# BST 261: Data Science II

## Lecture 10

Heather Mattie

Department of Biostatistics
Harvard T.H. Chan School of Public Health
Harvard University

April 18, 2018

# Object Detection

# Object Detection

- Goal: locate and classify objects in an image
- Train CNN on cropped images of objects, where the object takes up most of the space in the image
- Sliding windows detection algorithm
  - Slide a window across your image
  - In each region covered by the window, try to detect object (classify every region as containing an object or not)
  - Very computationally expensive, especially for small window and small stride
  - Bigger windows or strides result in fewer regions and less computational expense, but could hurt performance
  - Won't output the most accurate bounding boxes

# Bounding Box Predictions

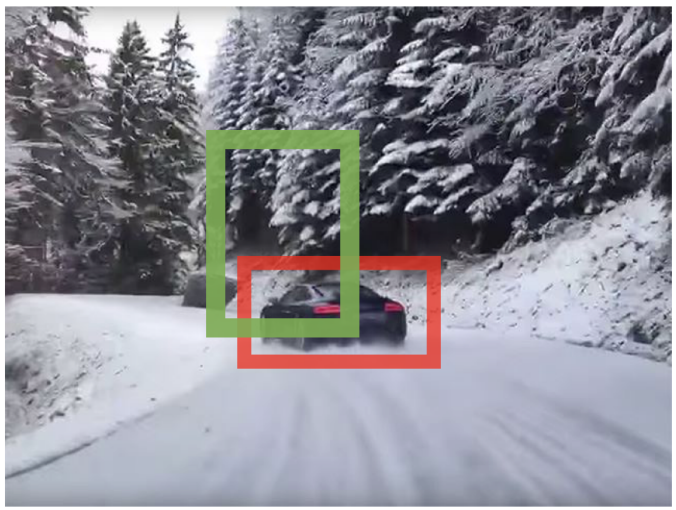
- One way to predict more accurate bounding boxes is by implementing the YOLO (You Only Look Once) algorithm
  - Redmon et al. 2015

    `https://pjreddie.com/media/files/papers/yolo.pdf`
- Split image into grid cells
- Assign the object to the grid cell containing the midpoint of the object
- Works well when there is only 1 object in a particular cell
- Cuts down on computational cost because it can be run as a single convolutional implementation
  - So fast it performs well for real time object detection

# Bounding Box Predictions

# Intersection Over Union

- How well is your algorithm working in terms of finding the bounding boxes?
- One metric to measure the performance of the algorithm is Intersection over Union
- $IoU = \frac{\text{size of intersection}}{\text{size of union}}$
- “Correct" if $IoU \geq 0.5$, or some other threshold
- Basically measures the overlap of the predicted bounding box with the ground truth bounding box



# Non-max Suppression

- Your algorithm may detect the same object multiple times
- Non-max suppression is a way to make sure you detect each object only once

