most common method to reduce overfitting on image data is to artificially enlarge the dataset using label-preserving transformations (e.g., [25, 4, 5]). We employ two distinct forms of data augmentation, both of which allow transformed images to be produced from the original images with very little computation, so the transformed images do not need to be stored on disk. In our implementation, the transformed images are generated in Python code on the CPU while the GPU is training on the previous batch of images. So these data augmentation schemes are, in effect, computationally free.

The first form of data augmentation consists of generating image translations and horizontal reflections. We do this by extracting random $224 \times 224$ patches (and their horizontal reflections) from the $256 \times 256$ images and training our network on these extracted patches. This increases the size of our training set by a factor of 2048, though the resulting training examples are, of course, highly interdependent. Without this scheme, our network suffers from substantial overfitting, which would have forced us to use much smaller networks. At test time, the network makes a prediction by extracting five $224 \times 224$ patches (the four corner patches and the center patch) as well as their horizontal reflections (hence ten patches in all), and averaging the predictions made by the network's softmax layer on the ten patches.

The second form of data augmentation consists of altering the intensities of the RGB channels in training images. Specifically, we perform PCA on the set of RGB pixel values throughout the ImageNet training set. To each training image, we add multiples of the found principal components.

This is the reason why the input images in Figure 2 are $224 \times 224 \times 3$-dimensional.
Figure 2: An illustration of the architecture of our CNN, explicitly showing the delineation of responsibilities between the two GPUs. One GPU runs the layer-parts at the top of the figure while the other runs the layer-parts at the bottom. The GPUs communicate only at certain layers. The network’s input is 150,528-dimensional, and the number of neurons in the network’s remaining layers is given by 253,440–186,624–64,896–64,896–43,264–4096–4096–1000.

The second convolutional layer takes as input the (response-normalized and pooled) output of the first convolutional layer and filters it with 256 kernels of size $5 \times 5 \times 48$.

The third, fourth, and fifth convolutional layers are connected to one another without any intervening pooling or normalization layers. The third convolutional layer has 384 kernels of size $3 \times 3 \times 256$ connected to the (normalized, pooled) outputs of the second convolutional layer. The fourth convolutional layer has 384 kernels of size $3 \times 3 \times 192$, and the fifth convolutional layer has 256 kernels of size $3 \times 3 \times 192$.

The fully-connected layers have 4096 neurons each.

Reducing Overfitting

Our neural network architecture has 60 million parameters. Although the 1000 classes of ILSVRC make each training example impose 10 bits of constraint on the mapping from image to label, this turns out to be insufficient to learn so many parameters without considerable overfitting. Below, we describe the two primary ways in which we combat overfitting.

4.1 Data Augmentation

The easiest and most common method to reduce overfitting on image data is to artificially enlarge the dataset using label-preserving transformations (e.g., [25, 4, 5]). We employ two distinct forms of data augmentation, both of which allow transformed images to be produced from the original images with very little computation, so the transformed images do not need to be stored on disk. In our implementation, the transformed images are generated in Python code on the CPU while the GPU is training on the previous batch of images. So these data augmentation schemes are, in effect, computationally free.

The first form of data augmentation consists of generating image translations and horizontal reflections. We do this by extracting random $224 \times 224$ patches (and their horizontal reflections) from the $256 \times 256$ images and training our network on these extracted patches. This increases the size of our training set by a factor of 2048, though the resulting training examples are, of course, highly interdependent. Without this scheme, our network suffers from substantial overfitting, which would have forced us to use much smaller networks. At test time, the network makes a prediction by extracting five $224 \times 224$ patches (the four corner patches and the center patch) as well as their horizontal reflections (hence ten patches in all), and averaging the predictions made by the network’s softmax layer on the ten patches.

The second form of data augmentation consists of altering the intensities of the RGB channels in training images. Specifically, we perform PCA on the set of RGB pixel values throughout the ImageNet training set. To each training image, we add multiples of the found principal components.

This "softmax" layer is Logistic Regression!

