Lecture 4: Regularization + Optimization
Reminder: Assignment 1

Was due on Friday!
Assignment 2

• Released last night
• Use SGD to train linear classifiers and fully-connected networks
• After today, can do linear classifiers section
• After Wednesday, can do fully-connected networks
• If you have a hard time computing derivatives, wait for next Monday’s lecture on backprop
• Due Friday September 25, 11:59pm EDT
Questions During Lecture

• Lecturing + watching chat is hard!

• I won’t actively watch chat constantly – instead I’ll go back and address questions on chat when I reach a logical pause point
Last Time: Linear Classifiers

**Algebraic Viewpoint**

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

**Visual Viewpoint**

One template per class

**Geometric Viewpoint**

Hyperplanes cutting up space
Last Time: Loss Functions quantify preferences

- We have some dataset of \((x, y)\)
- We have a **score function**:
- We have a **loss function**:

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

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

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

Linear classifier
Last Time: Loss Functions quantify preferences

- We have some dataset of \((x, y)\)
- We have a **score function**: 
- We have a **loss function**:

### Softmax

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

### SVM

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

**Problem:** Loss functions encourage good performance on training data but we really care about test data

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

Linear classifier
Overfitting

A model is **overfit** when it performs too well on the training data, and has poor performance for unseen data.
Overfitting

A model is **overfit** when it performs too well on the training data, and has poor performance for unseen data.

**Example:** Linear classifier with 1D inputs, 2 classes, softmax loss

\[ s_i = w_i x + b_i \]

\[ p_i = \frac{\exp(s_i)}{\exp(s_1) + \exp(s_2)} \]

\[ L = -\log(p_y) \]
Overfitting

A model is **overfit** when it performs too well on the training data, and has poor performance for unseen data.

Example: Linear classifier with 1D inputs, 2 classes, softmax loss

\[ s_i = w_i x + b_i \]
\[ p_i = \frac{\exp(s_i)}{\exp(s_1) + \exp(s_2)} \]
\[ L = -\log(p_y) \]

Both models have perfect accuracy on train data!
Overfitting

A model is **overfit** when it performs too well on the training data, and has poor performance for unseen data.

Example: Linear classifier with 1D inputs, 2 classes, softmax loss

\[
s_i = w_i x + b_i
\]

\[
p_i = \frac{\exp(s_i)}{\exp(s_1) + \exp(s_2)}
\]

\[
L = -\log(p_y)
\]

Both models have perfect accuracy on train data!

Low loss, but unnatural “cliff” between training points
Regularization: Beyond Training Error

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

**Data loss**: Model predictions should match training data.
Regularization: Beyond Training Error

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

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

\( \lambda \) is a hyperparameter giving regularization strength
Regularization: Beyond Training Error

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

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

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

**Simple examples**

- \( \text{L2 regularization: } R(W) = \sum_{k,l} W_{k,l}^2 \)
- \( \text{L1 regularization: } R(W) = \sum_{k,l} |W_{k,l}| \)

\( \lambda \) is a hyperparameter giving regularization strength
Regularization: Beyond Training Error

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

\[ \lambda \text{ is a hyperparameter giving regularization strength} \]

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

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

**Simple examples**

- **L2 regularization**: \( R(W) = \sum_{k,l} W_{k,l}^2 \)
- **L1 regularization**: \( R(W) = \sum_{k,l} |W_{k,l}| \)

**More complex**: Dropout, Batch normalization, Cutout, Mixup, Stochastic depth, etc...
Regularization: Prefer Simpler Models

Example: Linear classifier with 1D inputs, 2 classes, softmax loss

\[ s_i = w_i x + b_i \quad p_i = \frac{\exp(s_i)}{\exp(s_1) + \exp(s_2)} \]

\[ L = -\log(p_y) + \lambda \sum_i w_i^2 \]
Regularization: Prefer Simpler Models

