Administrative

- how is the assignment going?

- btw, the notes get updated all the time based on your feedback

- no lecture on Monday
Lecture 4: Optimization
Image Classification

assume given set of discrete labels
{dog, cat, truck, plane, ...}
Data-driven approach
1. Score function

\[ f(x_i, W, b) = W x_i + b \]
1. Score function

\[ f(x_i, W, b) = WX_i + b \]

2. Two loss functions

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

\[ L_i = \sum_{j \neq y_i} \max(0, f(x_i, W)_j - f(x_i, W)_{y_i} + \Delta) \]
Matrix multiply + bias offset

\[
\begin{bmatrix}
0.01 & -0.05 & 0.1 & 0.05 \\
0.7 & 0.2 & 0.05 & 0.16 \\
0.0 & -0.45 & -0.2 & 0.03 \\
\end{bmatrix}
\begin{bmatrix}
-15 \\
22 \\
-44 \\
56 \\
\end{bmatrix}
\begin{bmatrix}
0.0 \\
0.2 \\
-0.3 \\
\end{bmatrix}
\]

Hinge loss (SVM)

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

Cross-entropy loss (Softmax)

\[
\begin{align*}
0.058 & \quad \text{exp} \quad 0.016 \\
2.36 & \quad \text{normalize} \quad 0.631 \\
1.32 & \quad \text{to sum to one} \quad 0.353 \\
\end{align*}
\]

\[
- \log(0.353) = 0.452
\]
Three key components to training Neural Nets:

1. Score function
2. Loss function
3. Optimization
Brief aside: Image Features

- In practice, very rare to see Computer Vision applications that train linear classifiers on pixel values
Brief aside: Image Features

- In practice, very rare to see Computer Vision applications that train linear classifiers on pixel values
Example: Color (Hue) Histogram
Example: HOG features

8x8 pixel region, quantize the edge orientation into 9 bins

(images from vlfeat.org)
Example: Bag of Words

1. Resize patch to a fixed size (e.g. 32x32 pixels)
2. Extract HOG on the patch (get 144 numbers)
   repeat for each detected feature

   gives a matrix of size [number_of_features x 144]

Problem: different images will have different numbers of features. Need fixed-sized vectors for linear classification
Example: Bag of Words

1. Resize patch to a fixed size (e.g. 32x32 pixels)
2. Extract HOG on the patch (get 144 numbers)

repeat for each detected feature

gives a matrix of size
[number_of_features x 144]
Example: Bag of Words

visual word vectors

144

learn k-means centroids
“vocabulary of visual words

e.g. 1000 centroids

1000-d vector

histogram of visual words
Brief aside: Image Features
Most recognition systems are built on the same Architecture

(slide from Yann LeCun)
Most recognition systems are built on the same Architecture.

CNNs: end-to-end models

First stage: dense SIFT, HOG, GIST, sparse coding, RBM, auto-encoders.....
Second stage: K-means, sparse coding, LCC.....
Pooling: average, L2, max, max with bias (elastic templates).....
Convolutional Nets: same architecture, but everything is trained.

(slide from Yann LeCun)
Visualizing the loss function
Visualizing the (SVM) loss function

\[ L_i = \sum_{j \neq y_i} \max(0, f(x_i, W)_j - f(x_i, W)_{y_i} + \Delta) \]

\[ L_i(W + \alpha W_1) \]
Visualizing the (SVM) loss function

\[ L_i = \sum_{j \neq y_i} \max(0, f(x_i, W)_j - f(x_i, W)_{y_i} + \Delta) \]

\[ L_i(W + aW_1) \]
Visualizing the (SVM) loss function

\[ L_i = \sum_{j \neq y_i} \max(0, f(x_i, W)_j - f(x_i, W)_{y_i} + \Delta) \]

\[ L_i (W + aW_1 + bW_2) \]
Visualizing the (SVM) loss function

\[ L_i = \sum_{j \neq y_i} \max(0, f(x_i, W)_j - f(x_i, W)_{y_i} + \Delta) \]

the full data loss:

\[ L_i(W + aW_1 + bW_2) \]
Visualizing the (SVM) loss function

\[ L_i = \sum_{j \neq y_i} \max(0, w_j^T x_i - w_{y_i}^T x_i + 1) \]

