Lecture 3: Regularization and Optimization
Administrative: Assignment 1

Released last week, due **Fri 4/15 at 11:59pm**
Administrative: Project proposal

Due **Mon 4/18**

TA expertise are posted on the webpage.

(http://cs231n.stanford.edu/office_hours.html)
Administrative: Ed

Please make sure to check and read all pinned Ed posts.
Image Classification: A core task in Computer Vision

(assume given a set of labels)
{dog, cat, truck, plane, ...}
Recall from last time: Challenges of recognition

- Viewpoint
- Illumination
- Deformation
- Occlusion
- Clutter
- Intraclass Variation
Recall from last time: data-driven approach, kNN

1-NN classifier

5-NN classifier

train

validation

test

train

test

Fei-Fei Li, Jiajun Wu, Ruohan Gao

Lecture 3 - 7

April 05, 2022
Recall from last time: Linear Classifier

\[ f(x, W) = Wx + b \]

**Algebraic Viewpoint**

\[ f(x, W) = Wx \]

**Visual Viewpoint**

One template per class

**Geometric Viewpoint**

Hyperplanes cutting up space

Class 1:
1 \( \leq \) L2 norm \( \leq \) 2

Class 2:
Everything else

Class 2:
Everything else
Interpreting a Linear Classifier: **Visual Viewpoint**

- airplane
- automobile
- bird
- cat
- deer
- dog
- frog
- horse
- ship
- truck

**Input image**

```
  56  231
   24   2

W
- 0.2 -0.5
  0.1  2.0
  1.5  1.3
  0.2 -0.3

b
  1.1
  3.2
- 1.2

Score
  -96.8
  437.9
  61.95```

Fei-Fei Li, Jiajun Wu, Ruohan Gao
Example with an image with 4 pixels, and 3 classes (cat/dog/ship)

**Algebraic Viewpoint**

\[ f(x, W) = Wx \]

**Input image**

W

b

**Score**

Cat score

Dog score

Ship score

\[ W \]

\[ b \]

\[ -96.8 \]

\[ 437.9 \]

\[ 61.95 \]

\[ 0.2 \]

\[ -0.5 \]

\[ 1.5 \]

\[ 1.3 \]

\[ 0.1 \]

\[ 2.0 \]

\[ 2.1 \]

\[ 0.0 \]

\[ 0 \]

\[ 0.25 \]

\[ 56 \]

\[ 231 \]

\[ 24 \]

\[ 2 \]
Interpreting a Linear Classifier: Geometric Viewpoint

\[ f(x, W) = Wx + b \]

Array of 32x32x3 numbers (3072 numbers total)
Suppose: 3 training examples, 3 classes. With some $W$ the scores $f(x, W) = W x$ are:

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

A **loss function** tells how good our current classifier is.

Given a dataset of examples $\{(x_i, y_i)\}_{i=1}^N$.

Where $x_i$ is image and $y_i$ is (integer) label.

Loss over the dataset is an average of loss over examples:

$$L = \frac{1}{N} \sum_i L_i(f(x_i, W), y_i)$$
Softmax vs. SVM

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

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

Matrix multiply + bias offset

\( W \)

\[ \begin{bmatrix}
0.01 & -0.05 & 0.1 & 0.05 \\
0.7 & 0.2 & 0.05 & 0.16 \\
0.0 & -0.45 & -0.2 & 0.03 \\
\end{bmatrix} \]

\( b \)

\[ \begin{bmatrix}
-15 \\
22 \\
-44 \\
56 \\
\end{bmatrix} + \begin{bmatrix}
0.0 \\
0.2 \\
-0.3 \\
\end{bmatrix} \]

Hinge loss (SVM)

\[ \max(0, -2.85 - 0.28 + 1) + \max(0, 0.86 - 0.28 + 1) = 1.58 \]

cross-entropy loss (Softmax)

\[ \exp \]

\[ \begin{bmatrix}
-2.85 \\
0.86 \\
0.28 \\
\end{bmatrix} \]

\[ \begin{bmatrix}
0.058 \\
2.36 \\
1.32 \\
\end{bmatrix} \]

\[ \begin{bmatrix}
0.016 \\
0.631 \\
0.353 \\
\end{bmatrix} \]

