Linear Classifiers
Reading

• http://cs231n.github.io/linear-classify/
Announcements

• Last project - P4 (AlexNet)
  – Out Wednesday 5/27
  – Due Wednesday 6/3
• Optional HW3
  – mainly to help prepare you for the final
  – out tonight, due by 6/1 if you want it graded
  – no solution key will be released
  – you may collaborate freely
• Takehome final exam
  – out 6/8 (Mon), due 6/11 (Thu)
• 597P – today is the last day to opt in for P/NP
Goals

• Understand the benefits and limitations of linear classifiers over KNN.
• Understand the mathematical formulation of a binary and multiclass linear classifier.
• Know the definition and purpose of a loss function.
• Understand the intuition behind the softmax/cross-entropy loss.
• Understand how to train a classifier by minimizing a loss function using gradient descent.
• Understand the intuition behind using Stochastic (Minibatch) Gradient Descent.
Nearest Neighbor Classifier
Image classification - Multiclass classification

Which of these is it: dog, cat or zebra?

Dog
Simple Image Classification with KNN

• $\phi$: Convert to grayscale and unravel into a vector.

• $h$: Classify using majority label of the $k$ nearest neighbors according to a distance metric $d$. 
k-Nearest Neighbor on images **never used.**

- **Very slow at test time**
- Distance metrics on pixels are not informative

(Original)

Original | Boxed | Shifted | Tinted

(all 3 images have same L2 distance to the one on the left)
KNN: Bottom Line

• Fast to train but slow to predict
• Distance metrics don’t behave well for high-dimensional image vectors
Classifying Images: Let’s simplify

- Nearest Neighbor Classifier
- Linear Classifier
Linear classifiers

- Finding nearest neighbor is slow.
- Basic idea:
  - Training time: find a line that separates the data
  - Testing time: which side of the line is $\phi(x)$ on?
  - Fast to compute
  - Restrictive – data must be linearly separable
Linear classifiers

• A linear classifier corresponds to a hyperplane
  – Equivalent of a line in high-dimensional space
  – Equation: $w^T x + b = 0$

• Points on the same side are the same class
Does this ever work?

• It’s easier to be linearly separable in high-dimensional space.
• But simple linear classifiers still don’t work on most interesting data.
Some history from the Antedeepluvian Era

• Example pipeline from days of yore:
  – Detect corners and extract SIFT features
  – Collect features into a “bag of features”
  – (if you’re feeling fancy) maintain some spatial information
  – Somehow convert feature bag to fixed size
  – Apply linear classifier

• Key idea $\phi$ is designed by hand, while $h$ is learned from data.
Some history of the Antedeepluvian Era

• Key idea: $\phi$ is designed by hand, while $h$ is learned from data.

• Nowadays: learn both from data - “end-to-end”: image goes in, label comes out.
  – Enabled only recently by bigger
    • labeled datasets
    • compute power (GPUs)
Linear classifiers

- Equation: $w^T x + b = 0$
- Points on the same side are the same class
We have a classifier

- $h(x) = w^T x + b$ gives a score

- Score negative: red
- Score positive: blue

- Does it solve the runtime issues of KNN? Yes!
Multiclass Linear Classifiers:
Stack multiple $w^\top$ into a matrix.

input image

stretch pixels into single column

\[ W = \begin{bmatrix}
0.2 & -0.5 & 0.1 & 2.0 \\
1.5 & 1.3 & 2.1 & 0.0 \\
0 & 0.25 & 0.2 & -0.3 \\
\end{bmatrix} \]

\[ w^\top = \begin{bmatrix}
W_1 \\
W_2 \\
W_3 \\
\end{bmatrix} \]

\[ b = \begin{bmatrix}
56 \\
231 \\
24 \\
2 \\
1.1 \\
3.2 \\
-1.2 \\
\end{bmatrix} \]

\[ f(x_i; W, b) = \begin{bmatrix}
-96.8 \\
437.9 \\
61.95 \\
\end{bmatrix} \]

cat score
dog score
ship score
The Bias Trick

\[ \mathbf{w}^T \mathbf{x} + b \]

\[ \begin{bmatrix} \mathbf{w}^T & b \end{bmatrix} \mathbf{x}^T = \mathbf{w}^T \mathbf{x} + b \]
The Bias Trick

• Fold b into an additional dimension of w
• Add a fixed 1 to all feature vectors.

• Now, $h(x) = w^T x$
We have a classifier

- $h(x) = w^T x$ gives a score

- Score negative: red
- Score positive: blue

- Where does $w$ come from?
How do we find a good W?

• Step 1: For a given W, decide on a **Loss Function**: a measure of how much we dislike the line.

