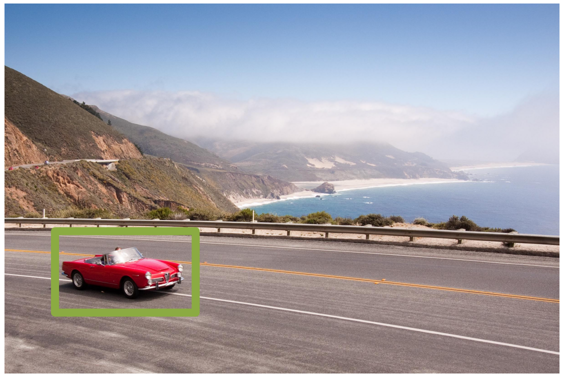
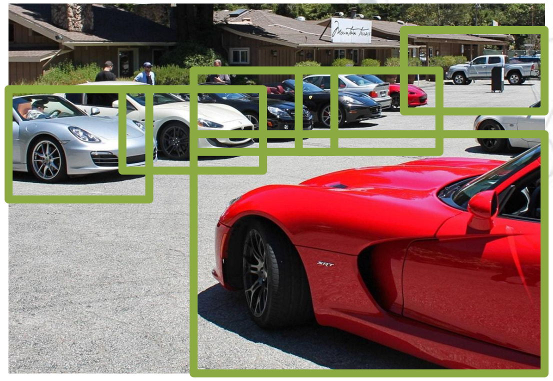
BST 261: Data Science II
Lecture 13

Advanced Topics
Object Localization and Detection, Face Recognition, Advanced Network Architectures

Heather Mattie
Harvard T.H. Chan School of Public Health
Spring 2 2019

Object Detection and Location

Localization and Detection

Car

(Classification)

Car, but where?

(Classification with localization)

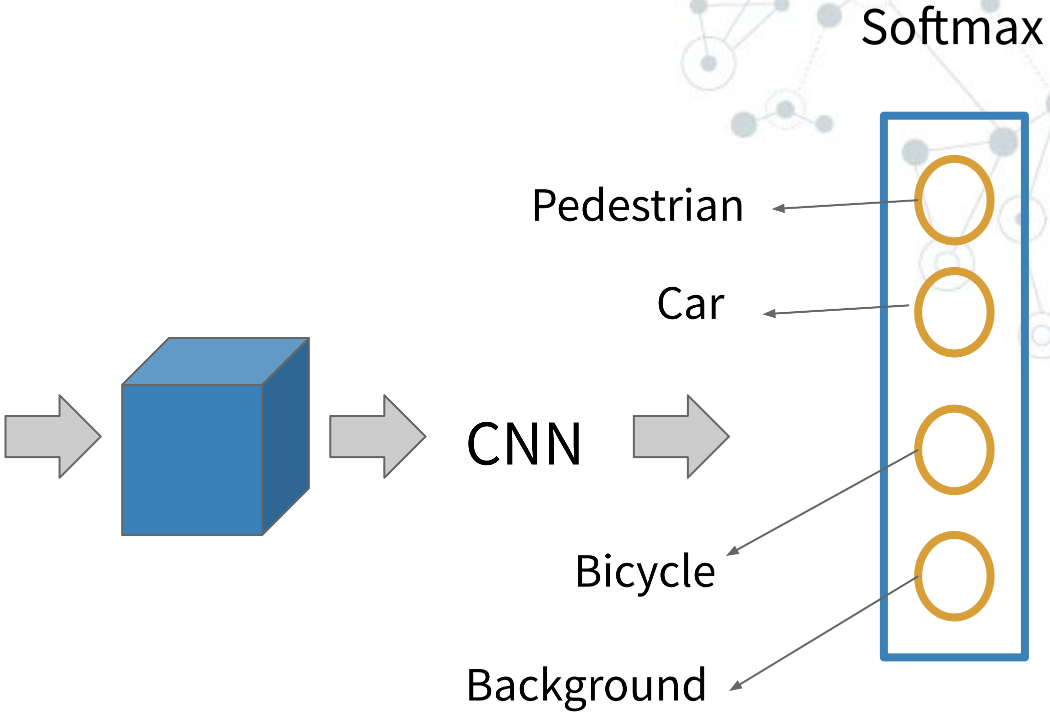
Multiple cars

(Detection)

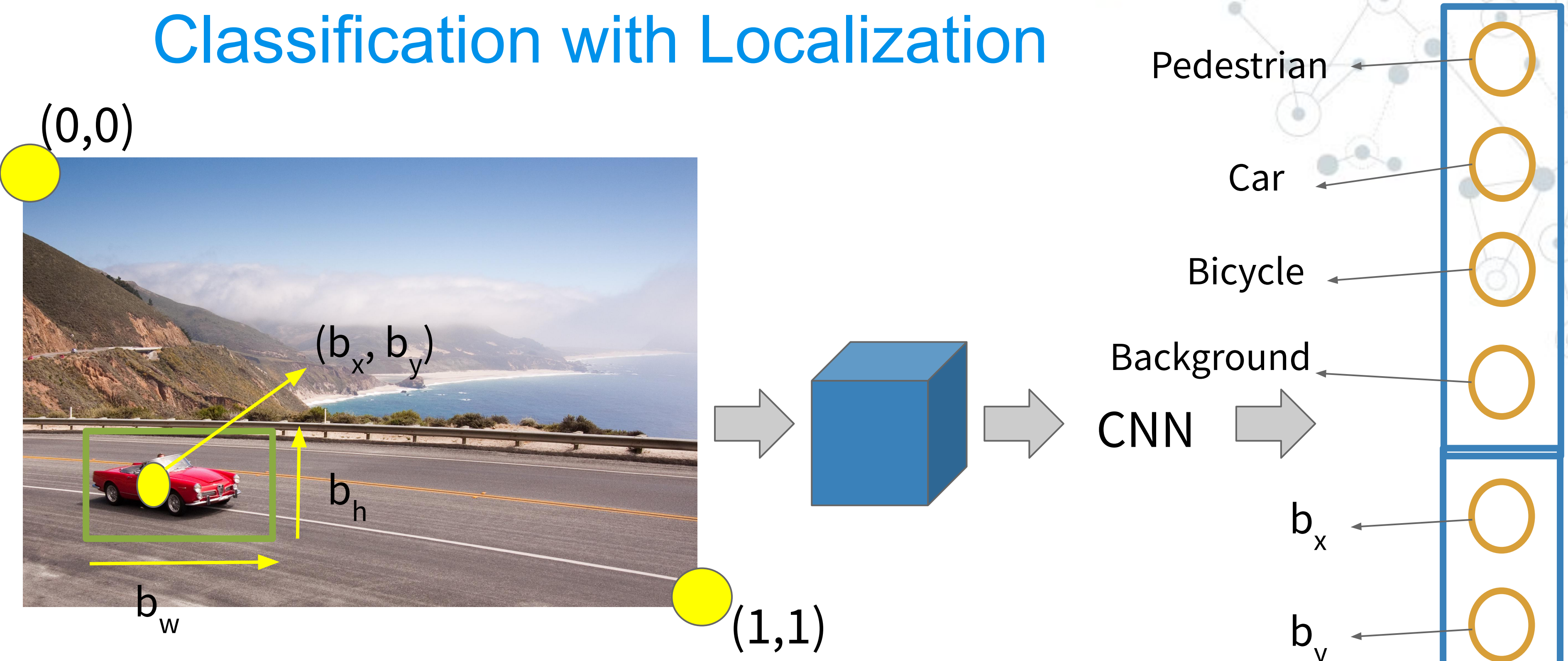
1 object

Multiple objects; could be from different classes

Classification



Classification with Localization



- To train a network to detect and locate objects,
  we need a lot of training data with bounding box labels:
  - (b_x, b_y): the x and y coordinates of the center of the object
  - b_w: the width of the bounding box
  - b_h: the height of the bounding box
  - New image label: [class, b_x, b_y, b_w, b_h]

Classification with Localization

Classes

Pedestrian (c_1)

Car (c_2)

Bicycle (c_3)

Background (no object)

Note: assuming there is only 1 object in image

Probability there is an object in the image (and not just background)

New y label: [p_d, b_x, b_y, b_w, b_h, c_1, c_2, c_3]

Loss:

$$L(\hat{y}, y) = (\hat{y}_1, y_1)^2 + (\hat{y}_2, y_2)^2 \ldots (\hat{y}_8, y_8)^2 \quad \text{if } p_d = 1$$

$$L(\hat{y}, y) = (\hat{y}_1, y_1)^2 \quad \text{if } p_d = 0$$

Localization and Detection

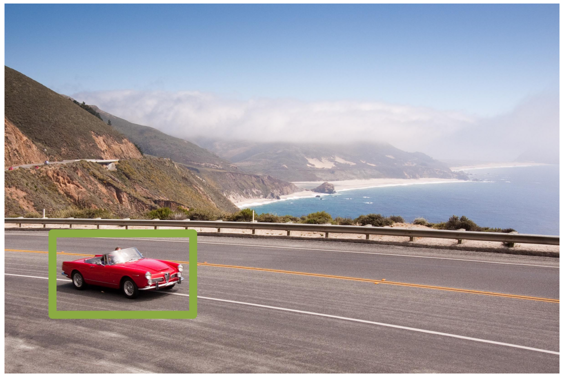
<u>Classes</u>

Pedestrian (c_1)

Car (c_2)

Bicycle (c_3)

Background (no object)



$y = [1, 0.25, 0.75, 0.15, 0.15, 0, 1, 0]$

New y label: $[p_d, b_x, b_y, b_w, b_h, c_1, c_2, c_3]$

Loss:

$L(\hat{y}, y) = (\hat{y}_1, y_1)^2 + (\hat{y}_2, y_2)^2 \ldots (\hat{y}_8, y_8)^2$ if $p_d = 1$

$L(\hat{y}, y) = (\hat{y}_1, y_1)^2$ if $p_d = 0$

Localization and Detection

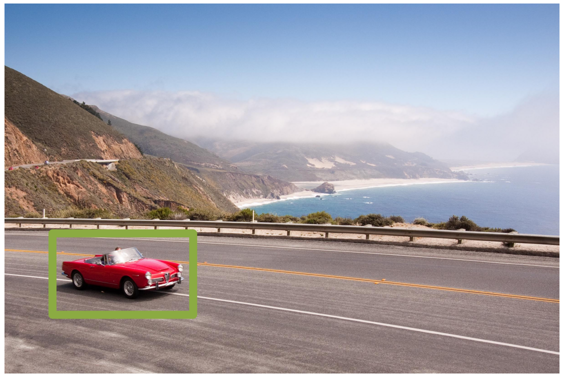
<u>Classes</u>

Pedestrian (c_1)

Car (c_2)

Bicycle (c_3)

Background (no object)



y = [1, 0.25, 0.25, 0.25, 0.25, 0, 1, 0]

New y label: [p_d, b_x, b_y, b_w, b_h, c_1, c_2, c_3]

Loss:

$$L(\hat{y}, y) = (\hat{y}_1, y_1)^2 + (\hat{y}_2, y_2)^2 \ldots (\hat{y}_8, y_8)^2 \quad \text{if } p_d = 1$$

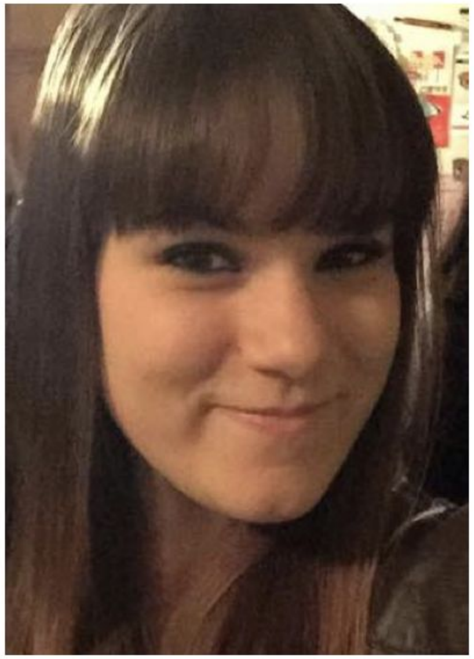
$$L(\hat{y}, y) = (\hat{y}_1, y_1)^2 \quad \text{if } p_d = 0$$

y = [0, ?, ?, ?, ?, ?, ?, ?]

