What it takes to train ConvNets

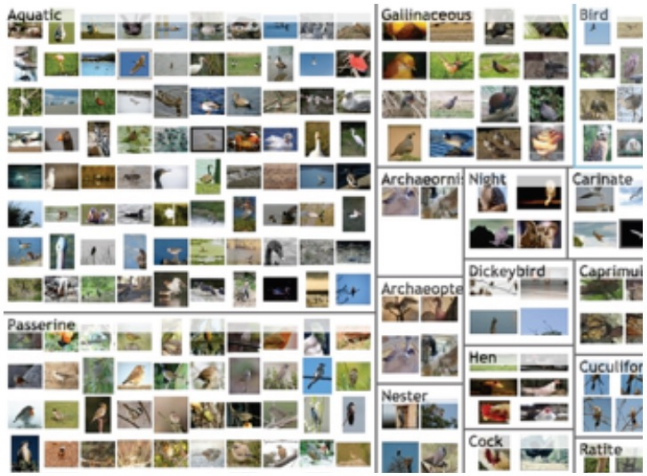
Admin updates

- Starting today, have everything for hw1
- See handout in Canvas' "files"
- Tests still failing, but have tried your best?
  - Mention this at the top of README.md
  - TA (may) give partial grades accordingly

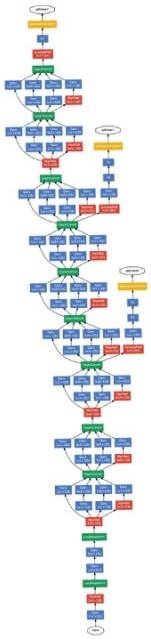
How do you actually train these things?

Roughly speaking:

Gather labeled data



Find a ConvNet architecture

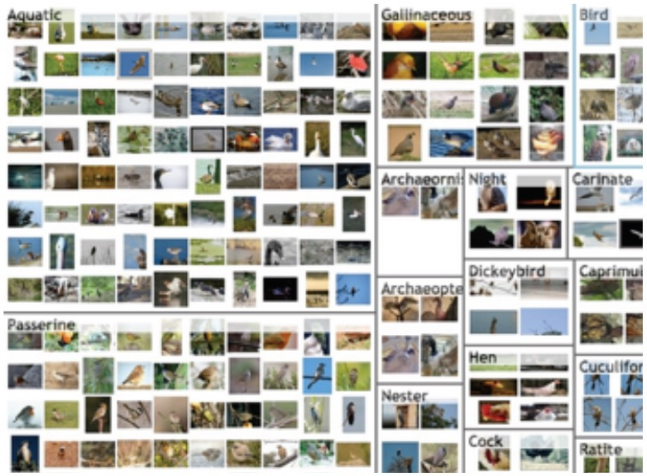


Minimize the loss

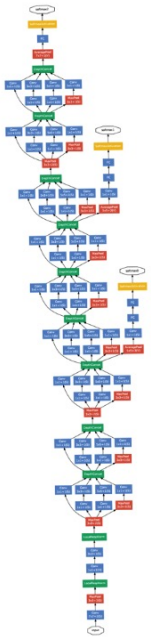
How do you actually train these things?

Roughly speaking:

Gather labeled data



Find a ConvNet architecture



Minimize the loss

Microsoft COCO dataset

https://arxiv.org/abs/1405.0312

Microsoft COCO
Common Objects in Context

Tsung-Yi Lin
Cornell Tech

Michael Maire
TTI Chicago

Serge Belongie
Cornell Tech

Lubomir Bourdev
Facebook

James Hays
Brown University

Pietro Perona
Caltech

Deva Ramanan
UC Irvine

Ross Girshick
Microsoft Research

Piotr Dollar
Microsoft Research

Larry Zitnick
Microsoft Research

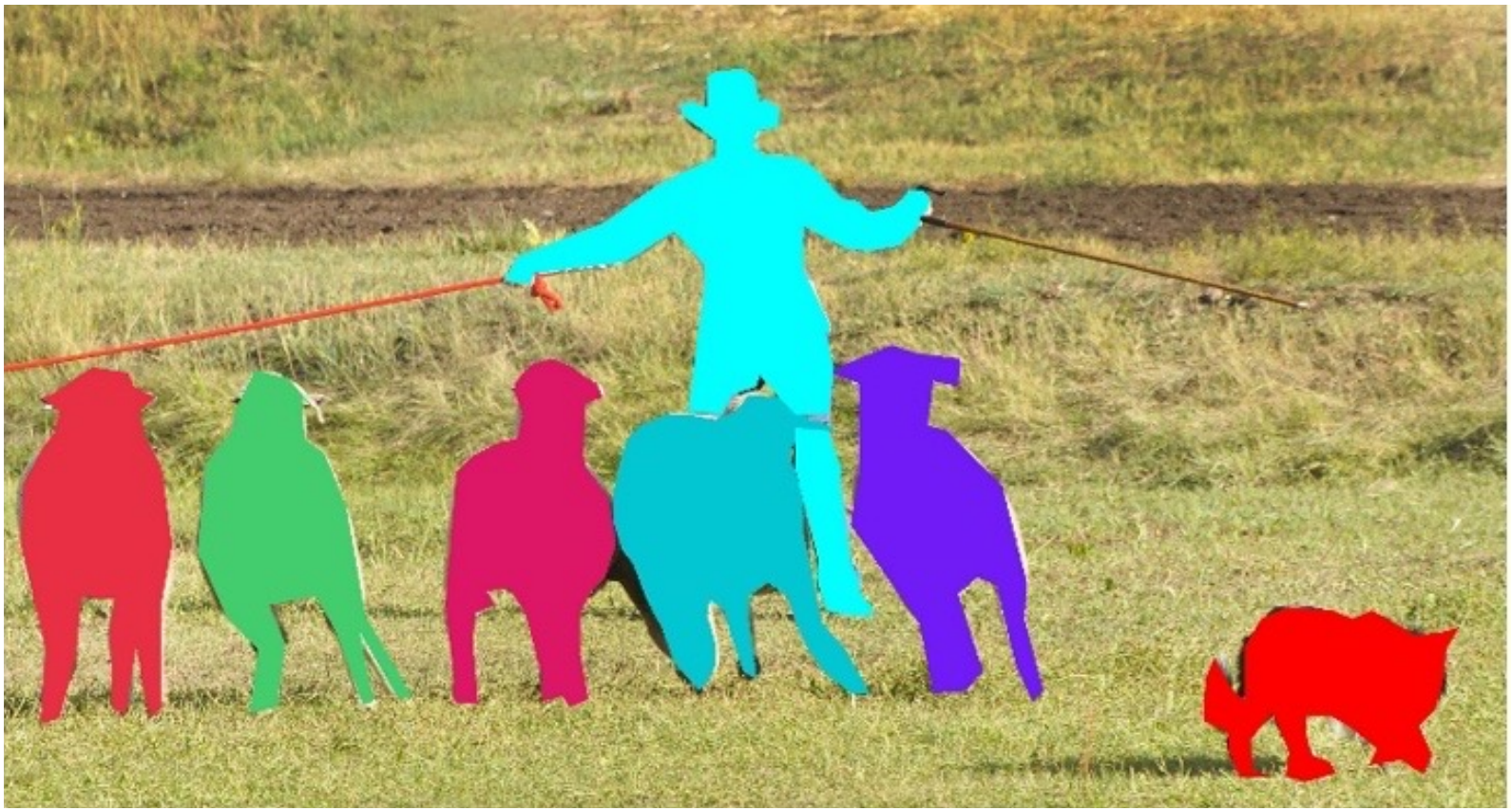
https://cocodataset.org

https://cocodataset.org

https://cocodataset.org

✓ Instance segmentation

✓ Non-iconic Images



https://cocodataset.org

Iconic object images

https://cocodataset.org

Iconic scene images




https://cocodataset.org

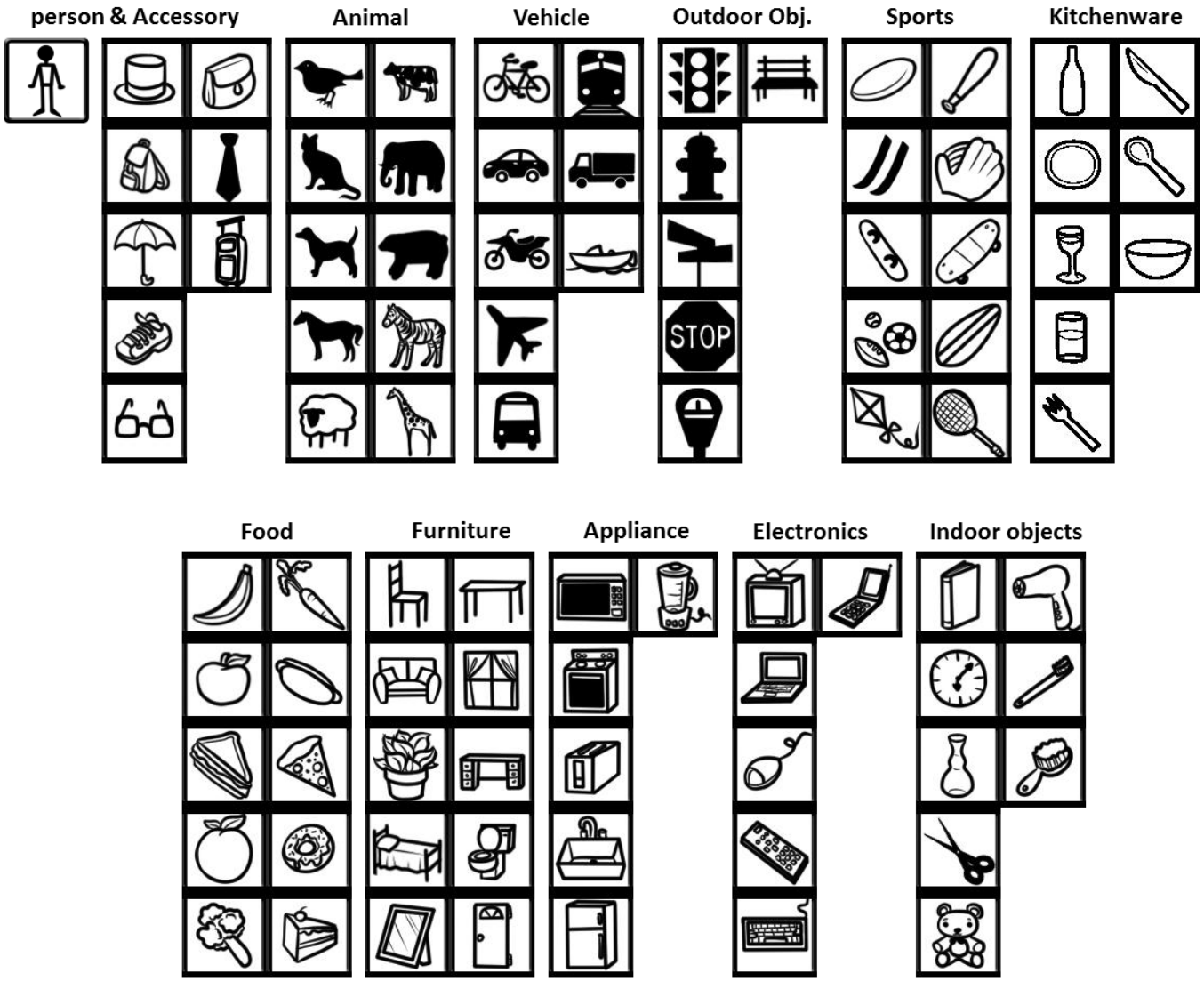
Non-iconic images

https://cocodataset.org

https://cocodataset.org

https://cocodataset.org

Object categories



https://cocodataset.org

flickr

(All creative commons)

330,000 images

https://cocodataset.org

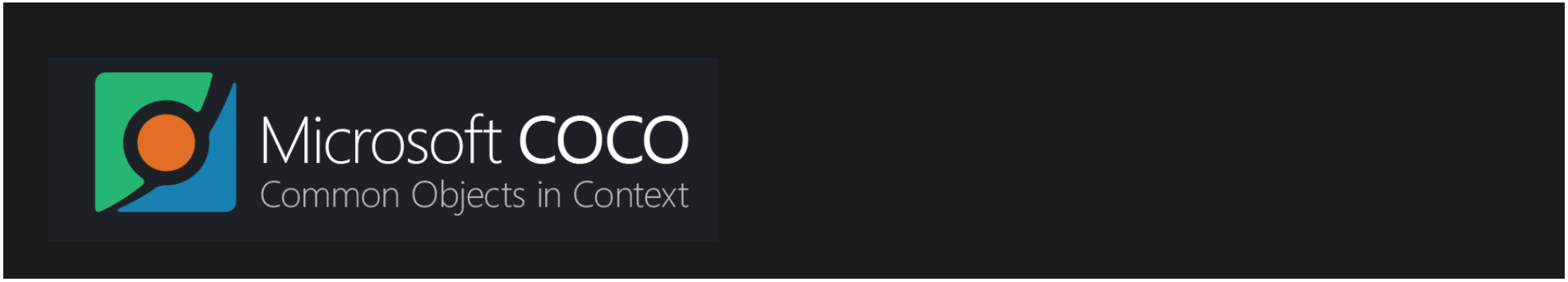
Search for “Dog” (…iconic)



https://cocodataset.org

Search for “Dog + Car” (context!)

https://cocodataset.org



Annotation pipeline

https://cocodataset.org

https://cocodataset.org

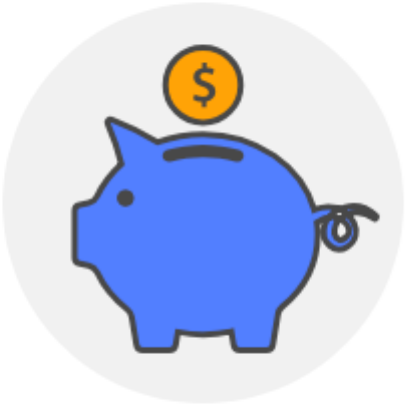
https://cocodataset.org

Amazon Mechanical Turk

amazon mechanical turk beta

Get Started with Amazon Mechanical Turk

or



Create Tasks

Human intelligence through an API. Access a global, on-demand, 24/7 workforce.