Suppose there are 3 examples with 3 classes (class 0, 1, 2 in sequence), then this becomes:

\[
L_0 = \max(0, w_1^T x_0 - w_0^T x_0 + 1) + \max(0, w_2^T x_0 - w_0^T x_0 + 1) \\
L_1 = \max(0, w_0^T x_1 - w_1^T x_1 + 1) + \max(0, w_2^T x_1 - w_1^T x_1 + 1) \\
L_2 = \max(0, w_0^T x_2 - w_2^T x_2 + 1) + \max(0, w_1^T x_2 - w_2^T x_2 + 1) \\
L = (L_0 + L_1 + L_2)/3
\]
Visualizing the (SVM) loss function

\[ L_i = \sum_{j \neq y_i} \left[ \max(0, w_j^T x_i - w_{y_i}^T x_i + 1) \right] \]

Suppose there are 3 examples with 3 classes (class 0, 1, 2 in sequence), then this becomes:

\[
L_0 = \max (0, w_1^T x_0 - w_0^T x_0 + 1) + \max (0, w_2^T x_0 - w_0^T x_0 + 1) \\
L_1 = \max (0, w_0^T x_1 - w_1^T x_1 + 1) + \max (0, w_2^T x_1 - w_1^T x_1 + 1) \\
L_2 = \max (0, w_0^T x_2 - w_2^T x_2 + 1) + \max (0, w_1^T x_2 - w_2^T x_2 + 1) \\
L = (L_0 + L_1 + L_2)/3
\]
Visualizing the (SVM) loss function

\[ L_i = \sum_{j \neq y_i} \max(0, \mathbf{w}_j^T \mathbf{x}_i - \mathbf{w}_{y_i}^T \mathbf{x}_i + 1) \]

Suppose there are 3 examples with 3 classes (class 0, 1, 2 in sequence), then this becomes:

\[
\begin{align*}
L_0 &= \max \left(0, \mathbf{w}_1^T \mathbf{x}_0 - \mathbf{w}_0^T \mathbf{x}_0 + 1\right) + \max \left(0, \mathbf{w}_2^T \mathbf{x}_0 - \mathbf{w}_0^T \mathbf{x}_0 + 1\right) \\
L_1 &= \max \left(0, \mathbf{w}_0^T \mathbf{x}_1 - \mathbf{w}_1^T \mathbf{x}_1 + 1\right) + \max \left(0, \mathbf{w}_2^T \mathbf{x}_1 - \mathbf{w}_1^T \mathbf{x}_1 + 1\right) \\
L_2 &= \max \left(0, \mathbf{w}_0^T \mathbf{x}_2 - \mathbf{w}_2^T \mathbf{x}_2 + 1\right) + \max \left(0, \mathbf{w}_1^T \mathbf{x}_2 - \mathbf{w}_2^T \mathbf{x}_2 + 1\right) \\
L &= \frac{(L_0 + L_1 + L_2)}{3}
\end{align*}
\]
Visualizing the (SVM) loss function

$$L_i = \sum_{j \neq y_i} \left[ \max(0, w_j^T x_i - w_{y_i}^T x_i + 1) \right]$$

Suppose there are 3 examples with 3 classes (class 0, 1, 2 in sequence), then this becomes:

$$L_0 = \max (0, w_1^T x_0 - w_0^T x_0 + 1) + \max(0, w_2^T x_0 - w_0^T x_0 + 1)$$
$$L_1 = \max (0, w_0^T x_1 - w_1^T x_1 + 1) + \max(0, w_2^T x_1 - w_1^T x_1 + 1)$$
$$L_2 = \max (0, w_0^T x_2 - w_2^T x_2 + 1) + \max(0, w_1^T x_2 - w_2^T x_2 + 1)$$
$$L = (L_0 + L_1 + L_2) / 3$$

Question: CIFAR-10 has 50,000 training images, 5,000 per class and 10 labels. How many occurrences of one classifier row in the full data loss?
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)
```
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)
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)
```

what's up with 0.0001?
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
```
Fun aside:
When $W = 0$, what is the CIFAR-10 loss for SVM and Softmax?

\[ L = \frac{1}{N} \sum_i \sum_{j \neq y_i} \left[ \max(0, f(x_i; W)_j - f(x_i; W)_{y_i} + \Delta) \right] + \lambda \sum_k \sum_l W_{k,l}^2 \]

\[ L = \frac{1}{N} \sum_i - \log \left( \frac{e^{f_{y_i}}}{\sum_j e^{f_j}} \right) + \lambda \sum_k \sum_l W_{k,l}^2 \]
Strategy #2: A better but still very bad idea solution: Random local search