Example: Linear classifier with 1D inputs, 2 classes, softmax loss

\[ s_i = w_i x + b_i \quad p_i = \frac{\exp(s_i)}{\exp(s_1) + \exp(s_2)} \]

\[ L = -\log(p_y) + \lambda \sum_i w_i^2 \]

Regularization term causes loss to increase for model with sharp cliff
Regularization: Expressing Preferences

L2 Regularization

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

\[ x = [1, 1, 1, 1] \]

\[ w_1 = [1, 0, 0, 0] \]

\[ w_2 = [0.25, 0.25, 0.25, 0.25] \]

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

Same predictions, so data loss will always be the same
Regularization: Expressing Preferences

L2 Regularization

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

L2 regularization prefers weights to be “spread out”

\[ x = [1, 1, 1, 1] \]

\[ w_1 = [1, 0, 0, 0] \]

\[ w_2 = [0.25, 0.25, 0.25, 0.25] \]

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

Same predictions, so data loss will always be the same
Finding a good $W$

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

**Loss function** consists of **data loss** to fit the training data and **regularization** to prevent overfitting.
$w^* = \arg \min_w L(w)$
Idea #1: Random Search (bad idea!)

```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)
```
Idea #1: Random Search (bad idea!)

# 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!
Idea #1: Random Search (bad idea!)

# 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%)
Idea #2: Follow the slope
Idea #2: Follow the slope

In 1-dimension, the derivative of a function gives the slope:

$$\frac{df}{dx} = \lim_{h \to 0} \frac{f(x + h) - f(x)}{h}$$
Idea #2: Follow the slope

In 1-dimension, the **derivative** of a function gives the slope:

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

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**.
<table>
<thead>
<tr>
<th>current $W$:</th>
<th>gradient $dL/dW$:</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>[?, ?, ?, ?, ?, ?, ?, ?, ?,...,...]</td>
</tr>
</tbody>
</table>

**Loss 1.25347**
<table>
<thead>
<tr>
<th>current W:</th>
<th>W + h (first dim):</th>
<th>gradient $dL/dW$:</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0.34,</td>
<td>[0.34 + 0.0001,</td>
<td>[?,</td>
</tr>
<tr>
<td>-1.11,</td>
<td>-1.11,</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>
</tbody>
</table>

**loss 1.25347**  
**loss 1.25322**
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 $dL/dW$:  

\[
\begin{bmatrix}
-2.5, \\
?, \\
?, \\
?, \\
?, \\
?, \\
?, \\
?, \\
?,
\end{bmatrix}
\]

\[
\frac{(1.25322 - 1.25347)}{0.0001} = -2.5
\]
<table>
<thead>
<tr>
<th>current W:</th>
<th>( W + h ) (second dim):</th>
<th>gradient ( \frac{dL}{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>loss 1.25347</td>
<td>loss 1.25353</td>
<td></td>
</tr>
</tbody>
</table>
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 (second dim):  

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

loss 1.25353

gradient dL/dW:  

\[
\begin{bmatrix}
-2.5, \\
0.6, \\
?, \\
?, \\
?,
\end{bmatrix}
\]

\[
\frac{(1.25353 - 1.25347)/0.0001}{0.0001} = 0.6
\]

\[
\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x + h) - f(x)}{h}
\]
<table>
<thead>
<tr>
<th>current W:</th>
<th>W + h (third dim):</th>
<th>gradient dL/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>
</tbody>
</table>

loss 1.25347

loss 1.25347
<table>
<thead>
<tr>
<th>current W:</th>
<th>$W + h$ (third dim):</th>
<th>gradient $\frac{dL}{dW}$:</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>$[-2.5, 0.6, 0.0, ?, ?, 0.0001]$</td>
</tr>
</tbody>
</table>