Create a Requester account

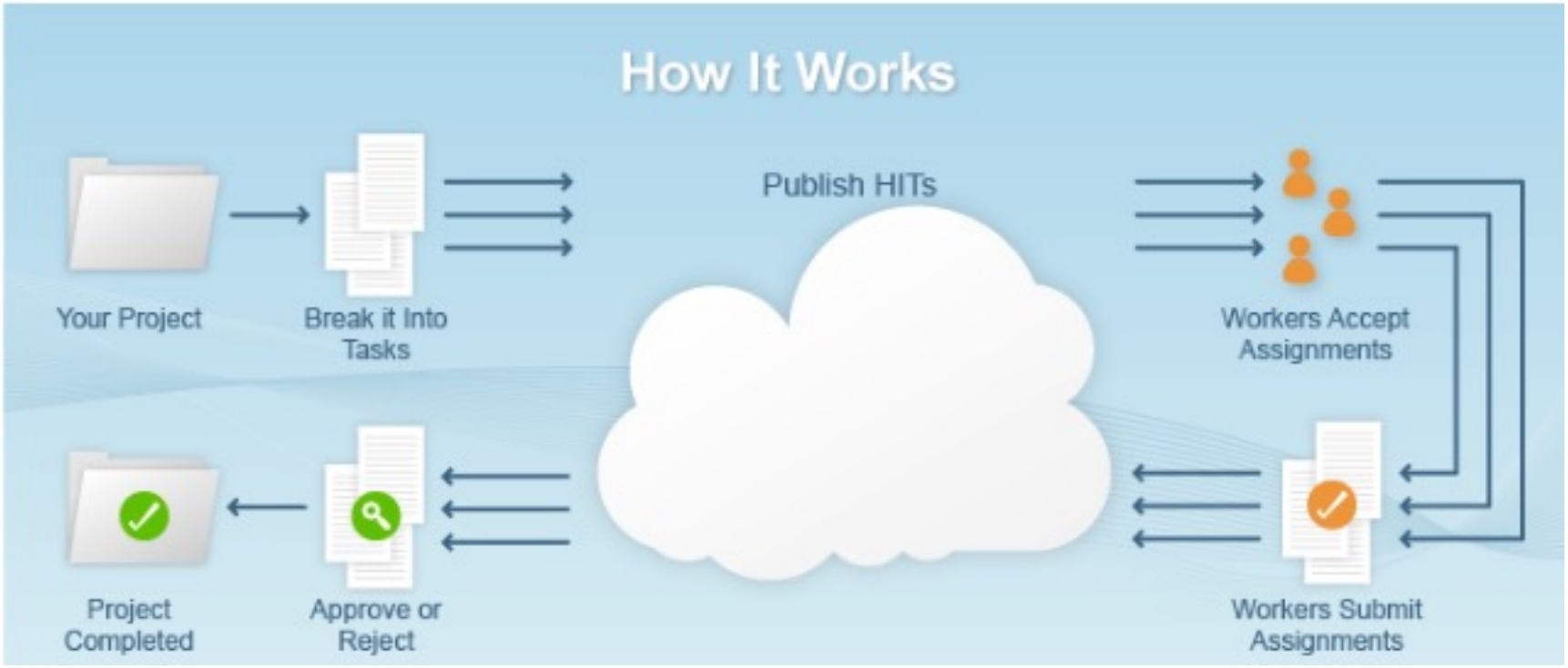
Make Money

Make money in your spare time. Get paid for completing simple tasks.

