Lecture 15: Introduction to Deep Learning

COS 429: Computer Vision
Image Classification: A core task in Computer Vision

(assume given set of discrete labels)
{dog, cat, truck, plane, ...}

cat
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!

This image by Nikita is licensed under CC-BY 2.0

Credit: Fei-Fei Li & Justin Johnson & Serena Yeung
Challenges: Illumination
Challenges: Deformation
Challenges: Occlusion
Challenges: Background Clutter
Challenges: Intraclass variation
An image classifier

Unlike e.g. sorting a list of numbers,

**no obvious way** to hard-code the algorithm for recognizing a cat, or other classes.

```python
def classify_image(image):
    # Some magic here?
    return class_label
```
Attempts have been made

John Canny, “A Computational Approach to Edge Detection”, IEEE TPAMI 1986
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
```

Credit: Fei-Fei Li & Justin Johnson & Serena Yeung
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


Credit: Fei-Fei Li & Justin Johnson & Serena Yeung
Example Dataset: CIFAR10

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

Test images and nearest neighbors


Credit: Fei-Fei Li & Justin Johnson & Serena Yeung
## Distance Metric to compare images

**L1 distance:**

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

<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</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>

Credit: Fei-Fei Li & Justin Johnson & Serena Yeung
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
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

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

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

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

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

```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
```
What does this look like?
K-Nearest Neighbors

Instead of copying label from nearest neighbor, take **majority vote** from K closest points
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
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 perfectly 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 \texttt{train} and \texttt{test}, choose hyperparameters that work best on test data

\texttt{train} \hspace{1cm} \texttt{test}
Setting Hyperparameters

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

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

Your Dataset

| train | test |

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

Your Dataset

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

train | test

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

Better!

train | validation | test

Credit: Fei-Fei Li & Justin Johnson & Serena Yeung
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.
What does this look like?

Credit: Fei-Fei Li & Justin Johnson & Serena Yeung
What does this look like?

Credit: Fei-Fei Li & Justin Johnson & Serena Yeung
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
Recall CIFAR10

50,000 training images
each image is 32x32x3

10,000 test images.
Parametric Approach

Image

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

$f(x, W)$

10 numbers giving class scores

$W$

parameters or weights
Parametric Approach: Linear Classifier

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

Array of 32x32x3 numbers (3072 numbers total)

Image

\( W \)

parameters or weights

10 numbers giving class scores

Credit: Fei-Fei Li & Justin Johnson & Serena Yeung
Parametric Approach: Linear Classifier

$\text{Image}$

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

$\mathbf{W}$
parameters or weights

$\mathbf{f}(\mathbf{x}, \mathbf{W}) = \mathbf{W} \mathbf{x}$

$\mathbf{f}(\mathbf{x}, \mathbf{W})$ \hspace{1cm} 10 numbers giving class scores

$\mathbf{W}$ \hspace{1cm} 3072x1

$\mathbf{x}$ \hspace{1cm} 10x1

$\mathbf{x}$ \hspace{1cm} 10x3072

Credit: Fei-Fei Li & Justin Johnson & Serena Yeung
Parametric Approach: Linear Classifier

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

- **Image**: Array of $32 \times 32 \times 3$ numbers (3072 numbers total)
- **Parameters or Weights**: $W$ (10x3072)
- **Class Scores**: 10 numbers

Credit: Fei-Fei Li & Justin Johnson & Serena Yeung
Example with an image with 4 pixels, and 3 classes (cat/dog/ship)

```
<table>
<thead>
<tr>
<th></th>
<th>0.2</th>
<th>-0.5</th>
<th>0.1</th>
<th>2.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>56</td>
<td>1.5</td>
<td>1.3</td>
<td>2.1</td>
<td>0.0</td>
</tr>
<tr>
<td>231</td>
<td>0</td>
<td>0.25</td>
<td>0.2</td>
<td>-0.3</td>
</tr>
<tr>
<td>24</td>
<td>0</td>
<td>0.25</td>
<td>0.2</td>
<td>-0.3</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0.25</td>
<td>0.2</td>
<td>-0.3</td>
</tr>
</tbody>
</table>
```

\[ W \]

