Training, Transfer Learning, & Generative Models

By Abe Davis
With some slides from
Jin Sun, Noah Snavely, Philipp Isola

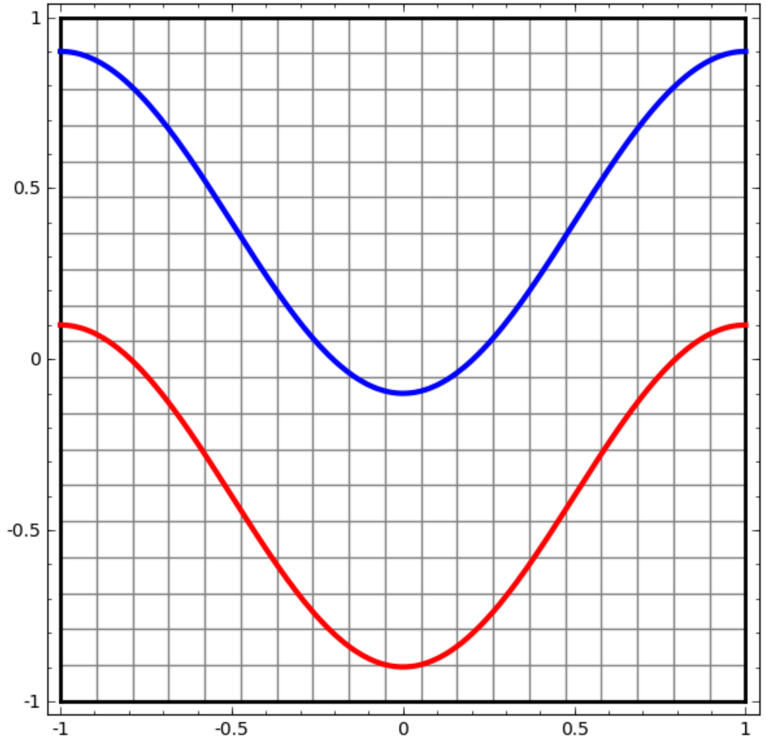
Announcements

- Project 5 (Convolutional Neural Networks) released today
  - Due Wednesday, April 29
- Take-home final exam planned May 11-14

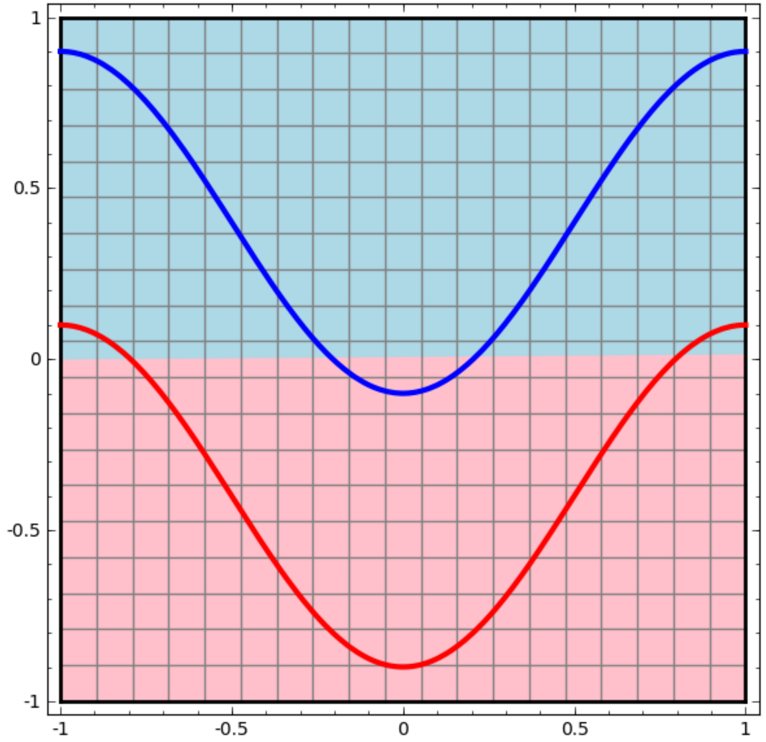
This Lecture (and maybe part of the next one)

- Visualizing Deep Classification
- A Review of Overfitting
- Regularization in Deep Learning
- How to Train Deep Nets
- Transfer Learning
- Generative Models
- Transpose Convolution

Visualizing Linear Classification



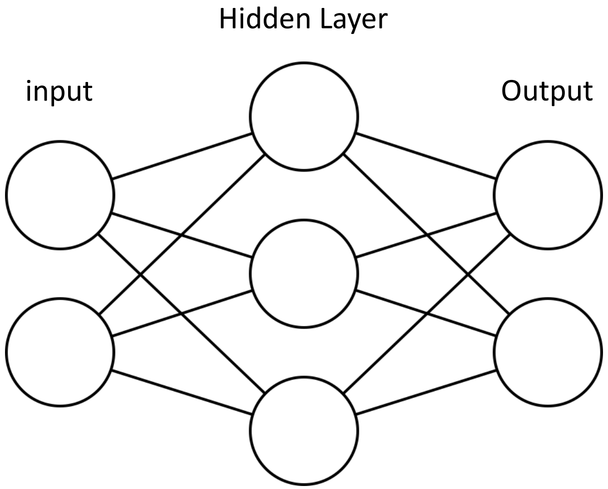
Classification Problem: Separate Red & Blue



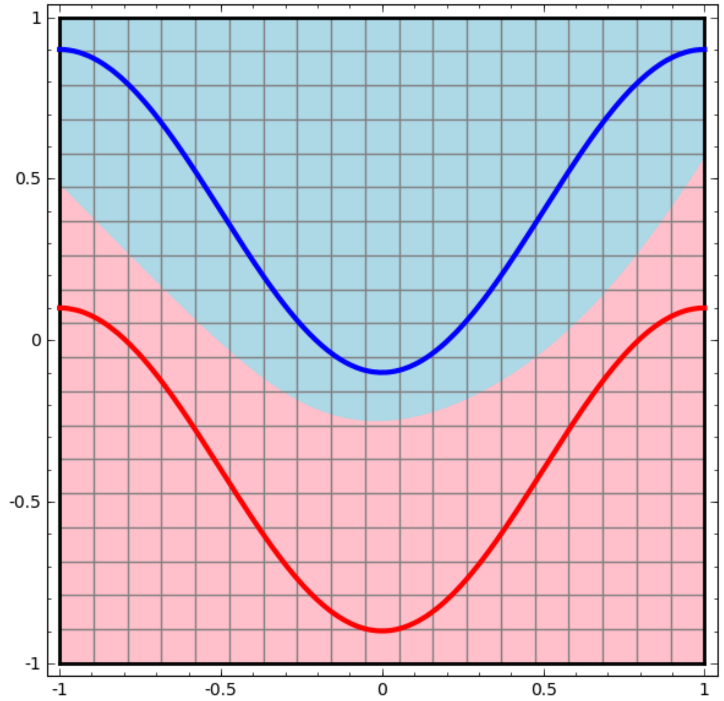
Linear Solution

Based on http://colah.github.io/posts/2014-03-NN-Manifolds-Topology/

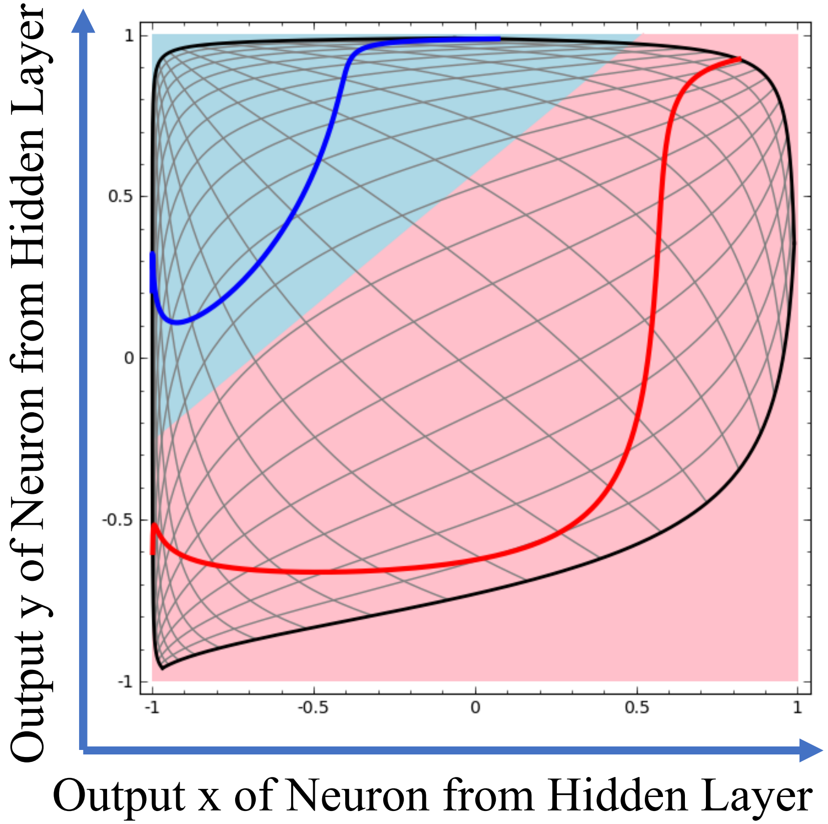
Visualizing Classification With a Neural Network



Example Network



Classification Results for Every Point in Original Space



Classification Results for Every Point in Transformed Feature Space

Based on http://colah.github.io/posts/2014-03-NN-Manifolds-Topology/

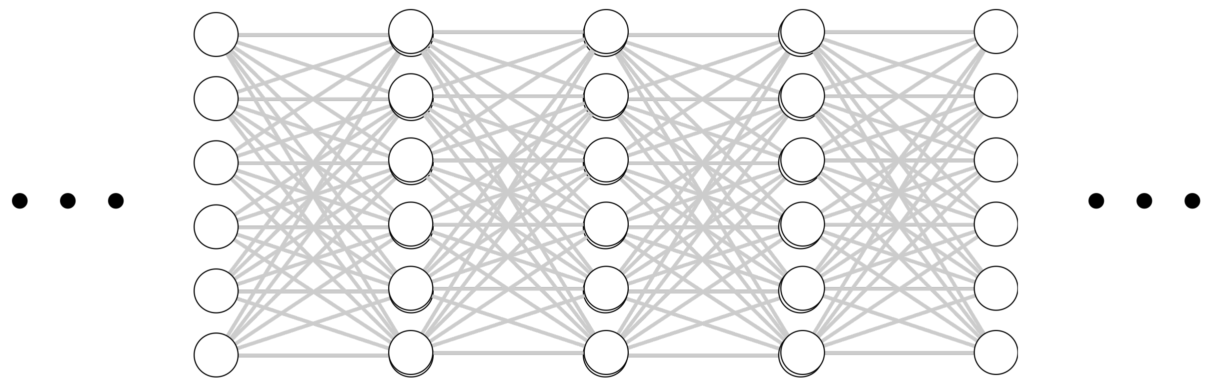
Demo

https://cs.stanford.edu/people/karpathy/convnetjs/demo/classify2d.html

What Makes Training Deep Nets Hard?

- It's easy to get high training accuracy:
  - Use a huge, fully connected network with tons of layers
  - Let it memorize your training data

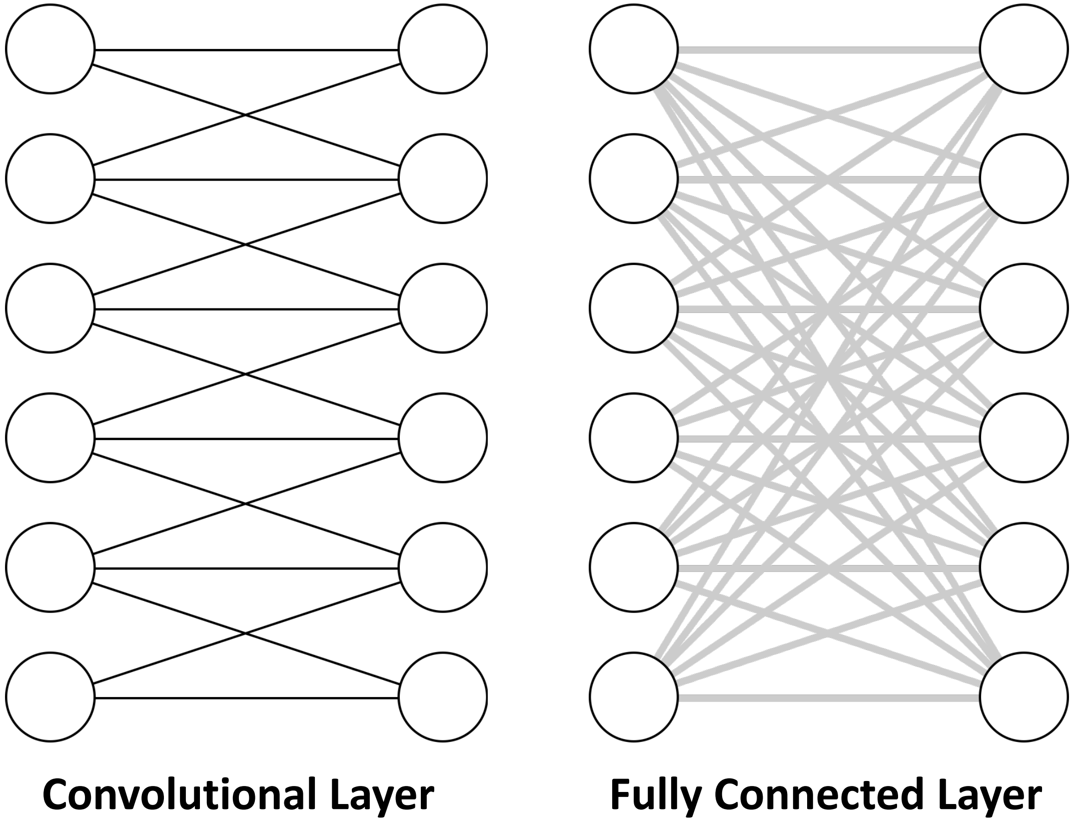
- Its hard to get high *test* accuracy



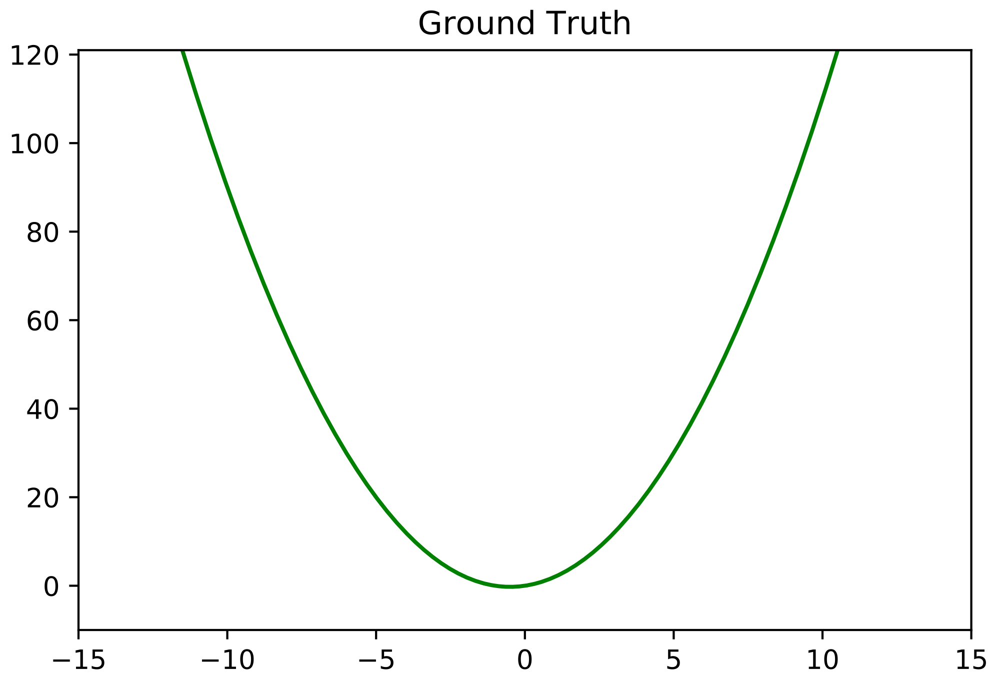
This would be an example of overfitting

Related Question: Why Convolutional Layers?

