Data Augmentation

- type of regularization / reduces overfitting
- augments data without changing the class label
- only applied to training set, NOT validation data!
- randomly jitters data
  - translation
  - rotation
  - flipping
  - shearing

Flowers-17

- 17 distinct species of flowers
- overfits b/c
  - not enough data (only 60 samples per class)

Training CNNs

- need 1000 - 5000 examples per class

Transfer learning / Feature Extraction

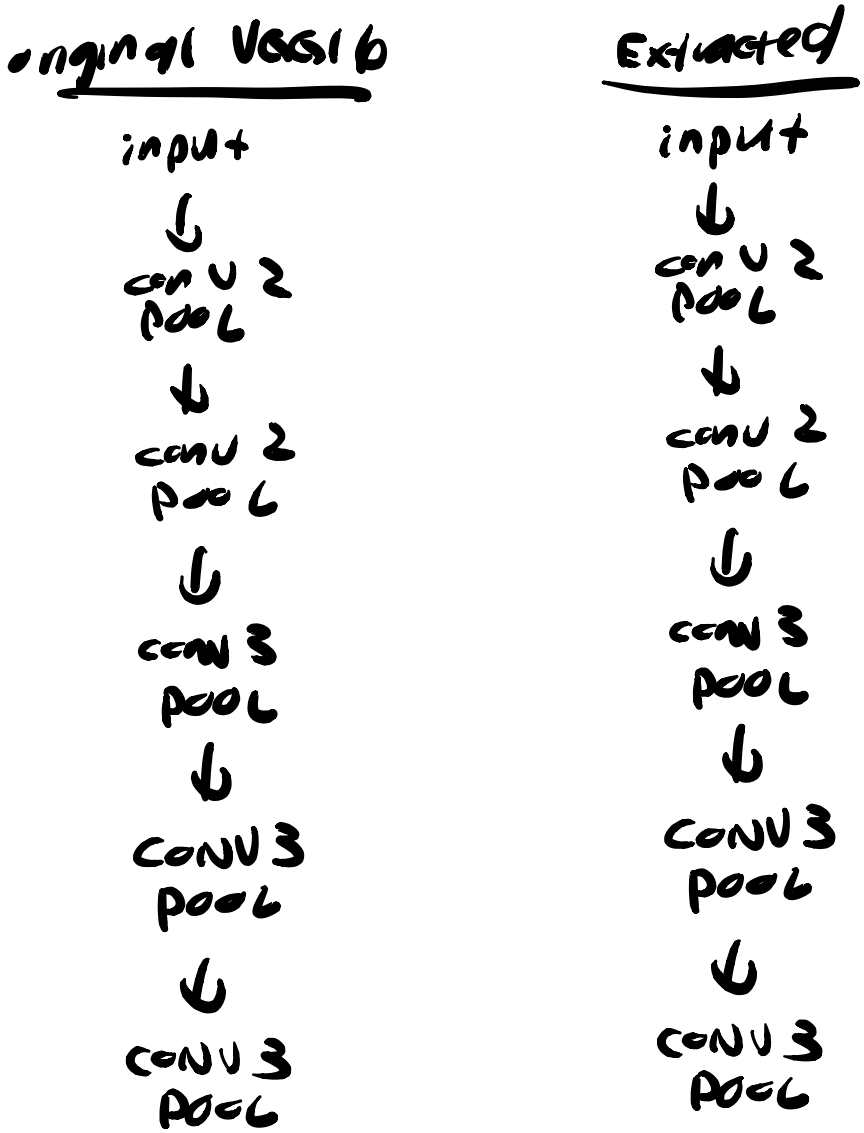
- use a pre-trained model as a "shortcut" to learn patterns in data not originally trained on
- Types
  1. Treating networks as arbitrary feature extractors
  2. Removing FC layers of existing network

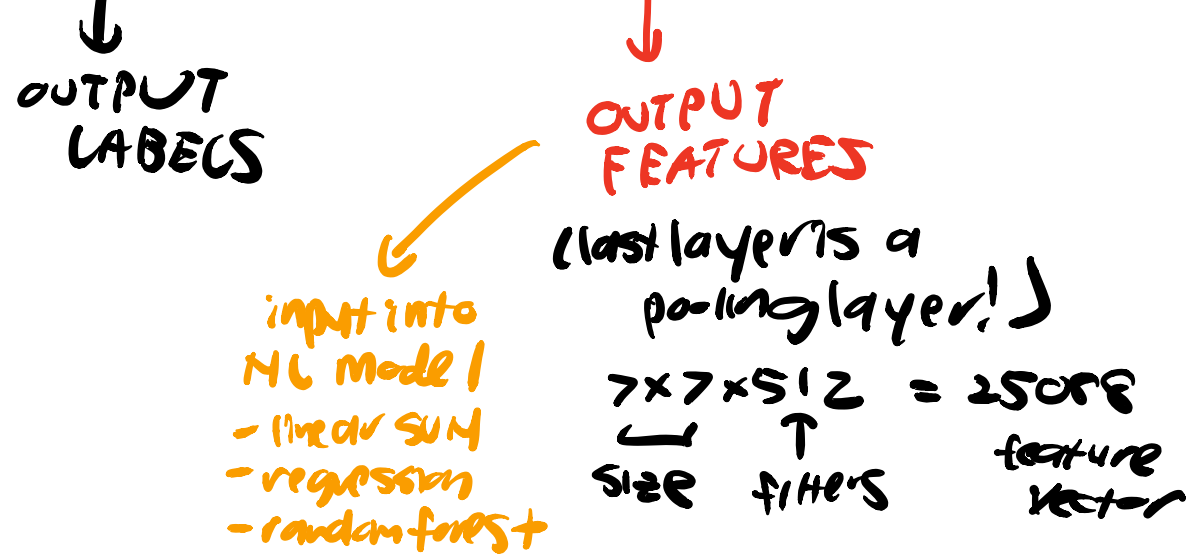
→ placing new FC layer

→ fine-tuning weights to recognize new object classes

Extracting features from Pre-trained CNN

1. stop propagation at arbitrary layer
2. extract values from network
3. uses feature vectors





