CS5670: Computer Vision

Training Deep Networks

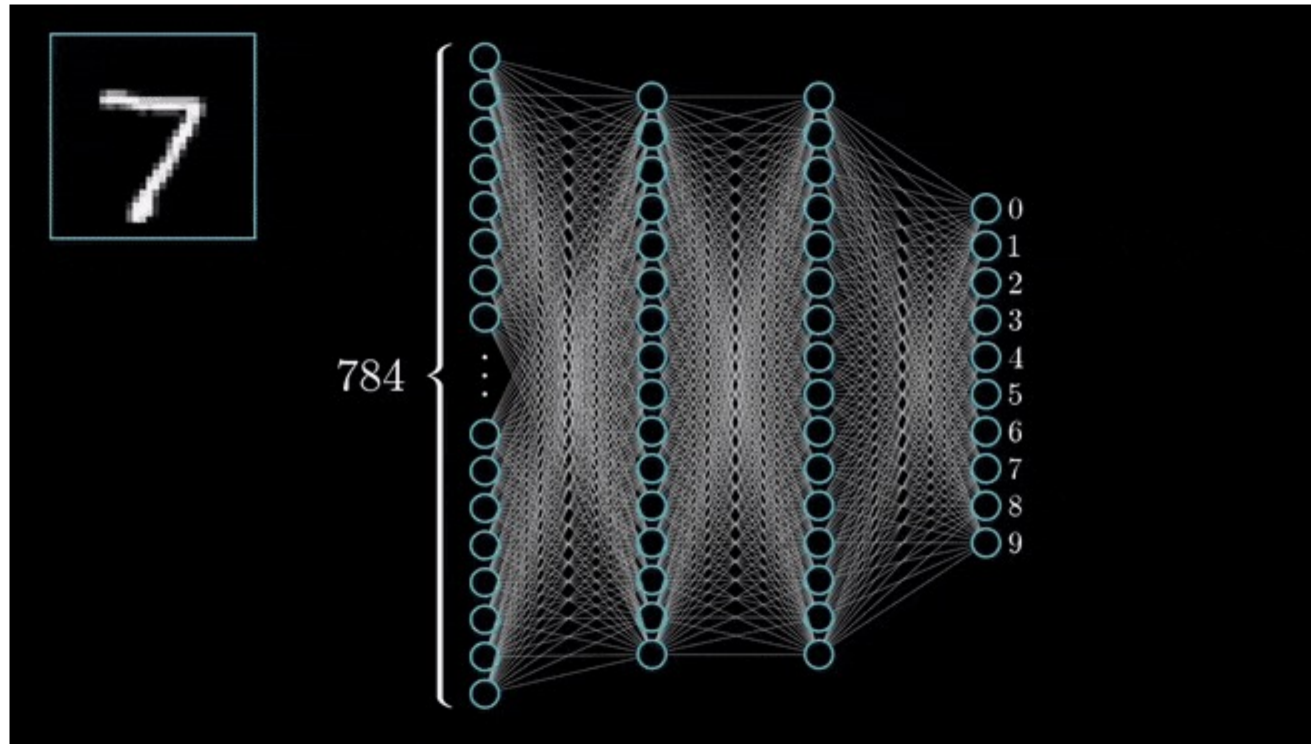


Image credit: https://blog.imarticus.org/what-are-some-tips-and-tricks-for-training-deep-neural-networks/

Some content adapted from material from Andrej Karpathy, Sean Bell, Kavita Bala, and

Announcements

- Project 5 (Neural Radiance Fields) due Weds, May 3 by 8pm
- In class final on May 9
  - Open book, open note
- Course evaluations are open starting Monday, May 1
  - We would love your feedback!
  - Small amount of extra credit for filling out
    - What you write is still anonymous, instructors only see whether students filled it out
  - Link coming soon

Readings

- Convolutional neural networks
  - Szeliski (2nd Edition) Chapter 5.4
- Neural Rendering
  - Szeliski (2nd Edition) Chapter 14.6
- Best practices for training CNNs
  - http://cs231n.github.io/neural-networks-2/
  - http://cs231n.github.io/neural-networks-3/

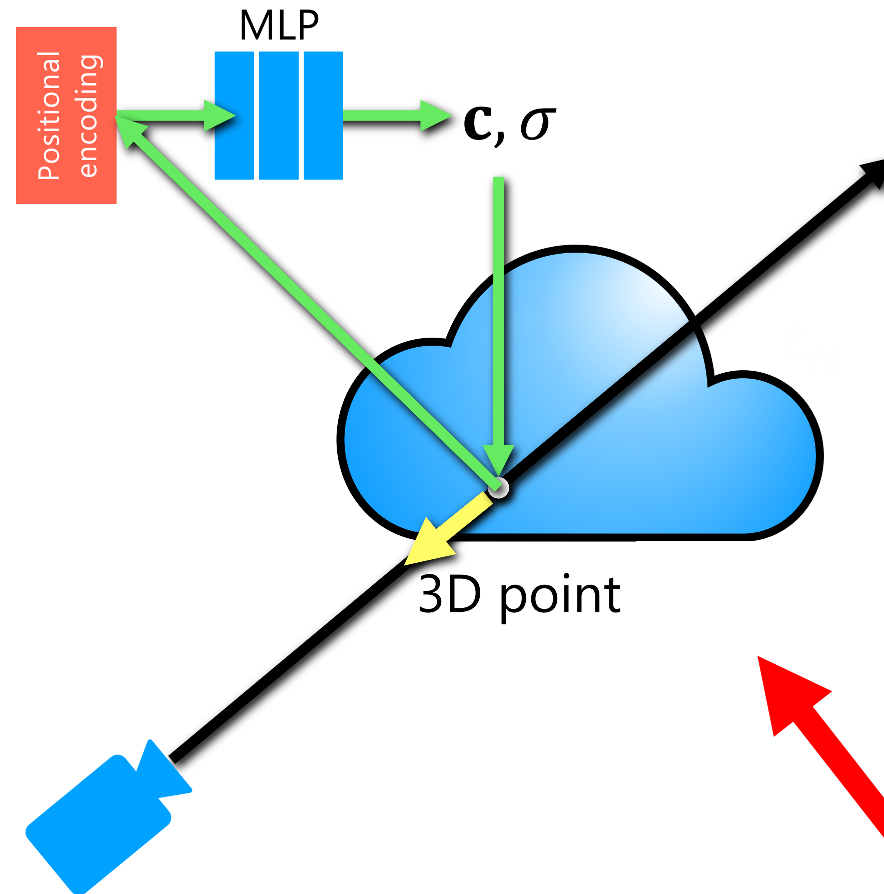
NeRF Recap

NeRF: Summary

- Represent the scene as volumetric colored “fog”
- Store the fog color and density at each point as an MLP mapping 3D position (x, y, z) to color **c** and density σ
- Render image by shooting a ray through the fog for each pixel and accumulating a color
- Optimize MLP parameters by rendering to a set of known viewpoints and comparing to ground truth images
- Can think of this as a learning problem where we train to reproduce the known images, and generalize to new views

NeRF Results

Extension: view-dependent neural field



Include the ray direction in the input to the MLP allows for capturing and rendering view-dependent effects (e.g., shiny surfaces)

NeRF encodes convincing view-dependent effects using directional dependence

Adapted from material from Pratul Sriniva

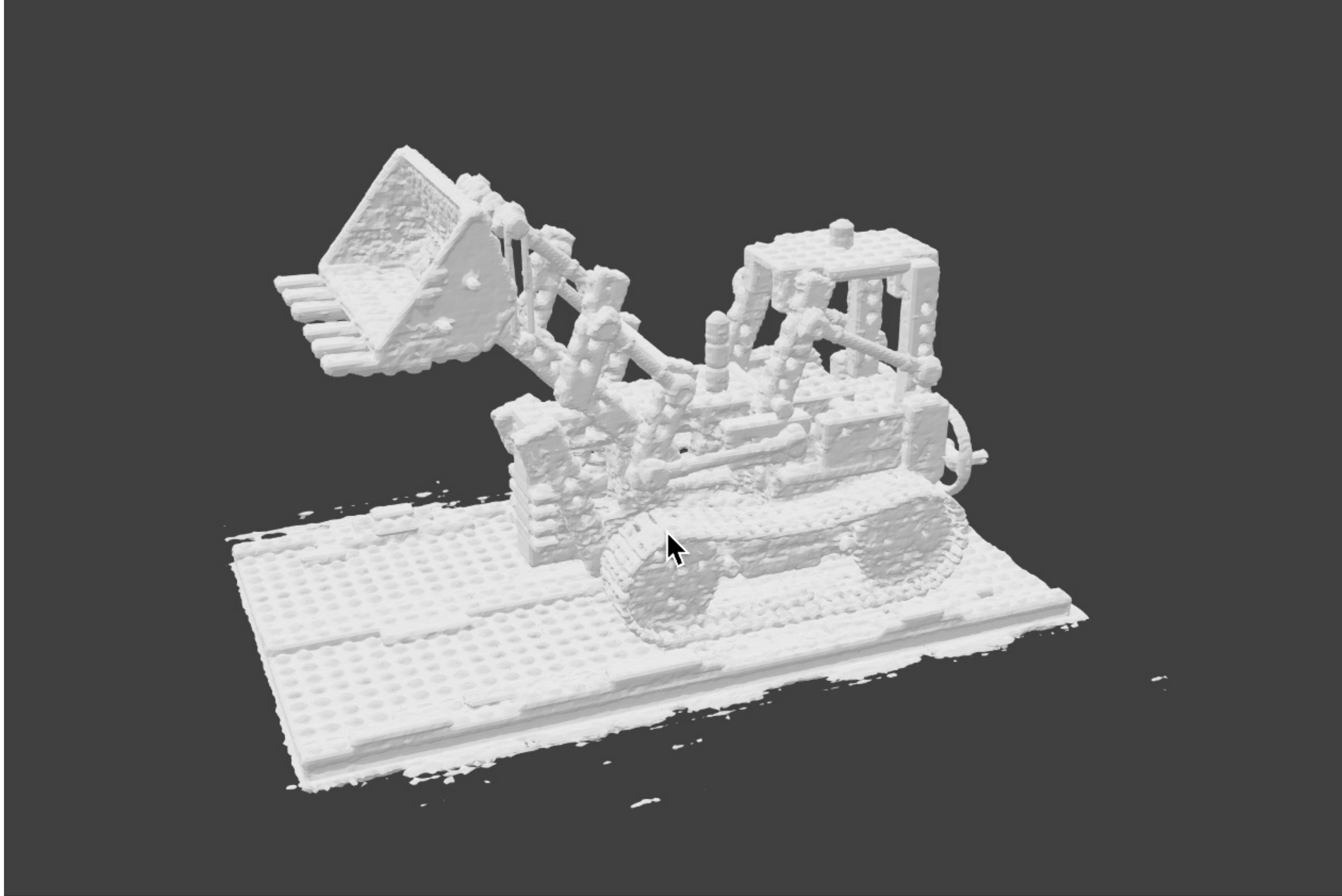
NeRF encodes convincing view-dependent effects using directional dependence