Create a Worker account

ataset.org

Amazon Mechanical Turk



https://cocodataset.org

Divide and Conquer

1. Category Labeling

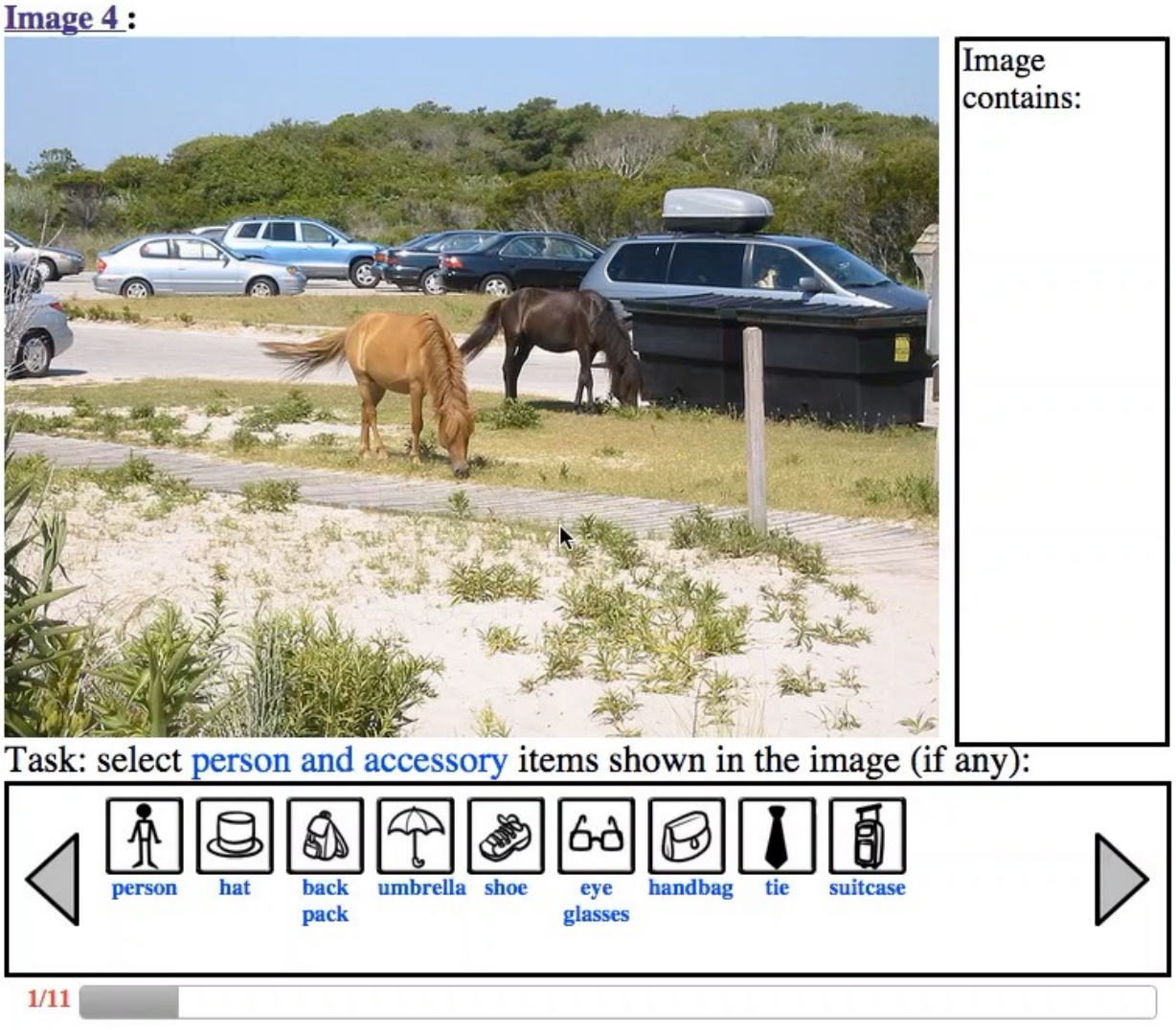
dog, bottle

2. Instance spotting

3. Instance segmentation

https://cocodataset.org

1. Category Labeling



https://cocodataset.org

2. Instance Spotting

https://cocodataset.org

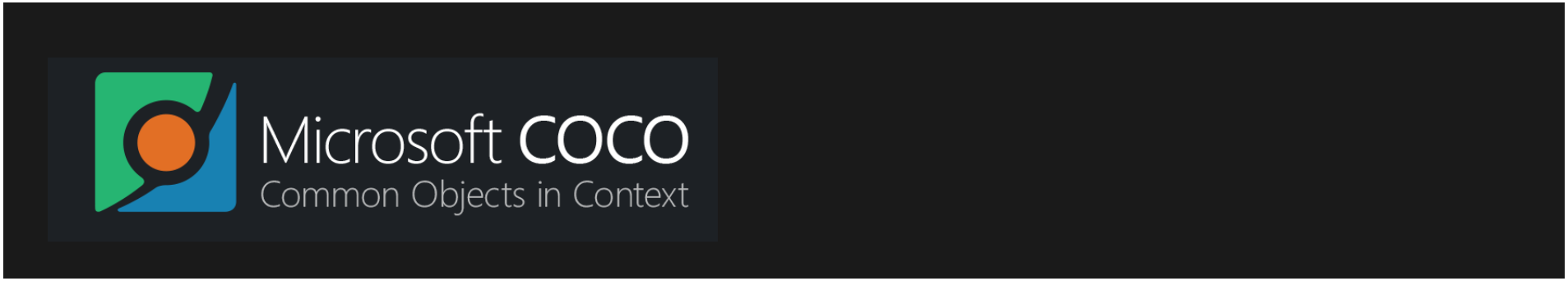
3. Instance Segmentation

https://cocodataset.org


1

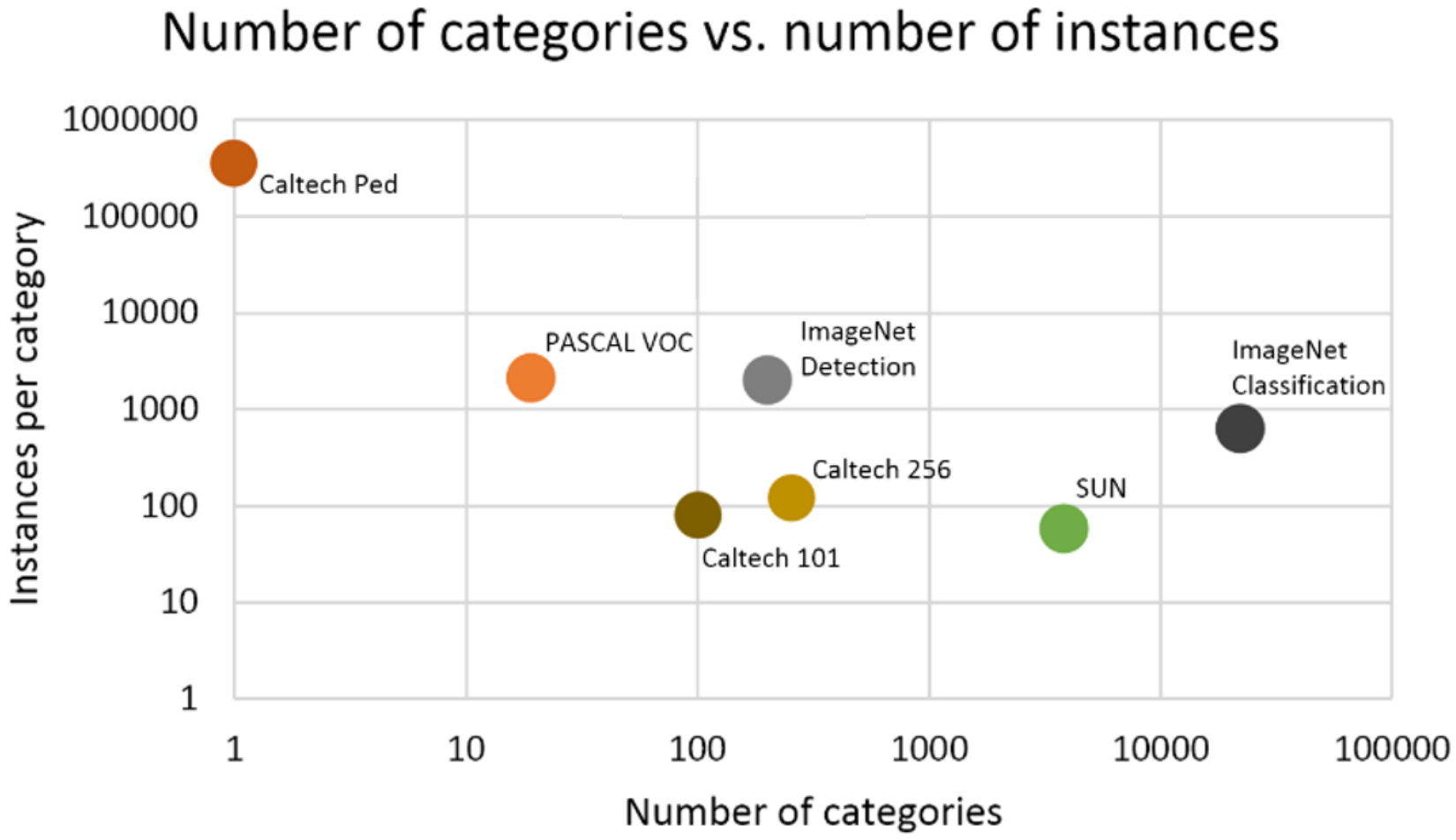

After training

https://cocodataset.org

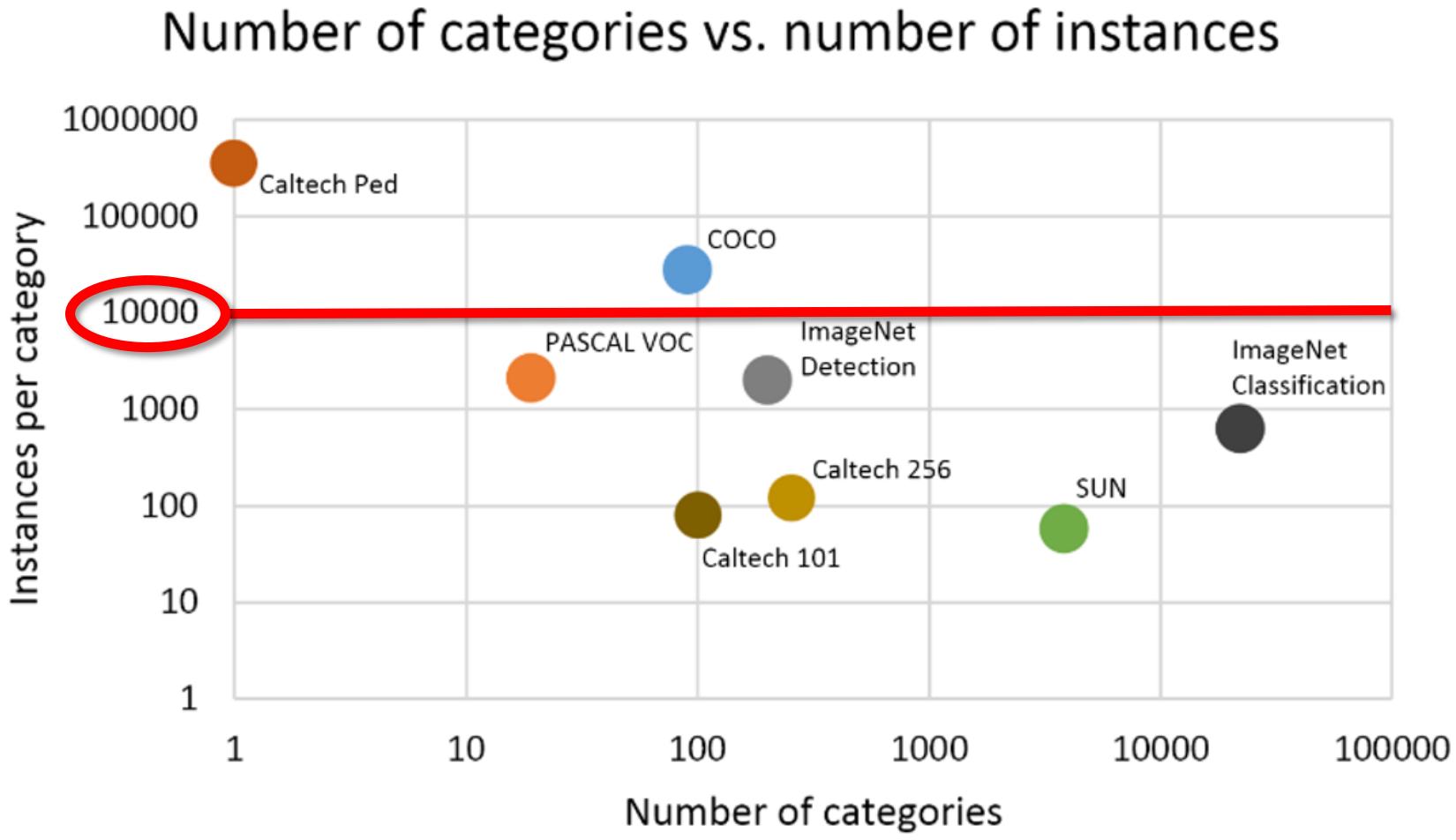


Properties

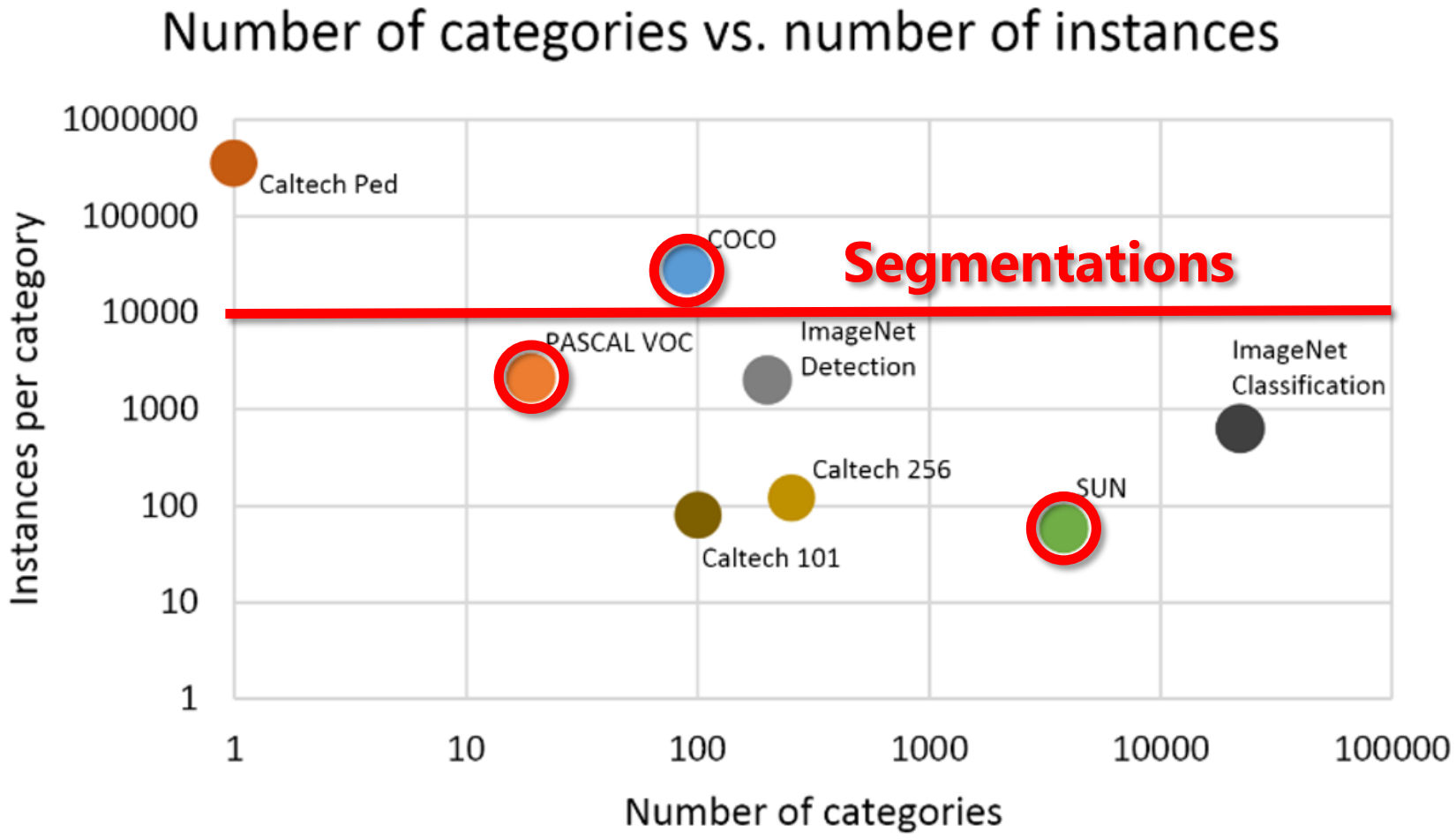
https://cocodataset.org



https://cocodataset.org



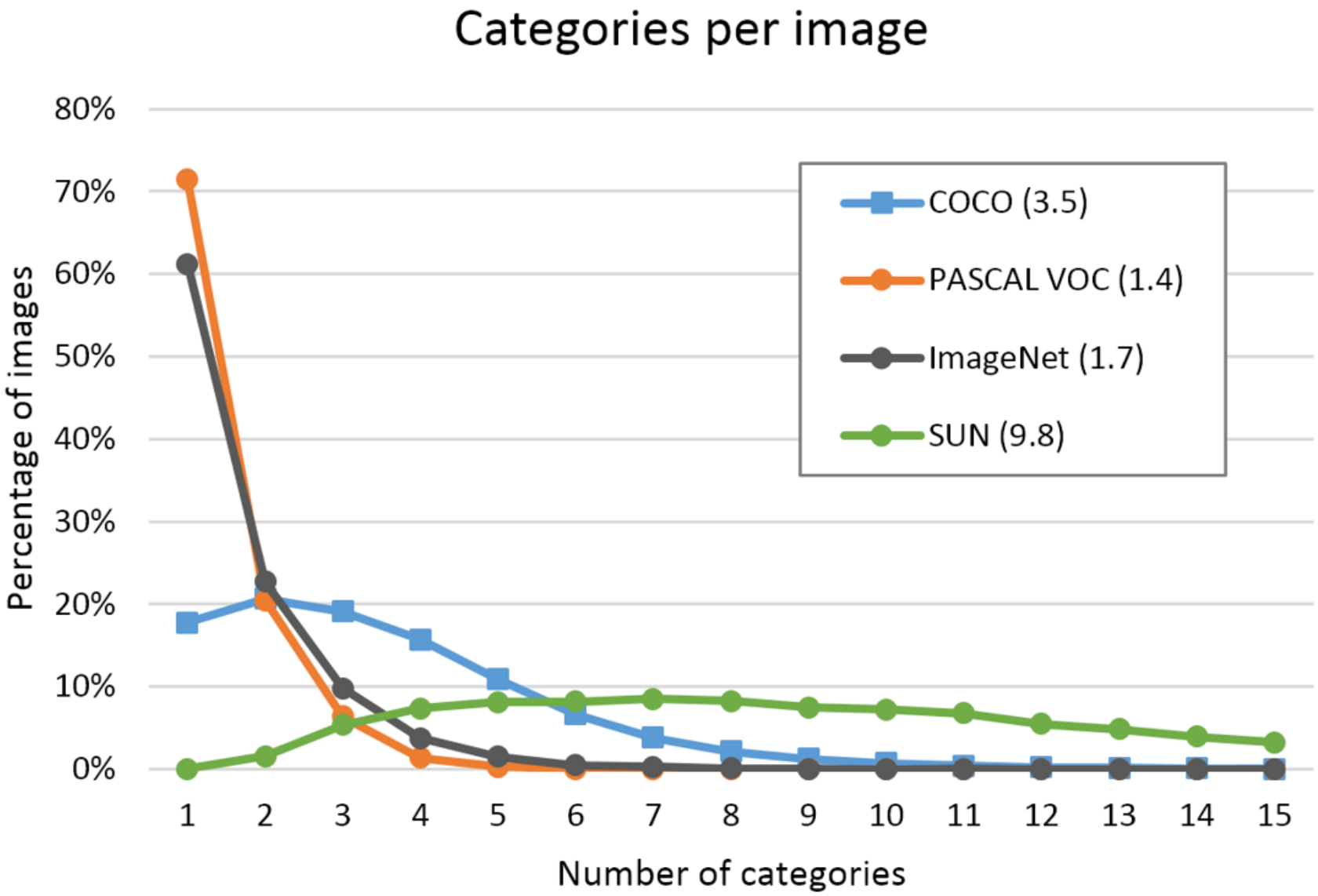
https://cocodataset.org



https://cocodataset.org

- 330,000 images
- >2 million instances (700k people)
- Every instance is segmented
- 7.7 instances per image (3.5 categories)

https://cocodataset.org



https://cocodataset.org

Detection Performance

(DPM V5)

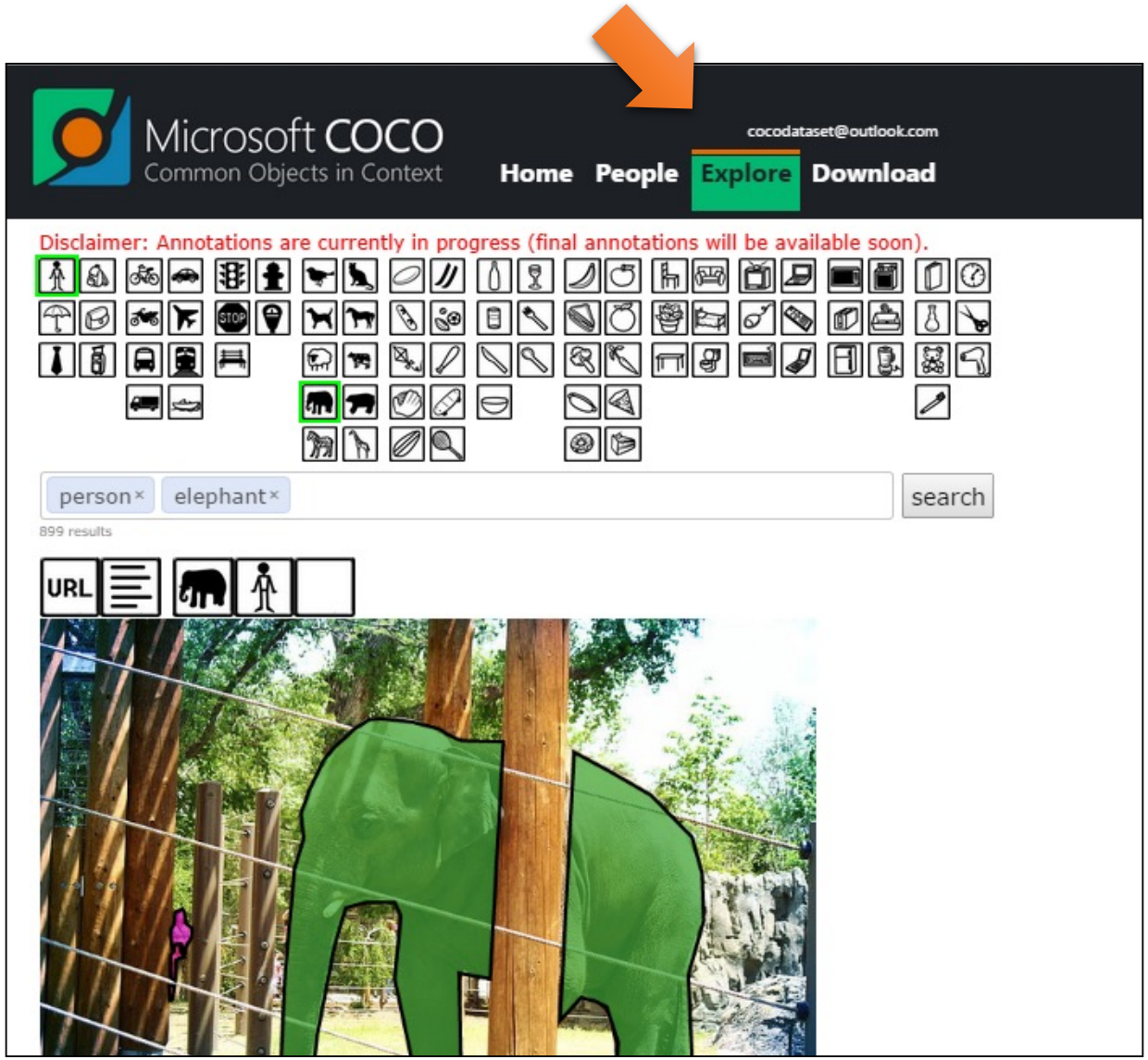
	Person (mAP)	Average (mAP)
PASCAL VOC	41.3	29.6
MS COCO	17.5	16.9

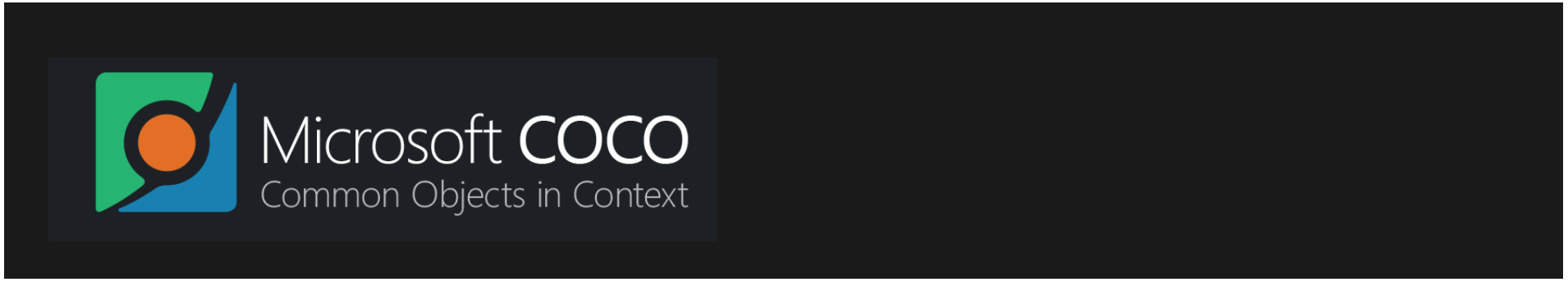
[44] P. Felzenszwalb, R. Girshick, D. McAllester, and D. Ramanan, "Object detection with discriminatively trained part-based models," *PAMI*, vol. 32, no. 9, pp. 1627–1645, 2010.