\[ -\log(0.353) = 0.452 \]
\[ f(x, W) = Wx \]

\[ L = \frac{1}{N} \sum_{i=1}^{N} \sum_{j \neq y_i} \max(0, f(x_i; W)_j - f(x_i; W)_{y_i} + 1) \]

Q: Suppose that we found a \( W \) such that \( L = 0 \). Is this \( W \) unique?
\[ f(x, W) = Wx \]

\[ L = \frac{1}{N} \sum_{i=1}^{N} \sum_{j \neq y_i} \max(0, f(x_i; W)_j - f(x_i; W)_{y_i} + 1) \]

E.g. Suppose that we found a \( W \) such that \( L = 0 \). Is this \( W \) unique?

No! 2\( W \) is also has \( L = 0 \)!
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></th>
<th>car</th>
<th></th>
<th>frog</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3.2</td>
<td>1.3</td>
<td>4.9</td>
<td>2.0</td>
<td>-1.7</td>
<td>2.2</td>
</tr>
<tr>
<td>Losses:</td>
<td>2.9</td>
<td>0</td>
<td>0</td>
<td></td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

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

Before:
\[
= \max(0, 1.3 - 4.9 + 1) + \max(0, 2.0 - 4.9 + 1)
= \max(0, -2.6) + \max(0, -1.9)
= 0 + 0
= 0
\]

With $W$ twice as large:
\[
= \max(0, 2.6 - 9.8 + 1) + \max(0, 4.0 - 9.8 + 1)
= \max(0, -6.2) + \max(0, -4.8)
= 0 + 0
= 0
\]
\[ f(x, W) = Wx \]

\[ L = \frac{1}{N} \sum_{i=1}^{N} \sum_{j \neq y_i} \max(0, f(x_i; W)_j - f(x_i; W)_{y_i} + 1) \]

E.g. Suppose that we found a \( W \) such that \( L = 0 \). Is this \( W \) unique?

No! 2\( W \) is also has \( L = 0! \)

How do we choose between \( W \) and 2\( W \)?
Regularization -

\[ L(W) = \frac{1}{N} \sum_{i=1}^{N} L_i(f(x_i, W), y_i) \]

**Data loss**: Model predictions should match training data
Regularization

\[ L(W) = \frac{1}{N} \sum_{i=1}^{N} L_i(f(x_i, W), y_i) + \lambda R(W) \]

**Data loss:** Model predictions should match training data

**Regularization:** Prevent the model from doing *too* well on training data
Regularization intuition: toy example training data
Regularization intuition: Prefer Simpler Models

\[ f_1 \]

\[ f_2 \]
Regularization: Prefer Simpler Models

Regularization pushes against fitting the data too well so we don’t fit noise in the data.
Regularization

\[ L(W) = \frac{1}{N} \sum_{i=1}^{N} L_i(f(x_i, W), y_i) + \lambda R(W) \]

**Data loss**: Model predictions should match training data

**Regularization**: Prevent the model from doing *too* well on training data

**Occam’s Razor**: Among multiple competing hypotheses, the simplest is the best, William of Ockham 1285-1347
Regularization

\[ L(W) = \frac{1}{N} \sum_{i=1}^{N} L_i(f(x_i, W), y_i) + \lambda R(W) \]

- **Data loss**: Model predictions should match training data
- **Regularization**: Prevent the model from doing too well on training data

\[ \lambda \text{ = regularization strength (hyperparameter)} \]
Regularization

\[ L(W) = \frac{1}{N} \sum_{i=1}^{N} L_i(f(x_i, W), y_i) + \lambda R(W) \]

Data loss: Model predictions should match training data

Regularization: Prevent the model from doing too well on training data

Simple examples

L2 regularization: \( R(W) = \sum_k \sum_l W_{k,l}^2 \)
L1 regularization: \( R(W) = \sum_k \sum_l |W_{k,l}| \)