• Step 2: use **optimization** to find the W that **minimizes** the loss function.
Loss Functions

• Step 1: For a given $W$, decide on a **Loss Function**: a measure of how much we dislike this classifier.

• Step 2: use **optimization** to find the $W$ that *minimizes* the loss function.
  
  – Linear regression: solvable in *closed form* $\min \|Ax - b\|$
  
  – Useful loss functions in vision: no closed form.
Loss Functions

• Step 1: For a given $W$, decide on a **Loss Function**: a measure of how much we dislike this classifier.

• Loss Function intuition:
  – loss should be large if many data points are misclassified
  – loss should be small (0?) if all data is classified correctly.
Loss function: Ideas
Softmax Classifier / Cross-Entropy Loss: Intuition

$W^T x$ gives us a vector of scores, one per class (each row of $W$ is a classifier)

Wouldn’t it be nice to interpret these as probabilities?
Softmax Classifier / Cross-Entropy Loss: Intuition

$W^T x$ gives us a vector of scores, one per class (each row of $W$ is a classifier)

Wouldn’t it be nice to interpret these as probabilities? But they’re not...
- can be $< 0$
- don’t all sum to 1

But we can treat them as unnormalized log probabilities.
Softmax Classifier / Cross-Entropy Loss

$f = W^T x$ gives us a vector of scores, one per class (each row of $W$ is a classifier)

**Softmax normalization**: Exponentiate to get all positive values, then normalize to sum to 1:

$$p(x_i \text{ is class } k) = \frac{e^{f_k}}{\sum_j e^{f_j}}$$
Softmax Classifier / Cross-Entropy Loss

\[ f = W^T x \] gives us a vector of scores, one per class (each row of \( W \) is a classifier)

**Softmax normalization:** Exponentiate to get all positive values, then normalize to sum to 1:

\[ p(x_i \text{ is class } k) = \frac{e^{f_k}}{\sum_j e^{f_j}} \]

**Cross-entropy loss:** measure KL divergence between the predicted distribution and the true distribution:

\[ L_i = - \log \left( \frac{e^{f_{y_i}}}{\sum_j e^{f_j}} \right) \] on \( x_i \)
Cross-Entropy Loss: Intuition

$KL$ divergence

how different are these histograms?
Taking stock

- We have:
  - $\phi = \text{unravel}(\text{rgb2gray}(\text{img}))$, a feature extractor
  - $h(x) = W^T x$, a multiclass linear classifier
  - $L = \sum_{i=1}^{N} L_i$, a loss function

$$L_\theta = -\log \left( \frac{e^{f_{y_i}}}{\sum_j e^{f_j}} \right)$$
Taking stock

- We have:
  - $\phi = \text{unravel(rgb2gray(img))}$, a feature extractor
  - $h(x) = W^T x$, a multiclass linear classifier
  - $L = \ldots$, a loss function
    \[
    L_i = -\log \left( \frac{e^{f_{y_i}}}{\sum_j e^{f_j}} \right)
    \]

- We don’t have:
  - a way to find a $W$ that results in a small $L$. 
Loss Functions

• Step 1: For a given $W$, decide on a **Loss Function**: a measure of how much we dislike this classifier.

• Step 2: use **optimization** to find the $W$ that *minimizes* the loss function.
  – Linear regression: solvable in closed form
  – Most of the time: no closed form.
How do we find a $W$ that minimizes $L$?

- Bad idea: Random search.

```python
# assume $X_{\text{train}}$ is the data where each column is an example (e.g. 3073 x 50,000)
# assume $Y_{\text{train}}$ 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)
```
How’d that go for you?

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 ~95%)
Finding a $W$ that minimizes $L$

- A better idea: walk downhill.
Gradient Descent: Generally

• Gradient of the loss function with respect to the *weights* tells us how to change the weights to improve the loss.
Gradient Descent

```python
# Vanilla Gradient Descent

while True:
    weights_grad = evaluate_gradient(loss_fun, data, weights)
    weights += - step_size * weights_grad # perform parameter update
```
Gradient Descent: Intuition
Gradient Descent: Intuition
Gradient Descent: Demo

  
  - select “Softmax” radio button at the bottom
Gradient Descent: Generally

• Gradient of the loss function with respect to the weights tells us how to change the weights to improve the loss.

• $L(X; W)$ depends on
  – All data points $x_1..x_n$
  – Very expensive to evaluate
Stochastic Gradient Descent

$L(X; W)$ depends on
- All data points $x_1..x_n$
- Weights $W$

• Very expensive to evaluate if you have a lot of data.

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

• Idea: consider only a few data points at a time.
• Loss is now computed using only a small batch (minibatch) of data points.
• Update weights the same way using the gradient of L wrt the weights.
Stochastic Gradient Descent: Intuition