Image Recognition
The task in Image Classification is to predict a single label (or a distribution over labels as shown here to indicate our confidence) for a given image. Images are 3-dimensional arrays of integers from 0 to 255, of size Width x Height x 3. The 3 represents the three color channels Red, Green, Blue.
The Problem: Semantic Gap

What the computer sees

An image is just a big grid of numbers between [0, 255]:

e.g. 800 x 600 x 3
(3 channels RGB)
**Challenges:** Viewpoint variation

All pixels change when the camera moves!
Challenges: Illumination
Challenges: Deformation
Challenges: Occlusion
Challenges: Background Clutter
Challenges: Intraclass variation
An image classifier

```
def classify_image(image):
    # Some magic here?
    return class_label
```

Unlike e.g. sorting a list of numbers, **no obvious way** to hard-code the algorithm for recognizing a cat, or other classes.
Attempts have been made
Machine Learning: Data-Driven Approach

1. Collect a dataset of images and labels
2. Use Machine Learning to train a classifier
3. Evaluate the classifier on new images

Example training set

```python
def train(images, labels):
    # Machine learning!
    return model

def predict(model, test_images):
    # Use model to predict labels
    return test_labels
```

- airplane
- automobile
- bird
- cat
- deer
First classifier: Nearest Neighbor

```python
def train(images, labels):
    # Machine learning!
    return model

def predict(model, test_images):
    # Use model to predict labels
    return test_labels
```

- Memorize all data and labels
- Predict the label of the most similar training image
Example Dataset: CIFAR10

10 classes
50,000 training images
10,000 testing images

airplane
automobile
bird
cat
deer
dog
frog
horse
ship
truck

Example Dataset: **CIFAR10**

10 classes
50,000 training images
10,000 testing images

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

Test images and nearest neighbors
Distance Metric to compare images

L1 distance:

\[ d_1(I_1, I_2) = \sum_p |I_{1p} - I_{2p}| \]

<table>
<thead>
<tr>
<th>test image</th>
<th>training image</th>
<th>pixel-wise absolute value differences</th>
</tr>
</thead>
<tbody>
<tr>
<td>56 32 10 18</td>
<td>10 20 24 17</td>
<td>46 12 14 1 add 456</td>
</tr>
<tr>
<td>90 23 128 133</td>
<td>8 10 89 100</td>
<td>82 13 39 33</td>
</tr>
<tr>
<td>24 26 178 200</td>
<td>12 16 178 170</td>
<td>12 10 0 30</td>
</tr>
<tr>
<td>2 0 255 220</td>
<td>4 32 233 112</td>
<td>2 32 22 108</td>
</tr>
</tbody>
</table>
import numpy as np

class NearestNeighbor:
    def __init__(self):
        pass

    def train(self, X, y):
        """ X is N x D where each row is an example. Y is 1-dimension of size N ""
        # the nearest neighbor classifier simply remembers all the training data
        self.Xtr = X
        self.ytr = y

    def predict(self, X):
        """ X is N x D where each row is an example we wish to predict label for ""
        num_test = X.shape[0]
        # lets make sure that the output type matches the input type
        Ypred = np.zeros(num_test, dtype = self.ytr.dtype)

        # loop over all test rows
        for i in range(num_test):
            # find the nearest training image to the i'th test image
            # using the l1 distance (sum of absolute value differences)
            distances = np.sum(np.abs(self.Xtr - X[i,:]), axis = 1)
            min_index = np.argmin(distances)  # get the index with smallest distance
            Ypred[i] = self.ytr[min_index]  # predict the label of the nearest example

        return Ypred
import numpy as np

class NearestNeighbor:
    def __init__(self):
        pass

    def train(self, X, y):
        """ X is N x D where each row is an example. Y is 1-dimension of size N """
        # the nearest neighbor classifier simply remembers all the training data
        self.Xtr = X
        self.ytr = y

    def predict(self, X):
        """ X is N x D where each row is an example we wish to predict label for """
        num_test = X.shape[0]
        # lets make sure that the output type matches the input type
        Ypred = np.zeros(num_test, dtype = self.ytr.dtype)