```
<table>
<thead>
<tr>
<th></th>
<th>1.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>56</td>
<td>3.2</td>
</tr>
<tr>
<td>231</td>
<td>3.2</td>
</tr>
<tr>
<td>24</td>
<td>3.2</td>
</tr>
<tr>
<td>2</td>
<td>3.2</td>
</tr>
</tbody>
</table>
```

\[ \begin{align*}
W & = C + D + S \\
C & = \text{Cat score} \quad \text{(-96.8)} \\
D & = \text{Dog score} \quad \text{(437.9)} \\
S & = \text{Ship score} \quad \text{(61.95)}
\end{align*} \]

Credit: Fei-Fei Li & Justin Johnson & Serena Yeung
Interpreting a Linear Classifier

What is this thing doing?

\[ f(x,W) = Wx + b \]
Interpreting a Linear Classifier

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

Example trained weights of a linear classifier trained on CIFAR-10:

Credit: Fei-Fei Li & Justin Johnson & Serena Yeung
Interpreting a Linear Classifier

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

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

Credit: Fei-Fei Li & Justin Johnson & Serena Yeung
Hard cases for a linear classifier

Class 1:
number of pixels > 0 odd

Class 2:
number of pixels > 0 even

Class 1:
$1 \leq \text{L2 norm} \leq 2$

Class 2:
Everything else

Class 1:
Three modes

Class 2:
Everything else
Neural Network

Linear classifiers

Credit: Fei-Fei Li & Justin Johnson & Serena Yeung
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>Class</th>
<th>Score 1</th>
<th>Score 2</th>
<th>Score 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>airplane</td>
<td>-3.45</td>
<td>-0.51</td>
<td>3.42</td>
</tr>
<tr>
<td>automobile</td>
<td>-8.87</td>
<td>6.04</td>
<td>4.64</td>
</tr>
<tr>
<td>bird</td>
<td>0.09</td>
<td>5.31</td>
<td>2.65</td>
</tr>
<tr>
<td>cat</td>
<td>2.9</td>
<td>4.22</td>
<td>5.1</td>
</tr>
<tr>
<td>deer</td>
<td>4.48</td>
<td>4.19</td>
<td>2.64</td>
</tr>
<tr>
<td>dog</td>
<td>8.02</td>
<td>3.58</td>
<td>5.55</td>
</tr>
<tr>
<td>frog</td>
<td>3.78</td>
<td>4.49</td>
<td>-4.34</td>
</tr>
<tr>
<td>horse</td>
<td>1.06</td>
<td>4.37</td>
<td>-1.5</td>
</tr>
<tr>
<td>ship</td>
<td>-0.36</td>
<td>-2.09</td>
<td>-4.79</td>
</tr>
<tr>
<td>truck</td>
<td>-0.72</td>
<td>-2.93</td>
<td>6.14</td>
</tr>
</tbody>
</table>

Credit: Fei-Fei Li & Justin Johnson & Serena Yeung
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>label</td>
<td>1.3</td>
<td>4.9</td>
<td>2.0</td>
</tr>
<tr>
<td>loss</td>
<td>2.2</td>
<td>2.5</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:

<table>
<thead>
<tr>
<th>Class</th>
<th>Score</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>

Previous losses:

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: $s = f(x_i, W)$

Least-squares regression has the form

$$L_i = \sum_i ||s_i - y_i||^2$$

The SVM loss has the form:

$$L_i = \sum_{j \neq y_i} \begin{cases} 0 & \text{if } s_{y_i} \geq s_j + 1 \\ s_j - s_{y_i} + 1 & \text{otherwise} \end{cases} = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$
Softmax Classifier (Multinomial Logistic Regression)

cat  3.2

frog  -1.7

car  5.1
**Softmax Classifier** (Multinomial Logistic Regression)

scores = unnormalized log probabilities of the classes.

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

cat \quad 3.2

car \quad 5.1

frog \quad -1.7
**Softmax Classifier** (Multinomial Logistic Regression)

Scores = unnormalized log probabilities of the classes.

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

where

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

cat \hspace{0.5cm} 3.2

car \hspace{0.5cm} 5.1

frog \hspace{0.5cm} -1.7
Softmax Classifier (Multinomial Logistic Regression)

scores = unnormalized log probabilities of the classes.

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

where \( s = f(x_i; W) \)

<table>
<thead>
<tr>
<th>Class</th>
<th>Score</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 function
Softmax Classifier (Multinomial Logistic Regression)

scores = unnormalized log probabilities of the classes.

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

where 

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

Want to maximize the log likelihood, or (for a loss function) to minimize the negative log likelihood of the correct class:

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

<table>
<thead>
<tr>
<th>cat</th>
<th>3.2</th>
</tr>
</thead>
<tbody>
<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)

scores = unnormalized log probabilities of the classes.

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

where

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

Want to maximize the log likelihood, or (for a loss function) to minimize the negative log likelihood of the correct class:

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

In summary:

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

$$L_i = -\log \left( \frac{e^{y_i}}{\sum_j e^{s_j}} \right)$$

<table>
<thead>
<tr>
<th>Object</th>
<th>Unnormalized Log Probability</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>

Credit: Fei-Fei Li & Justin Johnson & Serena Yeung
**Softmax Classifier** (Multinomial Logistic Regression)

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

- **unnormalized log probabilities**
  - cat: 3.2
  - car: 5.1
  - frog: -1.7

- **unnormalized probabilities**
  - cat: \( e^{3.2} = 24.5 \)
  - car: \( e^{5.1} = 164.0 \)
  - frog: \( e^{-1.7} = 0.18 \)

Credit: Fei-Fei Li & Justin Johnson & Serena Yeung
Softmax Classifier (Multinomial Logistic Regression)

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

<table>
<thead>
<tr>
<th></th>
<th>Unnormalized probabilities</th>
<th>Unnormalized log probabilities</th>
<th>Probabilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>cat</td>
<td>3.2</td>
<td>24.5</td>
<td>0.13</td>
</tr>
<tr>
<td>car</td>
<td>5.1</td>
<td>164.0</td>
<td>0.87</td>
</tr>
<tr>
<td>frog</td>
<td>-1.7</td>
<td>0.18</td>
<td>0.00</td>
</tr>
</tbody>
</table>

Credit: Fei-Fei Li & Justin Johnson & Serena Yeung
Softmax Classifier (Multinomial Logistic Regression)

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

unnormalized log probabilities

<table>
<thead>
<tr>
<th>cat</th>
<th>3.2</th>
<th>24.5</th>
<th>0.13</th>
</tr>
</thead>
<tbody>
<tr>
<td>car</td>
<td>5.1</td>
<td>164.0</td>
<td>0.87</td>
</tr>
<tr>
<td>frog</td>
<td>-1.7</td>
<td>0.18</td>
<td>0.00</td>
</tr>
</tbody>
</table>

exp
normalize

unnormalized probabilities

\[ L_i = -\log(0.13) = 0.89 \]
Softmax Classifier (Multinomial Logistic Regression)

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

Q: What is the min/max possible loss \( L_i \)?

<table>
<thead>
<tr>
<th></th>
<th>cat</th>
<th>frog</th>
<th>car</th>
</tr>
</thead>
<tbody>
<tr>
<td>( s )</td>
<td>3.2</td>
<td>-1.7</td>
<td>5.1</td>
</tr>
</tbody>
</table>

unnormalized log probabilities

\[ \exp(3.2) \rightarrow 24.5 \]
\[ \exp(-1.7) \rightarrow 0.18 \]
\[ \exp(5.1) \rightarrow 164.0 \]

unnormalized probabilities

\[ \frac{24.5}{164.0+0.18+0.00} = 0.13 \]
\[ \frac{164.0}{164.0+0.18+0.00} = 0.87 \]
\[ \frac{0.18}{164.0+0.18+0.00} = 0.00 \]

probabilities

\[ L_i = -\log(0.13) = 0.89 \]
Softmax Classifier (Multinomial Logistic Regression)

$$L_i = -\log\left( \frac{e^{s_{y_i}}}{\sum_j e^{s_j}} \right)$$

Q2: Usually at initialization $W$ is small so all $s \approx 0$. What is the loss?

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>cat</td>
<td>3.2</td>
<td>24.5</td>
<td>0.13</td>
</tr>
<tr>
<td>car</td>
<td>5.1</td>
<td>164.0</td>
<td>0.87</td>
</tr>
<tr>
<td>frog</td>
<td>-1.7</td>
<td>0.18</td>
<td>0.00</td>
</tr>
</tbody>
</table>

unnormalized log probabilities

exp → probabilities

normalize

$\text{L}_i = -\log(0.13) = 0.89$
least-squares regression loss

$$(2-0.28)^2 = 2.96$$

matrix multiply + bias offset

$W = \begin{bmatrix} 0.01 & -0.05 & 0.1 & 0.05 \\ 0.7 & 0.2 & 0.05 & 0.16 \\ 0.0 & -0.45 & -0.2 & 0.03 \end{bmatrix}$

$x_i = \begin{bmatrix} -15 \\ 22 \\ -44 \\ 56 \end{bmatrix}$

$b = \begin{bmatrix} 0.0 \\ 0.2 \\ -0.3 \end{bmatrix}$

$y_i = 2$

cross-entropy loss (Softmax)

$\exp(\begin{bmatrix} -2.85 \\ 0.86 \\ 0.28 \end{bmatrix}) = \begin{bmatrix} 0.058 \\ 2.36 \\ 1.32 \end{bmatrix}$

normalize (to sum to one)

$\begin{bmatrix} 0.016 \\ 0.631 \\ 0.353 \end{bmatrix}$

