

Table of Contents

PyImageSearch	3
<i>Webpage</i>	3
1. What is the difference between categorical and binary cross-entropy	3
2. How do you save and load Keras model	3
3. Data augmentation with Keras	3
4. Name some motion detection algorithm	4
5. Explain the problematic behind motion detection	5
6. Build a simple motion detector	5
7. How do you run a python OpenCV script on startup	5
8. Deep learning on Raspberry Pi	6
9. Ball tracking with OpenCV	6
10. How is object tracking defined	7
11. Describe an ideal object tracking algorithm	7
12. Centroid tracking with OpenCV	7
13. Name several tracking algorithms	9
14. Explain KFC tracking algorithm (Kernelized correlation filter)	10
15. Explain CSRT tracking algorithm (Discriminative Correlation Filter)	11
16. Mosse Tracker (Minimum Output Sum of Squared Error)	11
17. Explain dlib	11
18. How do you combine object detection and object tracking into one algorithm	12
19. What are the limitations using multiple object trackers	12
20. How would you perform tracking of multiple objects (single CPU)	12
21. How does multiprocessing work	13
22. How do you set up a number of single CPUs	15
23. What is the difference between multi-threading and multi-processing	15
24. What is the difference between 1 and 2 stage detectors	15
25. What are the drawbacks of YOLO	17
<i>Starter Bundle</i>	19
Chapter 9:	19
1. What is the bias trick?	19
2. Explain the gradient descent	19
3. Explain the stochastic gradient descent	20
4. Explain momentum	22
5. Explain Nesterov acceleration	22
6. What is the difference between adaptive learning methods to “classical” optimizers	23
7. Explain Adagrad	23
8. Explain adadelat	24
9. Explain RMSProp	24
10. Explain Adam	25
11. What is an encoder – decoder network	25
12. What types of regularization do you have	26
13. Why do we need regularization	26
14. How does regularization work	27
Chapter 10	27
15. What types of activation functions do we have	27
16. What activation function would you use	29
17. What are the “ingredients”	29

18. Explain various weight initialization method	29
Chapter 11	30
<i>Practitioner bundle</i>	<i>30</i>
<i>ImageNet Bundle</i>	<i>30</i>
Chapter 1:	30
19. What are the advantages of mxnet library?	30
20. How do multiple GPU scale?	31
Chapter 16	31
21. Drawback of R-CNN	31
22. Explain SSD	31
23. Explain non-maximum suppression (NMS)	34
24. What is the difference between YOLO and SSD	34
Questions	34
25. Explain transfer learning	34
26. What hyperparameters do you tune in a network	35
27. How do you search for optimal parameters	35
28. How many parameters does a CNN need	35
29. Explain max pooling in a CNN	36
30. What is pooling used for	36
31. What is 1x1 convolution good for	36
32. Explain padding types	37
33. Explain filters	37
34. Define evaluation metrics	37
35. What is fine tuning	40
36. Define a model in Keras	40
37. What is ROS	41
38. What udacity projects did you work on	42
39. What is catkin	42
40. Explain over and underfitting	43
41. describe inception module	43
Describe the ResNet residual module	43
42. Keras Conv2D and Convolutional Layers	44
Projects for Interview	47
<i>Cell recognition and tracking</i>	<i>47</i>
<i>Traffic counter</i>	<i>48</i>
<i>Traffic sign recognition</i>	<i>48</i>
<i>Smart home systems</i>	<i>49</i>

PylImageSearch

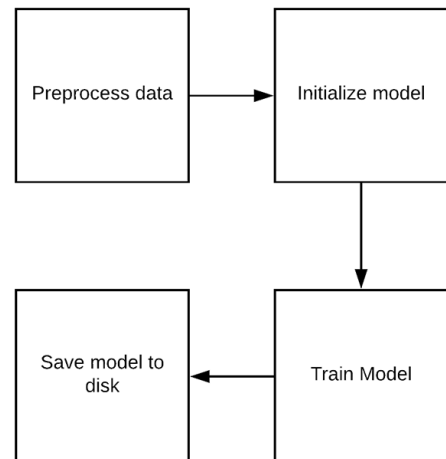
Webpage

1. What is the difference between categorical and binary cross-entropy

Binary cross-entropy is used if the model has two output classes. Categorical entropy is used if model has > 2 classes.

2. How do you save and load Keras model

Your preprocessing steps for training and validation must be *identical* to the training steps when loading your model and classifying new images. These following steps are necessary to save the model:



When loading the model, the new data/images have to be processed in the same way in order to make meaningful predictions.



3. Data augmentation with Keras

```

Keras - Save and Load Your Deep Learning Models
42 # initialize the training training data augmentation object
43 trainAug = ImageDataGenerator(
44     rescale=1 / 255.0,
45     rotation_range=20,
46     zoom_range=0.05,
47     width_shift_range=0.05,
48     height_shift_range=0.05,
49     shear_range=0.05,
50     horizontal_flip=True,
51     fill_mode="nearest")
52
53 # initialize the validation (and testing) data augmentation object
54 valAug = ImageDataGenerator(rescale=1 / 255.0)
  
```

Data augmentation is the process of generating new images from a dataset with random modifications. It results in a better deep learning model and I almost always recommend it (it is especially important for small datasets).

```
Keras - Save and Load Your Deep Learning Models
53 # initialize the training generator
54 trainGen = trainAug.flow_from_directory(
55     TRAIN_PATH,
56     class_mode="categorical",
57     target_size=(64, 64),
58     color_mode="rgb",
59     shuffle=True,
60     batch_size=BS)
61
62 # initialize the validation generator
63 valGen = valAug.flow_from_directory(
64     VAL_PATH,
```

The three generators above actually produce images on demand during training / validation / testing per our augmentation objects and the parameters given here.

```
# import the necessary packages
from keras.preprocessing.image import ImageDataGenerator
from keras.optimizers import SGD
from pyimagesearch.resnet import ResNet
from sklearn.metrics import classification_report
```

4. Name some motion detection algorithm

Before we get started coding in this post, let me say that there are many, *many* ways to perform motion detection, tracking, and analysis in OpenCV. Some are very simple. And others are very complicated. The two primary methods are forms of Gaussian Mixture Model-based foreground and background segmentation:

1. *An improved adaptive background mixture model for real-time tracking with shadow detection* by KaewTraKulPong et al., available through the `cv2.BackgroundSubtractorMOG` function.
2. *Improved adaptive Gaussian mixture model for background subtraction* by Zivkovic, and *Efficient Adaptive Density Estimation per Image Pixel for the Task of Background Subtraction*, also by Zivkovic, available through the `cv2.BackgroundSubtractorMOG2` function.

And in newer versions of OpenCV we have Bayesian (probability) based foreground and background segmentation, implemented from Godbehere et al.'s 2012 paper, *Visual Tracking of Human Visitors under Variable-Lighting Conditions for a Responsive Audio Art Installation*. We can find this implementation in the `cv2.createBackgroundSubtractorGMG` function (we'll be waiting for OpenCV 3 to fully play with this function though).

All of these methods are concerned with *segmenting the background from the foreground* (and they even provide mechanisms for us to discern between actual motion and just shadowing and small lighting changes)!

5. Explain the problematic behind motion detection

The *background* of our video stream is largely *static and unchanging* over consecutive frames of a video. Therefore, if we can model the background, we monitor it for substantial changes. If there is a substantial change, we can detect it — this change normally corresponds to *motion* on our video.

Problems:

Now obviously in the real-world this assumption can easily fail. Due to shadowing, reflections, lighting conditions, and any other possible change in the environment, our background can look quite different in various frames of a video. And if the background appears to be different, it can throw our algorithms off. That's why the most successful background subtraction/foreground detection systems *utilize fixed mounted cameras* and in *controlled lighting conditions*

6. Build a simple motion detector

- Use first frame as a background
- Due to tiny variations in the digital camera sensors, no two frames will be 100% the same — some pixels will *most certainly* have different intensity values. That said, we need to account for this and apply Gaussian smoothing to average pixel intensities across an 21×21 region. This helps **smooth out high frequency noise** that could throw our motion detection algorithm off.
- In order to avoid false motion detection, caused by different lightning conditions, we instead take the **weighted mean of previous frames along with the current frame**. This means that our script can *dynamically adjust* to the background, even as the time of day changes along with the lighting conditions.
- Based on the weighted average of frames, we then subtract the weighted average from the current frame, leaving us with what we call a *frame delta*

$\text{delta} = |\text{background_model} - \text{current_frame}|$

- the delta is less than 25, we discard the pixel and set it to black (i.e. background). If the delta is greater than 25, we'll set it to white (i.e. foreground)
- Apply contour detection `cv2.findContours()`
- loop over contours and filter small contours away from the image
- draw a bounding box around each contour that passed the criteria
- from there one can upload or save the images on the disk. or do something else

7. How do you run a python OpenCV script on startup

- make the file executable with `chmod +x file.sh`
- write a shell scrip to launch the file

- update crontab:
`sudo crontab -e`
`@reboot /home/pi/pi-reboot/on_reboot.sh`

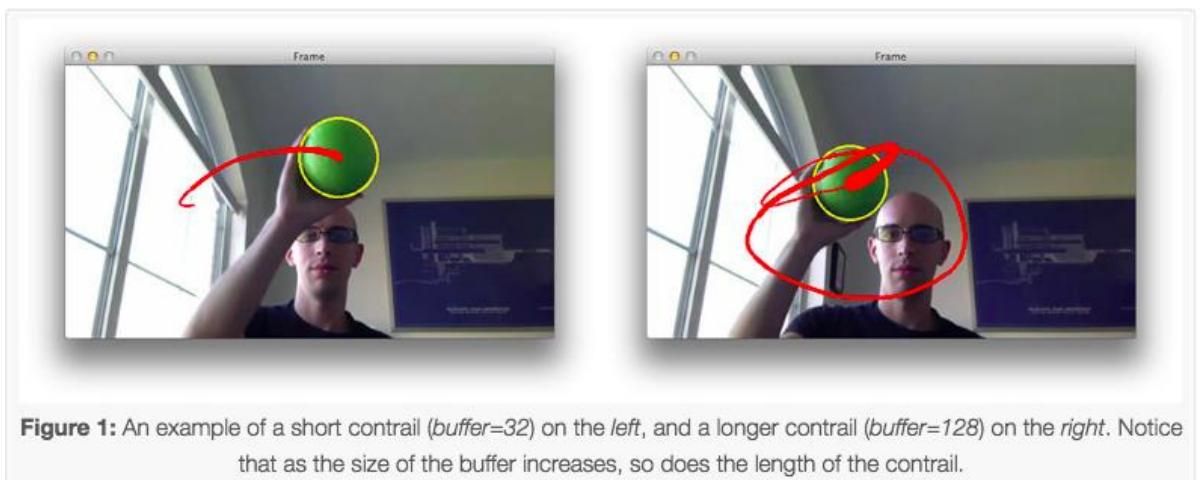
8. Deep learning on Raspberry Pi

As I've mentioned before, the Raspberry Pi is not suitable for training a neural network (outside of "toy" examples). However, the Raspberry Pi can be used to *deploy* a neural network [once it has already been trained](#) (provided the model can fit into a small enough memory footprint, of course).

- Use python 2.7 because of raspbian and pre-compiled tensorflow version
- models are saved as HDF5
- Increasing the swap will enable you to use the Raspberry Pi SD card for additional memory (a critical step when trying to compile and install large libraries on the memory-limited Raspberry Pi). Don't forget to revert the swap size when the project is done.