★ use CNN as an intermediary feature extractor
NOT as the classifier!!!
Downstream ML classifier takes care of learning the underlying patterns of features extracted from CNN

HDF5
- binary data format
- stores large numerical datasets
- facilitates easy access/computation on rows of datasets
- stored hierarchically as "groups"
- "dataset" (ie. numpy array) stored in "groups"
- standardized
- written in C, but can be accessed via python using h5py

HDF5 Python class

- accept input data
  ie. extracted features from VGG16
- write to HDF5 dataset
  ie. save features
  generate dataset from raw images
  (for faster training)

- (N , dim)
  - N: # images
  - dim: size of each object
    FC = 1000
    image = 32×32×3
    ] examples

Extract Features

- Example:
  - extract via VGG16
    (remove last FC, save features)
    after training
  - train linear model

    → get high accuracy!
- networks are capable of:
  - performing transfer learning
  - encoding discriminative features
    into output activations

    → use features in our own custom

image classifiers!

Networks as Feature Extractors

- use pre-trained CNN to classify class labels outside of what it was originally trained on!
  - i.e. VGG, Inception, ResNet
  - (more powerful than hand-designed algorithms like HOG, SIFT, LBP...)
- when new CNN/DL problem:
  - can applying feature extraction give ↑ accuracy?
    - → skip network training process!!!
      - saves ALOT of time/effort

In Chapter 3 we learned how to treat a pre-trained Convolutional Neural Network as feature extractor. Using this feature extractor, we forward propagated our dataset of images through the network, extracted the activations at a given layer, and saved the values to disk. A standard machine learning classifier (in this case, Logistic Regression) was then trained on top of the CNN features, exactly as we would do if we were using hand-engineered features such as SIFT [15], HOG [14], LBPs [16], etc. This CNN feature extractor approach, called *transfer learning*, obtained remarkable accuracy, far higher than any of our previous experiments on the Animals, CALTECH-101, or Flowers-17 dataset.

Ranked Accuracy

- Rank-1 Accuracy
  - care about the top-1 predictions only

$$\frac{\#\text{correct predictions of TOP classifier}}{\#\text{datapoints in dataset}} = \%\text{ predictions of correct \#1 predictions}$$

- Rank-5 Accuracy

  - care about the top-5 predictions

$$\frac{\#\text{ correct predictions of top-5 classifiers}}{\#\text{ datapoints in dataset}} = \%\text{ predictions of correct top-5 predictions}$$

does the ground-truth label exist in the top 5 results?

yes → correct +1
no → wrong +0

- For challenging datasets (ie. Siberian Husky vs eskimo dog),
  use rank-5 to check if network is still improving
  (ideally rank-1 as well, but harder for difficult datasets)

- often used in papers + DL challenges

Fine Tuning

But there is *another* type of transfer learning, one that can actually *outperform* the feature extraction method if you have sufficient data. This method is called *fine-tuning* and requires us to perform "network surgery". First, we take a scalpel and cut off the final set of fully-connected layers (i.e., the "head" of the network) from a pre-trained Convolutional Neural Network, such as VGG, ResNet, or Inception. We then replace the head with a *new* set of fully-connected layers with random initializations. From there *all layers below the head* are frozen so their weights cannot be updated (i.e., the backward pass in backpropagation does not reach them).

We then train the network using a very small learning rate so the new set of `FC` layers can start to learn patterns from the *previously learned* `CONV` layers earlier in the network. Optionally, we may unfreeze the rest of the network and continue training. Applying fine-tuning allows us to apply pre-trained networks to recognize classes that they *were not originally trained on*; furthermore, this method can lead to higher accuracy than feature extraction.

- type of transfer learning
- applied to DL models that <u>have already been trained</u>

ie. VGG, ResNet, Inception trained on ImageNet

- pros
  - super powerful method to obtain image classifiers from pre-trained CNNs on custom data sets (more powerful than feature extractors)
- cons
  - more work/setup
  - FC parameters can affect network accuracy
  - can't rely on regularization else might conflict w/ pre-trained model

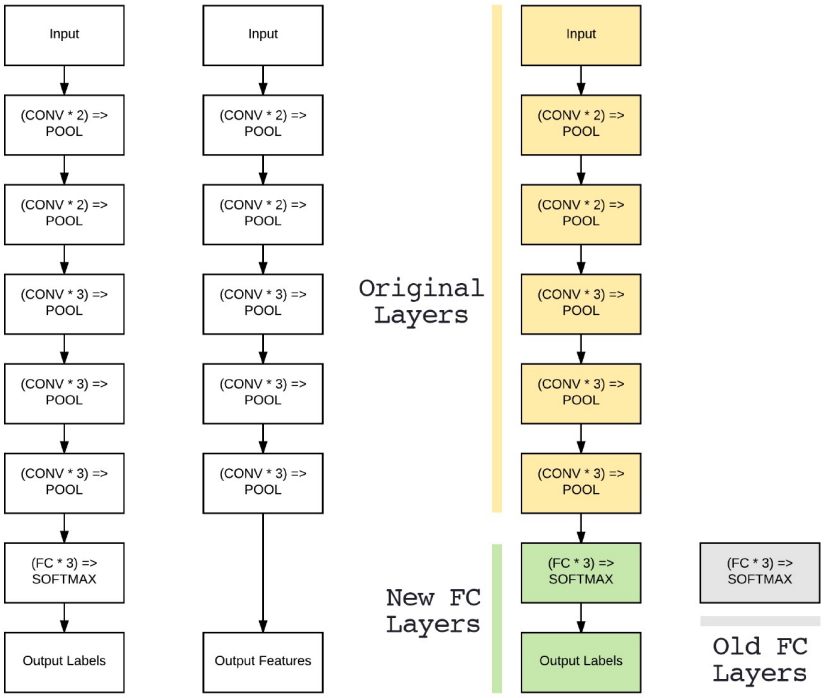
- Freezing layers
  - warm up FC layers
  - fully forward prop, but NOT backward prop
    → prevent random initialized FC layer to affect precisely trained layers
  - ~ 10-30 epochs