- A fully connected layer can generally represent the same functions as a convolutional one
  - Think of the convolutional layer as a version of the FC layer with constraints on parameters
- What is the advantage of CNNs?



A Review of Overfitting




```
/Users/abedavis/anaconda2/envs/dremix/lib/python2.7/site-packages/librosa/__init__.py:3
5: FutureWarning: You are using librosa with Python 2. Please note that librosa 0.7 will
be the last version to support Python 2, after which it will require Python 3 or later.
  FutureWarning)

TEST_DIR=/Users/abedavis/Code/MyRepos/python/abepy/notebooks/Scratch/TEMP_MEDIAGRAPH_TES
T DIR
```

In [24]:

```python
start_t = -10;
end_t = 10;
n_samples = 500;
n_data_points = 15;
trange=8.0;
jitter = np.random.rand(n_data_points)*2-1;
jitter_amp = 0;
noise_amp = 25.0;

sample_times = np.linspace(start=-1.0, stop = 1.0, num = n_samples, endpoint=True)*trang
data_times = np.linspace(start=-n_data_points, stop = n_data_points, num = n_data_points
data_times = data_times*(np.true_divide(trange, n_data_points));
tsig_gt = UnstructuredTimeSignal(data_times, np.square(data_times)+data_times);

tsig_noise = UnstructuredTimeSignal(data_times, tsig_gt.sample_values+(np.random.rand(n_
```

executed in 10ms, finished 17:43:20 2020-04-21

In [25]:

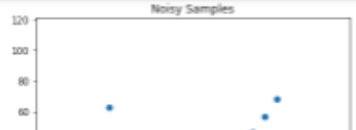
```python
# tsig_gt.plotLine(color='green')
# tsig_noise.plotPoints()
```

executed in 4ms, finished 17:43:21 2020-04-21

In [50]:

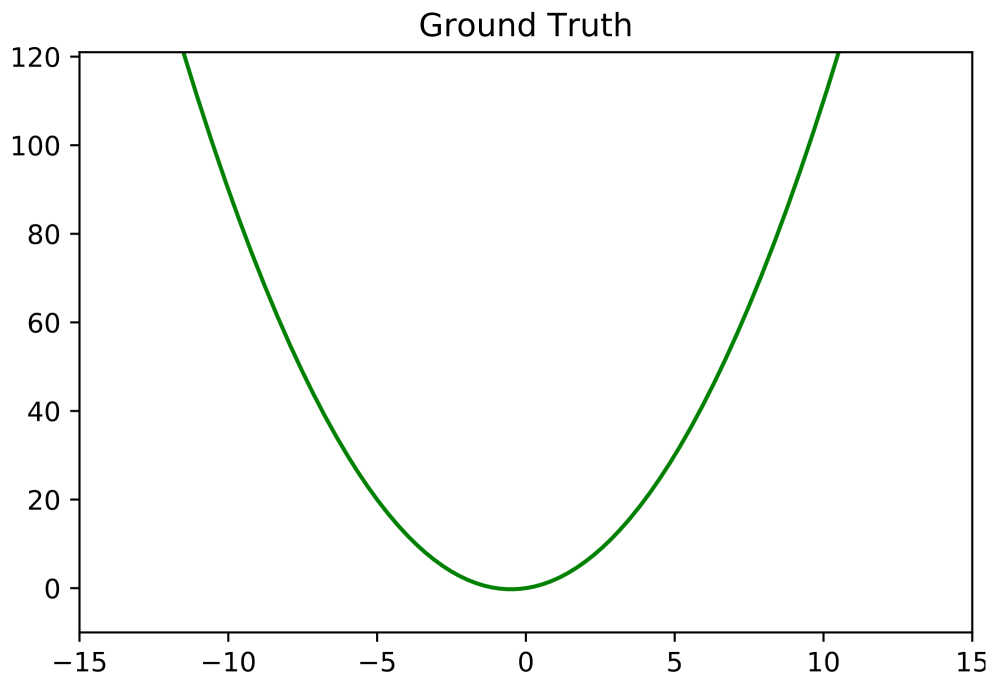
```python
fits = [];

ylim = [-10, np.power(trange+3, 2.0)]
xlim = [-15, 15];

polfunc = tsig_gt.getPolyFitFunc(deg=2);
pfit = UnstructuredTimeSignal(sample_times, polfunc(sample_times));
# pfit.plotLine(color='green');
tsig_noise.plotPoints();
plt.ylim(*ylim);
plt.xlim(*xlim);
plt.title('Noisy Samples');
plt.savefig('/Users/abedavis/Documents/Abe/Teaching/CS5670/2020/gans/overfitting/figs/sa
plt.show()

polfunc = tsig_gt.getPolyFitFunc(deg=2);
pfit = UnstructuredTimeSignal(sample_times, polfunc(sample_times));
pfit.plotLine(color='green');
plt.ylim(*ylim);
plt.xlim(*xlim);
plt.title('Ground Truth');
plt.savefig('/Users/abedavis/Documents/Abe/Teaching/CS5670/2020/gans/overfitting/figs/gr
plt.show()


polfunc = tsig_gt.getPolyFitFunc(deg=2);
pfit = UnstructuredTimeSignal(sample_times, polfunc(sample_times));
pfit.plotLine(color='green');
tsig_noise.plotPoints();
plt.ylim(*ylim);
plt.xlim(*xlim);
plt.title('Ideal Fit');
plt.savefig('/Users/abedavis/Documents/Abe/Teaching/CS5670/2020/gans/overfitting/figs/id
plt.show()


for a in range(n_data_points+1):
    polfunc = tsig_noise.getPolyFitFunc(deg=a);
    pfit = UnstructuredTimeSignal(sample_times, polfunc(sample_times));
    fits.append(pfit);
    pfit.plotLine(color='red');
    tsig_noise.plotPoints();
    plt.ylim(*ylim);
    plt.xlim(*xlim);
    plt.title('Poly Fit Degree {}'.format(a))

    plt.savefig('/Users/abedavis/Documents/Abe/Teaching/CS5670/2020/gans/overfitting/fig
    plt.show();
```

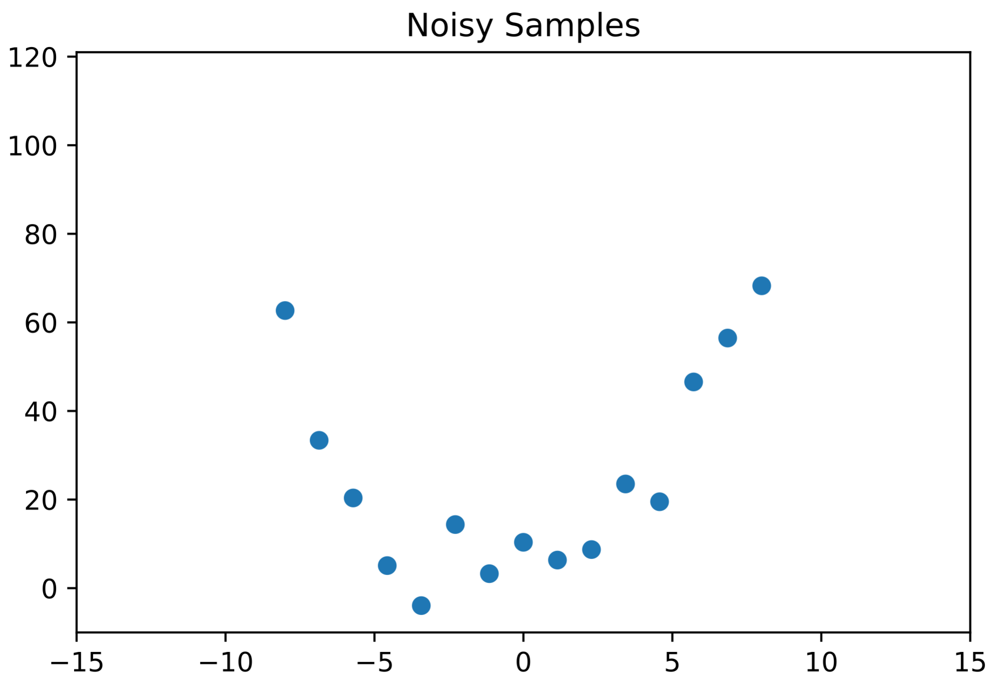
executed in 2.79s, finished 18:13:00 2020-04-21



Overfitting: More Parameters, More Problems

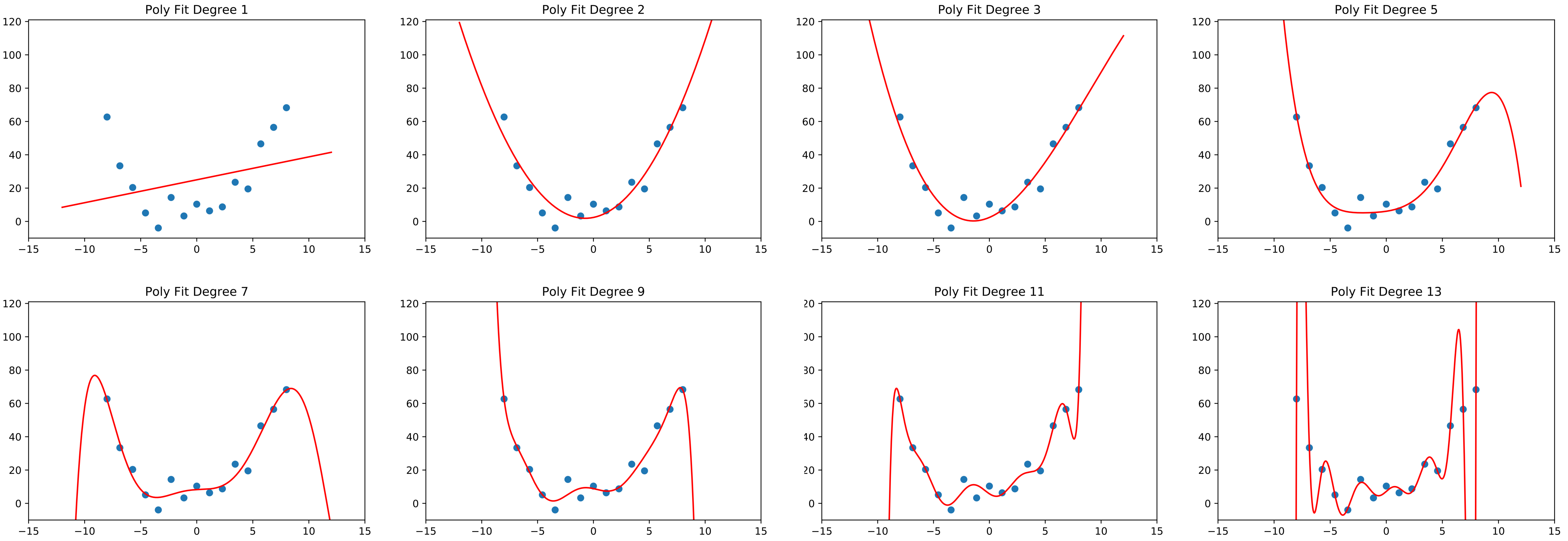
- Non-Deep Example: consider the function $x^2 + x$
- Let's take some noisy samples of the function...





Overfitting: More Parameters, More Problems

- Now lets fit a polynomial to our samples of the form $P_N(x) = \sum_{k=0}^{N} x^k p_k$



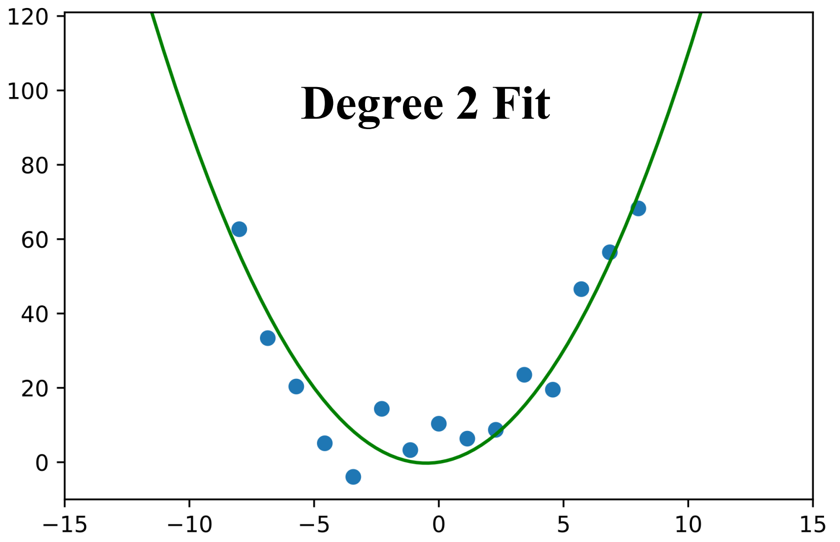
Overfitting: More Parameters, More Problems

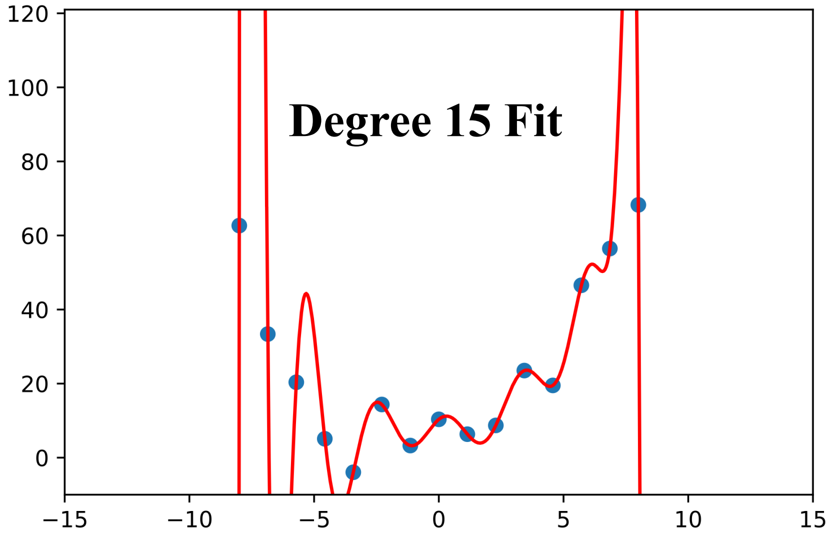
- A Model with more parameters can represent more functions

- E.g.,: if $P_N(x) = \sum_{k=0}^{N} x^k p_k$ then $P_2 \in P_{15}$

- More parameters will often **reduce training error** but **increase testing error**. This is *overfitting*.