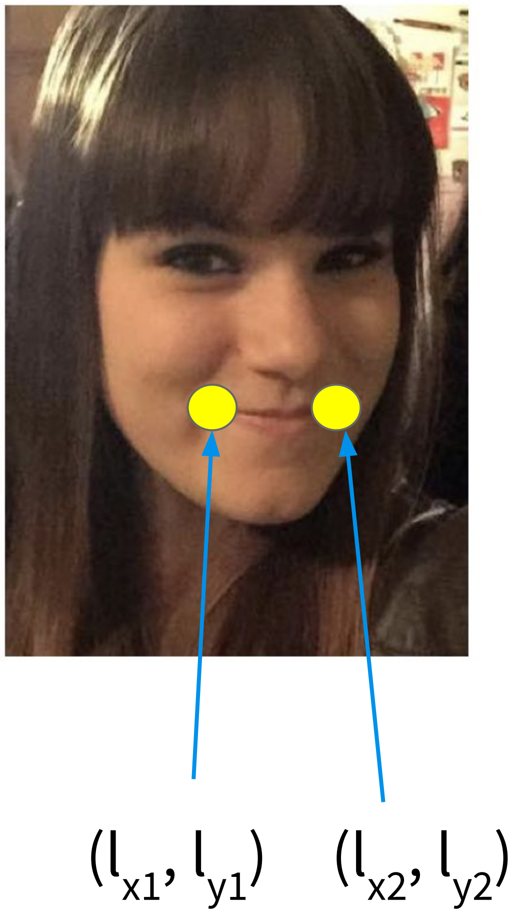
The ?s are essentially ignored

Landmark Detection

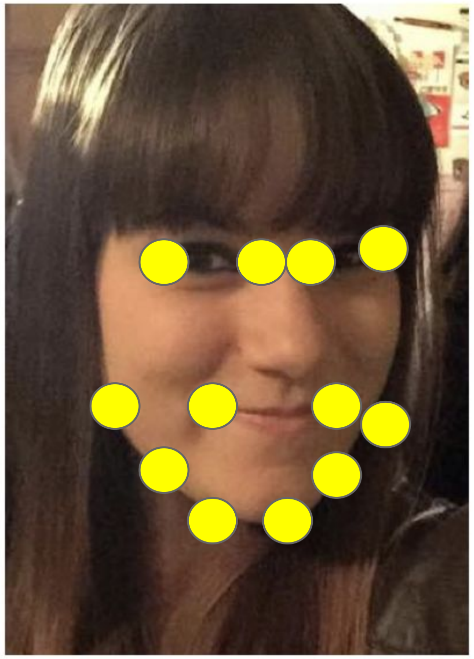
Landmark Detection



Landmark Detection



Landmark Detection



Label every point

Input: image with n landmarks

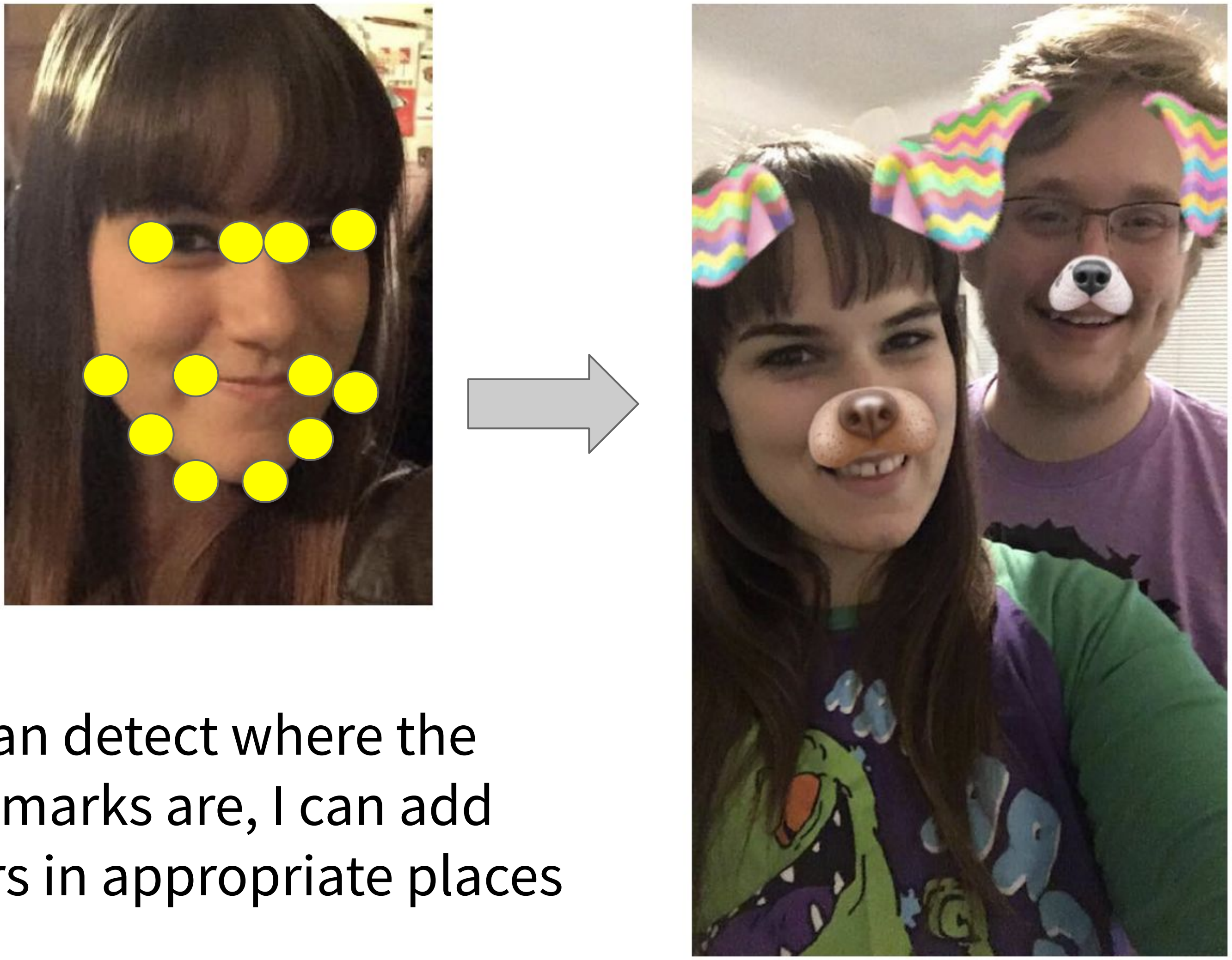
Output:

$[p_{face}, l_{x1}, l_{y1}, l_{x2}, l_{y2}, \ldots, l_{xn}, l_{yn}]$

Fit CNN to output if the image is of a face and the locations of the landmarks if it is a face

Note: landmarks have to be consistent across all training images, i.e. landmark # 1 is the left corner of the right eye

Landmark Detection

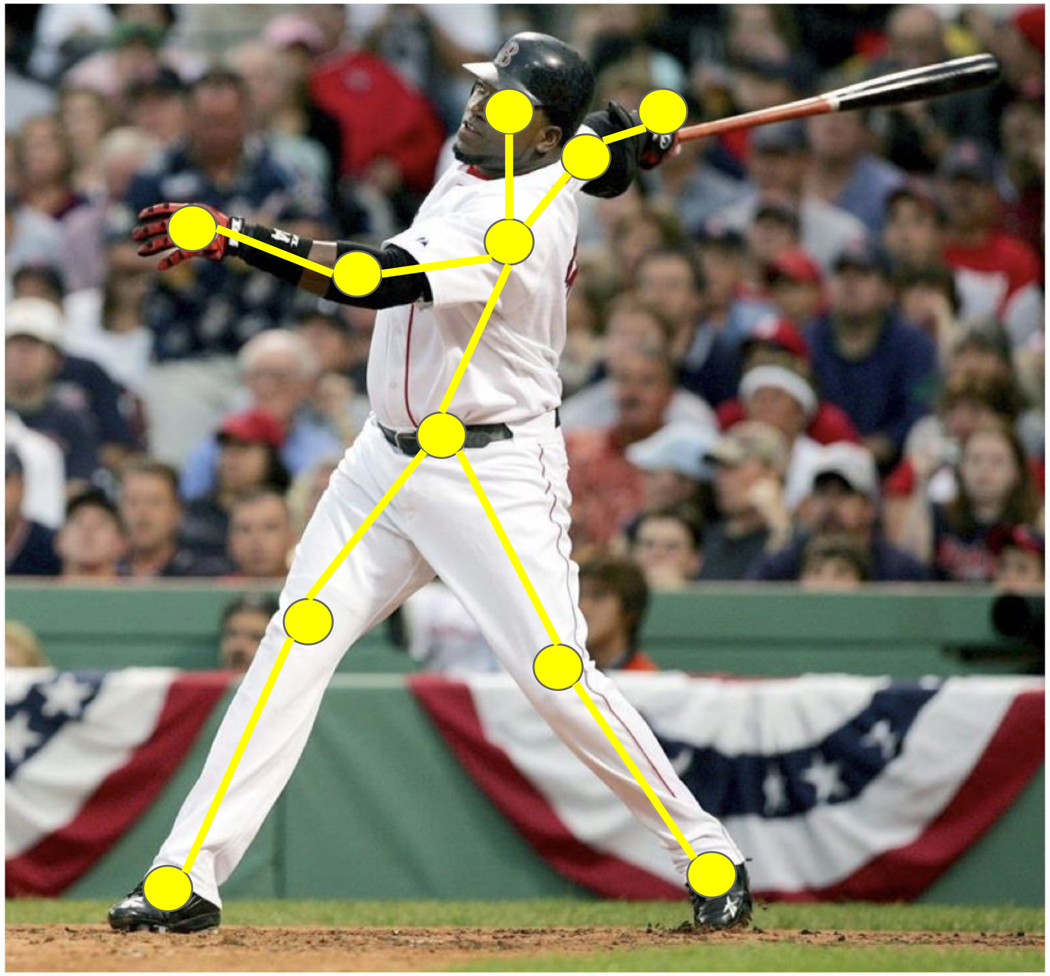


If I can detect where the landmarks are, I can add filters in appropriate places

Landmark Detection

Landmark Detection

"Pose" detection



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

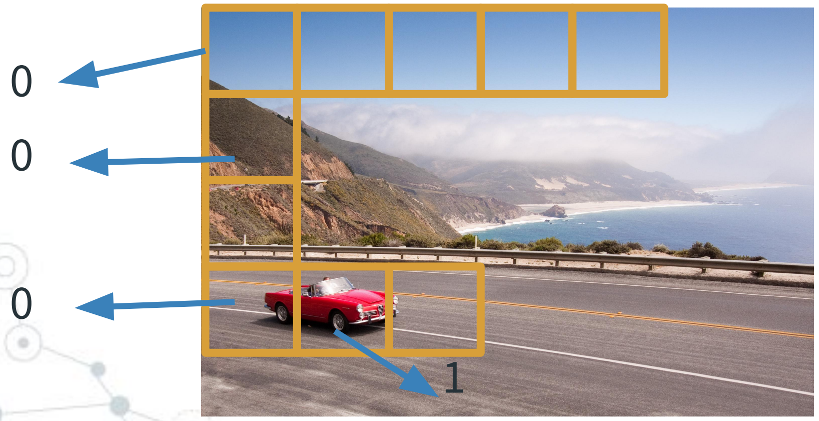
X: image of a car
y: 1

X: image of a car
y: 1

X: image of not a car
y: 0

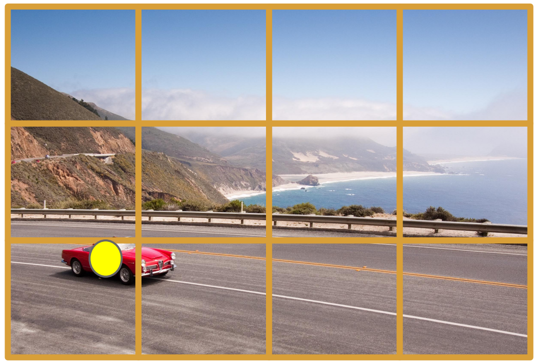
Object Detection

- 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
  - Overly dramatic YOLO video
- 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
- GitHub repo with easy implementation in Keras

