Indian Institute of Information Technology, Allahabad

Optimization & Regularization

By

Dr. Satish Kumar Singh & Dr. Shiv Ram Dubey
Computer Vision and Biometrics Lab
Department of Information Technology
Indian Institute of Information Technology, Allahabad

TEAM

Computer Vision and Biometrics Lab (CVBL)

Department of Information Technology

Indian Institute of Information Technology Allahabad

Course Instructors

Dr. Satish Kumar Singh, Associate Professor, IIIT Allahabad (Email: sk.singh@iiita.ac.in)

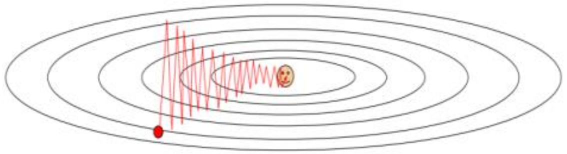
Dr. Shiv Ram Dubey, Assistant Professor, IIIT Allahabad (Email: srdubey@iiita.ac.in)

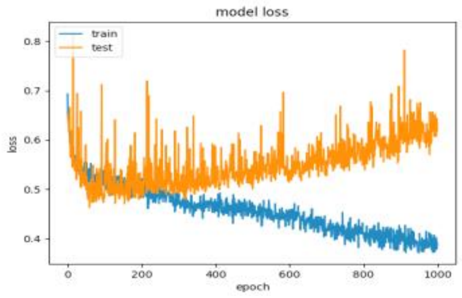
DISCLAIMER

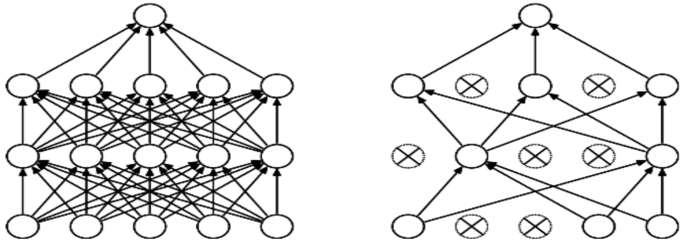
The content (text, image, and graphics) used in this slide are adopted from many sources for academic purposes. Broadly, the sources have been given due credit appropriately. However, there is a chance of missing out some original primary sources. The authors of this material do not claim any copyright of such material.

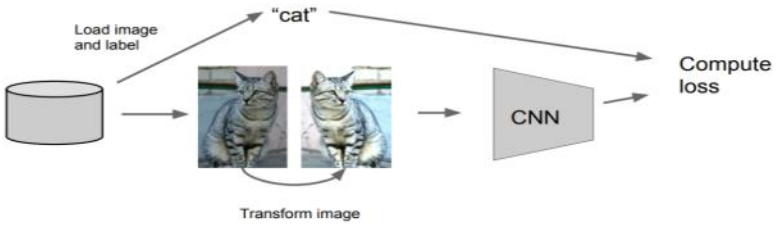
Training Aspects of CNN

- Optimization
- Learning Rate
- Regularization
- Dropout
- Batch Normalization
- Data Augmentation
- Transfer Learning
- Interpreting Loss Curve









Optimization

Source: cs231n

MINI-BATCH SGD

Loop:

1. **Sample** a batch of data
2. **Forward** prop it through the graph (network), get loss
3. **Backprop** to calculate the gradients
4. **Update** the parameters using the gradient

Source: cs231n

STOCHASTIC GRADIENT DESCENT (SGD)

The procedure of repeatedly evaluating the gradient of loss function and then performing a parameter update.

Vanilla (Original) Gradient Descent:

```
while True:
  dx = compute_gradient(x)
  x -= learning_rate * dx
```

Source: cs231n

SGD + MOMENTUM

SGD

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

```
while True:
  dx = compute_gradient(x)
  x -= learning_rate * dx
```

Source: cs231n

SGD + MOMENTUM

SGD

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

```
while True:
  dx = compute_gradient(x)
  x -= learning_rate * dx
```

SGD+Momentum

$$v_{t+1} = \rho v_t + \nabla f(x_t)$$
$$x_{t+1} = x_t - \alpha v_{t+1}$$

Source: cs231n

SGD + MOMENTUM

SGD

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

```
while True:
  dx = compute_gradient(x)
  x -= learning_rate * dx
```

SGD+Momentum

$$v_{t+1} = \rho v_t + \nabla f(x_t)$$
$$x_{t+1} = x_t - \alpha v_{t+1}$$

```
vx = 0
while True:
  dx = compute_gradient(x)
  vx = rho * vx + dx
  x -= learning_rate * vx
```

Source: cs231n

SGD + MOMENTUM

SGD

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

```
while True:
  dx = compute_gradient(x)
  x -= learning_rate * dx
```

SGD+Momentum

$$v_{t+1} = \rho v_t + \nabla f(x_t)$$
$$x_{t+1} = x_t - \alpha v_{t+1}$$

```
vx = 0
while True:
  dx = compute_gradient(x)
  vx = rho * vx + dx
  x -= learning_rate * vx
```

- Build up "velocity" in any direction that has consistent gradient
- Rho gives "friction"; typically rho=0.9 or 0.99

Source: cs231n

SGD + MOMENTUM

SGD

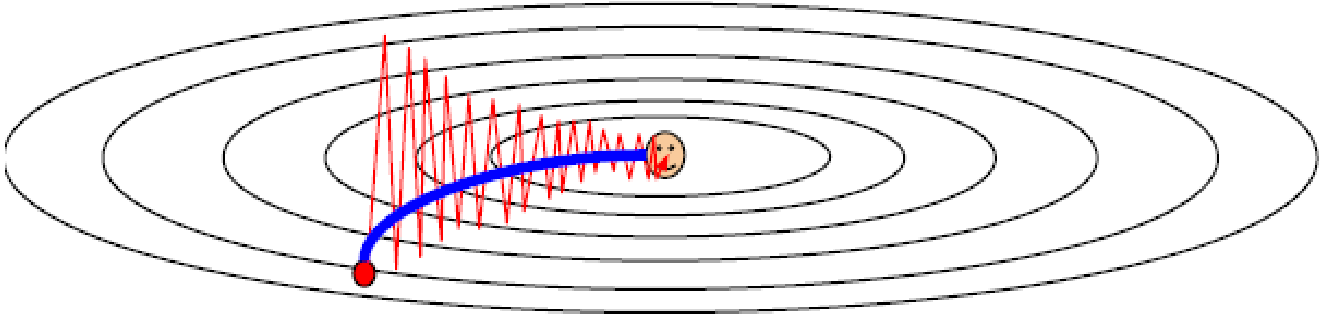
$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

```python
while True:
    dx = compute_gradient(x)
    x -= learning_rate * dx
```

SGD+Momentum

$$v_{t+1} = \rho v_t + \nabla f(x_t)$$
$$x_{t+1} = x_t - \alpha v_{t+1}$$

```python
vx = 0
while True:
    dx = compute_gradient(x)
    vx = rho * vx + dx
    x -= learning_rate * vx
```



Source: cs231n

ADAGRAD

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

Added element-wise scaling of the gradient based on the historical sum of squares in each dimension

Duchi et al, "Adaptive subgradient methods for online learning and stochastic optimization", JMLR 2011

Source: cs231n

ADAGRAD

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

What happens to the step size over long time?

Duchi et al, "Adaptive subgradient methods for online learning and stochastic optimization", JMLR 2011

Source: cs231n

ADAGRAD

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

What happens to the step size over long time?

Effective learning rate diminishing problem

Duchi et al, "Adaptive subgradient methods for online learning and stochastic optimization", JMLR 2011

Source: cs231n

RMSPROP

AdaGrad

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

RMSProp

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared = decay_rate * grad_squared + (1 - decay_rate) * dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

Tieleman and Hinton, 2012

Source: cs231n

ADAM

Kingma and Ba, "Adam: A method for stochastic optimization", ICLR 2015

```
first_moment = 0
second_moment = 0
while True:
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  x -= learning_rate * first_moment / (np.sqrt(second_moment) + 1e-7))
```

Source: cs231n

ADAM

Kingma and Ba, "Adam: A method for stochastic optimization", ICLR 2015

```
first_moment = 0
second_moment = 0
while True:
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  x -= learning_rate * first_moment / (np.sqrt(second_moment) + 1e-7))
```

Sort of like RMSProp with Momentum

Source: cs231n

ADAM

Kingma and Ba, "Adam: A method for stochastic optimization", ICLR 2015

```
first_moment = 0
second_moment = 0
while True:
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  x -= learning_rate * first_moment / (np.sqrt(second_moment) + 1e-7))
```

Sort of like RMSProp with Momentum

Problem:

Initially, second_moment=0 and beta2=0.999

After 1st iteration, second_moment -> close to zero

So, very large step for update of x

Source: cs231n

ADAM (WITH BIAS CORRECTION)

Kingma and Ba, "Adam: A method for stochastic optimization", ICLR 2015

```
first_moment = 0
second_moment = 0
for t in range(num_iterations):
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  first_unbias = first_moment / (1 - beta1 ** t)
  second_unbias = second_moment / (1 - beta2 ** t)
  x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7))
```

AdaGrad/
RMSProp

Bias Correction

Momentum

Bias correction for the fact that first and second moment estimates start at zero

Source: cs231n

ADAM (WITH BIAS CORRECTION)

Kingma and Ba, "Adam: A method for stochastic optimization", ICLR 2015

```
first_moment = 0
second_moment = 0
for t in range(num_iterations):
    dx = compute_gradient(x)
    first_moment = beta1 * first_moment  + (1 - beta1) * dx
    second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
    first_unbias = first_moment / (1 - beta1 ** t)
    second_unbias = second_moment / (1 - beta2 ** t)
    x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7))
```

AdaGrad/ RMSProp

Bias Correction

Momentum

Bias correction for the fact that first and second moment estimates start at zero

Adam with **beta1 = 0.9**,
beta2 = 0.999, and **learning_rate = 1e-3 or 5e-4**
is a **great starting point** for many models!

Source: cs231n

OPTIMIZER

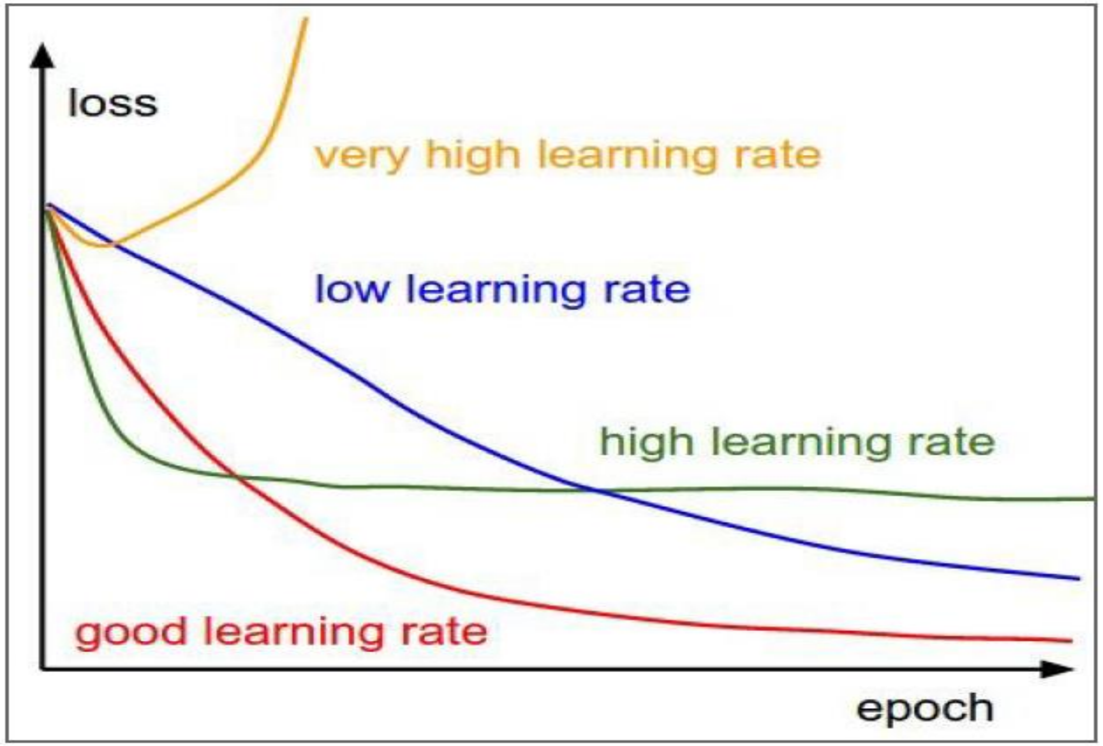
In Practice:

- **Adam** is a good default choice in most cases
    - Try out RADAM, diffGrad and AdaBelief

More Optimizer: **http://ruder.io/optimizing-gradient-descent/**

LEARNING RATE

SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyperparameter.

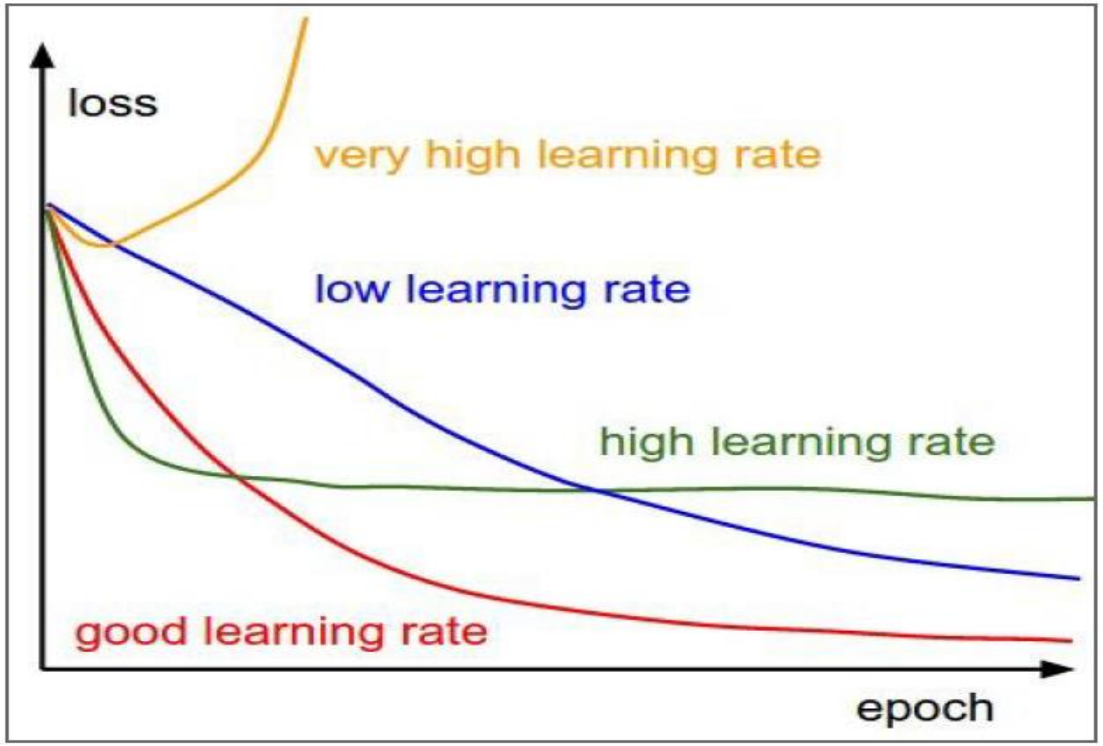


Q: Which one of these learning rates is best to use?