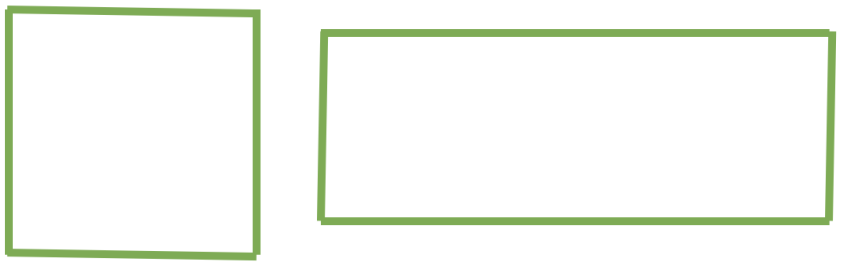
# Non-max Suppression

# Non-max Suppression

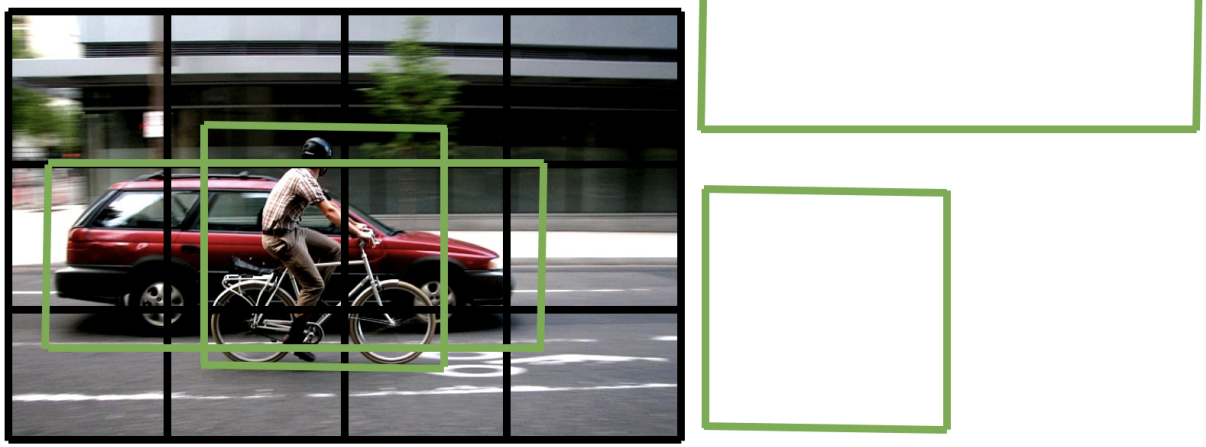
- Discard all boxes with $p_d \leq 0.6$ (or some other threshold)
- While there are remaining boxes:
  - Pick the box with the largest $p_d$ and output that as the prediction
  - Discard any remaining box with $IoU \leq 0.5$ with the box output in the previous step

# Anchor Boxes

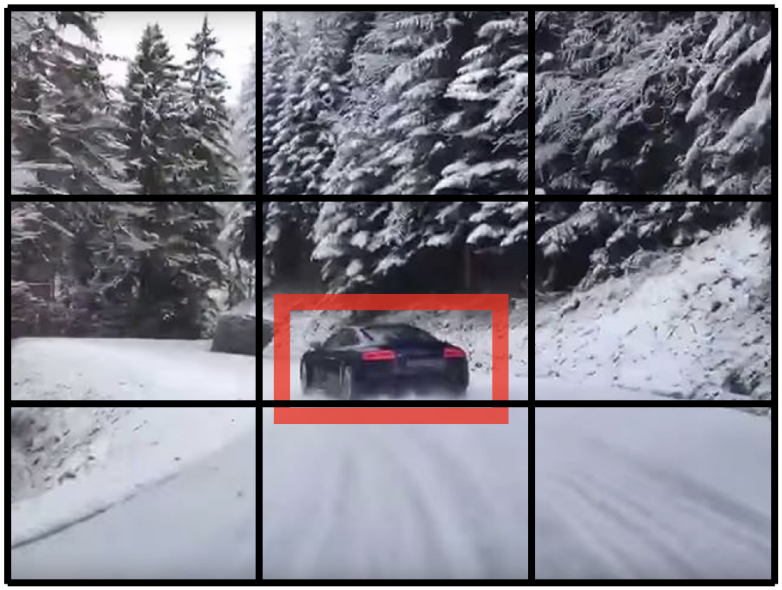
- So far we have assumed a grid cell can only detect one object
- What if you want to detect multiple objects in the same cell?
- Originally, we assigned each object in an image to the grid cell that contained its midpoint
- Now, we will assign an object to a grid cell that contains its midpoint **and** an anchor box for that cell with the highest $IoU$



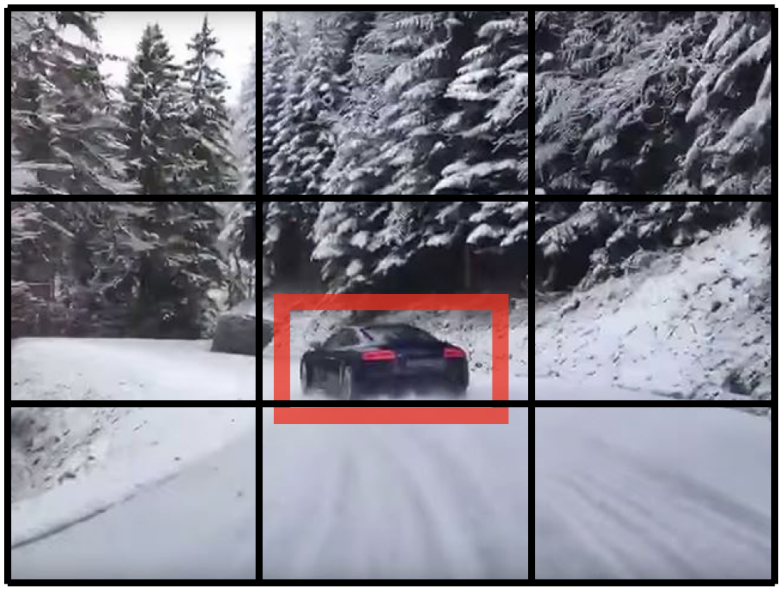
# Anchor Box Algorithm



# YOLO Algorithm



# YOLO Algorithm



# YOLO Algorithm

# Object detection in Keras Tutorial

https://towardsdatascience.com/
object-detection-with-neural-networks-a4e2c46b4491