[45] R. Girshick, P. Felzenszwalb, and D. McAllester, "Discriminatively trained deformable part models, release 5," *PAMI*, 2012.

https://cocodataset.org

https://cocodataset.org





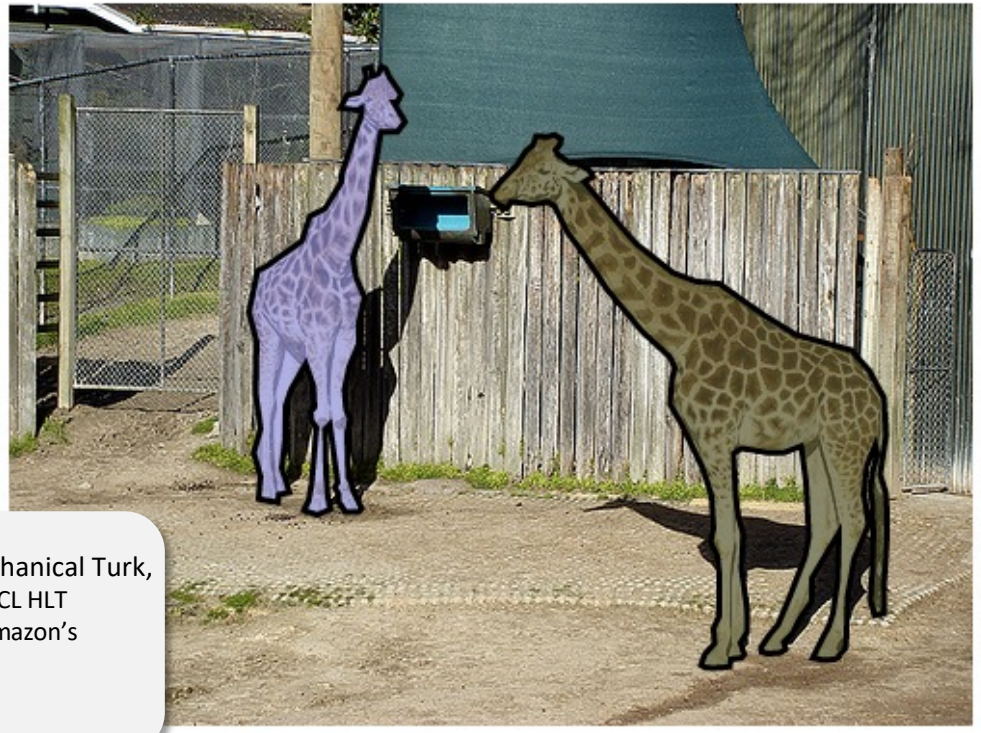
Beyond detection

https://cocodataset.org

Beyond detection

✓ Sentences

two giraffe standing next to each other in front of a wooden fence.
two giraffes standing in the dirt near a gate.
two giraffes stand by a food box awaiting the goods.
two giraffes are standing next to a wooden fence.
two giraffes standing alone by a picket fence.

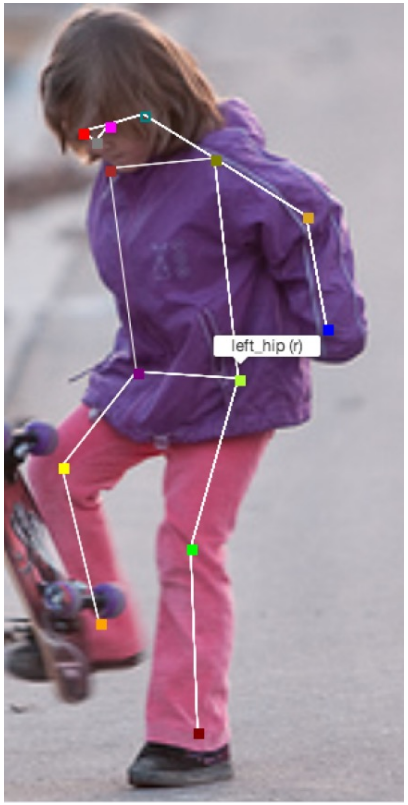


Collecting Image Annotations Using Amazon's Mechanical Turk, C. Rashtchian, P. Young, M. Hodosh, J. Hockenmaier, NAACL HLT Workshop on Creating Speech and Language Data with Amazon's Mechanical Turk, 2010

https://cocodataset.org

Beyond detection

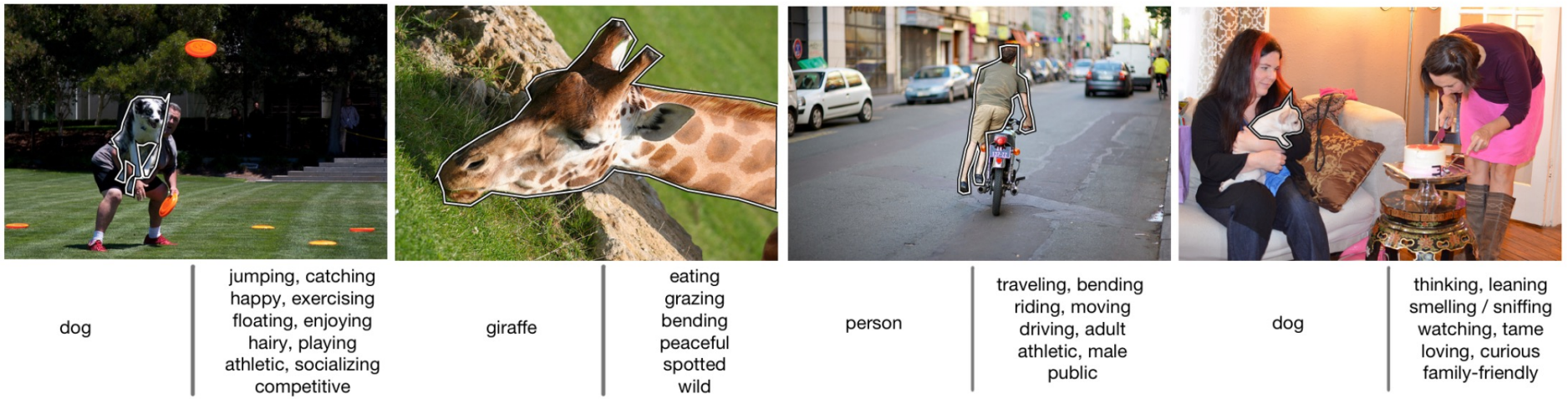
✓ Keypoints
(provided by Facebook)



https://cocodataset.org

Beyond detection

✓ Attributes



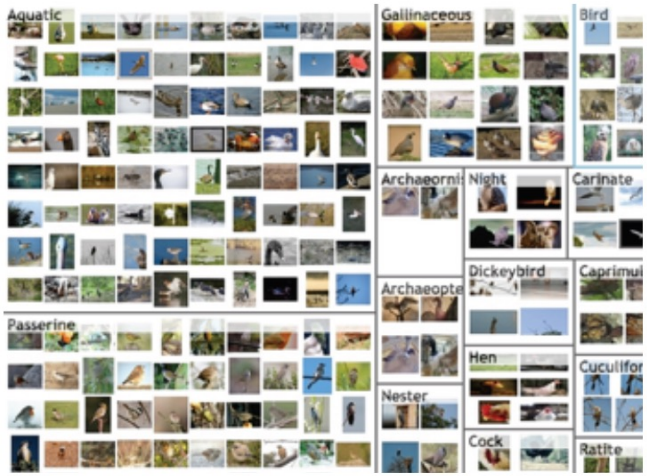
Genevieve Patterson, James Hays.
COCO Attributes: Attributes for People, Animals, and Objects. ECCV 2016.

https://cocodataset.org

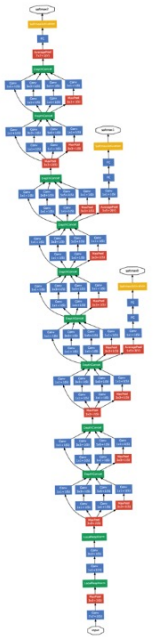
How do you actually train these things?

Roughly speaking:

Gather labeled data



Find a ConvNet architecture



Minimize the loss

Mini-batch Gradient Descent

Loop:

1. Sample a batch of training data (~100 images)
2. Forwards pass: compute loss (avg. over batch)
3. Backwards pass: compute gradient
4. Update all parameters

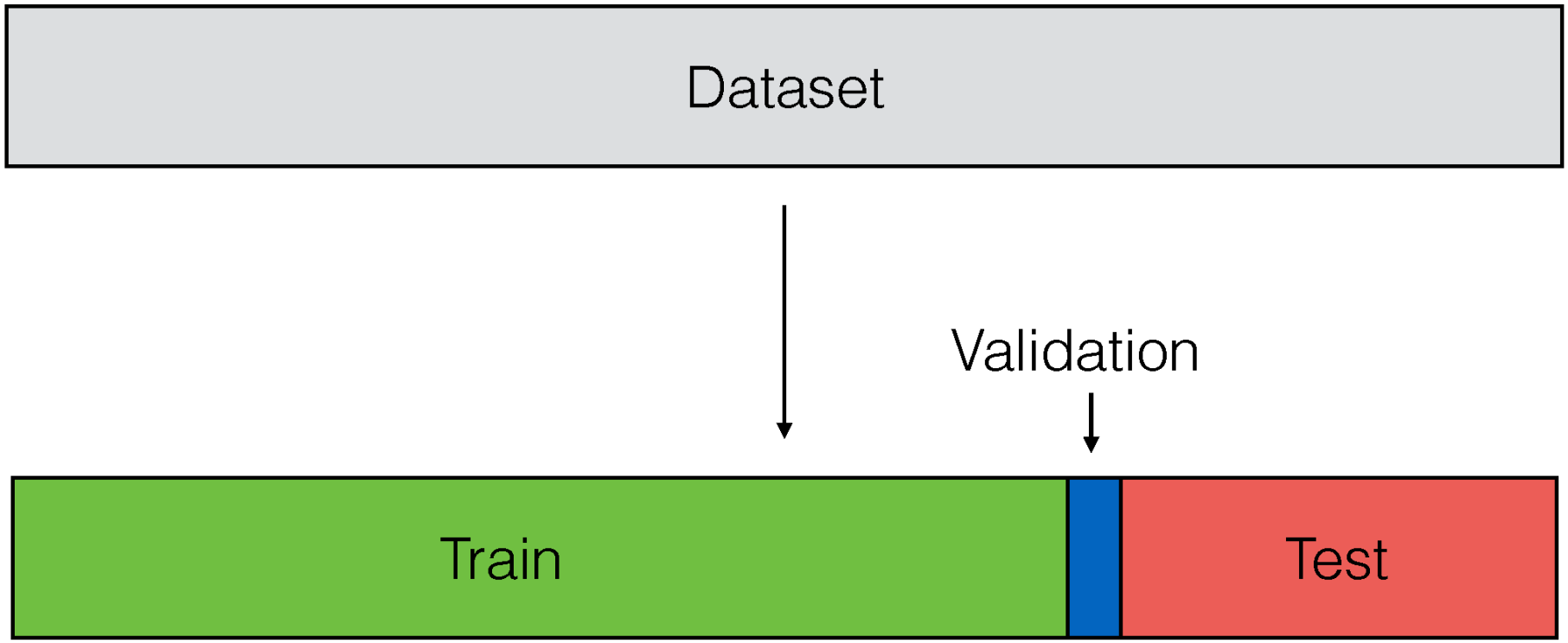
Note: usually called “stochastic gradient descent” even though SGD has a batch size of 1

Training a convolutional neural network

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

(0) Dataset split

Split your data into “train”, “validation”, and “test”:



(0) Dataset split

Train: gradient descent and fine-tuning of parameters

Validation: determining hyper-parameters (learning rate, regularization strength, etc) and picking an architecture

Test: estimate real-world performance
(e.g. accuracy = fraction correctly classified)

(0) Dataset split

Be careful with false discovery:

To avoid false discovery, once we have used a test set once, we should *not use it again* (but nobody follows this rule, since it's expensive to collect datasets)

Instead, try and avoid looking at the test score until the end

(1) Data preprocessing

Preprocess the data so that learning is better conditioned:

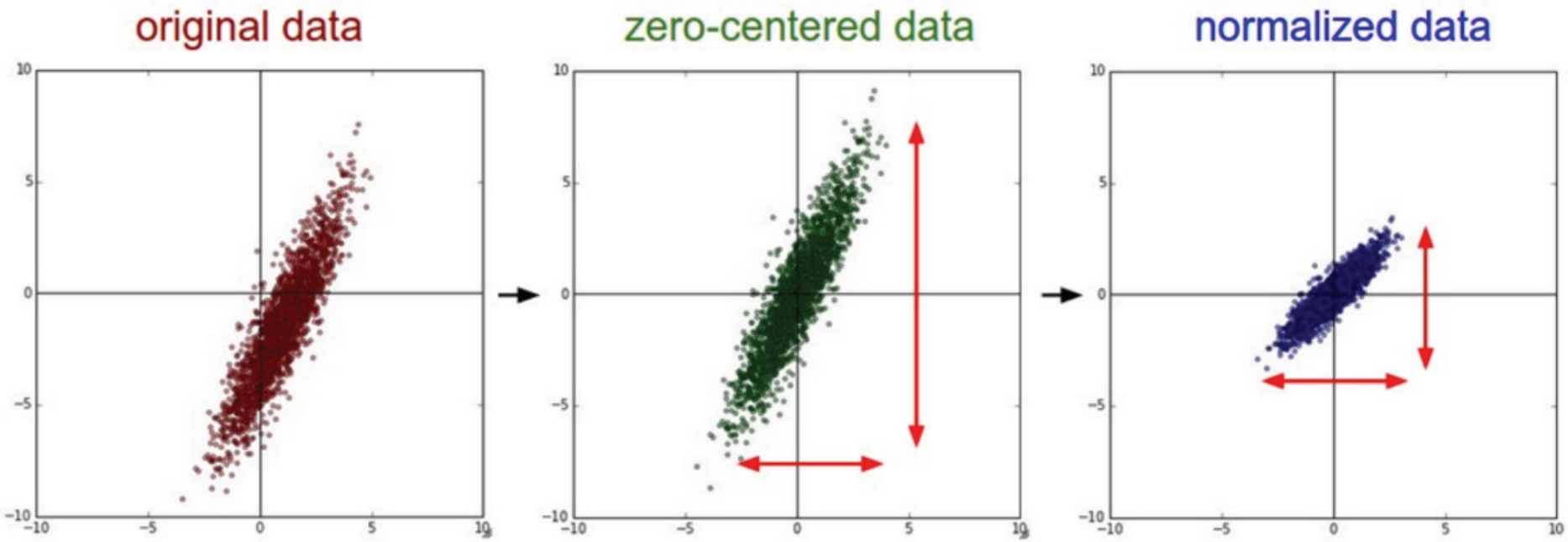
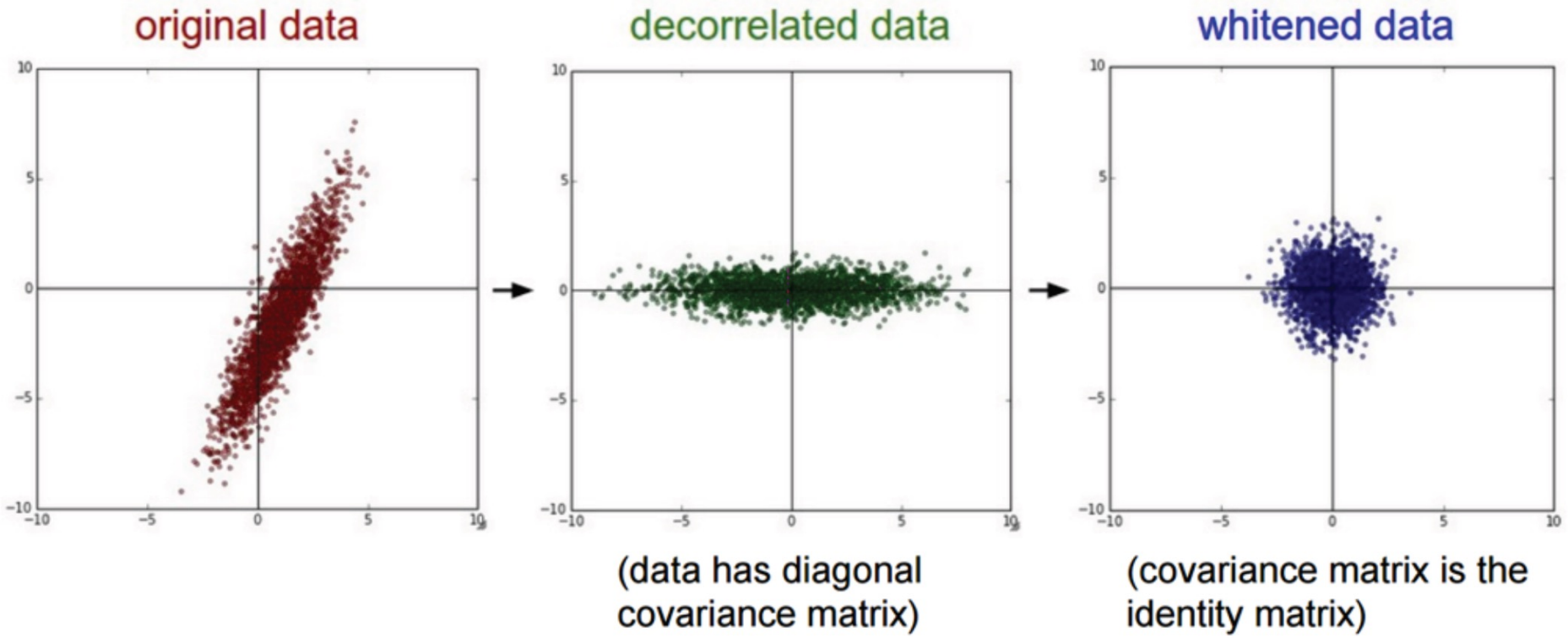