- RMSprop
  - good for quickly obtaining reasonable performance
    ie. "warming up" FC layers

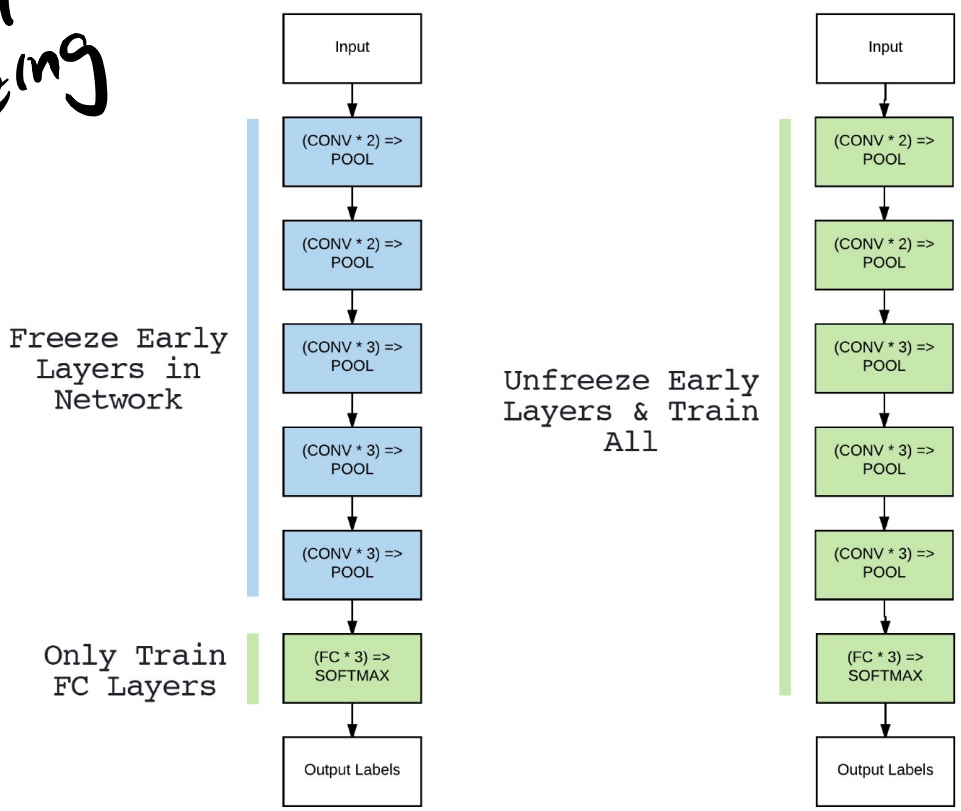
- SGD
  - for actual full/some network training after unfreeze

Network Surgery

classifier. When performing fine-tuning, we actually remove the head from the network, just as in feature extraction (*middle*). However, unlike feature extraction, when we perform *fine-tuning* we actually **build a new fully-connected head** and **place it on top of the original architecture** (*right*).

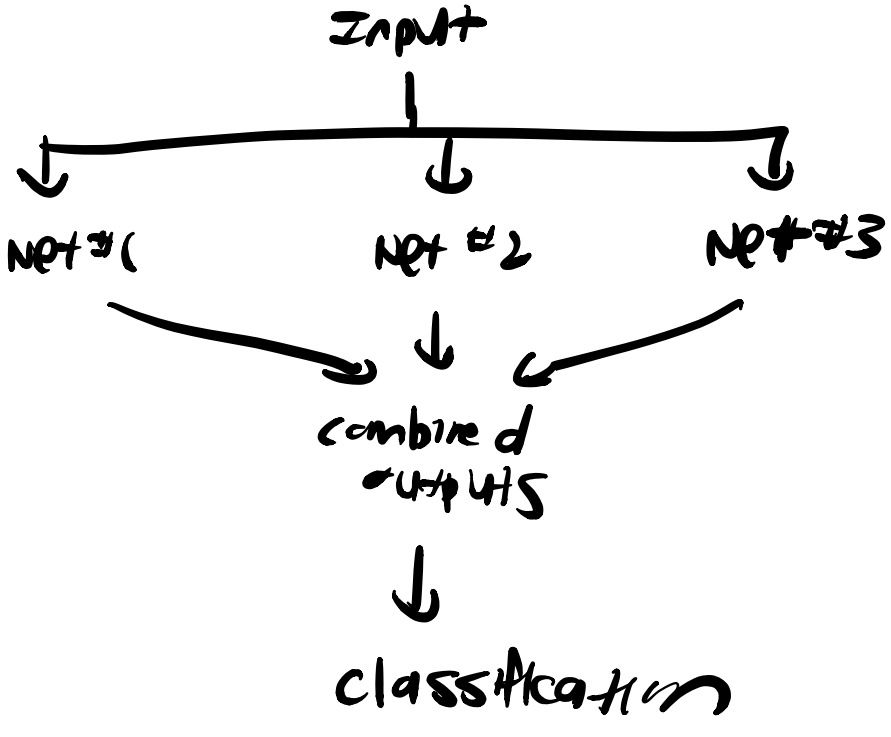


Training: Freezing/ Unfreezing



Ensemble Methods

- take multiple classifiers → one big meta-classifier
- combine output predictions of multiple classifiers via voting/averaging to yield result
  - ie. AdaBoost, Random forests
- Jensen's Inequality
  - "convex combined (average) ensemble will have error less than or equal to avg error of individual models"



- avg results ⇒ increase in classification accuracy!
- Two ways:
  1. train single network multiple times w/

different output paths to save weights

② use for loop to train N networks and output serialized model at end of each run

- NEVER jump into training an ensemble
  → test each network individually first!
    - architecture
    - optimizer
    - hyperparameters
- improves accuracy, but computationally expensive