# Face Recognition

# Terminology

- Recognition
  - Have a database of $K$ persons
  - Get an input image
  - Output ID if the image is any of the $K$ persons, or “not recognized" if not like any of the $K$ persons

- Verification
  - Input image and name/ID
  - Output whether the input image is that of the claimed person

# Face Recognition

- One-shot Learning: learning from 1 example to recognize that person again
- Major downside: needs to be re-trained every time another person is added to group





# Similarity Function

- Similarity function: quantify how similar or different two images are
- $s(\text{img}1, \text{img}2) =$ degree of difference between images
- If $s(\text{img}1, \text{img}2) \leq \tau \rightarrow$ two images are of same person
- If $s(\text{img}1, \text{img}2) > \tau \rightarrow$ two images are of different people





# Siamese Network

- Define the similarity network as

$$s(x^{(1)}, x^{(2)}) = \left\| f(x^{(1)}) - f(x^{(2)}) \right\|_2^2 \tag{1}$$

- Learn parameters so that:
  - If $x^{(i)}, x^{(j)}$ are the same person, $\left\| f(x^{(i)}) - f(x^{(j)}) \right\|_2^2$ is small
  - If $x^{(i)}, x^{(j)}$ are different people, $\left\| f(x^{(i)}) - f(x^{(j)}) \right\|_2^2$ is large

# Learning Objective

Anchor
(A)

Positive
(P)

Anchor
(A)

Negative
(N)

- We want

$$\|f(A) - f(P)\|_2^2 + \alpha \leq \|f(A) - f(N)\|_2^2 \tag{2}$$

- Or equivalently,

$$\|f(A) - f(P)\|_2^2 - \|f(A) - f(N)\|_2^2 + \alpha \leq 0 \tag{3}$$

- Ref: Schroff et al., 2015, FaceNet: A unified embedding for face recognition and clustering

# Loss Function

- Given 3 images A, P, and N:

$$\mathcal{L}(A, P, N) = \max(\|f(A) - f(P)\|_2^2 - \|f(A) - f(N)\|_2^2 + \alpha, 0) \tag{4}$$

$$J = \sum_{i=1}^{m} \mathcal{L}(A^{(i)}, P^{(i)}, N^{(i)}) \tag{5}$$

- Note that multiple pictures of each person are needed for this to be effective

# Choosing A, P, and N

- During training, if A, P, and N are chosen randomly, $s(A, P) + \alpha \leq s(A, N)$ is easily satisfied (it's really easy to randomly pick two very different looking people)
- It's better to choose A, P, and N such that training is more difficult and will be better at recognizing differences on test sets