Source: cs231n

LEARNING RATE

SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyperparameter.



=> Learning rate decay over time!

step decay:
e.g. decay learning rate by half every few epochs.

exponential decay:

$$\alpha = \alpha_0 e^{-kt}$$

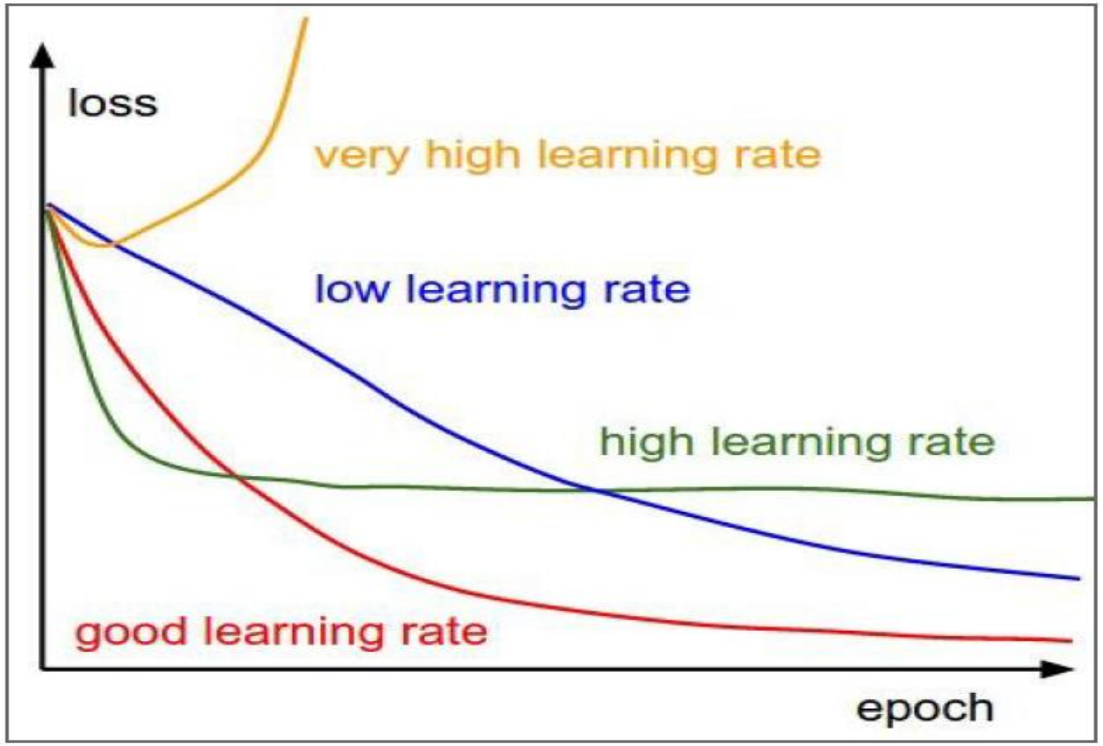
1/t decay:

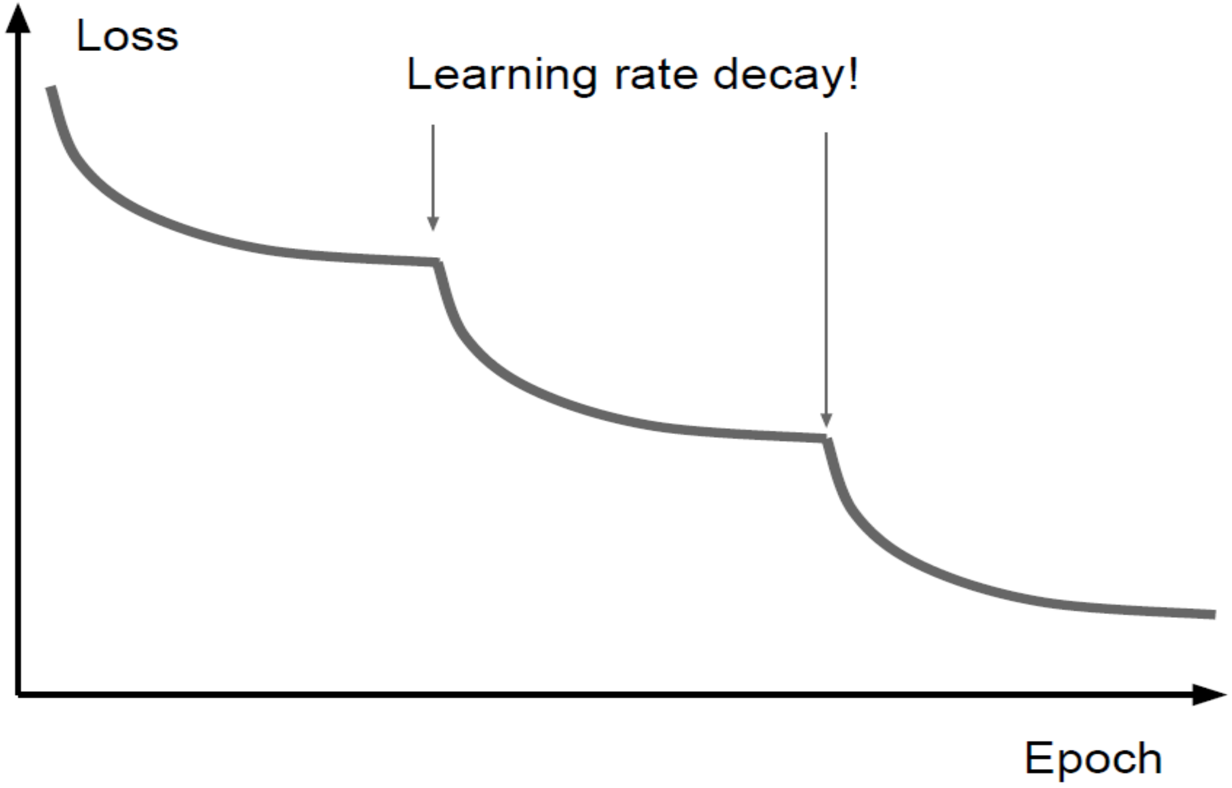
$$\alpha = \alpha_0 / (1 + kt)$$

Source: cs231n

LEARNING RATE

SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyperparameter.





Source: cs231n

LEARNING RATE

In Practice:

- **Learning rate** with step decay is commonly used
    - **Step decay:** reduce rate by a constant factor every few epochs, e.g., by 0.5 every 5 epochs, 0.1 every 20 epochs
    - **Manual:** watch validation error and reduce learning rate whenever it stops improving
        - "Patience" hyperparameter: number of epochs without improvement before reducing learning rate

- **Warmup:** train with a low learning rate for a first few epochs, or linearly increase learning rate before transitioning to normal decay schedule (Goyal et al., 2018)

WHEN TO STOP TRAINING?

- Monitor validation error to decide when to stop
  - "Patience" hyperparameter: number of epochs without improvement before stopping
  - *Early stopping* can be viewed as a kind of regularization

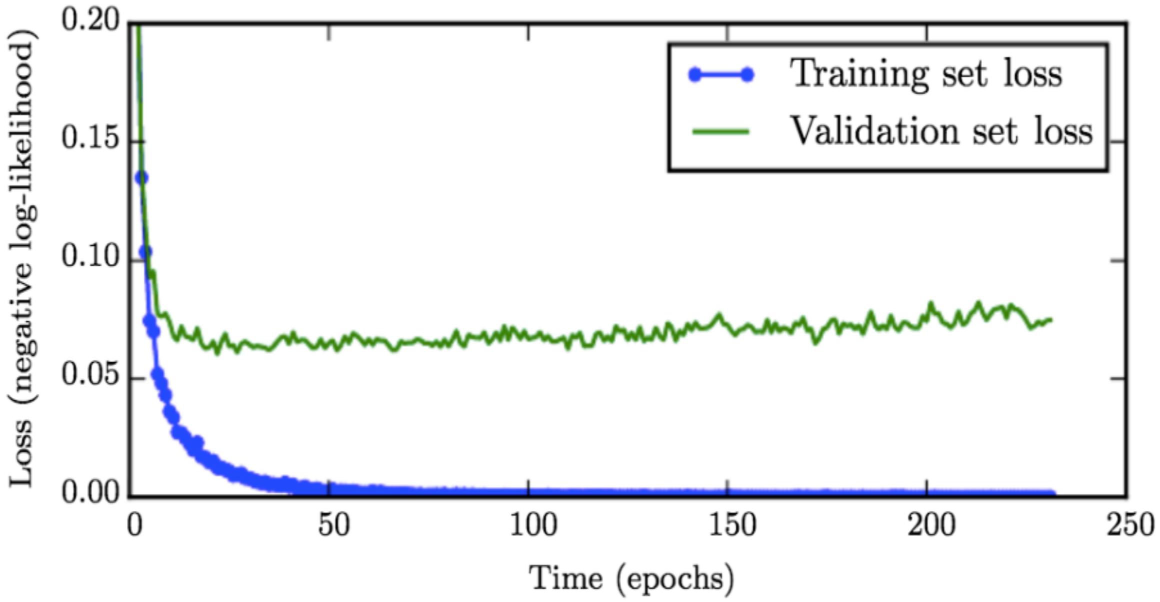
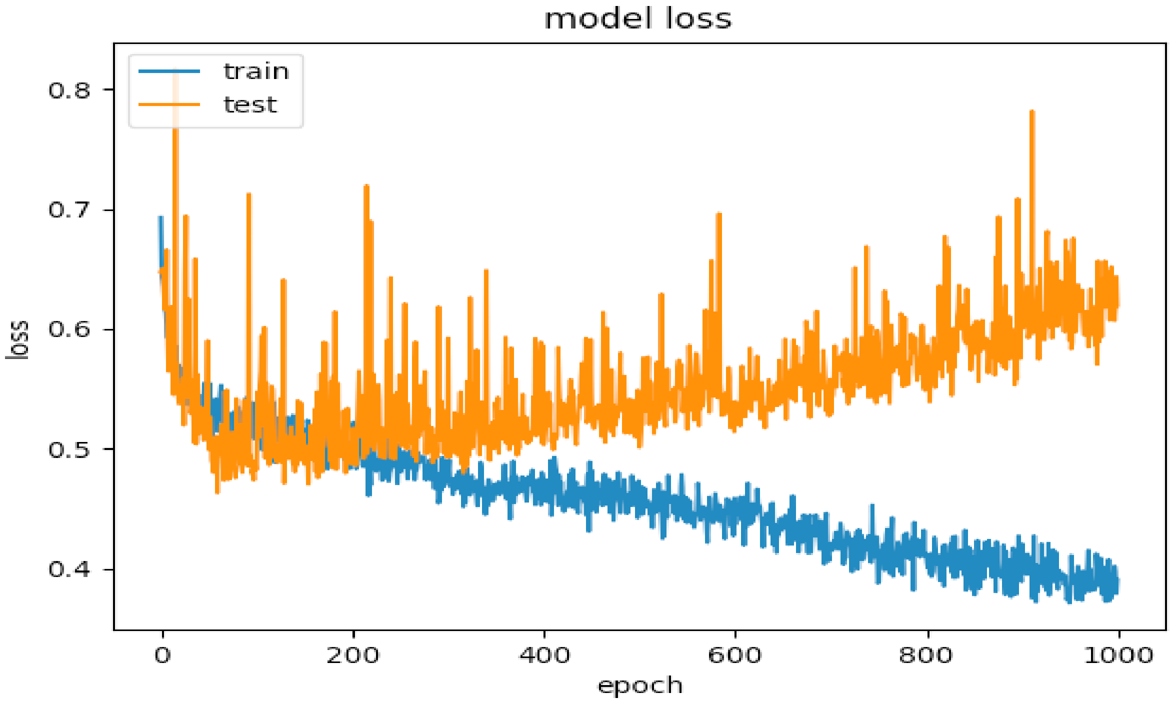


Figure from [Deep Learning Book]()