        # loop over all test rows
        for i in xrange(num_test):
            # find the nearest training image to the i'th test image
            # using the L1 distance (sum of absolute value differences)
            distances = np.sum(np.abs(self.Xtr - X[i,:]), axis = 1)
            min_index = np.argmin(distances) # get the index with smallest distance
            Ypred[i] = self.ytr[min_index] # predict the label of the nearest example

        return Ypred

Nearest Neighbor classifier

Memorize training data
Nearest Neighbor classifier

For each test image:
Find closest train image
Predict label of nearest image

```python
import numpy as np
class NearestNeighbor:
    def __init__(self):
        pass

    def train(self, X, y):
        """X is N x D where each row is an example. Y is 1-dimension of size N ""
        # the nearest neighbor classifier simply remembers all the training data
        self.Xtr = X
        self.ytr = y

    def predict(self, X):
        """X is N x D where each row is an example we wish to predict label for ""
        num_test = X.shape[0]
        # lets make sure that the output type matches the input type
        Ypred = np.zeros(num_test, dtype = self.ytr.dtype)

        # loop over all test rows
        for i in xrange(num_test):
            # find the nearest training image to the i'th test image
            # using the L1 distance (sum of absolute value differences)
            distances = np.sum(np.abs(self.Xtr - X[i,:]), axis = 1)
            min_index = np.argmin(distances) # get the index with smallest distance
            Ypred[i] = self.ytr[min_index] # predict the label of the nearest example

        return Ypred
```
Nearest Neighbor classifier

Q: With N examples, how fast are training and prediction?

A: Train $O(1)$, predict $O(N)$

This is bad: we want classifiers that are fast at prediction; slow for training is ok.
What does this look like?
K-Nearest Neighbors

Instead of copying label from nearest neighbor, take \textit{majority vote} from K closest points
What does this look like?
K-Nearest Neighbors: Distance Metric

L1 (Manhattan) distance

\[ d_1(I_1, I_2) = \sum_p |I_{1p} - I_{2p}| \]

L2 (Euclidean) distance

\[ d_2(I_1, I_2) = \sqrt{\sum_p (I_{1p} - I_{2p})^2} \]
K-Nearest Neighbors: Distance Metric

**L1 (Manhattan) distance**

\[ d_1(I_1, I_2) = \sum_p |I_1^p - I_2^p| \]

**L2 (Euclidean) distance**

\[ d_2(I_1, I_2) = \sqrt{\sum_p (I_1^p - I_2^p)^2} \]
Hyperparameters

What is the best value of $k$ to use?
What is the best distance to use?

These are hyperparameters: choices about the algorithm that we set rather than learn

Very problem-dependent.
Must try them all out and see what works best.
Setting Hyperparameters

**Idea #1**: Choose hyperparameters that work best on the data

Your Dataset
Setting Hyperparameters

Idea #1: Choose hyperparameters that work best on the data

BAD: $K = 1$ always works on training data

Your Dataset
Setting Hyperparameters

**Idea #1**: Choose hyperparameters that work best on the data

BAD: $K = 1$ always works perfectly on training data

Your Dataset

**Idea #2**: Split data into **train** and **test**, choose hyperparameters that work best on test data

| train | test |
Setting Hyperparameters

Idea #1: Choose hyperparameters that work best on the data

BAD: K = 1 always works perfectly on training data

Idea #2: Split data into train and test, choose hyperparameters that work best on test data

BAD: No idea how algorithm will perform on new data
Setting Hyperparameters

**Idea #1:** Choose hyperparameters that work best on the data

BAD: K = 1 always works perfectly on training data

<table>
<thead>
<tr>
<th>Your Dataset</th>
</tr>
</thead>
<tbody>
<tr>
<td>train</td>
</tr>
<tr>
<td>test</td>
</tr>
</tbody>
</table>

**Idea #2:** Split data into train and test, choose hyperparameters that work best on test data

BAD: No idea how algorithm will perform on new data

<table>
<thead>
<tr>
<th>train</th>
<th>validation</th>
<th>test</th>
</tr>
</thead>
</table>

**Idea #3:** Split data into train, val, and test; choose hyperparameters on val and evaluate on test

Better!

<table>
<thead>
<tr>
<th>train</th>
<th>validation</th>
<th>test</th>
</tr>
</thead>
</table>
Setting Hyperparameters

Idea #4: Cross-Validation: Split data into folds, try each fold as validation and average the results

Useful for small datasets, but not used too frequently in deep learning
Setting Hyperparameters

Example of 5-fold cross-validation for the value of $k$.

Each point: single outcome.