9. Ball tracking with OpenCV

- We'll be using deque, a [list-like data structure](#) with super fast appends and pops to maintain a list of the past N (x, y) -locations of the ball in our video stream. Maintaining such a queue allows us to draw the "contrail" of the ball as its being tracked.
- A second optional argparse argument, `--buffer` is the maximum size of our deque, which maintains a list of the previous (x, y) -coordinates of the ball we are tracking. This deque allows us to draw the "contrail" of the ball, detailing its past locations. A smaller queue will lead to a shorter tail whereas a larger queue will create a longer tail (since more points are being tracked):



- blur the frame to reduce high frequency noise and allow us to focus on the structural objects inside the frame, such as the ball. Finally, we'll convert the frame to the HSV color space.
- construct a mask for the color "green", then perform a series of dilations and erosions to remove any small blobs left in the mask
- In the case of multiple objects, each object would need its own deque

10. How is object tracking defined

1. Taking an initial set of object detections (such as an input set of bounding box coordinates)
2. Creating a unique ID for each of the initial detections
3. And then tracking each of the objects as they move around frames in a video, maintaining the assignment of unique IDs

Furthermore, object tracking allows us to **apply a unique ID to each tracked object**, making it possible for us to count unique objects in a video. Object tracking is paramount to building a **person counter** (which we'll do later in this series).

11. Describe an ideal object tracking algorithm

- Only require the object detection phase once (i.e., when the object is initially detected)
- Will be extremely fast — *much* faster than running the actual object detector itself
- Be able to handle when the tracked object "disappears" or moves outside the boundaries of the video frame
- Be robust to occlusion
- Be able to pick up objects it has "lost" in between frames

12. Centroid tracking with OpenCV

In future posts in this object tracking series, I'll start going into more advanced kernel-based and correlation-based tracking algorithms.

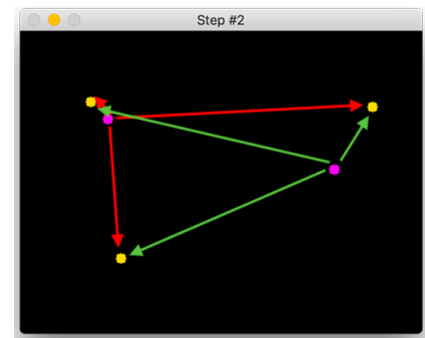
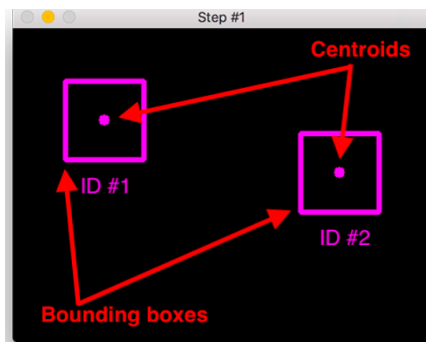
This object tracking algorithm is called *centroid tracking* as it relies on the Euclidean distance between (1) *existing* object centroids (i.e., objects the centroid tracker has already seen before) and (2) new object centroids between subsequent frames in a video.

The primary assumption of the centroid tracking algorithm is that a given object will potentially move in between subsequent frames, but the *distance* between the

centroids for frames F_t and F_{t+1} will be *smaller* than all other distances between objects.

Step #1: Accept bounding box coordinates and compute centroids

- The centroid tracking algorithm assumes that we are passing in a set of bounding box (x, y) -coordinates for each detected object in **every single frame**.
- These bounding boxes can be produced by any type of object detector you would like (color thresholding + contour extraction, Haar cascades, HOG + Linear SVM, SSDs, Faster R-CNNs, etc.), provided that they are computed for every frame in the video.
- Once we have the bounding box coordinates we must compute the “centroid”, or more simply, *the center* (x, y) -coordinates of the bounding box. **Figure 1** above demonstrates accepting a set of bounding box coordinates and computing the centroid.
- Since these are the first initial set of bounding boxes presented to our algorithm we will assign them unique IDs

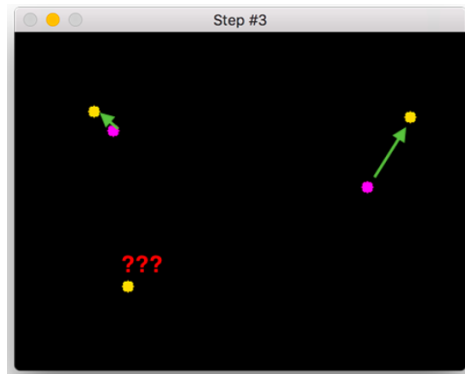


Step #2: Compute Euclidean distance between new bounding boxes and existing objects

- For every subsequent frame in our video stream we apply **Step #1** of computing object centroids; however, instead of assigning a new unique ID to each detected object (which would defeat the purpose of object tracking), we first need to determine if we can *associate* the *new* object centroids (yellow) with the *old* object centroids (purple). To accomplish this process, we compute the Euclidean distance (highlighted with green arrows) between each pair of existing object centroids and input object centroids.
- From **Figure 2** you can see that we have this time detected three objects in our image. The two pairs that are close together are two existing objects.

Step #3: Update (x, y) -coordinates of existing objects

- In **Figure 3** you can see how our centroid tracker algorithm chooses to associate centroids that minimize their respective Euclidean distances.



- New objects have to be registered

Step #4: Register new objects

- Assigning it a new object ID
- Storing the centroid of the bounding box coordinates for that object
- We can then go back to **Step #2** and repeat the pipeline of steps for every frame in our video stream.

Step #5: Deregister old objects

- old objects will be deregistered when they cannot be matched to any existing objects for a total of N subsequent frames.

Drawbacks

- BBs have to be computed for every frame
- if objects overlap ID switching might occur
- It's important to understand that the overlapping/occluded object problem is *not* specific to centroid tracking — it happens for many other object trackers as well, including advanced ones.

13. Name several tracking algorithms

1. **BOOSTING Tracker:** Based on the same algorithm used to power the machine learning behind Haar cascades (AdaBoost), but like Haar cascades, is over a decade old. This tracker is slow and doesn't work very well. Interesting only for legacy reasons and comparing other algorithms. (*minimum OpenCV 3.0.0*)
2. **MIL Tracker:** Better accuracy than BOOSTING tracker but does a poor job of reporting failure. (*minimum OpenCV 3.0.0*)
3. **KCF Tracker:** Kernelized Correlation Filters. Faster than BOOSTING and MIL. Similar to MIL and KCF, does not handle full occlusion well. (*minimum OpenCV 3.1.0*)

4. **CSRT Tracker:** Discriminative Correlation Filter (with Channel and Spatial Reliability). Tends to be more accurate than KCF but slightly slower. (*minimum OpenCV 3.4.2*)
5. **MedianFlow Tracker:** Does a nice job reporting failures; however, if there is too large of a jump in motion, such as fast moving objects, or objects that change quickly in their appearance, the model will fail. (*minimum OpenCV 3.0.0*)
6. **TLD Tracker:** I'm not sure if there is a problem with the OpenCV implementation of the TLD tracker or the actual algorithm itself, but the TLD tracker was incredibly prone to false-positives. I do not recommend using this OpenCV object tracker. (*minimum OpenCV 3.0.0*)
7. **MOSSE Tracker:** Very, very fast. Not as accurate as CSRT or KCF but a good choice if you need pure speed. (*minimum OpenCV 3.4.1*)
8. **GOTURN Tracker:** The only deep learning-based object detector included in OpenCV. It requires additional model files to run (will not be covered in this post). My initial experiments showed it was a bit of a pain to use even though it reportedly handles viewing changes well (my initial experiments didn't confirm this though). I'll try to cover it in a future post, but in the meantime, take a look at [Satya's writeup](#). (*minimum OpenCV 3.2.0*)

My personal suggestion is to:

- Use CSRT when you need **higher object tracking accuracy and can tolerate slower FPS throughput**
- Use KCF when you need **faster FPS throughput** but can **handle slightly lower object tracking accuracy**
- Use **MOSSE when you need pure speed**

Check this page for more information:

<https://www.learnopencv.com/object-tracking-using-opencv-cpp-python/>

14. Explain KFC tracking algorithm (Kernelized correlation filter)

The basic idea of the correlation filter tracking is estimating an optimal image filter such that the filtration with the input image produces a desired response. The desired response is typically of a Gaussian shape centered at the target location, so the score decreases with the distance.

KFC stands for **Kernelized Correlation Filters**. This tracker builds on the ideas presented in the previous two trackers. This tracker utilizes that fact that the multiple positive samples used in the MIL tracker have large overlapping regions. This overlapping data leads to some nice mathematical properties that is exploited by this tracker to make tracking faster and more accurate at the same time.

Pros: Accuracy and speed are both better than MIL and it reports tracking failure better than BOOSTING and MIL. If you are using OpenCV 3.1 and above, I recommend using this for most applications.

Cons : Does not recover from full occlusion.

15. Explain CSRT tracking algorithm (*Discriminative Correlation Filter*)

In the Discriminative Correlation Filter with Channel and Spatial Reliability (DCF-CSR), we use the spatial reliability map for adjusting the filter support to the part of the selected region from the frame for tracking. This ensures enlarging and localization of the selected region and improved tracking of the non-rectangular regions or objects. It uses only 2 standard features (HoGs and Colornames). It also operates at a comparatively lower fps (25 fps) but gives higher accuracy for object tracking.

16. Mosse Tracker (*Minimum Output Sum of Squared Error*)

Minimum Output Sum of Squared Error (MOSSE) uses adaptive correlation for object tracking which produces stable correlation filters when initialized using a single frame. MOSSE tracker is robust to variations in lighting, scale, pose, and non-rigid deformations. It also detects occlusion based upon the peak-to-sidelobe ratio, which enables the tracker to pause and resume where it left off when the object reappears. MOSSE tracker also operates at a higher fps (450 fps and even more). To add to the positives, it is also very easy to implement, is as accurate as other complex trackers and much faster. But, on a performance scale, it lags behind the deep learning based trackers.

17. Explain dlib

The [dlib correlation tracker implementation](#) is based on Danelljan et al.'s 2014 paper, [Accurate Scale Estimation for Robust Visual Tracking](#).

Their work, in turn, builds on the popular MOSSE tracker from Bolme et al.'s 2010 work, [Visual Object Tracking using Adaptive Correlation Filters](#). While the MOSSE tracker works well for objects that are translated, it often fails for objects that change in scale.

The work of Danelljan et al. proposed utilizing a scale pyramid to accurately estimate the scale of an object after the optimal translation was found. This breakthrough allows us to track objects that change in both (1) translation and (2) scaling throughout a video stream — and furthermore, we can perform this tracking in real-time.

```
tracker = dlib.correlation_tracker()
```

18. How do you combine object detection and object tracking into one algorithm

Highly accurate object trackers will *combine* the concept of object detection and object tracking into a single algorithm, typically divided into two phases:

- **Phase 1 — Detecting:** During the detection phase we are running our computationally more expensive object tracker to (1) detect if new objects have entered our view, and (2) see if we can find objects that were “lost” during the tracking phase. For each detected object we create or update an object tracker with the new bounding box coordinates. Since our object detector is more computationally expensive we only run this phase once every N frames.
- **Phase 2 — Tracking:** When we are not in the “detecting” phase we are in the “tracking” phase. For each of our detected objects, we create an object tracker to track the object as it moves around the frame. Our object tracker should be faster and more efficient than the object detector. We’ll continue tracking until we’ve reached the N -th frame and then re-run our object detector. The entire process then repeats.

The benefit of this hybrid approach is that we can apply highly accurate object detection methods *without* as much of the computational burden.