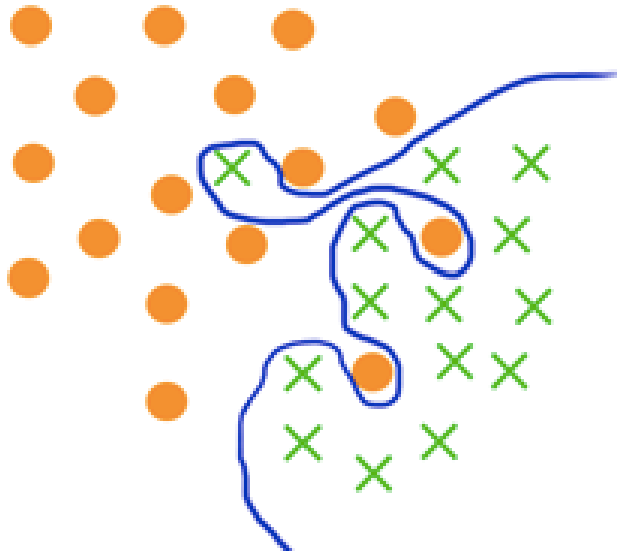


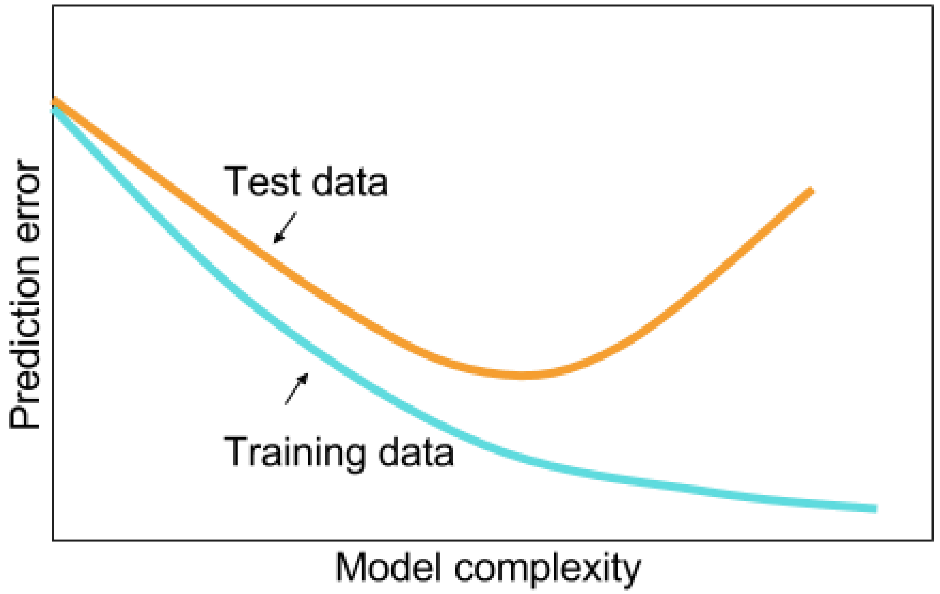
Regularization

Image Source: https://stackoverflow.com/questions/44909134/how-to-avoid-overfitting-on-a-simple-feed-forward-network/44985765

REGULARIZATION

- Techniques for controlling the capacity of a neural network to prevent overfitting





REGULARIZATION

$$L(W) = \underbrace{\frac{1}{N}\sum_{i=1}^{N} L_i(f(x_i, W), y_i)}$$

Data loss: Model predictions should match training data

REGULARIZATION

λ = regularization strength (hyperparameter)

$$L(W) = \underbrace{\frac{1}{N}\sum_{i=1}^{N} L_i(f(x_i, W), y_i)}_{\text{Data loss}} + \underbrace{\lambda R(W)}_{\text{Regularization}}$$

Data loss: Model predictions should match training data

Regularization: Prevent the model from doing *too* well on training data

Source: cs231n

REGULARIZATION

λ = regularization strength (hyperparameter)

$$L(W) = \underbrace{\frac{1}{N}\sum_{i=1}^{N} L_i(f(x_i, W), y_i)}_{} + \underbrace{\lambda R(W)}_{}$$

Data loss: Model predictions should match training data

Regularization: Prevent the model from doing *too* well on training data

Simple examples

<u>L2 regularization:</u> $R(W) = \sum_k \sum_l W_{k,l}^2$

L1 regularization: $R(W) = \sum_k \sum_l |W_{k,l}|$

Elastic net (L1 + L2): $R(W) = \sum_k \sum_l \beta W_{k,l}^2 + |W_{k,l}|$

Source: cs231n

REGULARIZATION

λ = regularization strength (hyperparameter)

$$L(W) = \underbrace{\frac{1}{N}\sum_{i=1}^{N} L_i(f(x_i, W), y_i)}_{\text{Data loss}} + \underbrace{\lambda R(W)}_{\text{Regularization}}$$

Data loss: Model predictions should match training data

Regularization: Prevent the model from doing *too* well on training data

Why regularize?

- Express preferences over weights
- Make the model *simple* so it works on test data
- Improve optimization by adding curvature

Source: cs231n

REGULARIZATION

$$x = [1,1,1,1]$$

$$w_1 = [1,0,0,0]$$

$$w_2 = [0.25,0.25,0.25,0.25]$$

Source: cs231n

REGULARIZATION

$$x = [1,1,1,1]$$

$$w_1 = [1,0,0,0]$$

$$w_2 = [0.25,0.25,0.25,0.25]$$

$$w_1 \,.\, x = w_2 \,.\, x = 1$$

Source: cs231n

REGULARIZATION

$$x = [1,1,1,1]$$

$$w_1 = [1,0,0,0]$$

$$w_2 = [0.25,0.25,0.25,0.25]$$

$$w_1 \, . x = w_2 \, . x = 1$$

Which W to consider?

Source: cs231n

REGULARIZATION

$$x = [1,1,1,1]$$

$$w_1 = [1,0,0,0]$$

$$w_2 = [0.25,0.25,0.25,0.25]$$

$$w_1 \, . x = w_2 \, . x = 1$$

L2 Regularization

$$R(W) = \sum_k \sum_l W_{k,l}^2$$

Source: cs231n

REGULARIZATION

$$x = [1,1,1,1]$$

$$w_1 = [1,0,0,0]$$

$$\boxed{w_2 = [0.25,0.25,0.25,0.25]}$$

$$w_1 \, . x = w_2 \, . x = 1$$

L2 regularization likes to "spread out" the weights

L2 Regularization

$$R(W) = \sum_k \sum_l W_{k,l}^2$$

Source: cs231n

OTHER TYPES OF REGULARIZATION

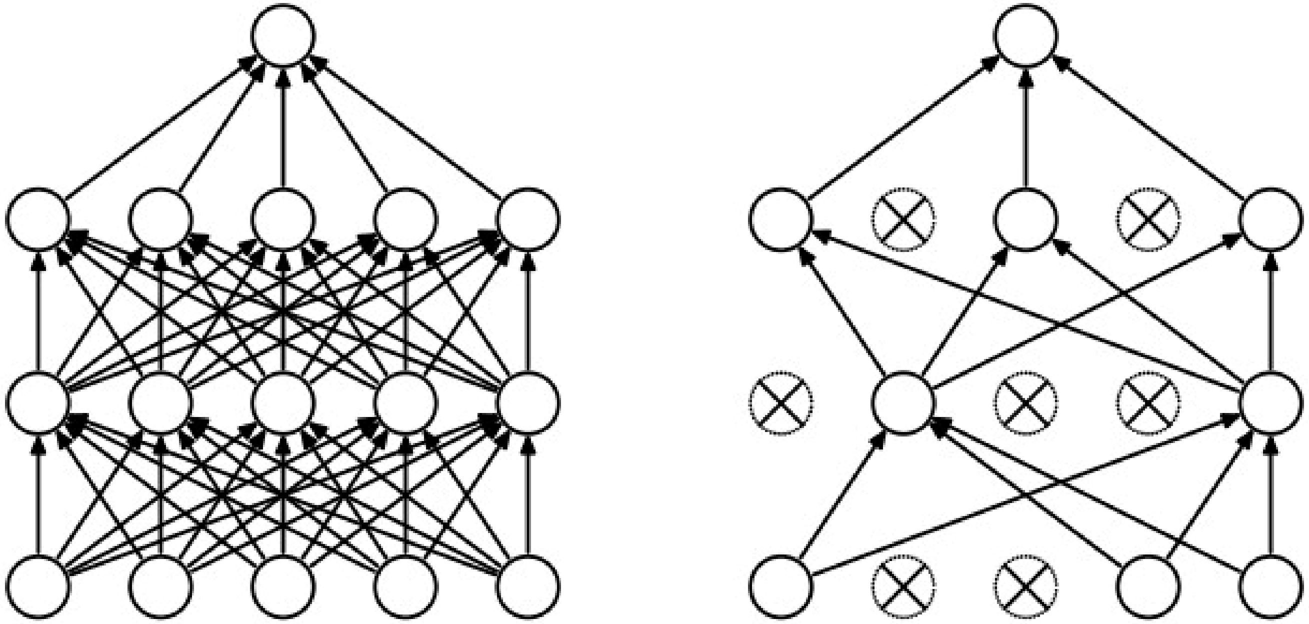
- Dropout
- Batch Normalization
- Data Augmentation
    - Adding noise to the inputs
    - Recall motivation of max margin criterion

Dropout

DROPOUT

In each forward pass, randomly set some neurons to zero

Probability of dropping is a hyperparameter; 0.5 is common

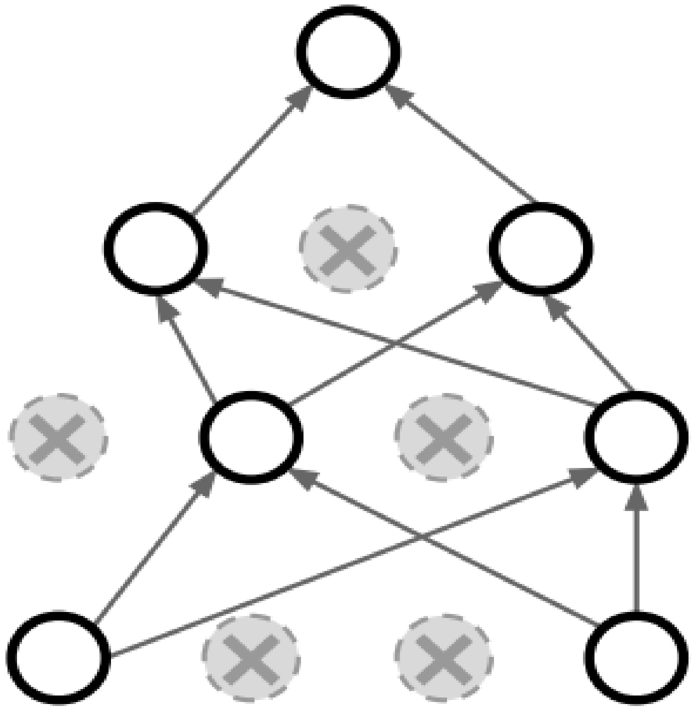


(a) Standard Neural Net   (b) After applying dropout.

Dropout: A simple way to prevent neural networks from overfitting [Srivastava JMLR 2014]

DROPOUT

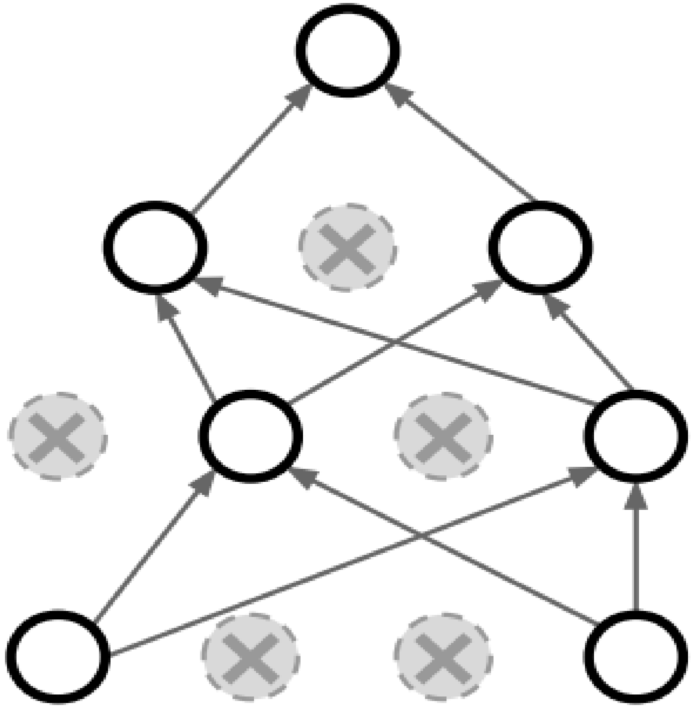
How can this possibly be a good idea?



Source: cs231n

DROPOUT

How can this possibly be a good idea?

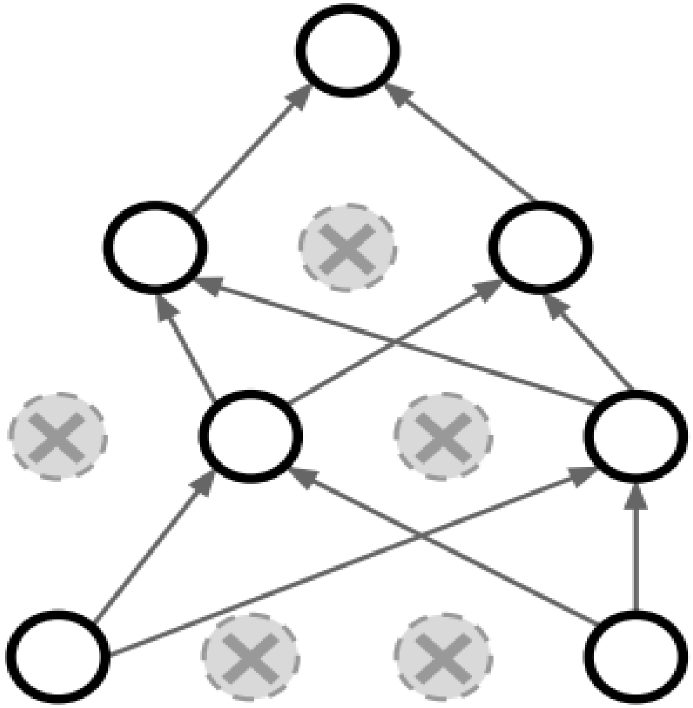


- Intuitions
  - Prevent "co-adaptation" of units, increase robustness to noise
  - Train *implicit ensemble*

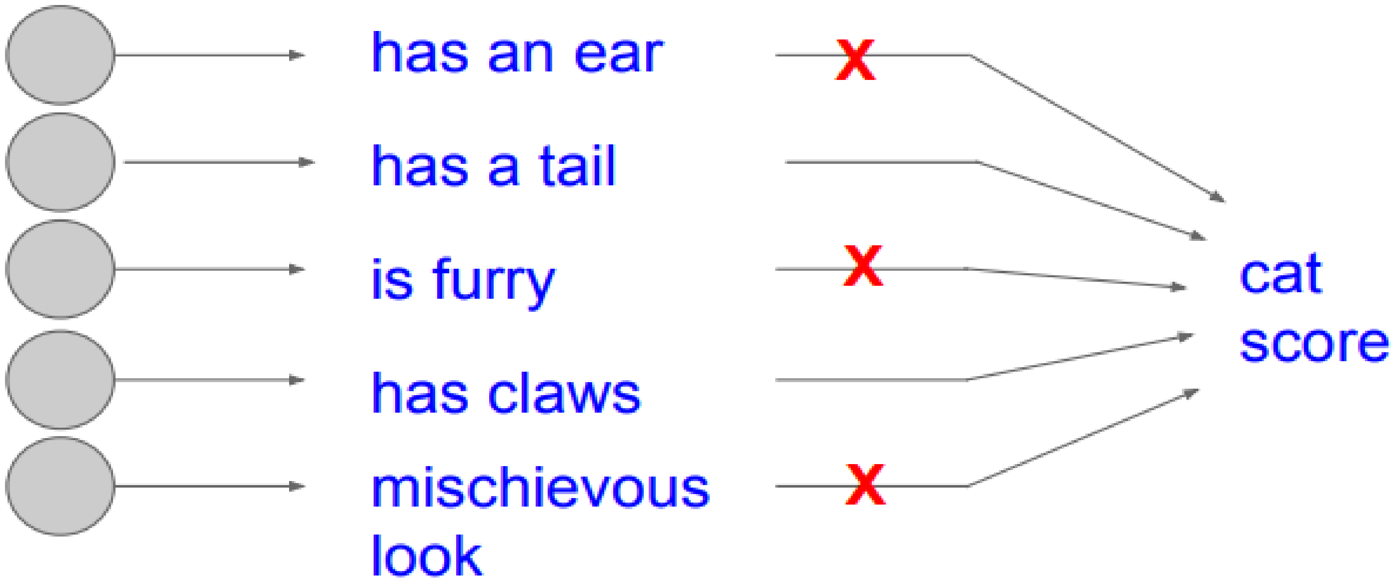
Source: cs231n

DROPOUT

How can this possibly be a good idea?