<p>| loss 1.25347 | loss 1.25347 | $(1.25347 - 1.25347)/0.0001 = 0.0$ |</p>
<table>
<thead>
<tr>
<th>current W:</th>
<th>$W + h$ (third dim):</th>
<th>gradient $dL/dW$:</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, 0.0, ?, ?, ?, ?, ...]</td>
</tr>
<tr>
<td>loss 1.25347</td>
<td>loss 1.25347</td>
<td>Numeric Gradient:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Slow: $O$(#dimensions)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Approximate</td>
</tr>
</tbody>
</table>
Loss is a function of $W$

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

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

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

Want $\nabla_w L$
Loss is a function of $W$: Analytic Gradient

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

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

$s = f(x, W) = WX$

Want $\nabla_w L$

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

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

loss 1.25347

gradient $dL/dW$: 

\[
\begin{array}{c}
-2.5, \\
0.6, \\
0, \\
0.2, \\
0.7, \\
-0.5, \\
1.1, \\
1.3, \\
-2.1, ...
\end{array}
\]

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

gradient dL/dW:

\[ \begin{bmatrix}
-2.5, \\
0.6, \\
0, \\
0.2, \\
0.7, \\
-0.5, \\
1.1, \\
1.3, \\
-2.1, \\
\end{bmatrix} \]

dL/dW = ... (some function data and W)

(In practice we will compute dL/dW using backpropagation; see Lecture 6)
Computing Gradients

- **Numeric 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.
Computing Gradients

- **Numeric 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**.

```python
def grad_check_sparse(f, x, analytic_grad, num_checks=10, h=1e-7):
    """sample a few random elements and only return numerical in this dimensions."""
```
Computing Gradients

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

```python
torch.autograd.gradcheck(func, inputs, eps=1e-06, atol=1e-05, rtol=0.001, raise_exception=True, check_sparse_nnz=False, nondet_tol=0.0)
```

Check gradients computed via small finite differences against analytical gradients w.r.t. tensors in `inputs` that are of floating point type and with `requires_grad=True`.

The check between numerical and analytical gradients uses `allclose()`.
Computing Gradients

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

```python
torch.autograd.gradgradcheck(func, inputs, grad_outputs=None, eps=1e-06, atol=1e-05, rtol=0.001, gen_non_contig_grad_outputs=False, raise_exception=True, nondet_tol=0.0)
```

Check gradients of gradients computed via small finite differences against analytical gradients w.r.t. tensors in `inputs` and `grad_outputs` that are of floating point type and with `requires_grad=True`.

This function checks that backpropagating through the gradients computed to the given `grad_outputs` are correct.
Gradient Descent

Iteratively step in the direction of the negative gradient (direction of local steepest descent)

```
# Vanilla gradient descent
w = initialize_weights()
for t in range(num_steps):
    dw = compute_gradient(loss_fn, data, w)
    w -= learning_rate * dw
```

**Hyperparameters:**
- Weight initialization method
- Number of steps
- Learning rate
Gradient Descent

Iteratively step in the direction of the negative gradient (direction of local steepest descent)

```python
# Vanilla gradient descent
w = initialize_weights()
for t in range(num_steps):
    dw = compute_gradient(loss_fn, data, w)
    w -= learning_rate * dw
```

**Hyperparameters:**
- Weight initialization method
- Number of steps
- Learning rate
Gradient Descent

Iteratively step in the direction of the negative gradient (direction of local steepest descent)

```
# Vanilla gradient descent
w = initialize_weights()
for t in range(num_steps):
    dw = compute_gradient(loss_fn, data, w)
    w -= learning_rate * dw
```

**Hyperparameters:**
- Weight initialization method
- Number of steps
- Learning rate
Batch Gradient Descent

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

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

Full sum expensive when N is large!
Stochastic Gradient Descent (SGD)

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

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

# Stochastic gradient descent

```python
w = initialize_weights()
for t in range(num_steps):
    minibatch = sample_data(data, batch_size)
    dw = compute_gradient(loss_fn, minibatch, w)
    w -= learning_rate * dw
```

Full sum expensive when N is large!