```python
W = np.random.randn(10, 3073) * 0.001  # generate random starting W
bestloss = float("inf")
for i in xrange(1000):
    step_size = 0.0001
    Wtry = W + np.random.randn(10, 3073) * step_size
    loss = L(Xtr_cols, Ytr, Wtry)
    if loss < bestloss:
        W = Wtry
        bestloss = loss
print 'iter %d loss is %f' % (i, bestloss)
```
Strategy #2: A better but still very bad idea solution: Random local search

```python
W = np.random.randn(10, 3073) * 0.001  # generate random starting W
bestloss = float("inf")
for i in xrange(1000):
    step_size = 0.0001
    Wtry = W + np.random.randn(10, 3073) * step_size
    loss = L(Xtr_cols, Ytr, Wtry)
    if loss < bestloss:
        W = Wtry
        bestloss = loss
print 'iter %d loss is %f' % (i, bestloss)
```
gives 21.4%!
Strategy #3: **Following the gradient**

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 dimension, the **gradient** is the vector of (partial derivatives).
Evaluation the gradient numerically

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

def eval_numerical_gradient(f, x):
    
    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

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

"finite difference approximation"

```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

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

in practice:

\[
\frac{[f(x + h) - f(x - h)]}{2h}
\]

“centered difference formula”

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

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

```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=['readonly'])
    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
```

# to use the generic code above we want a function that takes a single argument
# (the weights in our case) so we close over X_train and Y_train

```python
def CIFAR10_loss_fun(W):
    return L(X_train, Y_train, W)
```

W = np.random.rand(10, 3073) * 0.001 # random weight vector
def df = eval_numerical_gradient(CIFAR10_loss_fun, W) # get the gradient
performing a parameter update

```python
loss_original = CIFAR10_loss_fun(W)  # the original loss
print 'original loss: %f' % (loss_original, )

# lets see the effect of multiple step sizes
for step_size_log in [-5,-4,-3,-2,-1,0,1,2]:
    step_size = 10 ** step_size_log
    W_new = W - step_size * df  # new position in the weight space
    loss_new = CIFAR10_loss_fun(W_new)
    print 'for step size %f new loss: %f' % (step_size, loss_new)

# prints:
# original loss: 2.200718
```
performing a parameter update

```python
loss_original = CIFAR10_loss_fun(W) # the original loss
print 'original loss: %f' % (loss_original, )

# lets see the effect of multiple step sizes
for step_size_log in [-5,-4,-3,-2,-1,0,1,2]:
    step_size = 10 ** step_size_log
    W_new = W - step_size * df # new position in the weight space
    loss_new = CIFAR10_loss_fun(W_new)
    print 'for step size %f new loss: %f' % (step_size, loss_new)

# prints:
# original loss: 2.200718
# for step size 1.000000e-10 new loss: 2.200652
# for step size 1.000000e-09 new loss: 2.200057
# for step size 1.000000e-08 new loss: 2.194116
# for step size 1.000000e-07 new loss: 2.135493
# for step size 1.000000e-06 new loss: 1.647802
# for step size 1.000000e-05 new loss: 2.844355
# for step size 1.000000e-04 new loss: 25.558142
# for step size 1.000000e-03 new loss: 254.086573
# for step size 1.000000e-02 new loss: 2539.370888
# for step size 1.000000e-01 new loss: 25392.214036
```
original $W$

negative gradient direction
The problems with numerical gradient:

```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
        x[ix] += h # increment by h
        fxh = f(x) # evaluate f(x + h)
        x[ix] -= h # 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
```
The problems with numerical gradient:

- approximate
- very slow to evaluate
We need something better...

\[ L = \frac{1}{N} \sum_{i} \sum_{j \neq y_i} \left[ \max(0, f(x_i; W)_j - f(x_i; W)_{y_i} + \Delta) \right] + \lambda R(W) \]
We need something better...

\[ L = \frac{1}{N} \sum_i \sum_{j \neq y_i} \left[ \max(0, f(x_i; W)_j - f(x_i; W)_{y_i} + \Delta) \right] + \lambda R(W) \]
\[ L_i = \sum_{j \neq y_i} \left[ \max(0, w_j^T x_i - w_{y_i}^T x_i + 1) \right] \]

\[ L_i = \max \left( 0, w_1^T x_i - w_0^T x_i + 1 \right) \\
+ \max \left( 0, w_2^T x_i - w_0^T x_i + 1 \right) \\
+ \max \left( 0, w_3^T x_i - w_0^T x_i + 1 \right) \]
\[ L_i = \sum_{j \neq y_i} \left[ \max(0, w_j^T x_i - w_{y_i}^T x_i + 1) \right] \]