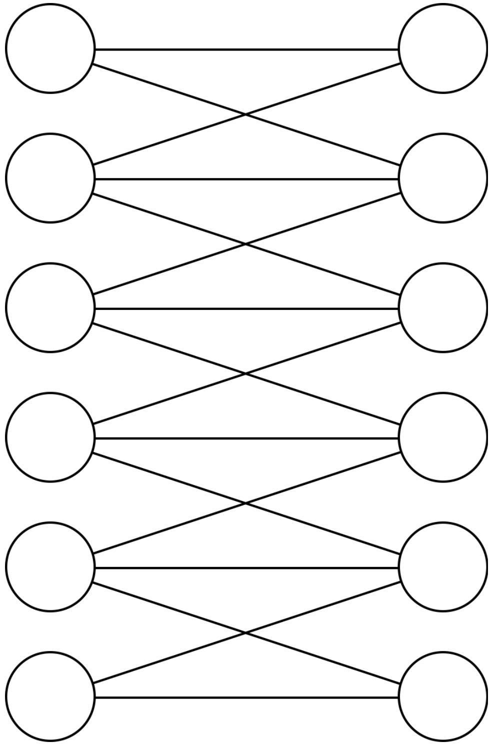
- When overfitting happens, models do not generalize well.

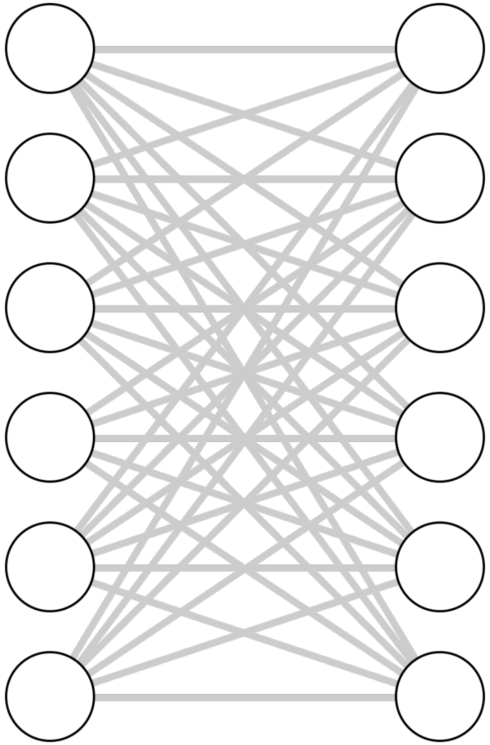




Deep Learning: More Parameters, More Problems?

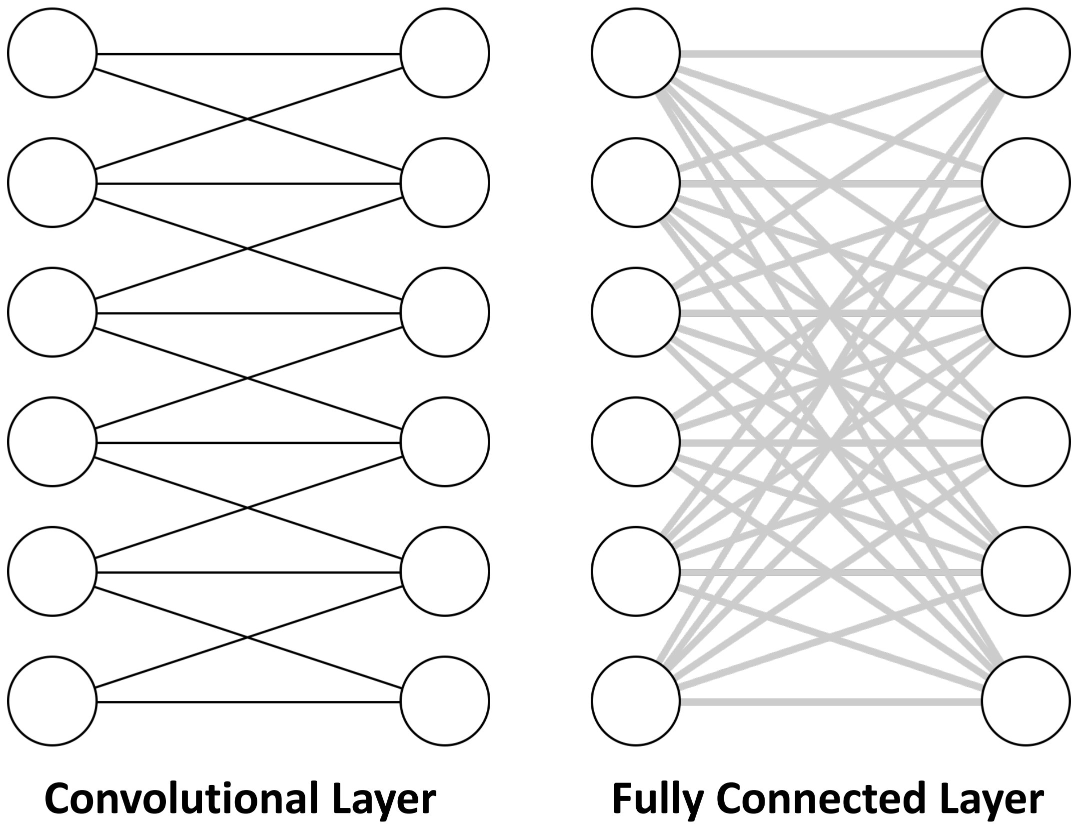
- More parameters let us represent a larger space of functions
- The larger that space is, the harder our optimization becomes
- This means we need:
  - More data
  - More compute resources
  - Etc.


Convolutional Layer


Fully Connected Layer

Deep Learning: More Parameters, More Problems?

A convolutional layer looks for components of a function that are spatially-invariant



How to Avoid Overfitting: Regularization

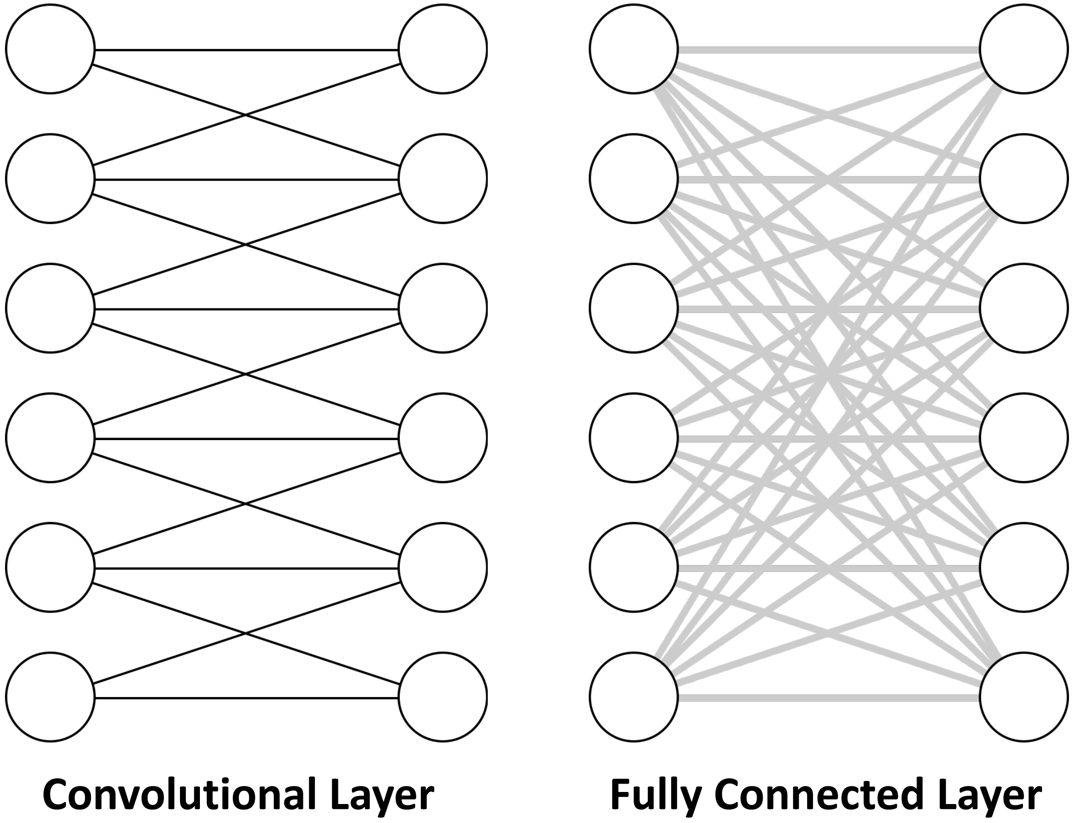
- In general:
  - More parameters means higher risk of overfitting
  - More constraints/conditions on parameters can help

- If a model is overfitting, we can
  - Collect more data to train on
  - *Regularize*: add some additional information or assumptions to better constrain learning

- Regularization can be done through:
  - the design of architecture
  - the choice of loss function
  - the preparation of data
  - …

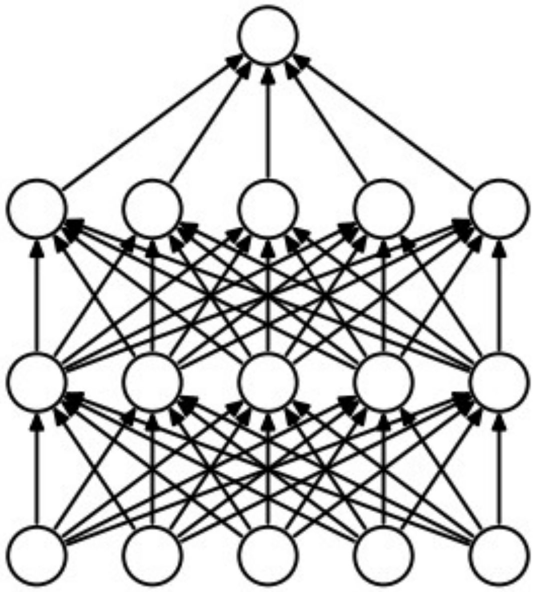
Regularization: Architecture Choice

- "Bigger" architectures (typically, those with more parameters) tend to be more at risk of overfitting.

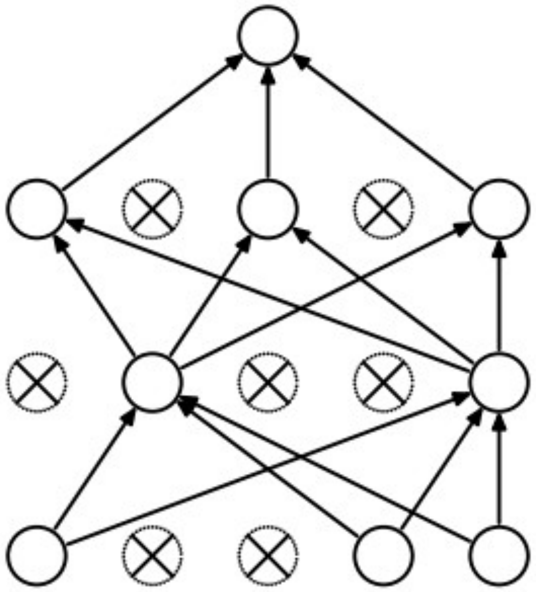


Regularization: Dropout

- At training time, randomly “drop” (zero out) some fraction of the connections in your network
- This will prevent your network from relying too heavily on any specific connections
- Encourages redundancy/consensus across various paths through the network



(a) Standard Neural Net

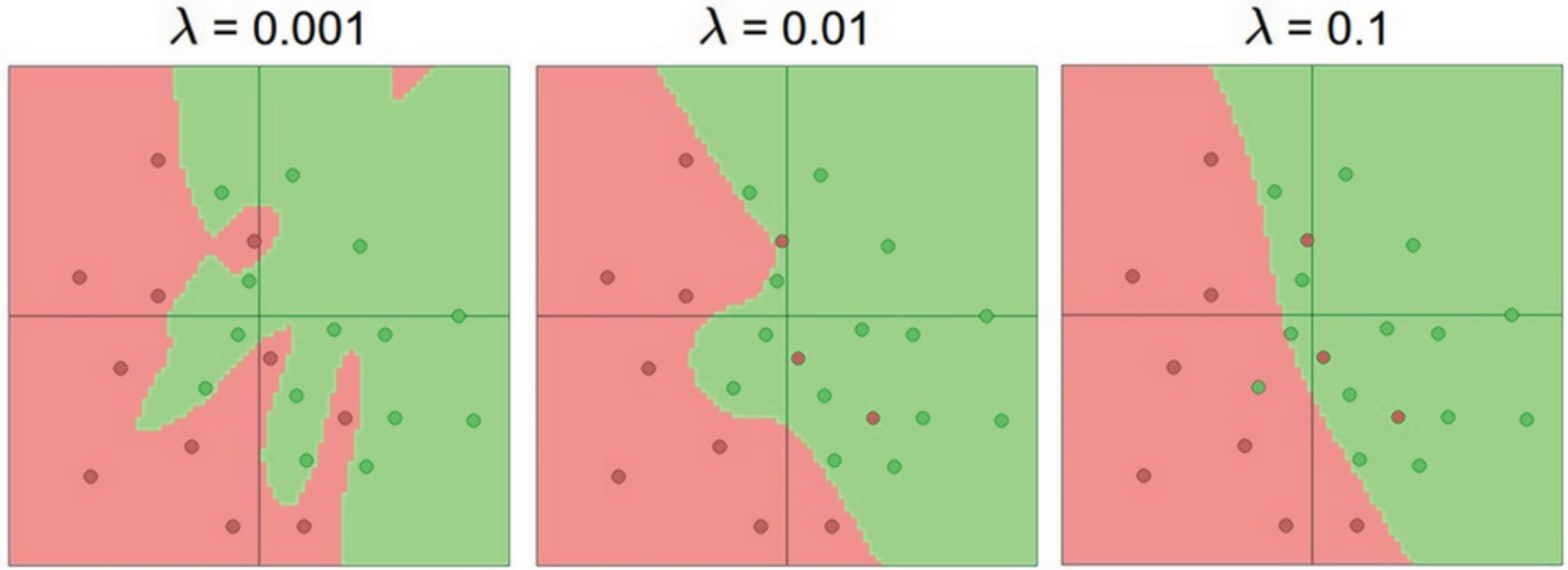


(b) After applying dropout.