19. What are the limitations using multiple object trackers

Limitations:

- the more trackers we created, the slower our pipeline ran.
- we cannot use the same object tracker instance to track multiple objects.
 - For example, suppose we have 10 objects in a video that we would like to track, implying that:
 - We need to create 10 object tracker instances
 - And therefore, we’ll see the frames per second throughput of our pipeline decrease by a factor of 10.
- Solution: Multithreading or Multiprocessing

20. How would you perform tracking of multiple objects (single CPU)

- Let’s begin the **object detection phase**
- Next, we proceed to loop over the detections to find objects belonging to the “person” class since our input video is a human foot race
- Now that we’ve located each “person” in the frame, let’s instantiate our trackers and draw our initial bounding box(es) + class label(s)
- Compute the bounding box of each *detected* object.

- Instantiate and pass the bounding box coordinates to the *tracker*. The bounding box is especially important here. We need to create a `dlib.rectangle` for the bounding box and pass it to the `start_track` method. From there, `dlib` can start to track the object.
- Finally, we populate the `trackers` list with the individual tracker
- If the length of our `detections` list is greater than zero, we know we are in the **object tracking phase**:
- In the object *tracking* phase, we loop over all trackers and corresponding labels
- Then we proceed to update each object position. In order to update the position, we simply pass the `rgb` image.

21. How does multiprocessing work

To spawn this process we need to provide a function that Python can call, which Python will then take and create a brand new process + execute it:

```
Multi-object tracking with dlib
10 def start_tracker(box, label, rgb, inputQueue, outputQueue):
11     # construct a dlib rectangle object from the bounding box
12     # coordinates and then start the correlation tracker
13     t = dlib.correlation_tracker()
14     rect = dlib.rectangle(box[0], box[1], box[2], box[3])
15     t.start_track(rgb, rect)
```

Keep in mind how Python multiprocessing works — Python will call this function and then create a brand new interpreter to execute the code within. Therefore, each `start_tracker` spawned process will be independent from its parent. To communicate with the Python driver script we need to leverage either [Pipes or Queues](#). Both types of objects are thread/process safe, accomplished using locks and semaphores.

In essence, we are creating a simple producer/consumer relationship:

- Our parent process will **produce** new frames and add them to the queue of a particular object tracker.
- The child process will then **consume** the frame, apply object tracking, and then return the updated bounding box coordinates.

I decided to use `Queue` objects for this post; however, keep in mind that you could use a `Pipe` if you wish — be sure to refer to the [Python multiprocessing documentation](#) for more details on these objects.

IMPROVEMENT

The dlib multi-object tracking Python scripts I've shared with you today will work just fine for processing *shorter video streams*; however, if you intend on utilizing this implementation for *long-running production environments* (in the order of many hours to days of video) there are two primary improvements I would suggest you make:

The **first improvement** would be to **utilize processing pools** rather than spawning a brand new process for each object to be tracked.

The implementation covered here today constructs a brand new Queue and Process for each object that we need to track.

For today's purposes that's fine, but consider if you wanted to track 50 objects in a video — this implies that you would spawn 50 processes, one for each object. At that point, the overhead of your system managing all those processes will destroy any increase in FPS throughput. Instead, you would want to utilize processing pools.

If your system has N processor cores, then you would want to create a pool with $N - 1$ processes, leaving one core to your operating system to perform system operations. Each of these processes should perform multiple object tracking, maintaining a list of object trackers, similar to the first multi-object tracking we covered today.

This improvement will allow you to utilize all cores of your processor without the overhead of having to spawn many independent processes.

The **second improvement** I would make is to **clean up the processes and queues**. In the event that dlib reports an object as "lost" or "disappeared" we are not returning from the `start_tracker` function, implying that that process will live for the life of the parent script and only be killed when the parent exits.

Again, that's fine for our purposes here today, but if you intend on utilizing this code in production environments, you should:

1. Update the `start_tracker` function to return once dlib reports the object as lost.
 2. Delete the `inputQueue` and `outputQueue` for the corresponding process as well.
- Failing to perform this cleanup will lead to needless computational consumption and memory overhead for long-running jobs.

The **third improvement** is to **improve tracking accuracy by running the object detector every N frames** (rather than just once at the start).

22. How do you set up a number of single CPUs

With the Pool methods: `pool = mp.Pool(processes=4)`

23. What is the difference between multi-threading and multi-processing

Python interpreter was designed with simplicity in mind and has a thread-safe mechanism, the so-called “GIL” (Global Interpreter Lock). In order to prevent conflicts between threads, it executes only one statement at a time (so-called serial processing, or single-threading).

The difference is in memory usage.

Threading:

If we submit “jobs” to different threads, those jobs can be pictured as “sub-tasks” of a single process and those threads will usually have access to the same memory areas (i.e., shared memory). This approach can easily lead to conflicts in case of improper synchronization, for example, if processes are writing to the same memory location at the same time.

Processing:

A safer approach (although it comes with an additional overhead due to the communication overhead between separate processes) is to submit multiple processes to completely separate memory locations (i.e., distributed memory): Every process will run completely independent from each other.

24. What is the difference between 1 and 2 stage detectors

When it comes to deep learning-based object detection, there are three primary object detectors you’ll encounter:

- R-CNN and their variants, including the original R-CNN, Fast R- CNN, and Faster R-CNN
- Single Shot Detector (SSDs)
- YOLO
-

R-CNNs are one of the first deep learning-based object detectors and are an example of a **two-stage detector**.

1. In the first R-CNN publication, [Rich feature hierarchies for accurate object detection and semantic segmentation](#), (2013) Girshick et al. proposed an object

detector that required an algorithm such as [Selective Search](#) (or equivalent) to propose candidate bounding boxes that could contain objects.

2. These regions were then passed into a CNN for classification, ultimately leading to one of the first deep learning-based object detectors.

The problem with the standard R-CNN method was that it was *painfully slow* and not a complete end-to-end object detector.

Girshick et al. published a second paper in 2015, entitled [Fast R- CNN](#). The Fast R-CNN algorithm made considerable improvements to the original R-CNN, namely increasing accuracy and reducing the time it took to perform a forward pass; however, the model still relied on an external region proposal algorithm.

It wasn't until Girshick et al.'s follow-up 2015 paper, [Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks](#), that R-CNNs became a true end-to-end deep learning object detector by removing the Selective Search requirement and instead relying on a Region Proposal Network (RPN) that is (1) fully convolutional and (2) can predict the object bounding boxes and "objectness" scores (i.e., a score quantifying how likely it is a region of an image may contain an image). The outputs of the RPNs are then passed into the R-CNN component for final classification and labeling.

While R-CNNs tend to very accurate, the biggest problem with the R-CNN family of networks is their speed — they were incredibly slow, obtaining only 5 FPS on a GPU.

To help increase the speed of deep learning-based object detectors, both Single Shot Detectors (SSDs) and YOLO use a *one-stage detector strategy*.

These algorithms treat object detection as a regression problem, taking a given input image and simultaneously learning bounding box coordinates and corresponding class label probabilities.

In general, single-stage detectors tend to be less accurate than two-stage detectors *but are significantly faster*.

YOLO is a great example of a single stage detector.

First introduced in 2015 by Redmon et al., their paper, [You Only Look Once: Unified, Real-Time Object Detection](#), details an object detector capable of super real-time object detection, obtaining **45 FPS** on a GPU.

Note: A smaller variant of their model called “Fast YOLO” claims to achieve 155 FPS on a GPU.

YOLO has gone through a number of different iterations, including [YOLO9000: Better, Faster, Stronger](#) (i.e., YOLOv2), capable of detecting over 9,000 object detectors.

Redmon and Farhadi are able to achieve such a large number of object detections by performing joint training for both object detection and classification. Using joint training the authors trained YOLO9000 simultaneously on both the ImageNet classification dataset and COCO detection dataset. The result is a YOLO model, called YOLO9000, that can predict detections for object classes that don’t have labeled detection data.

While interesting and novel, YOLOv2’s performance was a bit underwhelming given the title and abstract of the paper.

On the 156 class version of COCO, YOLO9000 achieved 16% mean Average Precision (mAP), and yes, while YOLO can detect 9,000 separate classes, the accuracy is not quite what we would desire.

Redmon and Farhadi recently published a new YOLO paper, [YOLOv3: An Incremental Improvement](#) (2018). YOLOv3 is significantly larger than previous models but is, in my opinion, the best one yet out of the YOLO family of object detectors

25. What are the drawbacks of YOLO

Limitations and drawbacks of the YOLO object detector

Arguably the largest limitation and drawback of the YOLO object detector is that:

1. It does not always handle small objects well
2. It *especially* does not handle objects grouped close together

The reason for this limitation is due to the YOLO algorithm itself:

- The YOLO object detector divides an input image into an $S \times S$ grid where each cell in the grid predicts only a single object.
- If there exist multiple, small objects in a single cell then YOLO will be unable to detect them, ultimately leading to missed object detections.

Therefore, if you know your dataset consists of many small objects grouped close together then you should not use the YOLO object detector.

In terms of small objects, Faster R-CNN tends to work the best; however, it's also the slowest.

SSDs can also be used here; however, SSDs can also struggle with smaller objects (but not as much as YOLO).

SSDs often give a nice tradeoff in terms of speed and accuracy as well.

It's also worth noting that YOLO ran slower than SSDs in this tutorial. In my previous tutorial on [OpenCV object detection](#) we utilized an SSD — a single forward pass of the SSD took ~0.03 seconds.

However, from this tutorial, we know that a forward pass of the YOLO object detector took ~0.3 seconds, *approximately an order of magnitude slower!*

If you're using the pre-trained deep learning object detectors OpenCV supplies you may want to consider using SSDs over YOLO. From my personal experience, I've rarely encountered situations where I needed to use YOLO over SSDs:

- I have found SSDs much easier to train and their performance in terms of accuracy almost always outperforms YOLO (at least for the datasets I've worked with).
- YOLO may have excellent results on the COCO dataset; however, I have not found that same level of accuracy for my own tasks.

I, therefore, tend to use the following guidelines when picking an object detector for a given problem:

1. If I know I need to detect small objects and speed is not a concern, I tend to use Faster R-CNN.
2. If speed is absolutely paramount, I use YOLO.
3. If I need a middle ground, I tend to go with SSDs.

In most of my situations I end up using SSDs or RetinaNet — both are a great balance between the YOLO/Faster R-CNN.

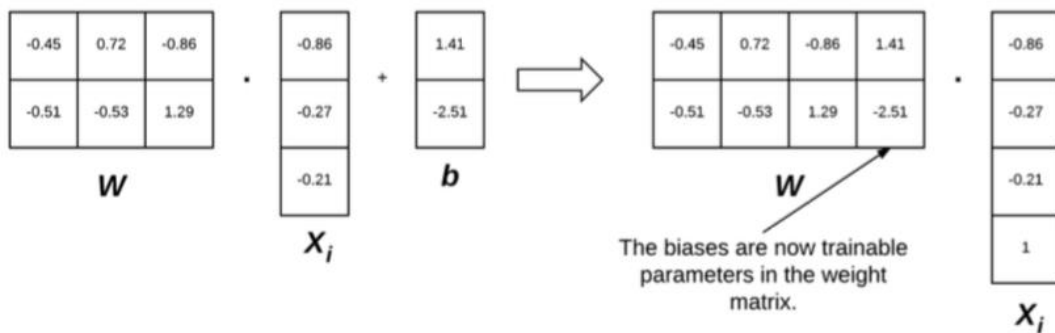
Starter Bundle

Chapter 9:

1. What is the bias trick?

It's a combination of the bias and the weight matrix by adding one extra dimension (i.e. column) to our input data X that holds a constant 1.

$$f(x_i, W, b) = Wx_i + b \rightarrow f(x_i, W) = Wx_i$$



In this way, we can treat the bias as a *learnable parameter within the weight matrix* that we don't have to explicitly keep track of in a separate variable.