\[ L_i = \max (0, w_1^T x_i - w_0^T x_i + 1) + \max (0, w_2^T x_i - w_0^T x_i + 1) + \max (0, w_3^T x_i - w_0^T x_i + 1) \]

\[ \nabla_{w_j} L_i = 1(w_j^T x_i - w_{y_i}^T x_i + \Delta > 0)x_i \]
\[ L_i = \sum_{j \neq y_i} \max(0, w_j^T x_i - w_{y_i}^T x_i + 1) \]

\[ L_i = \max (0, w_1^T x_i - w_0^T x_i + 1) + \max (0, w_2^T x_i - w_0^T x_i + 1) + \max (0, w_3^T x_i - w_0^T x_i + 1) \]

\[ \nabla_{w_{y_i}} L_i = - \left( \sum_{j \neq y_i} 1(\nabla_{y_i} (w_j^T x_i - w_{y_i}^T x_i + \Delta > 0)) \right) x_i \]
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 check: Words of caution

\[ L_0 = \max \left( 0, w_1^T x_0 - w_0^T x_0 + 1 \right) + \max \left( 0, w_2^T x_0 - w_0^T x_0 + 1 \right) \]
\[ L_1 = \max \left( 0, w_0^T x_1 - w_1^T x_1 + 1 \right) + \max \left( 0, w_2^T x_1 - w_1^T x_1 + 1 \right) \]
\[ L_2 = \max \left( 0, w_0^T x_2 - w_2^T x_2 + 1 \right) + \max \left( 0, w_1^T x_2 - w_2^T x_2 + 1 \right) \]
\[ L = \frac{L_0 + L_1 + L_2}{3} \]

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

```python
# Vanilla Gradient Descent

while True:
    weights_grad = evaluate_gradient(loss_fun, data, weights)
    weights += - step_size * weights_grad  # perform parameter update
```
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 \(~100\) examples.
e.g. Krizhevsky ILSVRC ConvNet used 256 examples
Stochastic Gradient Descent (SGD)

- use a single example at a time

(Also sometimes called on-line Gradient Descent)
Summary

- Always use mini-batch gradient descent

- Incorrectly refer to it as “doing SGD” as everyone else (or call it batch gradient descent)

- The mini-batch size is a hyperparameter, but it is not very common to cross-validate over it (usually based on practical concerns, e.g. space/time efficiency)
Fun question: Suppose you were training with mini-batch size of 100, and now you switch to mini-batch of size 1. Your learning rate (step size) should:

- increase
- decrease
- stay the same
- become zero
The dynamics of Gradient Descent

\[
L = \frac{1}{N} \sum_i \sum_{j \neq y_i} \left[ \max(0, f(x_i; W)_j - f(x_i; W)_{y_i} + \Delta) \right] + \lambda \sum_k \sum_l W_{k,l}^2
\]

\[
L = \frac{1}{N} \sum_i - \log \left( \frac{e^{f_{yi}}}{\sum_j e^{f_j}} \right) + \lambda \sum_k \sum_l W_{k,l}^2
\]
The dynamics of Gradient Descent

pull some weights up and some down

\[ L = \frac{1}{N} \sum_i \sum_{j \neq y_i} \left[ \max(0, f(x_i; W)_j - f(x_i; W)_{y_i} + \Delta) \right] + \lambda \sum_k \sum_l W_{k,l}^2 \]

always pull the weights down

\[ L = \frac{1}{N} \sum_i - \log \left( \frac{e^{f_{y_i}}}{\sum_j e^{f_j}} \right) + \lambda \sum_k \sum_l W_{k,l}^2 \]
Momentum Update

\[
\text{weights\_grad} = \text{evaluate\_gradient}(\text{loss\_fun}, \text{data}, \text{weights})
\]

\[
\text{vel} = \text{vel} \times 0.9 - \text{step\_size} \times \text{weights\_grad}
\]

\[
\text{weights} += \text{vel}
\]
Many other ways to perform optimization...

- Second order methods that use the Hessian (or its approximation): BFGS, LBFGS, etc.

- Currently, the lesson from the trenches is that well-tuned SGD+Momentum is very hard to beat for CNNs.
Summary
- We looked at image features, and saw that CNNs can be thought of as learning the features in end-to-end manner
- We explored intuition about what the loss surfaces of linear classifiers look like
- We introduced gradient descent as a way of optimizing loss functions, as well as batch gradient descent and SGD.
- Numerical gradient: slow (:, approximate (:, easy to write :)
- Analytic gradient: fast (:), exact (:), error-prone :(
- In practice: Gradient check (but be careful)
Next class:

Becoming a backprop ninja