Advanced optimization methods

① reduce training time
② reduce volatility in larger range of hyperparameters
③ obtain higher classification accuracy than SGD

- Learning Rate S
  - most important hyperparameter to tune
  - hard/tedious/time-consuming to tune

Adaptive Learning Rates

- SGD

$$W \mathrel{+}= (-\alpha)(dW)$$

- Adagrad

$$\text{cache} \mathrel{+}= dW^2$$

$$dW \mathrel{+}= \frac{(-\alpha)(dW)}{\text{eps} + \sqrt{\text{cache}}}$$

eps: smoothing / div 0 errors

cache: keep track of which weights changing most freq

- frequently updated / large gradients will scale update size down → lowers learning rate
- infrequently updated / small gradients will scale update size up → raises learning rate
- $\alpha = 0.01$ usually ← allow adagrad to tune/scale

- accumulate cache
  - → gradients get infinitesimally small

⇒ Adagrad RARELY used in deep networks

- Adadelta
  - extension of Adagrad
  - Adagrad
    - updates cache w/ ALL previously squared gradients
  - Adadelta
    - updates cache with a small number of past gradients

- RMSprop

$$cache = (decayrate)(cache) + (1 - decayrate)(dW^2)$$

$$W \mathrel{+}= \frac{-\alpha\, dW}{eps + \sqrt{cache}}$$

- uses moving average of weights in cache
  - → previous weights are weighted less than most recent in current calculation

- more effective than both Adagrad & Adadelta
- converges faster than SGD (sign.)

- Adam

mean → $m = \beta_1 m + (1-\beta_1) dW$

variance → $v = \beta_2 v + (1-\beta_2) dW^2$

$$W \mathrel{+}= \frac{-\alpha m}{eps + \sqrt{v}}$$

- "smoothed" version of m (vs raw calc dW)
- $\beta_1 = 0.9$, $\beta_2 = 0.999$] almost always used, rarely changed
- works better than RMSprop

- Nadam

- RMSprop w/ momentum
  - ↑ Nesterov acceleration

Choosing an Optimization method

- adaptive learning alg performed favorable
  → no clear winner
- master 2-3 opt algorithms
  → "drive" the algorithm w/ experienced hyperparameter tuning
- hyperparameter most important !!!!
  (despite slower speed of convergence in SGDs)
- Most used DL opt algs:
  1. SGD
  2. RMSprop
  3. Adam
- Recommendation (w/ momentum)
  - master/apply SGD first
  - expose yourself to as many DL problems as possible using a specific opt. alg. and learn how to tune the hyperparameters
  - mastering an opt. alg. is an art

that requires much practice!!!

- Dependant on familiarity w/
  - dataset
  - model architecture
  - opt alg (and associated hyperparameters)