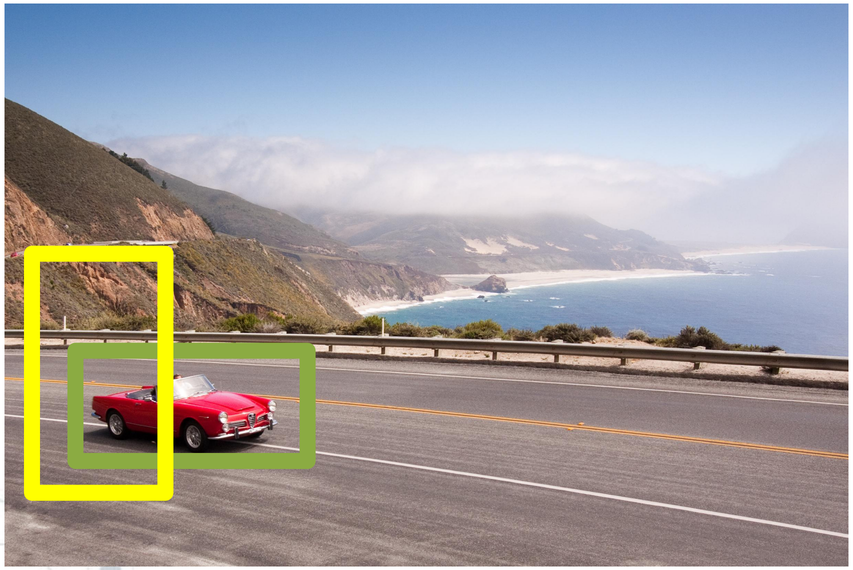


Evaluating Object Localization

- 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

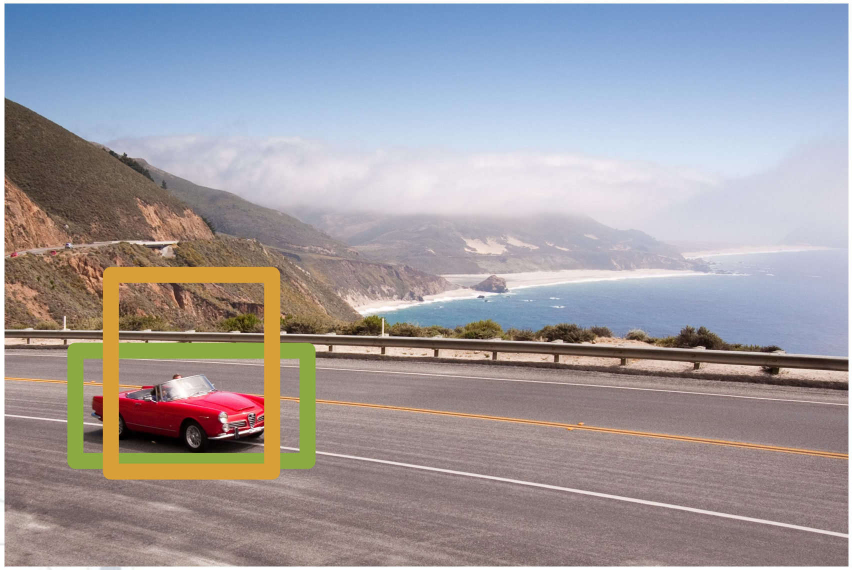


Evaluating Object Localization

- 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

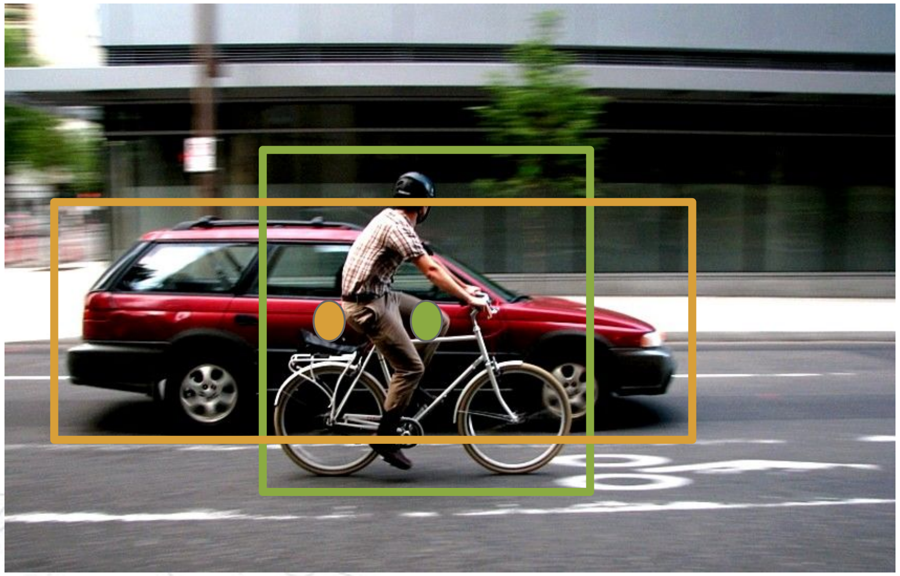


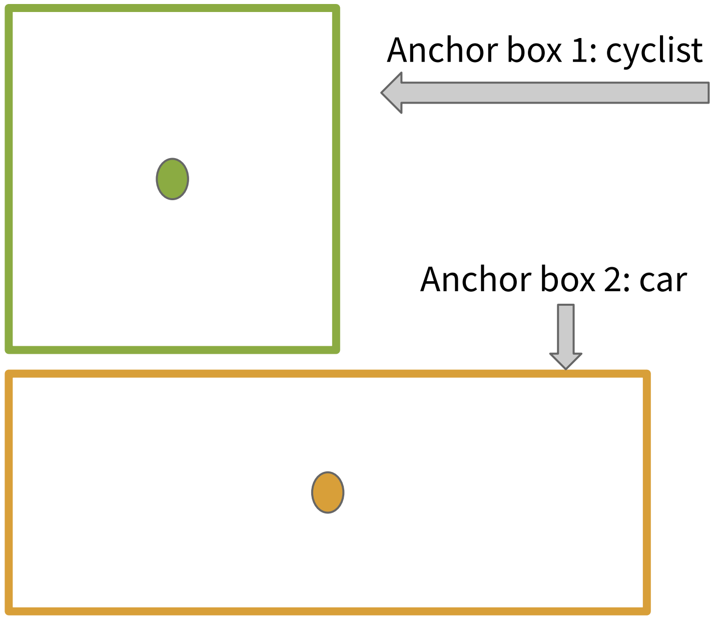
Non-max suppression

- Your algorithm may detect the same object multiple times
- Non-max suppression is a way to make sure you detect each object only once
- Discard all bounding boxes with $p_d \leq 0.6$ (or some other threshold)
- While there are remaining boxes:
  - Pick the bounding box with the largest p_d and output that as the prediction
  - Discard any remaining bounding box with IoU ≥ 0.5 with the box output in the previous step
- Repeat this process independently for each type of object you are trying to detect

Anchor Boxes

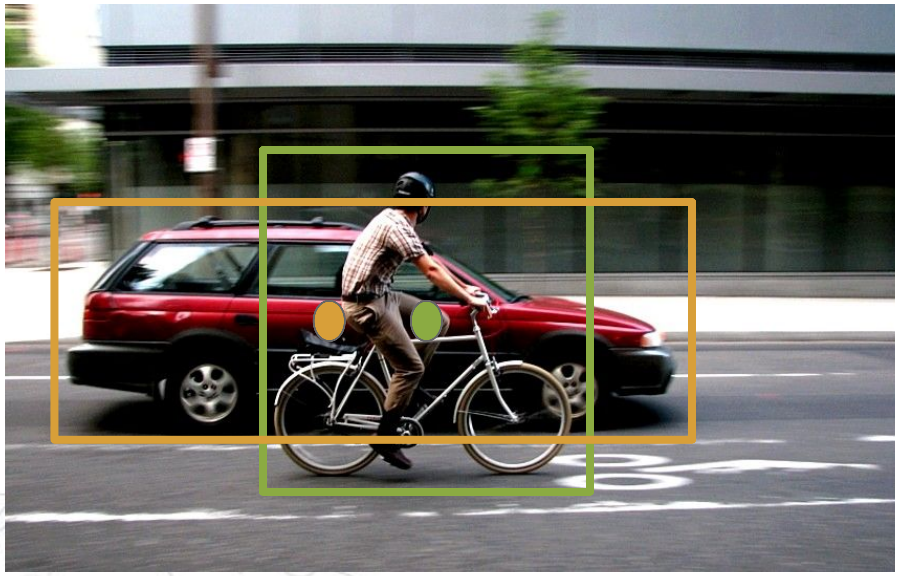
- So far we have assumed a grid cell can only detect one object
- What if you want to **detect multiple objects in the same cell**?
- Originally, we assigned each object in an image to the grid cell that contained its midpoint
- Now, we will assign an object to a grid cell that contains its midpoint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

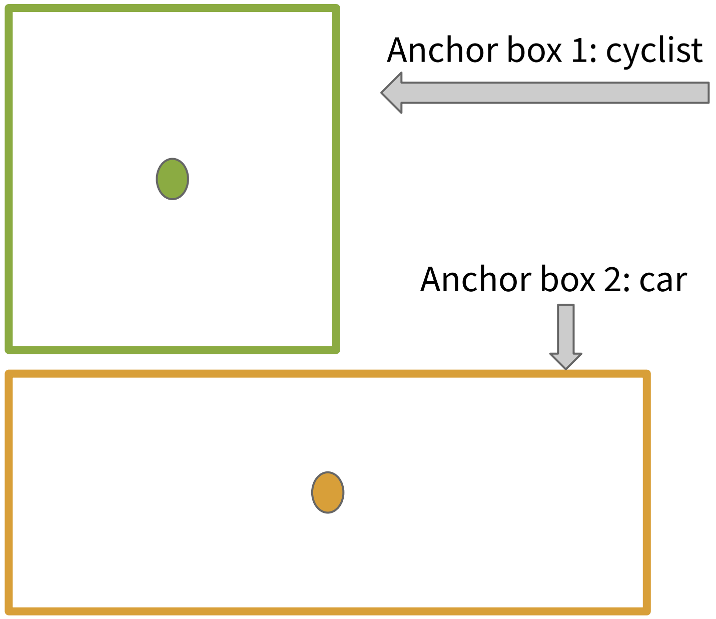




Anchor Boxes

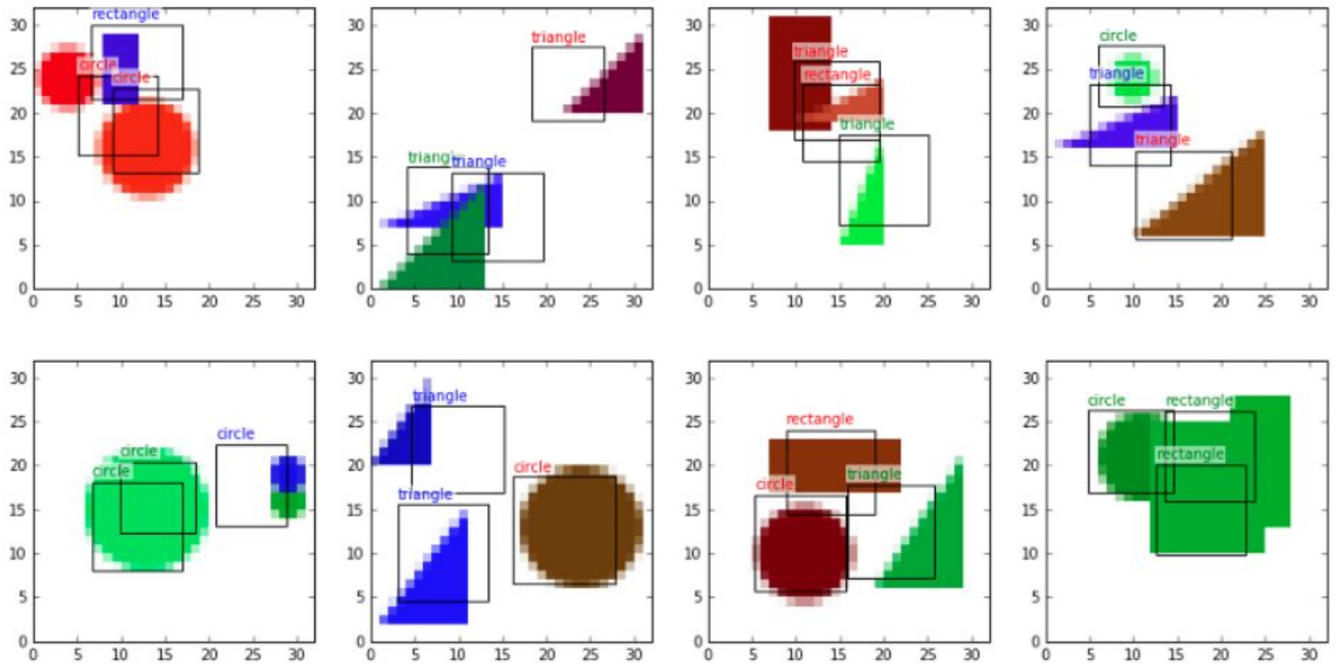
$$y = [\underbrace{p_d, b_x, b_y, b_w, b_h, c_1, c_2, c_3}_{\text{Anchor box 1}}, \underbrace{p_d, b_x, b_y, b_w, b_h, c_1, c_2, c_3}_{\text{Anchor box 2}}]$$





Object Detection Tutorial

- Object detection with neural networks - a simple tutorial using Keras



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

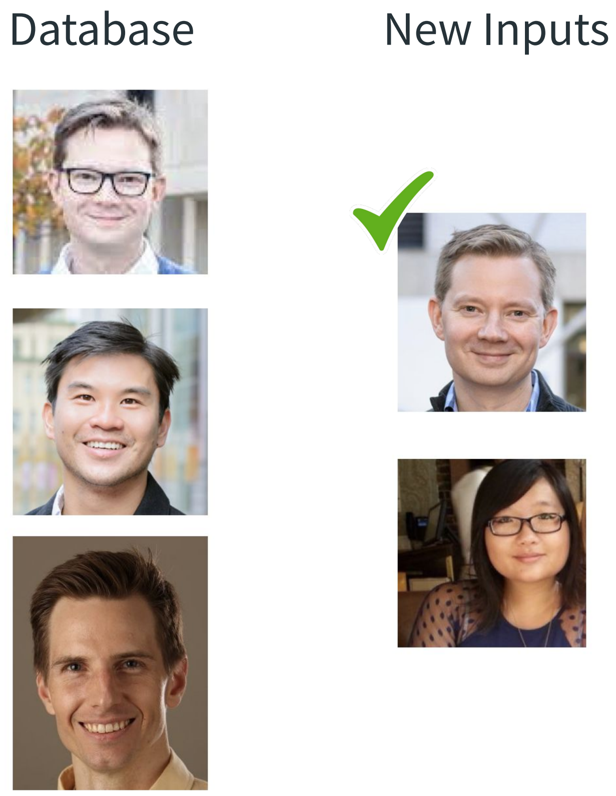
- One-shot Learning: learning from 1 example to recognize that person again
- Major downside: needs to be re-trained every time another person is added to group

Database

New Inputs

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



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

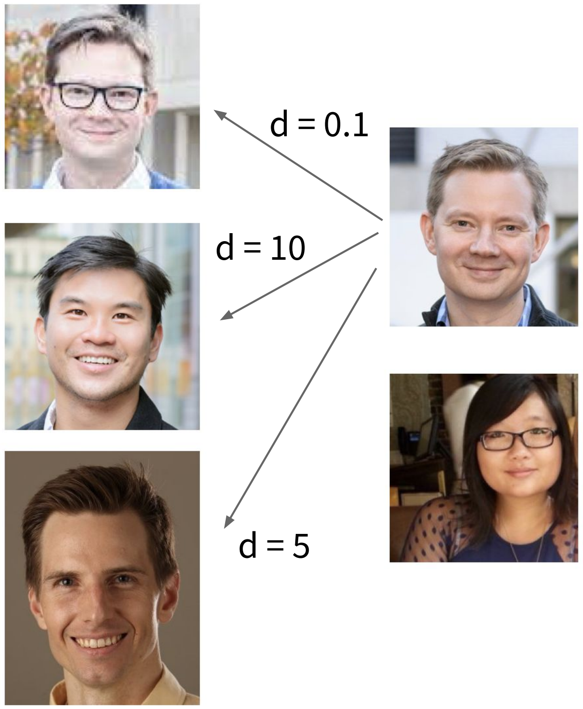
- Similarity function: quantify how similar or different two images are
- If difference is large, the images are of two different people
- If difference is small, the images are of the same person
- DeepFace by Taigman et al 2014