Figure: Andrej Karpathy

(1) Data preprocessing

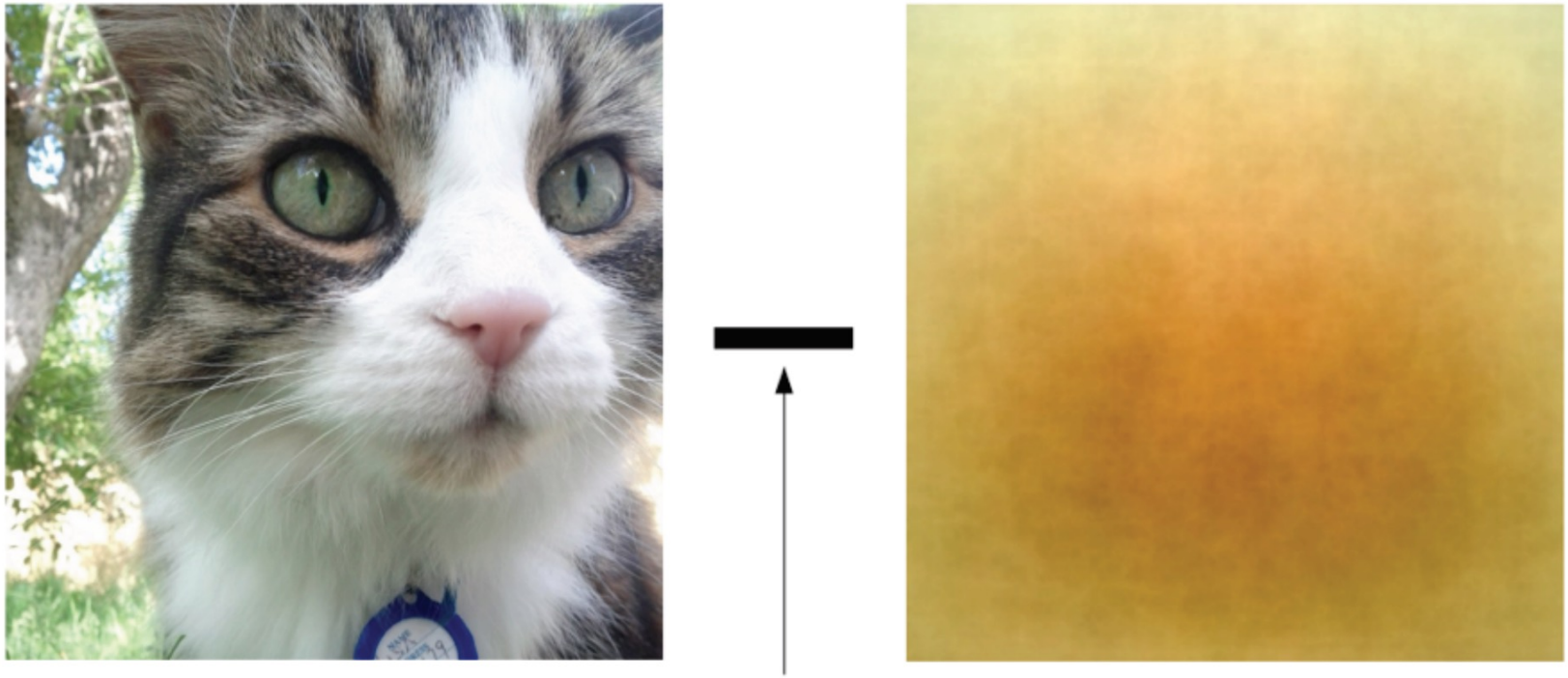
In practice, you may also see **PCA** and **Whitening** of the data:



Slide: Andrej Karpathy

(1) Data preprocessing

For ConvNets, typically only the mean is subtracted.



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

Figure: Alex Krizhevsky

(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

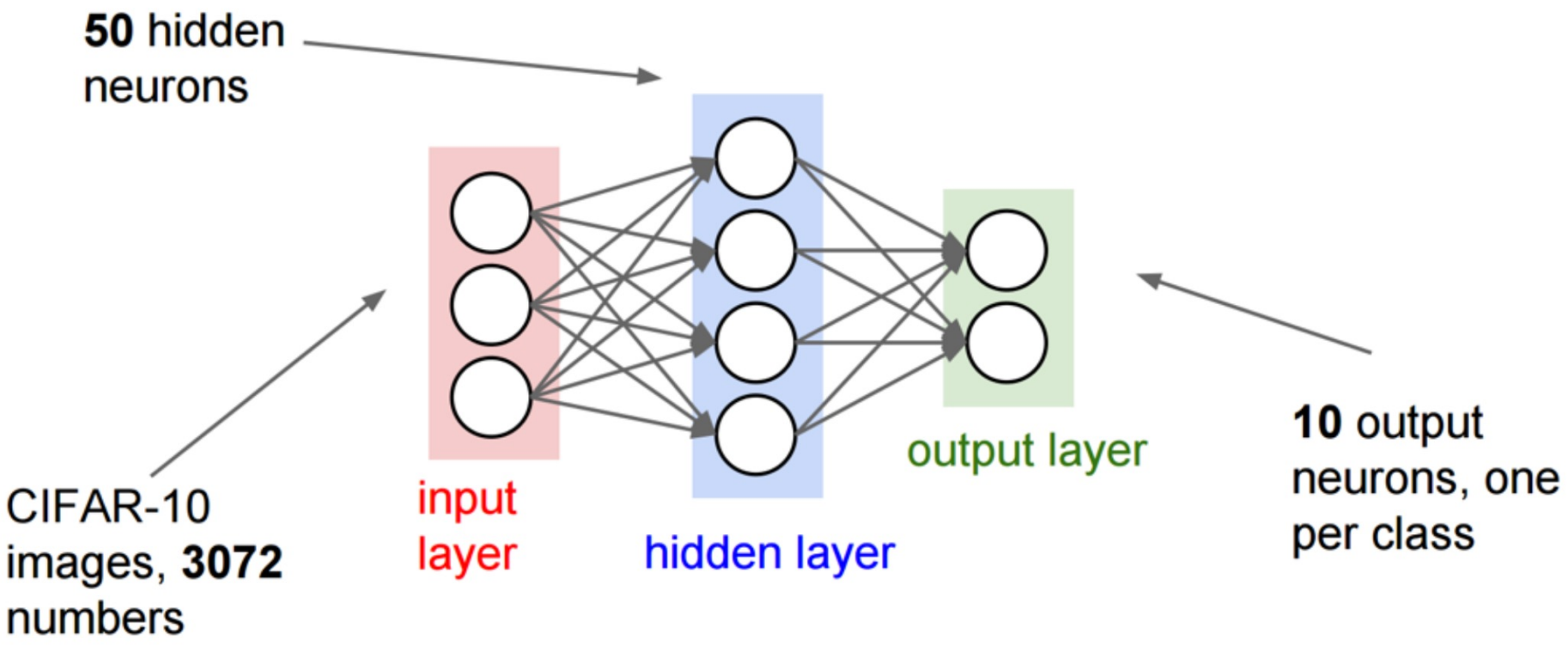
Randomly reflect horizontally

Perform the augmentation live during training

Figure: Alex Krizhevsky

(2) Choose your architecture

Toy example: one hidden layer of size 50



Slide: Andrej Karpathy

https://towardsdatascience.com/convolutional-neural-networks-most-common-architectures-6a2b5d22479d

Most Popular Convolutional Neural Networks Architectures

Learn about their structure and how to implement them!

Victor Roman Mar 21, 2020 · 8 min read ★

Picture from Unsplash

Most Common Architectures

There are research teams fully dedicated to developing deep learning architectures for CNN and to training them in huge datasets, so we will take advantage of this and use them instead of creating a new architecture every time we face a new problem.

This will provide us with stability and precision.

The most common deep learning architectures for CNN today are:

- VGG
- ResNet
- Inception
- Xception

(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)

(the magnitude is important and this is not optimal — more on this later)

Set the bias to zero (or small nonzero):

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

Slide: Andrej Karpathy

(3) Check that the loss is reasonable

```
def init_two_layer_model(input_size, hidden_size, output_size):
  # initialize a model
  model = {}
  model['W1'] = 0.0001 * np.random.randn(input_size, hidden_size)
  model['b1'] = np.zeros(hidden_size)
  model['W2'] = 0.0001 * np.random.randn(hidden_size, output_size)
  model['b2'] = np.zeros(output_size)
  return model
```

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
loss, grad = two_layer_net(X_train, model, y_train, 0.0)
print loss
```

disable regularization

returns the loss and the gradient for all parameters

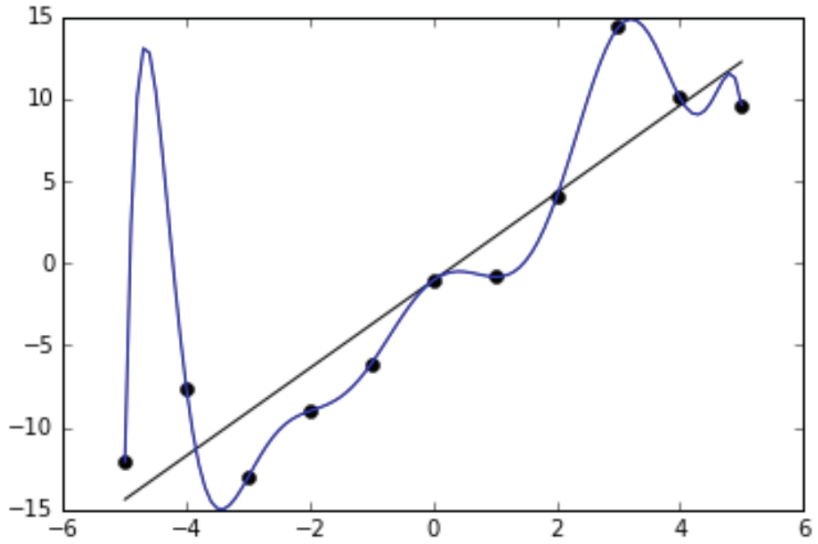
Slide: Andrej Karpathy

Overfitting

Overfitting: modeling noise in the training set instead of the "true" underlying relationship

Underfitting: insufficiently modeling the relationship in the training set

General rule: models that are "bigger" or have more capacity are more likely to overfit

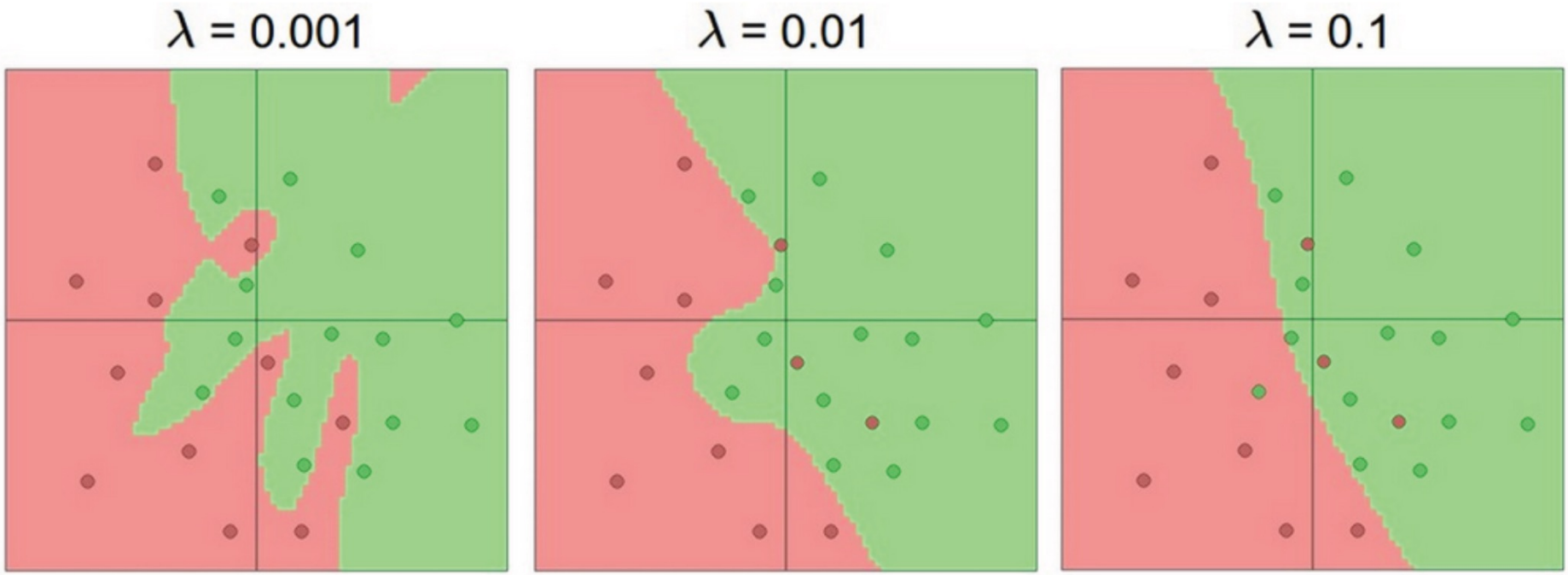


[Image: https://en.wikipedia.org/wiki/File:Overfitted_Data.png]

Regularization

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]

“Weight decay”

Regularization is also called “weight decay” because the weights “decay” each iteration:

$$L_{\text{reg}} = \lambda \frac{1}{2} \|W\|_2^2 \quad \longrightarrow \quad \frac{\partial L}{\partial W} = \lambda W$$

Gradient descent step:

$$W \leftarrow W - \alpha\lambda W - \frac{\partial L_{\text{data}}}{\partial W}$$

Weight decay: $\alpha\lambda$ (weights always decay by this amount)

Note: biases are sometimes excluded from regularization

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

Example Regularizers

L2 regularization $L_{\text{reg}} = \lambda \frac{1}{2} ||W||_2^2$

(L2 regularization encourages small weights)

L1 regularization $L_{\text{reg}} = \lambda ||W||_1 = \lambda \sum_{ij} |W_{ij}|$

(L1 regularization encourages sparse weights:
weights are encouraged to reduce to exactly zero)

"Elastic net" $L_{\text{reg}} = \lambda_1 ||W||_1 + \lambda_2 ||W||_2^2$

(combine L1 and L2 regularization)

Max norm

Clamp weights to some max norm $||W||_2^2 \leq c$

(3) Check that the loss is reasonable