Elastic net (L1 + L2): \( R(W) = \sum_k \sum_l \beta W_{k,l}^2 + |W_{k,l}| \)

\( \lambda \) = regularization strength (hyperparameter)
Regularization

\[ L(W) = \frac{1}{N} \sum_{i=1}^{N} L_i(f(x_i, W), y_i) + \lambda R(W) \]

**Data loss:** Model predictions should match training data

**Regularization:** Prevent the model from doing too well on training data

\( \lambda = \) regularization strength (hyperparameter)

**Simple examples**
- **L2 regularization:** \( R(W) = \sum_k \sum_l W_{k,l}^2 \)
- **L1 regularization:** \( R(W) = \sum_k \sum_l |W_{k,l}| \)
- **Elastic net (L1 + L2):** \( R(W) = \sum_k \sum_l \beta W_{k,l}^2 + |W_{k,l}| \)

**More complex:**
- Dropout
- Batch normalization
- Stochastic depth, fractional pooling, etc
Regularization

$$L(W) = \frac{1}{N} \sum_{i=1}^{N} L_i(f(x_i, W), y_i) + \lambda R(W)$$

\(\lambda\) = regularization strength (hyperparameter)

Data loss: Model predictions should match training data

Regularization: Prevent the model from doing too well on training data

Why regularize?
- Express preferences over weights
- Make the model simple so it works on test data
- Improve optimization by adding curvature
Regularization: Expressing Preferences

\[
x = [1, 1, 1, 1]
\]
\[
w_1 = [1, 0, 0, 0]
\]
\[
w_2 = [0.25, 0.25, 0.25, 0.25]
\]
\[
w_1^T x = w_2^T x = 1
\]

L2 Regularization

\[
R(W) = \sum_k \sum_l W_{k,l}^2
\]

Which of \(w_1\) or \(w_2\) will the L2 regularizer prefer?
Regularization: Expressing Preferences

L2 Regularization

\[ R(W) = \sum_k \sum_l W_{k,l}^2 \]

Which of \( w_1 \) or \( w_2 \) will the L2 regularizer prefer?

L2 regularization likes to “spread out” the weights

\[
\begin{align*}
    x &= [1, 1, 1, 1] \\
    w_1 &= [1, 0, 0, 0] \\
    w_2 &= [0.25, 0.25, 0.25, 0.25] \\
    w_1^T x &= w_2^T x = 1
\end{align*}
\]
Regularization: Expressing Preferences

\[ x = [1, 1, 1, 1, 1] \]
\[ w_1 = [1, 0, 0, 0, 0] \]
\[ w_2 = [0.25, 0.25, 0.25, 0.25, 0.25] \]

L2 Regularization

\[ R(W) = \sum_k \sum_l W_{k,l}^2 \]

Which of \( w_1 \) or \( w_2 \) will the L2 regularizer prefer?

L2 regularization likes to “spread out” the weights

Which one would L1 regularization prefer?

\[ w_1^T x = w_2^T x = 1 \]
Recap

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

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

**Softmax**

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

**SVM**

\[
L = \frac{1}{N} \sum_{i=1}^{N} L_i + R(W)
\]

**Full loss**
Recap

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

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

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

\[
L = \frac{1}{N} \sum_{i=1}^{N} L_i + R(W) \quad \text{Full loss}
\]
Interactive Web Demo

http://vision.stanford.edu/teaching/cs231n-demos/linear-classify/
Optimization
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 9.278940, 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)
```
Let's 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 ~99.7%)
Strategy #2: Follow the slope
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) along each dimension.