- Define the similarity network as

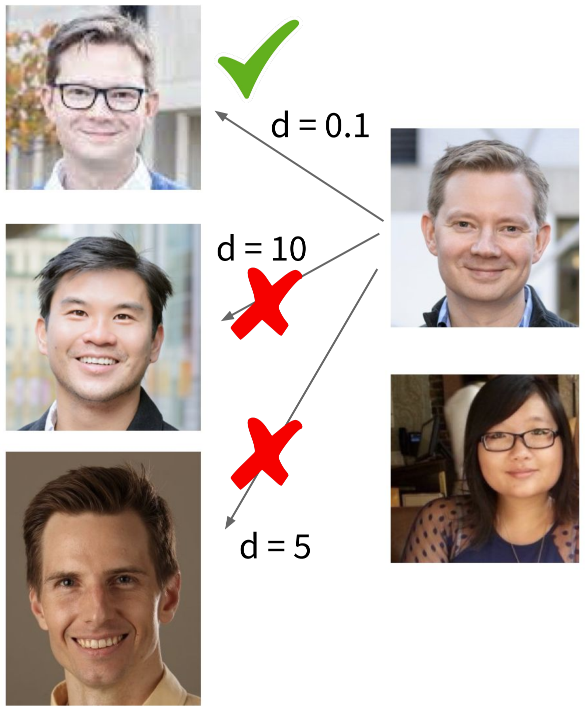
$$d(x^{(1)}, x^{(2)}) = \|f(x^{(1)}) - f(x^{(2)})\|_2^2$$

- Learn parameters such that if $x^{(i)}$ and $x^{(j)}$ are the same person, d is small, and if $x^{(i)}$ and $x^{(j)}$ are different people, d is large

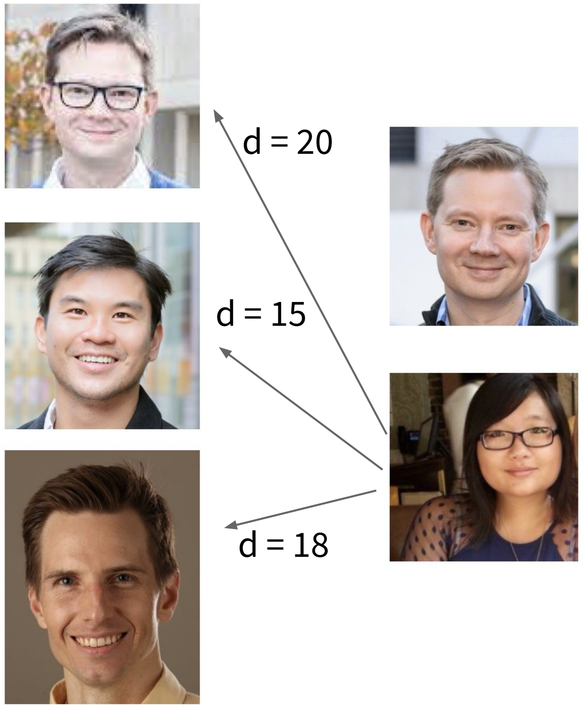
Similarity Function



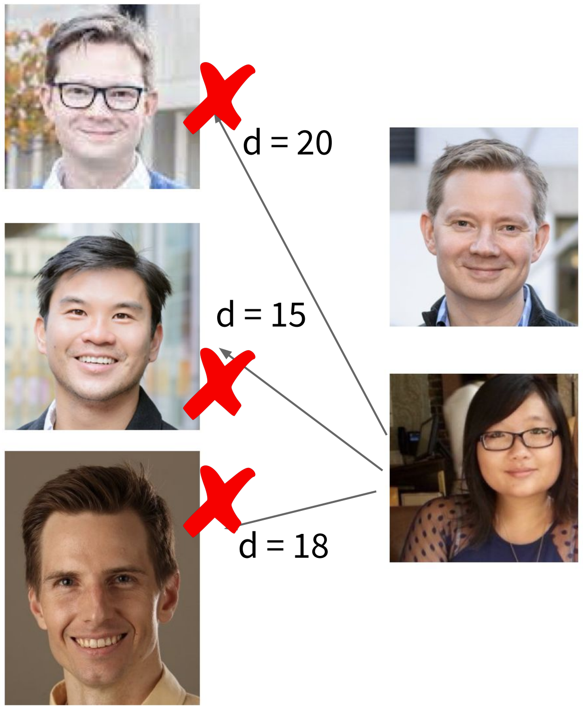
Similarity Function



Similarity Function



Similarity Function



Not in database

Triplet Loss

- To learn the parameters of your network, can use gradient descent to minimize the triplet loss
- Given 3 images A (anchor), P (positive) and N (negative), can we minimize the “triplet loss”:

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

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

Anchor
(A)

Positive
(P)

Anchor
(A)

Negative
(N)

Margin parameter - ensures the network doesn't just label every difference as 0

FaceNet

- During training, if A, P, and N are chosen randomly, $d(A,P) + \alpha \leq d(A,N)$ 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

Anchor
(A)

Positive
(P)

Anchor
(A)

Negative
(N)

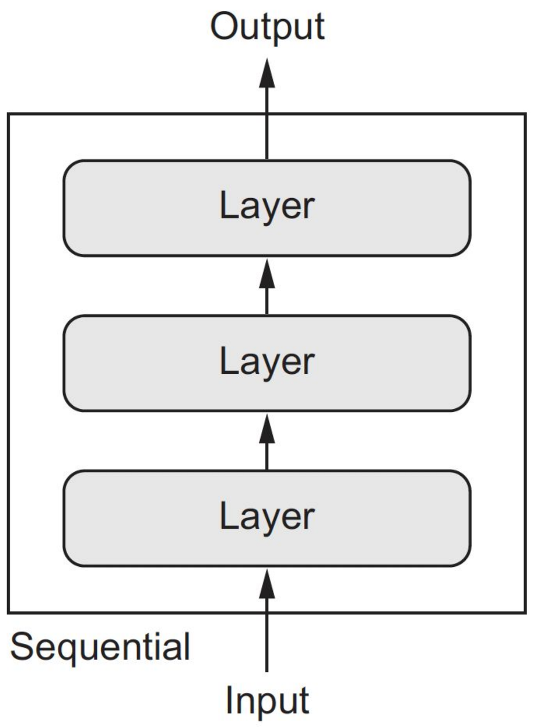
Advanced Architectures

Beyond the Sequential Model

- Throughout the course we have assumed each network has exactly one input and exactly one output, and that it consists of a linear stack of layers

- But what if we have **multiple types** of inputs? Or multiple types of outputs?

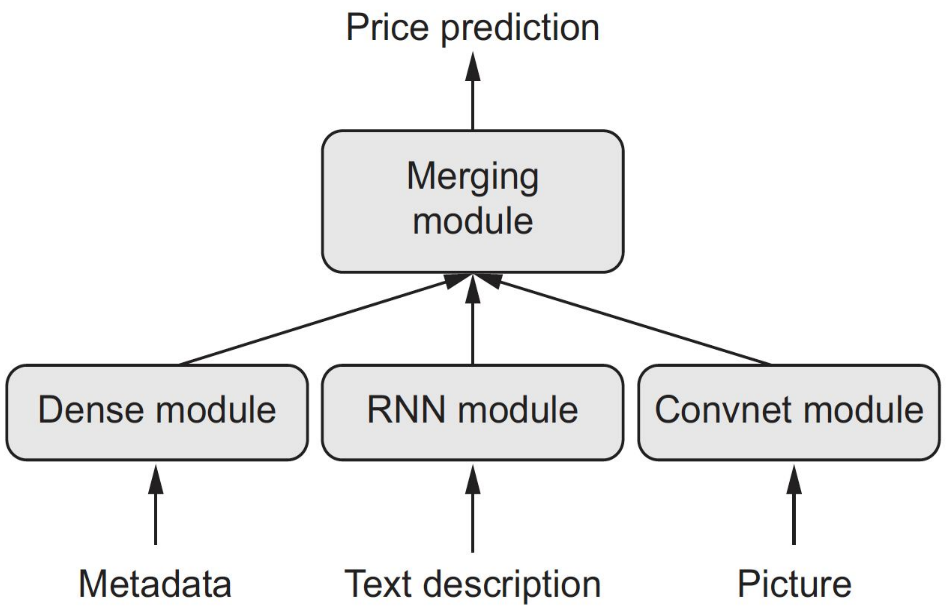
- We can change the network structure - Keras makes this easy to do



Multimodal (Multi-inputs) Model

- Multimodal inputs merge data coming from different input sources, processing each type of data using different kinds of neural layers

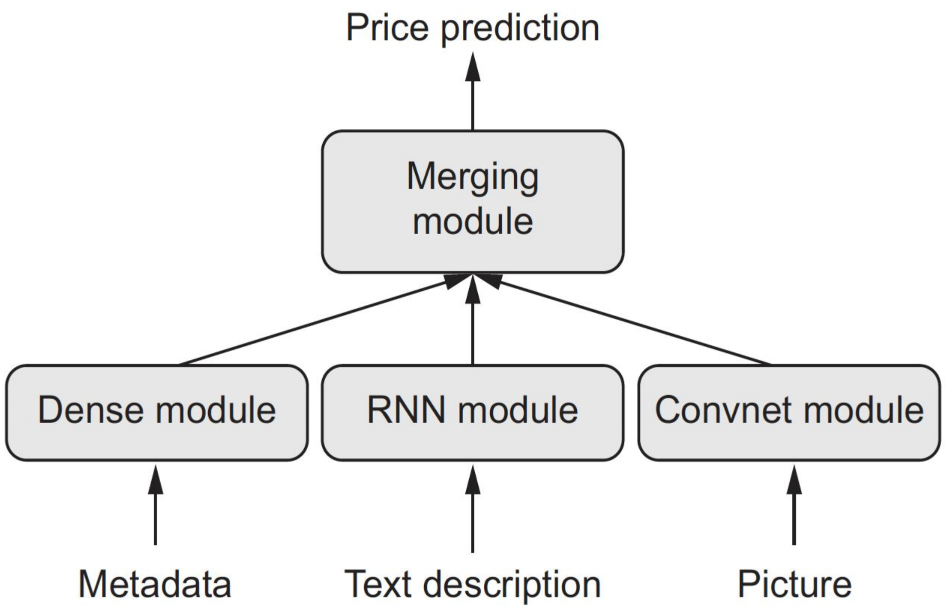
- Example: predict the most likely market price of a second-hand piece of clothing, using the following inputs:
  - User-provided **metadata** (brand, age, etc.)
  - User-provided **text** description
  - **Picture** of the item



Multimodal (Multi-inputs) Model

- Suboptimal approach: **train three separate models** and then do a weighted average of their predictions
  - Information may be redundant

- Better approach: **jointly learn** a more accurate model of the data by using a model that can see all available input modalities simultaneously: a model with three input branches



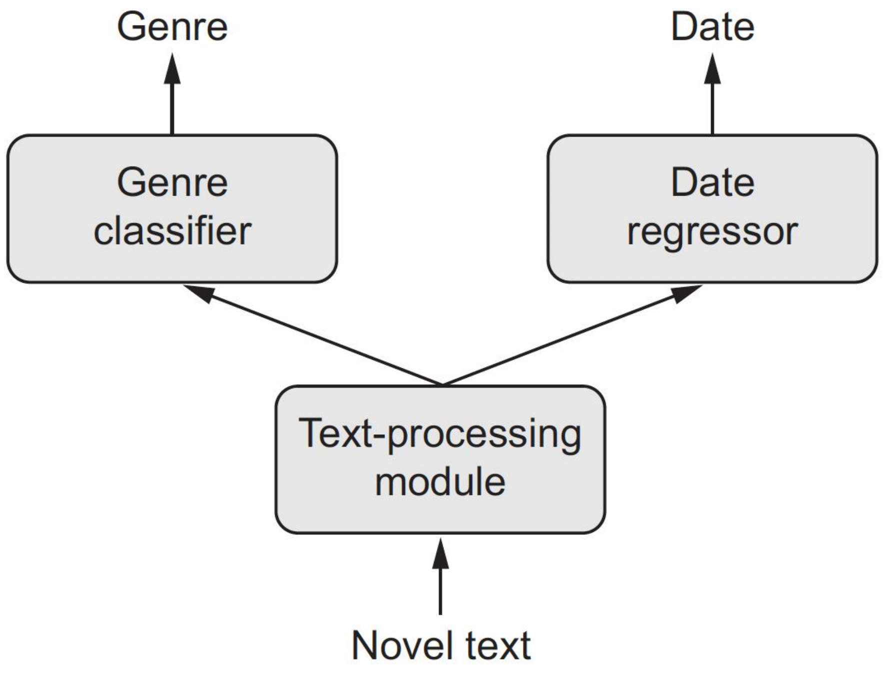
Multi-output (multihead) Model

- Predict multiple target attributes (outputs) of input data

- Example: predict the genre and date of a novel using the novel's text

- Suboptimal approach: **train two separate models**: one for the genre and one for the date
  - But these attributes aren't statistically independent

- Better approach: **jointly predict** both genre and date at the same time
  - Correlations between date and genre of the novel helps training



Directed Acyclic Graphs (DAGs)...