Srivastava, Nitish, et al. ”Dropout: a simple way to prevent neural networks from overfitting”, JMLR 2014

Regularization: In the Loss Function

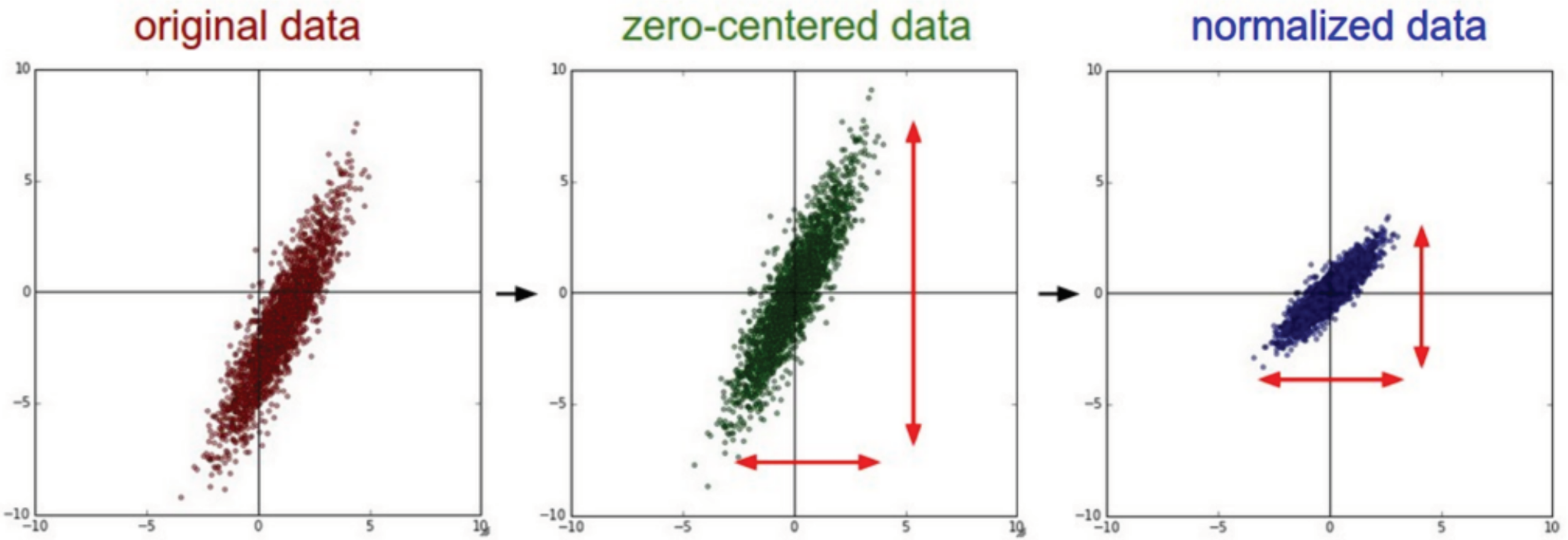
$$L = L_{\text{data}} + L_{\text{reg}} \qquad L_{\text{reg}} = \lambda \frac{1}{2} \|W\|_2^2$$



[Andrej Karpathy http://cs.stanford.edu/people/karpathy/convnetjs/demo/classify2d.html]

Regularization: In Data Preparation

Preprocess the data so that learning is better conditioned:



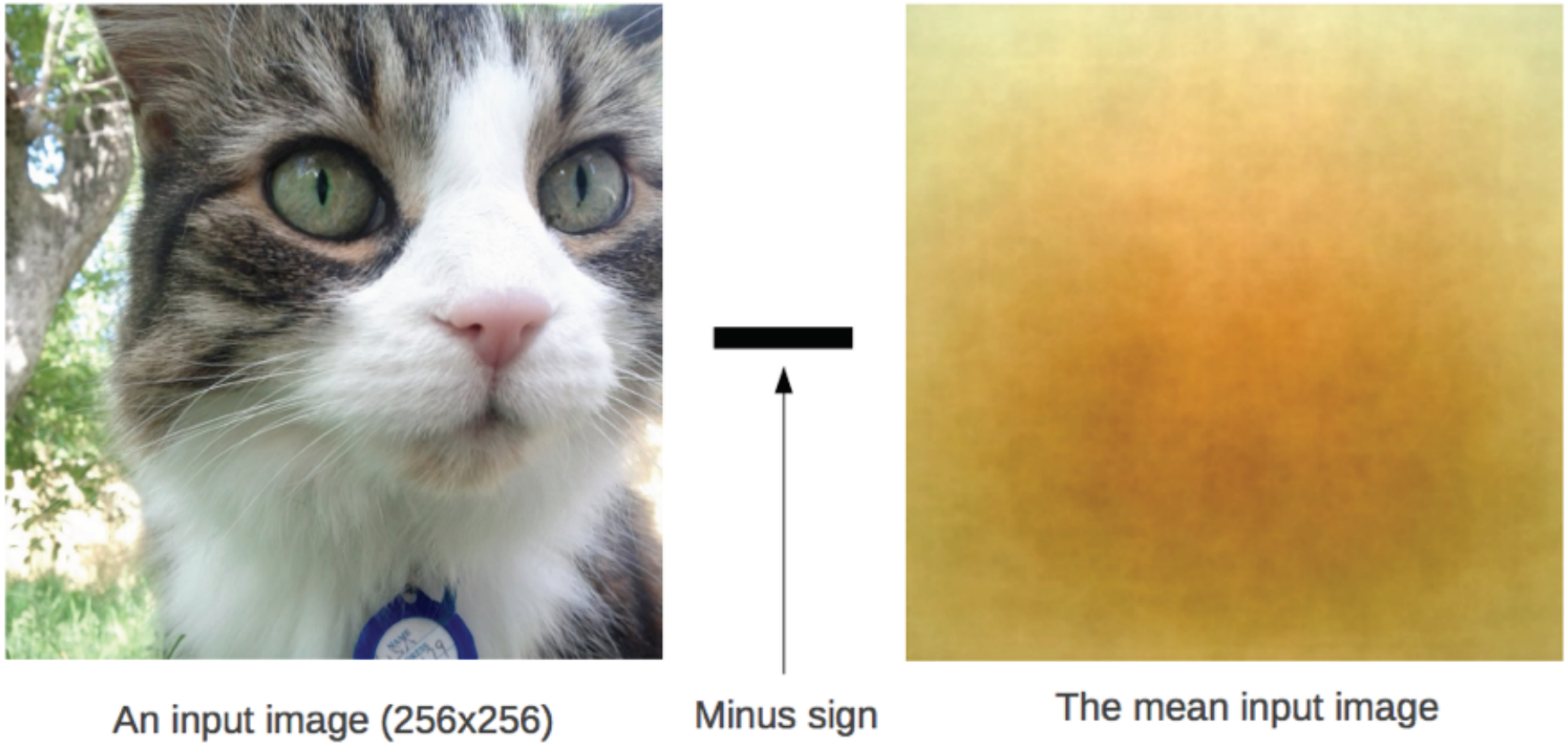

```
X -= np.mean(axis=0, keepdims=True)
```

```
X /= np.std(axis=0, keepdims=True)
```

Figure: Andrej Karpathy

Regularization: In Data Preparation

For ConvNets, typically only the mean is subtracted.



A per-channel mean also works (one value per R,G,B).

Figure: Alex Krizhevsky

Regularization: In Data Preparation

Augment the data — extract random crops from the input, with slightly jittered offsets. Without this, typical ConvNets (e.g. [Krizhevsky 2012]) overfit the data.

E.g. 224x224 patches extracted from 256x256 images

Randomly reflect horizontally

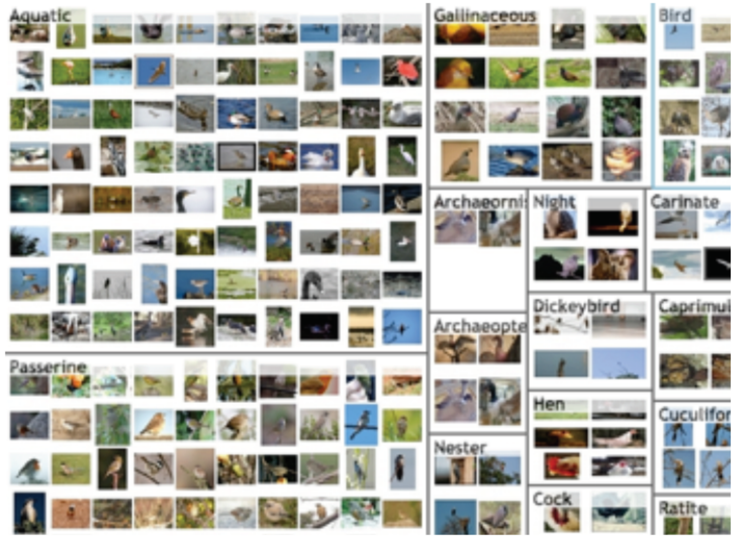
Perform the augmentation live during training

Figure: Alex Krizhevsky

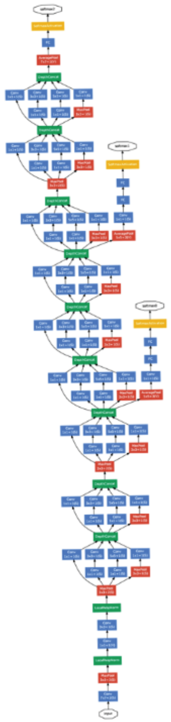
Putting It All Together: How To Train Deep Nets

Roughly speaking:

Gather labeled data



Find a ConvNet architecture



Minimize the loss

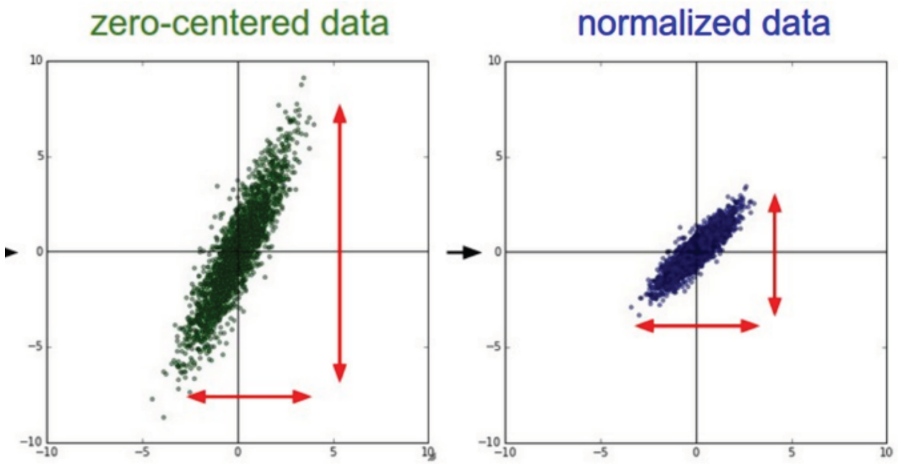
Training a Convolutional Neural Network

- Split and preprocess your data
- Choose your network architecture
- Initialize the weights
- Find a learning rate and regularization strength/strategy
- Minimize the loss and monitor the progress
- Fiddle with things until they work

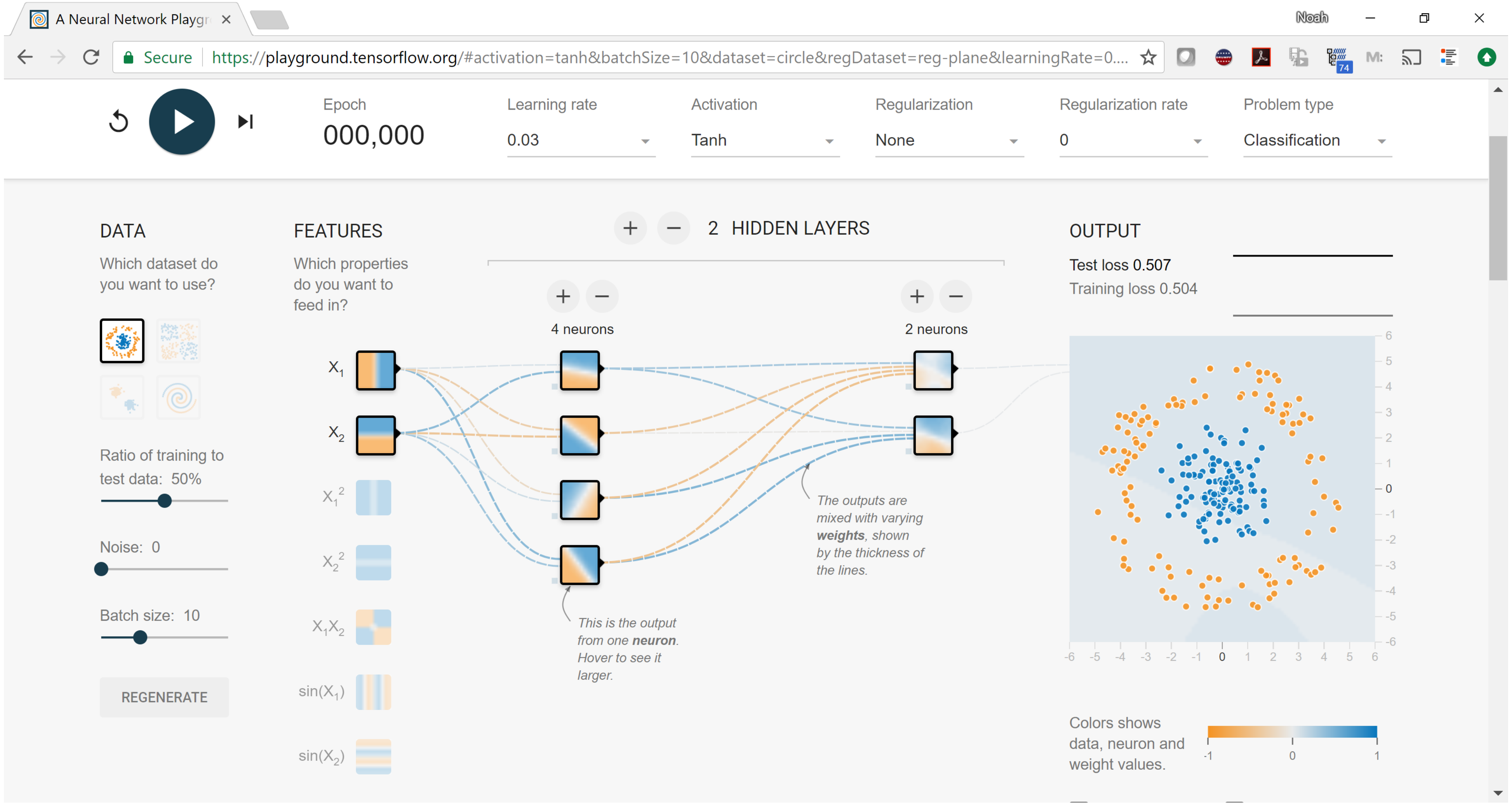
(1) Data Pre-Processing

Examples:

- Normalizing and centering Data
- Data Augmentation
  - Random Cropping
  - Mirror Flips



(2) Choose your architecture



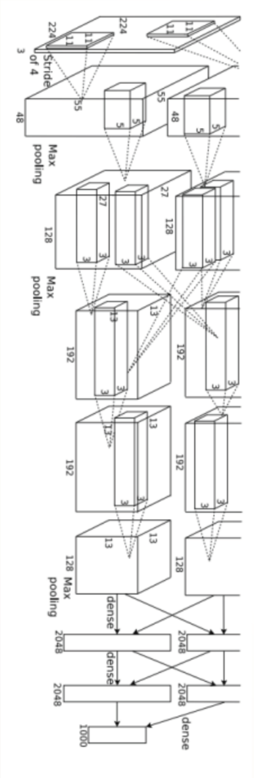
https://playground.tensorflow.org/

(we will come back to this later)

