Optimization

Gokberk Cinbis
Recap

Suppose: 3 training examples, 3 classes.
With some $W$ the scores $f(x, W) = Wx$ are:

<table>
<thead>
<tr>
<th></th>
<th>cat</th>
<th>3.2</th>
<th>1.3</th>
<th>2.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>car</td>
<td>5.1</td>
<td>4.9</td>
<td>2.5</td>
<td></td>
</tr>
<tr>
<td>frog</td>
<td>-1.7</td>
<td>2.0</td>
<td>-3.1</td>
<td></td>
</tr>
</tbody>
</table>

Losses: 2.9 0 10.9

Multiclass SVM loss:

Given an example $\left( x_i, y_i \right)$ where $x_i$ is the image and $y_i$ is the (integer) label, and using the shorthand for the scores vector:

$s = f(x_i, W)$

the SVM loss has the form:

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

$$= \max(0, 2.2 - (-3.1) + 1) + \max(0, 2.5 - (-3.1) + 1)$$

$$= \max(0, 5.3) + \max(0, 5.6)$$

$$= 5.3 + 5.6$$

$$= 10.9$$
Recap

- We have some dataset of \((x, y)\)
- We have a score function: \(s = f(x; W) = Wx\)
- We have a loss function:

\[
L_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right)
\]

\[
L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)
\]

\[
L = \frac{1}{N} \sum_{i=1}^{N} L_i + R(W) \quad \text{Full loss}
\]
Strategy #1: A first very bad idea solution: Random search

```python
# assume X_train is the data where each column is an example (e.g. 3073 x 50,000)
# assume Y_train are the labels (e.g. 1D array of 50,000)
# assume the function L evaluates the loss function

bestloss = float("inf")  # Python assigns the highest possible float value
for num in xrange(1000):
    W = np.random.randn(10, 3073) * 0.0001  # generate random parameters
    loss = L(X_train, Y_train, W)  # get the loss over the entire training set
    if loss < bestloss:  # keep track of the best solution
        bestloss = loss
        bestW = W
    print 'in attempt %d the loss was %f, best %f' % (num, loss, bestloss)

# prints:
# in attempt 0 the loss was 9.401632, best 9.401632
# in attempt 1 the loss was 8.959668, best 8.959668
# in attempt 2 the loss was 9.044034, best 8.959668
# in attempt 3 the loss was 9.278948, best 8.959668
# in attempt 4 the loss was 8.857370, best 8.857370
# in attempt 5 the loss was 8.943151, best 8.857370
# in attempt 6 the loss was 8.605604, best 8.605604
# ... (truncated: continues for 1000 lines)
```
Lets see how well this works on the test set...

```python
# Assume X_test is [3073 x 10000], Y_test [10000 x 1]
scores = Wbest.dot(Xte_cols) # 10 x 10000, the class scores for all test examples
# find the index with max score in each column (the predicted class)
Yte_predict = np.argmax(scores, axis = 0)
# and calculate accuracy (fraction of predictions that are correct)
np.mean(Yte_predict == Yte)
# returns 0.1555
```

15.5% accuracy! not bad!
(SOTA is ~95%)
Slide by Fei-Fei Li, Andrej Karpathy & Justin Johnson

Intro. to CV
Strategy #2: Follow the slope

In 1-dimension, the derivative of a function:

\[
\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x + h) - f(x)}{h}
\]

In multiple dimensions, the gradient is the vector of (partial derivatives).
current \( W \): 

\[
[0.34, -1.11, 0.78, 0.12, 0.55, 2.81, -3.1, -1.5, 0.33, \ldots]
\]

loss 1.25347

gradient \( dW \): 

\[
[?, ?, ?, ?, ?, ?, ?, ?, ?, \ldots]
\]
current W:

\[ [0.34, -1.11, 0.78, 0.12, 0.55, 2.81, -3.1, -1.5, 0.33, \ldots] \]

loss 1.25347

gradient dW:

current $W$:

\[
\begin{bmatrix}
0.34, \\
-1.11, \\
0.78, \\
0.12, \\
0.55, \\
2.81, \\
-3.1, \\
-1.5, \\
0.33,...
\end{bmatrix}
\]