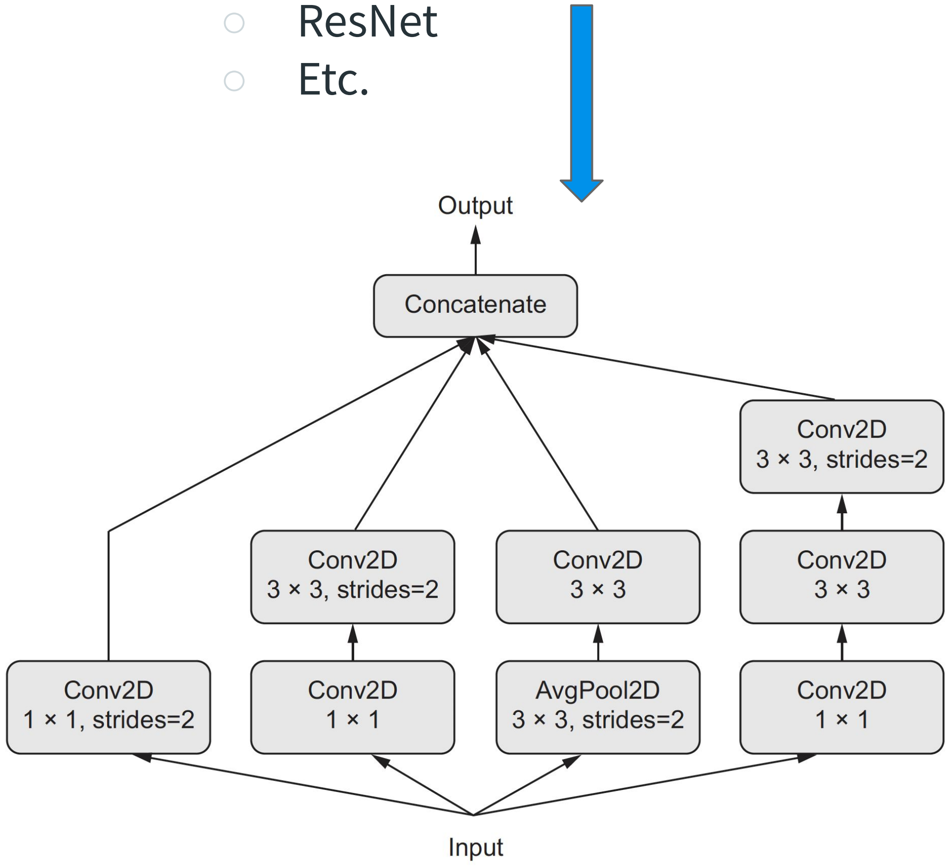
… not THOSE DAGs.

- Nonlinear network architectures
- Examples
  - Inception models
  - ResNet
  - Etc.

Directed Acyclic Graphs (DAGs)...

… not THOSE DAGs.

- Nonlinear network architectures
- Examples
  - Inception models
  - ResNet
  - Etc.

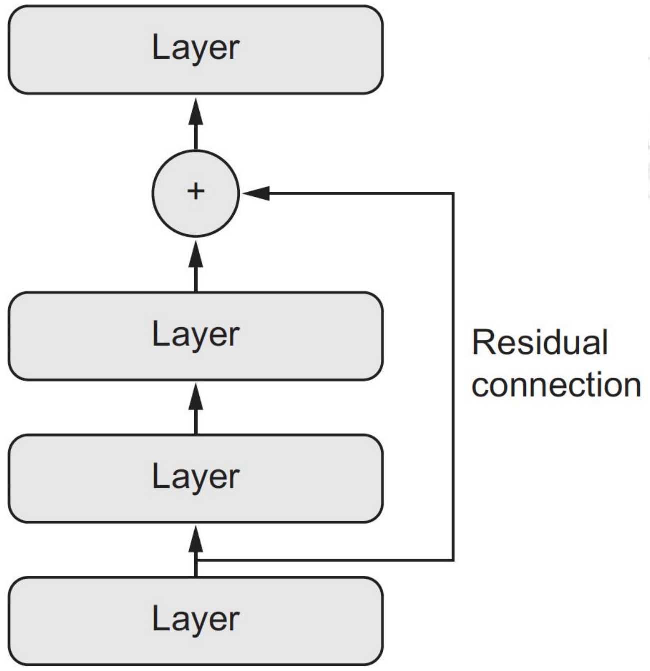


The input is processed by several convolutional branches whose outputs are merged back into a single tensor

Directed Acyclic Graphs (DAGs)...

… not THOSE DAGs.

- Nonlinear network architectures
- Examples
  - Inception models
  - ResNet
  - Etc.

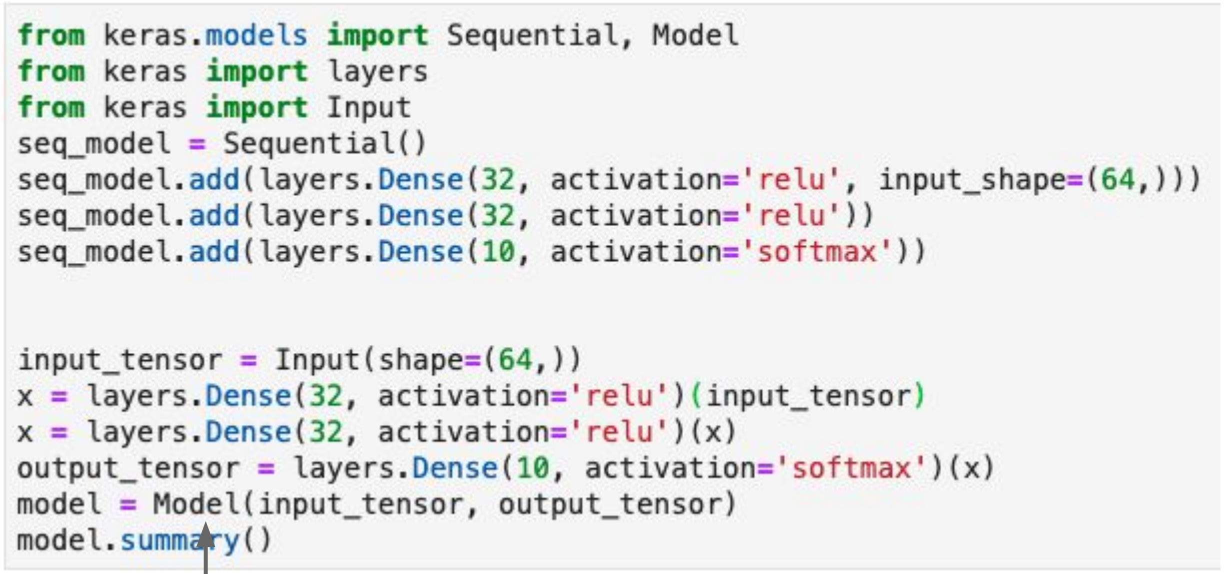


A residual connection consists of reinjecting previous representations into the downstream flow of data by adding a past output tensor to a later output tensor.

This helps prevent information loss along the data-processing flow.

The Functional API in Keras

- Directly manipulate tensors
- Use layers as **functions** that take tensors and return tensors



```python
from keras.models import Sequential, Model
from keras import layers
from keras import Input
seq_model = Sequential()
seq_model.add(layers.Dense(32, activation='relu', input_shape=(64,)))
seq_model.add(layers.Dense(32, activation='relu'))
seq_model.add(layers.Dense(10, activation='softmax'))


input_tensor = Input(shape=(64,))
x = layers.Dense(32, activation='relu')(input_tensor)
x = layers.Dense(32, activation='relu')(x)
output_tensor = layers.Dense(10, activation='softmax')(x)
model = Model(input_tensor, output_tensor)
model.summary()
```

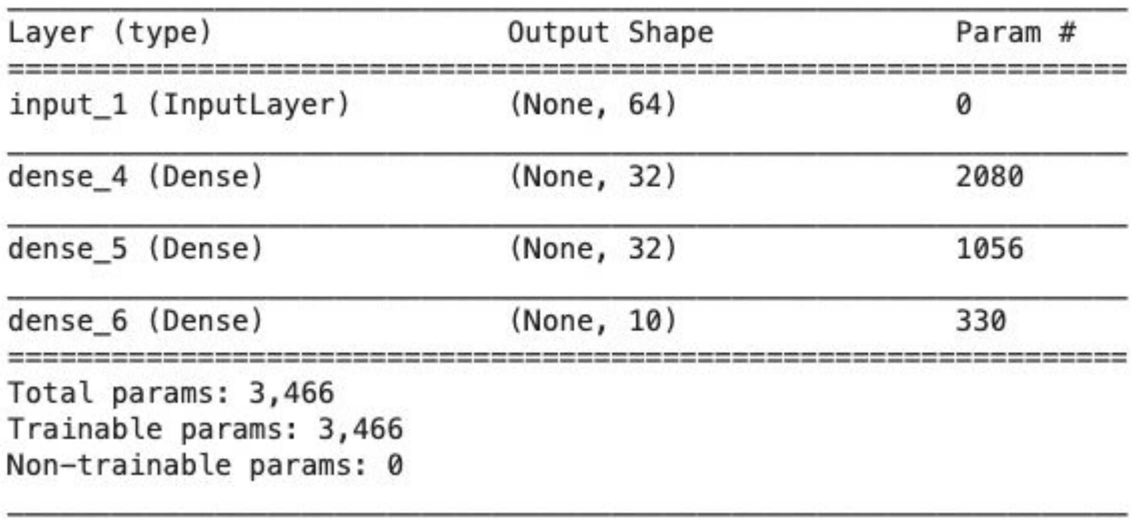
Sequential model (what we've seen before)

Functional equivalent to the above model

The Model class turns an input tensor and output tensor into a model

The Functional API in Keras

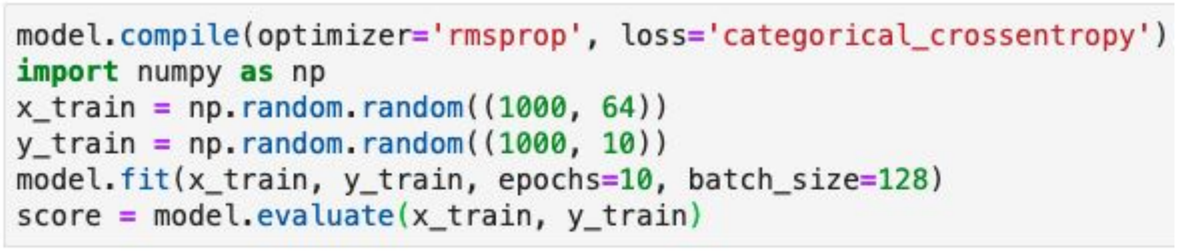
Keras retrieves every layer involved in going from the input tensor to the output tensor and brings them together into a graph-like structure (a Model)