2. Explain the gradient descent

Gradient descent is applied in order to find a W (weights) that yields minimal loss.

[SEP]

$$\frac{\partial}{\partial m} = \frac{2}{N} \sum_{i=1}^N -x_i(y_i - (mx_i + b))$$

$$\frac{\partial}{\partial b} = \frac{2}{N} \sum_{i=1}^N -(y_i - (mx_i + b))$$

```
def linear_regression(X, y, m_current=0, b_current=0, epochs=1000,
learning_rate=0.0001):
    N = float(len(y))
    for i in range(epochs):
        y_current = (m_current * X) + b_current
        cost = sum([data**2 for data in (y-y_current)]) / N
        m_gradient = -(2/N) * sum(X * (y - y_current))
        b_gradient = -(2/N) * sum(y - y_current)
        m_current = m_current - (learning_rate * m_gradient)
        b_current = b_current - (learning_rate * b_gradient)
    return m_current, b_current, cost
```

The gradient descent algorithm will run very slowly on large datasets. The reason for this “slowness” is because each iteration of gradient descent requires that we compute a prediction for each training point in our training data. It also turns out that computing predictions for every training data point before taking a step and updating our weight matrix W is computationally wasteful (and doesn’t help us in the long run). Instead, what we should do is **batch our updates -> SGD**.

3. Explain the stochastic gradient descent

a simple modification to the standard gradient descent algorithm that *computes the gradient* and *updates the weight matrix W* on small batches of training data, rather than the entire training set. While this modification leads to “more noisy” updates, it also allows us to take *more steps along the gradient* (one step per each batch versus one step per epoch), ultimately leading to faster convergence and no negative effects to loss and classification accuracy.

Below follows the pseudocode for vanilla gradient descent:

```
Vanilla gradient descent
1 while True:
2     Wgradient = evaluate_gradient(loss, data, W)
3     W += -alpha * Wgradient
```

And here we can see the pseudocode for Stochastic Gradient Descent:

```
Stochastic Gradient Descent (SGD)
1 while True:
2     batch = next_training_batch(data, 256)
3     Wgradient = evaluate_gradient(loss, batch, W)
4     W += -alpha * Wgradient
```

In general, the mini-batch size is not a hyperparameter that you should worry much about. You basically determine how many training examples will fit on your GPU/main memory and then use the nearest power of 2 as the batch size.

```

Stochastic Gradient Descent (SGD) with Python
47 # loop over the desired number of epochs
48 for epoch in np.arange(0, args["epochs"]):
49     # initialize the total loss for the epoch
50     epochLoss = []
51
52     # loop over our data in batches
53     for (batchX, batchY) in next_batch(X, y, args["batch_size"]):
54         # take the dot product between our current batch of
55         # features and weight matrix `W`, then pass this value
56         # through the sigmoid activation function
57         preds = sigmoid_activation(batchX.dot(W))
58
59         # now that we have our predictions, we need to determine
60         # our `error`, which is the difference between our predictions
61         # and the true values
62         error = preds - batchY
63
64         # given our `error`, we can compute the total loss value on
65         # the batch as the sum of squared loss
66         loss = np.sum(error ** 2)
67         epochLoss.append(loss)
68
69         # the gradient update is therefore the dot product between
70         # the transpose of our current batch and the error on the
71         # batch
72         gradient = batchX.T.dot(error) / batchX.shape[0]
73
74         # use the gradient computed on the current batch to take
75         # a "step" in the correct direction
76         W += -args["alpha"] * gradient
77
78     # update our loss history list by taking the average loss
79     # across all batches
80     lossHistory.append(np.average(epochLoss))

```

We then initialize an `epochLoss` list to store the loss value for *each* of the mini-batch gradient updates. As we'll see later in this code block, the `epochLoss` list will be used to compute the average loss over all mini-batch updates for an entire epoch.

Line 53 is the “core” of the Stochastic Gradient Descent algorithm and is what separates it from the vanilla gradient descent algorithm — *we loop over our training samples in mini-batches*.

For each of these mini-batches, we take the data, compute the dot product between it and the weight matrix, and then pass the results through the sigmoid activation function to obtain our predictions.

Investigating the actual loss values at the end of the 100th epoch, you'll notice that loss obtained by SGD is *nearly two orders of magnitude lower* than vanilla gradient descent (0.006 vs 0.447, respectively). This difference is due to the multiple weight updates per epoch, giving our model more chances to learn from the updates made to the weight matrix. This effect is even more pronounced on large datasets, such as ImageNet where we have millions of training examples and small, incremental

updates in our parameters can lead to a low loss (but not necessarily optimal) solution.

4. Explain momentum

Momentum just like Nesterov acceleration are extension to the standard SGD. Momentum is used to accelerate SGD, enabling it to learn faster by focusing on dimensions whose gradient point in the same direction.

As you travel down the hill, you build up more and more momentum, which in turn carries you faster down the hill.

Momentum builds upon the standard weight update to include a momentum term, thereby allowing our model to obtain lower loss (and higher accuracy) in less epochs. The momentum term should, therefore, *increase* the strength of updates for dimensions whose gradients point in the same direction and then *decrease* the strength of updates for dimensions whose gradients switch directions

5. Explain Nesterov acceleration

If we build up too much momentum, we may overshoot a local minimum and keep on rolling. Therefore, it would be advantageous to have a smarter roll, one that knows when to slow down, which is where Nesterov accelerated gradient comes in.

SEP



Figure 9.7: A graphical depiction of Nesterov acceleration. First, we make a big jump in the direction of the previous gradient, then measure the gradient where we ended up and make the correction.

Using standard momentum, we compute the gradient (small blue vector) and then take a big jump in the direction of the gradient (large blue vector). Under Nesterov acceleration we would first make a big jump in the direction of our *previous* gradient (brown vector), measure the gradient, and then make a *correction* (red vector) SEP

Recommendation:

As for Nesterov acceleration, I tend to use it on smaller datasets, but for larger datasets (such as ImageNet), I almost always avoid it. While Nesterov acceleration has sound theoretical guarantees, all major publications trained on ImageNet (e.g., AlexNet, VGGNet, ResNet, Inception, etc.) use SGD with momentum – *not a single paper from this seminal group utilizes Nesterov acceleration.* ^[LSEP]

6. What is the difference between adaptive learning methods to “classical” optimizers

With the latest incarnation of deep learning, there has been an explosion of new optimization techniques, each seeking to improve on SGD and provide the concept of *adaptive learning rates*. As we know, SGD modifies *all* parameters in a network *equally* in proportion to a given learning rate. However, given that the learning rate of a network is (1) the most important hyperparameter to tune and (2) a hard, tedious hyperparameter to set correctly, deep learning researchers have postulated that it’s possible to *adaptively* tune the learning rate (and in some cases, *per parameter*) as the network trains.

7. Explain Adagrad

Adagrad was first introduced by Duchi et al. Adagrad adapts the learning rate to the network parameters. Larger updates are performed on parameters that change infrequently while smaller updates are done on parameters that change frequently.

```
cache += (dW ** 2)
W += -lr * dW / (np.sqrt(cache) + eps)
```

The first parameter you’ll notice here is the cache – this variable maintains the per-parameter sum of squared gradients and is updated at *every mini-batch* in the training process.

We can then divide the $lr * dW$ by the square-root of the cache (adding in an epsilon value for smoothing and preventing division by zero errors). Scaling the update by all previous sum of square gradients allows us to adaptively update the parameters in our network.

Weights that have *frequently updated/large gradients* in the cache will scale the size of the update down, effectively *lowering* the learning rate for the parameter. On the other hand, weights that have *infrequent updates/smaller gradients* in the cache will scale up the size of the update, effectively *raising* the learning rate for the specific parameter.

Benefit:

The primary benefit of Adagrad is that we no longer have to manually tune the learning rate – most implementations of the Adagrad algorithm leave the initial

learning rate at 0.01 and allow the adaptive nature of the algorithm to tune the learning rate on a per-parameter basis.

Drawback:

However, the weakness of Adagrad can be seen by examining the cache. At each mini-batch, the squared gradients are accumulated in the denominator. Since the gradients are squared (and are therefore always positive), this accumulation keeps growing and growing during the training process. As we know, dividing a small number (the gradient) by a very large number (the cache) will result in an update that is infinitesimally small, too small for the network to actually learn anything in later epochs.

8. Explain adadelta

The Adadelta algorithm was proposed by Zeiler in their 2012 paper, *ADADELTA: An Adaptive Learning Rate Method*. Adadelta can be seen as an extension to Adagrad that seeks to reduce the monotonically decreasing learning rate caused by the cache. In the Adagrad algorithm, we update our cache with *all* of the previously squared gradients. However, Adadelta restricts this cache update by *only* accumulating a small number of past gradients – when actually implemented, this operation amounts to computing a *decaying average* of all past squared gradients.

9. Explain RMSProp

Similar to Adadelta, RMSprop attempts to rectify the negative effects of a globally accumulated cache by converting the cache into an exponentially weighted moving average.

```
cache = decay_rate * cache + (1 - decay_rate) * (dW ** 2)
W += -lr * dW / (np.sqrt(cache) + eps)
```

The first aspect of RMSprop you'll notice is that the actual update to the weight matrix W is identical to that of Adagrad – what matters here is how the cache is updated. The `decay_rate`, often defined as ρ , is a hyperparameter typically set to 0.9. Here we can see that previous entries in the cache will be weighted substantially smaller than new updates. This “moving average” aspect of RMSprop allows the cache to “leak out” old squared gradients and replace them with newer, “fresher” ones.

Again, the actual update to W is identical to that of Adagrad – the crux of the algorithm hinges on exponentially decaying the cache, enabling us to avoid monotonically decreasing learning rates during the training process. In practice, RMSprop tends to be more effective than both Adagrad and Adadelta when applied

to training a variety of deep learning networks. Furthermore, RMSprop tends to converge *significantly faster* than SGD

10. Explain Adam

The Adam (Adaptive Moment Estimation) optimization algorithm, proposed by Kingma and Ba in their 2014 paper, *Adam: A Method for Stochastic Optimization* [1] is essentially RMSprop only with momentum added to it:

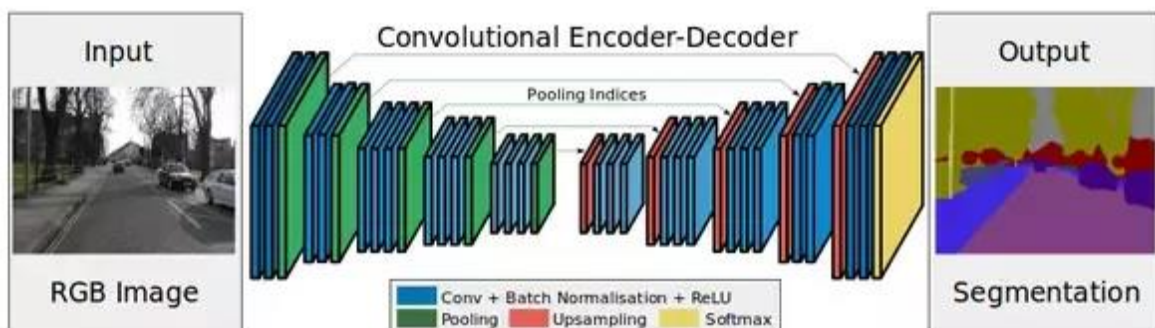
```
m = beta1 * m + (1 - beta1) * dW
v = beta2 * v + (1 - beta2) * (dW ** 2)
W += -lr * m / (np.sqrt(v) + eps)
```

The values of both m and v are similar to SGD momentum, relying on their respective previous values from time $t - 1$. The value m represents the first moment (mean) of the gradients while v is the second moment (variance).