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

**Hyperparameters:**
- Weight initialization
- Number of steps
- Learning rate
- Batch size
- Data sampling
Stochastic Gradient Descent (SGD)

\[ L(W) = \mathbb{E}_{(x,y) \sim p_{data}}[L(x, y, W)] + \lambda R(W) \]

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

Think of loss as an expectation over the full data distribution \( p_{data} \)

Approximate expectation via sampling
Stochastic Gradient Descent (SGD)

\[ L(W) = \mathbb{E}_{(x,y) \sim p_{\text{data}}} [L(x, y, W)] + \lambda R(W) \approx \frac{1}{N} \sum_{i=1}^{N} L(x_i, y_i, W) + \lambda R(W) \]

Think of loss as an expectation over the full data distribution \( p_{\text{data}} \)

Approximate expectation via sampling

\[ \nabla_W L(W) = \nabla_W \mathbb{E}_{(x,y) \sim p_{\text{data}}} [L(x, y, W)] + \lambda \nabla_W R(W) \approx \sum_{i=1}^{N} \nabla_W L_W(x_i, y_i, W) + \nabla_W R(W) \]
Interactive Web Demo

http://vision.stanford.edu/teaching/cs231n-demos/linear-classify/
Problems with SGD

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

Loss function has high **condition number**: ratio of largest to smallest singular value of the Hessian matrix is large
Problems 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
Problems with SGD

What if the loss function has a **local minimum** or **saddle point**?
Problems with SGD

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

Zero gradient, gradient descent gets stuck
Problems with SGD

Our gradients come from minibatches so they can be noisy!

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

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

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

```python
for t in range(num_steps):
    dw = compute_gradient(w)
    w *= learning_rate * dw
```
SGD + Momentum

SGD

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

```python
for t in range(num_steps):
dw = compute_gradient(w)
w = learning_rate * dw
```

SGD+Momentum

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

```python
v = 0
for t in range(num_steps):
dw = compute_gradient(w)
v = rho * v + dw
w = learning_rate * v
```

- 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

Momentum update:

- Combine gradient at current point with velocity to get step used to update weights
- Build up “velocity” as a running mean of gradients
- Rho gives “friction”; typically rho=0.9 or 0.99

\[
\begin{align*}
\nu_{t+1} &= \rho \nu_t + \nabla f(x_t) \\
x_{t+1} &= x_t - \alpha \nu_{t+1}
\end{align*}
\]

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

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

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

Local Minima  Saddle points

Gradient Noise

Poor Conditioning

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:

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

```python
v = 0
for t in range(num_steps):
    dw = compute_gradient(w)
    old_v = v
    v = rho * v - learning_rate * dw
    w = rho * old_v - (1 + rho) * v
```
Nesterov Momentum

- SGD
- SGD+Momentum
- Nesterov
AdaGrad

```python
grad_squared = 0
for t in range(num_steps):
    dw = compute_gradient(w)
    grad_squared += dw * dw
w -= learning_rate * dw / (grad_squared.sqrt() + 1e-7)
```

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
for t in range(num_steps):
    dw = compute_gradient(w)
    grad_squared += dw * dw
w -= learning_rate * dw / (grad_squared.sqrt() + 1e-7)

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

```
grad_squared = 0
for t in range(num_steps):
    dw = compute_gradient(w)
    grad_squared += dw * dw
    w -= learning_rate * dw / (grad_squared.sqrt() + 1e-7)
```

Q: What happens with AdaGrad?
AdaGrad

```python
grad_squared = 0
for t in range(num_steps):
    dw = compute_gradient(w)
    grad_squared += dw * dw
w -= learning_rate * dw / (grad_squared.sqrt() + 1e-7)
```

Q: What happens with AdaGrad?