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 64)                0
_________________________________________________________________
dense_4 (Dense)              (None, 32)                2080
_________________________________________________________________
dense_5 (Dense)              (None, 32)                1056
_________________________________________________________________
dense_6 (Dense)              (None, 10)                330
=================================================================
Total params: 3,466
Trainable params: 3,466
Non-trainable params: 0
_________________________________________________________________
```

The Functional API in Keras

Everything is the same when you compile, train and evaluate an instance of Model:



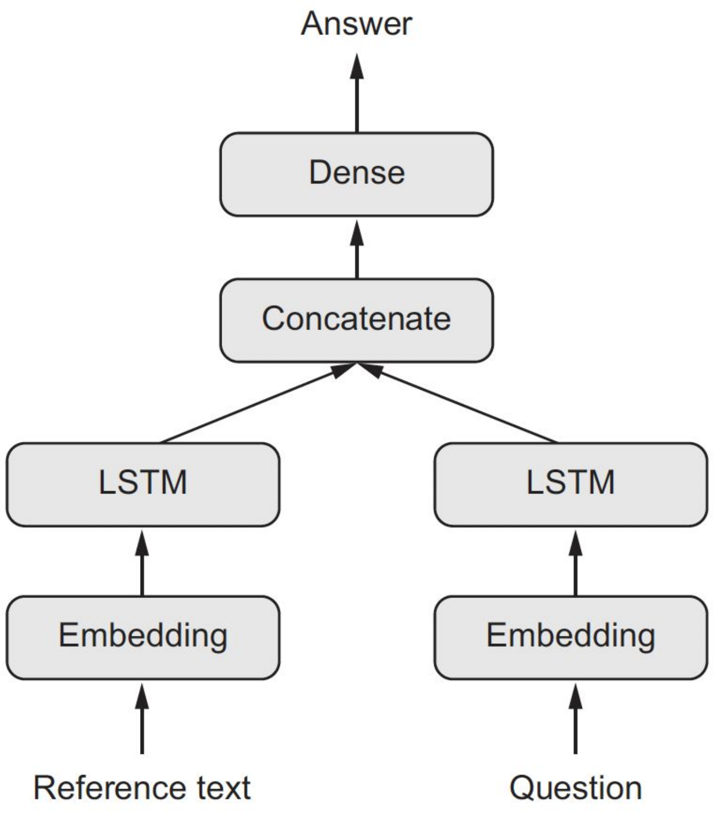
```
model.compile(optimizer='rmsprop', loss='categorical_crossentropy')
import numpy as np
x_train = np.random.random((1000, 64))
y_train = np.random.random((1000, 10))
model.fit(x_train, y_train, epochs=10, batch_size=128)
score = model.evaluate(x_train, y_train)
```

Multi-input models

- The functional API can be used to build models that have multiple inputs.

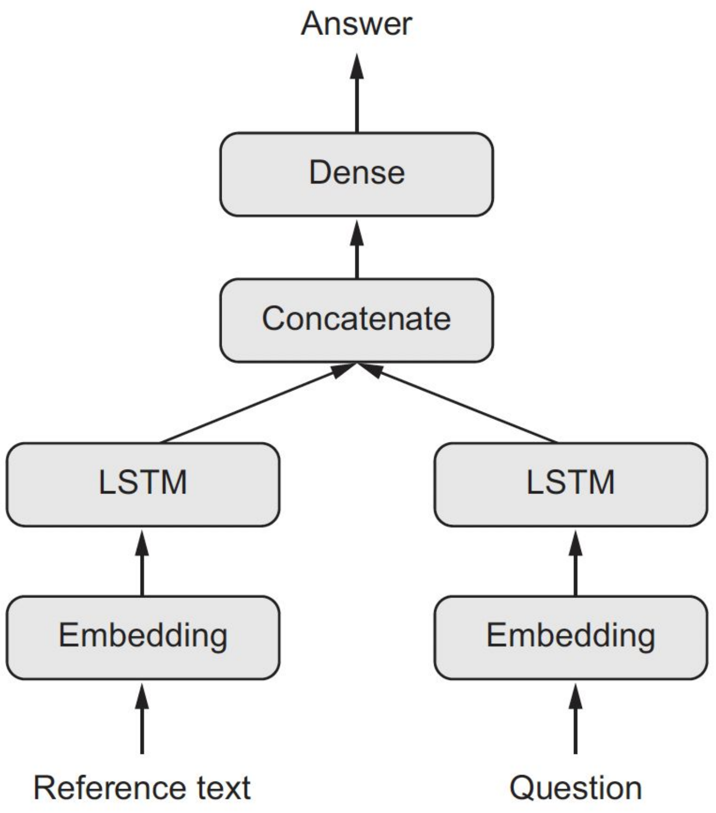
- Typically, such models at some point merge their different input branches using a layer that can combine several tensors: by adding them, concatenating them, and so on.

- This is usually done via a Keras merge operation such as keras.layers.add, keras.layers.concatenate, etc.

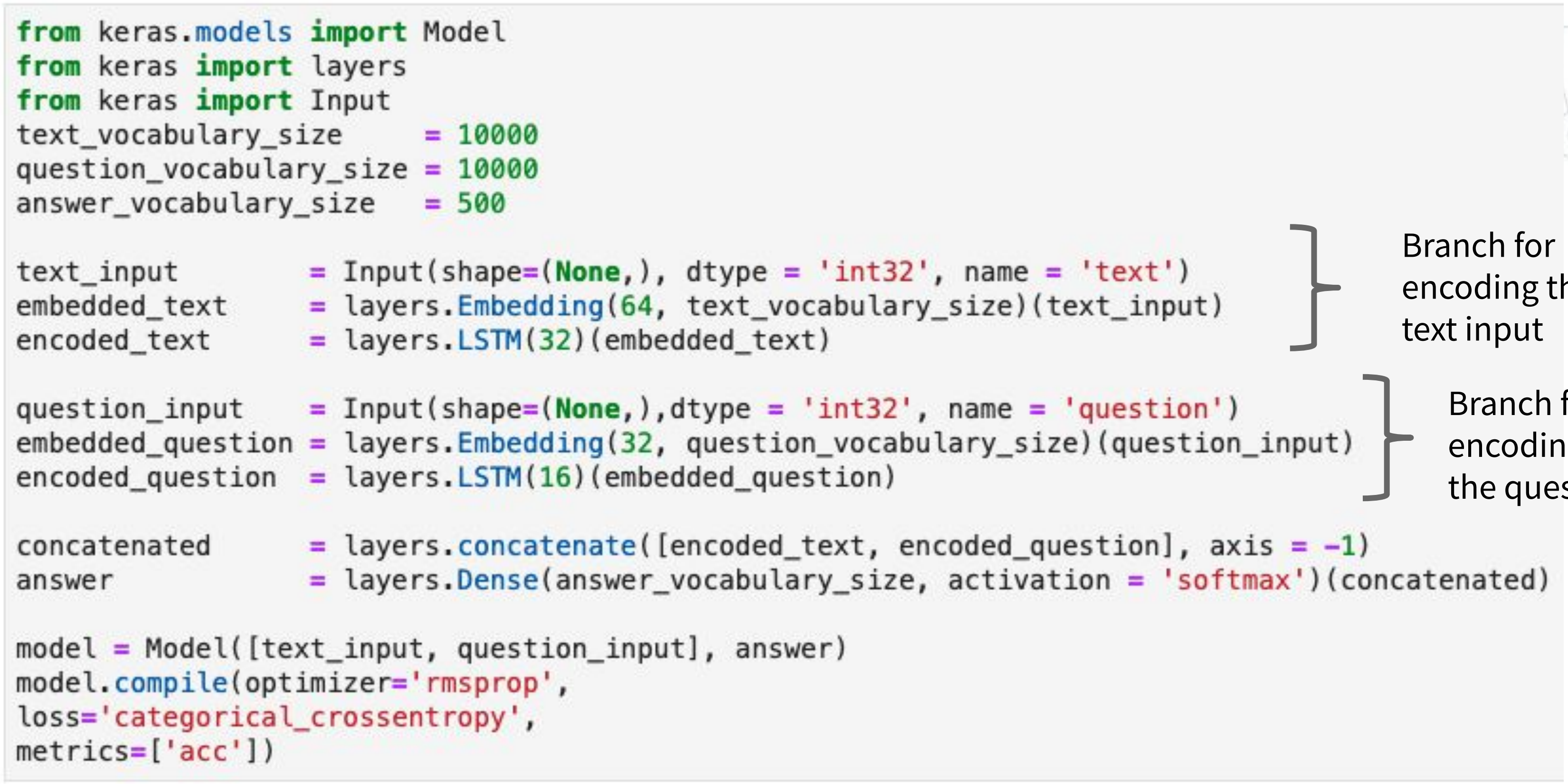


Multi-input models

- Example: question-answering model
- A typical question-answering model has two inputs: a natural-language question and a text snippet (such as a news article) providing information to be used for answering the question.
- The model must then produce an answer: in the simplest possible setup, this is a one-word answer obtained via a softmax over some predefined vocabulary



Multi-input models



```
from keras.models import Model
from keras import layers
from keras import Input
text_vocabulary_size     = 10000
question_vocabulary_size = 10000
answer_vocabulary_size   = 500

text_input         = Input(shape=(None,), dtype = 'int32', name = 'text')
embedded_text      = layers.Embedding(64, text_vocabulary_size)(text_input)
encoded_text       = layers.LSTM(32)(embedded_text)

question_input     = Input(shape=(None,),dtype = 'int32', name = 'question')
embedded_question  = layers.Embedding(32, question_vocabulary_size)(question_input)
encoded_question   = layers.LSTM(16)(embedded_question)

concatenated       = layers.concatenate([encoded_text, encoded_question], axis = -1)
answer             = layers.Dense(answer_vocabulary_size, activation = 'softmax')(concatenated)

model = Model([text_input, question_input], answer)
model.compile(optimizer='rmsprop',
loss='categorical_crossentropy',
metrics=['acc'])
```

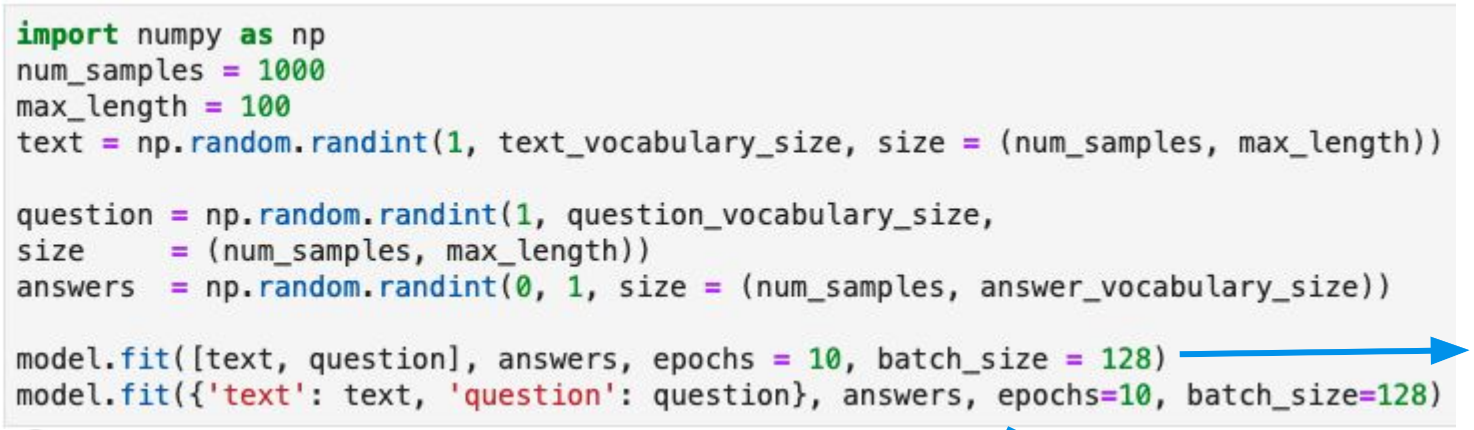
Branch for encoding the text input

Branch for encoding the question

Multi-input models

How do you train this two-input model?
There are two possible APIs:

1. You can feed the model a list of Numpy arrays as inputs, or
2. you can feed it a dictionary that maps input names to Numpy arrays.



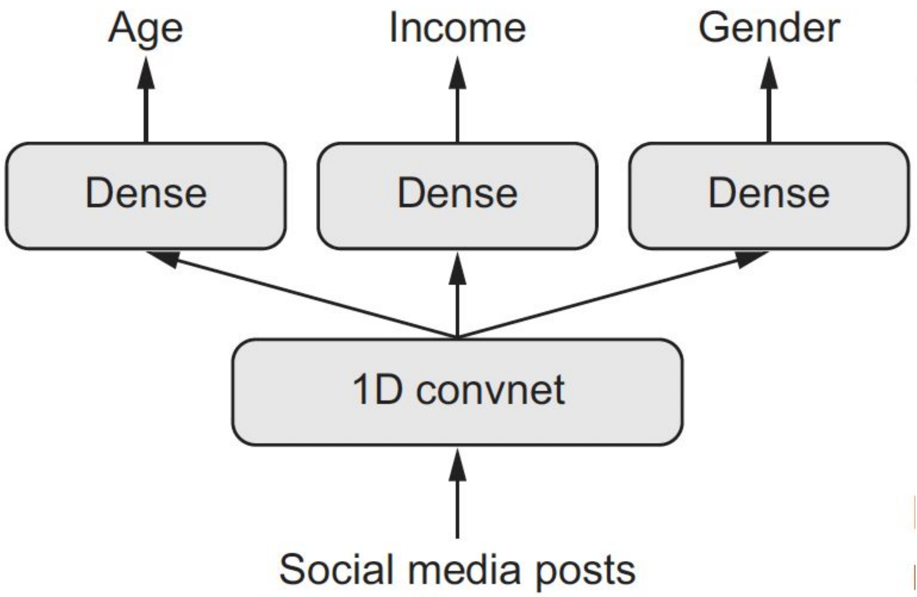
```
import numpy as np
num_samples = 1000
max_length = 100
text = np.random.randint(1, text_vocabulary_size, size = (num_samples, max_length))

question = np.random.randint(1, question_vocabulary_size,
size     = (num_samples, max_length))
answers  = np.random.randint(0, 1, size = (num_samples, answer_vocabulary_size))

model.fit([text, question], answers, epochs = 10, batch_size = 128)
model.fit({'text': text, 'question': question}, answers, epochs=10, batch_size=128)
```

Fitting using a list of inputs

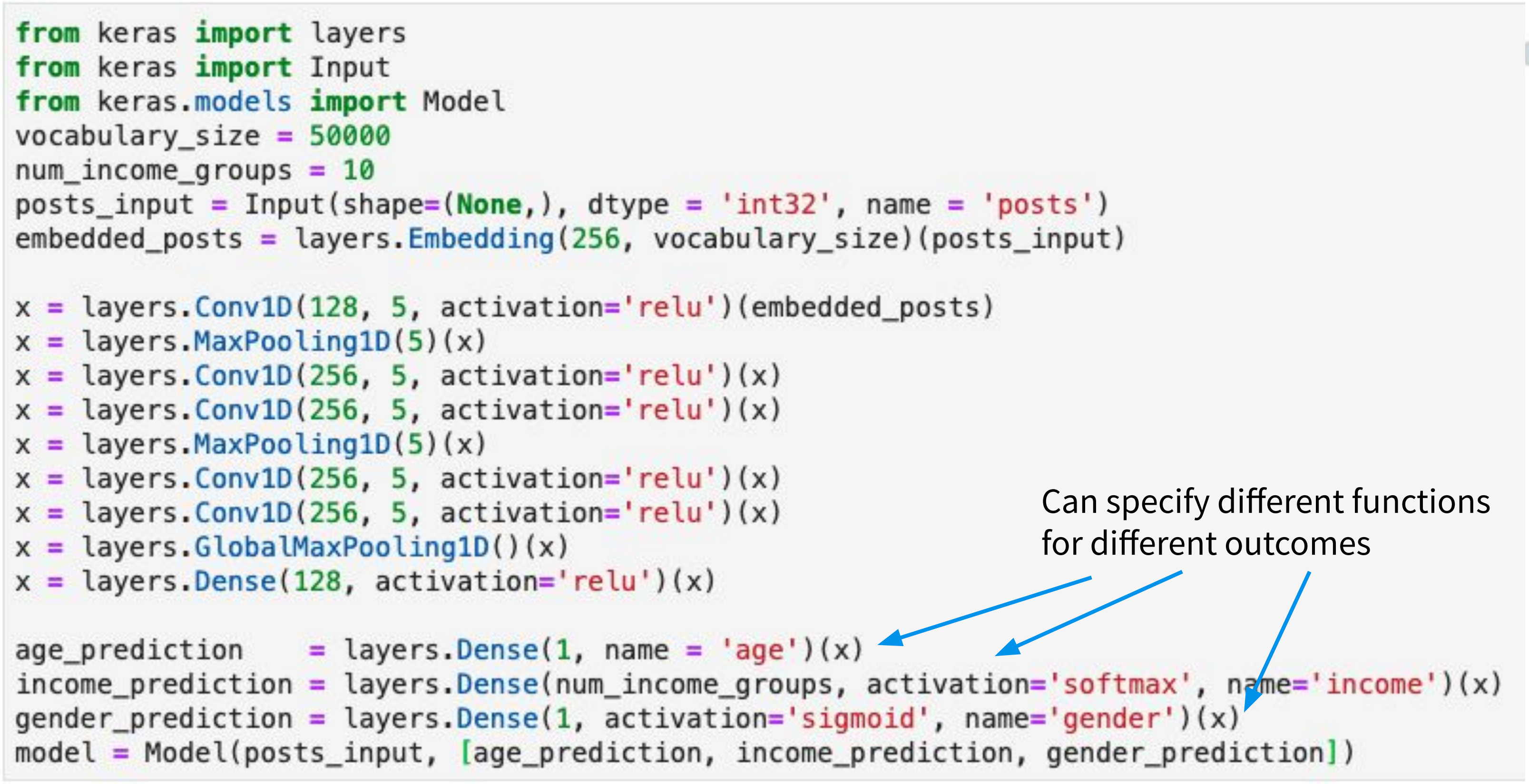
Fitting using a dictionary of inputs

Multi-output Models

- Example: a network that attempts to simultaneously predict different properties of the data, such as a network that takes as input a series of social media posts from a single anonymous person and tries to predict attributes of that person, such as age, gender, and income leve



Multi-output Models

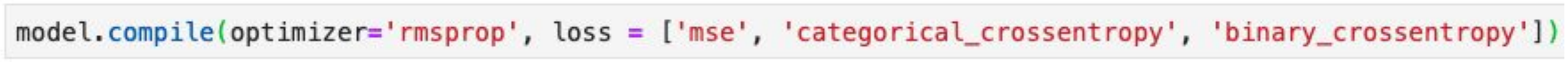