The actual update to W is near identical to RMSprop, only now we are using the “smoothed” version (due to computing the mean) of m rather than the raw gradient dW – using the mean tends to lead to more desirable updates as we can smooth over noisy updates in the raw dW values. Typically β_1 is set to 0.9 while β_2 is set to 0.999 – these values are rarely (if ever) changed when using the Adam optimizer.

11. What is an encoder – decoder network

Some network architectures explicitly aim to leverage this ability of neural networks to learn efficient representations. They use an encoder network to map raw inputs to feature representations, and a decoder network to take this feature representation as input, process it to make its decision, and produce an output. This is called an encoder-decoder network.



This is a network to perform semantic segmentation of an image. The left half of the network maps raw image pixels to a rich representation of a collection of feature

vectors. The right half of the network takes these features, produces an output and maps the output back into the “raw” format (in this case, image pixels).

12. What types of regularization do you have

There are various types of regularization techniques, such as L1 regularization, L2 regularization (commonly called “weight decay”), and Elastic Net [103], that are used by updating the loss function itself, adding an additional parameter to constrain the capacity of the model.

We also have types of regularization that can be *explicitly* added to the network architecture – dropout is the quintessential example of such regularization. We then have *implicit* forms of regularization that are applied during the training process. Examples of implicit regularization include data augmentation and early stopping. Inside this section, we’ll mainly be focusing on the parameterized regularization obtained by modifying our loss and update functions.

Types of Regularization Techniques

In general, you’ll see three common types of regularization that are applied directly to the loss function. The first, we reviewed earlier, L2 regularization (aka “weight decay”):

$$R(W) = \sum_i \sum_j W_{i,j}^2 \quad (9.14)$$

We also have L1 regularization which takes the absolute value rather than the square:

$$R(W) = \sum_i \sum_j |W_{i,j}| \quad (9.15)$$

Elastic Net [103] regularization seeks to combine both L1 and L2 regularization:

$$R(W) = \sum_i \sum_j \beta W_{i,j}^2 + |W_{i,j}| \quad (9.16)$$

13. Why do we need regularization

Regularization helps us control our model capacity, ensuring that our models are better at making (correct) classifications on data points that they were *not* trained on, which we call *the ability to generalize*. If we don’t apply regularization, our classifiers can easily become too complex and *overfit* to our training data, in which case we lose the ability to generalize to our testing data (and data points *outside* the testing set as well, such as new images in the wild).

However, too much regularization can be a bad thing. We can run the risk of *underfitting*, in which case our model performs poorly on the training data and is not able to model the relationship between the input data and output class labels (because we limited model capacity too much).

14. How does regularization work

We loop over all entries in the matrix and taking the sum of squares. The sum of squares in the L2 regularization penalty discourages large weights in our matrix W , preferring smaller ones. Why might we want to discourage large weight values? In short, by penalizing large weights, we can improve the ability to generalize, and thereby reduce overfitting.

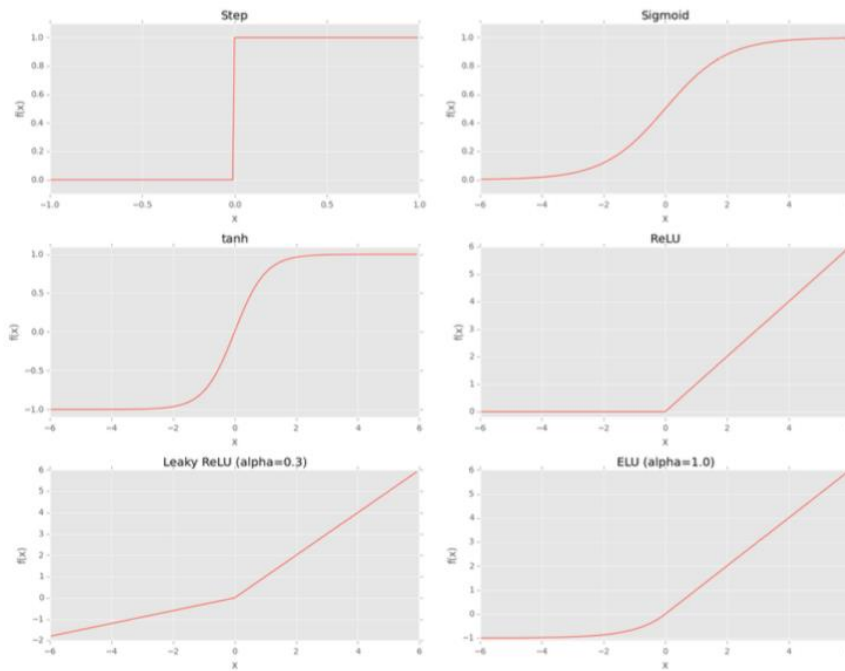
Think of it this way – the larger a weight value is, the more influence it has on the output prediction. Dimensions with larger weight values can almost singlehandedly control the output prediction of the classifier (provided the weight value is large enough, of course) which will almost certainly lead to overfitting

$$R(W) = \sum_i \sum_j W_{i,j}^2 \quad L = \frac{1}{N} \sum_{i=1}^N L_i + \lambda R(W)$$

L2 regularization (left) and embedded into our loss function, where L_i represents the loss function. L computes average loss over all samples in our training set complemented by

Chapter 10

15. What types of activation functions do we have



The sigmoid function

$$t = \sum_{i=1}^n w_i x_i \quad s(t) = 1 / (1 + e^{-t})$$

The sigmoid function is a better choice for learning than the simple step function since it: 1. Is continuous and differentiable everywhere. 2. Is symmetric around the y-axis. 3. Asymptotically approaches its saturation values.

The primary advantage here is that the smoothness of the sigmoid function makes it easier to devise learning algorithms. However, there are *two big problems* with the sigmoid function:

1. The outputs of the sigmoid are not zero centered.
2. Saturated neurons essentially kill the gradient, since the delta of the gradient will be extremely small.

The tanh function

Similar to sigmoid

ReLU

The ReLU function is not saturable and is also extremely computationally efficient. Empirically, the ReLU activation function tends to outperform *both* the sigmoid and *tanh* functions in nearly all applications.

However, a problem arises when we have a value of zero – *the gradient cannot be taken*. A variant of ReLUs, called *Leaky ReLUs* [108] allow for a small, non-zero gradient when the unit is not active.

Other ReLU forms

Parametric ReLUs, or PReLUs for short [101], build on Leaky ReLUs and allow the parameter α to be learned on an activation-by-activation basis, implying that each node in the network can learn a different “coefficient of leakage” separate from the other nodes.

ELU

The value of α is constant and *set when the network architecture is instantiated* – this is unlike PReLUs where α is learned. A typical value for α is $\alpha = 1.0$. Through the work of Clevert et al. (and my own anecdotal experiments), ELUs often obtain higher classification accuracy than ReLUs. ELUs rarely, if ever perform worse than your standard ReLU function.

16. What activation function would you use

My personal preference is to start with a ReLU, tune my network and optimizer parameters (architecture, learning rate, regularization strength, etc.) and note the accuracy. Once I am reasonably satisfied with the accuracy, I swap in an ELU and often notice a 1 – 5% improvement in classification accuracy depending on the dataset.

17. What are the “ingredients”

dataset
loss function
model
optimization method

18. Explain various weight initialization method

Glorot/Xavier Uniform and Normal

For the normal distribution the limit value is constructed by *averaging* the F_{in} and F_{out} together and then taking the square-root. A zero-center ($\mu = 0$) is then used:

```
>>> F_in = 64
>>> F_out = 32
>>> limit = np.sqrt(2 / float(F_in + F_out))
>>> W = np.random.normal(0.0, limit, size=(F_in, F_out))
```

Glorot/Xavier initialization can also be done with a uniform distribution where we place stronger restrictions on limit:

```
>>> F_in = 64
>>> F_out = 32
>>> limit = np.sqrt(6 / float(F_in + F_out))
>>> W = np.random.uniform(low=-limit, high=limit, size=(F_in, F_out))
```

This initialization method is recommend for most neural networks.

He et al./Kaiming/MSRA Uniform and Normal ^[L]_{SEP}

We typically used this method when we are training *very deep* neural networks that use a ReLU-like activation function (in particular, a “PReLU”, or Parametric Rectified Linear Unit). ^[L]_{SEP} To initialize the weights in a layer using He et al. initialization with a *uniform distribution* we set limit to be $limit = 6/F_{in}$, where F_{in} is the number of input units in the layer: ^[L]_{SEP}

```
>>> F_in = 64
>>> F_out = 32
>>> limit = np.sqrt(6 / float(F_in))
>>> W = np.random.uniform(low=-limit, high=limit, size=(F_in, F_out))
```

We can also use a *normal distribution* as well by setting $\mu = 0$ and $\sigma = \sqrt{2/F_{in}}$

```
>>> F_in = 64
>>> F_out = 32
>>> limit = np.sqrt(2 / float(F_in))
>>> W = np.random.normal(0.0, limit, size=(F_in, F_out))
```

Chapter 11

Practitioner bundle

ImageNet Bundle

Chapter 1:

19. What are the advantages of mxnet library?

It is really tedious and non-trivial to setup multiple GPUs in TensorFlow. The mxnet deep learning library (written in C++) provides bindings to the Python programming language and specializes in distributed, multi-machine learning.

20. How do multiple GPU scale?

Training performance is *heavily* dependent on the PCIe bus on your system, the specific architecture you are training, the number of layers in the network, and whether your network is bound via *computation* or *communication*. In general, training with two GPUs tends to improve speed by $\approx 1.8x$. When using four GPUs, performance scales to $\approx 2.5 - 3.5x$ scaling depending on your system.

Chapter 16

21. Drawback of R-CNN

The Region Proposal Network (RPN) needed to be trained in order to generate suggested bounding boxes before we could train the actual classifier to recognize objects in images. The problem was later mitigated by training the entire R-CNN architecture end-to-end, but prior to this discovery, it introduced a tedious pre-training process.

The second issue is that training took too long. A (Faster) R-CNN consists of multiple components, including:

1. A Region Proposal Network
2. A ROI Pooling Module^[L, b, SEP]
3. The final classifier

While all three fit together into a framework, they are still moving parts that slow down the entire training procedure.

22. Explain SSD

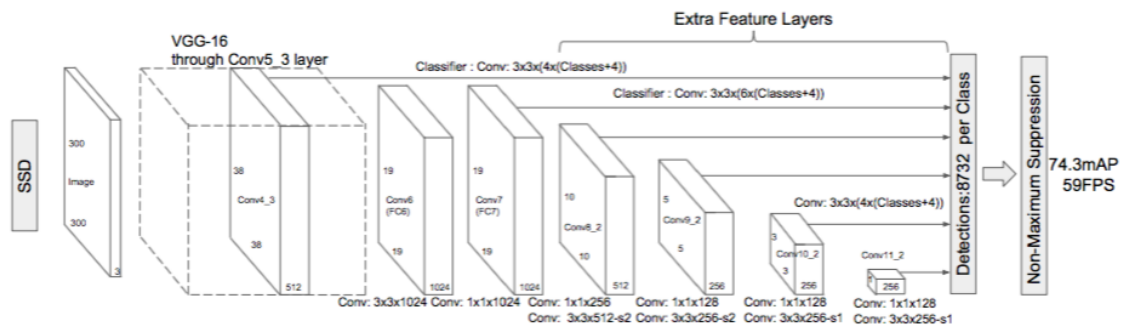
First introduced by Liu et al. in their 2015 paper, *SSD: Single Shot MultiBox Detector*. The Multibox component, used for the bounding box regression algorithm, comes from Szegedy (the same Christian Szegedy of Google's Inception network) et al.'s 2015 paper, *Scalable High Quality Object Detection*

Motivation

- The term **Single Shot** implies that both *localization* and *detection* are performed in a single forward pass of the network during inference time — the network only has to “look” at the image once to obtain final predictions.
- The term **Multibox** refers to Szegedy et al.'s original Multibox algorithm used for bounding box regression [59]. This algorithm enables SSDs to localize objects of different classes, even if their bounding boxes overlap. In many object detection algorithms, overlapping objects of different classes are often suppressed into a single bounding box with the highest confidence.