The slope in any direction is the **dot product** of the direction with the gradient. The direction of steepest descent is the **negative gradient**.
current $W$: 

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

loss 1.25347

gradient $dW$: 

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

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

loss 1.25347

$W + h$ (first dim):  

$$[0.34 + 0.0001, -1.11, 0.78, 0.12, 0.55, 2.81, -3.1, -1.5, 0.33, ...]$$  

loss 1.25322

gradient $dW$:  


$$(1.25322 - 1.25347)/0.0001 = -2.5$$

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

$$?, ?, ?, ...$$
<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, ( \ldots )</td>
<td>0.33, ( \ldots )</td>
<td>?</td>
</tr>
<tr>
<td><strong>loss 1.25347</strong></td>
<td><strong>loss 1.25353</strong></td>
<td><strong>( \ldots )</strong></td>
</tr>
</tbody>
</table>

Fei-Fei Li, Jiajun Wu, Ruohan Gao
current $W$: | $W + h$ (second dim): |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>[0.34, -1.11, 0.78, 0.12, 0.55, 2.81, -3.1, -1.5, 0.33,...]</td>
<td>[0.34, -1.11 + 0.0001, 0.78, 0.12, 0.55, 2.81, -3.1, -1.5, 0.33,...]</td>
</tr>
<tr>
<td>loss 1.25347</td>
<td>loss 1.25353</td>
</tr>
</tbody>
</table>

gradient $dW$: 

$[-2.5, 0.6, ?, ?, ?, ?,...]$

$(1.25353 - 1.25347)/0.0001 = 0.6$

\[
\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x + h) - f(x)}{h}
\]
<table>
<thead>
<tr>
<th>current $W$:</th>
<th>$W + h$ (third dim):</th>
<th>gradient $dW$:</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0.34, -1.11, 0.78, 0.12, 0.55, 2.81, -3.1, -1.5, 0.33, ...]</td>
<td>[0.34, -1.11, 0.78 + 0.0001, 0.12, 0.55, 2.81, -3.1, -1.5, 0.33, ...]</td>
<td>[-2.5, 0.6, ?, ?, ?, ?, ?, ?, ?, ...]</td>
</tr>
<tr>
<td>loss 1.25347</td>
<td>loss 1.25347</td>
<td></td>
</tr>
</tbody>
</table>
\[
\text{current } W: \quad \begin{bmatrix} 0.34, \\ -1.11, \\ 0.78, \\ 0.12, \\ 0.55, \\ 2.81, \\ -3.1, \\ -1.5, \\ 0.33, \ldots \end{bmatrix}, \quad \text{loss } 1.25347
\]

\[
\text{current } W + h \text{ (third dim): } \begin{bmatrix} 0.34, \\ -1.11, \\ 0.78 + 0.0001, \\ 0.12, \\ 0.55, \\ 2.81, \\ -3.1, \\ -1.5, \\ 0.33, \ldots \end{bmatrix}, \quad \text{loss } 1.25347
\]

\[
\text{gradient } dW: \begin{bmatrix} -2.5, \\ 0.6, \\ 0, \\ 0, \\ 0, \\ 0, \\ 0, \\ 0, \\ 0, \ldots \end{bmatrix}, \quad \frac{1.25347 - 1.25347}{0.0001} = 0
\]
### current W:

<table>
<thead>
<tr>
<th>0.34,</th>
<th>0.34,</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1.11,</td>
<td>-1.11,</td>
</tr>
<tr>
<td>0.78,</td>
<td>0.78 + 0.0001,</td>
</tr>
<tr>
<td>0.12,</td>
<td>0.12,</td>
</tr>
<tr>
<td>0.55,</td>
<td>0.55,</td>
</tr>
<tr>
<td>2.81,</td>
<td>2.81,</td>
</tr>
<tr>
<td>-3.1,</td>
<td>-3.1,</td>
</tr>
<tr>
<td>-1.5,</td>
<td>-1.5,</td>
</tr>
<tr>
<td>0.33,...]</td>
<td>0.33,...]</td>
</tr>
</tbody>
</table>

**loss 1.25347**

### W + h (third dim):

<table>
<thead>
<tr>
<th>0.34,</th>
<th>0.34,</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1.11,</td>
<td>-1.11,</td>
</tr>
<tr>
<td>0.78,</td>
<td>0.78,</td>
</tr>
<tr>
<td>0.12,</td>
<td>0.12,</td>
</tr>
<tr>
<td>0.55,</td>
<td>0.55,</td>
</tr>
<tr>
<td>2.81,</td>
<td>2.81,</td>
</tr>
<tr>
<td>-3.1,</td>
<td>-3.1,</td>
</tr>
<tr>
<td>-1.5,</td>
<td>-1.5,</td>
</tr>
<tr>
<td>0.33,...]</td>
<td>0.33,...]</td>
</tr>
</tbody>
</table>