The line goes through the mean, bars indicated standard deviation

(Seems that $k \sim 7$ works best for this data)
k-Nearest Neighbor on images never used.

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

(all 3 images have same L2 distance to the one on the left)
k-Nearest Neighbor on images **never used.**

- Curse of dimensionality

Dimensions = 1  
Points = 4

Dimensions = 2  
Points = $4^2$

Dimensions = 3  
Points = $4^3$
K-Nearest Neighbors: Summary

In **Image classification** we start with a **training set** of images and labels, and must predict labels on the **test set**

The **K-Nearest Neighbors** classifier predicts labels based on nearest training examples

Distance metric and K are **hyperparameters**

Choose hyperparameters using the **validation set**; only run on the test set once at the very end!
Linear Classification

Neural Network

Linear classifiers
Two young girls are playing with lego toy.

Boy is doing backflip on wakeboard

Man in black shirt is playing guitar.

Construction worker in orange safety vest is working on road.


Recall CIFAR10

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

10 classes

50,000 training images
each image is 32x32x3

10,000 test images.
Linear Support Vector Machine (SVM)

- **Hyperplane**
  \[ w^T x + b = 0 \]

- **Extra scale constraint:**
  \[ \min_{i=1,...,n} |w^T x_i + b| = 1 \]

- **This implies:**
  \[ w^T (x_a - x_b) = 2 \]
  \[ \rho = \frac{2}{||x_a - x_b||_2} = \frac{2}{||w||_2} \]
Parametric Approach

Image

Array of $32 \times 32 \times 3$ numbers (3072 numbers total)

$f(x, W)$

Parameters or weights

10 numbers giving class scores
Parametric Approach: Linear Classifier

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

Array of \(32 \times 32 \times 3\) numbers (3072 numbers total)

\(W\)

parameters or weights

Image

10 numbers giving class scores
Parametric Approach: Linear Classifier

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

Image

Array of $32 \times 32 \times 3$ numbers (3072 numbers total)

$W$ parameters or weights

10 numbers giving class scores
Parametric Approach: Linear Classifier

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

- **Image**
- Array of \(32 \times 32 \times 3\) numbers (3072 numbers total)
- **\(W\)** parameters or weights
- **\(f(x, W)\)** 10 numbers giving class scores
- **\(x\)** \(10 \times 1\)
- **\(W\)** \(10 \times 3072\)
- **\(b\)** \(10 \times 1\)
Example with an image with 4 pixels, and 3 classes (cat/dog/ship)

Input image

Stretch pixels into column

<table>
<thead>
<tr>
<th>56</th>
<th>231</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>56</th>
</tr>
</thead>
<tbody>
<tr>
<td>231</td>
</tr>
<tr>
<td>24</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>
Example with an image with 4 pixels, and 3 classes (cat/dog/ship)
Example with an image with 4 pixels, and 3 classes (cat/dog/ship)

Algebraic Viewpoint

\[ f(x, W) = Wx \]
Interpreting a Linear Classifier: Visual Viewpoint

airplane
automobile
bird
cat
deer
dog
frog
horse
ship
truck

Input image

\[
\begin{align*}
W & = \begin{bmatrix} 0.2 & -0.5 \\ 0.1 & 2.0 \end{bmatrix} \\
b & = \begin{bmatrix} 1.1 \\ 3.2 \end{bmatrix} \\
\end{align*}
\]

Score

\[
\begin{align*}
& = -96.8 \\
& = 437.9 \\
& = 61.95 \\
\end{align*}
\]
Interpreting a Linear Classifier: Geometric Viewpoint

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

Array of $32 \times 32 \times 3$ numbers (3072 numbers total)
Hard cases for a linear classifier

**Class 1:**
First and third quadrants

**Class 2:**
Second and fourth quadrants

**Class 1:**
1 <= L2 norm <= 2

**Class 2:**
Everything else

**Class 1:**
Three modes

**Class 2:**
Everything else
Linear Classifier: Three Viewpoints

**Algebraic Viewpoint**

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

**Visual Viewpoint**

One template per class

**Geometric Viewpoint**

Hyperplanes cutting up space
So far: Defined a (linear) score function $f(x,W) = Wx + b$

Example class scores for 3 images for some $W$:

<table>
<thead>
<tr>
<th></th>
<th>airplane</th>
<th>automobile</th>
<th>bird</th>
<th>cat</th>
<th>deer</th>
<th>dog</th>
<th>frog</th>
<th>horse</th>
<th>ship</th>
<th>truck</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-3.45</td>
<td>-8.87</td>
<td>0.09</td>
<td>2.9</td>
<td>4.48</td>
<td>8.02</td>
<td>3.78</td>
<td>1.06</td>
<td>-0.36</td>
<td>-0.72</td>
</tr>
<tr>
<td></td>
<td>-0.51</td>
<td>6.04</td>
<td>5.31</td>
<td>-4.22</td>
<td>-4.19</td>
<td>3.58</td>
<td>4.49</td>
<td>-4.37</td>
<td>-2.09</td>
<td>-2.93</td>
</tr>
<tr>
<td></td>
<td>3.42</td>
<td>4.64</td>
<td>2.65</td>
<td>2.64</td>
<td>5.55</td>
<td>5.1</td>
<td>3.42</td>
<td>4.93</td>
<td>-1.5</td>
<td>6.14</td>
</tr>
</tbody>
</table>

How can we tell whether this $W$ is good or bad?
Coming up:
- Loss function
- Optimization
- ConvNets!

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

(quantifying what it means to have a “good” W)
(quantifying what it means to have a “good” W)
(start with random \( W \) and find a \( W \) that minimizes the loss)
(tweak the functional form of \( f \))
Suppose: 3 training examples, 3 classes.
With some $W$ the scores $f(x, W) = Wx$ are:

<table>
<thead>
<tr>
<th>Class</th>
<th>Score 1</th>
<th>Score 2</th>
<th>Score 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>cat</td>
<td>3.2</td>
<td>1.3</td>
<td>2.2</td>
</tr>
<tr>
<td>car</td>
<td>5.1</td>
<td>4.9</td>
<td>2.5</td>
</tr>
<tr>
<td>frog</td>
<td>-1.7</td>
<td>2.0</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 a sum of loss over examples:

$$L = \frac{1}{N} \sum_i L_i(f(x_i, W), y_i)$$
Suppose: 3 training examples, 3 classes. With some \( W \) the scores \( f(x, W) = Wx \) are:

\[
\begin{array}{ccc}
\text{cat} & 3.2 & 1.3 & 2.2 \\
\text{car} & 5.1 & 4.9 & 2.5 \\
\text{frog} & -1.7 & 2.0 & -3.1 \\
\end{array}
\]

**Multiclass SVM loss:**

Given an example \((x_i, y_i)\) where \( x_i \) is the image and where \( y_i \) is the (integer) label, and using the shorthand for the scores vector: \( s = f(x_i, W) \)

the SVM loss has the form:

\[
L_i = \sum_{j \neq y_i} \left\{ \begin{array}{cl}
0 & \text{if } s_{y_i} \geq s_j + 1 \\
(s_j - s_{y_i} + 1) & \text{otherwise}
\end{array} \right.
\]

\[
= \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)
\]
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>car</th>
<th>frog</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2</td>
<td>1.3</td>
<td>2.2</td>
<td></td>
</tr>
<tr>
<td>5.1</td>
<td>4.9</td>
<td>2.5</td>
<td></td>
</tr>
<tr>
<td>-1.7</td>
<td>2.0</td>
<td>-3.1</td>
<td></td>
</tr>
</tbody>
</table>

**Multiclass SVM loss:**

"Hinge loss"

\[
L_i = \sum_{j \neq y_i} \begin{cases} 
0 & \text{if } s_{y_i} \geq s_j + 1 \\
(s_j - s_{y_i} + 1) & \text{otherwise}
\end{cases}
= \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)
\]
Suppose: 3 training examples, 3 classes.
With some $W$ the scores $f(x, W) = Wx$ are:

<table>
<thead>
<tr>
<th></th>
<th>Score 1</th>
<th>Score 2</th>
<th>Score 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>cat</td>
<td>3.2</td>
<td>1.3</td>
<td>2.2</td>
</tr>
<tr>
<td>car</td>
<td>5.1</td>
<td>4.9</td>
<td>2.5</td>
</tr>
<tr>
<td>frog</td>
<td>-1.7</td>
<td>2.0</td>
<td>-3.1</td>
</tr>
</tbody>
</table>

**Multiclass SVM loss:**

Given an example $(x_i, y_i)$ where $x_i$ is the image and $y_i$ is the (integer) label, and using the shorthand for the scores vector: $s = f(x_i, W)$

the SVM loss has the form:

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$
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>car</th>
<th>frog</th>
</tr>
</thead>
<tbody>
<tr>
<td></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>

**Multiclass SVM loss:**

Given an example $(x_i, y_i)$ where $x_i$ is the image and where $y_i$ is the (integer) label, and using the shorthand for the scores vector: $s = f(x_i, W)$

the SVM loss has the form:

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

$$= \max(0, 5.1 - 3.2 + 1)$$

$$+ \max(0, -1.7 - 3.2 + 1)$$

$$= \max(0, 2.9) + \max(0, -3.9)$$

$$= 2.9 + 0$$

$$= 2.9$$
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>Car</th>
<th>Frog</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cat</td>
<td>3.2</td>
<td>1.3</td>
<td>2.2</td>
</tr>
<tr>
<td>Car</td>
<td>5.1</td>
<td>4.9</td>
<td>2.5</td>
</tr>
<tr>
<td>Frog</td>
<td>-1.7</td>
<td>2.0</td>
<td>-3.1</td>
</tr>
</tbody>
</table>

**Multiclass SVM loss:**

Given an example $(x_i, y_i)$ where $x_i$ is the image and where $y_i$ is the (integer) label, and using the shorthand for the scores vector: $s = f(x_i, W)$

the SVM loss has the form:

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

$$= \max(0, 1.3 - 4.9 + 1) + \max(0, 2.0 - 4.9 + 1)$$

$$= \max(0, -2.6) + \max(0, -1.9)$$

$$= 0 + 0$$

$$= 0$$
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>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>score</td>
<td>1.3</td>
<td>4.9</td>
<td>2.0</td>
</tr>
<tr>
<td>score</td>
<td>2.2</td>
<td>2.5</td>
<td>-3.1</td>
</tr>
</tbody>
</table>

**Multiclass SVM loss:**

Given an example $(x_i, y_i)$ where $x_i$ is the image and $y_i$ is the (integer) label,

and using the shorthand for the scores vector: $s = f(x_i, W)$

the SVM loss has the form:

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

\[
= \max(0, 2.2 - (-3.1) + 1) \\
+ \max(0, 2.5 - (-3.1) + 1) \\
= \max(0, 6.3) + \max(0, 6.6) \\
= 6.3 + 6.6 \\
= 12.9
\]
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></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3.2</td>
<td>1.3</td>
<td>2.2</td>
</tr>
<tr>
<td>car</td>
<td>5.1</td>
<td>4.9</td>
<td>2.5</td>
</tr>
<tr>
<td>frog</td>
<td>-1.7</td>
<td>2.0</td>
<td>-3.1</td>
</tr>
</tbody>
</table>

Losses:
- cat: 2.9
- car: 0
- frog: 12.9

Multiclass SVM loss:

Given an example $(x_i, y_i)$ where $x_i$ is the image and $y_i$ is the (integer) label,

and using the shorthand for the scores vector: $s = f(x_i, W)$

the SVM loss has the form:

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

Loss over full dataset is average:

$$L = \frac{1}{N} \sum_{i=1}^{N} L_i$$

$L = (2.9 + 0 + 12.9)/3 = 5.27$
Suppose: 3 training examples, 3 classes. With some $W$ the scores $f(x, W) = WX$ are:

- Cat: 3.2, 1.3, 2.2
- Car: 5.1, 4.9, 2.5
- Frog: -1.7, 2.0, -3.1

Losses: 2.9, 0, 12.9

**Multiclass SVM loss:**

Given an example $(x_i, y_i)$ where $x_i$ is the image and where $y_i$ is the (integer) label, and using the shorthand for the scores vector: $s = f(x_i, W)$

the SVM loss has the form:

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

**Q:** What happens to loss if car scores change a bit?
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>frog</th>
<th>car</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2</td>
<td>-1.7</td>
<td>5.1</td>
<td>5.1</td>
</tr>
<tr>
<td>1.3</td>
<td>2.0</td>
<td>4.9</td>
<td>4.9</td>
</tr>
<tr>
<td>2.2</td>
<td>-3.1</td>
<td>2.5</td>
<td>2.5</td>
</tr>
</tbody>
</table>

Losses: 2.9, 0, 12.9

Multiclass SVM loss:

Given an example $(x_i, y_i)$ where $x_i$ is the image and where $y_i$ is the (integer) label,

and using the shorthand for the scores vector: $s = f(x_i, W)$

the SVM loss has the form:

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

Q2: what is the min/max possible loss?
Suppose: 3 training examples, 3 classes.
With some \( W \) the scores \( f(x, W) = Wx \) are:

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

**Multiclass SVM loss:**

Given an example \((x_i, y_i)\) where \( x_i \) is the image and where \( y_i \) is the (integer) label, and using the shorthand for the scores vector: \( s = f(x_i, W) \)

the SVM loss has the form:

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

**Q4:** What if the sum was over all classes? (including \( j = y_i \))
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>car</th>
<th>frog</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3.2</td>
<td>1.3</td>
<td>2.2</td>
</tr>
<tr>
<td></td>
<td>5.1</td>
<td>4.9</td>
<td>2.5</td>
</tr>
<tr>
<td></td>
<td>-1.7</td>
<td>2.0</td>
<td>-3.1</td>
</tr>
</tbody>
</table>

**Multiclass SVM loss:**

Given an example $(x_i, y_i)$ where $x_i$ is the image and where $y_i$ is the (integer) label, and using the shorthand for the scores vector: $s = f(x_i, W)$, the SVM loss has the form:

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

Q5: What if we used mean instead of sum?
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>Scores</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>Max</td>
<td>2.2</td>
<td>2.5</td>
<td>-3.1</td>
</tr>
</tbody>
</table>

**Multiclass SVM loss:**

Given an example $(x_i, y_i)$ where $x_i$ is the image and $y_i$ is the (integer) label, and using the shorthand for the scores vector: $s = f(x_i, W)$ the SVM loss has the form:

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

Q6: What if we used

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)^2$$
\[ f(x, W) = Wx \]
\[
L = \frac{1}{N} \sum_{i=1}^{N} \sum_{j \neq y_i} \max(0, f(x_i; W)_j - f(x_i; W)_{y_i} + 1)
\]

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

No! \( 2W \) is also has \( L = 0 \! \)!
How do we choose between \( W \) and \( 2W \)?
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>1.3</th>
<th>2.2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>car</td>
<td>5.1</td>
<td>4.9</td>
<td>2.5</td>
</tr>
<tr>
<td>frog</td>
<td>-1.7</td>
<td>2.0</td>
<td>-3.1</td>
</tr>
<tr>
<td>Losses:</td>
<td>2.9</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:**

$$= \max(0, 2.6 - 9.8 + 1) + \max(0, 4.0 - 9.8 + 1)$$

$$= \max(0, -6.2) + \max(0, -4.8)$$

$$= 0 + 0$$

$$= 0$$
Data loss: Model predictions should match training data

\[ L(W) = \frac{1}{N} \sum_{i=1}^{N} L_i(f(x_i, W), y_i) \]
Data loss: Model predictions should match training data

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

**Data loss**: Model predictions should match training data
\[ L(W) = \frac{1}{N} \sum_{i=1}^{N} L_i(f(x_i, W), y_i) \]

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

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

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

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

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

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

"Generalization"

we are mainly interested in reaching good performance for **test data** and not for 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

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

Occam’s Razor

“Among competing hypotheses, the simplest is the best”

William of Ockham 1287-1347
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

\( \lambda = \) 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 \textit{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

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

L2 regularization likes to “spread out” the weights
Softmax Classifier (Multinomial Logistic Regression)

Want to interpret raw classifier scores as probabilities

$$s = f(x_i; W)$$

$$P(Y = k | X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}$$ Softmax Function

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>cat</td>
<td>3.2</td>
</tr>
<tr>
<td>car</td>
<td>5.1</td>
</tr>
<tr>
<td>frog</td>
<td>-1.7</td>
</tr>
</tbody>
</table>
Softmax Classifier (Multinomial Logistic Regression)

Want to interpret raw classifier scores as probabilities

\[ s = f(x_i; W) \]

\[ P(Y = k | X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}} \]

Softmax Function

Probabilities must be \( \geq 0 \)

<table>
<thead>
<tr>
<th>Class</th>
<th>Raw Score</th>
<th>Unnormalized Probability</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>cat</td>
<td>3.2</td>
<td>24.5</td>
<td>0.18</td>
</tr>
<tr>
<td>car</td>
<td>5.1</td>
<td>164.0</td>
<td></td>
</tr>
<tr>
<td>frog</td>
<td>-1.7</td>
<td>0.18</td>
<td></td>
</tr>
</tbody>
</table>
**Softmax Classifier** (Multinomial Logistic Regression)