(2) Choose your architecture

Very common modern choice

"AlexNet"



[Krizhevsky et al. NIPS 2012]

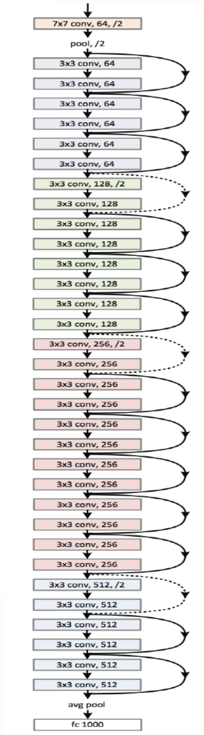
"GoogLeNet"

[Szegedy et al. CVPR 2015]

"VGG Net"

[Simonyan & Zisserman, ICLR 2015]

"ResNet"



[He et al. CVPR 2016]

(3) Initialize Your Weights

Set the weights to small random numbers:

```
W = np.random.randn(D, H) * 0.001
```

(matrix of small random numbers drawn from a Gaussian distribution)

Set the bias to zero (or small nonzero):

```
b = np.zeros(H)
```

(if you use ReLU activations, folks tend to initialize bias to small positive number)

Slide: Andrej Karpathy

(3) Start with a Small Portion of the Data

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
X_tiny = X_train[:20] # take 20 examples
y_tiny = y_train[:20]
best_model, stats = trainer.train(X_tiny, y_tiny, X_tiny, y_tiny,
                                  model, two_layer_net,
                                  num_epochs=200, reg=0.0,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = False,
                                  learning_rate=1e-3, verbose=True)
```

The above code:

- take the first 20 examples from CIFAR-10
- turn off regularization (reg = 0.0)
- use simple vanilla ‘sgd’

(3) Start with a Small Portion of the Data

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
X_tiny = X_train[:20] # take 20 examples
y_tiny = y_train[:20]
best_model, stats = trainer.train(X_tiny, y_tiny, X_tiny, y_tiny,
                                  model, two_layer_net,
                                  num_epochs=200, reg=0.0,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = False,
                                  learning_rate=1e-3, verbose=True)
```

Details:

'sgd': vanilla gradient descent (no momentum etc)

learning_rate_decay = 1: constant learning rate

sample_batches = False (full gradient descent, no batches)

epochs = 200: number of passes through the data

Slide: Andrej Karpathy

(3) Start with a Small Portion of the Data

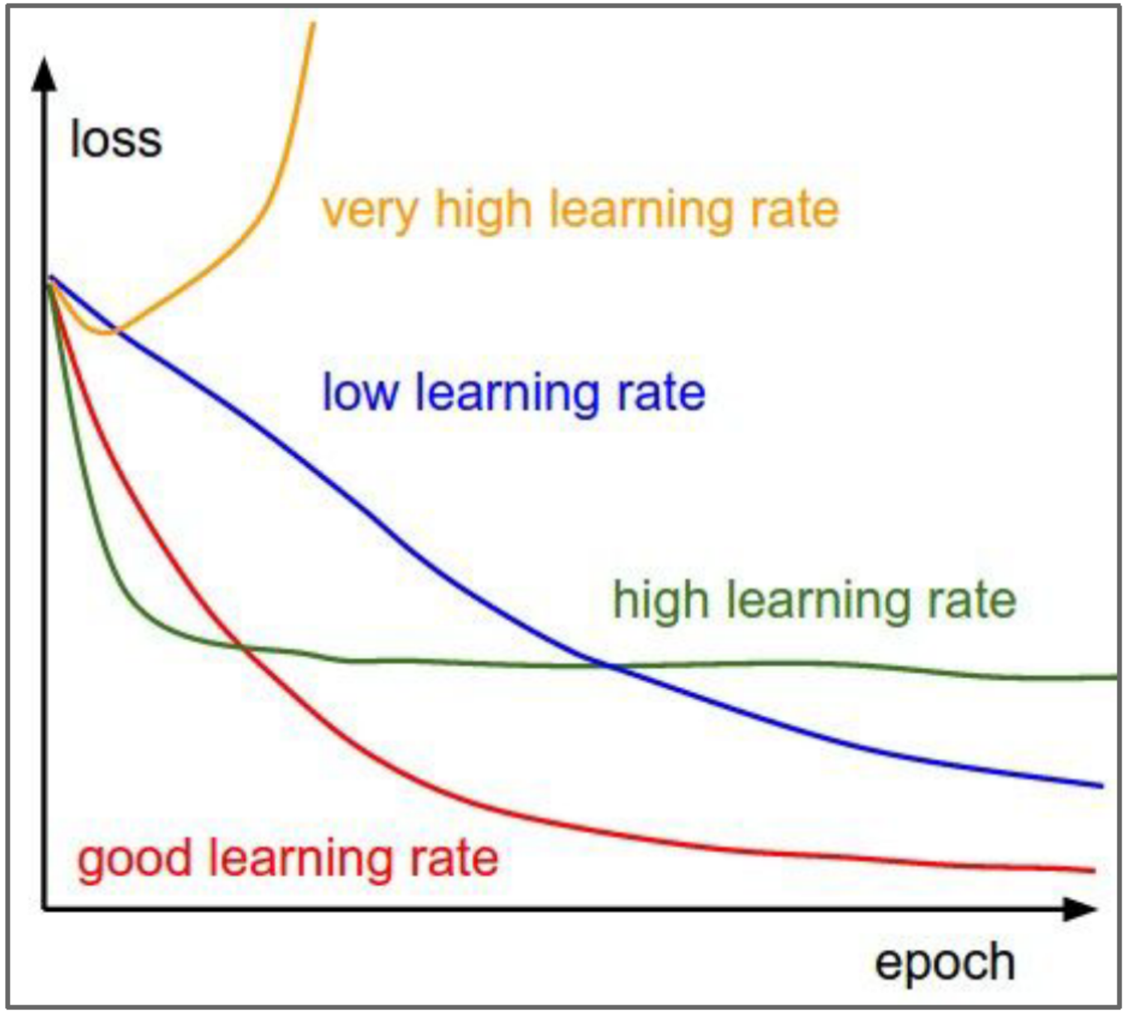
100% accuracy on the training set (good)

```
Finished epoch 1 / 200: cost 2.302603, train: 0.400000, val 0.400000, lr 1.000000e-03
Finished epoch 2 / 200: cost 2.302258, train: 0.450000, val 0.450000, lr 1.000000e-03
Finished epoch 3 / 200: cost 2.301849, train: 0.600000, val 0.600000, lr 1.000000e-03
Finished epoch 4 / 200: cost 2.301196, train: 0.650000, val 0.650000, lr 1.000000e-03
Finished epoch 5 / 200: cost 2.300044, train: 0.650000, val 0.650000, lr 1.000000e-03
Finished epoch 6 / 200: cost 2.297864, train: 0.550000, val 0.550000, lr 1.000000e-03
Finished epoch 7 / 200: cost 2.293595, train: 0.600000, val 0.600000, lr 1.000000e-03
Finished epoch 8 / 200: cost 2.285096, train: 0.550000, val 0.550000, lr 1.000000e-03
Finished epoch 9 / 200: cost 2.268094, train: 0.550000, val 0.550000, lr 1.000000e-03
Finished epoch 10 / 200: cost 2.234787, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 11 / 200: cost 2.173187, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 12 / 200: cost 2.076862, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 13 / 200: cost 1.974090, train: 0.400000, val 0.400000, lr 1.000000e-03
Finished epoch 14 / 200: cost 1.895885, train: 0.400000, val 0.400000, lr 1.000000e-03
Finished epoch 15 / 200: cost 1.820876, train: 0.450000, val 0.450000, lr 1.000000e-03
Finished epoch 16 / 200: cost 1.737430, train: 0.450000, val 0.450000, lr 1.000000e-03
Finished epoch 17 / 200: cost 1.642356, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 18 / 200: cost 1.535239, train: 0.600000, val 0.600000, lr 1.000000e-03
Finished epoch 19 / 200: cost 1.421527, train: 0.600000, val 0.600000, lr 1.000000e-03

Finished epoch 195 / 200: cost 0.002694, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 196 / 200: cost 0.002674, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 197 / 200: cost 0.002655, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 198 / 200: cost 0.002635, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 199 / 200: cost 0.002617, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 200 / 200: cost 0.002597, train: 1.000000, val 1.000000, lr 1.000000e-03
finished optimization. best validation accuracy: 1.000000
```

Slide: Andrej Karpathy

(4) Find a learning rate

- Too high won't converge
- Too low will converge slowly



Aside: Some Training Vocabulary

- An ***Epoch*** is one complete pass through your training data

- An ***iteration*** of SGD happens on a batch of examples.

- The ***Batch Size*** is the number of examples in a single training batch.

- The number of iterations per epoch depends on the total number of examples divided by the batch size.

(4b) Choosing a Learning Rate Schedule

How do we change the learning rate over time?

Various choices:

- Step down by a factor of 0.1 every 50,000 mini-batches (used by SuperVision [Krizhevsky 2012])
- Decrease by a factor of 0.97 every epoch (used by GoogLeNet [Szegedy 2014])
- Scale by sqrt(1-t/max_t) (used by BVLC to re-implement GoogLeNet)
- Scale by 1/t
- Scale by exp(-t)

Summary of things to fiddle

- Network architecture
- Learning rate, decay schedule, update type
- Regularization (L2, L1, maxnorm, dropout, …)
- Loss function (softmax, SVM, …)
- Weight initialization

Neural network parameters

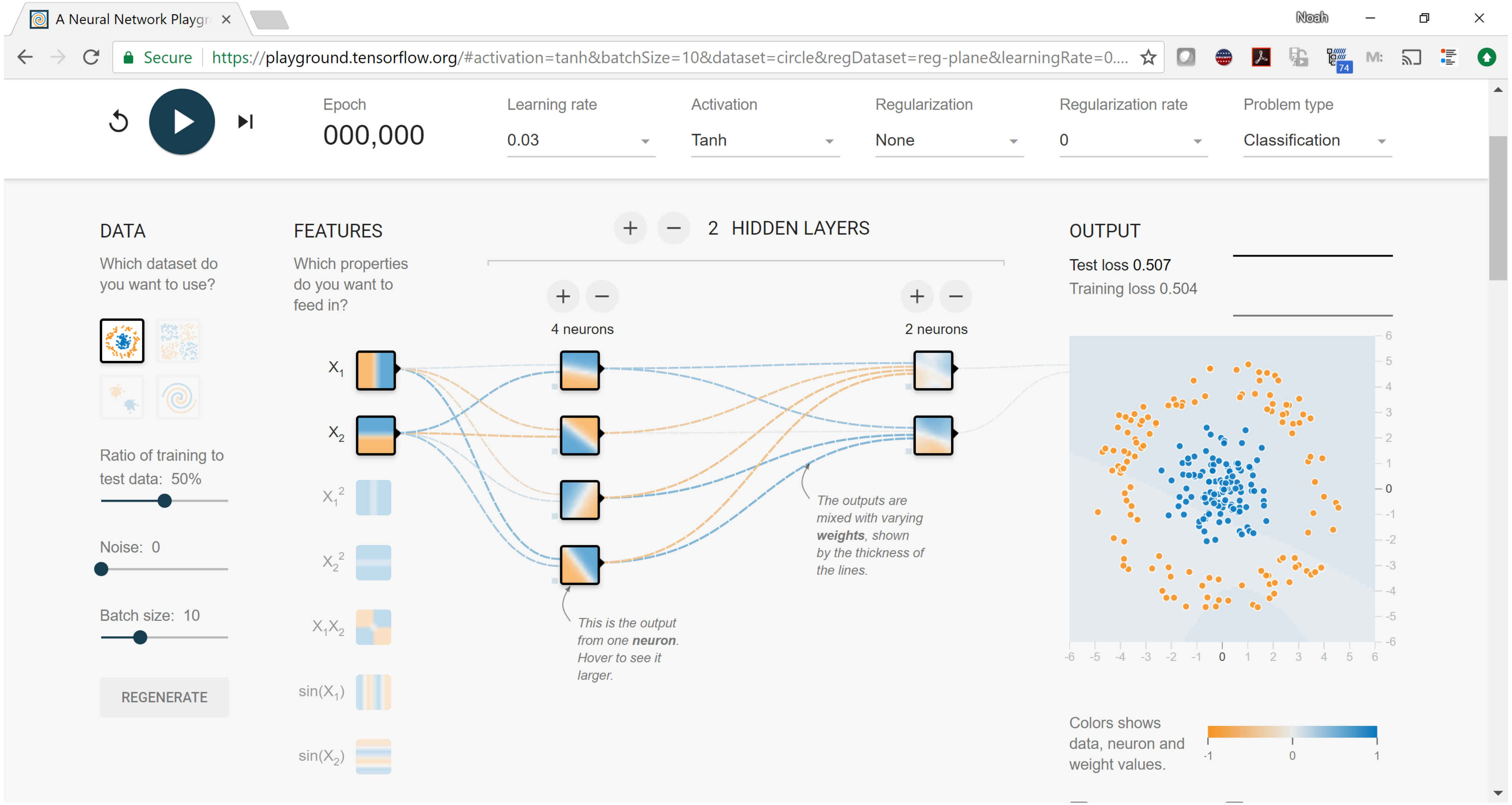
Summary of things to fiddle

- Network architecture
- Learning rate, decay schedule, update type (+batch size)
- Regularization (L2, L1, maxnorm, dropout, …)
- Loss function (softmax, SVM, …)
- Weight initialization

Neural network parameters

Questions?