Adapted from material from Pratul Sriniva

NeRF encodes detailed scene geometry with occlusion effects

Adapted from material from Pratul Sriniva

NeRF encodes detailed scene geometry



Adapted from material from Pratul Sriniva

Extension: Mip-NeRF 360

Mip-NeRF 360: Unbounded Anti-Aliased Neural Radiance Fields
Jonathan T. Barron, Ben Mildenhall, Dor Verbin, Pratul P. Srinivasan, and Peter Hedman,
CVPR 2022

Extension: NeRF in the Wild (NeRF-W)

Brandenburg Gate

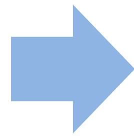
Sacre Coeur

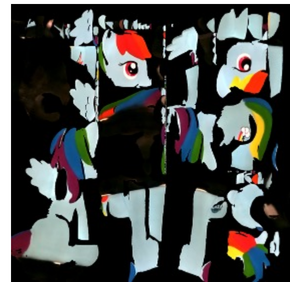
Trevi Fountain

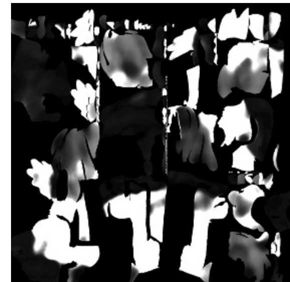
Martin-Brualla*, Radwan*, Sajjadi*, Barron, Dosovitskiy, Duckworth. *NeRF in the Wild*. CVPR 2021.
https://www.youtube.com/watch?v=mRAKVQj5LRA

Inverse graphics beyond shape and color

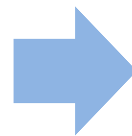
Input images of
an object







Reconstructed shape,
albedo, and materials



Reconstructed models inserted into scene with new lighting

Zhang, Luan, Li, Snavely. CVPR 2022

Questions?

Deep networks can be used for...

Image classification

f( ) = “apple”

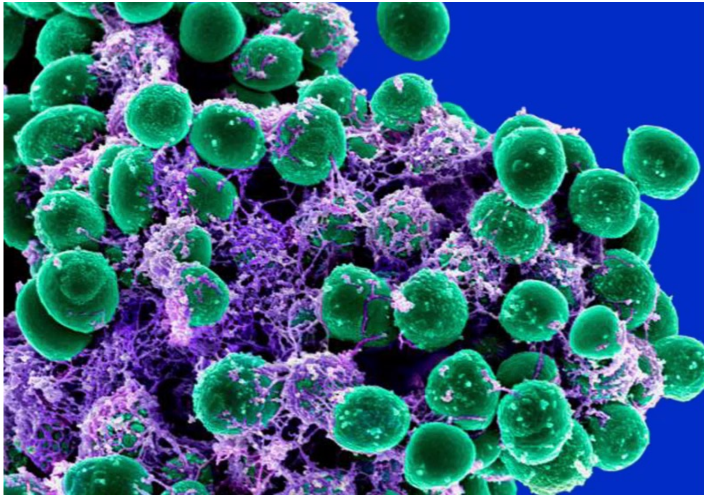
f( ) = “tomato”

f( ) = “cow”

View synthesis

And much more!

A Recent Example: *Segment Anything*

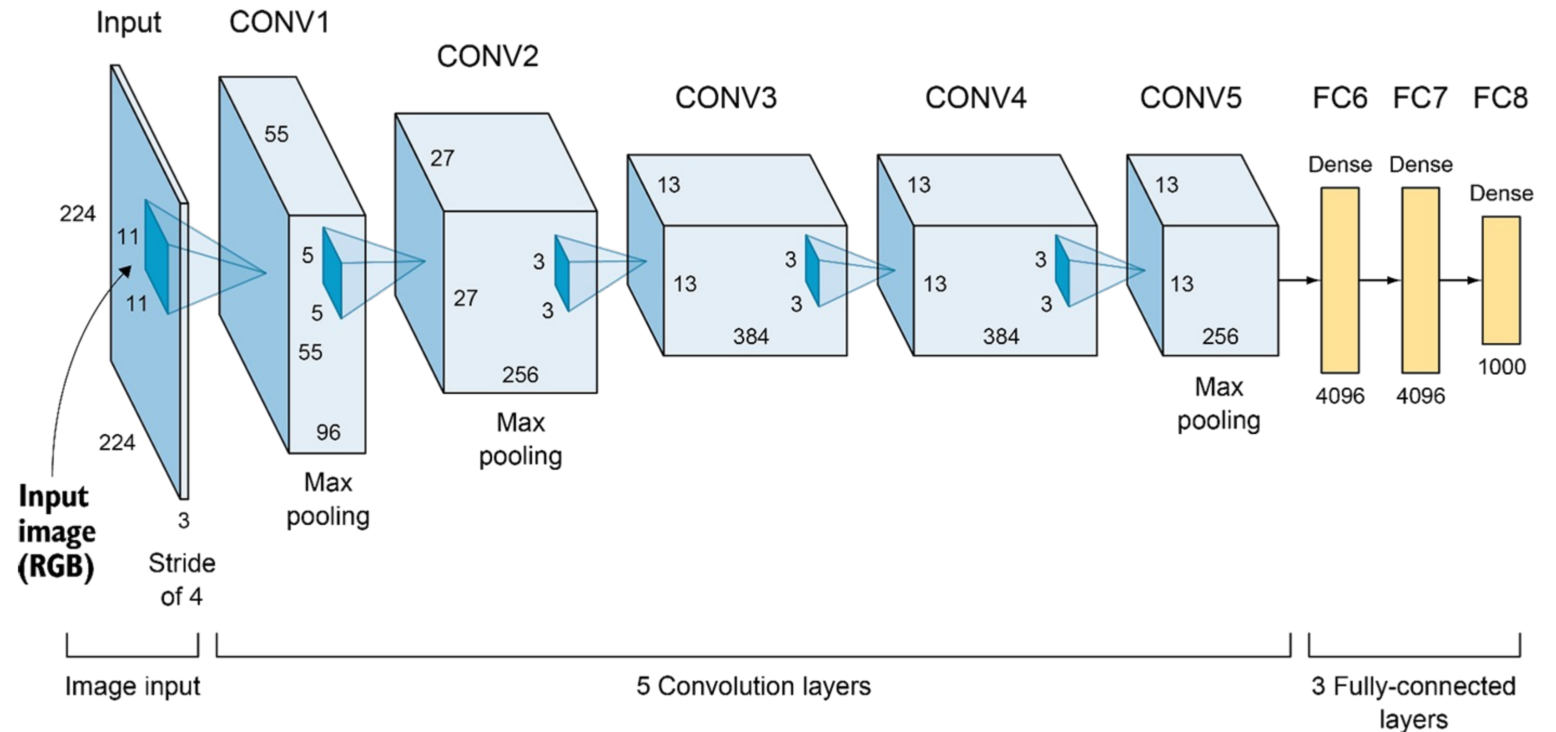


Segment Anything
Alexander Kirillov, Eric Mintun, Nikhila Ravi, Hanzi Mao, Chloe Rolland, Laura Gustafson, Tete Xiao, Spencer Whitehead, Alexander C. Berg, Wan-Yen Lo, Piotr Dollár, Ross Girshick

Back to convolutional neural networks

Layer types:

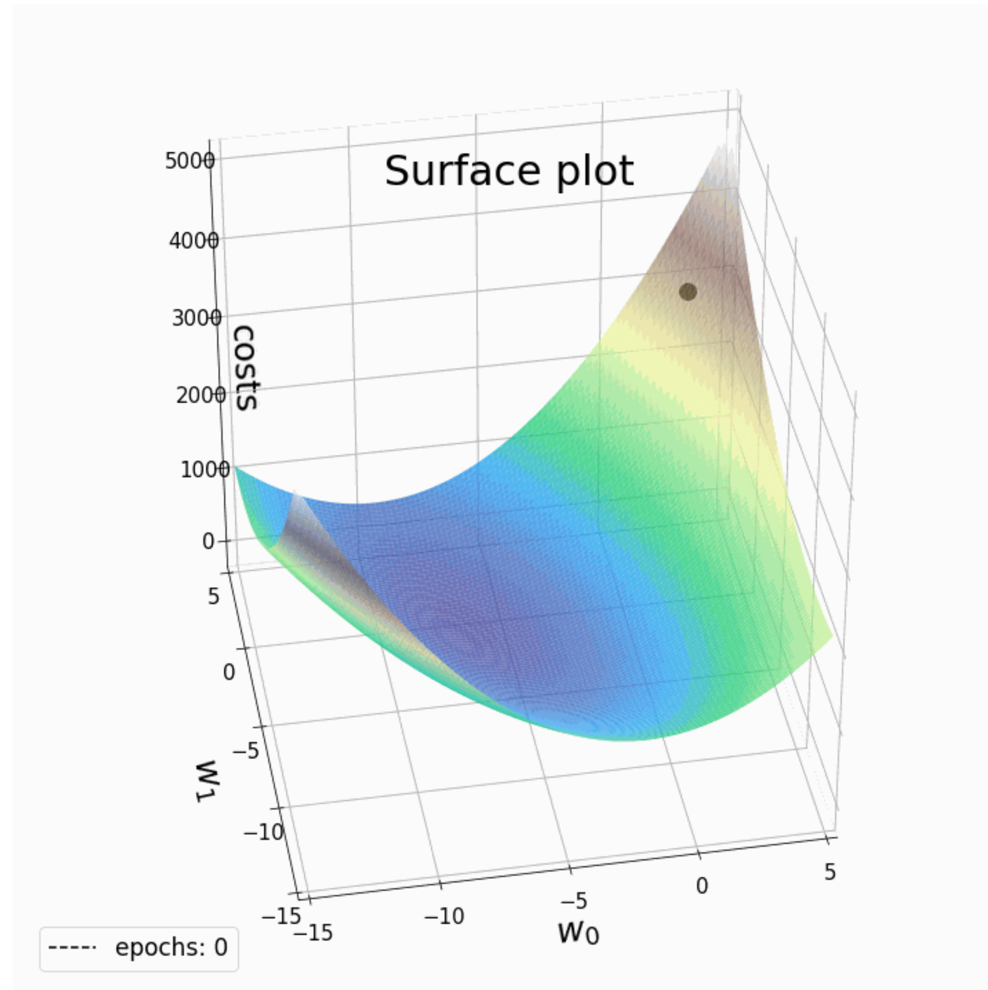
- Convolutional layer
- Pooling layer
- Fully-connected layer



Training a network

- Given a network architecture (CNN, MLP, etc) and some training data, how do we actually set the weights of the network?

