CSCI 497P/597P: Computer Vision
Scott Wehrwein

Softmax, Regularization, Gradient Descent
Reading

• [http://cs231n.github.io/optimization-1/](http://cs231n.github.io/optimization-1/)
Announcements
Goals

• Understand the intuition behind the softmax classifier with cross-entropy loss and its interpretation of scores as unnormalized log probabilities.

• Understand how to train a classifier by minimizing a loss function using gradient descent.

• Understand the intuition behind using Stochastic (Minibatch) Gradient Descent.
Linear classifiers

- Equation: $w^T x + b = 0$
- Points on the same side are the same class
Multiclass Linear Classifiers:
Stack multiple $w^\top$ into a matrix.
Multiclass Linear Classifier: Geometric Interpretation

Slide: Fei-Fei Li, Justin Johnson, & Serena Yeung
How do we find a good $W, b$?

- Step 1: For a given $W, b$, decide on a **Loss Function**: a measure of how much we dislike the line.
- Step 2: use **optimization** to find the $W, b$ that minimize the loss function.
Loss Functions

• Step 1: For a given $W$, $b$, decide on a **Loss Function**: a measure of how much we dislike this classifier.
  – Last time: SVM loss (binary case)
  – Today: Softmax + cross-entropy loss

• Step 2: use **optimization** to find the $W$, $b$ that minimize the loss function.
  – Today: gradient descent
Loss Functions

• Step 1: For a given $W$, $b$, 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 Functions – SVM Loss

• SVM Loss:
  – Insists that data points are not just correctly classified, but a certain distance from the hyperplane:
  
  \[ L_i = \max(0, x_i, 1 - y_i(w^T x_i + b)) \]

  \( x_i = \text{i’th data point} \)
  \( y_i = \text{i’th data point’s true label:} \)
  -1 if red
  +1 if green
Loss Functions – SVM Loss

• SVM Loss:
  – Insists that data points are not just correctly classified, but a certain distance from the hyperplane:
  
    \[ L_i = \max(0, x_i, 1 - y_i(w^T x_i + b)) \]

    \( x_i \) = i’th data point
    \( y_i \) = i’th data point’s true label:
    -1 if red
    +1 if green

  – \( L(w, b) = \Sigma_i L_i \)

  – Loss for a given line is the sum of the loss for all datapoints
Loss Functions – SVM Loss

- SVM Loss – multiclass case:
  - Insists that data points are not just correctly classified, but correct the class score is a certain amount higher than every other class score:
  - Let $f_j = \text{the score for class } j \ (f_j = w_j^T x)$

- $L_i = \sum_j \max(0, 1 + s_j - s_{y_i})$
Softmax Classifier / Cross-Entropy Loss: Intuition

\( W^T x + b \) 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?
Binary Equivalent:
Logistic Regression Loss
Softmax Classifier / Cross-Entropy Loss: Intuition

$W^T x + b$ 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? They’re not:
- not always nonnegative
- don’t sum to 1

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

\( f = W^T x \) 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:

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

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

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

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

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

Which do we prefer – \( W \), or 2\( W \)?

Slide: Fei-Fei Li, Justin Johnson, & Serena Yeung
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.

Slide: Fei-Fei Li, Justin Johnson, & Serena Yeung
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

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

*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):** \[ R(W) = \sum_k \sum_l \beta W_{k,l}^2 + |W_{k,l}| \]
Regularization: Prefer Simpler Models
Regularization: Prefer Simpler Models
Regularization: Prefer Simpler Models

Regularization pushes against fitting the data too well so we don't fit noise in the data.
\[ L(W) = \frac{1}{N} \sum_{i=1}^{N} L_i(f(x_i, W), y_i) + \lambda R(W) \]

Data loss: Model predictions should match training data

Regularization: Prevent the model from doing too well on training data
How do we find a good classifier?

• Step 1: For a given W, b, decide on a **Loss Function**: a measure of how much we dislike this classifier.
  – Last time: SVM loss (binary case)
  – Today: Softmax + cross-entropy loss

• Step 2: use **optimization** to find the W, b that *minimize* the loss function.
  – Today: gradient descent
Optimization
How do we find a $W$ that minimizes $L$?

- Bad idea: 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 9.048034, 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)
```

Slide: Fei-Fei Li, Justin Johnson, & Serena Yeung
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

# Vanilla Gradient Descent

```python
while True:
    weights_grad = evaluate_gradient(loss_fun, data, weights)
    weights += - step_size * weights_grad  # perform parameter update
```
Gradient descent: SVM loss

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

\[ \nabla_{w_{y_i}} L_i = -\left( \sum_{j \neq y_i} 1(w_j^T x_i - w_{\bar{y}_i}^T x_i + \Delta > 0) \right) x_i \]
# Vanilla Gradient Descent

```python
while True:
    weights_grad = evaluate_gradient(loss_fun, data, weights)
    weights += -step_size * weights_grad # perform parameter update
```
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
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.