- Finally, the term **Detector** implies that we'll be not only localizing the (x, y) -coordinates of a set of objects in an image, but also returning their class labels as well.

As we'll see, and according to Liu et al, the fundamental improvement in speed of SSDs comes from eliminating bounding box proposals and subsampling of pixels or features.



Architecture

- SSD starts with a base network. This network is typically pre-trained, normally on a large dataset such as ImageNet, enabling it to learn a rich set of discriminative features. Again, we'll use this network for transfer learning, propagating an input image to a pre-specified layer, obtaining the feature map, and then moving forward to the object detection layers
- We utilize the VGG layers up until conv_6 and then detach all other layers, including the fully-connected layers. A set of new CONV layers are then added to the architecture — these are the layers that make the SSD framework possible.
- Two important components:
 - We progressively reduce the volume size in deeper layers, as we would with a standard CNN
 - Each of the CONV layers connects to the final detection layer.

The fact that each feature map connects to the final detection layer is important — it allows the network to detect and localize objects in images at varying scales. Furthermore, this scale localization happens in a forward pass. No resample of feature maps is required, enabling SSDs to operate in an entire feedforward manner — this fact is what makes SSDs so fast and efficient.

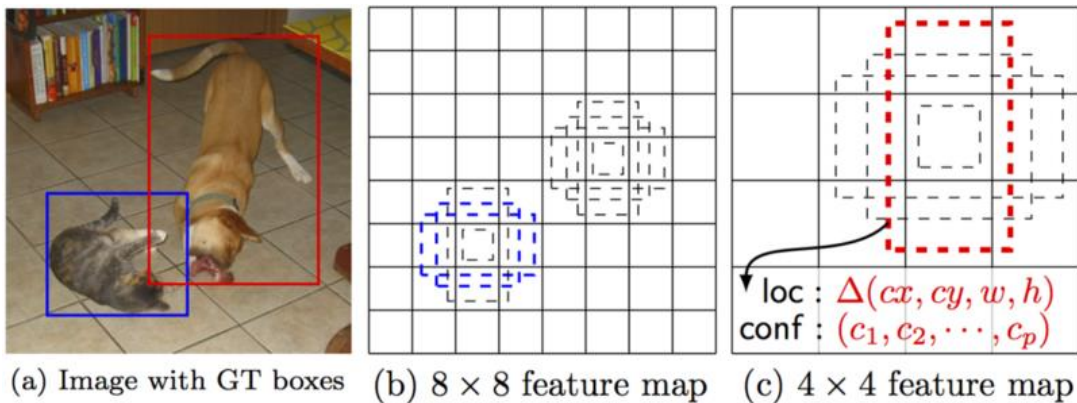
MultiBox, Priors, and Fixed Priors

- unlike MultiBox, every feature map cell is associated with a set of default bounding boxes of different dimensions and aspect ratios. These priors are manually (but carefully) chosen, whereas in MultiBox, they were chosen because their IoU with respect to the ground truth was over 0.5. This in theory should

allow SSD to generalise for any type of input, without requiring a pre-training phase for prior generation.

Training and running an SSD

- **Location Loss:** SSD uses [smooth L1-Norm](#) to calculate the location loss. While not as precise as L2-Norm, it is still highly effective and gives SSD more room for manoeuvre as it does not try to be “pixel perfect” in its bounding box prediction
- **Classification:** MultiBox does not perform object classification, whereas SSD does. Therefore, for each predicted bounding box, a set of c class predictions are computed, for every possible class in the dataset
- **Default Bounding Boxes:** It is recommended to configure a varied set of default bounding boxes, of different scales and aspect ratios to ensure most objects could be captured. The SSD paper has around 6 bounding boxes per feature map cell.



- **Features maps** (i.e. the results of the convolutional blocks) are a representation of the dominant features of the image at different scales, therefore running MultiBox on multiple feature maps increases the likelihood of any object (large and small) to be eventually detected, localized and appropriately classified.
- Non-max –suppression is then used to detect the correct bounding box
- **Hard negative mining:** During training, as most of the bounding boxes will have low IoU and therefore be interpreted as *negative* training examples, we may end up with a disproportionate amount of negative examples in our training set. Therefore, instead of using all negative predictions, it is advised to keep a ratio of negative to positive examples of around 3:1. The reason why you need to keep negative samples is because the network also needs to learn and be explicitly told what constitutes an incorrect detection.

Additional notes:

- more default boxes results in more accurate detection, although there is an impact on speed

- 80% of the time is spent on the base VGG-16 network: this means that with a faster and equally accurate network SSD's performance could be even better
- SSD produces worse performance on smaller objects, as they may not appear across all feature maps. Increasing the input image resolution alleviates this problem but does not completely address it

23. Explain non-maximum suppression (NMS)


Two variations:

Slow method done by Felzenszwalb et al. and the [Malisiewicz et al. method](#) which is over *100x faster*.

- if the bounding boxes integers, convert them to floats -- this is important since we'll be doing a bunch of divisions

According to Andrew NG:

Non-max suppression algorithm



Each output prediction is:

p_c
 b_x
 b_y
 b_h
 b_w

Discard all boxes with $p_c \leq 0.6$

→ While there are any remaining boxes:

- Pick the box with the largest p_c
Output that as a prediction.
- Discard any remaining box with $\text{IoU} \geq 0.5$ with the box output in the previous step

Andrew Ng

24. What is the difference between YOLO and SSD

Questions

25. Explain transfer learning

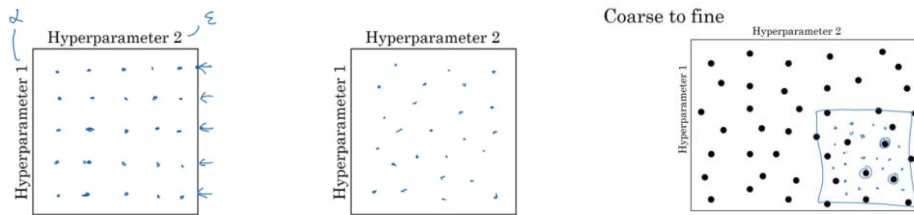
Check udacity page for explanation

26. What hyperparameters do you tune in a network

- learning rate
- momentum term beta
- adam hyperparameters beta1 beta2 epsilon
- number of layers
- number of hidden units
- learning rate decay

27. How do you search for optimal parameters

Try random values: Don't use a grid



Don't use a grid search, rely on random search instead. In a second step, you can focus on the region, which was giving good results and perform more exhaustive search there.

28. How many parameters does a CNN need

Setup

H = height, W = width, D = depth

- We have an input of shape 32x32x3 (HxWxD)
- 20 filters of shape 8x8x3 (HxWxD)
- A stride of 2 for both the height and width (S)
- Zero padding of size 1 (P)

Output Layer

- 14x14x20 (HxWxD)

Without parameter sharing

$$(8 * 8 * 3 + 1) * (14 * 14 * 20) = 756560$$

$8 * 8 * 3$ is the number of weights, we add 1 for the bias. Remember, each weight is assigned to every single part of the output ($14 * 14 * 20$). So we multiply these two numbers together and we get the final answer.

With parameter sharing

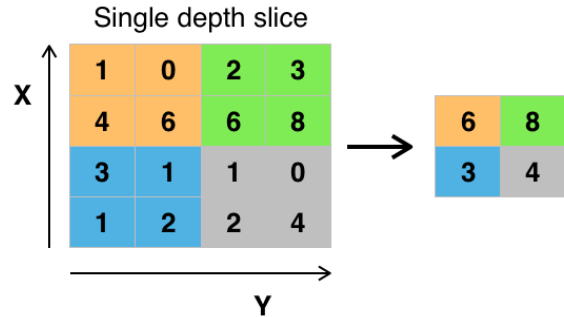
$$(8 * 8 * 3 + 1) * 20 = 3840 + 20 = 3860$$

That's 3840 weights and 20 biases. This should look similar to the answer from the previous quiz. The difference being it's just 20 instead of ($14 * 14 * 20$). Remember,

with weight sharing we use the same filter for an entire depth slice. Because of this we can get rid of $14 * 14$ and be left with only 20.

29. Explain max pooling in a CNN

Conceptually, the benefit of the max pooling operation is to reduce the size of the input, and allow the neural network to focus on only the most important elements. Max pooling does this by only retaining the maximum value for each filtered area, and removing the remaining values.



NOTE: For a pooling layer the output depth is the same as the input depth. Additionally, the pooling operation is applied individually for each depth slice.

```
new_height = (input_height - filter_height)/S + 1
new_width = (input_width - filter_width)/S + 1
```

30. What is pooling used for

- Prevent overfitting
- Decrease the size of the output

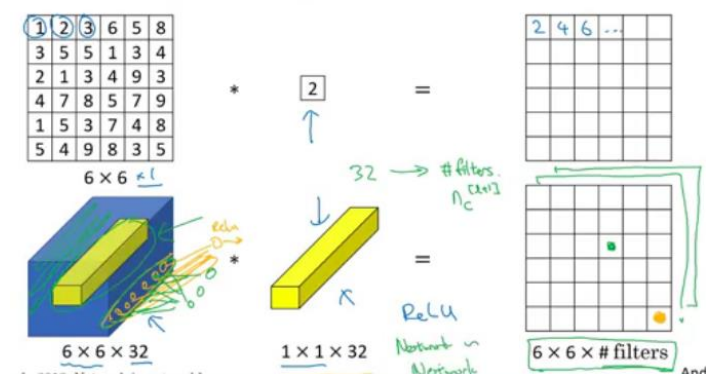
The correct answer is **decrease the size of the output** and **prevent overfitting**. Preventing overfitting is a consequence of reducing the output size, which in turn, reduces the number of parameters in future layers.

Recently, pooling layers have fallen out of favor. Some reasons are:

- Recent datasets are so big and complex we're more concerned about underfitting.
- Dropout is a much better regularizer.
- Pooling results in a loss of information. Think about the max pooling operation as an example. We only keep the largest of n numbers, thereby disregarding $n-1$ numbers completely.