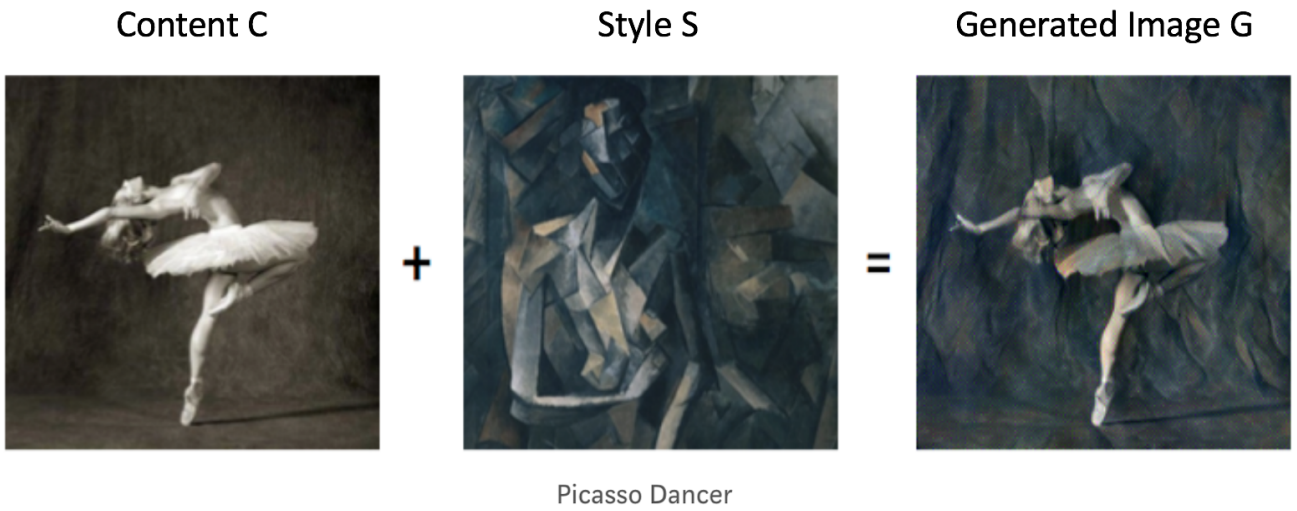
# Neural Style Transfer

# Neural Style Transfer

- Gatys et al., 2015, Image Style Transfer Using Convolutional Neural Networks

# Neural Style Transfer

- Awesome blog: `https://medium.com/artists-and-machine-intelligence/neural-artistic-style-transfer-a-comprehensive-look-f54d8649c199`




$$J(G) = \alpha J_{\text{content}}(C, G) + \beta J_{\text{style}}(S, G) \tag{6}$$

1. Initiate $G$ randomly
2. Use gradient descent to minimize $J(G)$

$$G := G - \frac{\partial}{\partial G} J(G) \tag{7}$$

- Let layer $l$ compute content cost
- Let $a^{[l](C)}$ and $a^{[l](G)}$ be the activation of layer $l$ on the images
- If $a^{[l](C)}$ and $a^{[l](G)}$ are similar, both images have similar content

$$J_{\text{content}}(C, G) = \frac{1}{2} \left\| a^{[l](C)} - a^{[l](G)} \right\| \tag{8}$$

# Style Matrix

- Let $a^{[l]}_{i,j,k} =$ activation at $(i, j, k)$
- $G^{[l]}$ is $n^{[l]}_C \times n^{[l]}_C$

$$G^{[l](S)}_{kk'} = \sum_{i=1}^{n^{[l]}_H} \sum_{j=1}^{n^{[l]}_W} a^{[l](S)}_{i,j,k} a^{[l](S)}_{i,j,k'} \tag{9}$$

$$G^{[l](G)}_{kk'} = \sum_{i=1}^{n^{[l]}_H} \sum_{j=1}^{n^{[l]}_W} a^{[l](G)}_{i,j,k} a^{[l](G)}_{i,j,k'} \tag{10}$$

# Style Cost Function

$$J^{[l]}_{\text{style}}(S, G) = \frac{1}{(2n^{[l]}_H n^{[l]}_W n^{[l]}_C)^2} \sum_k \sum_{k'} (G^{[l](S)}_{kk'} - G^{[l](G)}_{kk'}) \tag{11}$$

- Sometimes this is written as

$$J_{\text{style}}(S, G) = \sum_l \lambda^{[l]} J^{[l]}_{\text{style}}(S, G) \tag{12}$$

# Recurrent Neural Networks

# Recurrent Neural Networks

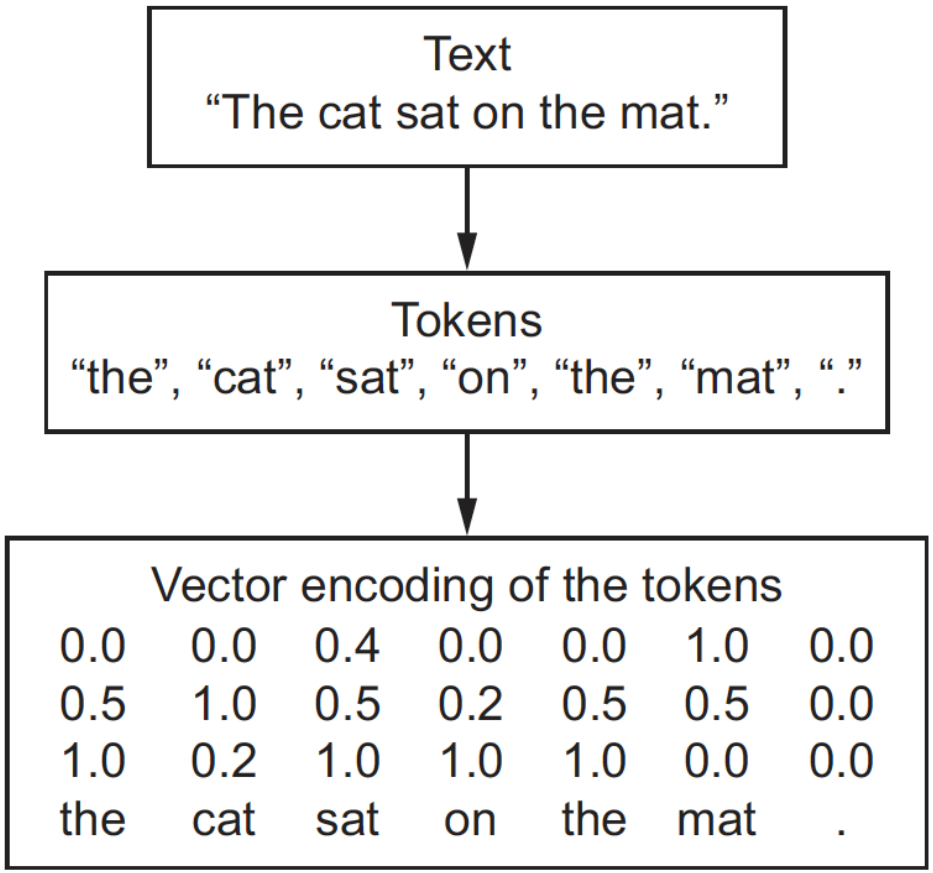
- We will now learn about deep learning models that can process text, timeseries, and sequence data in general
- The two algorithms we will cover are **recurrent neural networks** and **1D CNNs**
- Applications:
  - Document and timeseries classification e.g. identifying the topic of an article or the author of a book
  - Timeseries comparisons e.g. estimating how closely related two documents are
  - Sentiment analysis
  - Timeseries forecasting e.g. predicting weather (something that needs major improvement for Boston...)
  - Sequence-to-sequence learning e.g. decoding an English sentence into Turkish