```
from keras import layers
from keras import Input
from keras.models import Model
vocabulary_size = 50000
num_income_groups = 10
posts_input = Input(shape=(None,), dtype = 'int32', name = 'posts')
embedded_posts = layers.Embedding(256, vocabulary_size)(posts_input)

x = layers.Conv1D(128, 5, activation='relu')(embedded_posts)
x = layers.MaxPooling1D(5)(x)
x = layers.Conv1D(256, 5, activation='relu')(x)
x = layers.Conv1D(256, 5, activation='relu')(x)
x = layers.MaxPooling1D(5)(x)
x = layers.Conv1D(256, 5, activation='relu')(x)
x = layers.Conv1D(256, 5, activation='relu')(x)
x = layers.GlobalMaxPooling1D()(x)
x = layers.Dense(128, activation='relu')(x)

age_prediction    = layers.Dense(1, name = 'age')(x)
income_prediction = layers.Dense(num_income_groups, activation='softmax', name='income')(x)
gender_prediction = layers.Dense(1, activation='sigmoid', name='gender')(x)
model = Model(posts_input, [age_prediction, income_prediction, gender_prediction])
```

Can specify different functions for different outcomes

Multi-output Models

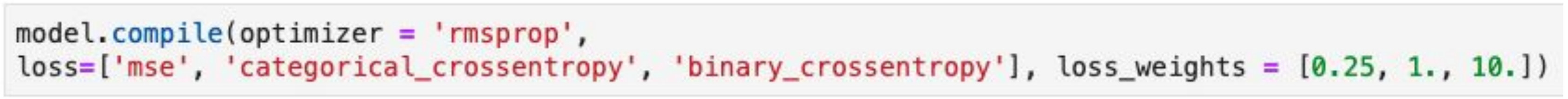
- This model requires the ability to specify different loss functions for different heads of the network:
    - Age prediction is a scalar regression task
    - Gender prediction is a binary classification task
- But because gradient descent requires you to minimize a scalar, you must combine these losses into a single value in order to train the model.
- The simplest way to combine different losses is to sum them all.
    - In Keras, you can use either a list or a dictionary of losses in **compile** to specify different objects for different outputs; the resulting loss values are summed into a global loss, which is minimized during training.



```
model.compile(optimizer='rmsprop', loss = ['mse', 'categorical_crossentropy', 'binary_crossentropy'])
```

Multi-output Models

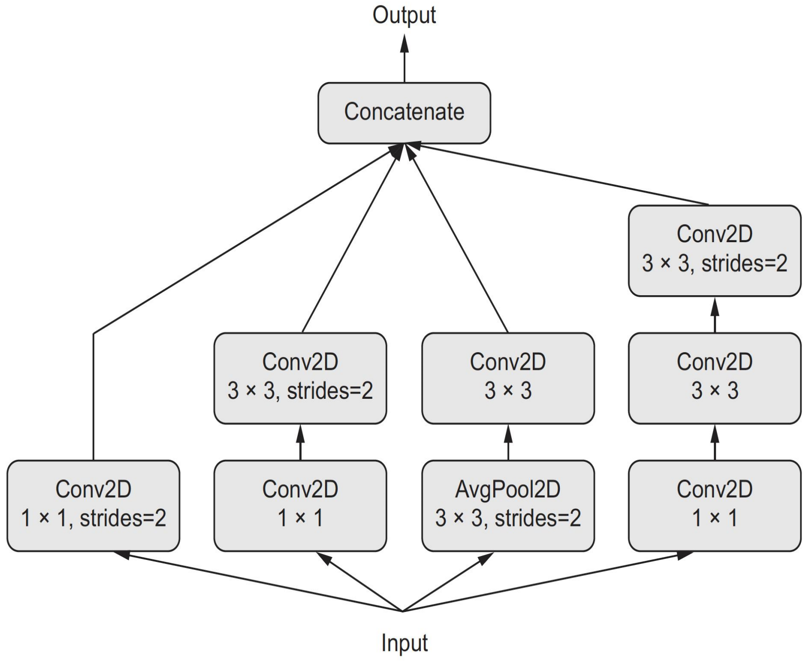
- A note on imbalanced loss contributions
  - Imbalances will cause the model representations to be optimized preferentially for the task with the largest individual loss, at the expense of the other tasks
  - To remedy this, you can assign different levels of importance to the loss values in their contribution to the final loss
  - This is particularly useful if the losses' values use different scales
  - Example:
    - MSE takes values around 3-5
    - Cross-entropy loss can be as low as 0.1
    - Here we could assign a weight of 10 for the cross-entropy loss and a weight of 0.25 to the MSE loss



```
model.compile(optimizer = 'rmsprop',
loss=['mse', 'categorical_crossentropy', 'binary_crossentropy'], loss_weights = [0.25, 1., 10.])
```

DAGs of Layers

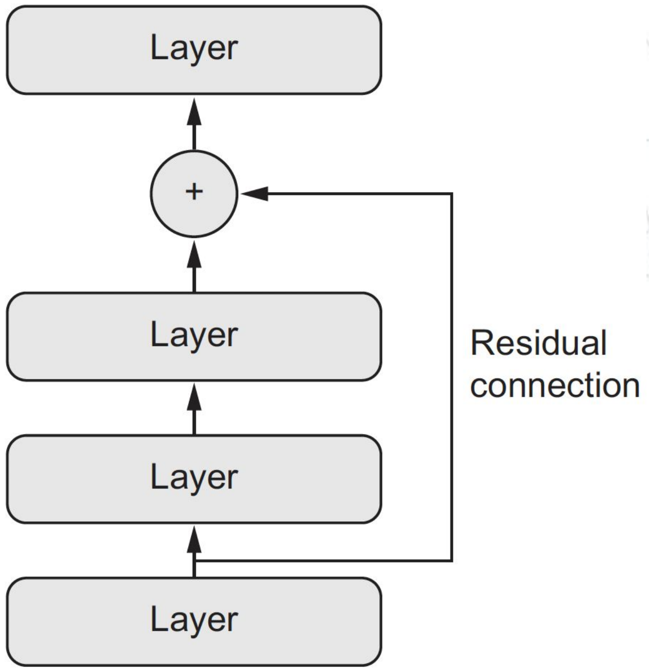
- You can also code complex network architectures in Keras
- Can specify independent branches
- Need to make sure the output of each branch is the same size so you can concatenate them at the end



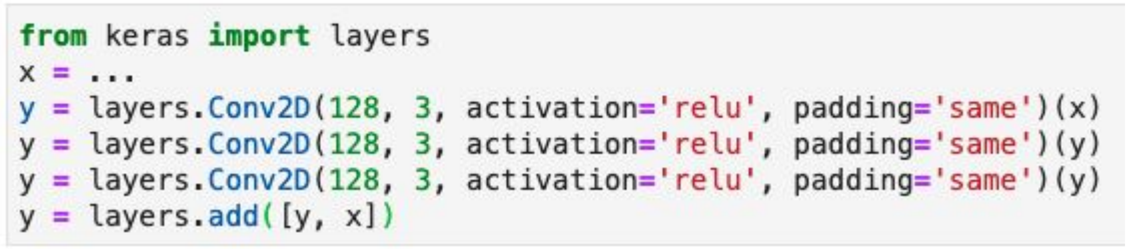
```python
from keras import layers

branch_a = layers.Conv2D(128, 1, activation='relu', strides=2)(x)

branch_b = layers.Conv2D(128, 1, activation='relu')(x)
branch_b = layers.Conv2D(128, 3, activation='relu', strides=2)(branch_b)

branch_c = layers.AveragePooling2D(3, strides=2)(x)
branch_c = layers.Conv2D(128, 3, activation='relu')(branch_c)

branch_d = layers.Conv2D(128, 1, activation='relu')(x)
branch_d = layers.Conv2D(128, 3, activation='relu')(branch_d)
branch_d = layers.Conv2D(128, 3, activation='relu', strides=2)(branch_d)

output = layers.concatenate([branch_a, branch_b, branch_c, branch_d], axis = -1)
```

Residual Connections

- Residual connections are a common graph-like network component found in many post-2015 network architectures, including Xception
- In general, adding residual connections to any model that has more than 10 layers is likely to be beneficial
- A residual connection consists of making the output of an earlier layer available as input to a later layer, effectively creating a shortcut in a sequential network.
  - Rather than being concatenated to the later activation, the earlier output is summed with the later activation, which assumes that both activations are the same size



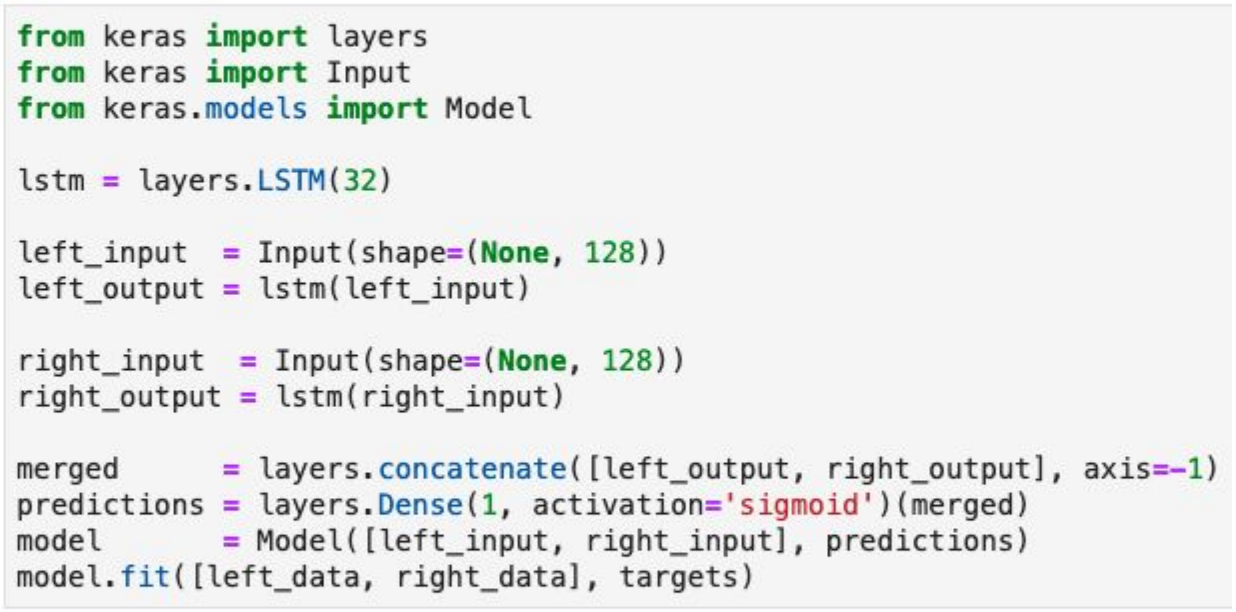
Reintroduces x



```
from keras import layers
x = ...
y = layers.Conv2D(128, 3, activation='relu', padding='same')(x)
y = layers.Conv2D(128, 3, activation='relu', padding='same')(y)
y = layers.Conv2D(128, 3, activation='relu', padding='same')(y)
y = layers.add([y, x])
```

Layer Weight Sharing

- One more important feature of the functional API is the ability to reuse a layer instance several times
- When you call a layer instance twice, instead of instantiating a new layer for each call, you reuse the same weights with every call. This allows you to build models that have shared branches—several branches that all share the same knowledge and perform the same operations. That is, they share the same representations and learn these representations simultaneously for different sets of inputs



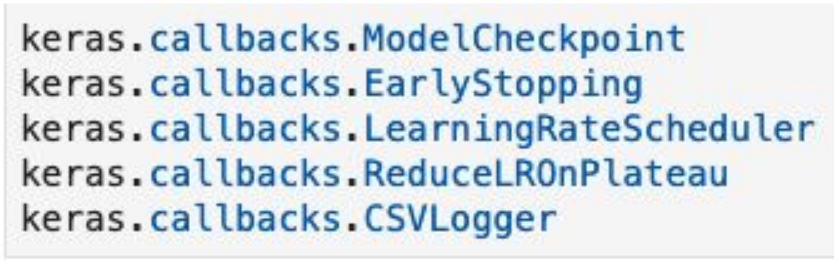
```
from keras import layers
from keras import Input
from keras.models import Model

lstm = layers.LSTM(32)

left_input  = Input(shape=(None, 128))
left_output = lstm(left_input)

right_input  = Input(shape=(None, 128))
right_output = lstm(right_input)

merged      = layers.concatenate([left_output, right_output], axis=-1)
predictions = layers.Dense(1, activation='sigmoid')(merged)
model       = Model([left_input, right_input], predictions)
model.fit([left_data, right_data], targets)
```

Callbacks

- When building and training a model, there are many decisions about hyperparameters you must make - but you don't always know where to start or which values will be the optimal choices
- A **callback** is an object that is passed to the model in the call to fit and that is called by the model at various points during training
- It has access to all the available data about the state of the model and its performance, and it can take action: interrupt training, save a model, load a different weight set, or otherwise alter the state of the model
- Examples:
  - **Model checkpointing**: saving the current weights of the model at different points during training
  - **Early stopping**: interrupting training when the validation loss is no longer improving (and saving the best model obtained during training)
  - **Dynamically adjusting the value of certain parameters during training**: such as the learning rate of the optimizer
  - **Logging training and validation metrics during training**, or visualizing the representations learned by the model as they are updated

Callbacks

- Examples of some of the available callbacks
- You can also create your own callbacks