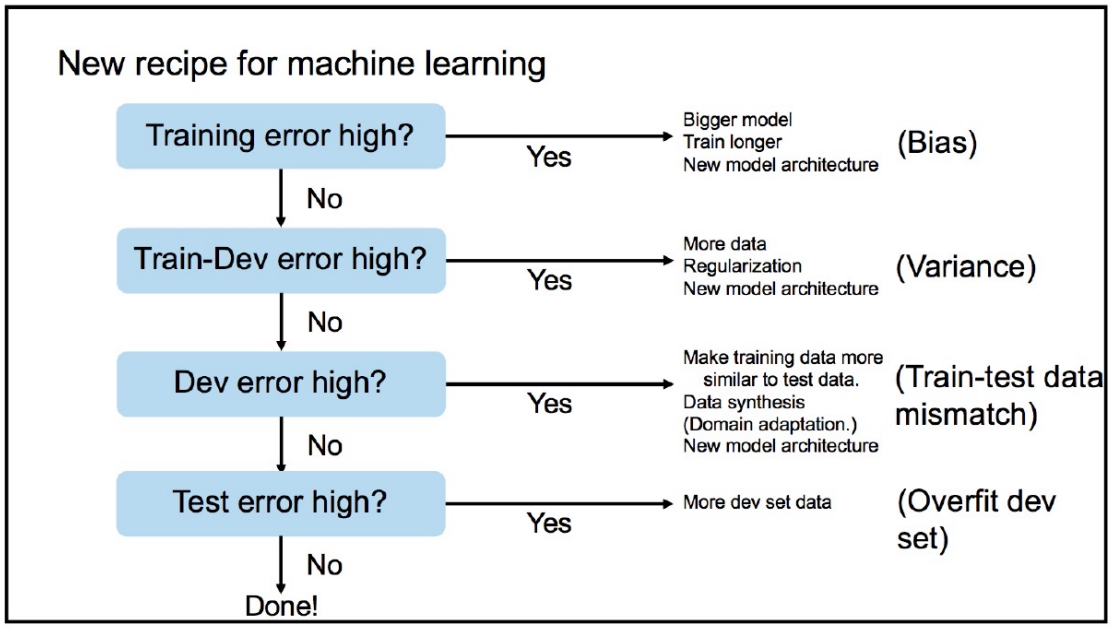
Training NNs

1. Dataset
2. Loss Function
3. NN architecture
4. optimization Method

Recipe for Training

1. Make sure your training data is representative of your validating/testing data!

2. Data Splits
   - training ← determines bias
   - training validating ← determines variance
   - validation



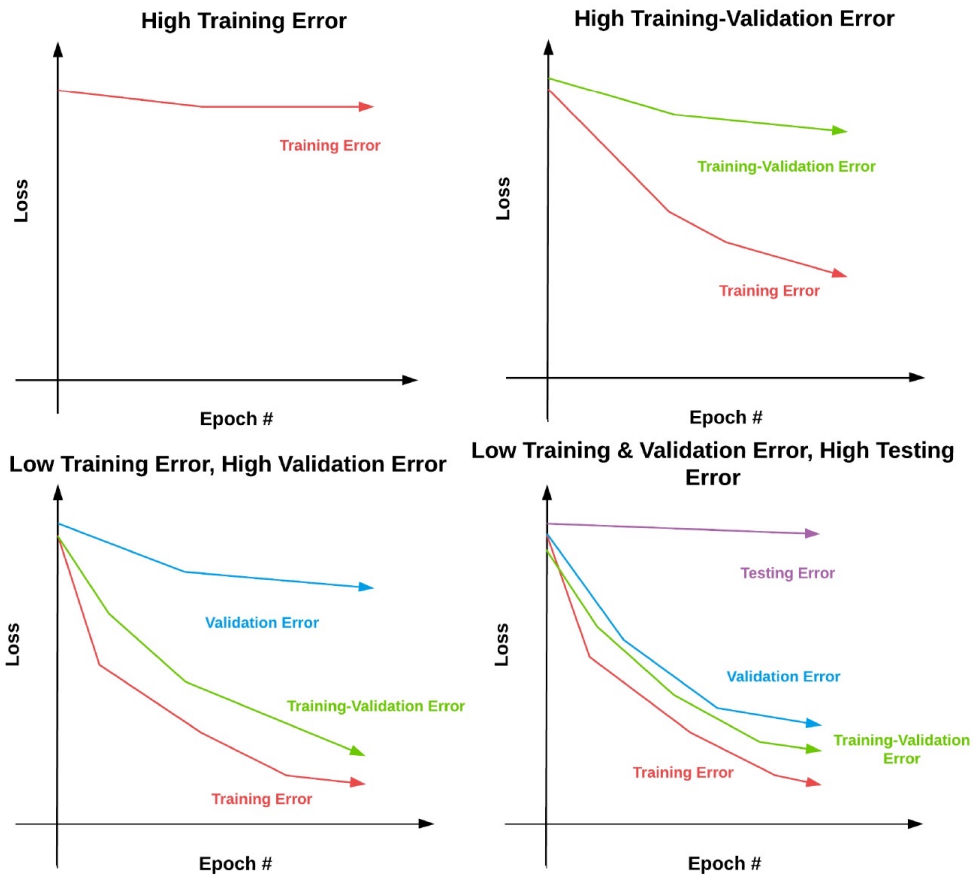


Figure 8.3: The four stages of Andrew Ng's machine learning recipe. **Top-left:** Our training error is high, implying that we need a more powerful model to represent the underlying patterns in the data. **Top-right:** Our training error has decreased, but our training-validation error is high. This implies we should obtain more data or apply strong regularization. **Bottom-left:** If both training and training-validation error are low, but validation error is high we should examine our training data and ensure it mimics our validation and testing sets properly. **Bottom-right:** If training, training-validation, and validation error are all low but testing error is high then we need to gather more training + validation data.

Transfer Learning vs. Train from Scratch

	Similar Dataset	Different Dataset
Small Dataset	Feature extraction using FC layers + classifier	Feature extraction using lower level CONV layers + classifier
Large Dataset	Fine-tuning likely to work, but might have to train from scratch	Fine-tuning worth trying, but will likely not work; likely have to train from scratch

Working w/ Large Datasets (too large to fit into memory)

- Data generators
  - access portion of dataset at a time (i.e. mini-batch)
- raw filepaths inefficient (I/O unoptimized)

  ⇒ generate HDF5 dataset of raw images

  ↑

  I/O optimized

- config files
  - path to input images
  - # class labels
  - training/validation/testing split info
  - path to HDF5 dataset
  - path to output models, plots, logs, etc.

- Mean subtraction

  ① store the avg red, green, blue pixel intensity

values across the entire dataset

② subtract mean value from every pixel in image

⇒ type of data normalization

- more effective/used than scaling [0,1]
- centers data ~0 mean
- enables network to learn faster

- HDF5
  - raw .jpg images use data compression so total size smaller, but slow I/O
  - HDF5 stores images as raw np arrays
    → increase storage costs
    decrease training time

- I/O latency is a huge problem, so best to optimize !!! (b/c training already takes time)

Image Generator

① open HDF5 dataset
② yield batches of images/training labels
③ continue until low loss/high accuracy achieved

Image Preprocessors

- Mean subtraction
  - subtract mean RGB pixel intensities from input
    $R = R - \mu_R$
    $G = G - \mu_G$
    $B = B - \mu_B$
  - reduces affect of lighting variations
- patch preprocessor
  - randomly extract $N \times N$ pixel regions from input
- over-sampling preprocessor
  - used at testing time
  - sample 5 regions & horizontal flip of inputs
    → 10 crops
  - boost accuracy by passing 10 crops through CNN & avg across 10 predictions

Practical Deep Learning

- NOT just the model architecture / optimizer!
- dataset handling/preprocessing VERY important!

classification

- binary cross-entropy
  - 2-class problem
- categorical cross-entropy
  - multi-class problem

Kaggle: Dogs vs Cats (via AlexNet)

- Process
  - save raw images to HDF5 ← I/O efficient, Training speed good for datasets unable to fit into memory
  - Image Data Generator (Keras) ← data aug
  - HDF5 Dataset Generator ← batch loading
  - image preprocessors

training	testing	
✗	✓	- Simple Preprocessor ← resize image
✓	✗	- Patch Preprocessor ← data aug
✓	✓	- Mean Preprocessor ← normalize pixel intensities
✓	✓	- ImageToArrayPreprocessor

- Training Monitor

convert images to Keras-compatible arrays

- Model
  - AlexNet (227 x 227 x 3)
    - Batch Normalization
    - Adam optimizer
    - Binary cross-entropy (2 classes)

- Evaluation
  - Rank-1 Accuracy (b/c we only have 2 classes...)
  - Progress Bar
  - Over-sampling (10-cropping) aka CropPreprocessor

92.6% ↑

94% accuracy →

final pred. is avg. accuracy across 10 crops
- boosts accuracy!