Progress along “steep” directions is damped; progress along “flat” directions is accelerated
RMSProp: “Leaky Adagrad”

```
grad_squared = 0
for t in range(num_steps):
    dw = compute_gradient(w)
    grad_squared += dw * dw
w -= learning_rate * dw / (grad_squared.sqrt() + 1e-7)
```

Tieleman and Hinton, 2012
RMSProp

- SGD
- SGD+Momentum
- RMSProp
Adam (almost): RMSProp + Momentum

```python
moment1 = 0
moment2 = 0
for t in range(num_steps):
    dw = compute_gradient(w)
    moment1 = beta1 * moment1 + (1 - beta1) * dw
    moment2 = beta2 * moment2 + (1 - beta2) * dw * dw
    w -= learning_rate * moment1 / (moment2.sqrt() + 1e-7)
```

Adam (almost): RMSProp + Momentum

```
moment1 = 0
moment2 = 0
for t in range(num_steps):
    dw = compute_gradient(w)
    moment1 = beta1 * moment1 + (1 - beta1) * dw
    moment2 = beta2 * moment2 + (1 - beta2) * dw * dw
    w -= learning_rate * moment1 / (moment2.sqrt() + 1e-7)
```

Adam (almost): RMSProp + Momentum

```python
moment1 = 0
moment2 = 0
for t in range(num_steps):
    dw = compute_gradient(w)
    moment1 = beta1 * moment1 + (1 - beta1) * dw
    moment2 = beta2 * moment2 + (1 - beta2) * dw * dw
    w -= learning_rate * moment1 / (moment2.sqrt() + 1e-7)
```


Adam

Momentum

AdaGrad / RMSProp

RMSProp

```python
grad_squared = 0
for t in range(num_steps):
    dw = compute_gradient(w)
    grad_squared = decay_rate * grad_squared + (1 - decay_rate) * dw * dw
    w -= learning_rate * dw / (grad_squared.sqrt() + 1e-7)
```
Adam (almost): RMSProp + Momentum

\[
\begin{align*}
\text{moment1} &= 0 \\
\text{moment2} &= 0 \\
\text{for } t \text{ in range(num_steps)}: \\
& \quad \text{dw} = \text{compute\_gradient}(w) \\
& \quad \text{moment1} = \text{beta1} \times \text{moment1} + (1 - \text{beta1}) \times \text{dw} \\
& \quad \text{moment2} = \text{beta2} \times \text{moment2} + (1 - \text{beta2}) \times \text{dw} \times \text{dw} \\
& \quad w \leftarrow \text{learning\_rate} \times \text{moment1} / (\text{moment2}\_\text{sqrt}() + 1e^{-7})
\end{align*}
\]

Q: What happens at t=0? (Assume beta2 = 0.999)

Adam (almost): RMSProp + Momentum

```
moment1 = 0
moment2 = 0
for t in range(num_steps):
    dw = compute_gradient(w)
    moment1 = beta1 * moment1 + (1 - beta1) * dw
    moment2 = beta2 * moment2 + (1 - beta2) * dw * dw
    moment1_unbias = moment1 / (1 - beta1 ** t)
    moment2_unbias = moment2 / (1 - beta2 ** t)
    w -= learning_rate * moment1_unbias / (moment2_unbias.sqrt() + 1e-7)
```

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

Adam (almost): RMSProp + Momentum

```python
moment1 = 0
moment2 = 0
for t in range(num_steps):
    dw = compute_gradient(w)
    moment1 = beta1 * moment1 + (1 - beta1) * dw
    moment2 = beta2 * moment2 + (1 - beta2) * dw * dw
    moment1_unbias = moment1 / (1 - beta1 ** t)
    moment2_unbias = moment2 / (1 - beta2 ** t)
    w -= learning_rate * moment1_unbias / (moment2_unbias.sqrt() + 1e-7)
```

**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, 5e-4, 1e-4 is a great starting point for many models!

Adam: Very Common in Practice!

for input to the CNN; each colored pixel in the image yields a 7D one-hot vector. Following common practice, the network is trained end-to-end using stochastic gradient descent with the Adam optimizer [22]. We anneal the learning rate to 0 using a half cosine schedule without restarts [28].