```
def init_two_layer_model(input_size, hidden_size, output_size):
  # initialize a model
  model = {}
  model['W1'] = 0.0001 * np.random.randn(input_size, hidden_size)
  model['b1'] = np.zeros(hidden_size)
  model['W2'] = 0.0001 * np.random.randn(hidden_size, output_size)
  model['b2'] = np.zeros(output_size)
  return model
```

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
loss, grad = two_layer_net(X_train, model, y_train, 1e3)
print loss
```

crank up regularization

loss went up, good. (sanity check)

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)
```


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

Let's start with small regularization and find the learning rate that makes the loss decrease:

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = True,
                                  learning_rate=1e-6, verbose=True)
```

(4) Find a learning rate

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = True,
                                  learning_rate=1e-6, verbose=True)
```

```
Finished epoch 1 / 10: cost 2.302576, train: 0.080000, val 0.103000, lr 1.000000e-06
Finished epoch 2 / 10: cost 2.302582, train: 0.121000, val 0.124000, lr 1.000000e-06
Finished epoch 3 / 10: cost 2.302558, train: 0.119000, val 0.138000, lr 1.000000e-06
Finished epoch 4 / 10: cost 2.302519, train: 0.127000, val 0.151000, lr 1.000000e-06
Finished epoch 5 / 10: cost 2.302517, train: 0.158000, val 0.171000, lr 1.000000e-06
Finished epoch 6 / 10: cost 2.302518, train: 0.179000, val 0.172000, lr 1.000000e-06
Finished epoch 7 / 10: cost 2.302466, train: 0.180000, val 0.176000, lr 1.000000e-06
Finished epoch 8 / 10: cost 2.302452, train: 0.175000, val 0.185000, lr 1.000000e-06
Finished epoch 9 / 10: cost 2.302459, train: 0.206000, val 0.192000, lr 1.000000e-06
Finished epoch 10 / 10: cost 2.302420, train: 0.190000, val 0.192000, lr 1.000000e-06
finished optimization. best validation accuracy: 0.192000
```

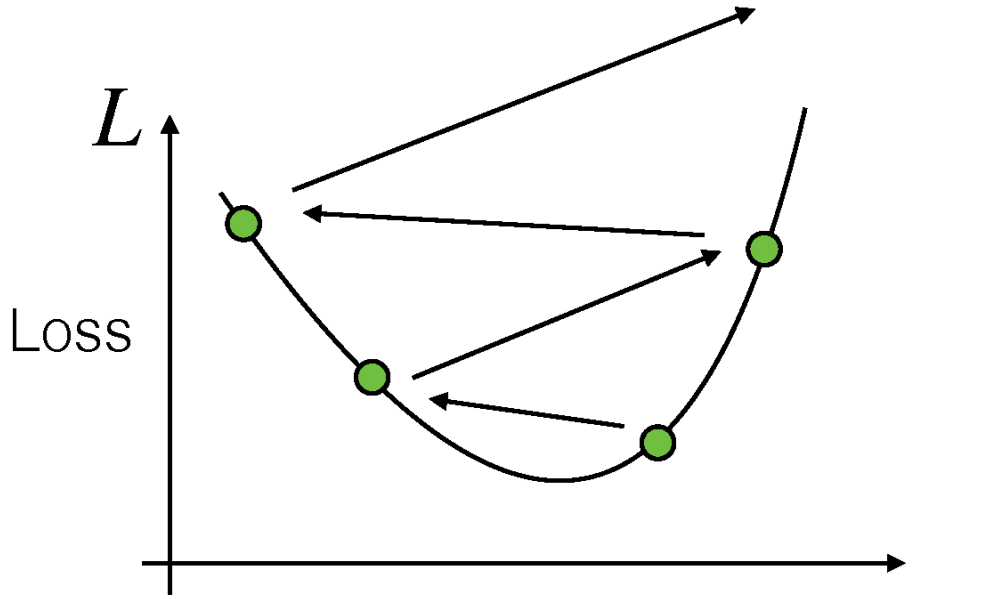
Loss barely changes **Why is the accuracy 20%?**

(learning rate is too low or regularization too high)

Slide: Andrej Karpathy

(4) Find a learning rate

Learning rate: 1e6 **(large)** — what could go wrong?



(4) Find a learning rate

Coarse to fine search

First stage: only a few epochs (passes through the data) to get a rough idea

Second stage: longer running time, finer search

Tip: if loss > 3 * original loss, quit early (learning rate too high)

Slide: Andrej Karpathy

(4) Find a learning rate

Training is slow. We normally visualize as we go.

Plot the loss

For very small learning rates, the loss decreases linearly and slowly

Larger learning rates tend to look more exponential

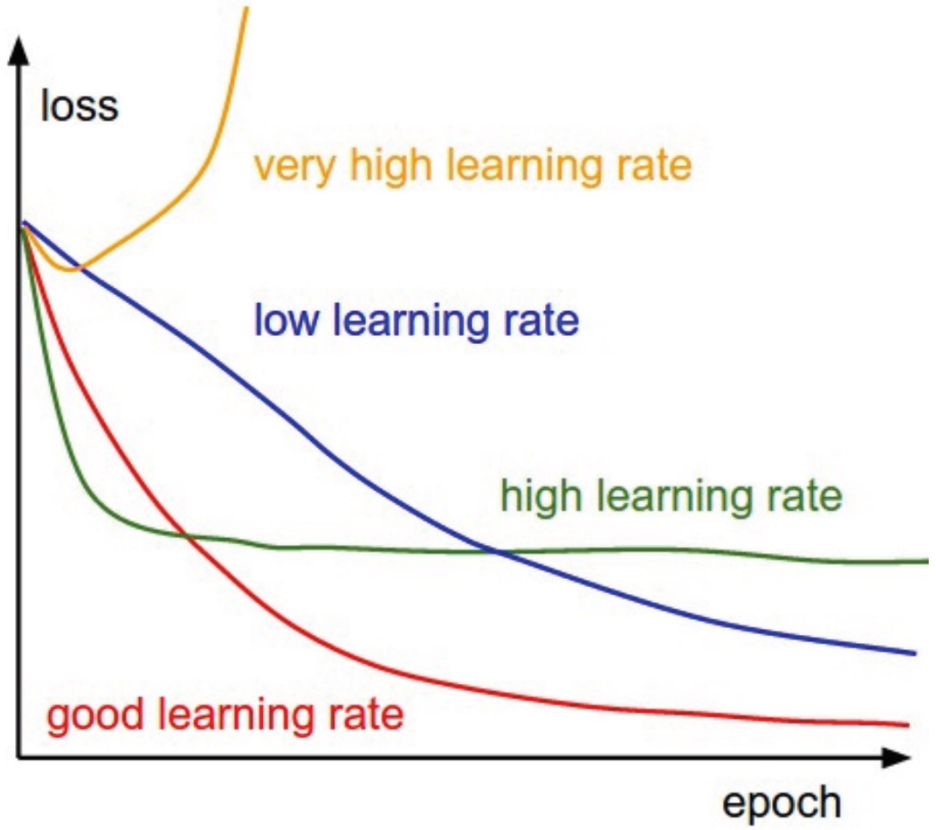
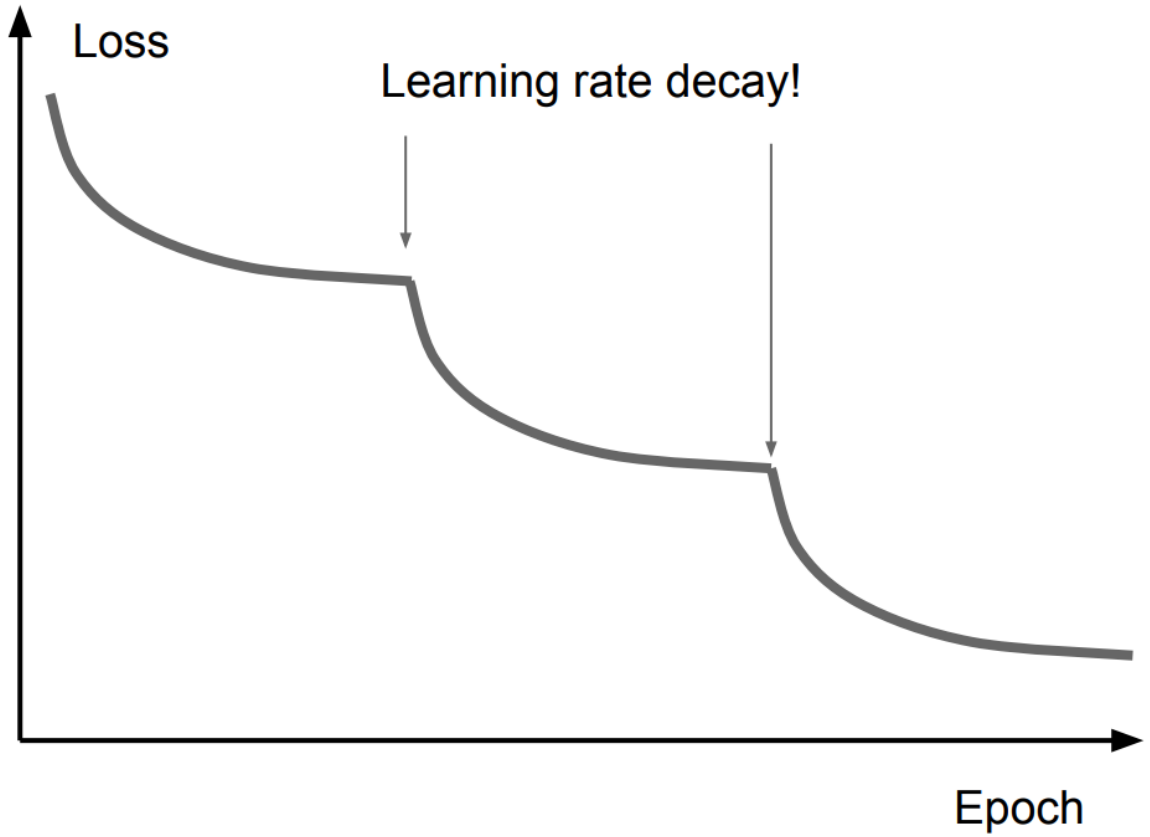


Figure: Andrej Karpathy