- obtaining Top-25 (>96%)
  - use transfer learning via feature extraction w/ ImageNet dataset via ResNet!
    - able to differentiate btwn breeds of animals so cat/dog should be easier
  - Plan
    - extract features from pre-trained ResNet
    - train Logistic Regression classifiers on these features

Summary

In this chapter we took a deep dive into the Kaggle Dogs vs. Cats dataset and studied two methods to obtain $> 90\%$ classification accuracy on it:

1. Training AlexNet from scratch.
2. Applying transfer learning via ResNet.

The AlexNet architecture is a seminal work first introduced by Krizhevsky et al. in 2012 [6]. Using our implementation of AlexNet, we reached 94 percent classification accuracy. This is a very respectable accuracy, especially for a network trained from scratch. Further accuracy can likely be obtained by:

1. Obtaining *more* training data.
2. Applying more aggressive data augmentation.
3. Deepening the network.

However, the 94 percent we obtained is not even enough for us to break our way into the top-25 leaderboard, let alone the top-5. Thus, to obtain our top-5 placement, we relied on transfer learning via feature extraction, specifically, the ResNet50 architecture trained on the ImageNet dataset. Since ImageNet contains *many* examples of both dog and cat breeds, applying a pre-trained network to this task is a natural, easy method to ensure we obtain higher accuracy with less effort. As our results demonstrated, we were able to obtain $\mathbf{98.69}\%$ classification accuracy, high enough to claim the *second position* on the Kaggle Dogs vs. Cats leaderboard.

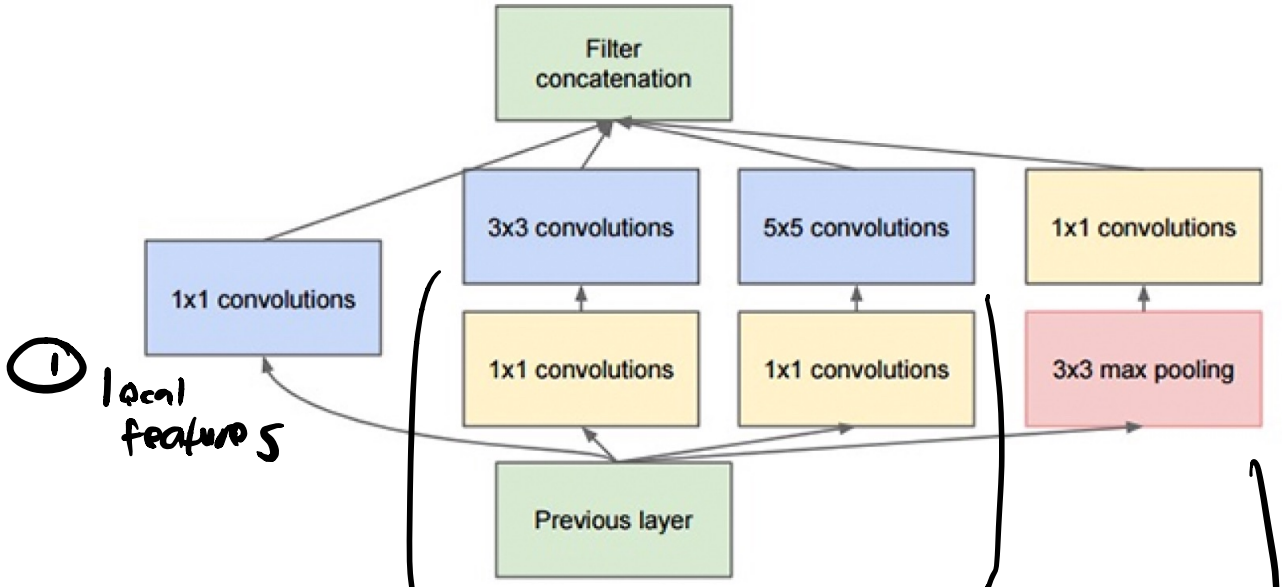
Google leNet

- small model compared to AlexNet and VGGNet
  - remove FC layers → use global avg. pooling
- micro-architecture
  - network in network
  - output from one layer can split and rejoin
- Inception Module
  - enable CNNs to learn CONV layers w/ multiple filter sizes

Inception Module

- micro-architecture
  - building blocks than enable nns to learn:
    1. faster
    2. more effectively
    3. increases network depth
- enable models to learn multiple filter sizes
  - → learn local features (smaller CONV) 1x1, 3x3
    + abstracted features (larger CONV) 5x5
- train on ImageNet dataset