Bakhtin, van der Maaten, Johnson, Gustafson, and Girshick, NeurIPS 2019

We train all models using Adam [23] with learning rate $10^{-4}$ and batch size 32 for 1 million iterations; training takes about 3 days on a single Tesla P100. For each minibatch we first update $f$, then update $D_{img}$ and $D_{obj}$.

Johnson, Gupta, and Fei-Fei, CVPR 2018

organized into three residual blocks. We train for 25 epochs using Adam [27] with learning rate $10^{-4}$ and 32 images per batch on 8 Tesla V100 GPUs. We set the cubify thresh-

Gkioxari, Malik, and Johnson, ICCV 2019

samed with each bit drawn uniformly at random. For gradient descent, we use Adam [29] with a learning rate of $10^{-3}$ and default hyperparameters. All models are trained with batch size 12. Models are trained for 200 epochs, or 400 epochs if being trained on multiple noise layers.

Zhu, Kaplan, Johnson, and Fei-Fei, ECCV 2018

16 dimensional vectors. We iteratively train the Generator and Discriminator with a batch size of 64 for 200 epochs using Adam [22] with an initial learning rate of 0.001.

Gupta, Johnson, et al, CVPR 2018

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

Justin Johnson

Lecture 4 - 82

September 14, 2020
Adam

- SGD
- SGD+Momentum
- RMSProp
- Adam
## Optimization Algorithm Comparison

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Tracks first moments (Momentum)</th>
<th>Tracks second moments (Adaptive learning rates)</th>
<th>Leaky second moments</th>
<th>Bias correction for moment estimates</th>
</tr>
</thead>
<tbody>
<tr>
<td>SGD</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>SGD+Momentum</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Nesterov</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>AdaGrad</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>RMSProp</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Adam</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
L2 Regularization vs Weight Decay

**Optimization Algorithm**

\[ L(w) = L_{data}(w) + L_{reg}(w) \]

\[ g_t = \nabla L(w_t) \]

\[ s_t = \text{optimizer}(g_t) \]

\[ w_{t+1} = w_t - \alpha s_t \]
L2 Regularization vs Weight Decay

**Optimization Algorithm**

\[
L(w) = L_{data}(w) + L_{reg}(w)
\]
\[
g_t = \nabla L(w_t)
\]
\[
s_t = \text{optimizer}(g_t)
\]
\[
w_{t+1} = w_t - \alpha s_t
\]

**L2 Regularization**

\[
L(w) = L_{data}(w) + \lambda |w|^2
\]
\[
g_t = \nabla L(w_t) = \nabla L_{data}(w_t) + 2\lambda w_t
\]
\[
s_t = \text{optimizer}(g_t)
\]
\[
w_{t+1} = w_t - \alpha s_t
\]

Loshchilov and Hutter, “Decoupled Weight Decay Regularization”, ICLR 2019
L2 Regularization vs Weight Decay

**Optimization Algorithm**
\[ L(w) = L_{data}(w) + L_{reg}(w) \]
\[ g_t = \nabla L(w_t) \]
\[ s_t = \text{optimizer}(g_t) \]
\[ w_{t+1} = w_t - \alpha s_t \]

**L2 Regularization**
\[ L(w) = L_{data}(w) + \lambda|w|^2 \]
\[ g_t = \nabla L_{data}(w_t) = \nabla L_{data}(w_t) + 2\lambda w_t \]
\[ s_t = \text{optimizer}(g_t) \]
\[ w_{t+1} = w_t - \alpha s_t \]

**Weight Decay**
\[ L(w) = L_{data}(w) \]
\[ g_t = \nabla L_{data}(w_t) \]
\[ s_t = \text{optimizer}(g_t) + 2\lambda w_t \]
\[ w_{t+1} = w_t - \alpha s_t \]