Gradient descent: iteratively follow the slope



https://laptrinhx.com/gradient-descent-animation-2-multiple-linear-regression-3070246823/

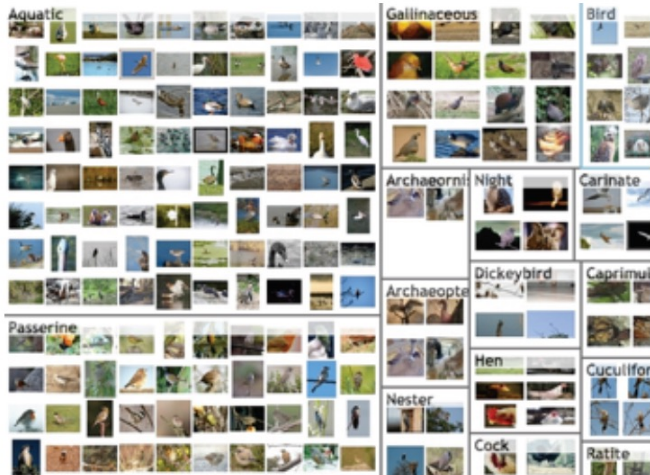
Stochastic gradient descent (SGD)

- Computing the exact gradient over the training set is expensive
- Train on batches of data (e.g., 32 images or 32 rays) at a time
- A full pass through the dataset (i.e., using batches that cover the training data) is called an **epoch**
- Usually need to train for multiple epochs, i.e., multiple full passes through the dataset to converge
- Stochastic gradient descent only approximates the true gradient, but works remarkably well in practice
- Use **backpropagation** to automatically compute gradients on each batch

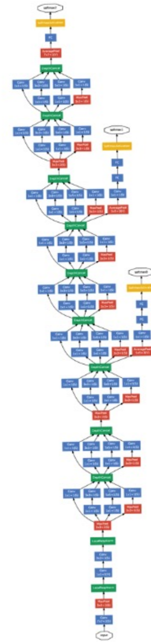
How do you actually train these things?

Roughly speaking:

Gather labeled data



Find a ConvNet architecture



Minimize the loss

But lots of details to get right!

Training a convolutional neural network

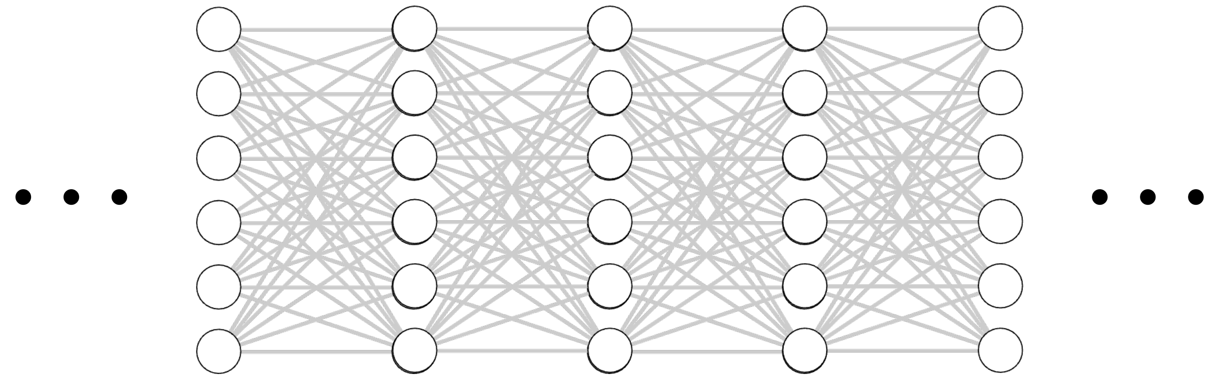
- Split and preprocess your data
- Choose your network architecture
- Initialize your network weights
- Find a learning rate and regularization weight
- Minimize the loss and monitor progress
- Fiddle with knobs…

Why so complicated?

- Training deep networks can be finicky – lots of parameters to learn, complex, non-linear optimization function

What makes training deep networks 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

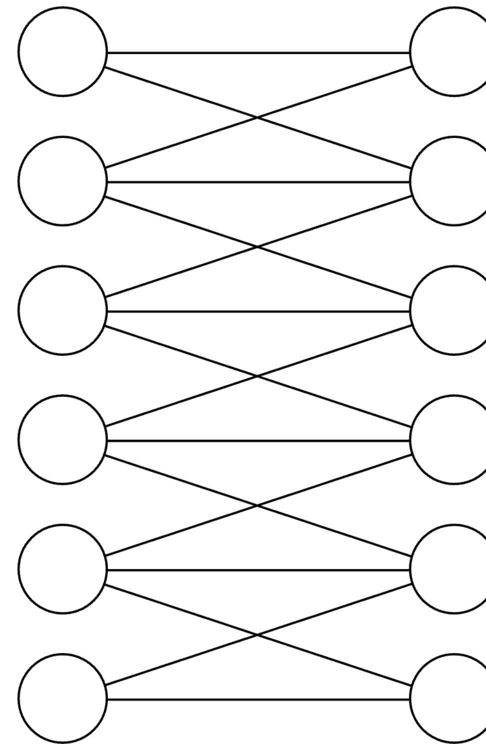


- It's harder to get high *test* accuracy

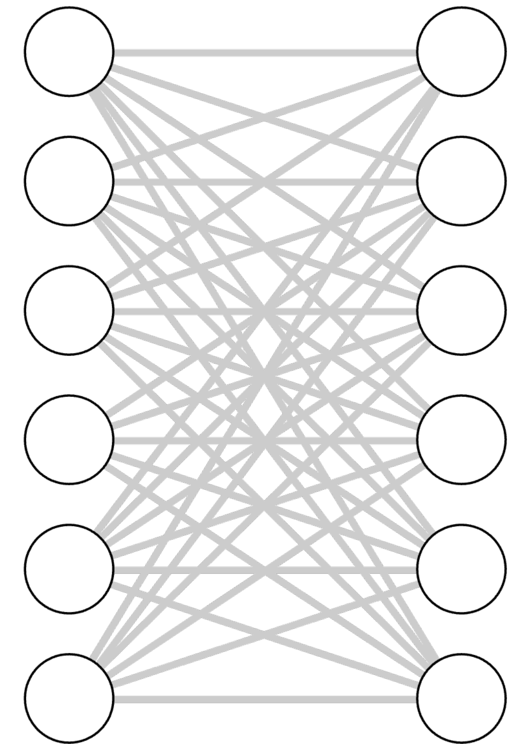
This would be an example of *overfitting*

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?