Forces the network to have a redundant representation;
Prevents co-adaptation of features



Source: cs231n

DROPOUT: TEST TIME

```python
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
```

At test time all neurons are active always
=> We must scale the activations so that for each neuron:
output at test time = expected output at training time

More common: "Inverted dropout"

Source: cs231n

DROPOUT: MORE COMMON: “INVERTED DROPOUT”

We drop and scale at train time and don't do anything at test time.

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

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

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

def predict(X):
  # ensembled forward pass
  H1 = np.maximum(0, np.dot(W1, X) + b1) # no scaling necessary
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  out = np.dot(W3, H2) + b3
```

test time is unchanged!

Source: cs231n

Batch Normalization

BATCH NORMALIZATION

"We want zero-mean unit-variance activations? lets make them so."

Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift [Ioffe and Szegedy 2015]

Source: cs231n

BATCH NORMALIZATION

"We want zero-mean unit-variance activations? lets make them so."

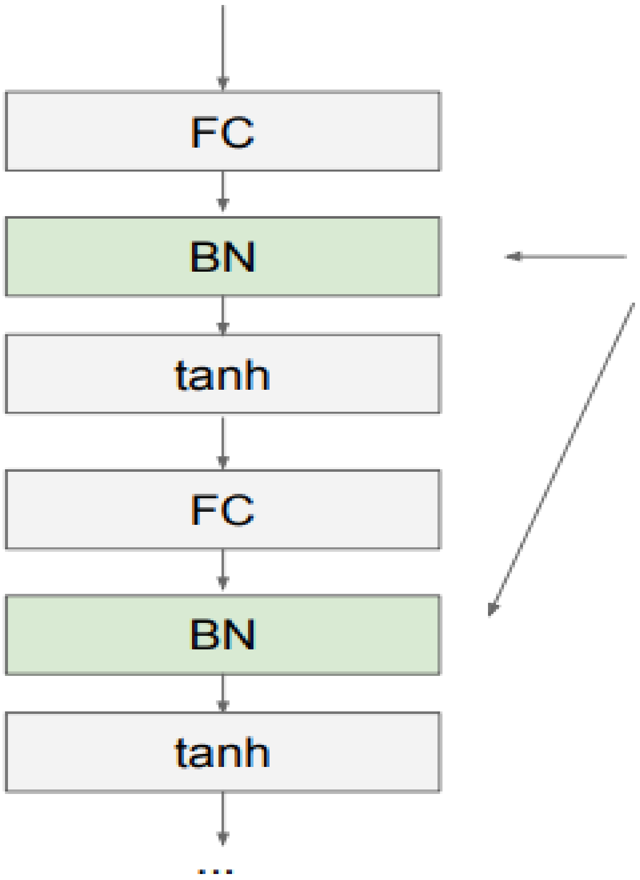
consider a batch of activations at some layer. To make each dimension zero-mean unit-variance, apply:

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift [Ioffe and Szegedy 2015]

Source: cs231n

BATCH NORMALIZATION

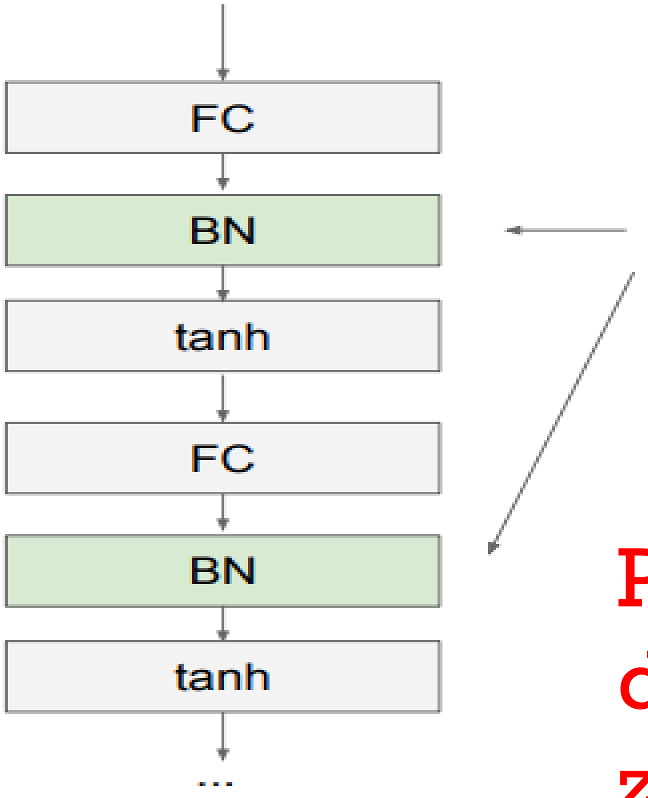


Usually inserted after Fully Connected or Convolutional layers, and before nonlinearity.

Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift [Ioffe and Szegedy 2015]

Source: cs231n

BATCH NORMALIZATION



Usually inserted after Fully Connected or Convolutional layers, and before nonlinearity.

Problem:
do we necessarily want a zero-mean unit-variance input?

Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift [Ioffe and Szegedy 2015]