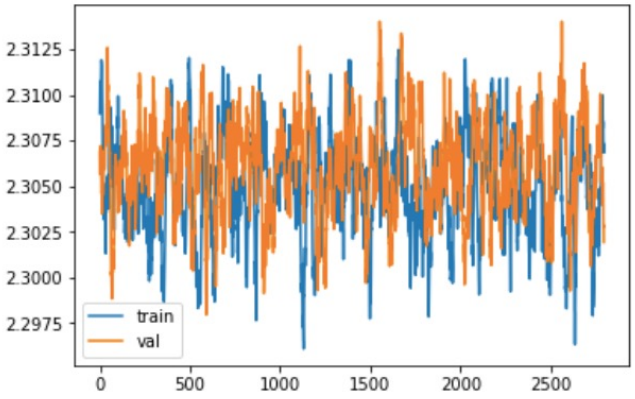
A typical phenomenon



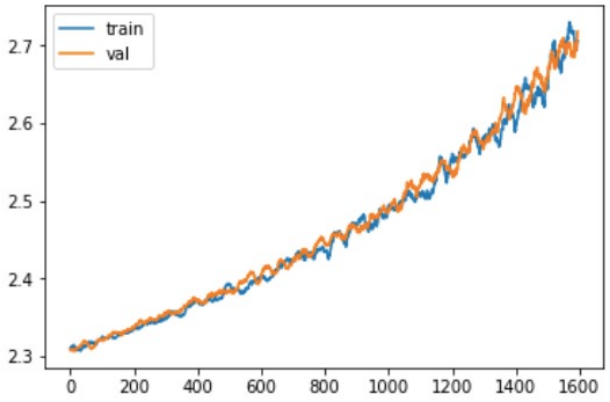
- Why does the learning curve look like this?

Image source: Stanford CS231n

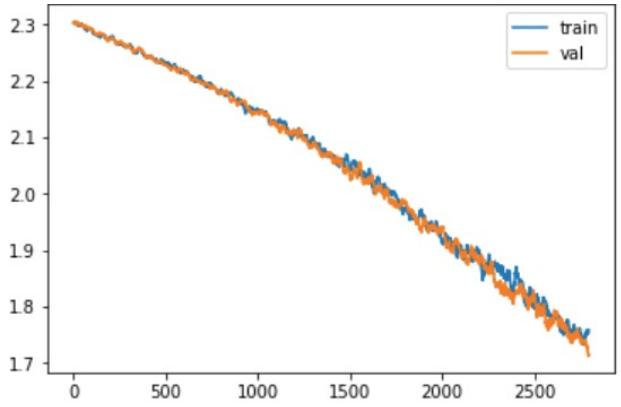
Debugging learning curves



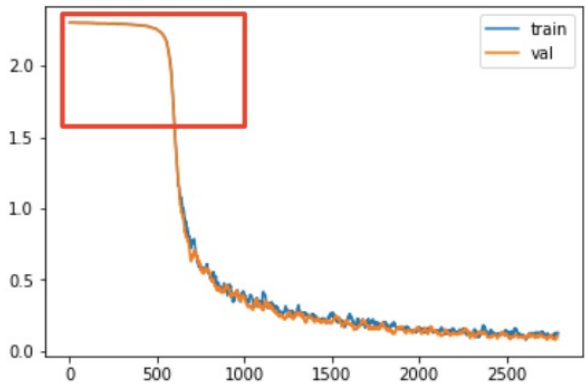
Not training
Bug in update calculation?



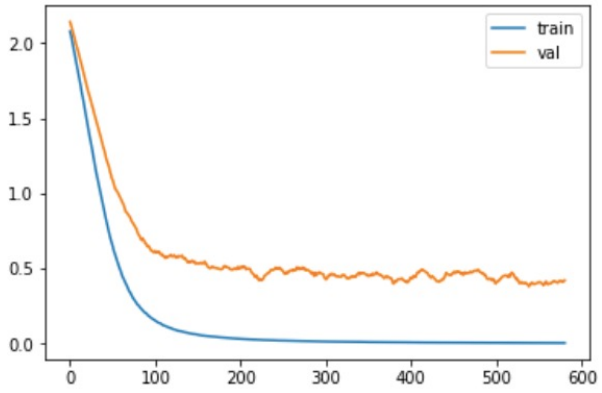
Error increasing
Bug in update calculation?



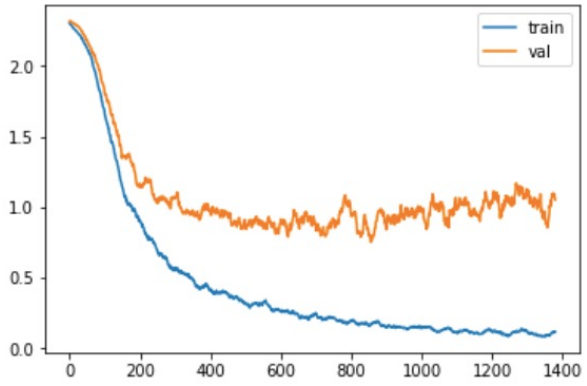
Error decreasing
Not converged yet



Slow start
Suboptimal initialization?



Possible overfitting



Definite overfitting

Image source: [Stanford CS231n]

Early stopping

- Idea: do not train a network to achieve too low training error
- Monitor validation error to decide when to stop

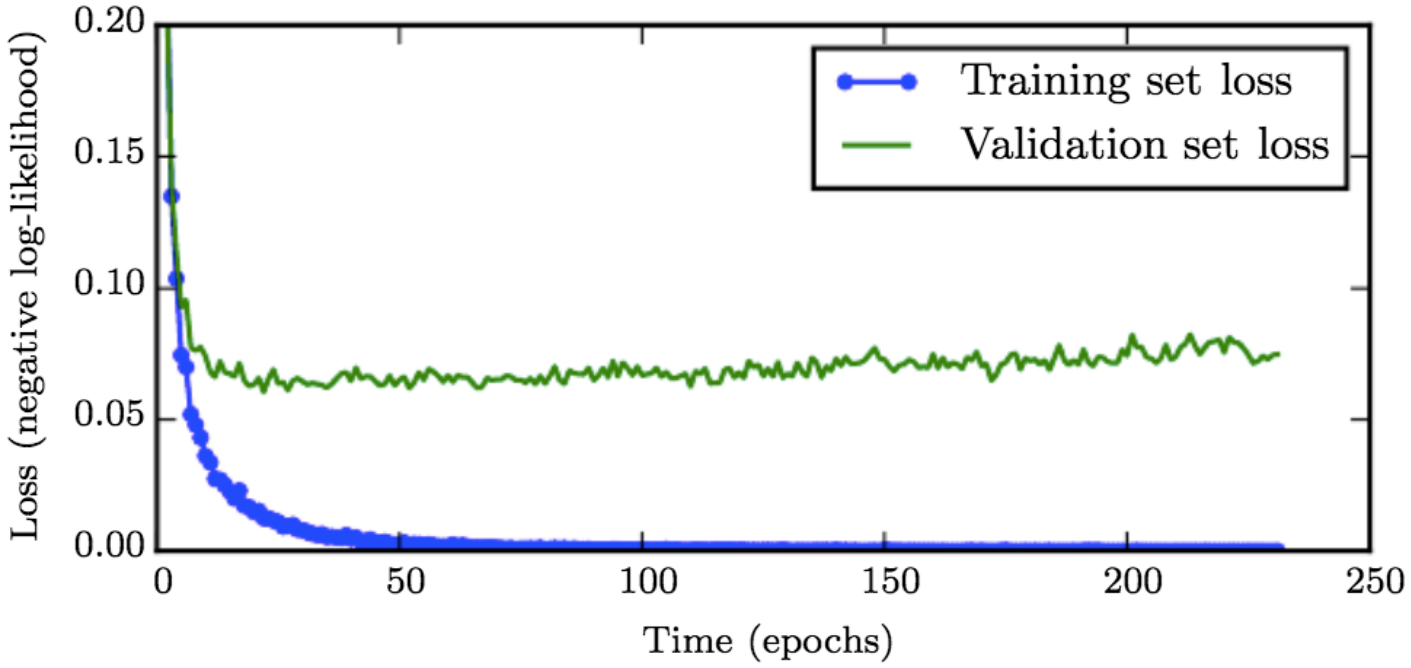


Figure from Deep Learning Book

(4) Find a learning rate

Normally, you don't have the budget for lots of cross-validation —> visualize as you go

Typical training loss:

Why is it varying so rapidly?

The width of the curve is related to the batchsize — if too noisy, increase the batch size

Possibly too linear
(learning rate too small)

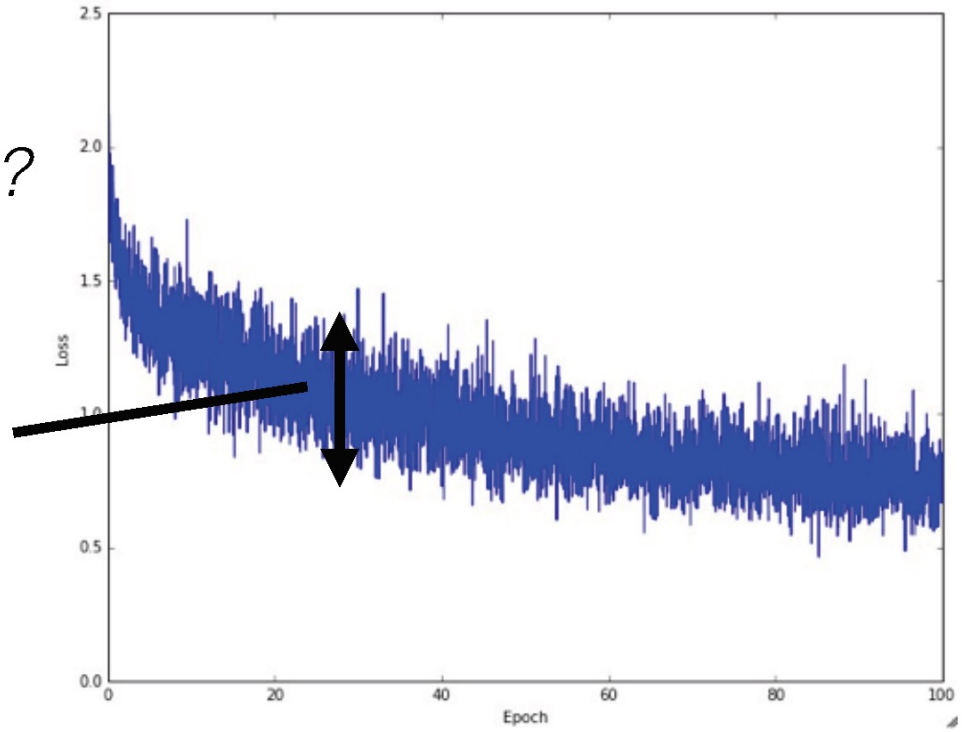


Figure: Andrej Karpathy

(4) Find a learning rate

Visualize the weights

Noisy weights: possibly regularization not strong enough

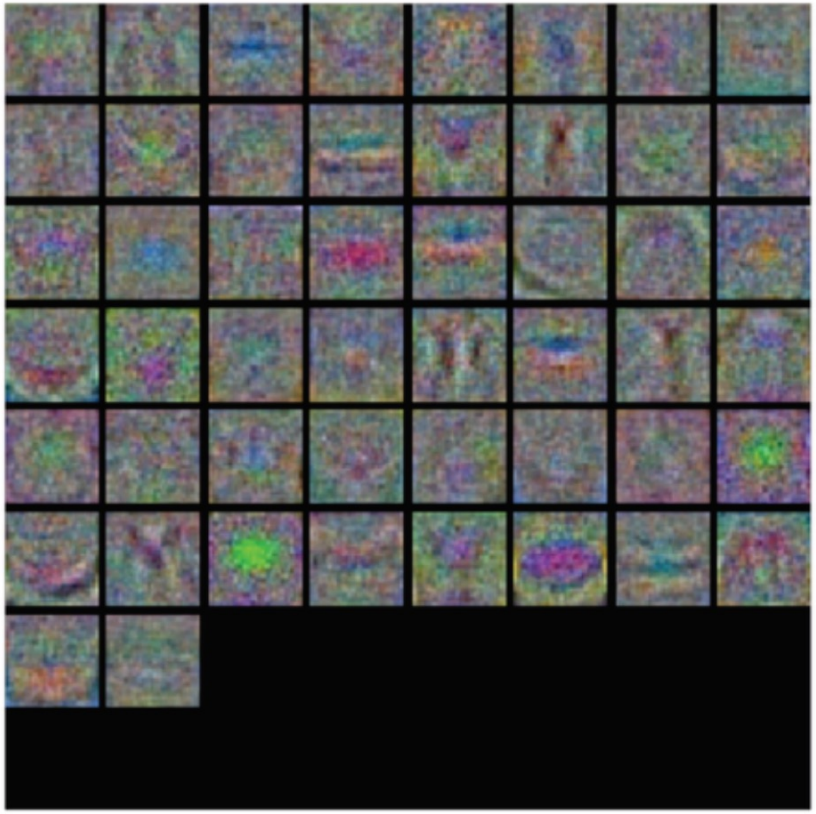
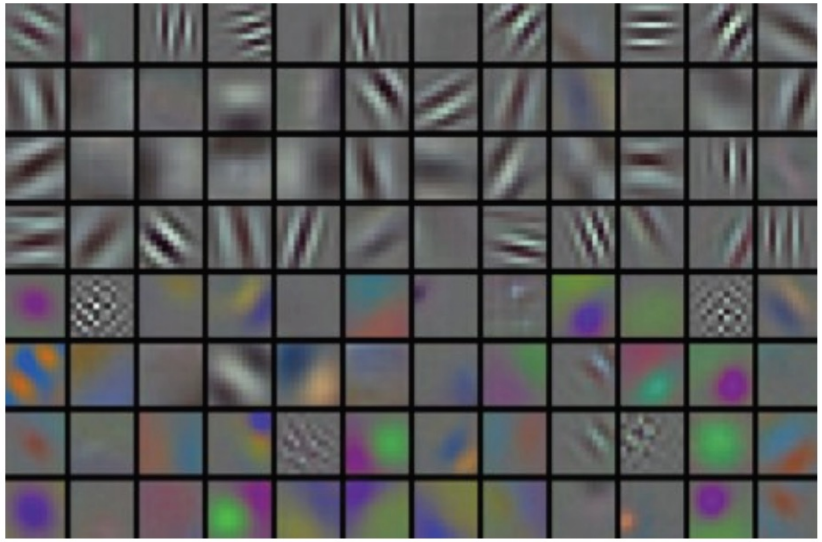


Figure: Andrej Karpathy

(4) Find a learning rate

Visualize the weights



Nice clean weights:
training is proceeding well

Figure: Alex Krizhevsky , Andrej Karpathy

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)

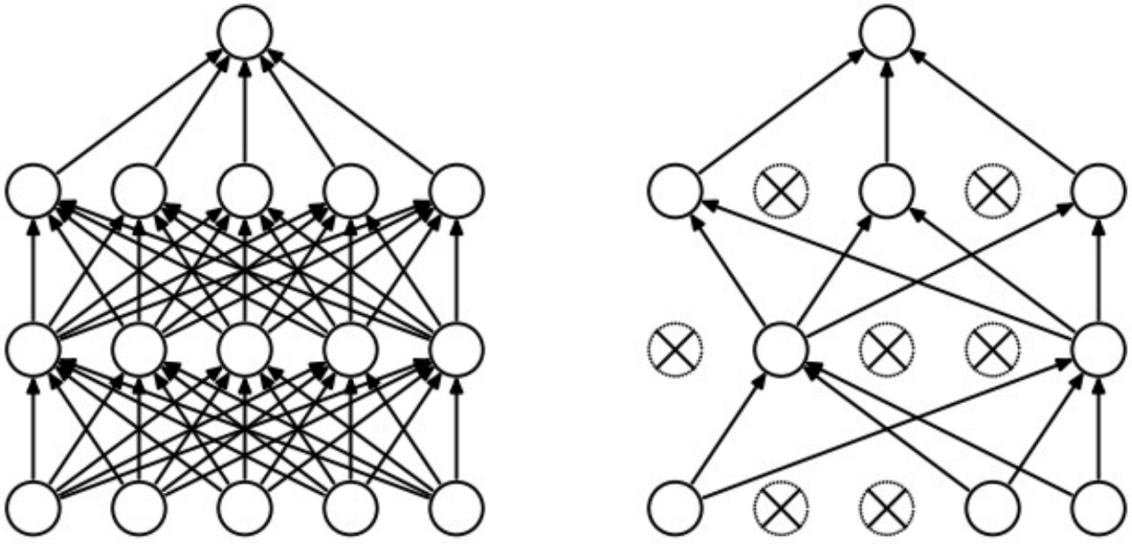
Summary of things to fiddle

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

Neural network parameters

Dropout

Simple but powerful technique to reduce overfitting:



(a) Standard Neural Net

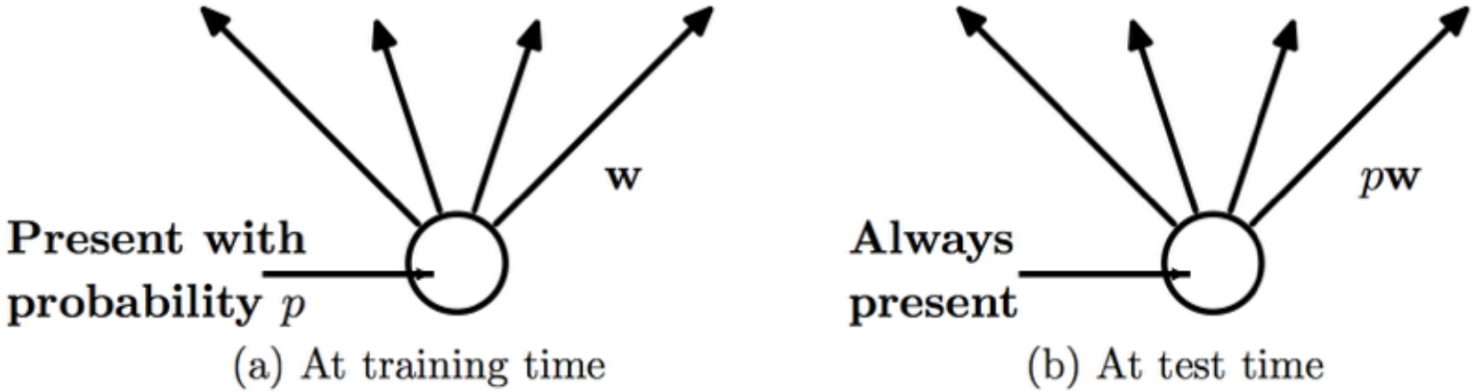
(b) After applying dropout.

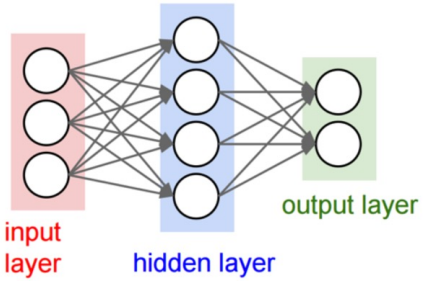
Note: Dropout can be interpreted as an approximation to taking the geometric mean of an ensemble of exponentially many models

[Srivasta et al, “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”, JMLR 2014]

Dropout

Simple but powerful technique to reduce overfitting:

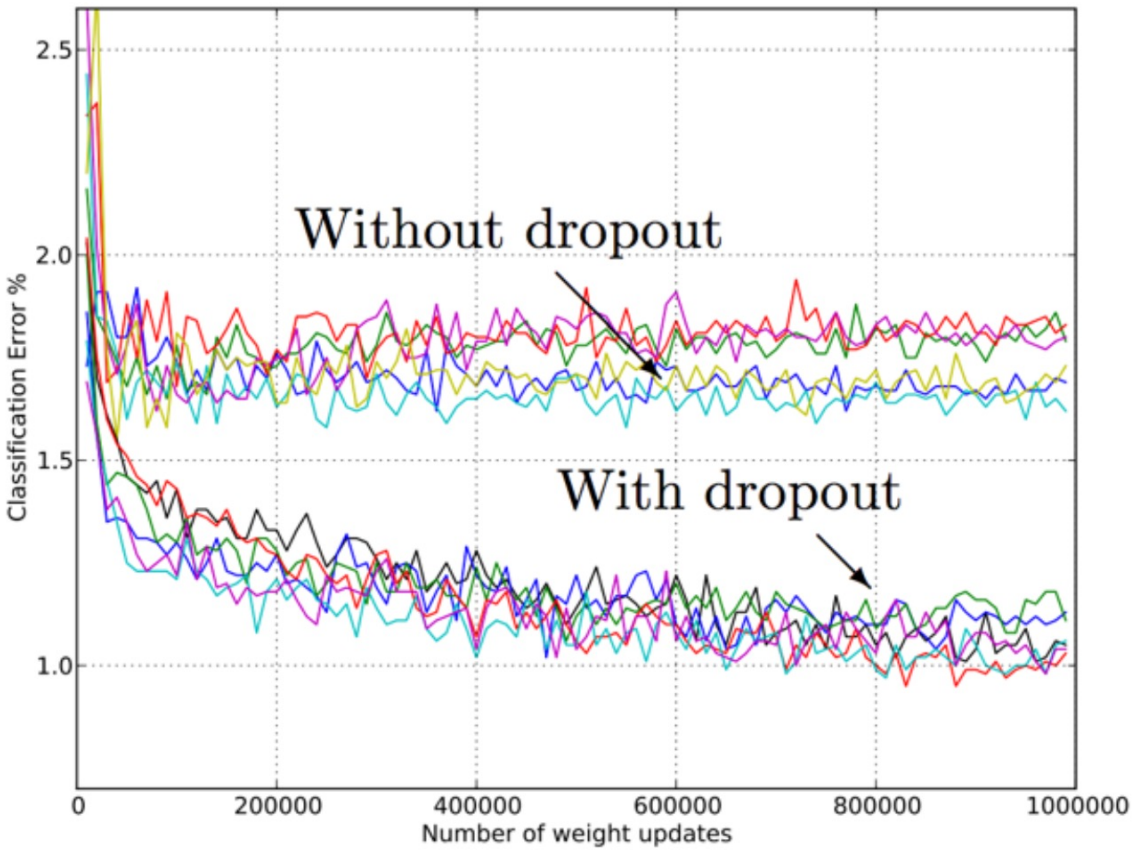




[Srivasta et al, "Dropout: A Simple Way to Prevent Neural Networks from Overfitting", JMLR 2014]

Dropout

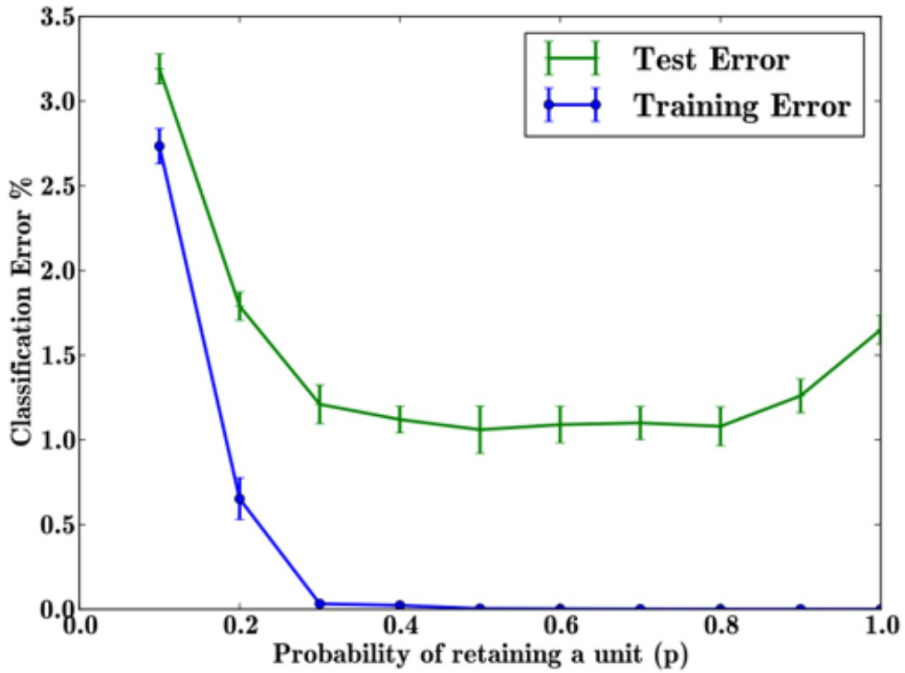
Simple but powerful technique to reduce overfitting:



[Srivasta et al, "Dropout: A Simple Way to Prevent Neural Networks from Overfitting", JMLR 2014]

Dropout

How much dropout? Around p = 0.5



Dropout

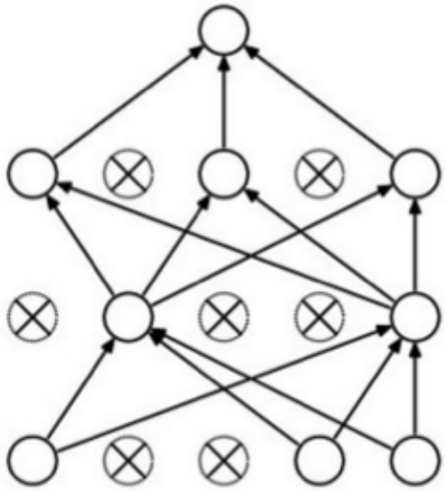
```python
p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
  """ X contains the data """