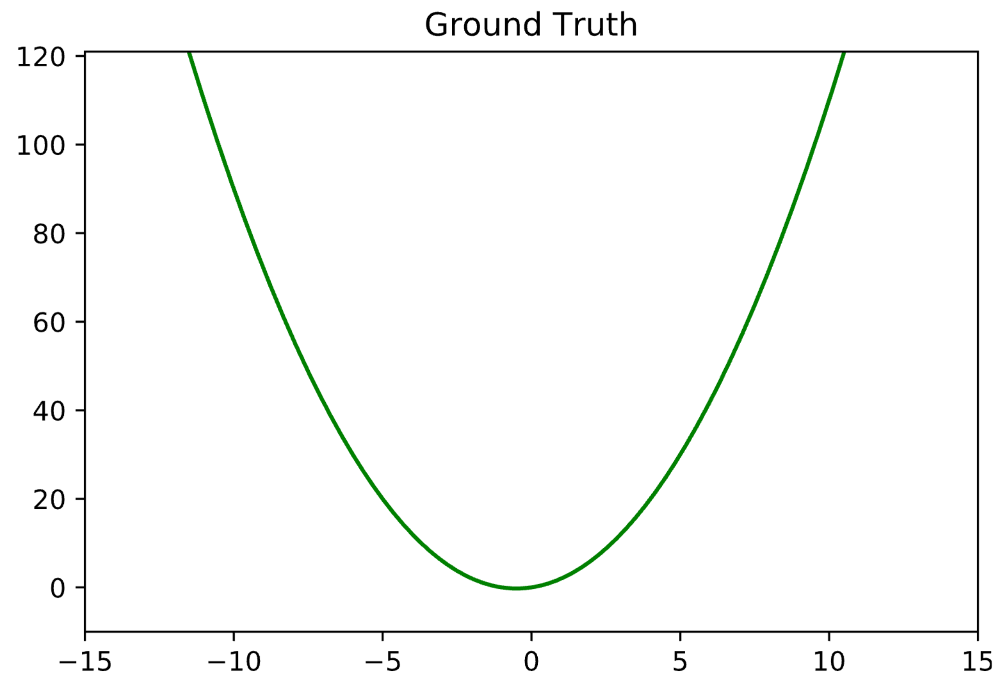
Convolutional Layer

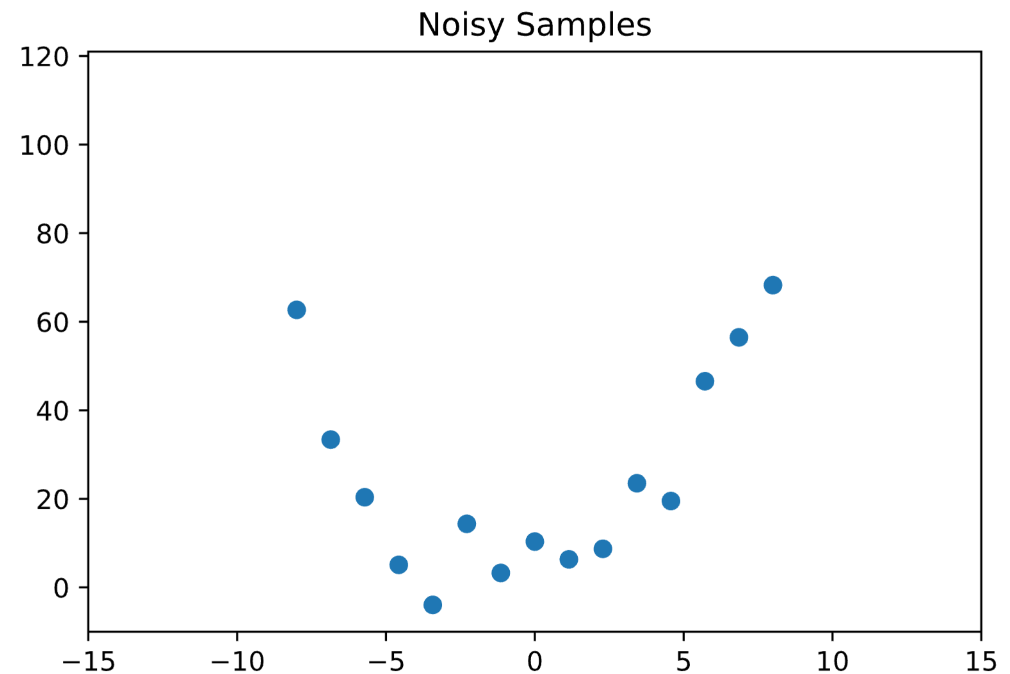


Fully Connected Layer

Overfitting: More Parameters, More Problems

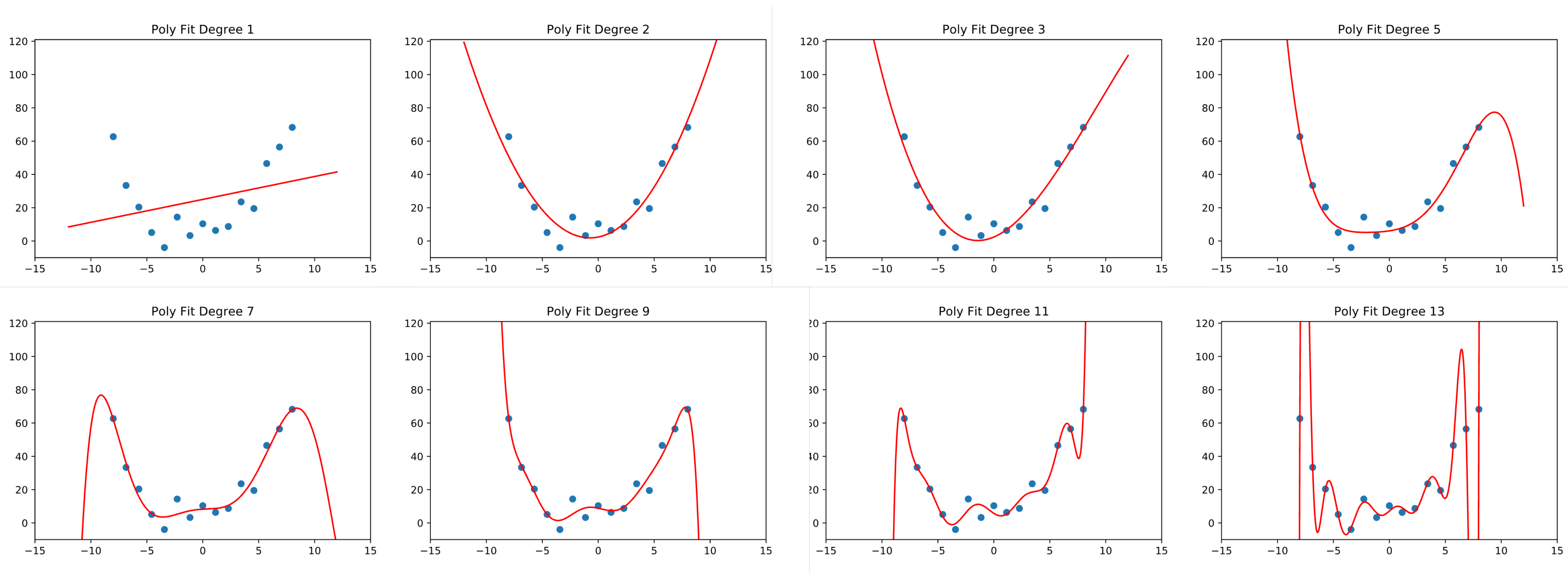
- Non-Deep Example: consider the function $x^2 + x$
- Let's take some noisy samples of the function…





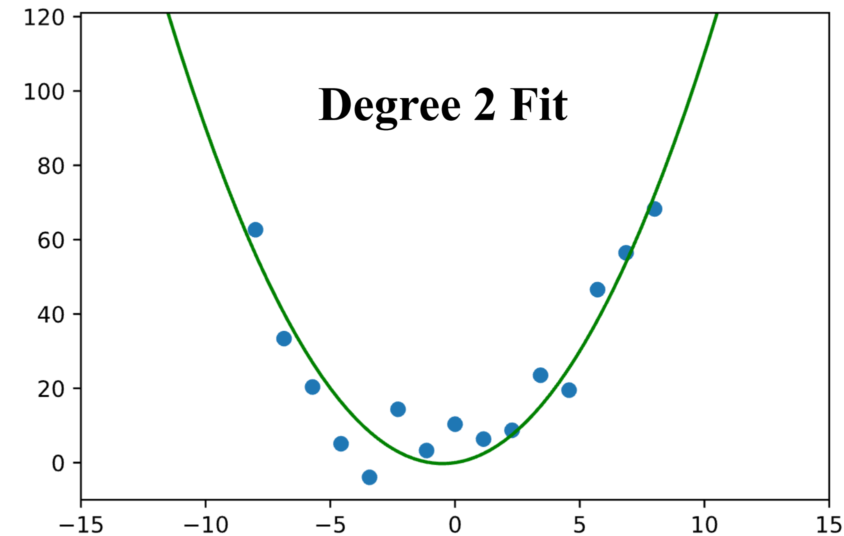
Overfitting: More Parameters, More Problems

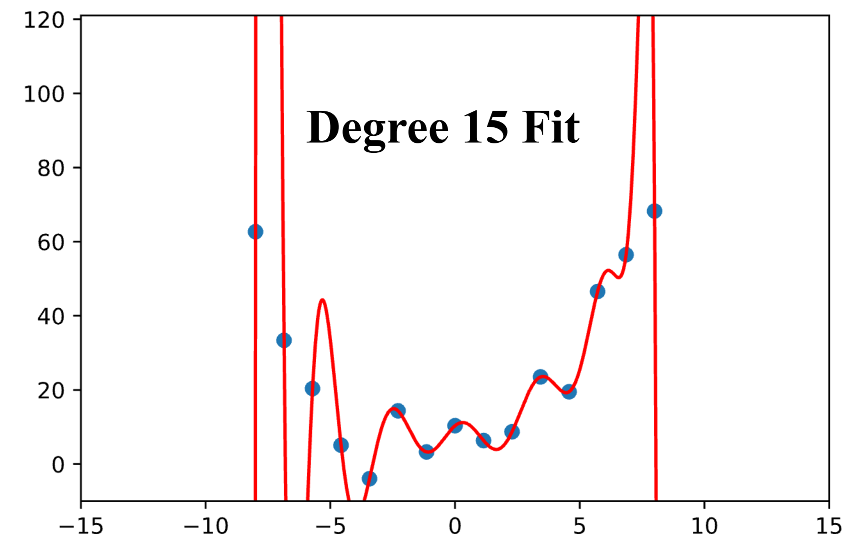
- Now lets fit a polynomial to our samples of the form $P_N(x) = \sum_{k=0}^{N} x^k p_k$



Overfitting: More Parameters, More Problems

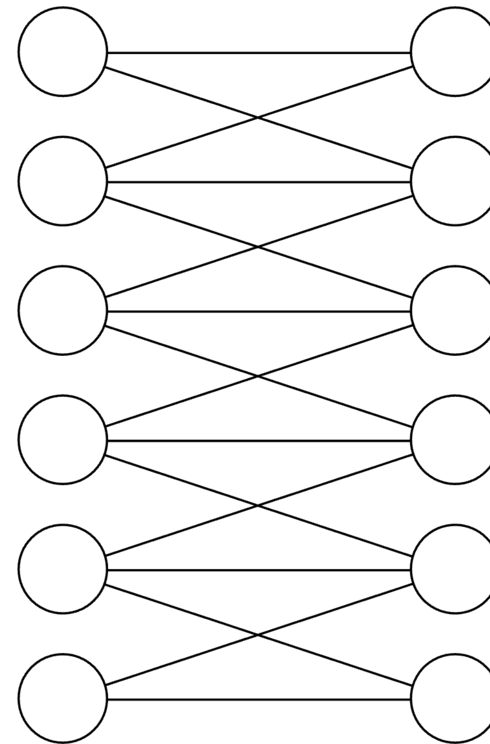
- A model with more parameters can represent more functions
- E.g.,: if $P_N(x) = \sum_{k=0}^{N} x^k p_k$ then $P_2 \in P_{15}$
- More parameters will often **reduce training error** but **increase testing error**. This is *overfitting*.
- When overfitting happens, models do not generalize well



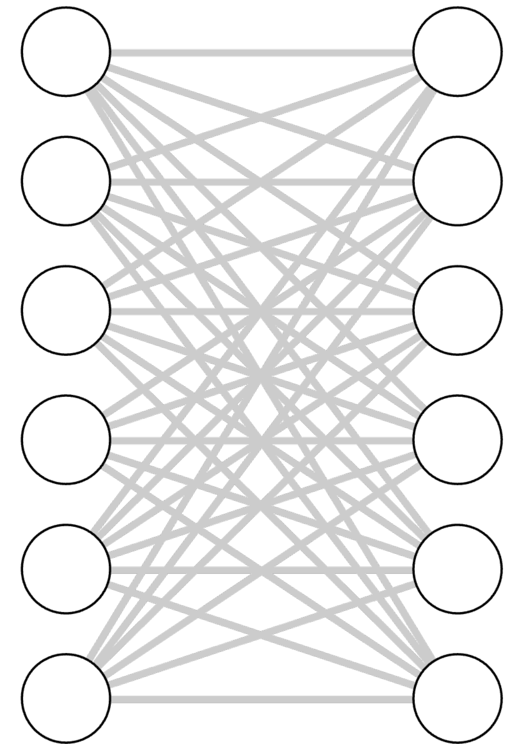


Deep Learning: More Parameters, More Problems?

- 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.



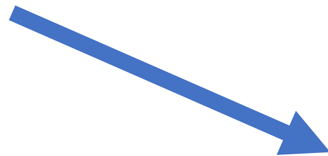
Convolutional Layer

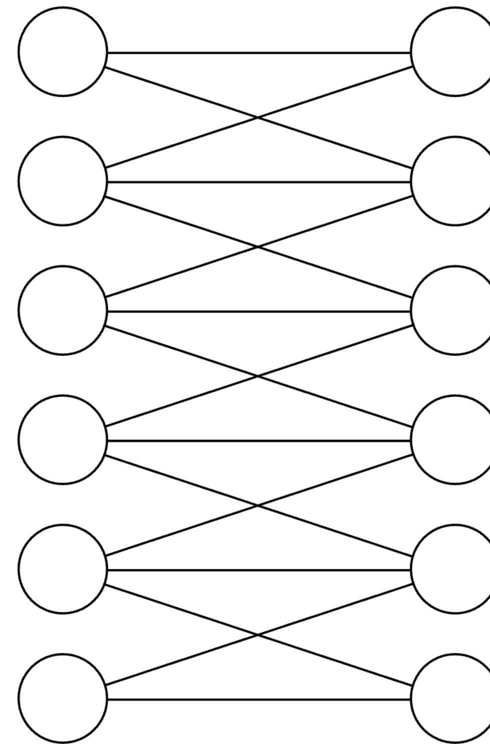


Fully Connected Layer

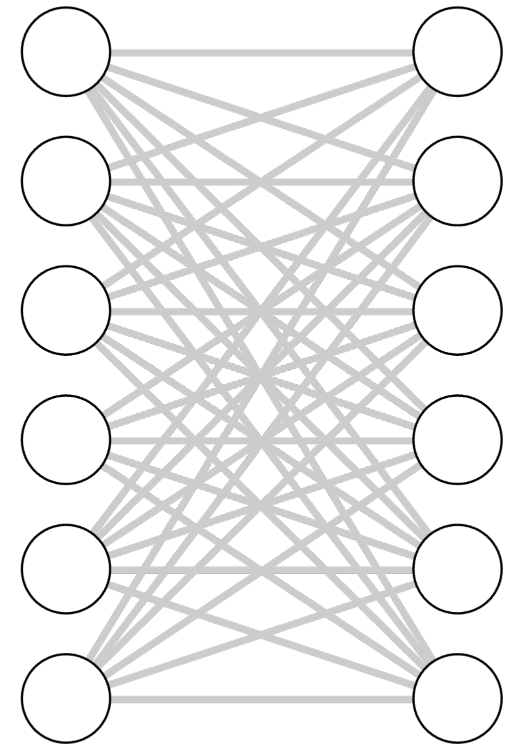
Deep Learning: More Parameters, More Problems?