Want to interpret raw classifier scores as **probabilities**

\[ s = f(x_i; W) \]

\[ P(Y = k | X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}} \]

Softmax Function

<table>
<thead>
<tr>
<th>Class</th>
<th>Score</th>
<th>Probabilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>cat</td>
<td>3.2</td>
<td>0.13</td>
</tr>
<tr>
<td>car</td>
<td>5.1</td>
<td>0.87</td>
</tr>
<tr>
<td>frog</td>
<td>-1.7</td>
<td>0.00</td>
</tr>
</tbody>
</table>

Probabilities must be >= 0

Probabilities must sum to 1

unnormalized probabilities

probabilities
Softmax Classifier (Multinomial Logistic Regression)

Want to interpret raw classifier scores as probabilities

\[ s = f(x_i; W) \]

\[ P(Y = k | X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}} \]

Probabilities must be \( \geq 0 \)

Probabilities must sum to 1

\[ L_i = -\log P(Y = y_i | X = x_i) \]

\[ L_i = -\log(0.13) = 0.89 \]

Unnormalized log-probabilities / logits

unnormalized probabilities

probabilities
Recap

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

\[
L_i = - \log\left( \frac{e^{sy_i}}{\sum_j e^{sj}} \right)
\]

**Softmax**

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

**SVM**

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

How do we find the best \(W\)?

But: How do we find \(W\)?
Optimization
looking for minima
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)
```
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%)
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 $W$:  

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

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 $dW$:  

\[
\begin{bmatrix}
-2.5, \\
?, \\
?, \\
(1.25322 - 1.25347)/0.0001 = -2.5 \\
?, \\
?, \\
?,
\end{bmatrix}
\]

\[
\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x + h) - f(x)}{h}
\]
<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>
</tbody>
</table>

**loss 1.25347** **loss 1.25347**
Gradient $dW$:

$\begin{align*}
&\text{current } W: \\
&\begin{bmatrix}
0.34, \\
-1.11, \\
0.78, \\
0.12, \\
0.55, \\
2.81, \\
-3.1, \\
-1.5, \\
0.33,\
\end{bmatrix} \\
&\text{loss } 1.25347
\end{align*}$

$\begin{align*}
&W + h \text{ (third dim)}: \\
&\begin{bmatrix}
0.34, \\
-1.11, \\
0.78 + 0.0001, \\
0.12, \\
0.55, \\
2.81, \\
-3.1, \\
-1.5, \\
0.33,\
\end{bmatrix} \\
&\text{loss } 1.25347
\end{align*}$

$\begin{align*}
\text{gradient } dW: \\
&\begin{bmatrix}
-2.5, \\
0.6, \\
0, \\
?, \\
?, \\
0, \\
?, \\
?, \\
?, \\
\end{bmatrix}
\end{align*}$

$(1.25347 - 1.25347)/0.0001 = 0$

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

| 0.34,   |
|———|   |
| -1.11, |
| 0.78 + 0.0001, |
| 0.12,  |
| 0.55,  |
| 2.81,  |
| -3.1,  |
| -1.5,  |
| 0.33,…]|

**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 $\nabla_W L$
In summary:

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

=>$\textbf{In practice:}$ Always use analytic gradient, but check implementation with numerical gradient. This is called a gradient check.
Gradient Descent

```
# 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

```python
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
```
Aside: Image Features

\[ f(x) = Wx \]

Class scores
Aside: Image Features

\[ f(x) = Wx \]

Class scores

Feature Representation
Image Features: Motivation

Cannot separate red and blue points with linear classifier
Image Features: Motivation

Cannot separate red and blue points with linear classifier

\[ f(x, y) = (r(x, y), \theta(x, y)) \]

After applying feature transform, points can be separated by linear classifier
Example: Color Histogram
Example: Histogram of Oriented Gradients (HoG)

Divide image into 8x8 pixel regions
Within each region quantize edge direction into 9 bins

Example: 320x240 image gets divided into 40x30 bins; in each bin there are 9 numbers so feature vector has 30*40*9 = 10,800 numbers
Aside: Image Features
Feature Extraction

10 numbers giving scores for classes

training

Image features vs ConvNets