  # forward pass for example 3-layer neural network
  H1 = np.maximum(0, np.dot(W1, X) + b1)
  U1 = np.random.rand(*H1.shape) < p # first dropout mask
  H1 *= U1 # drop!
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  U2 = np.random.rand(*H2.shape) < p # second dropout mask
  H2 *= U2 # drop!
  out = np.dot(W3, H2) + b3

  # backward pass: compute gradients... (not shown)
  # perform parameter update... (not shown)
```

Example forward pass with a 3-layer network using dropout



(note, here X is a single input)

Figure: Andrej Karpathy

Dropout

Test time: scale the activations

Expected value of a neuron *h* with dropout:

$$E[h] = ph + (1-p)0 = ph$$

```
def predict(X):
  # ensembled forward pass
  H1 = np.maximum(0, np.dot(W1, X) + b1) * p # NOTE: scale the activations
  H2 = np.maximum(0, np.dot(W2, H1) + b2) * p # NOTE: scale the activations
  out = np.dot(W3, H2) + b3
```

We want to keep the same expected value

Figure: Andrej Karpathy

Advanced optimizers

- Momentum (SGD + momentum)
- Adagrad (Adaptive Gradient Algorithm)
- RMSprop (Root Mean Square Propagation)
- Adam (Adaptive Moment Estimation)

SGD with momentum

What will SGD do?

[Image source]

SGD with momentum

- Introduce a “momentum” variable and associated “friction” coefficient :

  - Typically start with 0, gradually increase over time

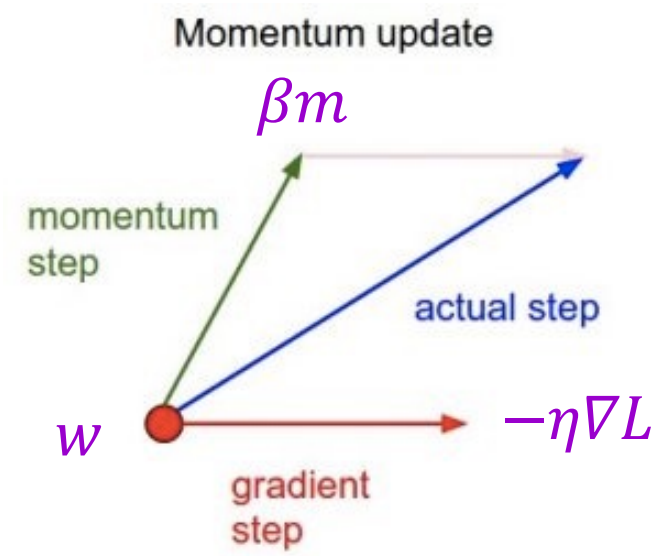


Image source

SGD with momentum

- Introduce a “momentum” variable and associated “friction” coefficient :

  - Move faster in directions with consistent gradient
  - Avoid oscillating in directions with large but inconsistent gradients

Standard SGD

SGD with momentum

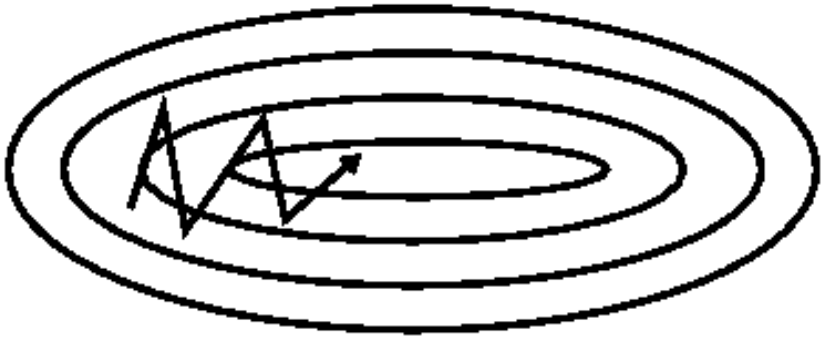
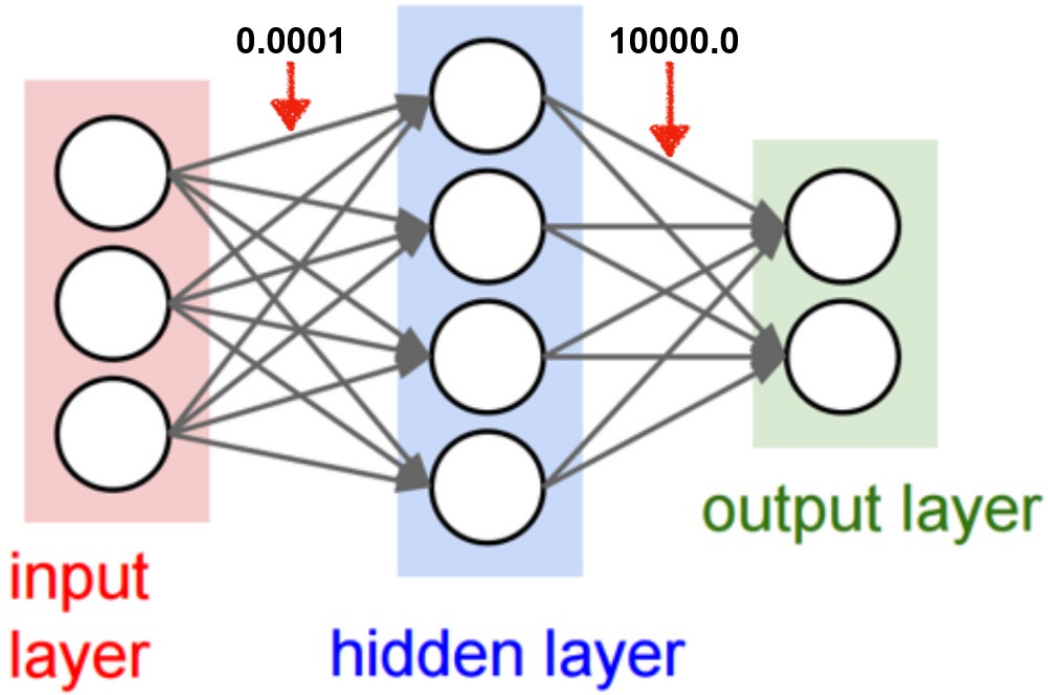


Image source

Not all gradients are the same…

- Gradients of different layers have different magnitudes
- Shouldn't we set different learning rates for different parameters?



Summary of optimizers

- SGD + momentum
- Adagrad (Adaptive Gradient Algorithm)
  - Keep track of history of gradients magnitudes
  - Problem: decay typically too fast
- RMSprop (Root Mean Square Propagation)
  - Fixes fast rate decay with an exponentially decaying average of squared gradients
- Adam (Adaptive Moment Estimation)
  - Combines Momentum and RMSprop by using both the moving average of gradients and the moving average of squared gradients

Other optimization tricks

- **Adding noise to gradients**: SGD with Langevin dynamics helps to jump out of local minima (e.g., Welling and Teh, 2011)

- **Cyclical learning rates**: increase learning rate from time to time to jump out of local minima (Smith 2017)

Bleeding edge

The Road Less Scheduled

Aaron Defazio, Xingyu Alice Yang, Harsh Mehta, Konstantin Mishchenko, Ahmed Khaled, Ashok Cutkosky

Existing learning rate schedules that do not require specification of the optimization stopping step T are greatly outperformed by learning rate schedules that depend on T. We propose an approach that avoids the need for this stopping time by eschewing the use of schedules entirely, while exhibiting state-of-the-art performance compared to schedules across a wide family of problems ranging from convex problems to large-scale deep learning problems. Our Schedule-Free approach introduces no additional hyper-parameters over standard optimizers with momentum. Our method is a direct consequence of a new theory we develop that unifies scheduling and iterate averaging. An open source implementation of our method is available at this https URL. Schedule-Free AdamW is the core algorithm behind our winning entry to the MLCommons 2024 AlgoPerf Algorithmic Efficiency Challenge Self-Tuning track.

Subjects: **Machine Learning (cs.LG)**; Artificial Intelligence (cs.AI); Optimization and Control (math.OC); Machine Learning (stat.ML)
Cite as: arXiv:2405.15682 **[cs.LG]**
(or arXiv:2405.15682v4 **[cs.LG]** for this version)
https://doi.org/10.48550/arXiv.2405.15682

Submission history

From: Aaron Defazio [view email]
[v1] Fri, 24 May 2024 16:20:46 UTC (688 KB)
[v2] Thu, 30 May 2024 21:50:15 UTC (689 KB)
[v3] Wed, 7 Aug 2024 17:44:58 UTC (689 KB)
[v4] Tue, 29 Oct 2024 22:40:23 UTC (654 KB)

References

Basic reading: No standard textbooks yet! Some good resources:

- https://sites.google.com/site/deeplearningsummerschool/
- http://www.deeplearningbook.org/
- http://www.cs.toronto.edu/~hinton/absps/NatureDeepReview.pdf