loss 1.25347

$W + h$ (first dim):

\[
\begin{bmatrix}
0.34 + 0.0001, \\
-1.11, \\
0.78, \\
0.12, \\
0.55, \\
2.81, \\
-3.1, \\
-1.5, \\
0.33,...
\end{bmatrix}
\]

loss 1.25322

gradient $dW$:

\[
\begin{bmatrix}
?, \\
?, \\
?, \\
?, \\
?, \\
?, \\
?, \\
?, \\
?,
\end{bmatrix}
\]
current $W$:  

\[
\begin{bmatrix}
0.34, \\
-1.11, \\
0.78, \\
0.12, \\
0.55, \\
2.81, \\
-3.1, \\
-1.5, \\
0.33, ...
\end{bmatrix}
\]

loss 1.25347

$W + h$ (first dim):  

\[
\begin{bmatrix}
0.34 + 0.0001, \\
-1.11, \\
0.78, \\
0.12, \\
0.55, \\
2.81, \\
-3.1, \\
-1.5, \\
0.33, ...
\end{bmatrix}
\]

loss 1.25322

gradient $dW$:  

\[
\begin{bmatrix}
-2.5, \\
?, \\
?, \\
?, \\
?, \\
(1.25322 - 1.25347)/0.0001 = -2.5
\end{bmatrix}
\]
<table>
<thead>
<tr>
<th>current $W$:</th>
<th>$W + h$ (second dim):</th>
<th>gradient $dW$:</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0.34,</td>
<td>[0.34,</td>
<td>[-2.5,</td>
</tr>
<tr>
<td>-1.11,</td>
<td>-1.11 + 0.0001,</td>
<td>? ,</td>
</tr>
<tr>
<td>0.78,</td>
<td>0.78,</td>
<td>? ,</td>
</tr>
<tr>
<td>0.12,</td>
<td>0.12,</td>
<td>? ,</td>
</tr>
<tr>
<td>0.55,</td>
<td>0.55,</td>
<td>? ,</td>
</tr>
<tr>
<td>2.81,</td>
<td>2.81,</td>
<td>? ,</td>
</tr>
<tr>
<td>-3.1,</td>
<td>-3.1,</td>
<td>? ,</td>
</tr>
<tr>
<td>-1.5,</td>
<td>-1.5,</td>
<td>? ,</td>
</tr>
<tr>
<td>0.33,...]</td>
<td>0.33,...]</td>
<td>? ,...</td>
</tr>
<tr>
<td><strong>loss 1.25347</strong></td>
<td><strong>loss 1.25353</strong></td>
<td></td>
</tr>
</tbody>
</table>

Slide by Fei-Fei Li, Andrej Karpathy & Justin Johnson
current $W$: [0.34, -1.11, 0.78, 0.12, 0.55, 2.81, -3.1, -1.5, 0.33, ...]

loss 1.25347

$W + h$ (second dim): [0.34, -1.11 + 0.0001, 0.78, 0.12, 0.55, 2.81, -3.1, -1.5, 0.33, ...]

loss 1.25353

gradient $dW$: [-2.5, 0.6, ?, ?, ?, ?]

(1.25353 - 1.25347)/0.0001 = 0.6

\[
\frac{df(x)}{dx} = \lim_{{h \to 0}} \frac{f(x + h) - f(x)}{h}
\]
current $W$:  

\[
\begin{bmatrix}
0.34, \\
-1.11, \\
0.78, \\
0.12, \\
0.55, \\
2.81, \\
-3.1, \\
-1.5, \\
0.33, \\
0.33, \\
\end{bmatrix}
\]

loss 1.25347

$W + h$ (third dim):  

\[
\begin{bmatrix}
0.34, \\
-1.11, \\
0.78 + 0.0001, \\
0.12, \\
0.55, \\
2.81, \\
-3.1, \\
-1.5, \\
0.33, \\
0.33, \\
\end{bmatrix}
\]

loss 1.25347

gradient $dW$:  

\[
\begin{bmatrix}
-2.5, \\
0.6, \\
?, \\
?, \\
?, \\
?, \\
?, \\
?, \\
?, \\
?,
\end{bmatrix}
\]
**current W:**

\[\begin{align*}
0.34, \\
-1.11, \\
0.78, \\
0.12, \\
0.55, \\
2.81, \\
-3.1, \\
-1.5, \\
0.33, \ldots
\end{align*}\]