31. What is 1x1 convolution good for

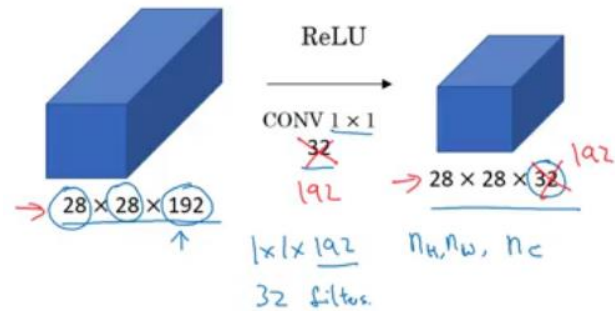
One-by-one convolution can either introduce non-linearity to the model



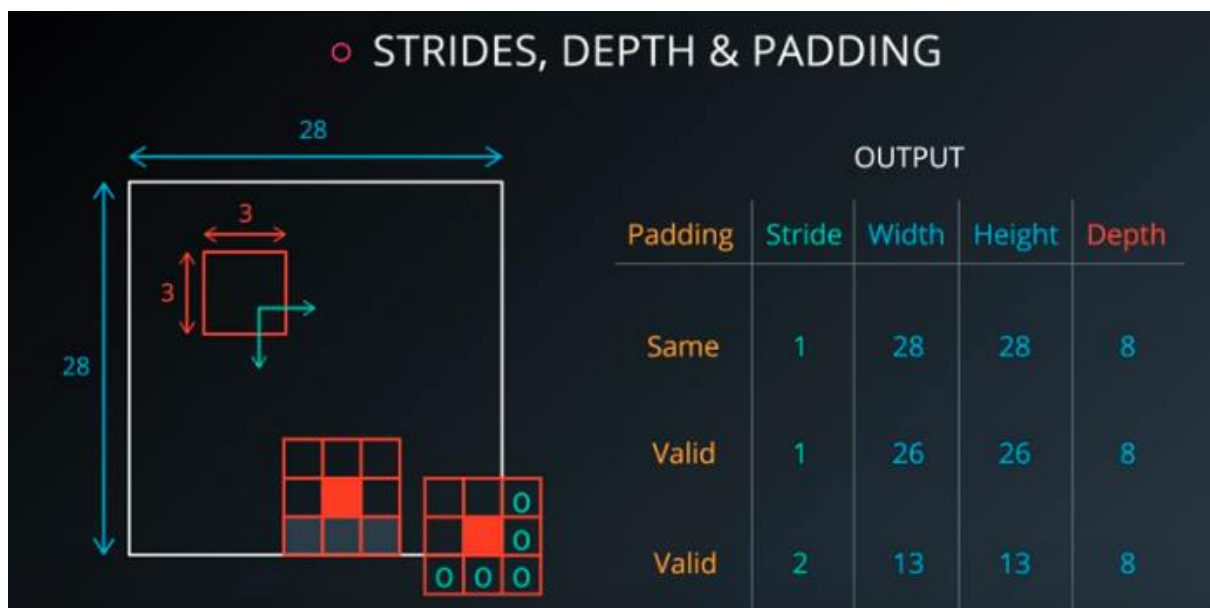
And by doing this at each of the 36 positions, each of the six by six positions, you end up with an output that is six by six by the number of filters. And this can carry out a pretty non-trivial computation on your input volume.

32. Explain padding types

```
new_height = (input_height - filter_height) / stride + 1
new_width = (input_width - filter_width) / stride + 1
```



33. Explain filters



```
new_height = (input_height - filter_height) / S + 1
new_width = (input_width - filter_width) / S + 1
```

34. Define evaluation metrics

Classification Accuracy

Classification Accuracy is what we usually mean, when we use the term accuracy. It is the ratio of number of correct predictions to the total number of input samples.

$$\text{Accuracy} = \frac{\text{Number of Correct predictions}}{\text{Total number of predictions made}}$$

The real problem arises, when the cost of misclassification of the minor class samples are very high. If we deal with a rare but fatal disease, the cost of failing to diagnose the disease of a sick person is much higher than the cost of sending a healthy person to more tests.

Confusion Matrix

		Actual class	
		Cat	Non-cat
Predicted class	Cat	5 True Positives	2 False Positives
	Non-cat	3 False Negatives	17 True Negatives

Accuracy for the matrix can be calculated by taking average of the values lying across the “main diagonal” i.e

Area under Curve

Before defining AUC, let us understand two basic terms :

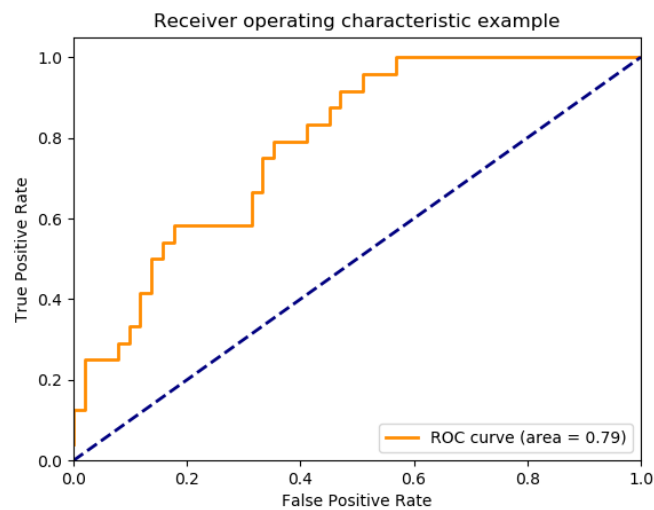
- **True Positive Rate (Sensitivity)** : True Positive Rate is defined as $TP / (FN + TP)$. True Positive Rate corresponds to the proportion of positive data points that are correctly considered as positive, with respect to all positive data points.

$$\text{TruePositiveRate} = \frac{\text{TruePositive}}{\text{FalseNegative} + \text{TruePositive}}$$

- **False Positive Rate (Specificity)** : False Positive Rate is defined as $FP / (FP + TN)$. False Positive Rate corresponds to the proportion of negative data points that are mistakenly considered as positive, with respect to all negative data points.

$$FalsePositiveRate = \frac{FalsePositive}{FalsePositive + TrueNegative}$$

False Positive Rate and True Positive Rate both have values in the range [0, 1]. FPR and TPR both are computed at threshold values such as (0.00, 0.02, 0.04, ..., 1.00) and a graph is drawn. AUC is the area under the curve of plot False Positive Rate vs True Positive Rate at different points in [0, 1].



F1 Score

F1 Score is used to measure a test's accuracy

F1 Score is the Harmonic Mean between precision and recall. The range for F1 Score is [0, 1]. It tells you how precise your classifier is (how many instances it classifies correctly), as well as how robust it is (it does not miss a significant number of instances).

High precision but lower recall, gives you an extremely accurate, but it then misses a large number of instances that are difficult to classify. The greater the F1 Score, the better is the performance of our model. Mathematically, it can be expressed as:

$$F1 = 2 * \frac{1}{\frac{1}{precision} + \frac{1}{recall}}$$

F1 Score tries to find the balance between precision and recall.

- **Precision** : It is the number of correct positive results divided by the number of positive results predicted by the classifier.
- **Recall** : It is the number of correct positive results divided by the number of **all** relevant samples (all samples that should have been identified as positive).

$$\text{Precision} = \frac{\text{True Positive}}{\text{True Positive} + \text{False Positive}}$$

$$\text{Recall} = \frac{\text{True Positive}}{\text{True Positive} + \text{False Negative}}$$

Mean Squared Error

Mean Squared Error(MSE) is quite similar to Mean Absolute Error, the only difference being that MSE takes the average of the **square** of the difference between the original values and the predicted values. The advantage of MSE being that it is easier to compute the gradient, whereas Mean Absolute Error requires complicated linear programming tools to compute the gradient. As, we take square of the error, the effect of larger errors become more pronounced than smaller error, hence the model can now focus more on the larger errors.

$$\text{Mean Squared Error} = \frac{1}{N} \sum_{j=1}^N (y_j - \hat{y}_j)^2$$

35. What is fine tuning

Fine-tuning is a type of transfer learning. We apply fine-tuning to deep learning models that have *already* been trained on a given dataset. Typically, these networks are state-of-the-art architectures such as VGG, ResNet, and Inception that have been trained on the ImageNet dataset.

36. Define a model in Keras


```
# Setup Keras
from keras.models import Sequential
from keras.layers.core import Dense, Activation, Flatten, Dropout
from keras.layers.convolutional import Conv2D
from keras.layers.pooling import MaxPooling2D
```

```
# Build the Final Test Neural Network in Keras Here
model = Sequential()
model.add(Conv2D(32, (3, 3), input_shape=(32, 32, 3)))
model.add(MaxPooling2D((2, 2)))
model.add(Dropout(0.5))
model.add(Activation('relu'))
model.add(Flatten())
model.add(Dense(128))
model.add(Activation('relu'))
model.add(Dense(5))
model.add(Activation('softmax'))
```

```
# preprocess data
X_normalized = np.array(X_train / 255.0 - 0.5 )

from sklearn.preprocessing import LabelBinarizer
label_binarizer = LabelBinarizer()
y_one_hot = label_binarizer.fit_transform(y_train)
```

```
# compile and fit the model
model.compile('adam', 'categorical_crossentropy', ['accuracy'])
history = model.fit(X_normalized, y_one_hot, epochs=10, validation_split=0.
```

```
# evaluate model against the test data
with open('small_test_traffic.p', 'rb') as f:
    data_test = pickle.load(f)

X_test = data_test['features']
y_test = data_test['labels']

# preprocess data
X_normalized_test = np.array(X_test / 255.0 - 0.5 )
y_one_hot_test = label_binarizer.fit_transform(y_test)

print("Testing")

metrics = model.evaluate(X_normalized_test, y_one_hot_test)
for metric_i in range(len(model.metrics_names)):
    metric_name = model.metrics_names[metric_i]
    metric_value = metrics[metric_i]
    print('{:}: {}'.format(metric_name, metric_value))
```

37. What is ROS

middleware = software für den Datenaustausch zwischen Anwendungsprogrammen, die unter verschiedenen Betriebssystemen oder in heterogenen Netzen arbeiten

Robot Operating System (ROS) is [robotics middleware](#) (i.e. collection of [software frameworks](#) for [robot](#) software development). Although ROS is not an [operating system](#), it provides services designed for a heterogeneous [computer cluster](#) such as [hardware abstraction](#), low-level [device control](#), implementation of commonly used functionality, [message-passing between processes](#), and package management. Running sets of ROS-based processes are represented in a [graph](#) architecture where processing takes place in nodes that may receive, post and multiplex sensor, control, state, planning, actuator, and other messages. Despite the importance of reactivity and [low latency](#) in robot control, ROS itself is *not* a [real-time OS](#) (RTOS). It is possible, however, to integrate ROS with real-time code.^[2] The lack of support for real-time systems has been addressed in the creation of ROS 2.0

38. What udacity projects did you work on

Sensor fusion with extended kalman filter

Localization

Localization is how we determine where our vehicle is in the world. GPS is great, but it's only accurate to within a few meters. We need single-digit centimeter-level accuracy! To achieve this, Mercedes-Benz engineers will demonstrate the principles of Markov localization to program a particle filter, which uses data and a map to determine the precise location of a vehicle.

Planning

The Mercedes-Benz team will take you through the three stages of planning. First, you'll apply model-driven and data-driven approaches to predict how other vehicles on the road will behave. Then you'll construct a finite state machine to decide which of several maneuvers your own vehicle should undertake. Finally, you'll generate a safe and comfortable trajectory to execute that maneuver.

Control

Ultimately, a self-driving car is still a car, and we need to send steering, acceleration, and brake commands to move the car through the world. Uber ATG will walk you through building a proportional-integral-derivative (PID) controller to actuate the vehicle.

System Integration

This is capstone of the entire Self-Driving Car Engineer Nanodegree Program! We'll introduce Carla, the Udacity self-driving car, and the Robot Operating System that controls her. You'll work with a team of other Nanodegree students to combine what you've learned over the course of the entire Nanodegree Program to drive Carla, a real self-driving car, around the Udacity test track!