**loss 1.25347**

### gradient dW:

| [-2.5,   |
| 0.6,    |
| 0,      |
| ?,      |
| ?,      |
| ?, ...] |

**Numeric Gradient**

- Slow! Need to loop over all dimensions
- Approximate
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 $\nabla_W L$

Use calculus to compute an analytic gradient
current $W$: 

$$\begin{bmatrix}
0.34, \\
-1.11, \\
0.78, \\
0.12, \\
0.55, \\
2.81, \\
-3.1, \\
-1.5, \\
-1.5, \\
0.33, & \ldots
\end{bmatrix}$$

loss 1.25347

gradient $dW$: 

$$\begin{bmatrix}
-2.5, \\
0.6, \\
0, \\
0.2, \\
0.7, \\
-0.5, \\
1.1, \\
1.3, \\
-2.1, & \ldots
\end{bmatrix}$$

dW = ... 
(some function data and $W$)
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

```python
# Vanilla Gradient Descent
while True:
    weights_grad = evaluate_gradient(loss_fun, data, weights)
    weights += - step_size * weights_grad # perform parameter update
```
negative gradient direction
Stochastic Gradient Descent (SGD)

\[ L(W) = \frac{1}{N} \sum_{i=1}^{N} L_i(x_i, y_i, W) + \lambda R(W) \]

\[ \nabla_W L(W) = \frac{1}{N} \sum_{i=1}^{N} \nabla_W L_i(x_i, y_i, W) + \lambda \nabla_W R(W) \]

Full sum expensive when N is large!

Approximate sum using a minibatch of examples
32 / 64 / 128 common

# 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
Optimization: Problem #1 with SGD

What if loss changes quickly in one direction and slowly in another? What does gradient descent do?

Aside: Loss function has high **condition number**: ratio of largest to smallest singular value of the Hessian matrix is large.
Optimization: Problem #1 with SGD

What if loss changes quickly in one direction and slowly in another?
What does gradient descent do?
Very slow progress along shallow dimension, jitter along steep direction

Loss function has high **condition number**: ratio of largest to smallest singular value of the Hessian matrix is large
What if the loss function has a local minima or saddle point?
Optimization: Problem #2 with SGD

What if the loss function has a local minima or saddle point?

Zero gradient, gradient descent gets stuck
What if the loss function has a local minima or saddle point?

Saddle points much more common in high dimension

Dauphin et al, "Identifying and attacking the saddle point problem in high-dimensional non-convex optimization", NIPS 2014
Optimization: Problem #2 with SGD

**saddle point** in two dimension

\[ f(x, y) = x^2 - y^2 \]

\[ \frac{\partial}{\partial x} (x^2 - y^2) = 2x \rightarrow 2(0) = 0 \]

\[ \frac{\partial}{\partial y} (x^2 - y^2) = -2y \rightarrow -2(0) = 0 \]

Optimization: Problem #3 with SGD

Our gradients come from minibatches so they can be noisy!

\[ L(W) = \frac{1}{N} \sum_{i=1}^{N} L_i(x_i, y_i, W) \]

\[ \nabla_W L(W) = \frac{1}{N} \sum_{i=1}^{N} \nabla_W L_i(x_i, y_i, W) \]
SGD + Momentum

Local Minima

Gradient Noise

Saddle points

Poor Conditioning

SGD

SGD+Momentum
SGD: the simple two line update code

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

```python
while True:
    dx = compute_gradient(x)
    x -= learning_rate * dx
```
SGD + Momentum:
continue moving in the general direction as the previous iterations

\[ x_{t+1} = x_t - \alpha \nabla f(x_t) \]

- Build up “velocity” as a running mean of gradients
- Rho gives “friction”; typically rho=0.9 or 0.99

Sutskever et al, “On the importance of initialization and momentum in deep learning”, ICML 2013
**SGD + Momentum:**
continue moving in the general direction as the previous iterations

**SGD**

\[ x_{t+1} = x_t - \alpha \nabla f(x_t) \]

**SGD+Momentum**

\[ v_{t+1} = \rho v_t + \nabla f(x_t) \]
\[ x_{t+1} = x_t - \alpha v_{t+1} \]