**loss 1.25347**

**W + h (third dim):**

\[\begin{align*}
0.34, \\
-1.11, \\
0.78 + 0.0001, \\
0.12, \\
0.55, \\
2.81, \\
-3.1, \\
-1.5, \\
0.33, \ldots
\end{align*}\]

**loss 1.25347**

**gradient dW:**

\[\begin{align*}
-2.5, \\
0.6, \\
0, \\
?, \\
?, \\
?, \ldots
\end{align*}\]

\[
\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x + h) - f(x)}{h}
\]

\[
\frac{(1.25347 - 1.25347)}{0.0001} = 0
\]
Evaluation the gradient numerically

\[
\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x + h) - f(x)}{h}
\]

```python
def eval_numerical_gradient(f, x):
    
    a naive implementation of numerical gradient of f at x
    - f should be a function that takes a single argument
    - x is the point (numpy array) to evaluate the gradient at
    
    fx = f(x) # evaluate function value at original point
    grad = np.zeros(x.shape)
    h = 0.00001

    # iterate over all indexes in x
    it = np.nditer(x, flags=['multi_index'], op_flags=['readwrite'])
    while not it.finished:
        
        # evaluate function at x+h
        ix = it.multi_index
        old_value = x[ix]
        x[ix] = old_value + h # increment by h
        fxh = f(x) # evaluate f(x + h)
        x[ix] = old_value # restore to previous value (very important!)

        # compute the partial derivative
        grad[ix] = (fxh - fx) / h # the slope
        it.iternext() # step to next dimension

    return grad
```
Evaluation the gradient numerically

\[ \frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x + h) - f(x)}{h} \]

- approximate
- very slow to evaluate

```python
def eval_numerical_gradient(f, x):
    
    a naive implementation of numerical gradient of f at x
    - f should be a function that takes a single argument
    - x is the point (numpy array) to evaluate the gradient at
    
    fx = f(x) # evaluate function value at original point
    grad = np.zeros(x.shape)
    h = 0.00001

    # iterate over all indexes in x
    it = np.nditer(x, flags=['multi_index'], op_flags=['readwrite'])
    while not it.finished:

        # evaluate function at x+h
        ix = it.multi_index
        old_value = x[ix]
        x[ix] = old_value + h # increment by h
        fxh = f(x) # evaluate f(x + h)
        x[ix] = old_value # restore to previous value (very important!)

        # compute the partial derivative
        grad[ix] = (fxh - fx) / h # the slope
        it.iternext() # step to next dimension

    return grad
```
This is silly. The loss is just a function of $W$:

\[ L = \frac{1}{N} \sum_{i=1}^{N} L_i + \sum_k W_k^2 \]

\[ L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1) \]

\[ s = f(x; W) = Wx \]

want $\nabla_W L$
This is silly. The loss is just a function of $W$:

$$L = \frac{1}{N} \sum_{i=1}^{N} L_i + \sum_k W_k^2$$

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

$$s = f(x; W) = Wx$$

want $\nabla_W L$
This is silly. The loss is just a function of $W$:

$$L = \frac{1}{N} \sum_{i=1}^{N} L_i + \sum_k W_k^2$$

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

$$s = f(x; W) = Wx$$

$$\nabla_W L = ...$$
current $W$: 

$[0.34, -1.11, 0.78, 0.12, 0.55, 2.81, -3.1, -1.5, -0.33, ...]$ 

loss 1.25347

gradient $dW$: 

$[-2.5, 0.6, 0, 0.2, 0.7, -0.5, 1.1, 1.3, -2.1, ...]$ 

$dW = ...$  
(some function data and $W$)
One more example on efficiency

- Consider the linear regression model:

\[ f(w) = \sum_{i=1}^{N} (w^T x_i - y_i)^2 \]