```
keras.callbacks.ModelCheckpoint
keras.callbacks.EarlyStopping
keras.callbacks.LearningRateScheduler
keras.callbacks.ReduceLROnPlateau
keras.callbacks.CSVLogger
```

Advanced Architecture Patterns

Batch Normalization

- Normalization is a broad category of methods that seek to make different samples seen by a machine learning model more similar to each other, which helps the model learn and generalize well to new data

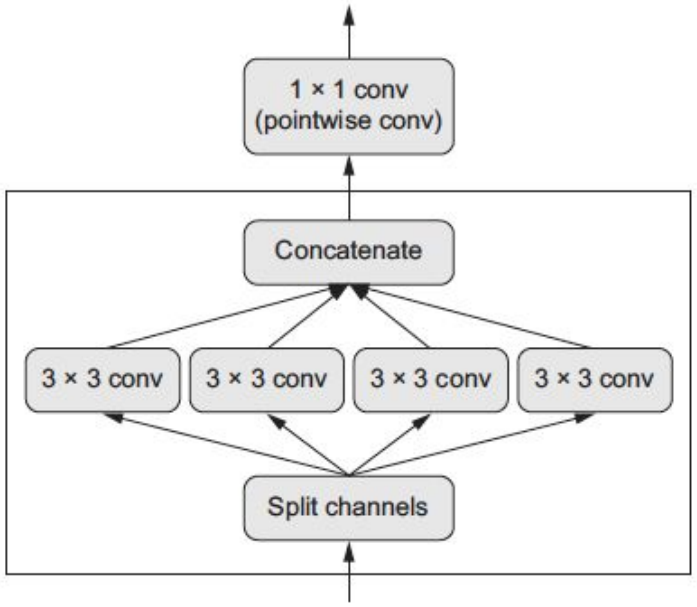
- We have already done normalization by transforming data to have mean 0 and standard deviation equal to 1

- Batch normalization is a type of layer (BatchNormalization in Keras)
  - It can adaptively normalize data even as the mean and variance change over time during training.
  - It works by internally maintaining an exponential moving average of the batch-wise mean and variance of the data seen during training.

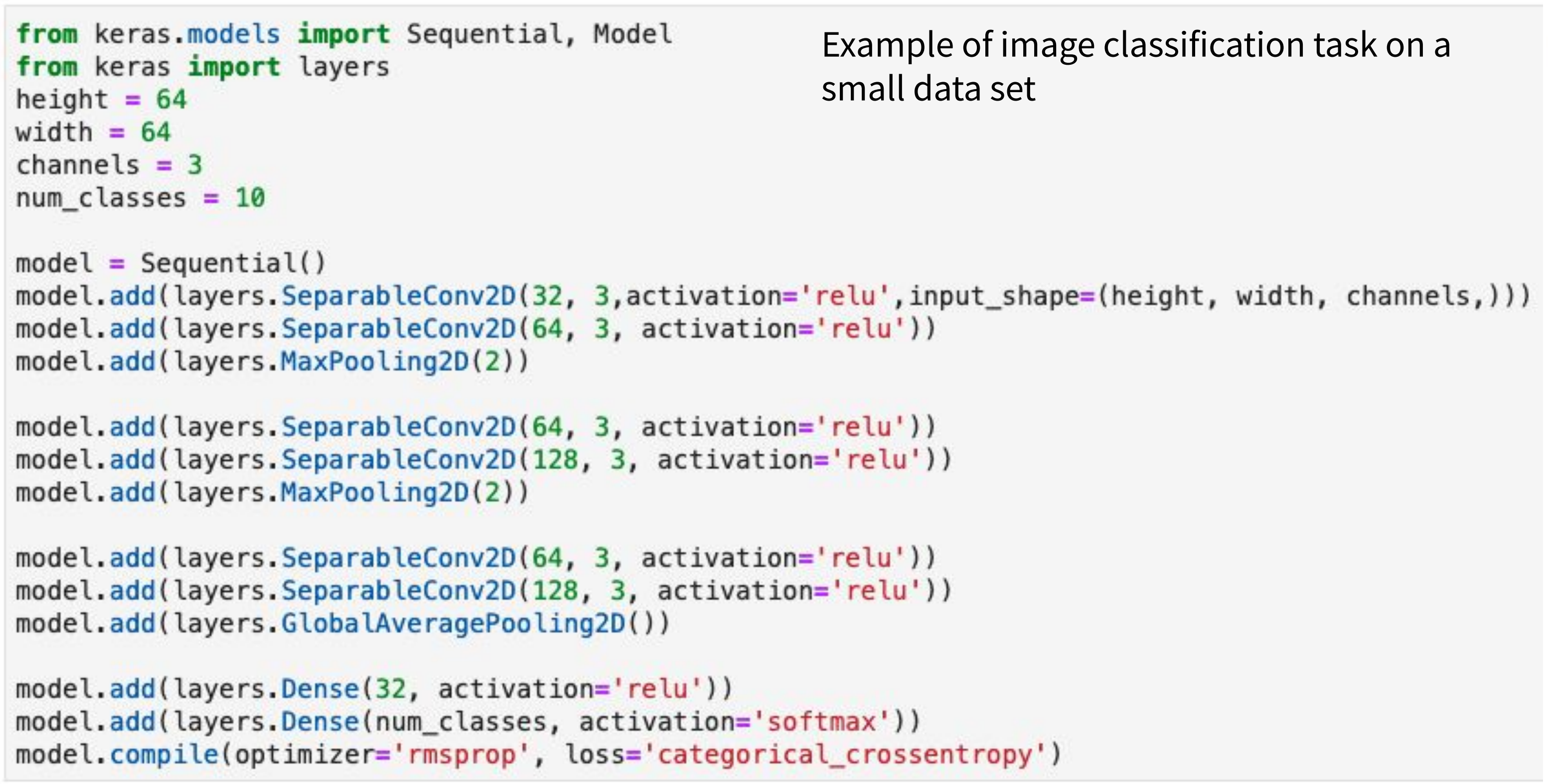
- The main effect of batch normalization is that it helps with gradient propagation—much like residual connections—and thus allows for deeper networks

Depthwise Separable Convolution

- A depthwise separable convolution layer performs a spatial convolution on each channel of its input, independently, before mixing output channels via a pointwise convolution
  - Separates the learning of spatial features and the learning of channel-wise features
  - Makes sense if you assume that spatial locations in the input are highly correlated but different channels are fairly independent
- It requires significantly fewer parameters and involves fewer computations, making it much faster
- Tends to learn better representations using less data, resulting in better-performing models
- Are the basis for the Xception architecture



Depthwise Separable Convolution



```
from keras.models import Sequential, Model
from keras import layers
height = 64
width = 64
channels = 3
num_classes = 10

model = Sequential()
model.add(layers.SeparableConv2D(32, 3,activation='relu',input_shape=(height, width, channels,)))
model.add(layers.SeparableConv2D(64, 3, activation='relu'))
model.add(layers.MaxPooling2D(2))

model.add(layers.SeparableConv2D(64, 3, activation='relu'))
model.add(layers.SeparableConv2D(128, 3, activation='relu'))
model.add(layers.MaxPooling2D(2))

model.add(layers.SeparableConv2D(64, 3, activation='relu'))
model.add(layers.SeparableConv2D(128, 3, activation='relu'))
model.add(layers.GlobalAveragePooling2D())

model.add(layers.Dense(32, activation='relu'))
model.add(layers.Dense(num_classes, activation='softmax'))
model.compile(optimizer='rmsprop', loss='categorical_crossentropy')
```

Example of image classification task on a small data set

Hyperparameter Optimization

- There is no way of knowing which values are the optimal ones before building and training your model
- Even with experience and intuition, your first pass at the values will be suboptimal
- There are no formal rules to tell you which values for are the best ones for your task
- You can (and we have in this course) tweak the value of each hyperparameter by hand
  - But this is inefficient
- It’s better to let the machine do this, and there is a whole field of research around this
  - Bayesian optimization
  - Genetic algorithms
  - Simple random search
  - Grid search
  - etc.

Hyperparameter Optimization

- The process:
  1. Choose a set of hyperparameters (automatically)
  2. Build the corresponding model
  3. Fit it to your training data and measure the final performance on validation data
  4. Choose the next set of hyperparameters to try (automatically)
  5. Repeat
  6. Eventually, measure performance on your test data

- Not a ton of tools available yet
  - One option is Hyperopt
    - A Python library for hyperparameter optimization that internally uses trees of Parzen estimators to predict sets of hyperparameters that are likely to work well
  - Another option is Hperas
    - Another library that integrates Hyperopt for use with Keras

Hyperparameter Optimization

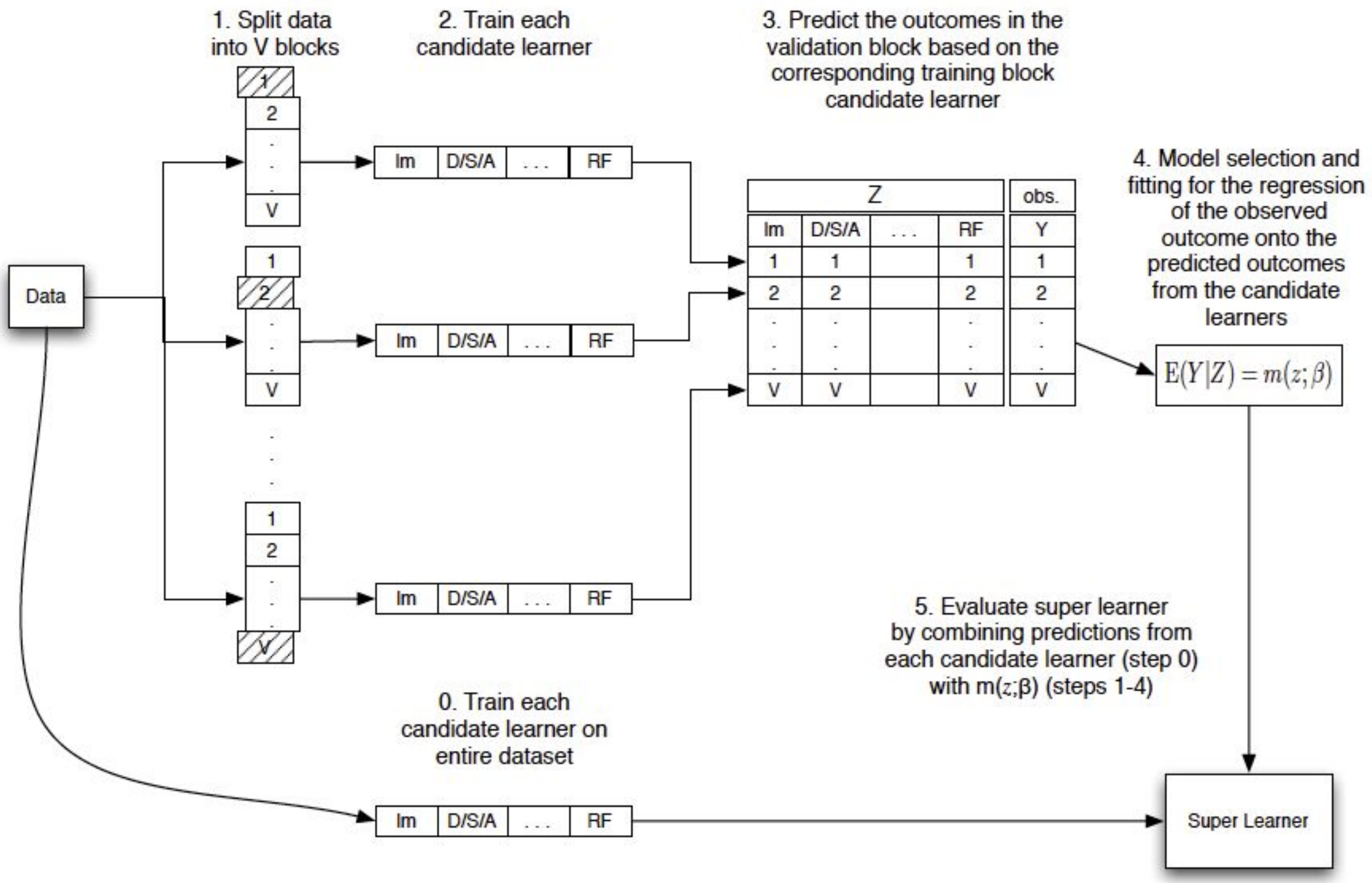
- Turns out that (for now), random search performs the best

- CAUTION!!
  - Can easily overfit to the validation data
  - Can be very computationally expensive

Model Ensembling

- Ensembling consists of pooling together the predictions of a set of different models to produce better predictions
- Ensemble models are **as good or better** than one model alone
- Assumes that different good models trained independently are likely to be good for different reasons - each model looks at slightly different aspects of the data to make its predictions, getting part of the "truth", but not all of it
- SuperLearner
  - In R
  - In Python

SuperLearner


1. Split data into V blocks
2. Train each candidate learner
3. Predict the outcomes in the validation block based on the corresponding training block candidate learner
4. Model selection and fitting for the regression of the observed outcome onto the predicted outcomes from the candidate learners
5. Evaluate super learner by combining predictions from each candidate learner (step 0) with m(z;β) (steps 1-4)
0. Train each candidate learner on entire dataset
Data
1
2
V
lm
D/S/A
...
RF
Z
obs.
Y
E(Y|Z) = m(z;β)
Super Learner