Source: cs231n

BATCH NORMALIZATION

Normalize:

$$\boxed{\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}}$$

And then allow the network to squash
the range if it wants to:

$$\boxed{y^{(k)} = \gamma^{(k)}\widehat{x}^{(k)} + \beta^{(k)}}$$

Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift [Ioffe and Szegedy 2015]

Source: cs231n

BATCH NORMALIZATION

Normalize:

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

And then allow the network to squash
the range if it wants to:

$$y^{(k)} = \gamma^{(k)} \widehat{x}^{(k)} + \beta^{(k)}$$

Note, the network can learn:

$$\gamma^{(k)} = \sqrt{\mathrm{Var}[x^{(k)}]}$$

$$\beta^{(k)} = \mathrm{E}[x^{(k)}]$$

to recover the identity mapping.

Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift [Ioffe and Szegedy 2015]

Source: cs231n

BATCH NORMALIZATION

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Batch Normalization: Accelerating Deep Network Training by
Reducing Internal Covariate Shift [Ioffe and Szegedy 2015]

Source: cs231n

BATCH NORMALIZATION

Note: at test time BatchNorm layer functions differently:

The mean/std are not computed based on the batch.

Instead, a single fixed empirical mean of activations during training is used.

(e.g. can be estimated during training with running averages)

Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift [Ioffe and Szegedy 2015]

Source: cs231n

BATCH NORMALIZATION

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \text{BN}_{\gamma,\beta}(x_i)\}$

At test time (usually):

$$\cancel{\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i}$$ // ~~mini-batch~~ mean
training set

$$\cancel{\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2}$$ // ~~mini-batch~~ variance
training set

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}$$ // normalize

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \text{BN}_{\gamma,\beta}(x_i)$$ // scale and shift

Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift [Ioffe and Szegedy 2015]

Source: cs231n

BATCH NORMALIZATION

Benefits

- Improves gradient flow through the network
- Allows higher learning rates and Accelerates convergence of training
- Reduces the strong dependence on initialization
- Acts as a form of regularization

Pitfalls

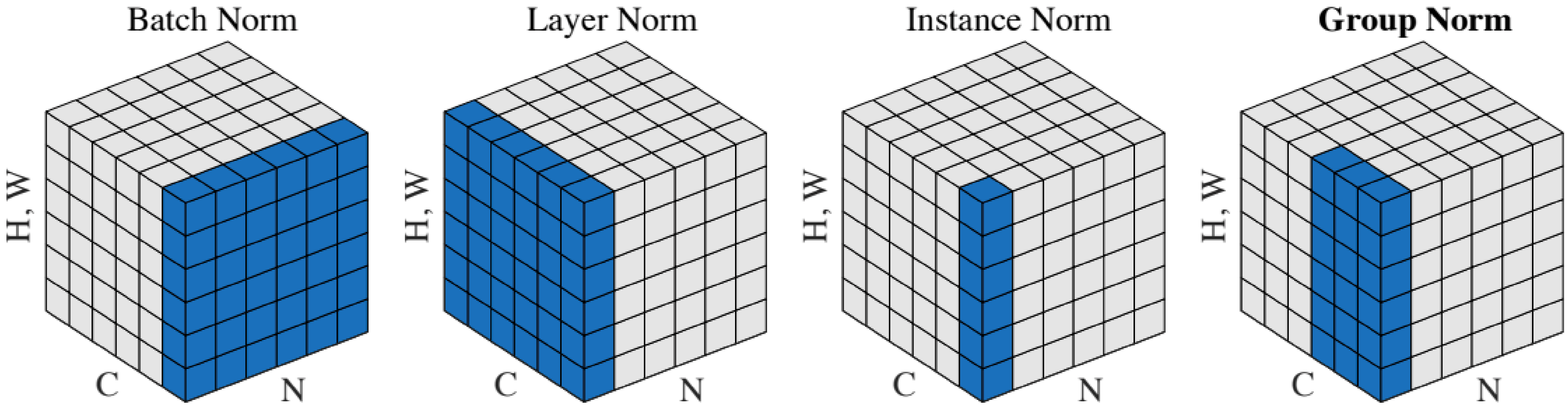
- Behavior depends on composition of mini-batches, can lead to hard-to-catch bugs if there is a mismatch between training and test regime (example)
- Doesn't work well for small mini-batch sizes
- Cannot be used in recurrent models

Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift [Ioffe and Szegedy 2015]

Source: cs231n

OTHER TYPES OF NORMALIZATION

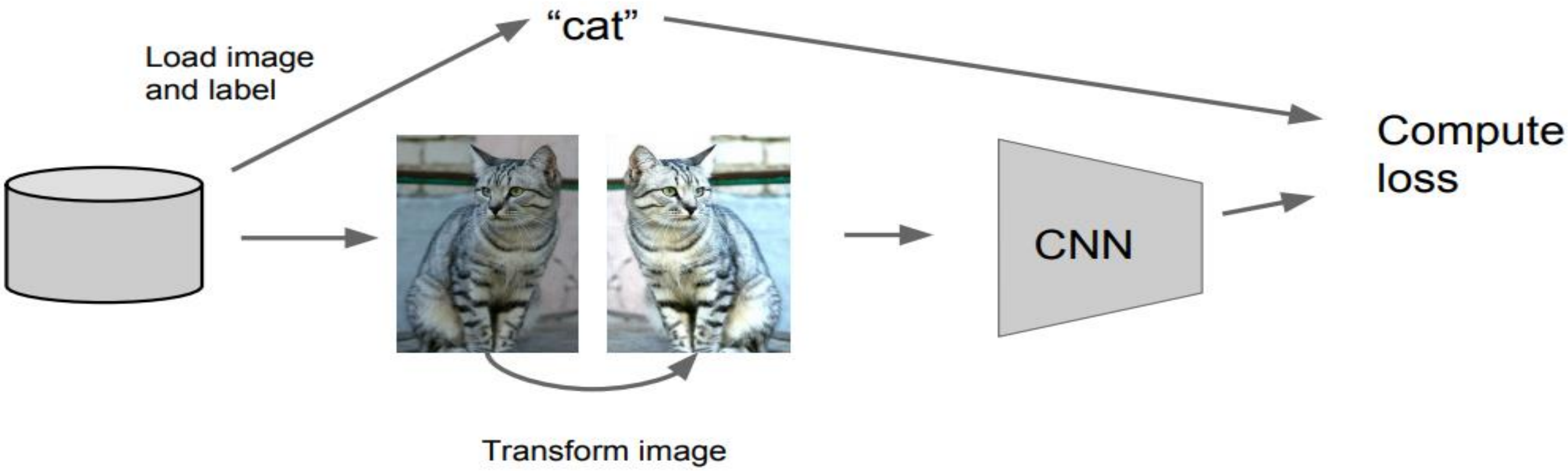
- Layer normalization (Ba et al., 2016)
- Instance normalization (Ulyanov et al., 2017)
- Group normalization (Wu and He, 2018)
- Weight normalization (Salimans et al., 2016)



Y. Wu and K. He, Group Normalization, ECCV 2018

Data Augmentation

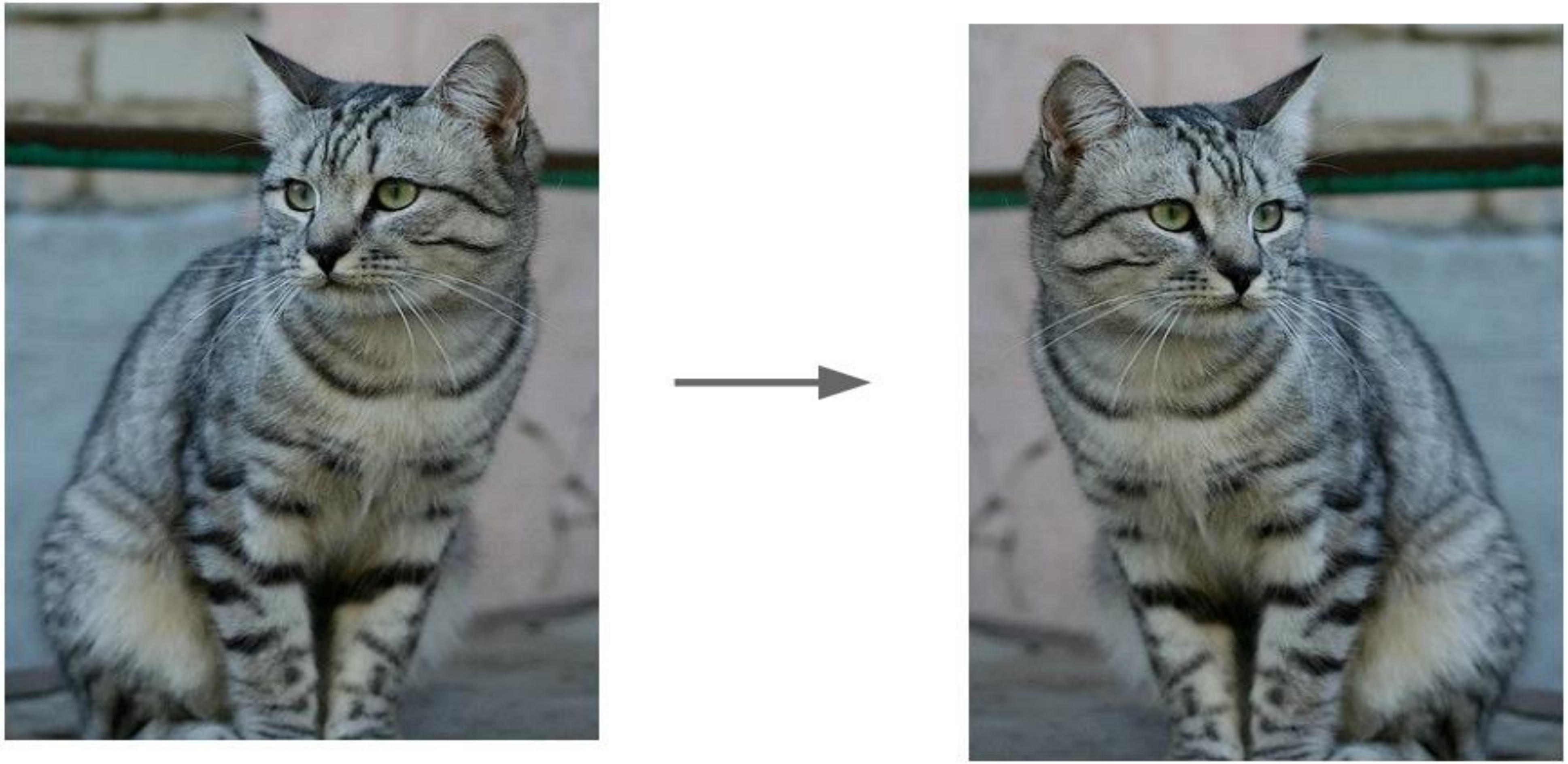
DATA AUGMENTATION (JITTERING)



Source: cs231n

DATA AUGMENTATION (JITTERING)

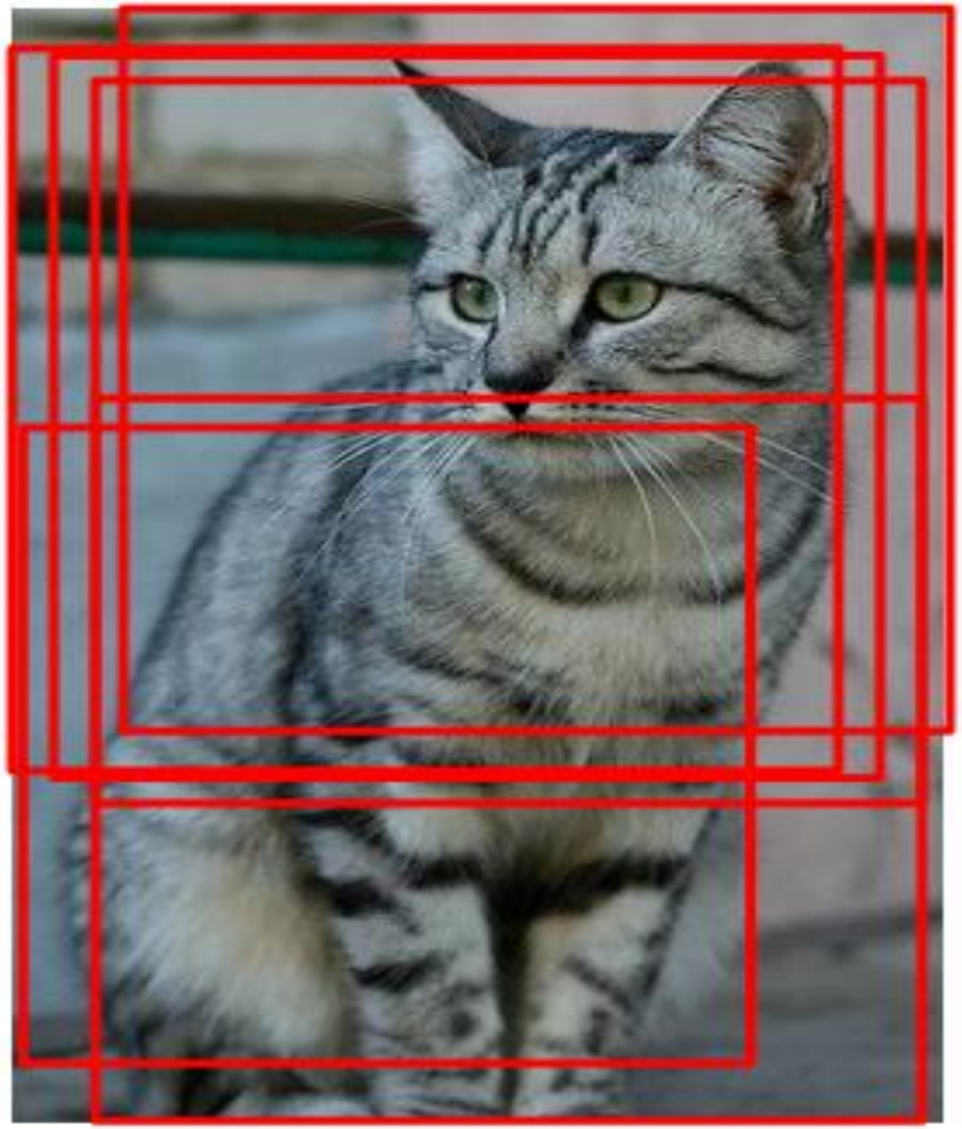
Horizontal Flips



Source: cs231n

DATA AUGMENTATION (JITTERING)

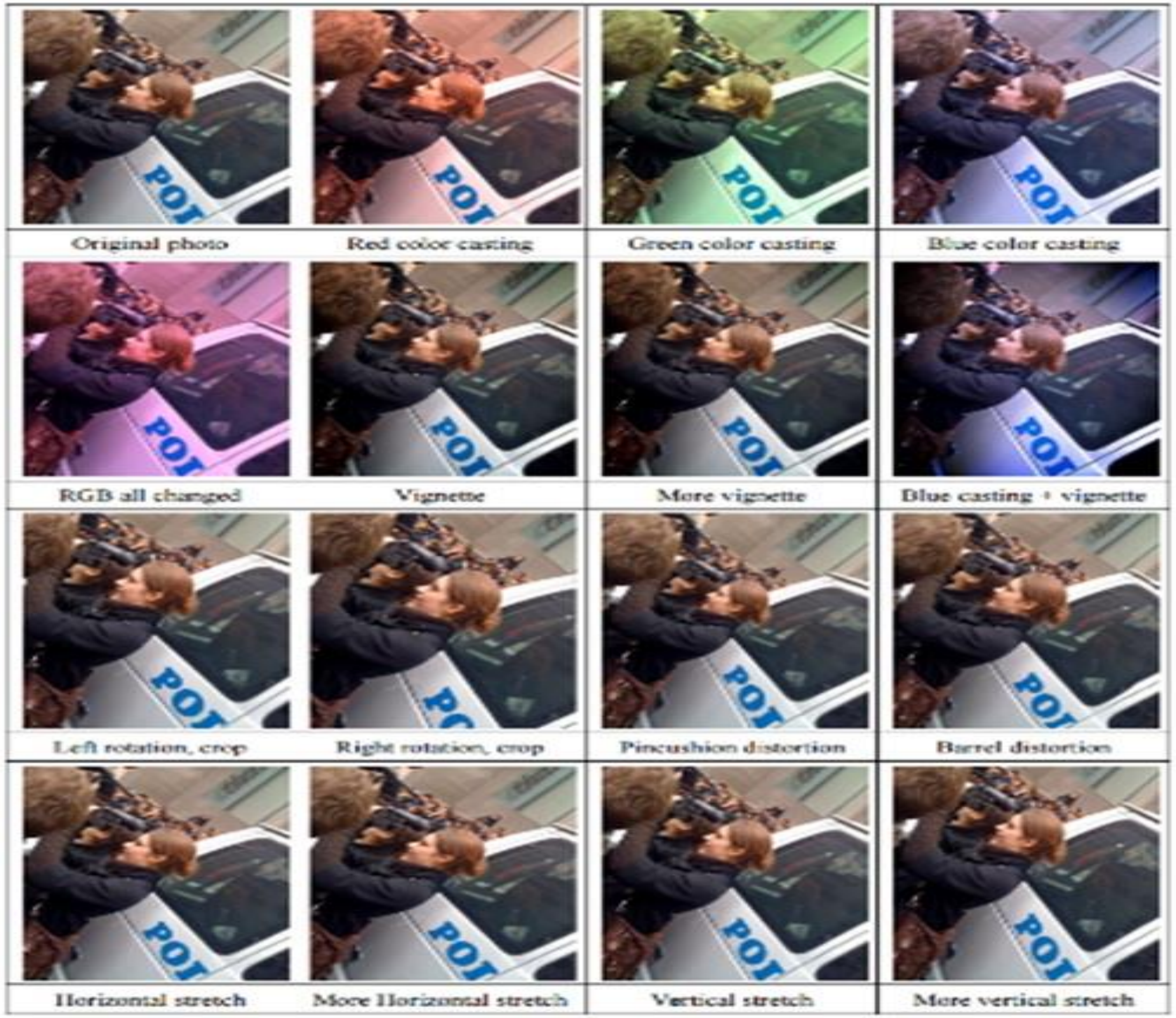
Random crops and scales



Source: cs231n

DATA AUGMENTATION (JITTERING)

- Create *virtual* training samples
- Get creative for your problem!
  - Horizontal flip
  - Random crop
  - Color casting
  - Randomize contrast
  - Randomize brightness
  - Geometric distortion
  - Rotation
  - Photometric changes

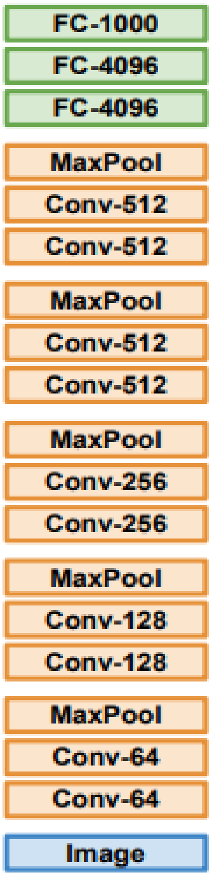


Deep Image [Wu et al. 2015]

Transfer Learning

TRANSFER LEARNING WITH CNNS

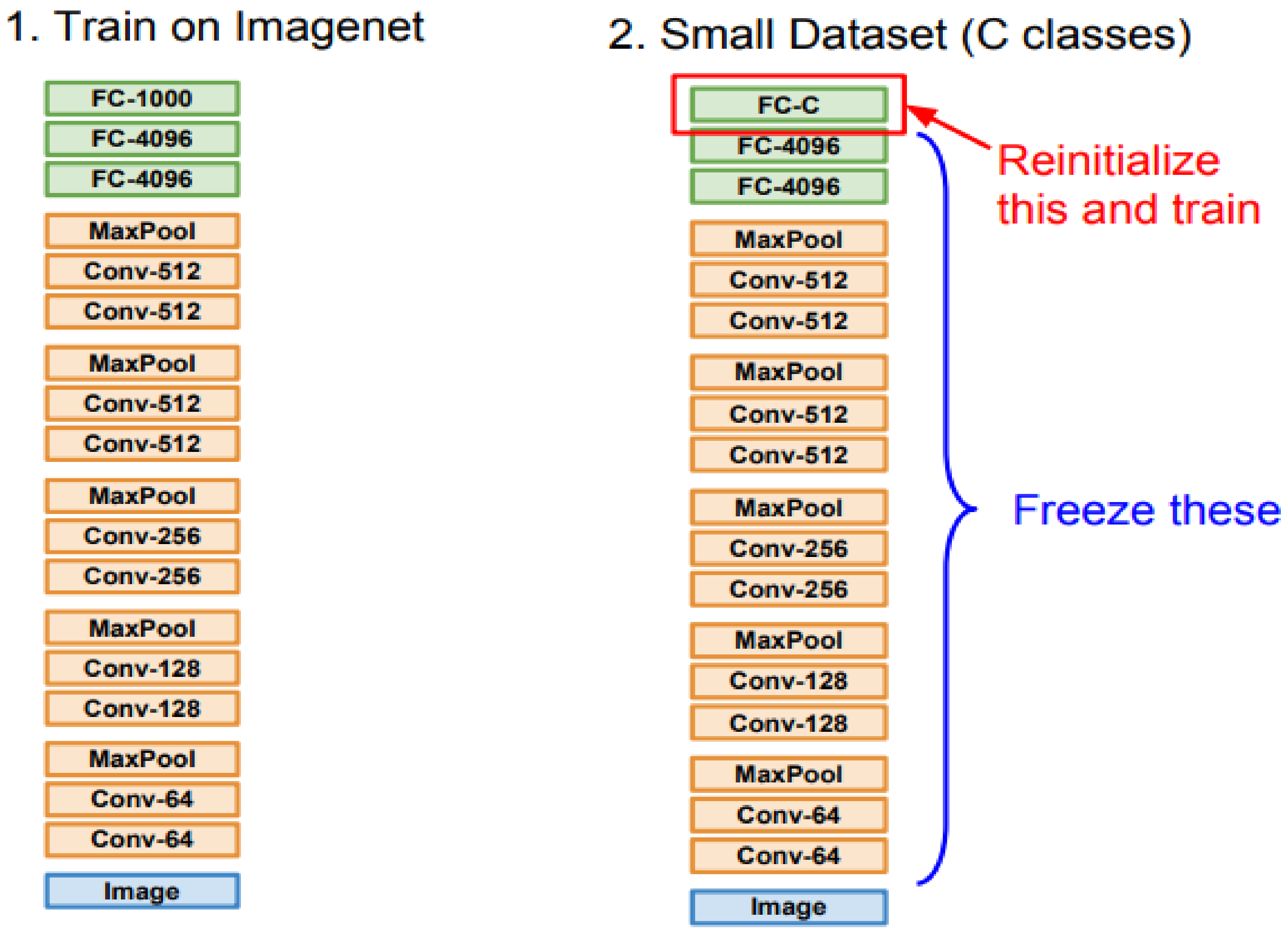
1. Train on Imagenet



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

Source: cs231n

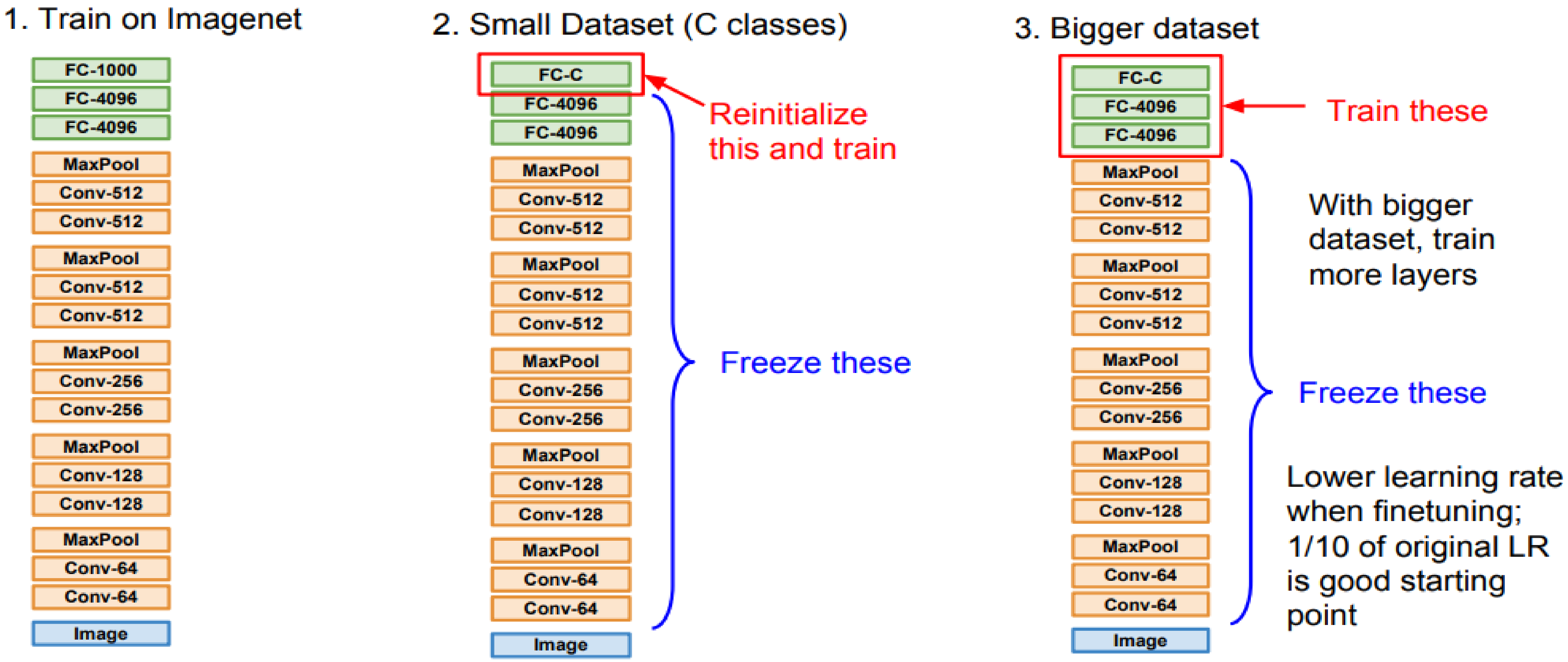
TRANSFER LEARNING WITH CNNS



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