- Minimizing \( f(w) \) learns a mapping from \( x \) (vector) to \( y \) (scalar):

\[ y = w^T x \]
Linear regression - numerical vs analytic gradient

Numerical gradient, for each $j$:

$$\frac{f(w + he_j) - f(w)}{h} = \sum_{i=1}^{N} \frac{((w + he_j)^T x_i - y_i)^2 - f(w)}{h}$$

(h: some small positive number, $e_j$ is standard unit vector at dim $j$)
Linear regression - numerical vs analytic gradient

Numerical gradient, for each $j$:

$$\frac{f(w + he_j) - f(w)}{h} = \sum_{i=1}^{N} \frac{((w + he_j)^T x_i - y_i)^2 - f(w)}{h}$$

($h$: some small positive number, $e_j$ is standard unit vector at dim $j$)

Analytic gradient:

$$\nabla f = 2 \sum_{i=1}^{N} x_i (w^T x_i - y_i)$$
Linear regression - numerical vs analytic gradient

Numerical gradient, for each $j$:

$$\frac{f(w + he_j) - f(w)}{h} = \frac{\sum_{i=1}^{N}((w + he_j)^T x_i - y_i)^2 - f(w)}{h}$$

(h: some small positive number, $e_j$ is standard unit vector at dim $j$)

Analytic gradient:

$$\nabla f = 2 \sum_{i=1}^{N} x_i (w^T x_i - y_i)$$

Which one is faster, why?
Linear regression - numerical vs analytic gradient

Numerical gradient, for each $j$:

$$\frac{f(w + he_j) - f(w)}{h} = \sum_{i=1}^{N} \frac{((w + he_j)^T x_i - y_i)^2 - f(w)}{h}$$

(h: some small positive number, $e_j$ is standard unit vector at dim $j$)

Analytic gradient:

$$\nabla f = 2 \sum_{i=1}^{N} x_i (w^T x_i - y_i)$$

All partial derivatives can be computed with a single loop (in total) over the examples
In summary:

- Numerical gradient: approximate, slow, easy to write
- Analytic gradient: exact, fast, error-prone

=>

**In practice:** Always use analytic gradient, but check implementation with numerical gradient. This is called a gradient check.
Gradient Descent

$$\min_x f(x)$$

# Vanilla Gradient Descent

```python
while True:
    weights_grad = evaluate_gradient(loss_fun, data, weights)
    weights += -step_size * weights_grad  # perform parameter update
```
Positive vs negative gradient
negative gradient direction
Remember the web demo

http://vision.stanford.edu/teaching/cs231n/linear-classify-demo/

Slide by Fei-Fei Li, Andrej Karpathy & Justin Johnson
Mini-batch Gradient Descent

- only use a small portion of the training set to compute the gradient.

```python
# Vanilla Minibatch Gradient Descent

while True:
    data_batch = sample_training_data(data, 256)  # sample 256 examples
    weights_grad = evaluate_gradient(loss_fun, data_batch, weights)
    weights += -step_size * weights_grad  # perform parameter update
```

Common mini-batch sizes are 32/64/128 examples
e.g. Krizhevsky ILSVRC ConvNet used 256 examples
Example of optimization progress while training a neural network.

(Loss over mini-batches goes down over time.)
The effects of step size (or "learning rate")
Mini-batch Gradient Descent

- only use a small portion of the training set to compute the gradient.

```python
# Vanilla Minibatch Gradient Descent

while True:
    data_batch = sample_training_data(data, 256) # sample 256 examples
    weights_grad = evaluate_gradient(loss_fun, data_batch, weights)
    weights += - step_size * weights_grad # perform parameter update
```

Common mini-batch sizes are 32/64/128 examples
e.g. Krizhevsky ILSVRC ConvNet used 256 examples

we will later look at more fancy update formulas
(momentum, Adagrad, RMSProp, Adam, …)
The effects of different update form formulas

(image credits to Alec Radford)
Next topic:

Backpropagation