- Build up “velocity” as a running mean of gradients
- Rho gives “friction”; typically rho=0.9 or 0.99

Sutskever et al, “On the importance of initialization and momentum in deep learning”, ICML 2013
SGD + Momentum: alternative equivalent formulation

You may see SGD+Momentum formulated different ways, but they are equivalent - give same sequence of $x$

Sutskever et al, “On the importance of initialization and momentum in deep learning”, ICML 2013
SGD+Momentum

Momentum update:

Combine gradient at current point with velocity to get step used to update weights

Nesterov, “A method of solving a convex programming problem with convergence rate $O(1/k^2)$”, 1983
Nesterov, “Introductory lectures on convex optimization: a basic course”, 2004
Sutskever et al, “On the importance of initialization and momentum in deep learning”, ICML 2013
Nesterov Momentum

Momentum update:

Combine gradient at current point with velocity to get step used to update weights

“Look ahead” to the point where updating using velocity would take us; compute gradient there and mix it with velocity to get actual update direction

Nesterov, “A method of solving a convex programming problem with convergence rate $O(1/k^2)$”, 1983
Nesterov, “Introductory lectures on convex optimization: a basic course”, 2004
Sutskever et al, “On the importance of initialization and momentum in deep learning”, ICML 2013
Nesterov Momentum

\[ v_{t+1} = \rho v_t - \alpha \nabla f(x_t + \rho v_t) \]
\[ x_{t+1} = x_t + v_{t+1} \]

“Look ahead” to the point where updating using velocity would take us; compute gradient there and mix it with velocity to get actual update direction.
Nesterov Momentum

\[ v_{t+1} = \rho v_t - \alpha \nabla f(x_t + \rho v_t) \]
\[ x_{t+1} = x_t + v_{t+1} \]

Annoying, usually we want update in terms of \( x_t, \nabla f(x_t) \).

“Look ahead” to the point where updating using velocity would take us; compute gradient there and mix it with velocity to get actual update direction.
Nesterov Momentum

\[ v_{t+1} = \rho v_t - \alpha \nabla f(x_t + \rho v_t) \]
\[ x_{t+1} = x_t + v_{t+1} \]

Annoying, usually we want update in terms of \( x_t, \nabla f(x_t) \).

Change of variables \( \tilde{x}_t = x_t + \rho v_t \) and rearrange:

“Look ahead” to the point where updating using velocity would take us; compute gradient there and mix it with velocity to get actual update direction.
Nesterov Momentum

\[ v_{t+1} = \rho v_t - \alpha \nabla f(x_t + \rho v_t) \]
\[ x_{t+1} = x_t + v_{t+1} \]

Annoying, usually we want update in terms of \( x_t, \nabla f(x_t) \)

Change of variables \( \tilde{x}_t = x_t + \rho v_t \) and rearrange:

\[ v_{t+1} = \rho v_t - \alpha \nabla f(\tilde{x}_t) \]
\[ \tilde{x}_{t+1} = \tilde{x}_t - \rho v_t + (1 + \rho)v_{t+1} \]
\[ = \tilde{x}_t + v_{t+1} + \rho(v_{t+1} - v_t) \]

“Look ahead” to the point where updating using velocity would take us; compute gradient there and mix it with velocity to get actual update direction

Nesterov Momentum

- SGD
- SGD+Momentum
- Nesterov
AdaGrad

\begin{verbatim}
grad_squared = 0
while True:
    dx = compute_gradient(x)
    grad_squared += dx * dx
    x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
\end{verbatim}

Added element-wise scaling of the gradient based on the historical sum of squares in each dimension

“Per-parameter learning rates”
or “adaptive learning rates”

Duchi et al, "Adaptive subgradient methods for online learning and stochastic optimization", JMLR 2011
AdaGrad

```
grad_squared = 0
while True:
    dx = compute_gradient(x)
    grad_squared += dx * dx
    x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

Q: What happens with AdaGrad?
AdaGrad

```python
grad_squared = 0
while True:
    dx = compute_gradient(x)
    grad_squared += dx * dx
    x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