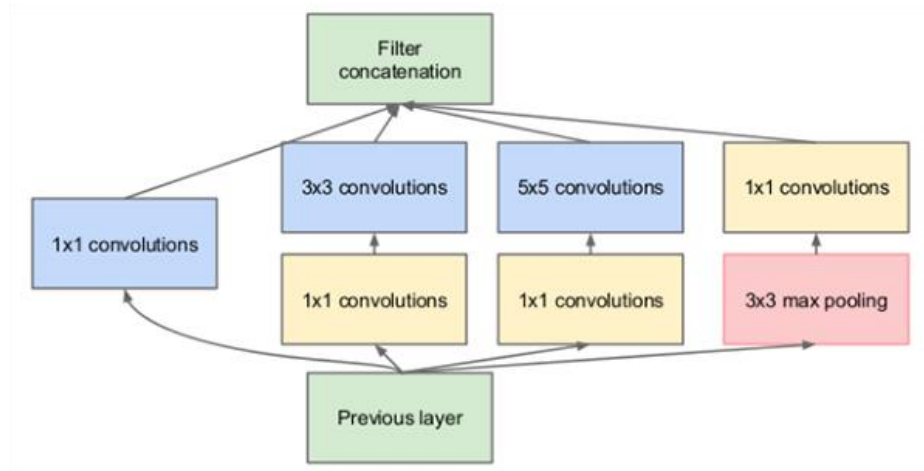
39. What is catkin

[catkin](#) is the official build system of ROS and the successor to the original ROS build system, [roscpp](#). [catkin](#) combines [CMake](#) macros and Python scripts to provide

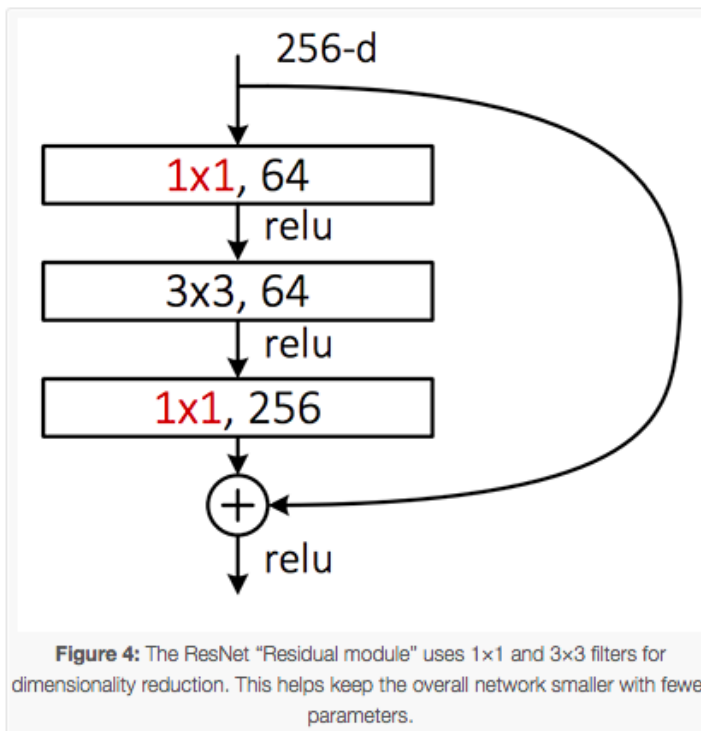
some functionality on top of CMake's normal workflow. [catkin](#) was designed to be more conventional than [roscmake](#), allowing for better distribution of packages, better cross-compiling support, and better portability. [catkin](#)'s workflow is very similar to [CMake](#)'s but adds support for automatic 'find package' infrastructure and building multiple, dependent projects at the same time.

40. Explain over and underfitting

41. describe inception module

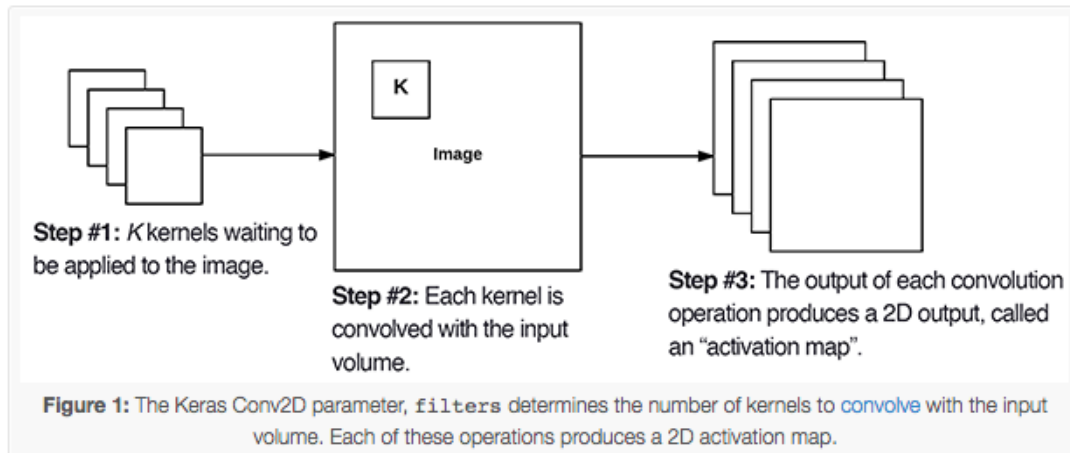


Describe the ResNet residual module



42. Keras Conv2D and Convolutional Layers

Filters



Notice at as our output spatial volume is decreasing our number of filters learned is increasing — this is a common practice in designing CNN architectures and one I recommend you do as well. As far as choosing the appropriate number of filters, I nearly always recommend using powers of 2 as the values.

You may need to tune the exact value depending on (1) the complexity of your dataset and (2) the depth of your neural network, but I recommend starting with filters in the range [32, 64, 128] in the earlier and increasing up to [256, 512, 1024] in the deeper layers.

Kernel size

If your input images are greater than 128×128 you may choose to use a kernel size > 3 to help (1) learn larger spatial filters and (2) to help reduce volume size.

Other networks, such as VGGNet, *exclusively* use (3, 3) filters throughout the entire network.

So how should you apply your `filter_size`?

First, examine your input image — is it larger than 128×128 ?

If so, consider using a 5×5 or 7×7 kernel to learn larger features and then quickly reduce spatial dimensions — then start working with 3×3 kernels:

If your images are smaller than 128×128 you may want to consider sticking with strictly 1×1 and 3×3 filters.

Strides

Typically you'll leave the `strides` parameter with the default `(1, 1)` value; however, you may occasionally increase it to `(2, 2)` to help reduce the size of the output volume (since the step size of the filter is larger).

Typically you'll see strides of 2×2 as a replacement to max pooling

In 2014, Springenberg et al. published a paper entitled [Striving for Simplicity: The All Convolutional Net](#) which demonstrated that replacing pooling layers with strided convolutions can increase accuracy in some situations.

Padding

With the `valid` parameter the input volume is not zero-padded and the spatial dimensions are allowed to reduce via the natural application of convolution.

If you instead want to preserve the spatial dimensions of the volume such that the output volume size matches the input volume size, then you would want to supply a value of `same` for the padding

While the default Keras `Conv2D` value is `valid` I will typically set it to `same` for the majority of the layers in my network and then either reduce spatial dimensions of my volume by either:

1. Max pooling
2. Strided convolution

I would recommend that you use a similar approach to padding with the Keras `Conv2D` class as well

Kernel initializer

The `kernel_initializer` defaults to `glorot_uniform`, the [Xavier Glorot uniform initialization](#) method, which is perfectly fine for the majority of tasks; however, for deeper neural networks you may want to use `he_normal` ([MSRA/He et al. initialization](#)) which works especially well when your network has a large number of parameters (i.e., VGGNet).

Kernel regularizer

Applying regularization helps you to:

1. *Reduce* the effects of overfitting
2. *Increase* the ability of your model to generalize

When working with large datasets and deep neural networks applying regularization is typically a *must*.

Normally you'll encounter either L1 or L2 regularization being applied — I will use L2 regularization on my networks if I detect signs of overfitting:

```
Keras Conv2D and Convolutional Layers
1 from keras.regularizers import l2
2 ...
3 model.add(Conv2D(32, (3, 3), activation="relu",
4     kernel_regularizer=l2(0.0005))
```

The amount of regularization you apply is a hyperparameter you will need to tune for your own dataset, but I find values of 0.0001-0.001 are good ranges to start with.

pyramidal?

dilated CNN?

transposed convolution?

masked R-cnn

Parallel programming (CPU vs GPU)

what is the retinanet architecture

draw diagrams from detector and classifier

draw digrams of networks

fully connected layer

valid vs same convolution

How to optimize a neural network

What are the architecture of state of the art DNNs

what is a linear hog classifier

How do you compute gradient descent

what are the differences between optimizers

What is the difference between L1 and L2 regularization

What parameters can you tune in your NN]

Adress cross entropy

How do you train a neural network_ How does backpropagation work_

How do you initialize multiple GPUs with tensorflow

how to you approach image stacks with cnns?

Checkout different network architectures – inception module ()

Checkou NGs lecture on cross-entropy loss

what is protobuf | what is prototxt | how can we deploy a network

What is a Haar cascades

Check also the udacity part II course

Yolo architecture

mobilenet-ssd architecture – detector+classifier?

How did I use Jenkins

transfer learning

retina model

my own

classical approach

Here *epochs* refer to the number of times we train our model to find the best slope and bias for our model to fit the data. Finally, *learning_rate* here refers to the speed of convergence, meaning how fast gradient descent finds the best parameters.

Projects for Interview

Cell recognition and tracking

2D time series with a resolution of 64x64. Because of other applications it was of uttermost importance to detect all cells and build a mask for it. This mask was then applied to the stack where all cells were given a cell type and the cell bodies were computed. Due to the functional nature of the data, the cell type identification was able to be done after the recording.

First, I implemented centroid, then I exchanged it for better one | Usefull if tissue is slowly drifting to a side.

recognition

- type of data

- what I was classifying

- what model

 - classical vs custom vs transfer learning

 - for transfer learning what network – check when the network was published

 - evaluation metrics

tracking

when was dlib implemented

- what types of tracking are there

how did I tracked the cells
possible problems

Traffic counter

- 2 recognize also the type of car - PKW vs LKW
- 2 in order to track two directions use a centroid tracker as well, which will get CentroidTracker instantiation to accept the list of rects , regardless of whether they were generated via object detection or object tracking. Our centroid tracker will associate object IDs with object locations.
- 2 possibilty for different mounting positions

recognition
mobilenetSSD

tracking

problems
car on car or a truck with many cars

Traffic sign recognition

problems with smaller object pyramids

- data acquisition
- deep learning
 - o rewrote opencv
 - o implemented a usecase with ssd, tested with others as well
 - o pyramidal networks
- integration in the car
 - o network deployment
 - o tensorrt
 - o protobuf

jenkins

caffe - The path to the Caffe “deploy” prototxt file.

- o
- o how did you deploy everything
- fusion
- ASIL
 - o what iso?
 - o unit test
 - what unit test
- SOTIF

Smart home systems

Security system

implemented for second classifier for rubbish bin and dhl? It would trigger an email

– then how would you send an email?

transfer learning

what network type

what about resources for pi

how does raspbian and pycharm work?

how do you send one image to the other pi?

dropbox api included

Amazon's Alexa

what type of network

how does lstm work

Evaluation metrics!!!

Augmentation

Notes

- your preprocessing steps for training and validation must be *identical* to the training steps when loading your model and classifying new images.

How do you load and save your Keras models

Note: your preprocessing steps for training and validation must be *identical* to the training steps when loading your model and classifying new images.

These threads will run independently without stopping the forward execution of the script (i.e., a *non-blocking* operation).