$- \log(0.353) = 0.452$

Credit: Fei-Fei Li & Justin Johnson & Serena Yeung
\[ L(W) = \frac{1}{N} \sum_{i=1}^{N} L_i(f(x_i, W), y_i) \]

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

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

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

**Regularization**: Model should be “simple”, so it works on test data
Data loss: Model predictions should match training data

Regularization: Model should be “simple”, so it works on test data

Occam’s Razor: “Among competing hypotheses, the simplest is the best”
William of Ockham, 1285 - 1347

\[
L(W) = \frac{1}{N} \sum_{i=1}^{N} L_i(f(x_i, W), y_i) + \lambda R(W)
\]
L2 Regularization (Weight Decay)

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

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

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

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

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

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

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

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

**Softmax**

**Full loss**

---

Credit: Fei-Fei Li & Justin Johnson & Serena Yeung
Optimization
Walking man image is CC0 1.0 public domain

Credit: Fei-Fei Li & Justin Johnson & Serena Yeung
Strategy: Follow the slope
Strategy: 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**.
<table>
<thead>
<tr>
<th>current $W:$</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>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>gradient $dW$:</th>
</tr>
</thead>
<tbody>
<tr>
<td>current $W$:</td>
</tr>
<tr>
<td>------------</td>
</tr>
<tr>
<td>[0.34, -1.11, 0.78, 0.12, 0.55, 2.81, -3.1, -1.5, 0.33, ...]</td>
</tr>
<tr>
<td>loss 1.25347</td>
</tr>
</tbody>
</table>
current $W$: 

$$[0.34, \ -1.11, \ 0.78, \ 0.12, \ 0.55, \ 2.81, \ -3.1, \ -1.5, \ 0.33, \ ...]$$

loss 1.25347

$W + h$ (first dim):

$$[0.34 + 0.0001, \ -1.11, \ 0.78, \ 0.12, \ 0.55, \ 2.81, \ -3.1, \ -1.5, \ 0.33, \ ...]$$

loss 1.25322

Gradient $dW$:

$$[-2.5, \ ?, \ ?, \ ?, \ ?]$$

$$(1.25322 - 1.25347)/0.0001 = -2.5$$

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x + h) - f(x)}{h}$$
<table>
<thead>
<tr>
<th>current W:</th>
<th>( W + h ) (second 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 + 0.0001, 0.78, 0.12, 0.55, 2.81, -3.1, -1.5, 0.33, ...]</td>
<td>[-2.5, ?, ?, ?, ?, ?, ?, ?, ?, ...]</td>
</tr>
</tbody>
</table>

loss 1.25347  
loss 1.25353
current $W$:

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

loss 1.25347

$W + h$ (second dim):

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

loss 1.25353

gradient $dW$:

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

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

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

\[
\begin{array}{cccccccc}
0.34, & -1.11, & 0.78, & 0.12, & 0.55, & 2.81, & -3.1, & -1.5, & 0.33, \ldots \\
\end{array}
\]

**W + h (third dim):**

\[
\begin{array}{cccccccc}
0.34, & -1.11, & 0.78 + 0.0001, & 0.12, & 0.55, & 2.81, & -3.1, & -1.5, & 0.33, \ldots \\
\end{array}
\]

**loss 1.25347**

**gradient dW:**

\[
\begin{array}{cccccccc}
-2.5, & 0.6, & 0, & 0, & 0, & 0, & 0, & \ldots \\
\end{array}
\]

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

Credit: Fei-Fei Li & Justin Johnson & Serena Yeung
This is silly. The loss is just a function of $W$:

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

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

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

want $\nabla_W L$
This is silly. The loss is just a function of $W$:

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

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

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

want $\nabla_W L$

Use calculus to compute an analytic 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 $dW$: 

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

dW = ... (some function data and $W$)
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 Descent