Source: cs231n

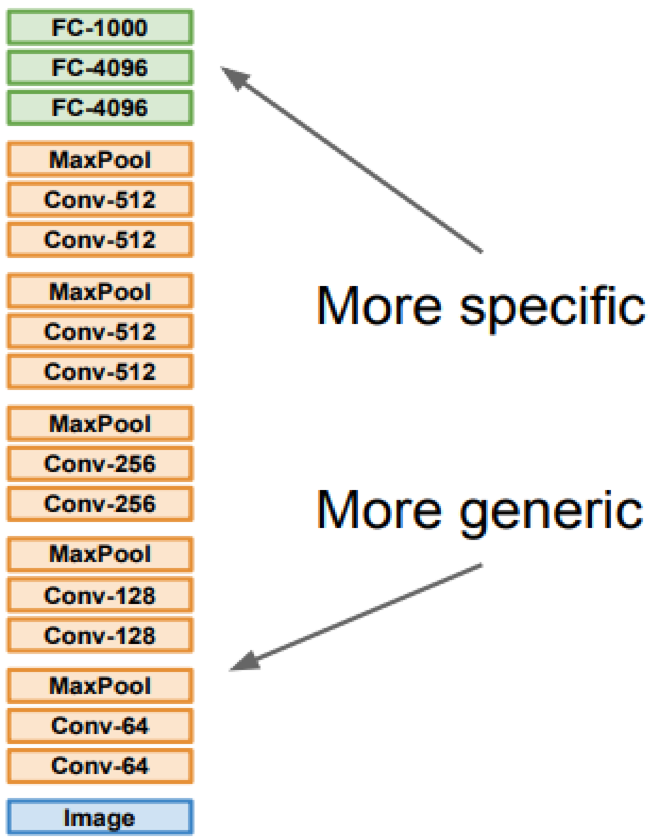
TRANSFER LEARNING WITH CNNS



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

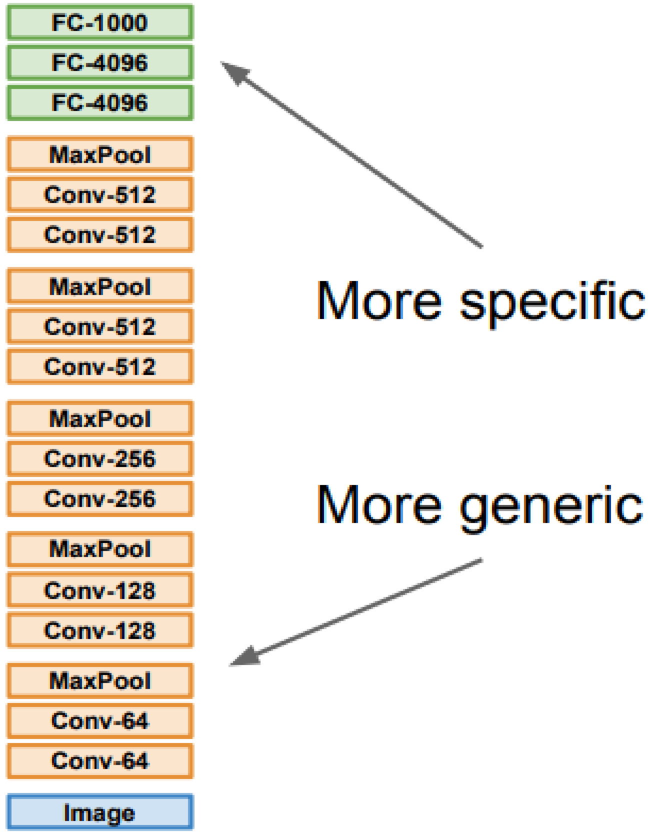
Source: cs231n

TRANSFER LEARNING WITH CNNS



Source: cs231n

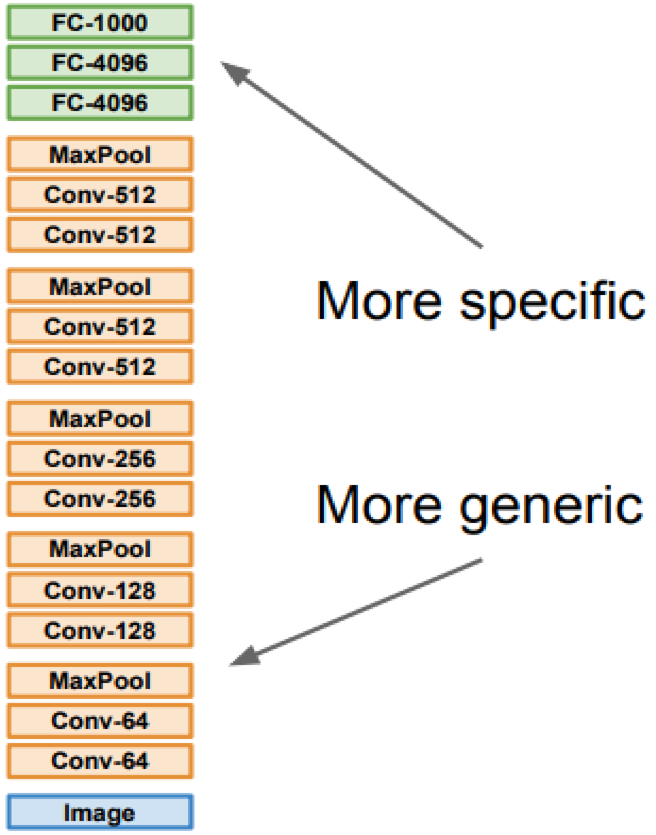
TRANSFER LEARNING WITH CNNS



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

Source: cs231n

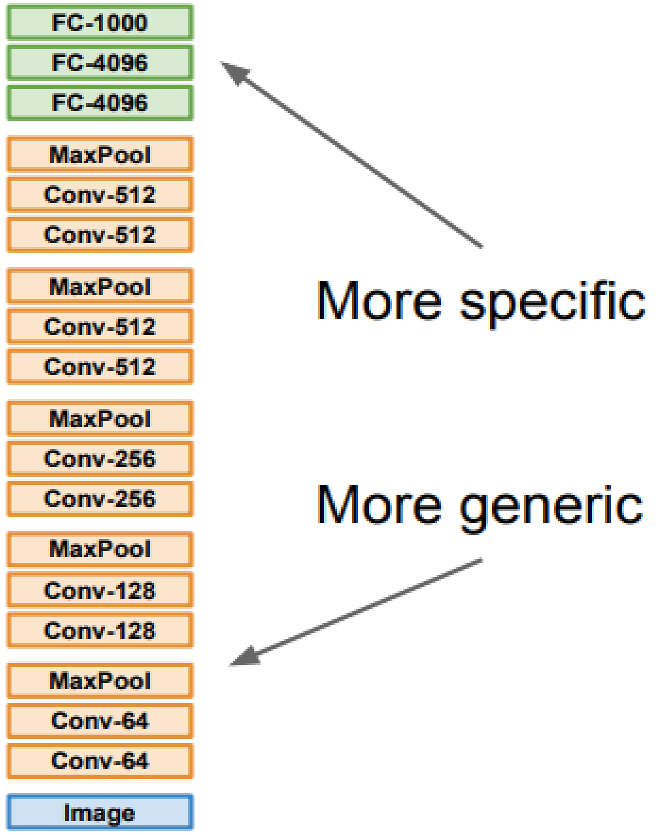
TRANSFER LEARNING WITH CNNS



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

Source: cs231n

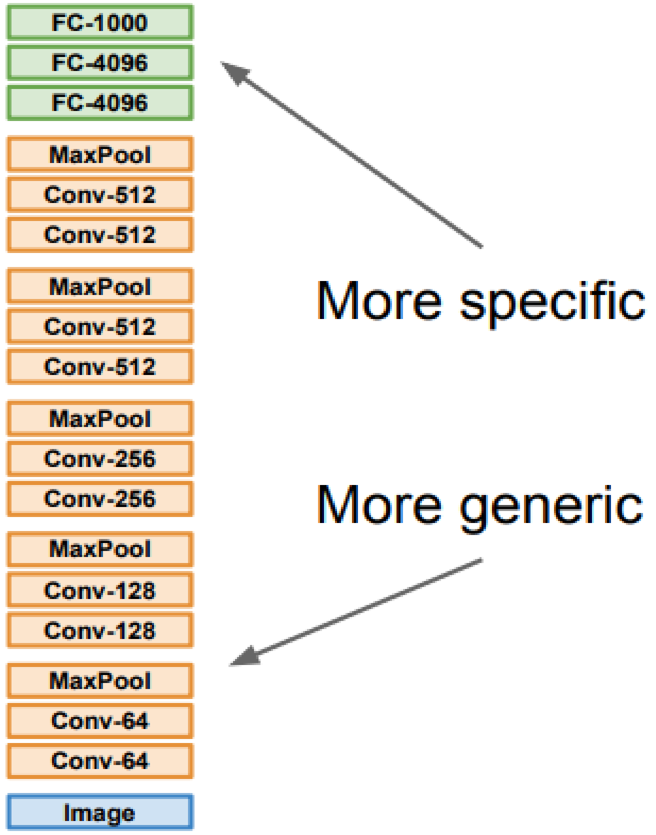
TRANSFER LEARNING WITH CNNS



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

Source: cs231n

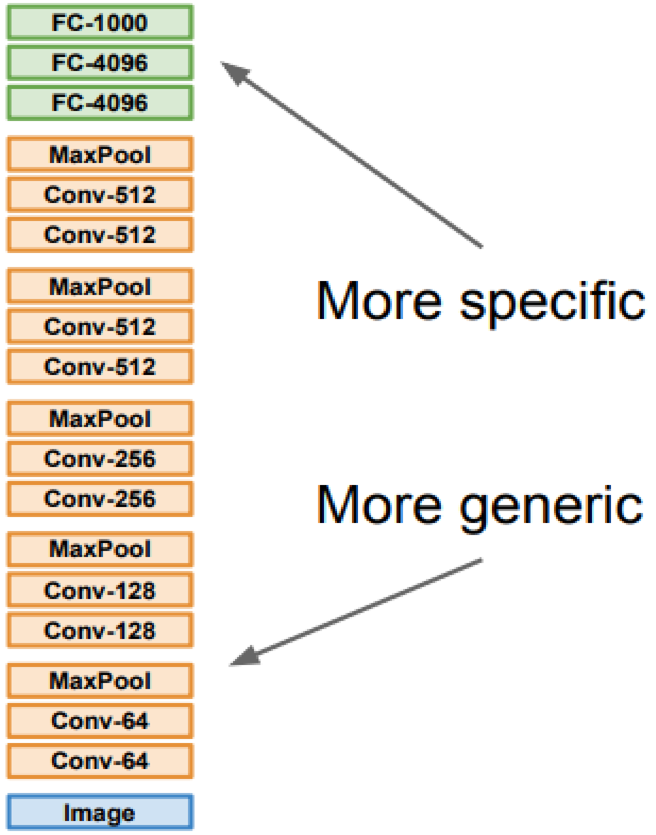
TRANSFER LEARNING WITH CNNS



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

Source: cs231n

TRANSFER LEARNING WITH CNNS



	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

Source: cs231n

TRANSFER LEARNING WITH CNNS

Takeaway for your projects and beyond:

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

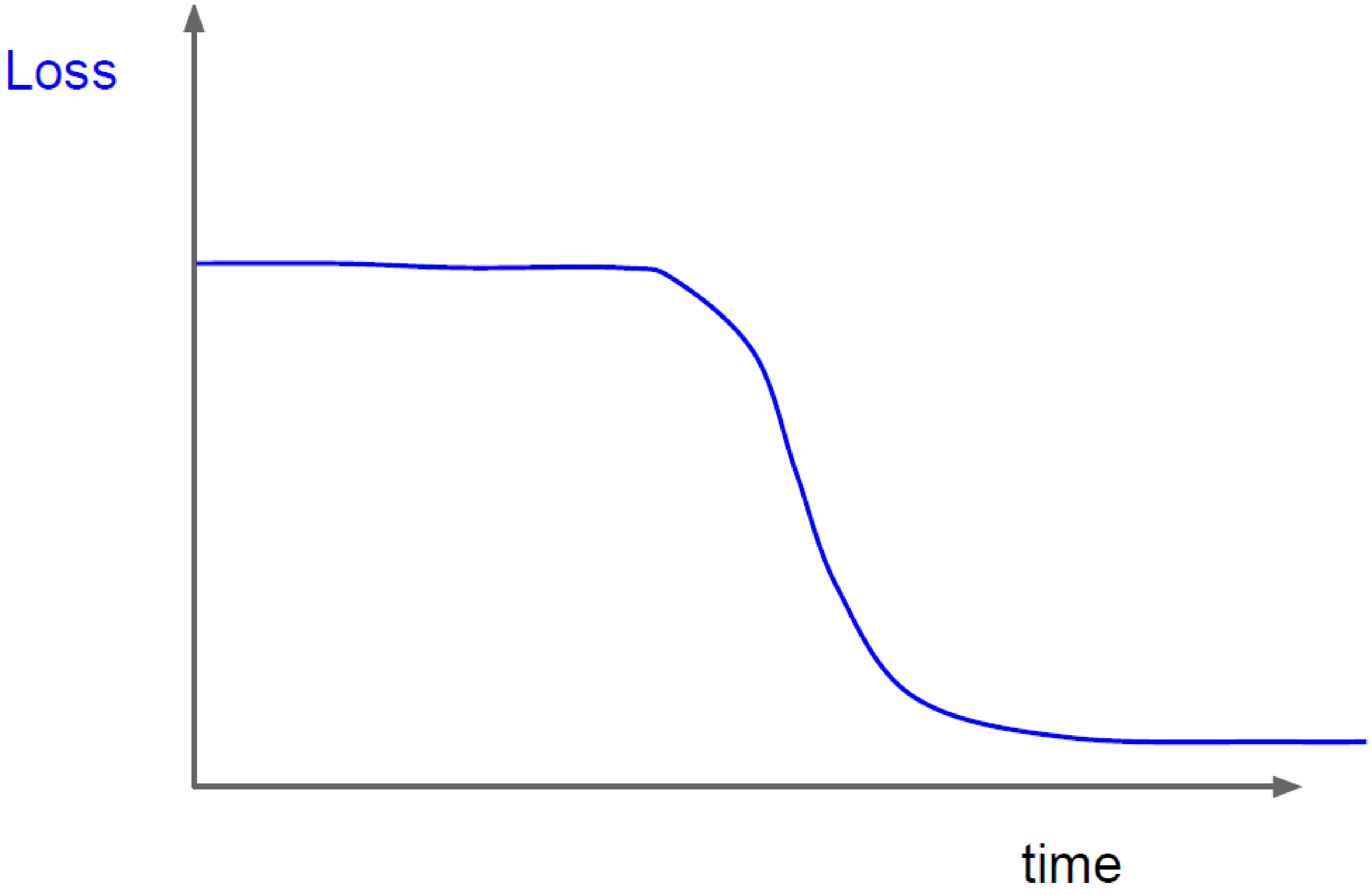
Deep learning frameworks provide a "Model Zoo" of pretrained models so you don't need to train your own

Caffe: https://github.com/BVLC/caffe/wiki/Model-Zoo
TensorFlow: https://github.com/tensorflow/models
PyTorch: https://github.com/pytorch/vision
Matconvnet: http://www.vlfeat.org/matconvnet/pretrained/

Source: cs231n

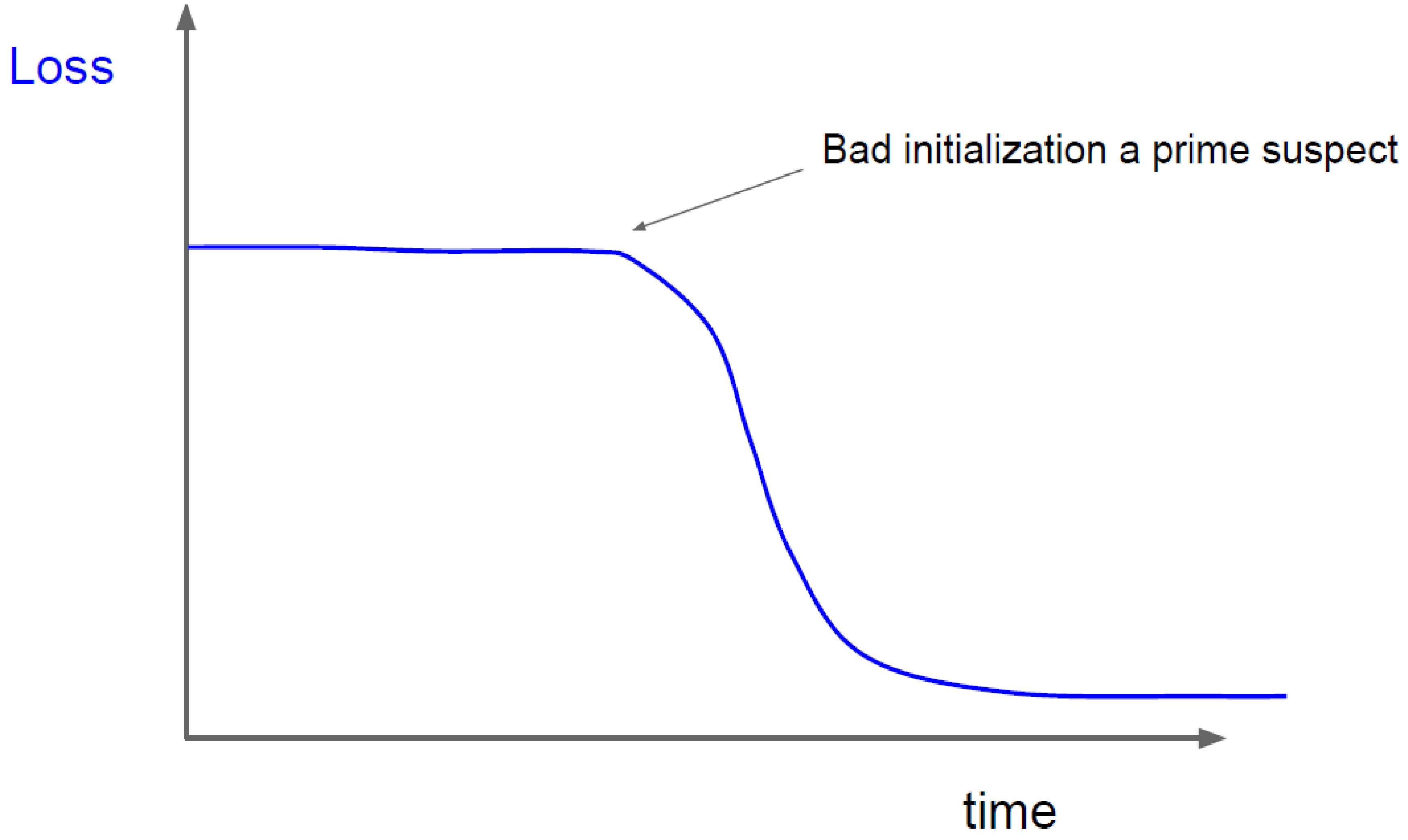
MONITOR AND VISUALIZE THE LOSS CURVE

MONITOR AND VISUALIZE THE LOSS CURVE



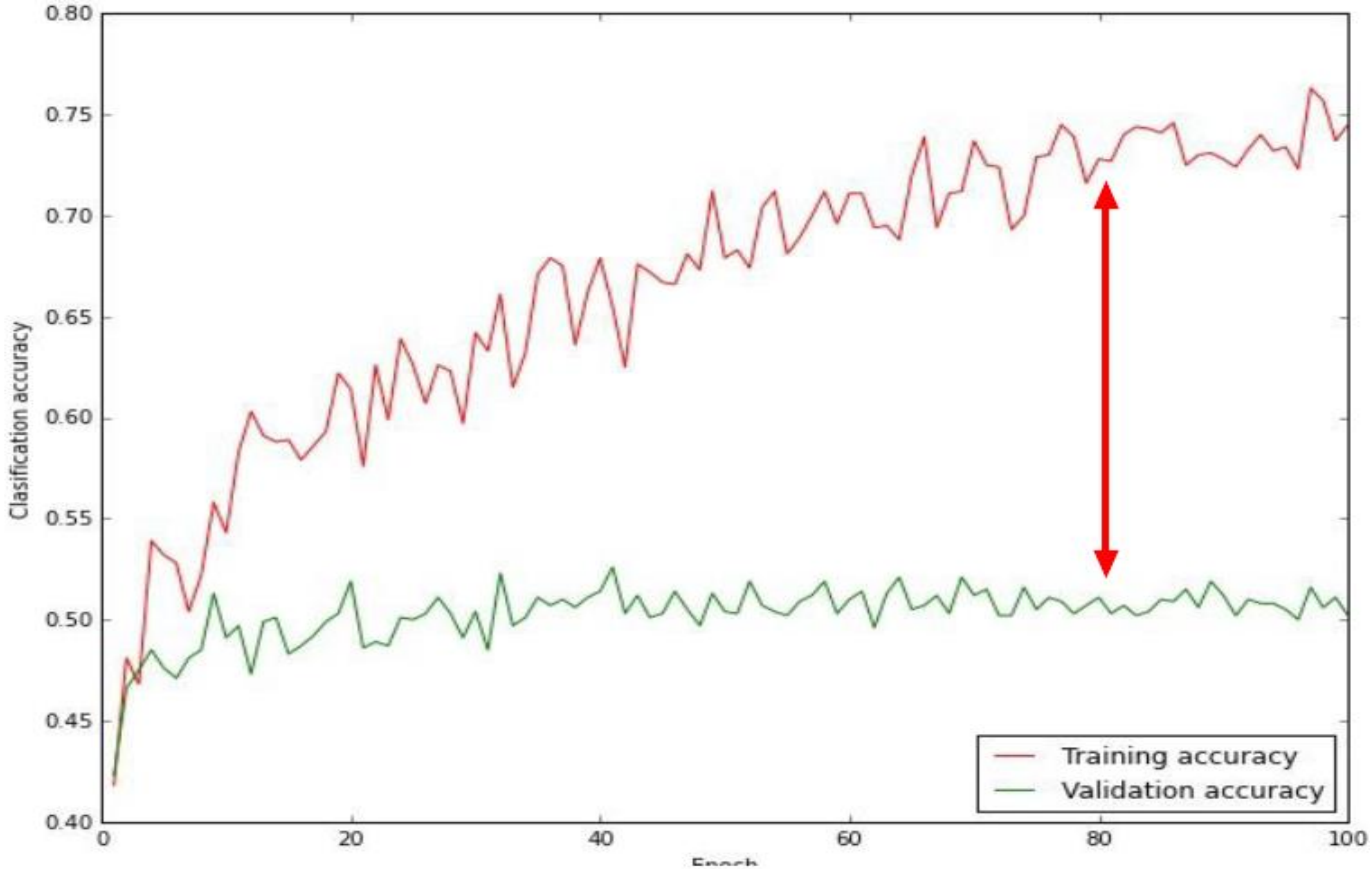
Source: cs231n

MONITOR AND VISUALIZE THE LOSS CURVE



Source: cs231n

MONITOR AND VISUALIZE THE LOSS CURVE

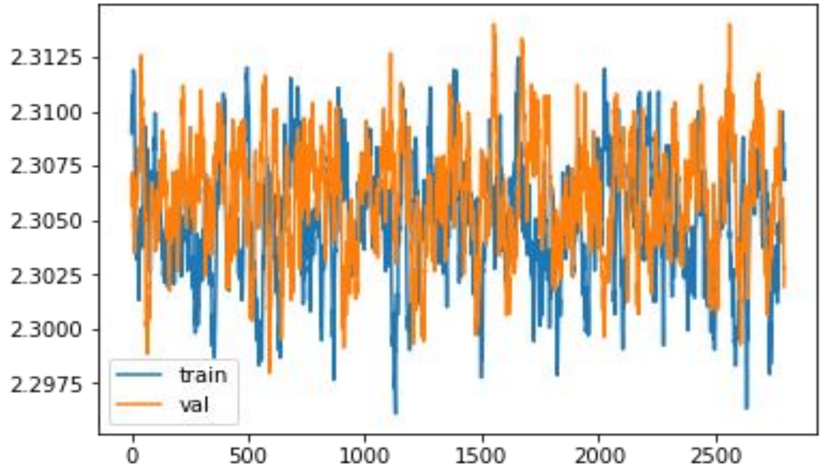


big gap = overfitting
=> increase
regularization strength?

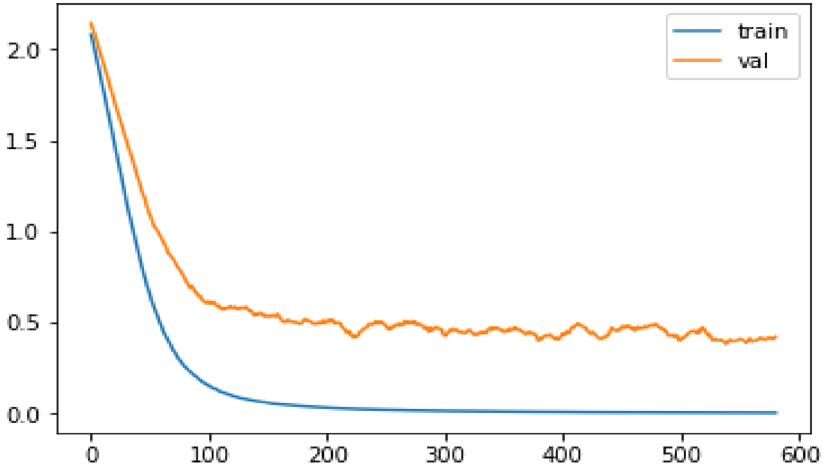
no gap
=> increase model
capacity?

Source: cs231n

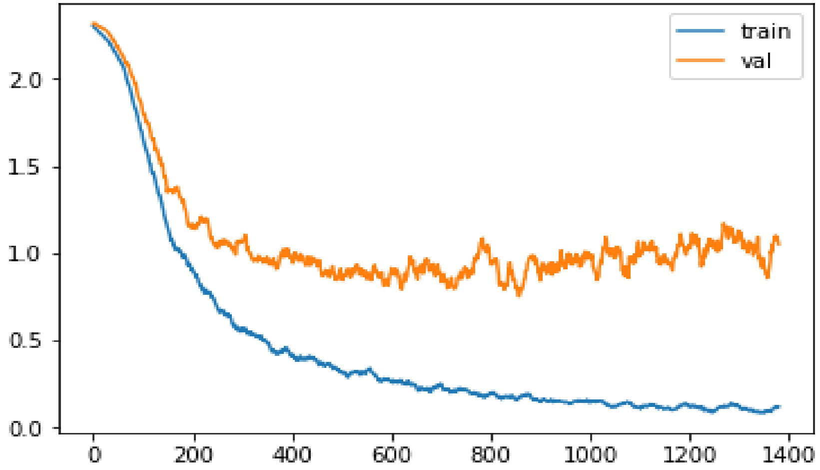