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





Convolutional Layer



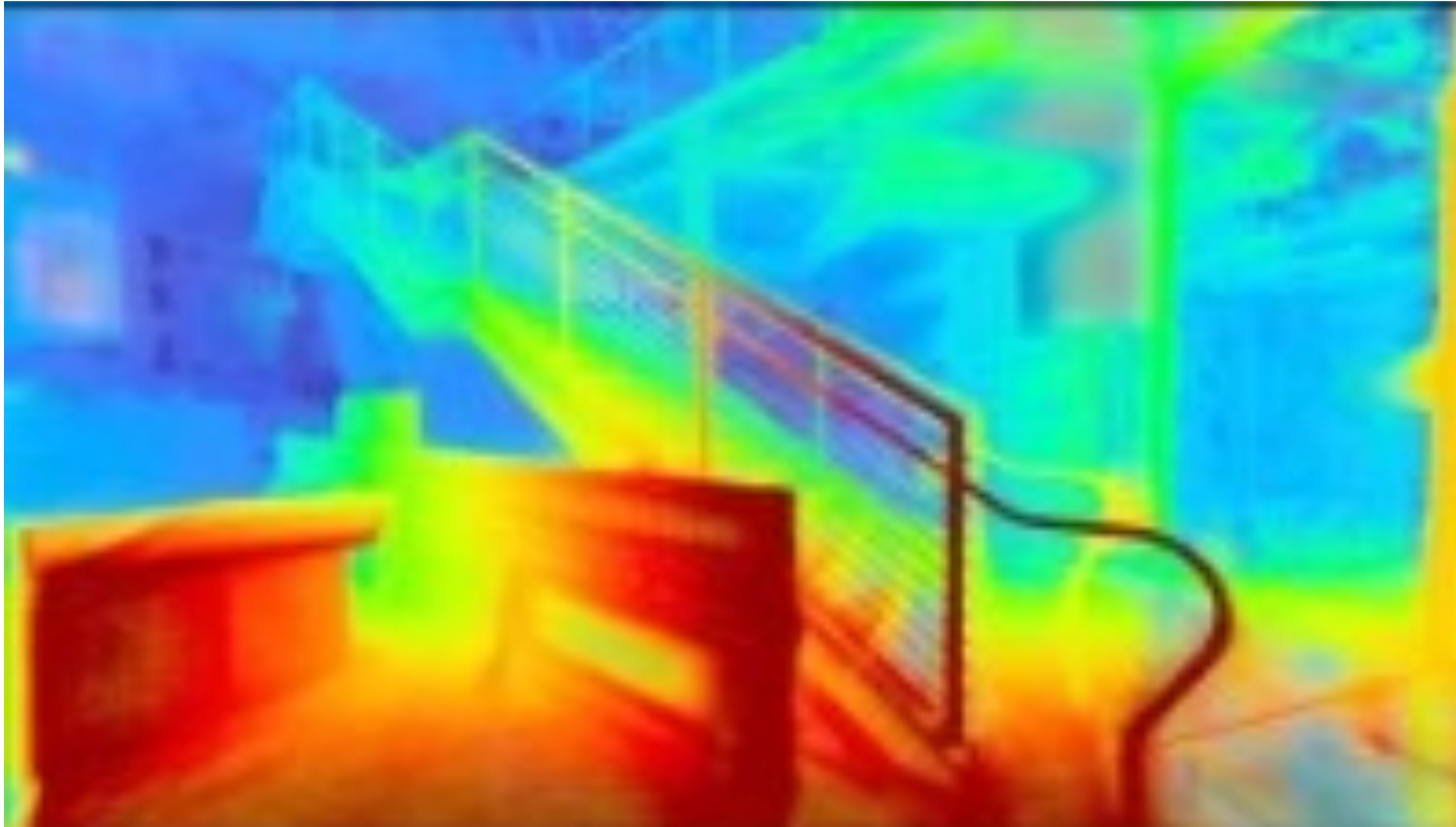
Fully Connected Layer

Overfitting in view synthesis

- What happens if you directly optimize an MPI to reconstruct a small set of input views?

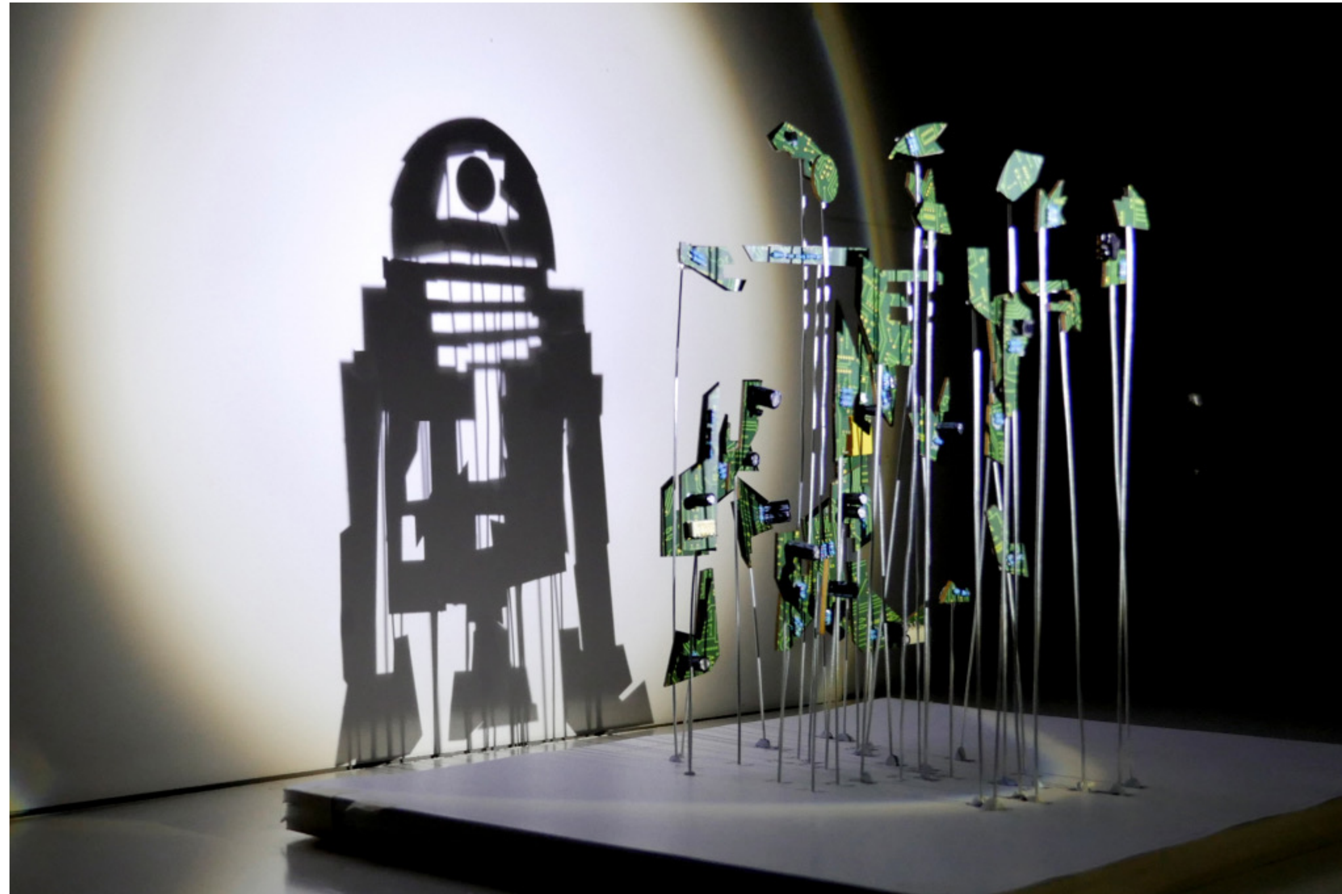
Overfitting in view synthesis

- Answer: you can exactly reconstruct the input views, but produce garbage for new views