Q: What happens with AdaGrad? Progress along “steep” directions is damped; progress along “flat” directions is accelerated.
AdaGrad

```python
grad_squared = 0
while True:
dx = compute_gradient(x)
grad_squared += dx * dx
x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

Q2: What happens to the step size over long time?
AdaGrad

```
grad_squared = 0
while True:
    dx = compute_gradient(x)
    grad_squared += dx * dx
    x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

Q2: What happens to the step size over long time? Decays to zero
RMSProp: “Leaky AdaGrad”

AdaGrad

```python
grad_squared = 0
while True:
    dx = compute_gradient(x)
    grad_squared += dx * dx
    x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

RMSProp

```python
grad_squared = 0
while True:
    dx = compute_gradient(x)
    grad_squared = decay_rate * grad_squared + (1 - decay_rate) * dx * dx
    x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

Tieleman and Hinton, 2012
RMSProp

- SGD
- SGD+Momentum
- RMSProp
- AdaGrad
  (stuck due to decaying lr)
Adam (almost)

```python
first_moment = 0
second_moment = 0
while True:
    dx = compute_gradient(x)
    first_moment = beta1 * first_moment + (1 - beta1) * dx
    second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
    x -= learning_rate * first_moment / (np.sqrt(second_moment) + 1e-7))
```

Adam (almost)

```
first_moment = 0
second_moment = 0
while True:
    dx = compute_gradient(x)
    first_moment = beta1 * first_moment + (1 - beta1) * dx
    second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
    x -= learning_rate * first_moment / (np.sqrt(second_moment) + 1e-7))
```

**Momentum**

**AdaGrad / RMSProp**

Sort of like RMSProp with momentum

**Q: What happens at first timestep?**

Adam (full form)

```python
firstMoment = 0
secondMoment = 0
for t in range(1, num_iterations):
    dx = compute_gradient(x)
    firstMoment = beta1 * firstMoment + (1 - beta1) * dx
    secondMoment = beta2 * secondMoment + (1 - beta2) * dx * dx
    firstUnbias = firstMoment / (1 - beta1 ** t)
    secondUnbias = secondMoment / (1 - beta2 ** t)
    x -= learning_rate * firstUnbias / (np.sqrt(secondUnbias) + 1e-7)
```

Bias correction for the fact that first and second moment estimates start at zero

Adam (full form)

```python
first_moment = 0
second_moment = 0
for t in range(1, num_iterations):
    dx = compute_gradient(x)
    first_moment = beta1 * first_moment + (1 - beta1) * dx
    second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
    first_unbias = first_moment / (1 - beta1 ** t)
    second_unbias = second_moment / (1 - beta2 ** t)
    x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7)
```

Momentum

Bias correction

AdaGrad / RMSProp

Bias correction for the fact that first and second moment estimates start at zero

Adam with beta1 = 0.9, beta2 = 0.999, and learning_rate = 1e-3 or 5e-4 is a great starting point for many models!

Adam

- SGD
- SGD+Momentum
- RMSProp
- Adam
Learning rate schedules

```python
# Vanilla Gradient Descent

while True:
    weights_grad = evaluate_gradient(loss_fun, data, weights)
    weights += -step_size * weights_grad  # perform parameter update
```
SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyperparameter.

Q: Which one of these learning rates is best to use?
SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyperparameter.

Q: Which one of these learning rates is best to use?

A: In reality, all of these are good learning rates.
Learning rate decays over time

**Step:** Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.
Learning Rate Decay

**Step:** Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

**Cosine:**

$$\alpha_t = \frac{1}{2} \alpha_0 \left(1 + \cos \left(\frac{t\pi}{T}\right)\right)$$

- $\alpha_0$: Initial learning rate
- $\alpha_t$: Learning rate at epoch $t$
- $T$: Total number of epochs
Learning Rate Decay

**Step:** Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

**Cosine:**

\[ \alpha_t = \frac{1}{2} \alpha_0 \left( 1 + \cos \left( \frac{t \pi}{T} \right) \right) \]

- \( \alpha_0 \): Initial learning rate
- \( \alpha_t \): Learning rate at epoch \( t \)
- \( T \): Total number of epochs