MONITOR AND VISUALIZE THE LOSS CURVE



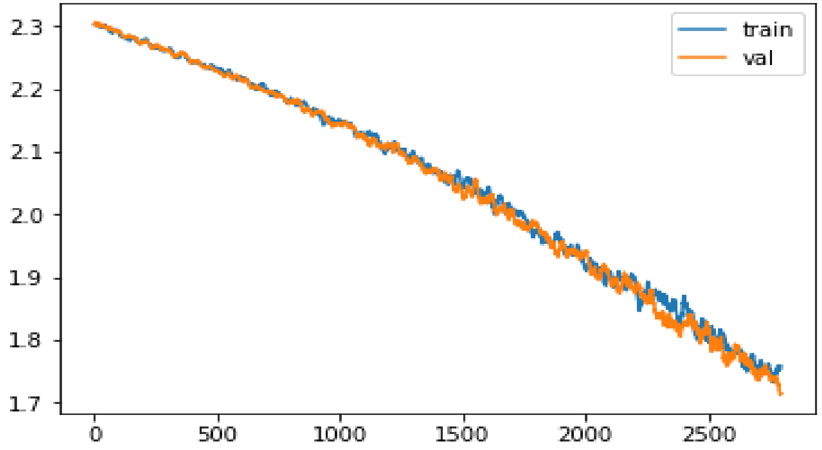
Not learning: gradients not applied to weights



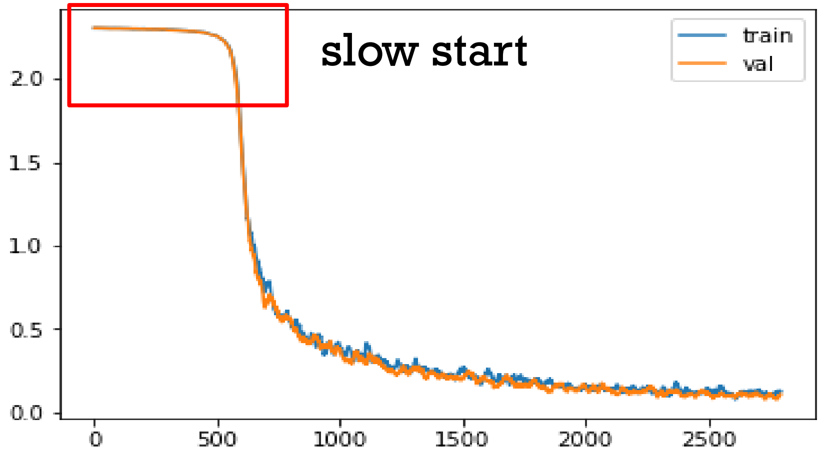
Overfit: model too large/dataset too small



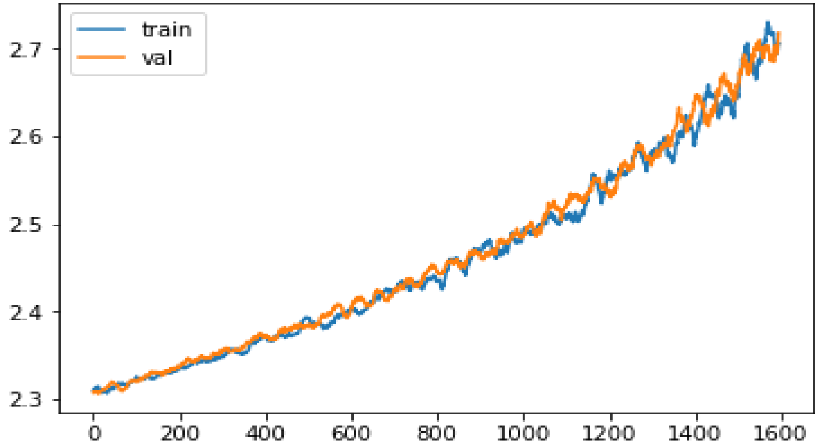
More extreme case of overfitting



Not converged yet: need longer training



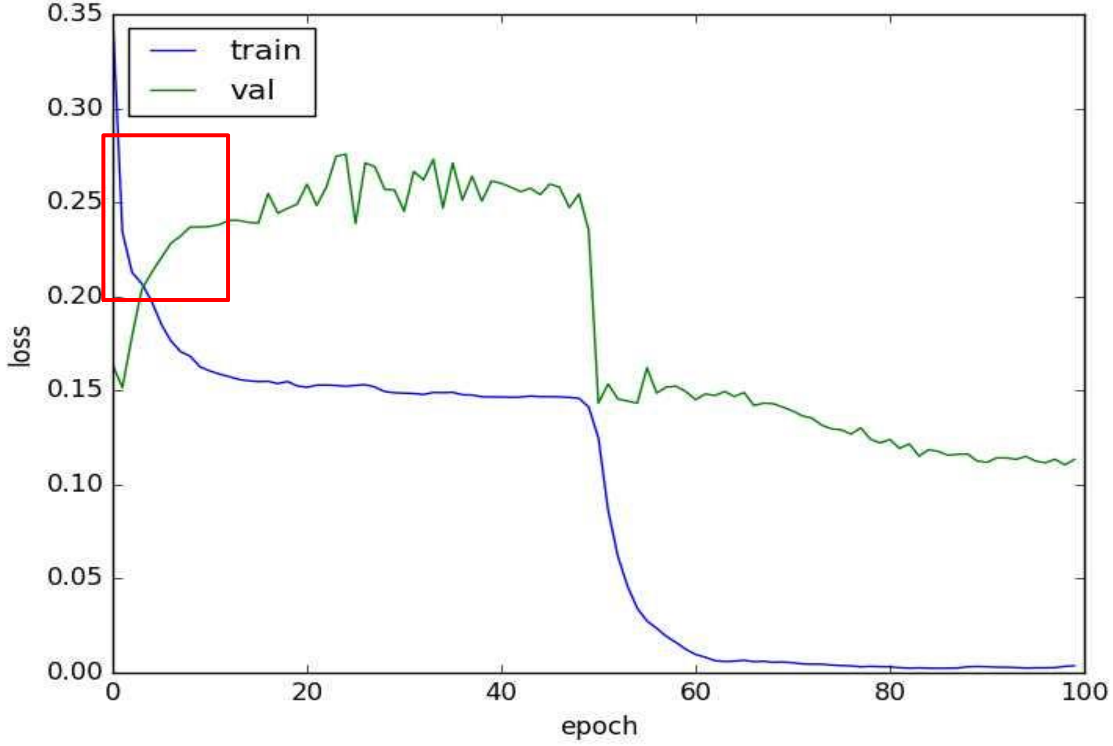
Slow start: initialization weights too small



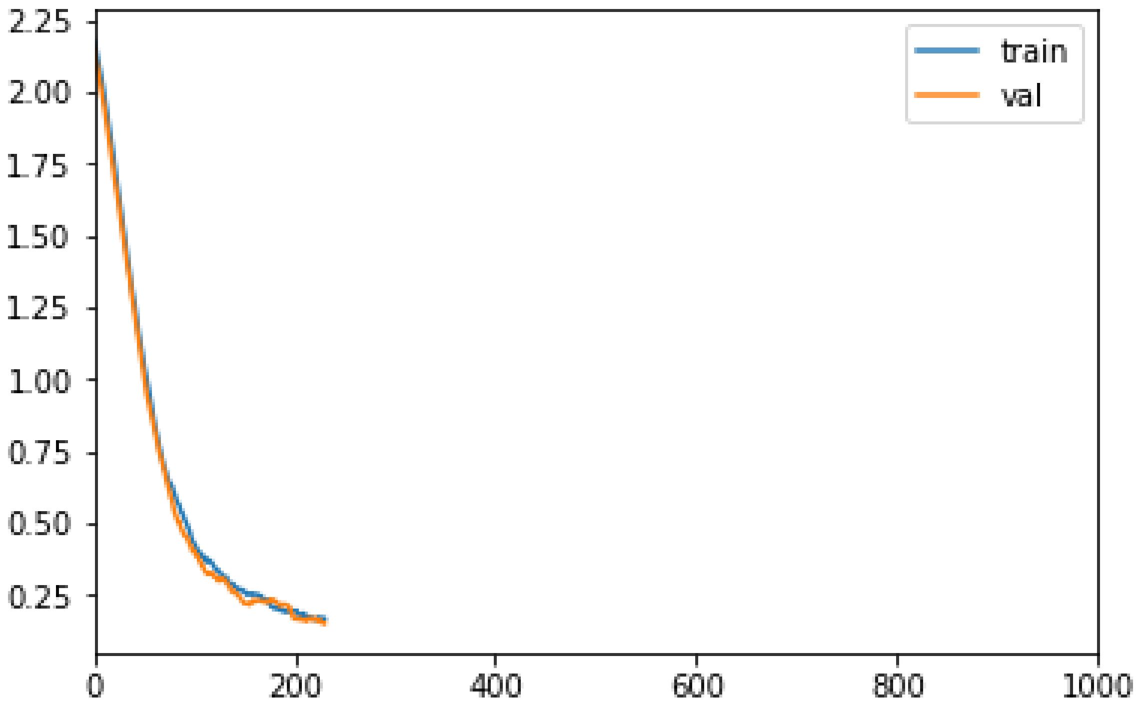
Applied the negative of gradients

Source: cs231n

MONITOR AND VISUALIZE THE LOSS CURVE



Problem: val set too small, statistics not meaningful

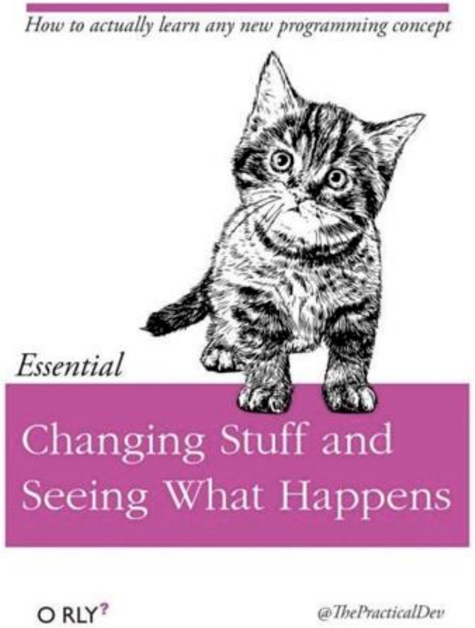


Get nans in the loss after a number of iterations: caused by high learning rate and numerical instability in models

Source: cs231n

ATTEMPT AT A CONCLUSION

- Training neural networks is still a black art
- Process requires close "babysitting"
- For many techniques, the reasons why, when, and whether they work are in active dispute – read everything but don't trust anything
- It all comes down to (principled) trial and error
- Further reading: A. Karpathy, A recipe for training neural networks



THINGS TO REMEMBER

- Training CNN
  - Adam is common (AMSGrad can be tried)
  - Learning rate: Step decay, Cyclic learning rate
  - Transfer learning, Fine tuning
- Regularization
  - L2/L1/Elastic regularization
  - Dropout and Dropconnect
  - Batch Norm
  - Data Augmentation: Flip, Crop, Contrast, etc.
- Interpreting Loss
  - Bad initialization
  - Overfitting
  - Slow/High learning rates
  - Update in wrong direction
  - Etc.

ACKNOWLEDGEMENT

- Deep Learning, Stanford University
- Introduction to Deep Learning, University of Illinois at Urbana-Champaign
- Introduction to Deep Learning, Carnegie Mellon University
- Convolutional Neural Networks for Visual Recognition, Stanford University
- Natural Language Processing with Deep Learning, Stanford University
- NVDIEA Deep Learning Teaching Kit