original Inception



① local features

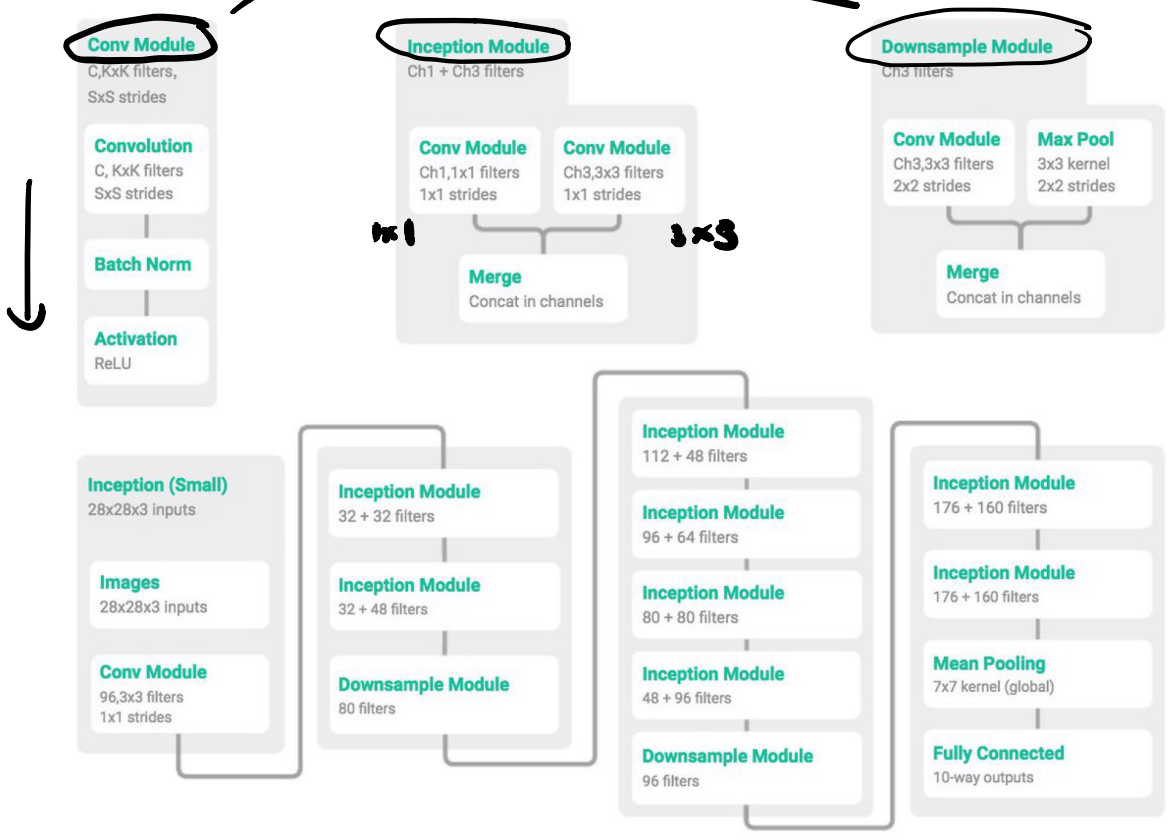
② local features + reduce dims

③ abstract features + reduce dims

④ pool projection branch - reduce volume size

Miniception

3 models used in MiniGoogleNet



Sequential vs. Functional

- Sequential
  - one layer after another
  - → model = Sequential()
    model.add()
- Functional API
  - use Model(). NOT Sequential() to define network
  - → output = Layer(parameters)(input)

GoogleNet

- Tools Used
  - training monitor
  - learning rate schedular → $\alpha = e^{-3}$ decay linearly over 70 epochs
  - data augmentation
  - SGD optimizer
  - GoogLeNet
- CIFAR10
  - ~50-100 epochs to train
  - will inevitably overfit

why 70 epochs?

⇒ 87.95% classification accuracy

⇒ need more aggressive regularization!
ie. L2 weight decay

Tiny ImageNet

- subset of ImageNet
- 200 classes
- each class has:
  - 500 training
  - 50 validation
  - 50 testing

  ] ground-truths provided
- original dataset varying widths & heights
  ⇒ need to resize images!
- Tiny ImageNet dataset
  ⇒ 64×64 centercropped

Practical Deep Learning

- NOT just about implementing CNNs and training them from scratch!
  ⇒ Need to develop simple scripts to parse data!!!
- DL is an iterative process!!!
- use SGD to obtain baseline

DeeperGoogleNet ← original Inception Module

prep
- ① encode labels to unique integers
- ② train/test split
- ③ convert raw images to HDF5 ← HDF5DatasetWriter
  train, val, test.hdf5
- ④ get RGB pixel intensity mean ← MeanPreprocessor
- ⑤ create deepergoogLeNet
  ... just read the diagrams in the chapter rip

train
- ⑥ Data augmentation (Keras)
- ⑦ Preprocessors for train/val
  - simple ← resize
  - mean ← normalization
  - imagetoarray ← Keras-compatible
  ] HDF5DatasetGenerator
- ⑧ optimizer
  - Adam
  - L2 regularization (learning rate)
- ⑨ callbacks
  - progress bar
  - training monitor → plot graphs every epoch
  - epoch checkpoints → serialize model weights every X epochs

eval
- ⑩ Rank-1 and Rank-5 Accuracy
  ⇒ check chapter 11 for Results
  (didn't have memory/time to run myself...)

ResNet

- residual module (type of micro-architecture)
- very deep layers w/ standard SGDs
- uses "smarter" weight initialization algorithms
- identity mapping
- does not use pooling layers often
  - → uses conv w/ stride > 1 instead

Identity Mappings

- take original input and add it to the output
  - → 1+1=2 addition
    - NOT concatenation!!!
- enables faster learning + larger learning rates!
- typically use Bottleneck variant
- pre-activation residual module variant
  - apply act/batch before conv

ch. 12 resnet

⇒ Did not implement code (yet)!

- ResNet w/ CIFAR10 → 93.58%
- ResNet w/ TinyImageNet → 58.03%

Object Detection

1. Sliding Windows
2. Image Pyramids
3. Non-Maxima Suppression

] before CNNs was popular

Haar Cascades

- "rapid object detection using Boosted Cascad of Simple Features"
- mostly used for face detection
- cons
  - high false positive rate
  - parameter reliant / noisy

HOG + Linear SVM

- "histogram of oriented gradients for human detection"
- pros
  - higher detection accuracy
  - fewer parameters
  - lower false positive rate
- cons
  - slower than Haar cascades

] sliding windows + image pyramids (scale) (translation)

Implementation

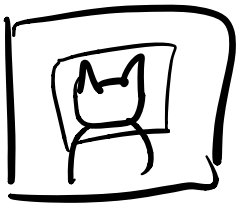
- sliding windows
- image pyramids

] extract ROI (region of interest)

- batch processing ← predict class labels

- non-maxima suppression ← removes overlapping bounding boxes

=> Deep-Learning Based Object Detector



- sliding windows/image pyramids
  slow + tedious
- parameter selection
  slow + tedious

=> Use Faster R-CNNs and SSDs instead to train end-to-end deep learning Object detectors

chp. 13 obj-detection

=> Did not implement code (yet)!