Loshchilov and Hutter, “Decoupled Weight Decay Regularization”, ICLR 2019
L2 Regularization vs Weight Decay

**Optimization Algorithm**
\[
\begin{align*}
L(w) &= L_{data}(w) + L_{reg}(w) \\
g_t &= \nabla L(w_t) \\
s_t &= \text{optimizer}(g_t) \\
w_{t+1} &= w_t - \alpha s_t
\end{align*}
\]

**L2 Regularization**
\[
\begin{align*}
L(w) &= L_{data}(w) + \lambda |w|^2 \\
g_t &= \nabla L(w_t) = \nabla L_{data}(w_t) + 2\lambda w_t \\
s_t &= \text{optimizer}(g_t) \\
w_{t+1} &= w_t - \alpha s_t
\end{align*}
\]

L2 Regularization and Weight Decay are equivalent for SGD, SGD+Momentum so people often use the terms interchangeably!

**Weight Decay**
\[
\begin{align*}
L(w) &= L_{data}(w) \\
g_t &= \nabla L_{data}(w_t) \\
s_t &= \text{optimizer}(g_t) + 2\lambda w_t \\
w_{t+1} &= w_t - \alpha s_t
\end{align*}
\]

Loshchilov and Hutter, “Decoupled Weight Decay Regularization”, ICLR 2019
L2 Regularization vs Weight Decay

**Optimization Algorithm**

\[ L(w) = L_{data}(w) + L_{\text{reg}}(w) \]
\[ g_t = \nabla L(w_t) \]
\[ s_t = \text{optimizer}(g_t) \]
\[ w_{t+1} = w_t - \alpha s_t \]

**L2 Regularization**

\[ L(w) = L_{data}(w) + \lambda |w|^2 \]
\[ g_t = \nabla L(w_t) = \nabla L_{data}(w_t) + 2\lambda w_t \]
\[ s_t = \text{optimizer}(g_t) \]
\[ w_{t+1} = w_t - \alpha s_t \]

L2 Regularization and Weight Decay are equivalent for SGD, SGD+Momentum so people often use the terms interchangeably.

But they are not the same for adaptive methods (AdaGrad, RMSProp, Adam, etc)

Weight Decay

\[ L(w) = L_{data}(w) \]
\[ g_t = \nabla L_{data}(w_t) \]
\[ s_t = \text{optimizer}(g_t) + 2\lambda w_t \]
\[ w_{t+1} = w_t - \alpha s_t \]

Loshchilov and Hutter, “Decoupled Weight Decay Regularization”, ICLR 2019
AdamW: Decoupled Weight Decay

Algorithm 2  Adam with L2 regularization and Adam with decoupled weight decay (AdamW)

1: given $\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}, \lambda \in \mathbb{R}$
2: initialize time step $t \leftarrow 0$, parameter vector $\theta_{t=0} \in \mathbb{R}^n$, first moment vector $m_{t=0} \leftarrow \theta$, second moment vector $v_{t=0} \leftarrow \theta$, schedule multiplier $\eta_{t=0} \in \mathbb{R}$.
3: repeat
4: $t \leftarrow t + 1$
5: $\nabla f_t(\theta_{t-1}) \leftarrow \text{SelectBatch}(\theta_{t-1})$ ▷ select batch and return the corresponding gradient
6: $g_t \leftarrow \nabla f_t(\theta_{t-1}) + \lambda \theta_{t-1}$
7: $m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t$ ▷ here and below all operations are element-wise
8: $v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2) g^2_t$
9: $\hat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ ▷ $\beta_1$ is taken to the power of $t$
10: $\hat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ ▷ $\beta_2$ is taken to the power of $t$
11: $\eta_t \leftarrow \text{SetScheduleMultiplier}(t)$ ▷ can be fixed, decay, or also be used for warm restarts
12: $\theta_t \leftarrow \theta_{t-1} - \eta_t \left( \alpha \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon) + \lambda \theta_{t-1} \right)$
13: until stopping criterion is met
14: return optimized parameters $\theta_t$