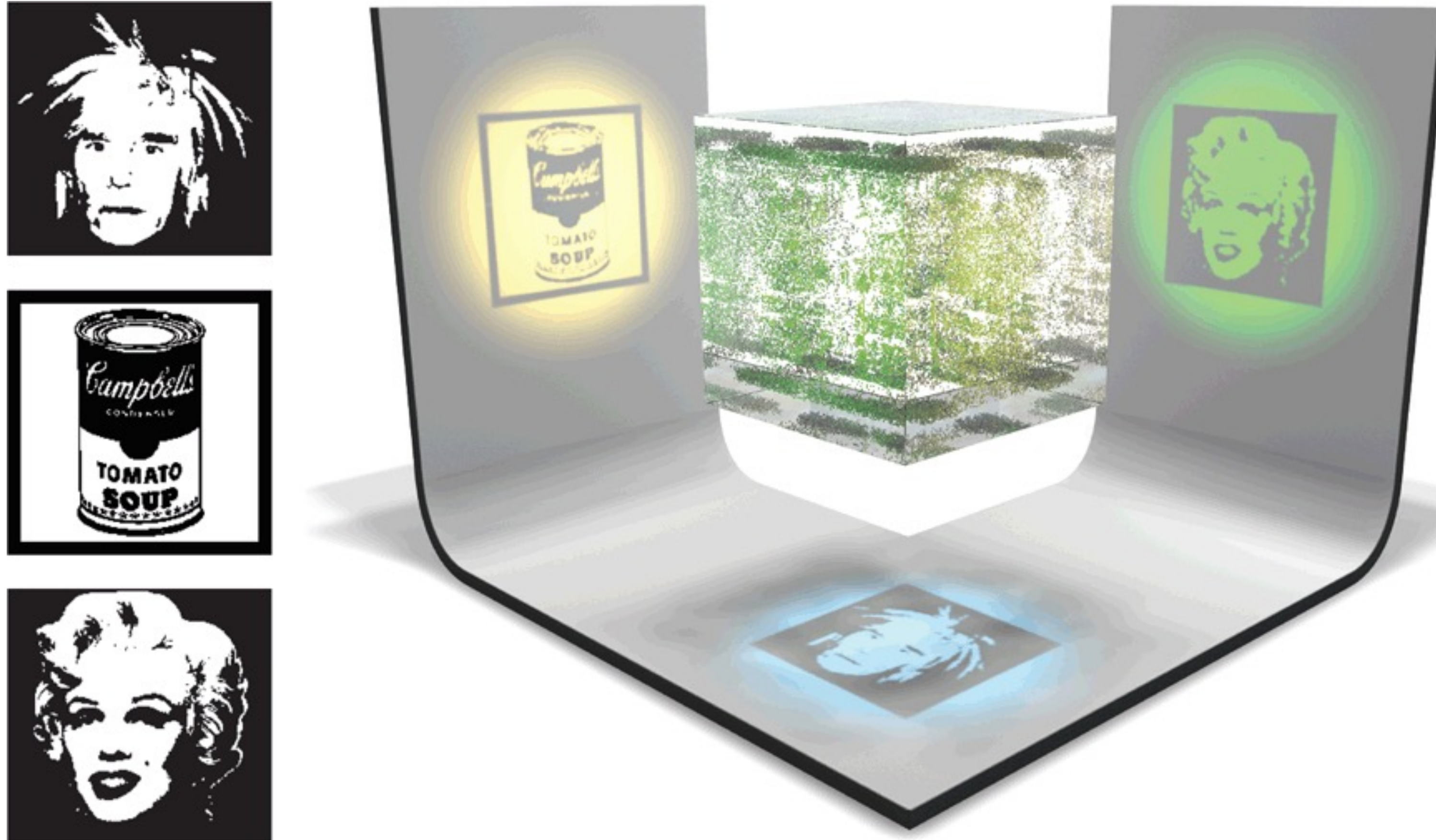
Overfitting in view synthesis

- Reminiscent of *shadow sculptures*



Anamorphic Star Wars Shadow Art by Red Hong Yi, via TKSST

Overfitting in view synthesis



[SHADOW ART]
Niloy J. Mitra, Mark Pauly
ACM SIGGRAPH Asia 2009

Overfitting in view sythesis

- MPI with 64 layers, each storing a 1024 x 768 RGBA image → ~200M parameters
- If we have 32 input RGB images of 1024x768 resolution → ~75M inputs
- **Many more parameters than measurements** → risk of overfitting

- Compare to NeRF: ~500K - 1M parameters

How to Avoid Overfitting: Regularization

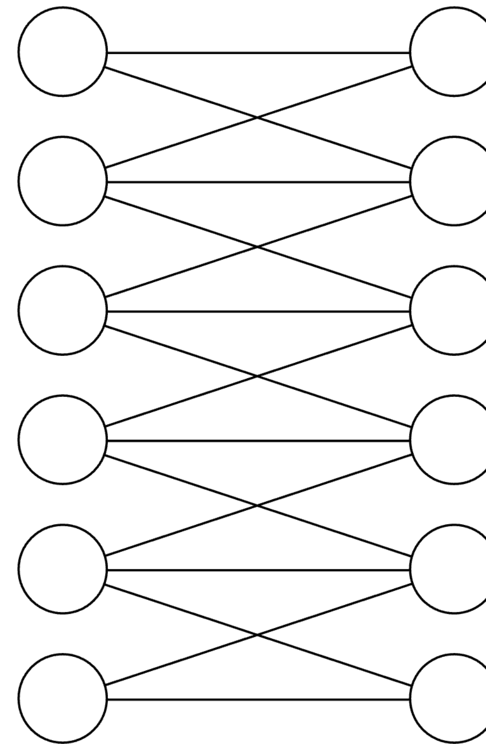
- In general:
  - More parameters means higher risk of overfitting
  - More constraints/conditions on parameters can help

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

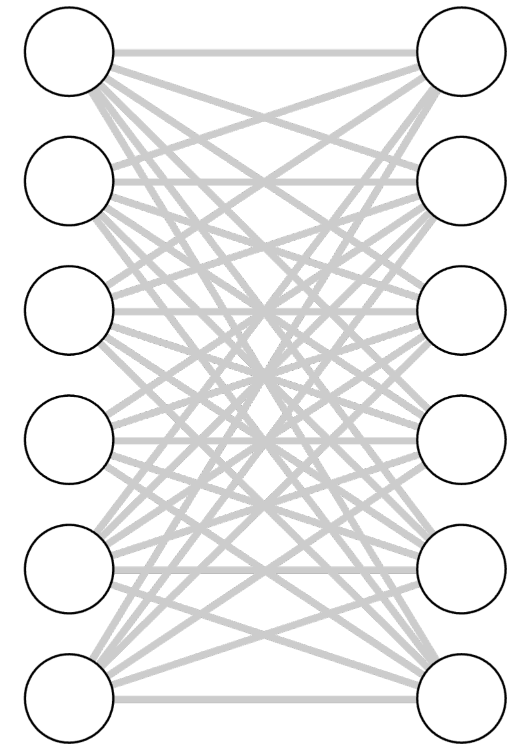
- Regularization can be done through:
  - the design of architecture
  - the choice of loss function
  - the preparation of data
  - …

Regularization: Architecture Choice

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



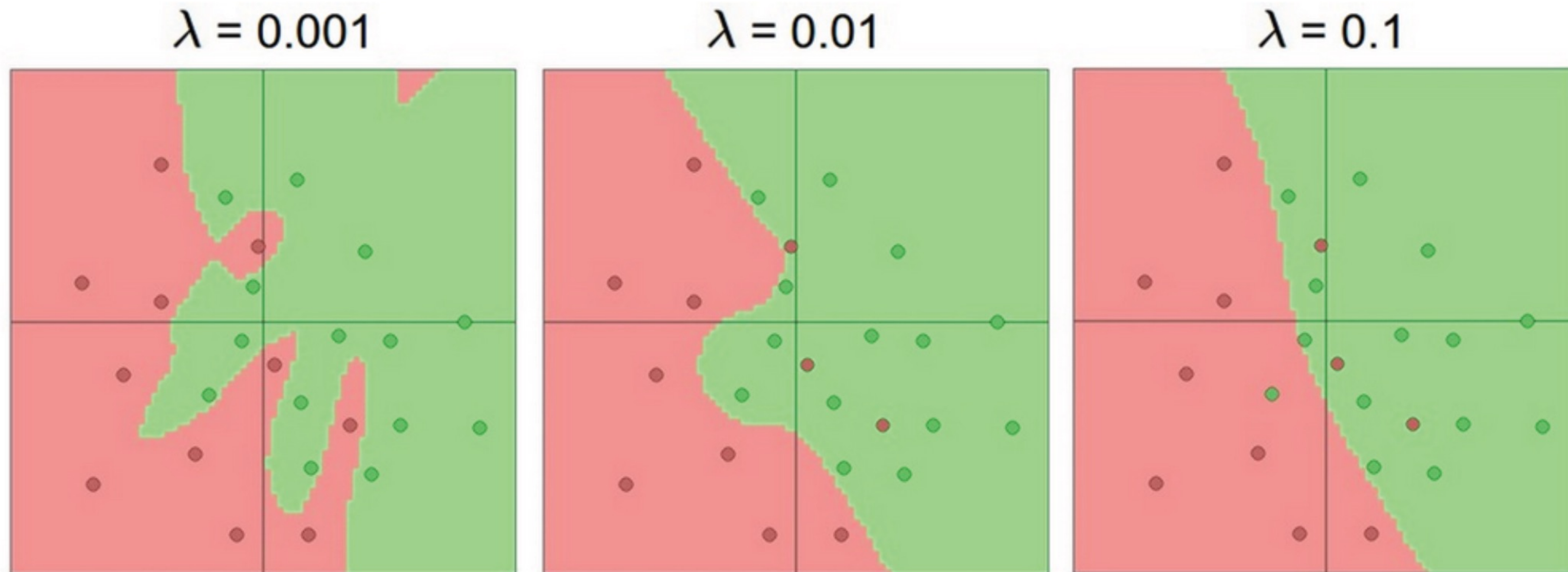
Convolutional Layer



Fully Connected Layer

Regularization reduces overfitting

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



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

(1) Data proprocessing

Preprocess the data so that learning is better conditioned:

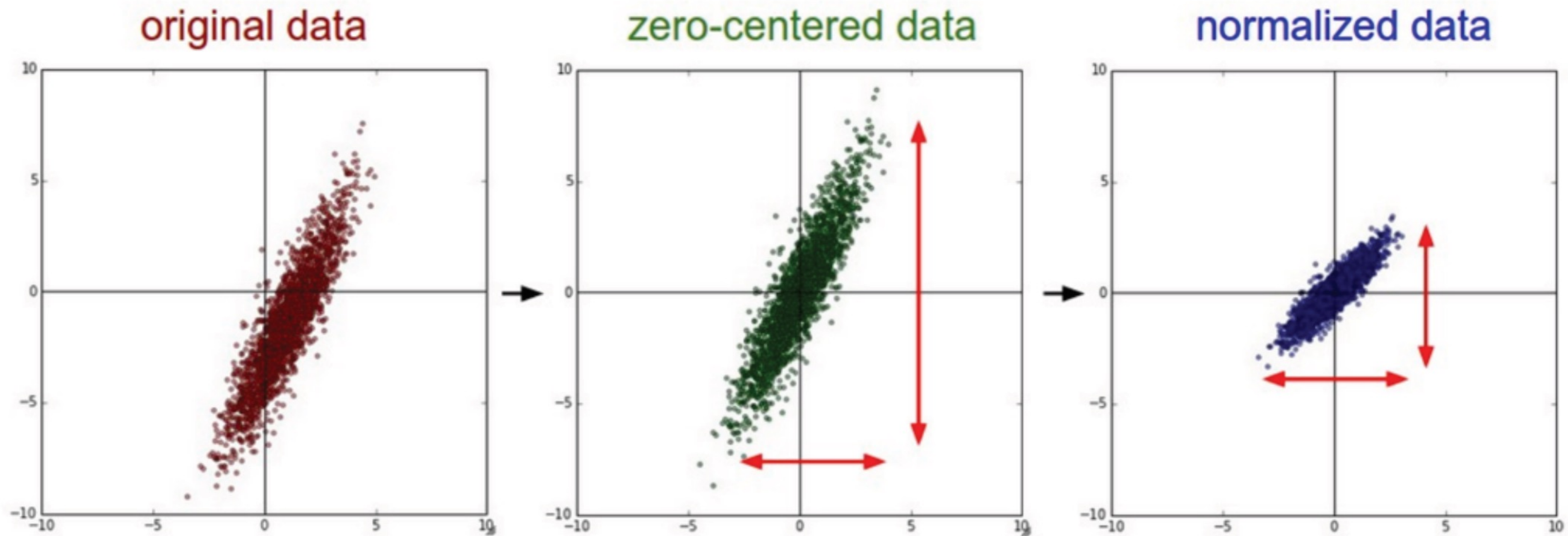


Figure: Andrej Karpathy

(1) Data proprocessing

An input image (256x256)

Minus sign

The mean input image

In practice, often perform a single mean RGB value, and divide by a per-channel standard deviation (recall MOPS, Normalized 8-Point Algorithm)

(1) Data proprocessing

```
225     # Data loading code
226     if args.dummy:
227         print("=> Dummy data is used!")
228         train_dataset = datasets.FakeData(1281167, (3, 224, 224), 1000, transforms.ToTensor())
229         val_dataset = datasets.FakeData(50000, (3, 224, 224), 1000, transforms.ToTensor())
230     else:
231         traindir = os.path.join(args.data, 'train')
232         valdir = os.path.join(args.data, 'val')
233         normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406],
234                                          std=[0.229, 0.224, 0.225])
235
236         train_dataset = datasets.ImageFolder(
237             traindir,
238             transforms.Compose([
239                 transforms.RandomResizedCrop(224),
```

Batch normalization

- Side note – can also perform normalization after each layer of the network to stabilize network training ("*batch normalization*")

(1) Data preprocessing

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

(2) Choose your architecture

A Neural Network Playgr

https://playground.tensorflow.org/#activation=tanh&batchSize=10&dataset=circle®Dataset=reg-plane&learningRate=0....