Deep Dream

- produce dream-like images via CNNs
- run a pre-trained CNN in reverse
- freeze weights and modify input image
  → feed back into CNN!
  (feedback loop)
- Inception model w/ ImageNet
- code:

- gradient ascent
  - → generates actual dream
- lower layers
  - → generate edges / geo patterns
- high layers
  - → visual patterns (dogs / cats / birds)

- process
  1. input image
  2. process input image at different scales (aka octaves)
  3. for each octave, maximize the entire layer tensors, and mix results together → upscale image
  4. utilize detail reinjection
  -

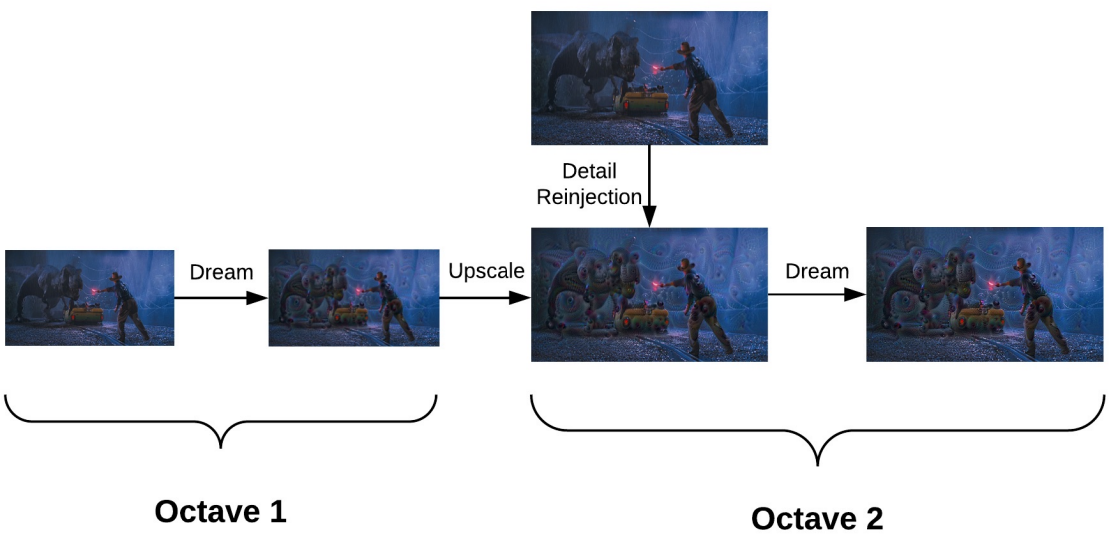


Figure 14.2: The actual DeepDream process involves (1) applying deep dreaming to in input image, (2) utilizing detail reinjnection, and (3) upscaling the image. We apply this process across a number of scales of an called octaves.

Neural Style Transfer

- apply style to content image
- 3-component loss functions
  - content loss
  - style loss
  - total variation loss

Content loss

- NN are hierarchical learners

① use pre-trained network (single layer!)

② use higher-level layer to serve as content loss
→ abstract qualities / larger features

③ compute activations of this layer for both content & style image

④ take L2-norm between these activations

style loss

- use multiple layers to get multi-scale representation of style & texture
- compute correlations b/w activation layers via Gram matrix
  ↑
  inner product of a set of feature maps

Total-variation loss

- operates only on the output image

Combining loss functions

- weighted combination of content & style, total variation loss

```
loss = (alpha * D(style(original_image) - style(gen_image))) +
       (beta * D(content(original_image) - content(gen_image))) +
       (gamma * tv(gen_image))
```

Neural style transfer

- works best with:
  - content images that do not require high levels of detail to be recognizable
  - style images that contain alot of texture
- varying content, style, and total-variation weights will give different results!

GANs

- generator → wants to minimize loss
  - accepts input vector of randomly generated noise
  - produces output "imitation" image
- discriminator → want to maximize loss
  - determines if an image is "real" or "fake"
- very difficult to train! ← evolving loss landscape

- Procedure
  1. randomly generate vector (noise)
  2. pass noise through generator
     → generate image
  3. mix real & fake images
  4. train discriminator w/ mixed set
  5. randomly generate vector (noise)
     → purposely label as "real"
  6. train GAN using noise vectors and "real" images (even though they are not real images)

- freeze discriminator weights when training generator
- each iteration
  1. generate random images, train discriminator to distinguish btwn the two
  2. generate more random images, purposely trying to fool the discriminator ← so generator can figure out weakness
     - update generator
     - freeze discriminator
  3. update generator weights based on discriminator feedback

Best Practices when Training GANs

- two losses to minimize
  - discriminator
  - generator

- goal is to find equilibrium btwn the two, NOT to seek minimized loss!

Radford et al. recommends the following architecture guidelines for more stable GANS:

- Replace any pooling layers with strided convolutions (we have seen this concept used in ResNet in Chapter 11)
- Use batch normalization in both the generator and discriminator
- Remove fully-connected layers in deeper networks
- Use ReLU in the generator except for the final layer which will utilize *tanh*
- Use Leaky ReLU in the discriminator

François Chollet then provided additional recommendations in his book [63]:

1. Sample random vectors from a *normal distribution* (i.e., Gaussian distribution) rather than a *uniform distribution*
2. Add dropout to the discriminator
3. Add noise to the class labels when training the discriminator
4. To reduce checkerboard pixel artifacts in the output image use a kernel size that is divisible by the stride when utilizing convolution or transposed convolution in both the generator and discriminator
5. If your adversarial loss rises dramatically while your discriminator loss falls to zero, try reducing the learning rate of the discriminator and increasing the dropout of the discriminator.

Keep in mind that these are all just *heuristics* found to work in a number of situations — we'll be using *some* of techniques suggested by both Radford et al. and Chollet, but not *all* of them. It is possible, and even *probable*, that the techniques listed here will not work on your GANs. Take the time now to set your expectations that you'll likely be running *orders of magnitude more experiments* when tuning the hyperparameters of your GANs as compared to previous experiments in this book.

Image Super Resolution

- process of upscaling and/or improving details in an image
- use SRCNNs to produce faster, better, end-to-end results

SRCNNs

- fully convolutional
  → way faster to train! (b/c no FC layers)
- train for filters, NOT for accuracy
  → no optimization or loss needed
- end-to-end

input → network → high-res output

- goal:
  - learn a set of filters to map low-res inputs to high-res outputs
  - inputs:
    - low-res input
    - high-res target

- patches = sub-images