---

Loshchilov and Hutter, “SGDR: Stochastic Gradient Descent with Warm Restarts”, ICLR 2017
Radford et al, “Improving Language Understanding by Generative Pre-Training”, 2018
Feichtenhofer et al, “SlowFast Networks for Video Recognition”, arXiv 2018
Learning Rate Decay

**Step:** Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

**Cosine:** \[ \alpha_t = \frac{1}{2} \alpha_0 \left(1 + \cos\left(\frac{t\pi}{T}\right)\right) \]

**Linear:** \[ \alpha_t = \alpha_0 \left(1 - \frac{t}{T}\right) \]

- \(\alpha_0\): Initial learning rate
- \(\alpha_t\): Learning rate at epoch \(t\)
- \(T\): Total number of epochs

Devlin et al, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding", 2018
Learning Rate Decay

**Step:** Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

**Cosine:** \[ \alpha_t = \frac{1}{2} \alpha_0 \left(1 + \cos\left(\frac{t \pi}{T}\right)\right) \]

**Linear:** \[ \alpha_t = \alpha_0 (1 - t/T) \]

**Inverse sqrt:** \[ \alpha_t = \frac{\alpha_0}{\sqrt{t}} \]

\( \alpha_0 \) : Initial learning rate  
\( \alpha_t \) : Learning rate at epoch t  
\( T \) : Total number of epochs

Vaswani et al, “Attention is all you need”, NIPS 2017
Learning Rate Decay: Linear Warmup

High initial learning rates can make loss explode; linearly increasing learning rate from 0 over the first ~5,000 iterations can prevent this.

Empirical rule of thumb: If you increase the batch size by N, also scale the initial learning rate by N

Goyal et al, “Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour”, arXiv 2017
First-Order Optimization

Loss

w1
First-Order Optimization

1. Use gradient form linear approximation
2. Step to minimize the approximation
Second-Order Optimization

(1) Use gradient **and Hessian** to form **quadratic** approximation
(2) Step to the **minima** of the approximation
Second-Order Optimization

second-order Taylor expansion:

\[ J(\theta) \approx J(\theta_0) + (\theta - \theta_0) \top \nabla_\theta J(\theta_0) + \frac{1}{2} (\theta - \theta_0) \top H(\theta - \theta_0) \]

Solving for the critical point we obtain the Newton parameter update:

\[ \theta^* = \theta_0 - H^{-1} \nabla_\theta J(\theta_0) \]

Q: Why is this bad for deep learning?
Second-Order Optimization

second-order Taylor expansion:

$$J(\theta) \approx J(\theta_0) + (\theta - \theta_0)^\top \nabla_\theta J(\theta_0) + \frac{1}{2} (\theta - \theta_0)^\top H(\theta - \theta_0)$$

Solving for the critical point we obtain the Newton parameter update:

$$\theta^* = \theta_0 - H^{-1} \nabla_\theta J(\theta_0)$$

Hessian has $O(N^2)$ elements
Inverting takes $O(N^3)$
$N = \text{(Tens or Hundreds of) Millions}$

Q: Why is this bad for deep learning?
Second-Order Optimization

\[ \theta^* = \theta_0 - H^{-1} \nabla_{\theta} J(\theta_0) \]

- Quasi-Newton methods (**BGFS** most popular): *instead of inverting the Hessian (O(n^3)), approximate inverse Hessian with rank 1 updates over time (O(n^2) each).*

- **L-BFGS** (Limited memory BFGS): *Does not form/store the full inverse Hessian.*
L-BFGS

- Usually works very well in full batch, deterministic mode i.e. if you have a single, deterministic $f(x)$ then L-BFGS will probably work very nicely.

- Does not transfer very well to mini-batch setting. Gives bad results. Adapting second-order methods to large-scale, stochastic setting is an active area of research.

Ba et al, “Distributed second-order optimization using Kronecker-factored approximations”, ICLR 2017
In practice:

- **Adam** is a good default choice in many cases; it often works ok even with constant learning rate
- **SGD+Momentum** can outperform Adam but may require more tuning of LR and schedule
- If you can afford to do full batch updates then try out **L-BFGS** (and don’t forget to disable all sources of noise)
Next time:

Introduction to neural networks

Backpropagation