Epoch 000,000 | Learning rate 0.03 | Activation Tanh | Regularization None | Regularization rate 0 | Problem type Classification

DATA

Which dataset do you want to use?

Ratio of training to test data: 50%

Noise: 0

Batch size: 10

REGENERATE

FEATURES

Which properties do you want to feed in?

X_1, X_2, X_1^2, X_2^2, X_1X_2, $\sin(X_1)$, $\sin(X_2)$

2 HIDDEN LAYERS

4 neurons

2 neurons

This is the output from one **neuron**. Hover to see it larger.

The outputs are mixed with varying **weights**, shown by the thickness of the lines.

OUTPUT

Test loss 0.507

Training loss 0.504

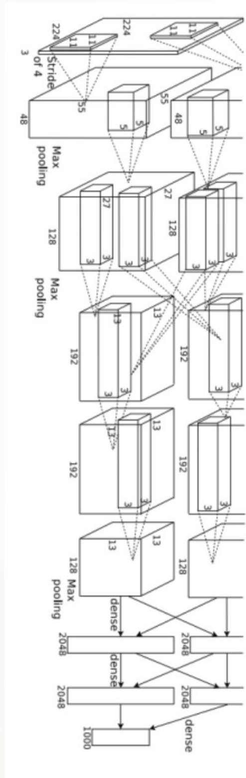
Colors shows data, neuron and weight values.

https://playground.tensorflow.org/

(2) Choose your architecture

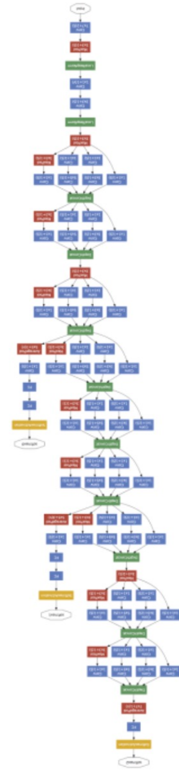
Very common modern choice for classification problems

"AlexNet"



[Krizhevsky et al. NIPS 2012]

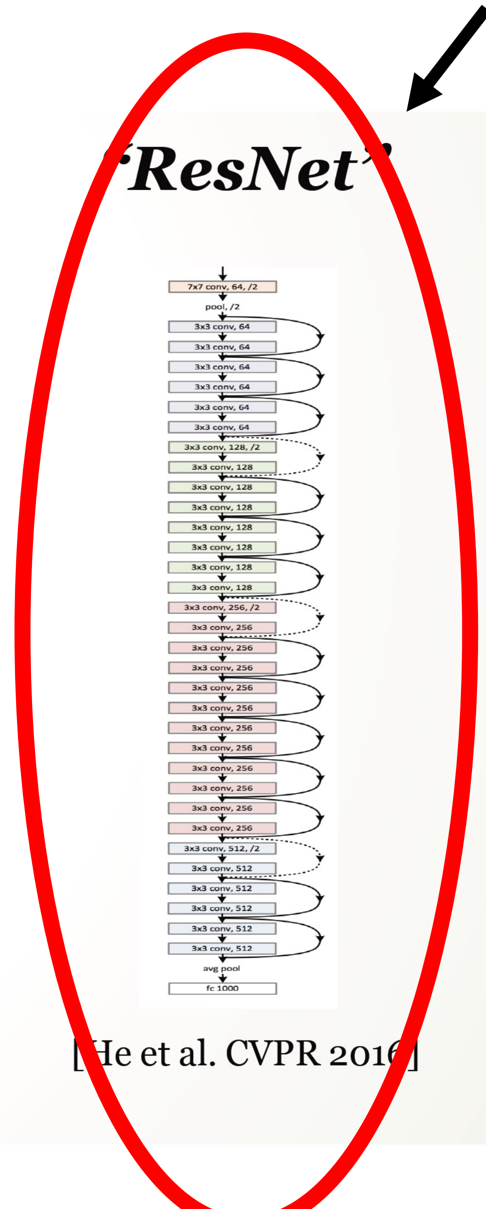
"GoogLeNet"



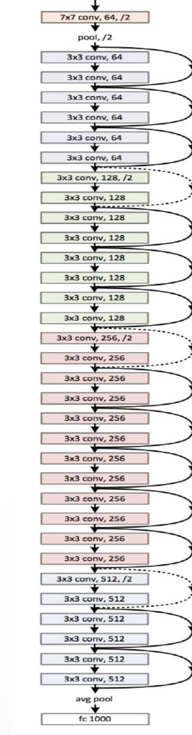
[Szegedy et al. CVPR 2015]

"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

(4) Overfit 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’