The rest is just some fancy feature extraction (discussed later in the course).
LOGISTIC REGRESSION
Logistic Regression

**Data:** Inputs are continuous vectors of length $M$. Outputs are discrete.

$$
\mathcal{D} = \{ \mathbf{x}^{(i)}, y^{(i)} \}_{i=1}^{N} \text{ where } \mathbf{x} \in \mathbb{R}^{M} \text{ and } y \in \{0, 1\}
$$

We are back to classification.

Despite the name logistic **regression**.
Key idea: Try to learn this hyperplane directly

Directly modeling the hyperplane would use a decision function:

\[ h(x) = \text{sign}(\theta^T x) \]

for:

\[ y \in \{-1, +1\} \]

Looking ahead:
- We’ll see a number of commonly used Linear Classifiers
- These include:
  - Perceptron
  - Logistic Regression
  - Naïve Bayes (under certain conditions)
  - Support Vector Machines
Background: Hyperplanes

Hyperplane (Definition 1):
\[ \mathcal{H} = \{ \mathbf{x} : \mathbf{w}^T \mathbf{x} = b \} \]

Hyperplane (Definition 2):
\[ \mathcal{H} = \{ \mathbf{x} : \mathbf{\theta}^T \mathbf{x} = 0 \quad \text{and} \quad x_0 = 1 \} \]
\[ \mathbf{\theta} = [b, w_1, \ldots, w_M]^T \]

Half-spaces:
\[ \mathcal{H}^+ = \{ \mathbf{x} : \mathbf{\theta}^T \mathbf{x} > 0 \text{ and } x_0 = 1 \} \]
\[ \mathcal{H}^- = \{ \mathbf{x} : \mathbf{\theta}^T \mathbf{x} < 0 \text{ and } x_0 = 1 \} \]

Notation Trick: fold the bias \( b \) and the weights \( \mathbf{w} \) into a single vector \( \mathbf{\theta} \) by prepending a constant to \( \mathbf{x} \) and increasing dimensionality by one!
Using gradient ascent for linear classifiers

Key idea behind today’s lecture:

1. Define a linear classifier (logistic regression)
2. Define an objective function (likelihood)
3. Optimize it with gradient descent to learn parameters
4. Predict the class with highest probability under the model
Using gradient ascent for linear classifiers

This decision function isn’t differentiable:

\[ h(x) = \text{sign}(\theta^T x) \]

Use a differentiable function instead:

\[ p_{\theta}(y = 1 | x) = \frac{1}{1 + e^{-\theta^T x}} \]

\[ \text{logistic}(u) \equiv \frac{1}{1 + e^{-u}} \]
Using gradient ascent for linear classifiers

This decision function isn’t differentiable:

\[ h(x) = \text{sign}(\theta^T x) \]

Use a differentiable function instead:

\[ p_\theta(y = 1|x) = \frac{1}{1 + \exp(-\theta^T x)} \]

logistic(\( u \)) \( \equiv \) \( \frac{1}{1 + e^{-u}} \)
## Logistic Regression

### Data:
Inputs are continuous vectors of length $M$. Outputs are discrete.

$$D = \{x^{(i)}, y^{(i)}\}_{i=1}^N$$
where $x \in \mathbb{R}^M$ and $y \in \{0, 1\}$

### Model:
Logistic function applied to dot product of parameters with input vector.

$$p_\theta(y = 1|x) = \frac{1}{1 + \exp(-\theta^T x)}$$

### Learning:
finds the parameters that minimize some objective function.

$$\theta^* = \arg\min_\theta J(\theta)$$

### Prediction:
Output is the most probable class.

$$\hat{y} = \arg\max_{y \in \{0,1\}} p_\theta(y|x)$$
Logistic Regression

Whiteboard

– Bernoulli interpretation
– Logistic Regression Model
– Decision boundary
Learning for Logistic Regression

Whiteboard

– Partial derivative for Logistic Regression
– Gradient for Logistic Regression
LOGISTIC REGRESSION ON GAUSSIAN DATA
Logistic Regression
Logistic Regression

Classification with Logistic Regression
LEARNING LOGISTIC REGRESSION
Maximum Conditional Likelihood Estimation

Learning: finds the parameters that minimize some objective function.

\[ \theta^* = \arg\min_{\theta} J(\theta) \]

We minimize the negative log conditional likelihood:

\[ J(\theta) = -\log \prod_{i=1}^{N} p_{\theta}(y^{(i)} | x^{(i)}) \]

Why?