# Text Data

- Text data can be understood as either a sequence of characters or a sequence of words
  - Most common to work at the level of words
- Like all other neural networks, we can't simply input raw text - we must **vectorize** the text: transform it into numeric tensors
- We can do this in multiple ways:
  - Segment text into words, and transform each word into a vector
  - Segment text into characters and transform each character into a vector
  - Extract n-grams (overlapping groups of multiple consecutive words or characters) of words or characters, and transform each n-gram into a vector
- The different units into which you break down text (words, characters, n-grams) are called **tokens**, and the action of breaking text into tokens is **tokenization**
- There are multiple ways to associate a vector with a token
  - One-hot encoding
  - Token embedding (or word embedding)

# One-hot Encoding

- Most common and most basic way to turn a token into a vector
- We used this with the IMDB and Reuters data sets
- First, associate a unique integer index with every word
- Then, turn the integer index $i$ into a binary vector of size $N$ (the size of the vocabulary, or number of words in the set)
- The vector is all 0s except for the $i$th entry, which is 1



# One-hot Encoding

```
from keras.preprocessing.text import Tokenizer

samples = ['The cat sat on the mat.', 'The dog ate my homework.']

# We create a tokenizer, configured to only take
# into account the top-1000 most common words
tokenizer = Tokenizer(num_words=1000)
# This builds the word index
tokenizer.fit_on_texts(samples)

# This turns strings into lists of integer indices.
sequences = tokenizer.texts_to_sequences(samples)

# You could also directly get the one-hot binary representations.
# Note that other vectorization modes than one-hot encoding are supported!
one_hot_results = tokenizer.texts_to_matrix(samples, mode='binary')

# This is how you can recover the word index that was computed
word_index = tokenizer.word_index
print('Found %s unique tokens.' % len(word_index))
```

# One-hot Hashing

- A variant of one-hot encoding is the **one-hot hashing trick**
- Useful when the number of unique tokens is too large to handle explicitly
- Instead of explicitly assigning an index to each word and keeping a reference of these indices in a dictionary, you can hash words into vectors of fixed size
- Main advantage: saves memory and allows generation of tokens before all of the data has been seen
- Main drawback: **hash collisions**
  - Two different words end up with the same hash
  - The likelihood of this decreases when the dimensionality of the hashing space is much larger than the total number of unique tokens being hashed

# One-hot Hashing

```
samples = ['The cat sat on the mat.', 'The dog ate my homework.']

# We will store our words as vectors of size 1000.
# Note that if you have close to 1000 words (or more)
# you will start seeing many hash collisions, which
# will decrease the accuracy of this encoding method.
dimensionality = 1000
max_length = 10

results = np.zeros((len(samples), max_length, dimensionality))
for i, sample in enumerate(samples):
    for j, word in list(enumerate(sample.split()))[:max_length]:
        # Hash the word into a "random" integer index
        # that is between 0 and 1000
        index = abs(hash(word)) % dimensionality
        results[i, j, index] = 1.
```