(4) Overfit 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

(4) Overfit a small portion of the data

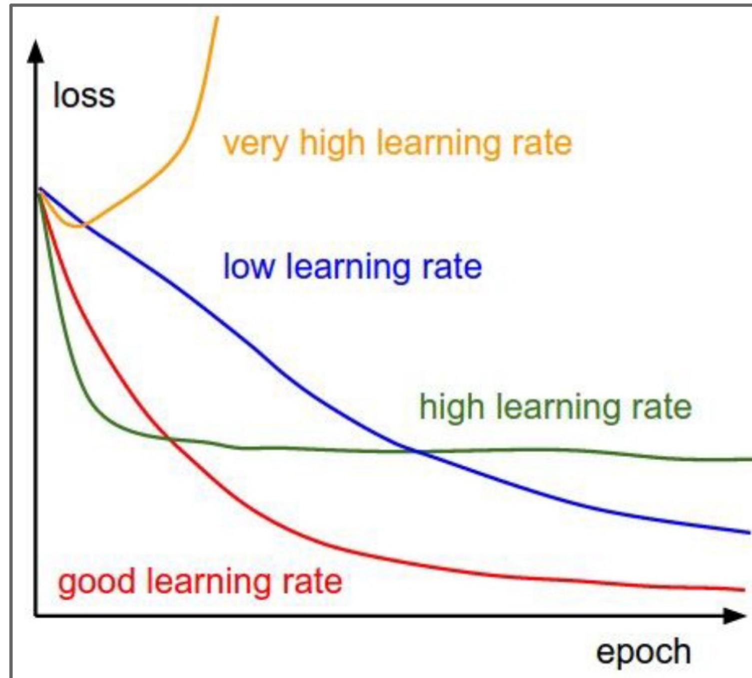
100% accuracy on the training set (good)

```
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



Q: Which one of these learning rates is best to use?

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 with

- 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?

Transfer learning

"You need a lot of data if you want to train/use CNNs for a new classification task"

Transfer learning

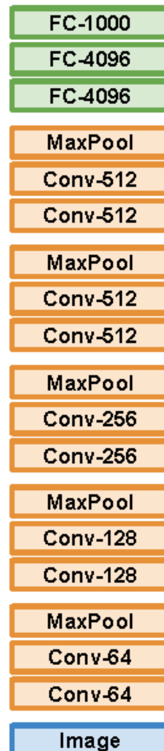
"You need a lot of data if you want to train/use CNNs for a new classification task"

BUSTED

Slide credit: Fei-Fei Li, Justin Johnson, and Serena Yeun

Transfer learning with CNNs

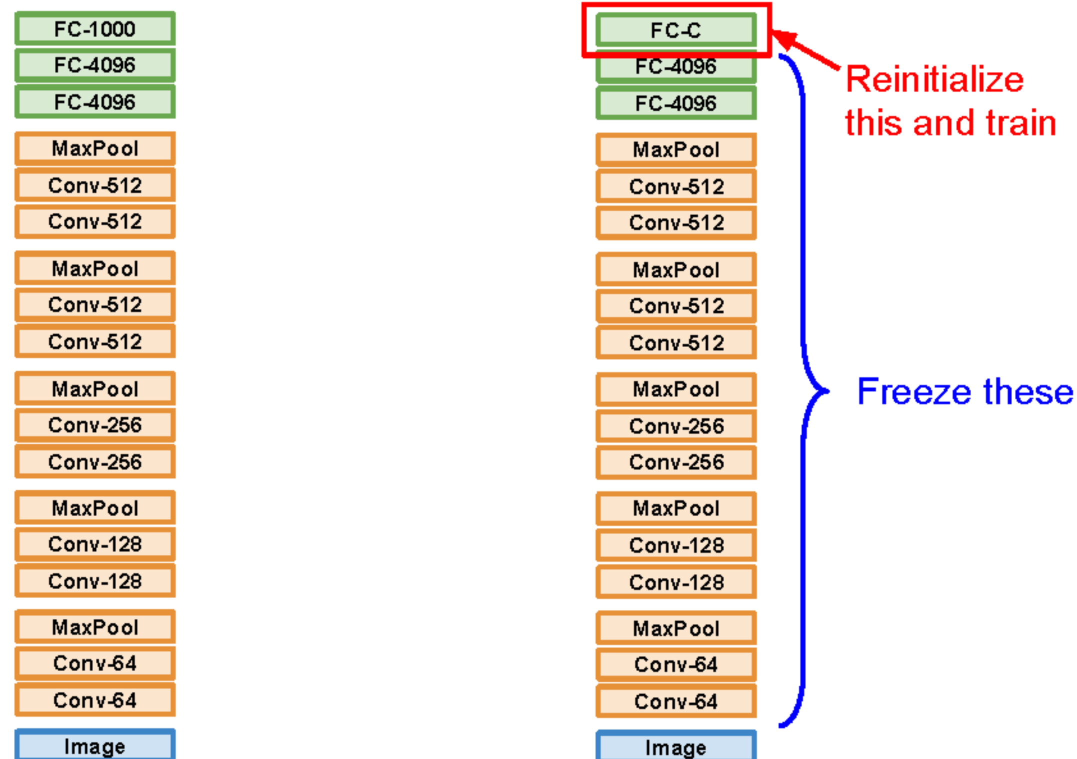
Step 1: Take a model trained on ImageNet



Slide credit: Fei-Fei Li, Justin Johnson, and Serena Yeur

Transfer learning with CNNs

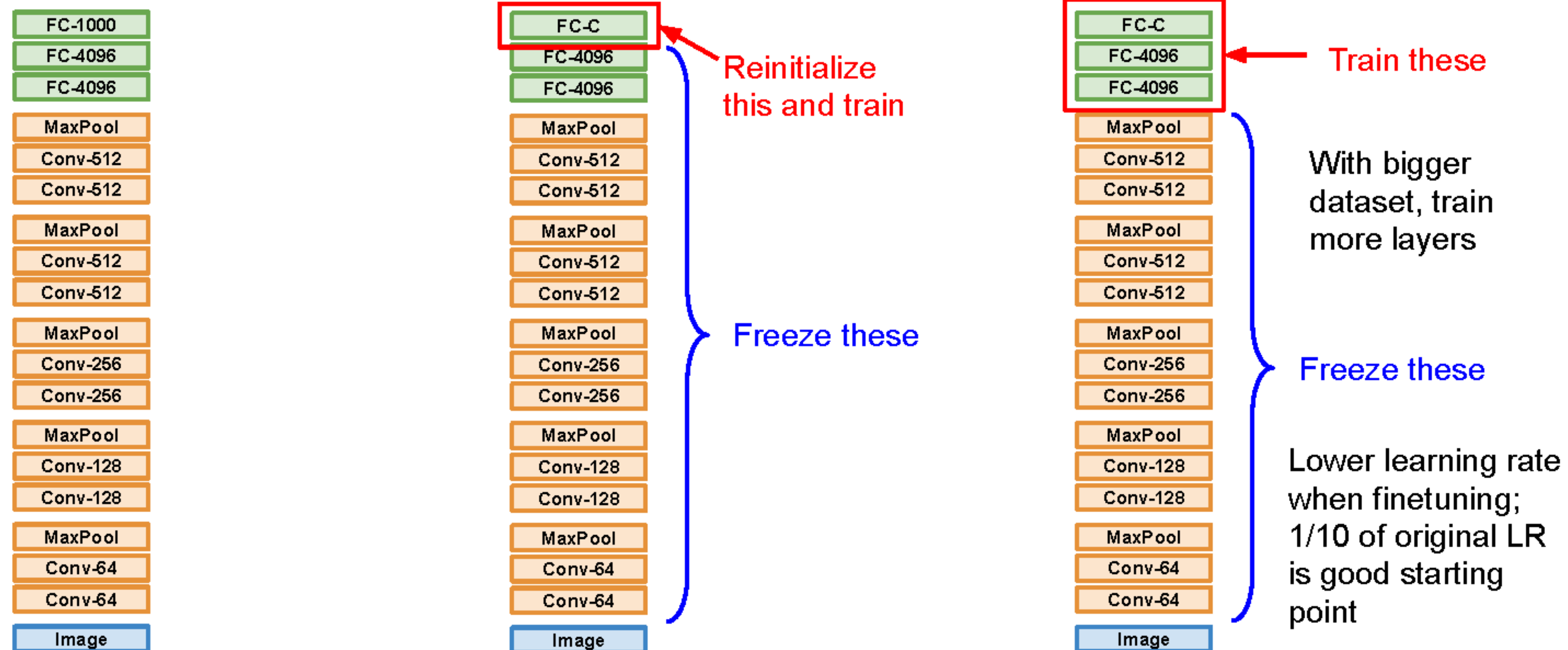
Step 2a: If you have a small amount of new data, adjust a small number of network weights



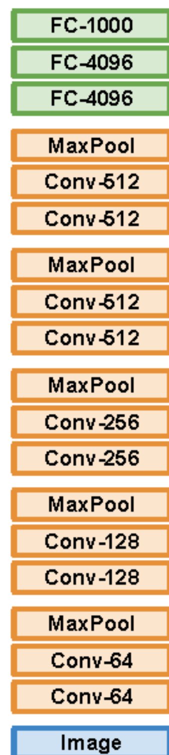
Slide credit: Fei-Fei Li, Justin Johnson, and Serena Yeur

Transfer learning with CNNs

Step 2b: If you have a larger amount of new data, adjust a larger number of network weights



Slide credit: Fei-Fei Li, Justin Johnson, and Serena Yeur

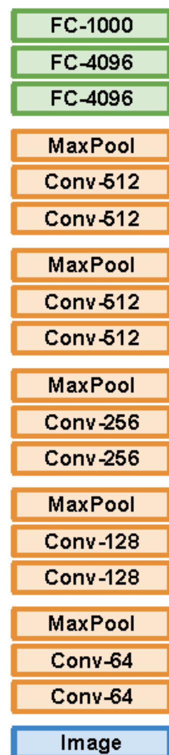


More specific

More generic

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

Slide credit: Fei-Fei Li, Justin Johnson, and Serena Yeun

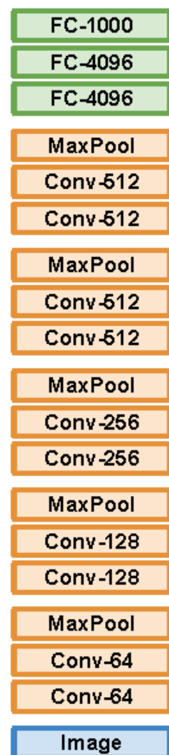


More specific

More generic

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

Slide credit: Fei-Fei Li, Justin Johnson, and Serena Yeun



More specific

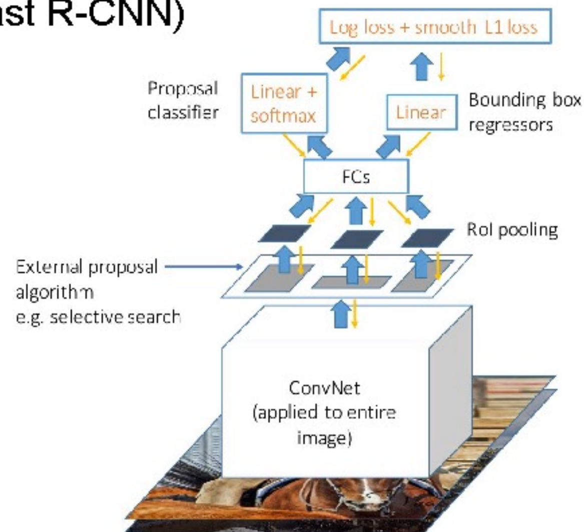
More generic

	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

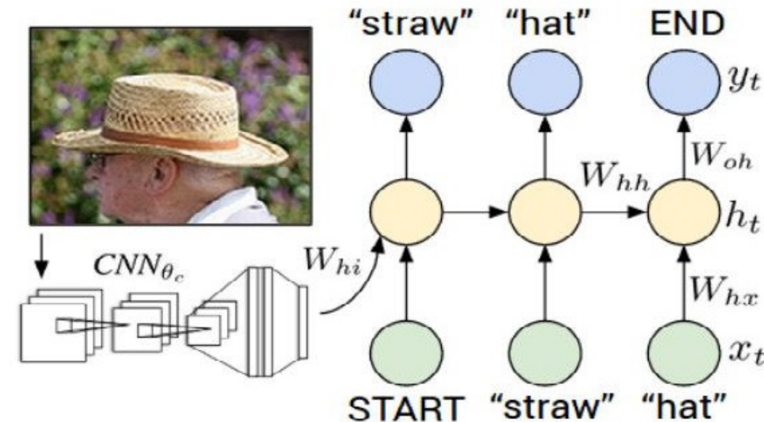
Slide credit: Fei-Fei Li, Justin Johnson, and Serena Yeu

Transfer learning with CNNs is pervasive

- It's the norm, not the exception



Girshick, "Fast R-CNN", ICCV 2015
Figure copyright Ross Girshick, 2015. Reproduced with permission.

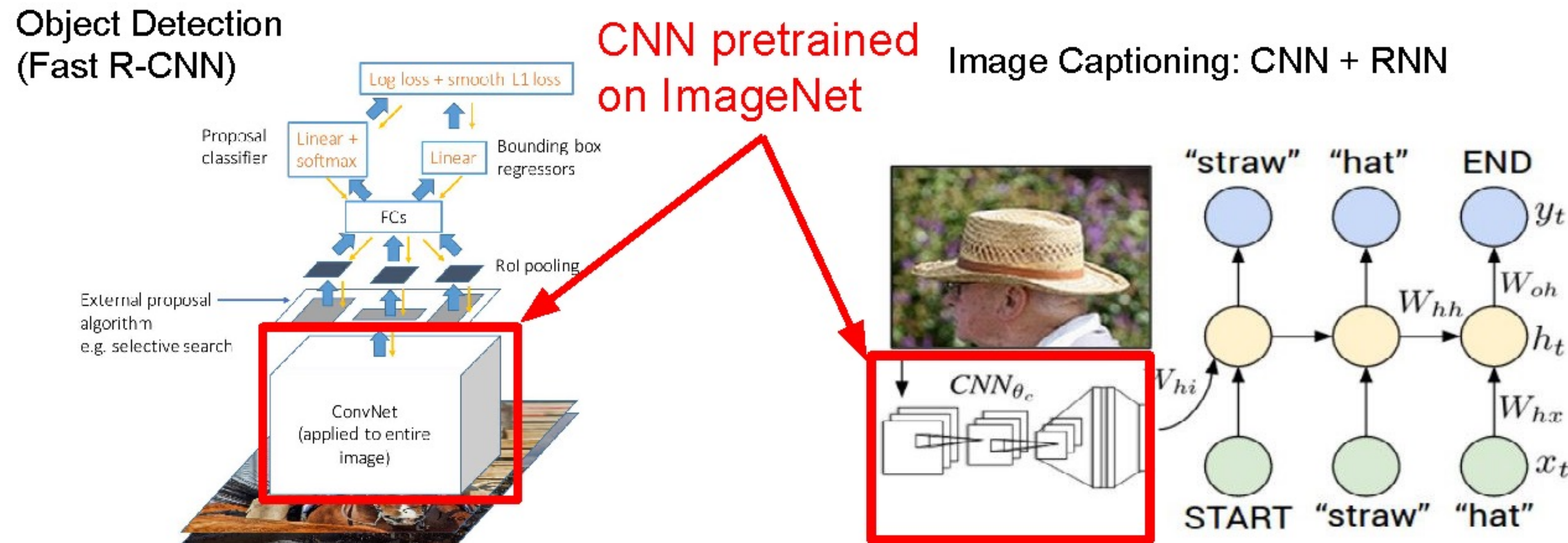


Karpathy and Fei-Fei, "Deep Visual-Semantic Alignments for Generating Image Descriptions", CVPR 2015
Figure copyright IEEE, 2015. Reproduced for educational purposes.

Slide credit: Fei-Fei Li, Justin Johnson, and Serena Yeung

Transfer learning with CNNs is pervasive

- It's the norm, not the exception



Slide credit: Fei-Fei Li, Justin Johnson, and Serena Yeun

Other pre-trained models are starting to become standard

- Swin-transformer pre-trained on ImageNet-21K
- DINO features
- Foundation models (Stable Diffusion, etc)

Takeaway for your projects and beyond

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

1. Find a large dataset with similar data (e.g., ImageNet), train a large CNN
2. Apply transfer learning to fine-tune on your data

For step 1, many existing models exist in "Model Zoos"

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

Slide credit: Fei-Fei Li, Justin Johnson, and Serena Yeur

Questions?