1. We can’t maximize likelihood (as in Naïve Bayes) because we don’t have a joint model \( p(x,y) \)
2. It worked well for Linear Regression (least squares is MCLE)
Maximum Conditional Likelihood Estimation

**Learning:** Four approaches to solving $\theta^* = \arg\min_{\theta} J(\theta)$

**Approach 1:** Gradient Descent
(take larger – more certain – steps opposite the gradient)

**Approach 2:** Stochastic Gradient Descent (SGD)
(take many small steps opposite the gradient)

**Approach 3:** Newton’s Method
(use second derivatives to better follow curvature)

**Approach 4:** Closed Form???
(set derivatives equal to zero and solve for parameters)
Learning: Four approaches to solving $\theta^* = \arg\min_{\theta} J(\theta)$

**Approach 1:** Gradient Descent
(take larger – more certain – steps opposite the gradient)

**Approach 2:** Stochastic Gradient Descent (SGD)
(take many small steps opposite the gradient)

**Approach 3:** Newton’s Method
(use second derivatives to better follow curvature)

**Approach 4:** Closed Form???
(set derivatives equal to zero and solve for parameters)

Logistic Regression does not have a closed form solution for MLE parameters.
SGD for Logistic Regression

Question:
Which of the following is a correct description of SGD for Logistic Regression?

Answer:
At each step (i.e. iteration) of SGD for Logistic Regression we...

A. (1) compute the gradient of the log-likelihood for all examples (2) update all the parameters using the gradient
B. (1) ask Matt for a description of SGD for Logistic Regression, (2) write it down, (3) report that answer
C. (1) compute the gradient of the log-likelihood for all examples (2) randomly pick an example (3) update only the parameters for that example
D. (1) randomly pick a parameter, (2) compute the partial derivative of the log-likelihood with respect to that parameter, (3) update that parameter for all examples
E. (1) randomly pick an example, (2) compute the gradient of the log-likelihood for that example, (3) update all the parameters using that gradient
F. (1) randomly pick a parameter and an example, (2) compute the gradient of the log-likelihood for that example with respect to that parameter, (3) update that parameter using that gradient
Algorithm 1 Gradient Descent

1: procedure GD(\(D, \theta^{(0)}\))
2: \(\theta \leftarrow \theta^{(0)}\)
3: while not converged do
4: \(\theta \leftarrow \theta - \gamma \nabla \theta J(\theta)\)
5: return \(\theta\)

In order to apply GD to Logistic Regression all we need is the gradient of the objective function (i.e. vector of partial derivatives).

\[
\nabla_\theta J(\theta) = \begin{bmatrix}
\frac{d}{d\theta_1} J(\theta) \\
\frac{d}{d\theta_2} J(\theta) \\
\vdots \\
\frac{d}{d\theta_M} J(\theta)
\end{bmatrix}
\]
Stochastic Gradient Descent (SGD)

We can also apply SGD to solve the MCLE problem for Logistic Regression.

We need a per-example objective:

Let $J(\theta) = \sum_{i=1}^{N} J^{(i)}(\theta)$

where $J^{(i)}(\theta) = -\log p_\theta(y^i|x^i)$. 
Logistic Regression vs. Perceptron

Question:
True or False: Just like Perceptron, one step (i.e. iteration) of SGD for Logistic Regression will result in a change to the parameters only if the current example is incorrectly classified.