Demo



https://playground.tensorflow.org/

(we will come back to this later)

Transfer Learning

"You need a lot of a data if you want to train/use CNNs"

Transfer Learning

"You need a lot of a data if you want to train/use CNNs"

BUSTED

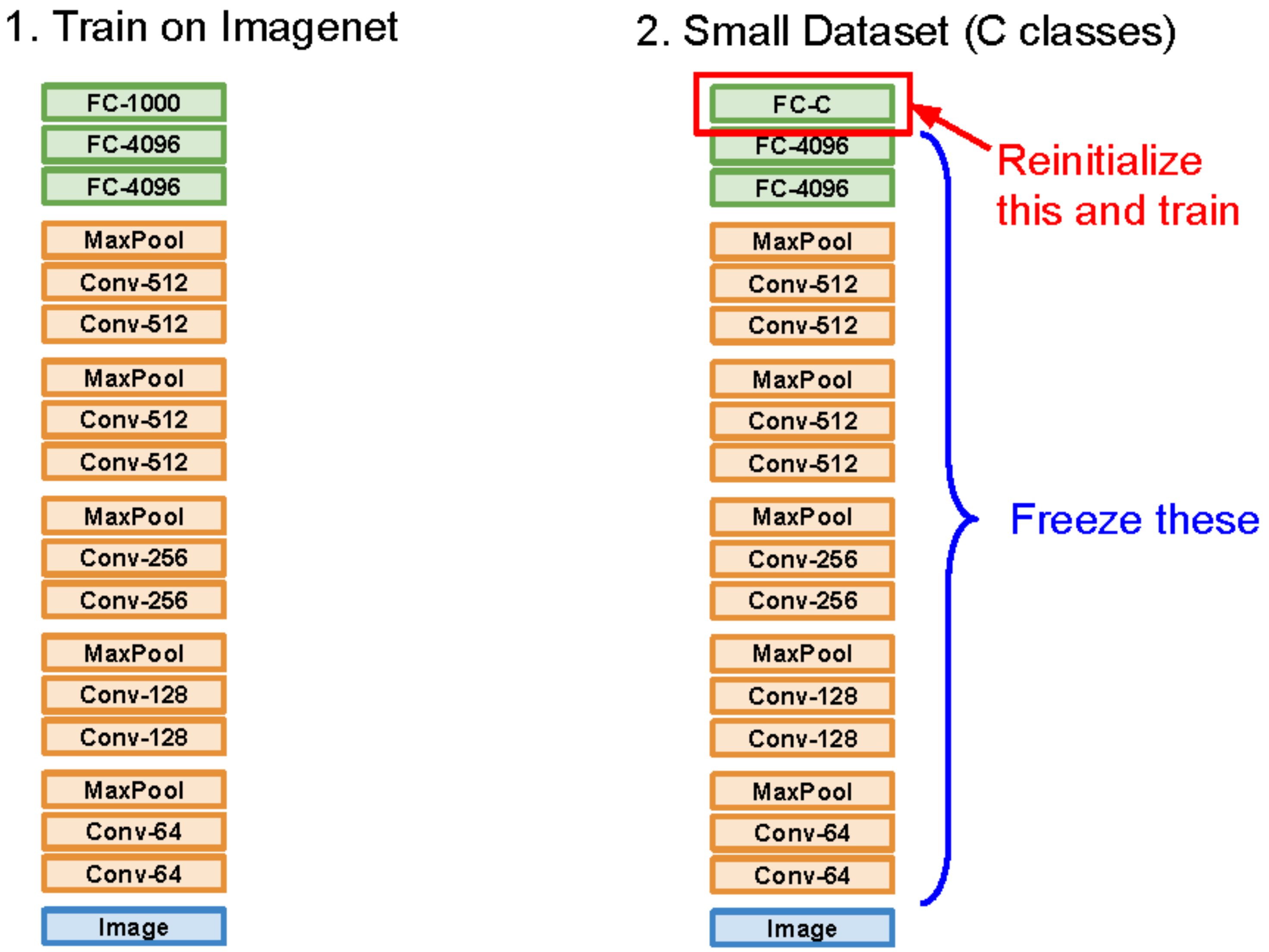
Transfer Learning with CNNs

Donahue et al, "DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition", ICML 2014
Razavian et al, "CNN Features Off-the-Shelf: An Astounding Baseline for Recognition", CVPR Workshops 2014

1. Train on Imagenet

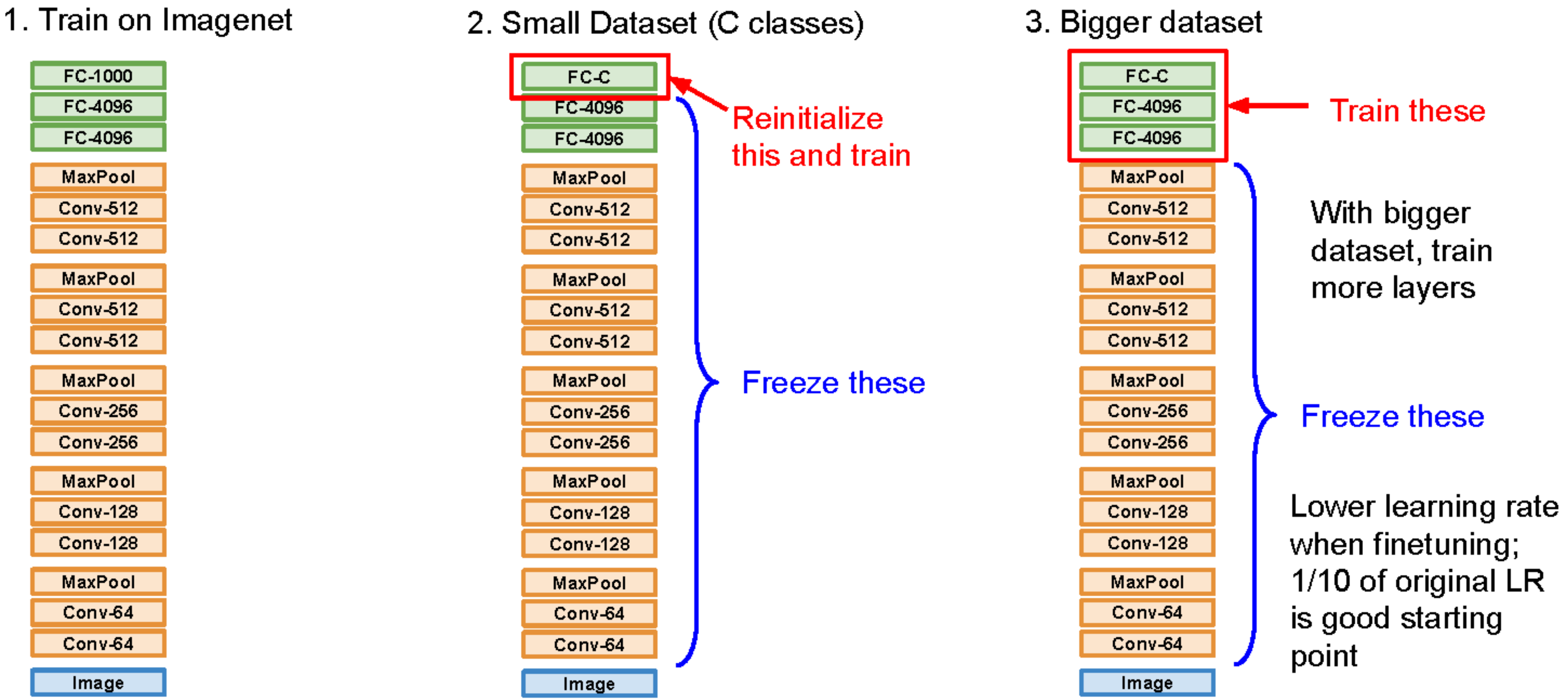
Transfer Learning with CNNs

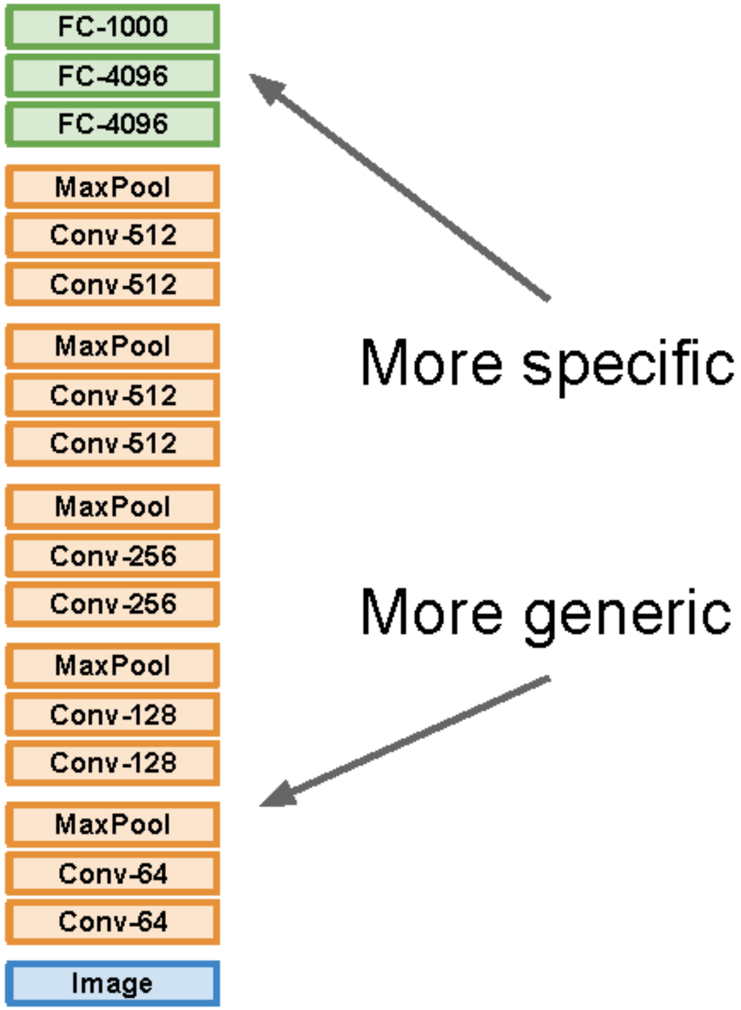
Donahue et al, "DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition", ICML 2014
Razavian et al, "CNN Features Off-the-Shelf: An Astounding Baseline for Recognition", CVPR Workshops 2014



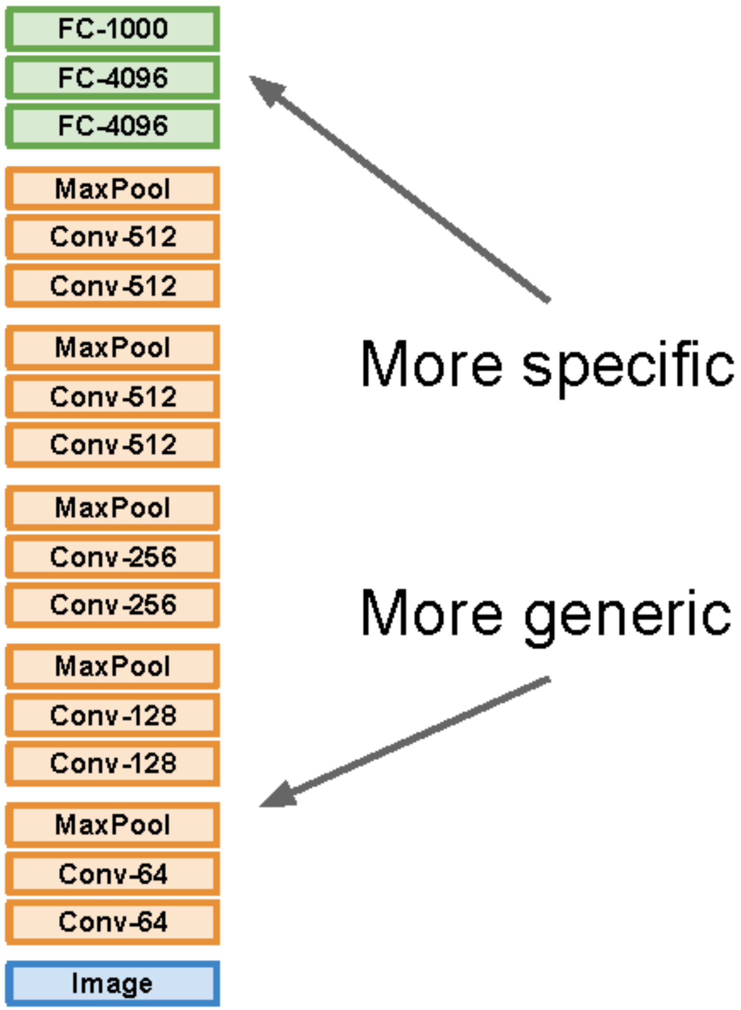
Transfer Learning with CNNs

Donahue et al, "DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition", ICML 2014
Razavian et al, "CNN Features Off-the-Shelf: An Astounding Baseline for Recognition", CVPR Workshops 2014

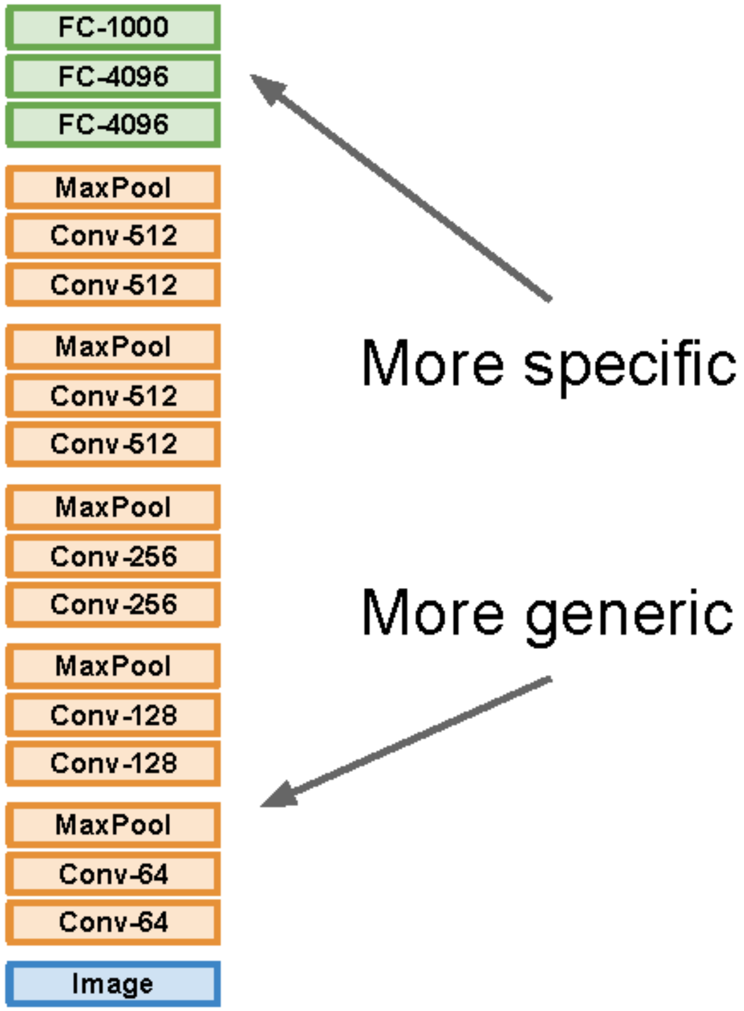




	very similar dataset	very different dataset
very little data	?	?
quite a lot of data	?	?



	very similar dataset	**very different dataset**
very little data	Use Linear Classifier on top layer	?
quite a lot of data	Finetune a few layers	?



	very similar dataset	**very different dataset**
very little data	Use Linear Classifier on top layer	You're in trouble… Try linear classifier from different stages
quite a lot of data	Finetune a few layers	Finetune a larger number of layers

Transfer learning with CNNs is pervasive…
(it's the norm, not an exception)

Object Detection
(Fast R-CNN)