```python
# Vanilla Gradient Descent

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

negative gradient direction

Credit: Fei-Fei Li & Justin Johnson & Serena Yeung
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
```
Interactive Web Demo

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

Credit: Fei-Fei Li & Justin Johnson & Serena Yeung
Image features
Image Features
Example: Color Histogram
Example: Histogram of Oriented Gradients (HoG)

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


Credit: Fei-Fei Li & Justin Johnson & Serena Yeung
Example: Bag of Words

Step 1: Build codebook

Extract random patches

Cluster patches to form “codebook” of “visual words”

Step 2: Encode images

Fei-Fei and Perona, “A bayesian hierarchical model for learning natural scene categories”, CVPR 2005
**Image Features: Motivation**

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

Cannot separate red and blue points with linear classifier

After applying feature transform, points can be separated by linear classifier
Image features vs ConvNets

Feature Extraction

Neural networks

(Before) Linear score function: \( f = Wx \)
Neural networks

(Before) Linear score function:

\[ f = Wx \]

(Now) 2-layer Neural Network

\[ f = W_2 \max(0, W_1x) \]
Neural networks

(Before) Linear score function: \( f = Wx \)

(Now) 2-layer Neural Network

\[ f = W_2 \max(0, W_1 x) \]
Neural networks

(Before) Linear score function: \( f = Wx \)

(Now) 2-layer Neural Network \( f = W_2 \max(0, W_1 x) \)
Neural networks

(Before) Linear score function: \( f = Wx \)

(Now) 2-layer Neural Network or 3-layer Neural Network

\[
f = W_2 \max(0, W_1 x)
\]

\[
f = W_3 \max(0, W_2 \max(0, W_1 x))
\]
Next time:

Backpropagation