Loshchilov and Hutter, "Decoupled Weight Decay Regularization", ICLR 2019
AdamW: Decoupled Weight Decay

Algorithm 2 Adam with L2 regularization and Adam with decoupled weight decay (AdamW)

1: given $\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}, \lambda \in \mathbb{R}$
2: initialize time step $t \leftarrow 0$, parameter vector $\theta_{t=0} \in \mathbb{R}^n$, first moment vector $m_{t=0} \leftarrow \theta$, second moment vector $v_{t=0} \leftarrow \theta$, schedule multiplier $n_{t=0} \in \mathbb{R}$

11: $\eta_t \leftarrow \text{SetScheduleMultiplier}(t)$ ▷ can be fixed, decay, or also be used for warm restarts
12: $\theta_t \leftarrow \theta_{t-1} - \eta_t \left( \alpha m_t/(\sqrt{v_t} + \epsilon) + \lambda \theta_{t-1} \right)
13: \text{until stopping criterion is met}
14: \text{return} optimized parameters $\theta_t$

Loshchilov and Hutter, “Decoupled Weight Decay Regularization”, ICLR 2019
So far: First-Order Optimization
So far: **First-Order** Optimization

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

1. Use gradient and Hessian to make quadratic approximation
2. Step to minimize the approximation
Second-Order Optimization

1. Use gradient and Hessian to make quadratic approximation
2. Step to minimize the approximation

Take bigger steps in areas of low curvature
Second-Order Optimization

Second-Order Taylor Expansion:

\[ L(w) \approx L(w_0) + (w - w_0)^\top \nabla_w L(w_0) + \frac{1}{2} (w - w_0)^\top H_w L(w_0) (w - w_0) \]

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

\[ w^* = w_0 - H_w L(w_0)^{-1} \nabla_w L(w_0) \]
Second-Order Optimization

Second-Order Taylor Expansion:

\[ L(w) \approx L(w_0) + (w - w_0)^\top \nabla_w L(w_0) + \frac{1}{2} (w - w_0)^\top H_w L(w_0) (w - w_0) \]

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

\[ w^* = w_0 - H_w L(w_0)^{-1} \nabla_w L(w_0) \]

Q: Why is this impractical?
Second-Order Optimization

Second-Order Taylor Expansion:

\[ L(w) \approx L(w_0) + (w - w_0)^T \nabla_w L(w_0) + \frac{1}{2} (w - w_0)^T H_w L(w_0) (w - w_0) \]

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

\[ w^* = w_0 - H_w L(w_0)^{-1} \nabla_w L(w_0) \]

Q: Why is this impractical?  
Hessian has \( O(N^2) \) elements  
Inverting takes \( O(N^3) \)  
\( N = (\text{Tens or Hundreds of}) \text{ Millions} \)
Second-Order Optimization

\[ w^* = w_0 - H_w L(w_0)^{-1} \nabla_w L(w_0) \]

- 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.*
Second-Order Optimization: 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. **SGD+Momentum** can outperform Adam but may require more tuning.

- If you can afford to do full batch updates then try out **L-BFGS** (and don’t forget to disable all sources of noise).
Summary

1. Use **Linear Models** for image classification problems
2. Use **Loss Functions** to express preferences over different choices of weights
3. Use **Regularization** to prevent overfitting to training data
4. Use **Stochastic Gradient Descent** to minimize our loss functions and train the model

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

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

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

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

\[
\begin{align*}
v &= 0 \\
\text{for } t \text{ in range(num_steps):} \\
& \quad dw = \text{compute_gradient}(w) \\
& \quad v = \rho \times v + dw \\
& \quad w := \text{learning_rate} \times v
\end{align*}
\]
Next time:
Neural Networks