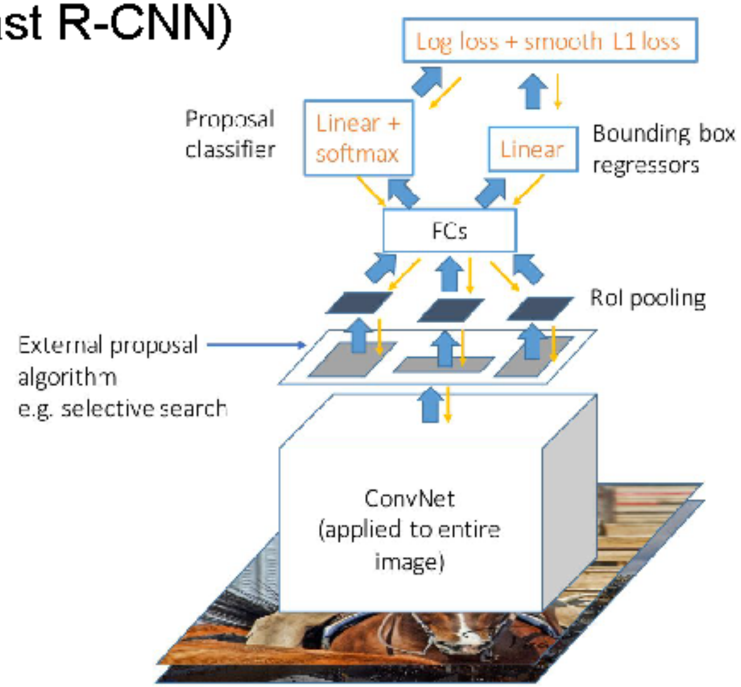
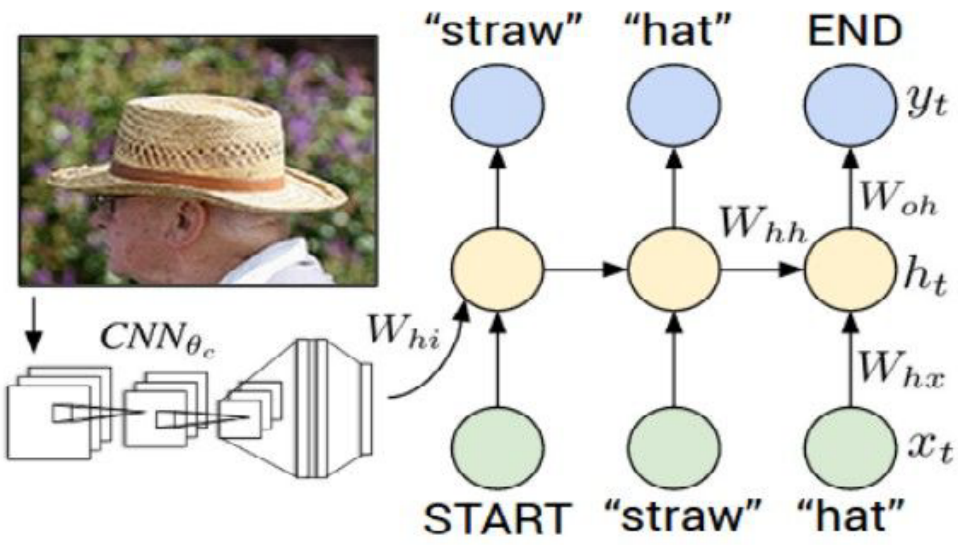


Image Captioning: CNN + RNN

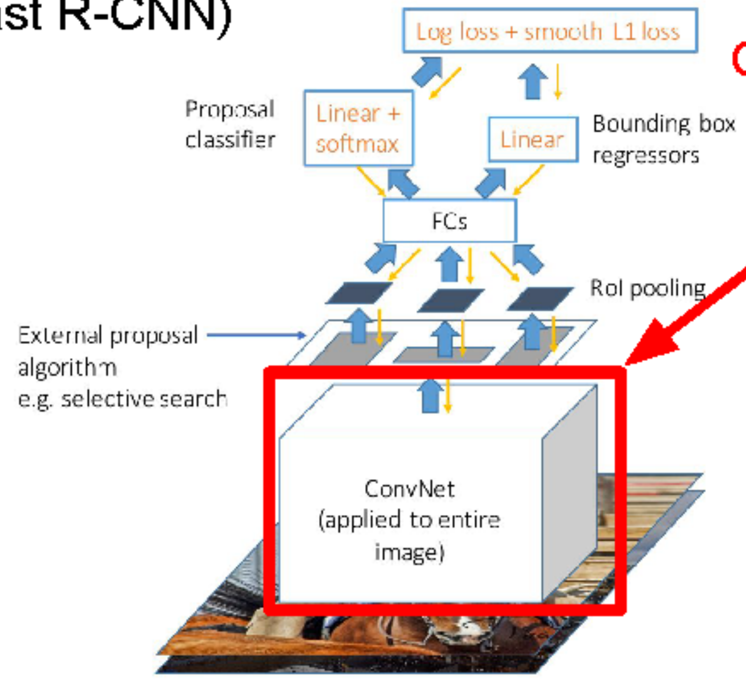


Girshick, "Fast R-CNN", ICCV 2015

Karpathy and Fei-Fei, "Deep Visual-Semantic Alignments for Generating Image Descriptions", CVPR 2015

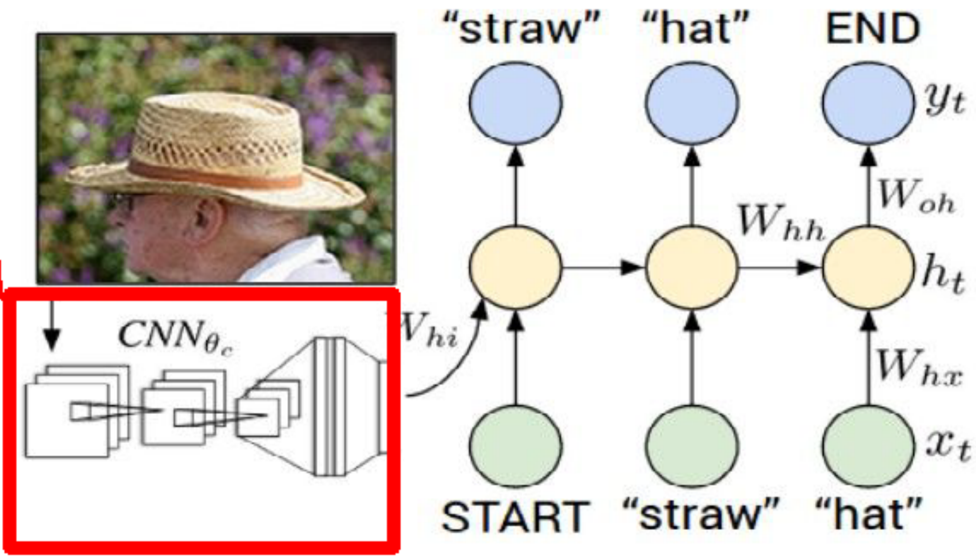
Transfer learning with CNNs is pervasive…
(it's the norm, not an exception)

Object Detection
(Fast R-CNN)



CNN pretrained on ImageNet

Image Captioning: CNN + RNN

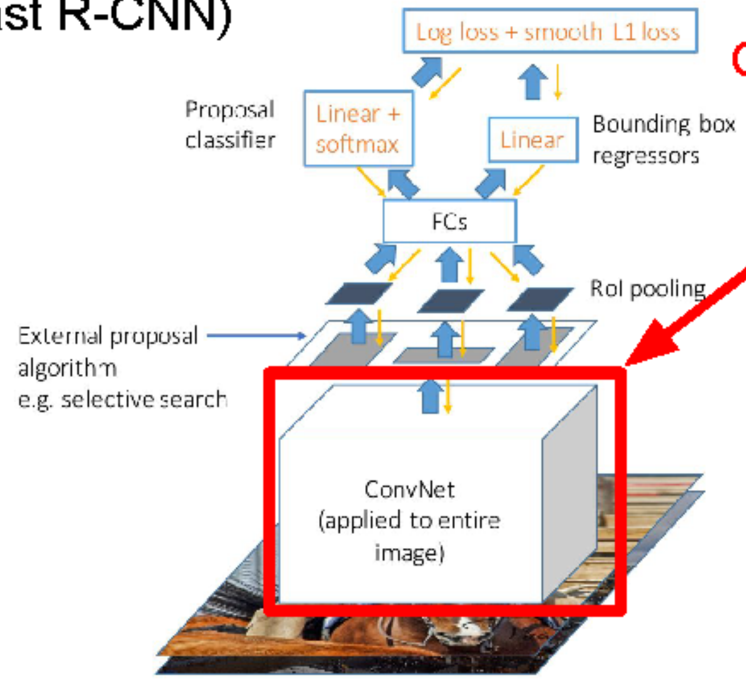


Girshick, "Fast R-CNN", ICCV 2015

Karpathy and Fei-Fei, "Deep Visual-Semantic Alignments for Generating Image Descriptions", CVPR 2015

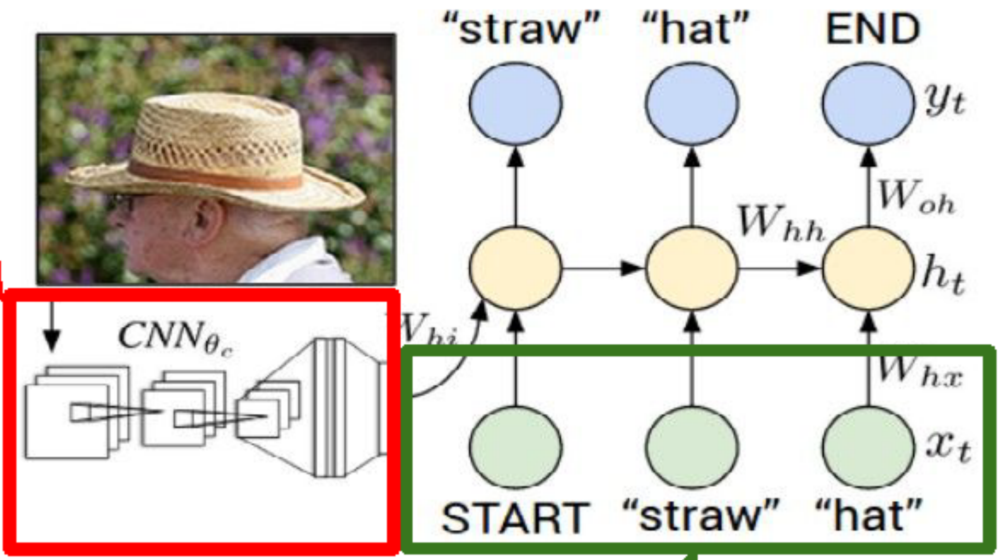
Transfer learning with CNNs is pervasive…
(it's the norm, not an exception)

Object Detection
(Fast R-CNN)



CNN pretrained on ImageNet

Image Captioning: CNN + RNN



Word vectors pretrained with word2vec

Girshick, "Fast R-CNN", ICCV 2015

Karpathy and Fei-Fei, "Deep Visual-Semantic Alignments for Generating Image Descriptions", CVPR 2015

Some Takeaways

Have some dataset of interest but it has < ~1M images?

1. Find a very large dataset that has similar data, train a big ConvNet there
2. Transfer learn to your dataset

Deep learning frameworks provide a "Model Zoo" of pretrained models so you don't need to train your own

TensorFlow: https://github.com/tensorflow/models
PyTorch: https://github.com/pytorch/vision

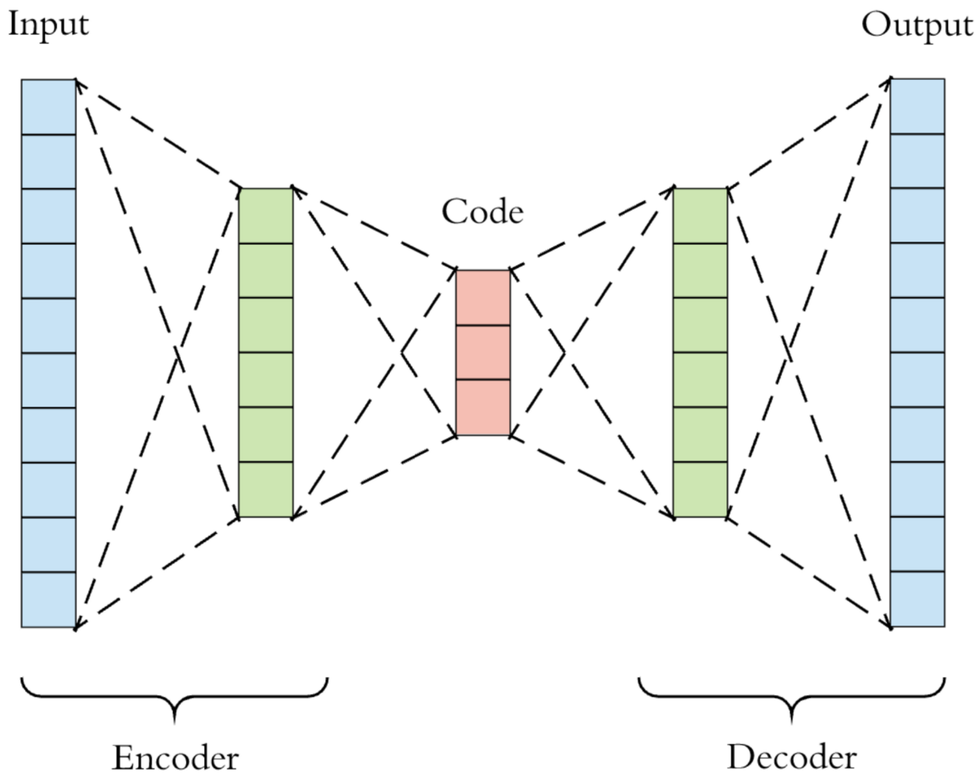
Common modern approach: start with a ResNet architecture pre-trained on ImageNet, and fine-tune on your (smaller) dataset

Questions?