Answer:
## Matching Game

**Goal:** Match the Algorithm to its Update Rule

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Update Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. SGD for Logistic Regression</td>
<td>[ h_\theta(x) = p(y</td>
</tr>
<tr>
<td>2. Least Mean Squares</td>
<td>[ h_\theta(x) = \theta^T x ]</td>
</tr>
<tr>
<td>3. Perceptron</td>
<td>[ h_\theta(x) = \text{sign}(\theta^T x) ]</td>
</tr>
</tbody>
</table>

| \[ \theta_k \leftarrow \theta_k + (h_\theta(x^{(i)}) - y^{(i)}) \] | 4. |
| \[ \theta_k \leftarrow \theta_k + \frac{1}{1 + \exp \lambda(h_\theta(x^{(i)}) - y^{(i)})} \] | 5. |
| \[ \theta_k \leftarrow \theta_k + \lambda(h_\theta(x^{(i)}) - y^{(i)})x_k^{(i)} \] | 6. |

A. 1=5, 2=4, 3=6
B. 1=5, 2=6, 3=4
C. 1=6, 2=4, 3=4
D. 1=5, 2=6, 3=6
E. 1=6, 2=6, 3=6
F. 1=6, 2=5, 3=5
G. 1=5, 2=5, 3=5
H. 1=4, 2=5, 3=6
OPTIMIZATION METHOD #4:
MINI-BATCH SGD
Mini-Batch SGD

• **Gradient Descent:**
  Compute true gradient exactly from all N examples

• **Stochastic Gradient Descent (SGD):**
  Approximate true gradient by the gradient of one randomly chosen example

• **Mini-Batch SGD:**
  Approximate true gradient by the average gradient of K randomly chosen examples
Mini-Batch SGD

while not converged: \( \theta \leftarrow \theta - \lambda g \)

Three variants of first-order optimization:

Gradient Descent: \( g = \nabla J(\theta) = \frac{1}{N} \sum_{i=1}^{N} \nabla J^{(i)}(\theta) \)

SGD: \( g = \nabla J^{(i)}(\theta) \) where \( i \) sampled uniformly

Mini-batch SGD: \( g = \frac{1}{S} \sum_{s=1}^{S} \nabla J^{(i_s)}(\theta) \) where \( i_s \) sampled uniformly \( \forall s \)
Summary

1. Discriminative classifiers directly model the conditional, $p(y|x)$

2. Logistic regression is a simple linear classifier, that retains a probabilistic semantics

3. Parameters in LR are learned by iterative optimization (e.g. SGD)
Logistic Regression Objectives

You should be able to...

• Apply the principle of maximum likelihood estimation (MLE) to learn the parameters of a probabilistic model
• Given a discriminative probabilistic model, derive the conditional log-likelihood, its gradient, and the corresponding Bayes Classifier
• Explain the practical reasons why we work with the \text{log} of the likelihood
• Implement logistic regression for binary or multiclass classification
• Prove that the decision boundary of binary logistic regression is linear
• For linear regression, show that the parameters which minimize squared error are equivalent to those that maximize conditional likelihood
MULTINOMIAL LOGISTIC REGRESSION
Multinomial Logistic Regression

**Chalkboard**

- Background: Multinomial distribution
- Definition: Multi-class classification
- Geometric intuitions
- Multinomial logistic regression model
- Generative story
- Reduction to binary logistic regression
- Partial derivatives and gradients
- Applying Gradient Descent and SGD
- Implementation w/ sparse features
Debug that Program!

**In-Class Exercise: Think-Pair-Share**
Debug the following program which is (incorrectly) attempting to run SGD for multinomial logistic regression

**Buggy Program:**

```python
while not converged:
    for i in shuffle([1,...,N]):
        for k in [1,...,K]:
            theta[k] = theta[k] - lambda * grad(x[i], y[i], theta, k)
```

Assume: `grad(x[i], y[i], theta, k)` returns the gradient of the negative log-likelihood of the training example `(x[i],y[i])` with respect to vector `theta[k]`. `lambda` is the learning rate. `N = # of examples. K = # of output classes. M = # of features. theta is a K by M matrix.`