Autoencoders: Unsupervised Dimensionality Reduction

- Learn a transformation into some compressed space (encoder)
- Learn a transformation from compressed space back to original content (decoder)
- Loss function can be difference between input and decoded output

- **Does not require labels!**



Autoencoders: Unsupervised Dimensionality Reduction

- Good way to learn useful features from large amounts of unlabeled data
  - E.g., for transfer learning

- We can do this with CNNs, but we need some way to expand feature dimensionality…
- For this we will use *Transpose Convolution*

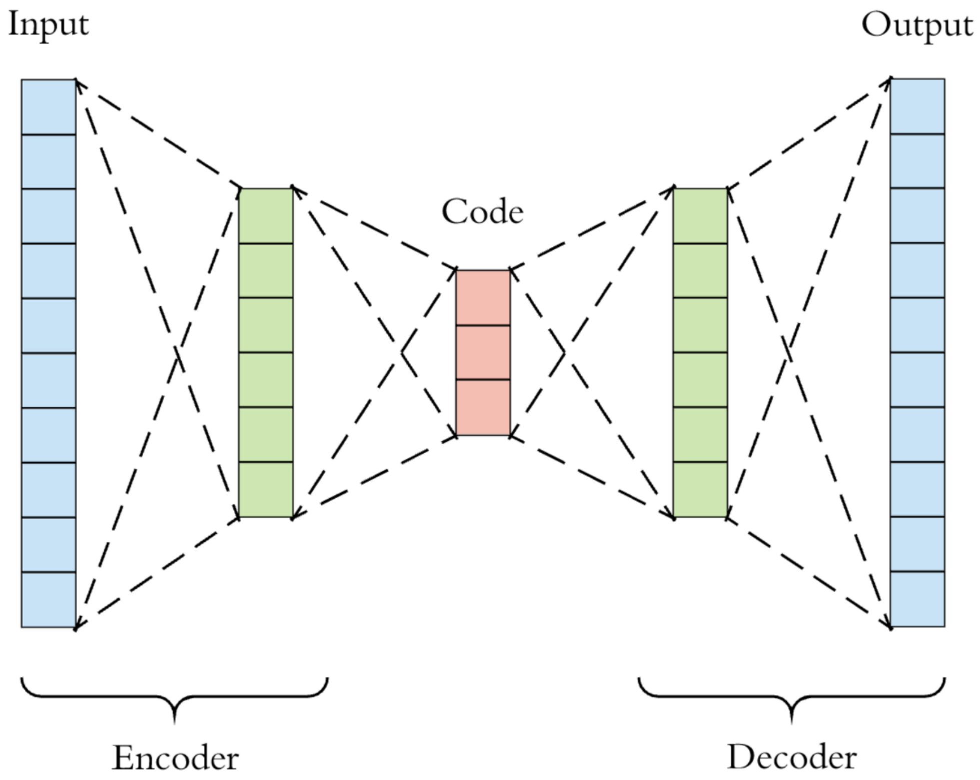


IMAGE COLORING

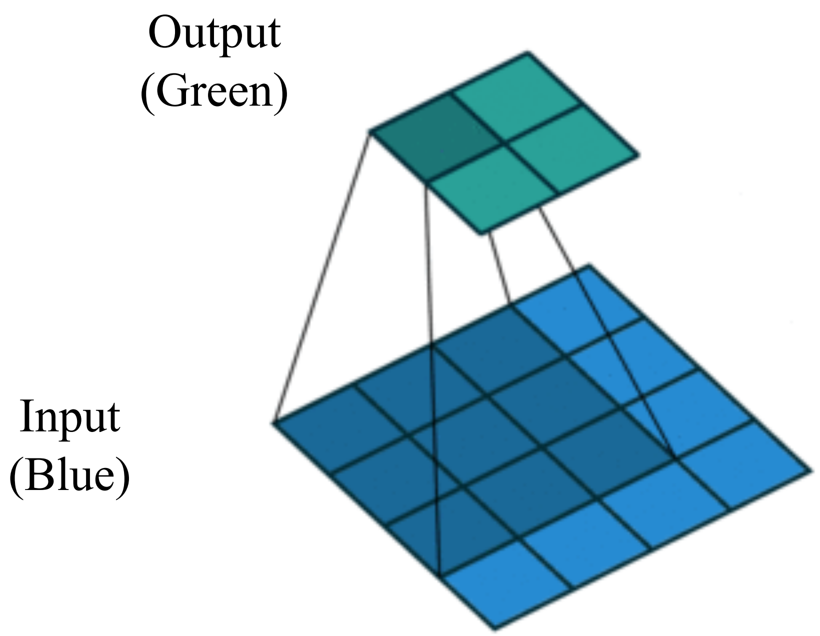
Before After

IMAGE NOISE REDUCTION

Before After

Regular Convolution

- ***Stride***: The step size used when computing the convolution
- ***Padding***: What is assumed about pixels “outside” of image bounds

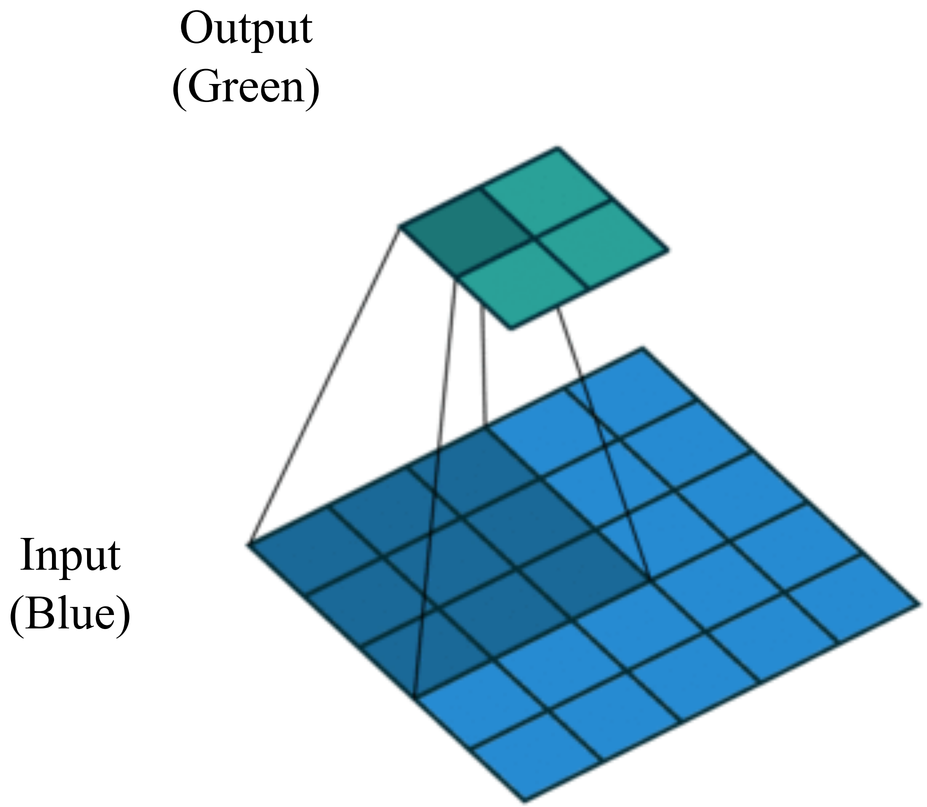


Kernel size: 3x3
Padding: 0
Stride: 0

Animations from: https://github.com/vdumoulin/conv_arithmetic

Regular Convolution

- ***Stride***: The step size used when computing the convolution
- ***Padding***: What is assumed about pixels “outside” of image bounds

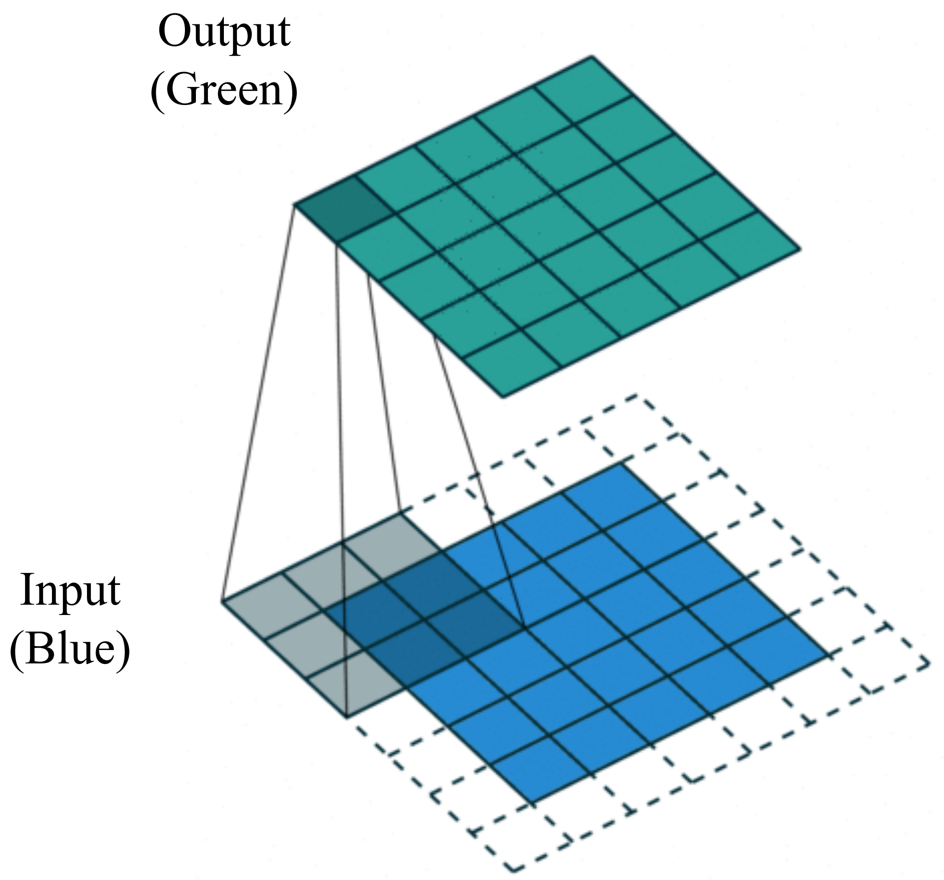


Kernel size: 3x3
Padding: 0
Stride: 1

Animations from: https://github.com/vdumoulin/conv_arithmetic

Regular Convolution

- ***Stride***: The step size used when computing the convolution
- ***Padding***: What is assumed about pixels "outside" of image bounds



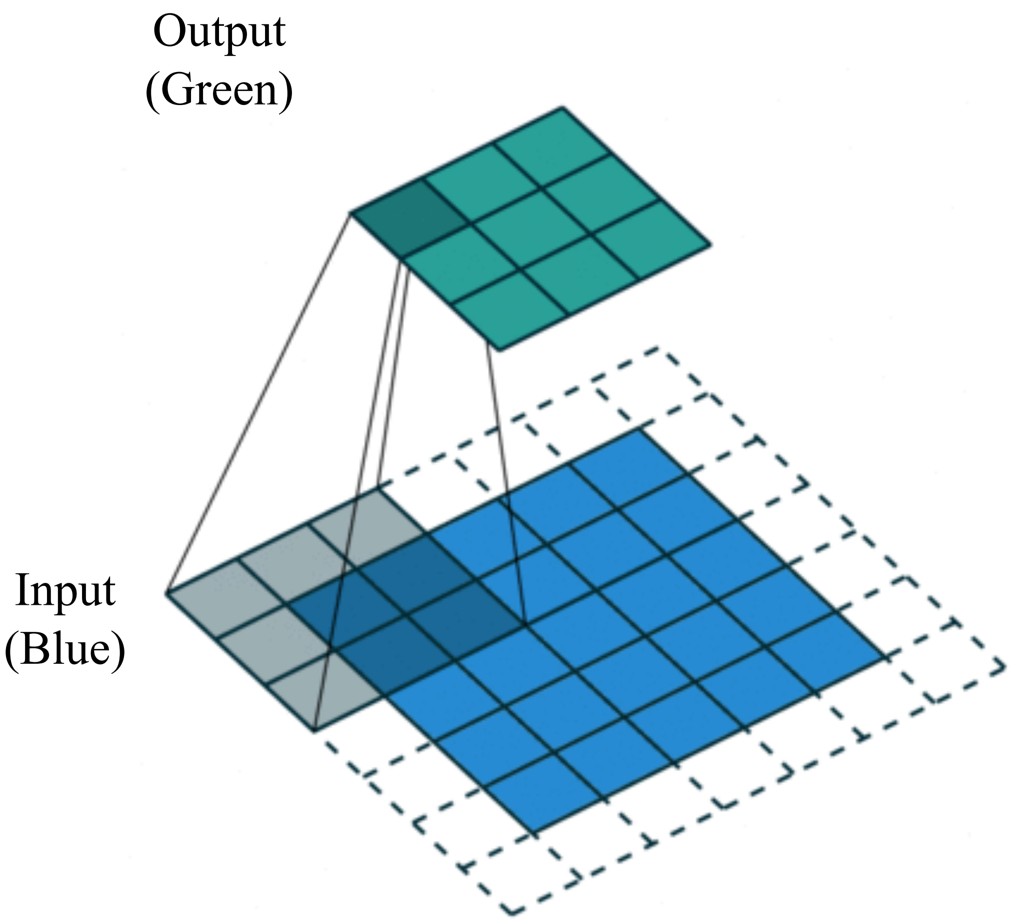
Kernel size: 3x3
Padding: "same" (1)
Stride: 0

Animations from: https://github.com/vdumoulin/conv_arithmetic

Regular Convolution

- ***Stride***: The step size used when computing the convolution
- ***Padding***: What is assumed about pixels "outside" of image bounds

- Stride is applied to the output and padding is applied to the input

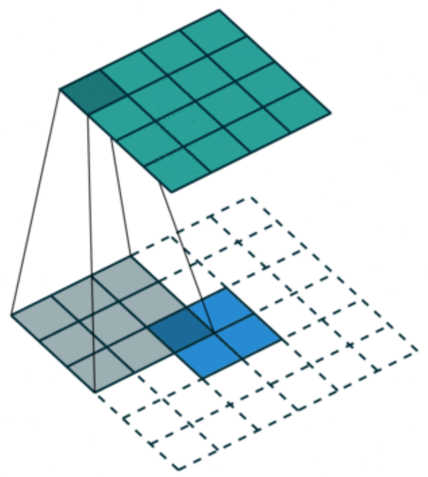


Kernel size: 3x3
Padding: 1
Stride: 1

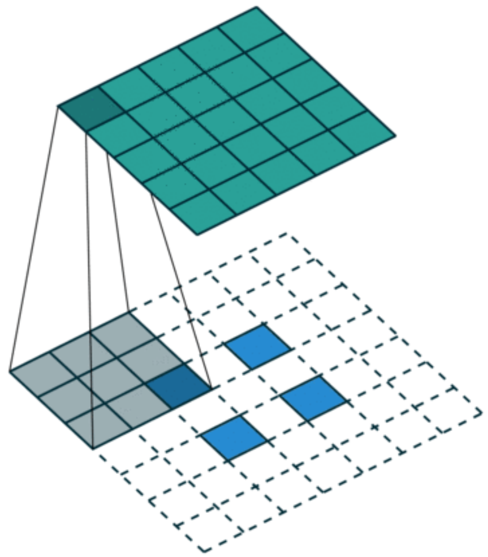
Animations from: https://github.com/vdumoulin/conv_arithmetic

Transpose Convolution: Upscaling Our Data

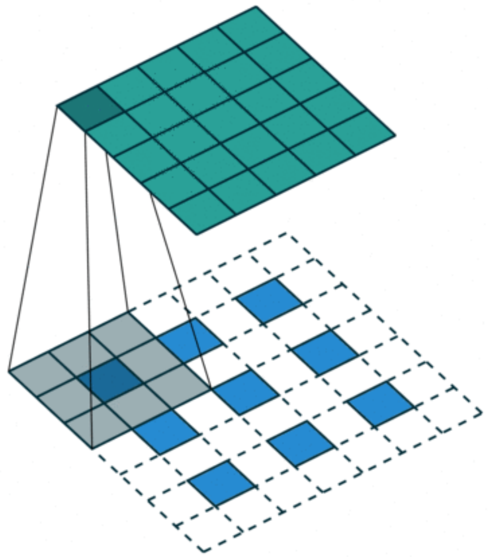
- Stride applied to input
- Padding applied to output (think of it as removing boundary pixels)



Kernel size: 3x3
Padding: 0
Stride: 0



Kernel size: 3x3
Padding: 0
Stride: 1



Kernel size: 3x3
Padding: 1
Stride: 1

Animations from: https://github.com/vdumoulin/conv_arithmetic

Generative Models

Abe Davis

Some slides from Jin Sun, Phillip Isola