

Training Algorithm for Neural Networks

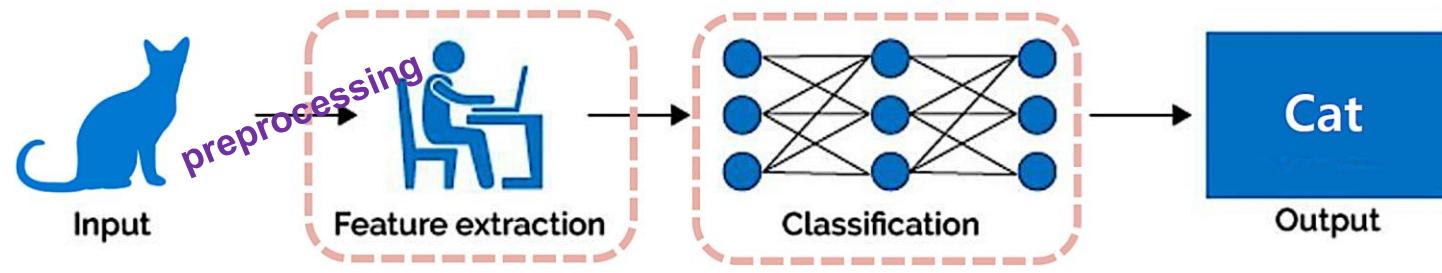
Part2

Jongbin Ryu

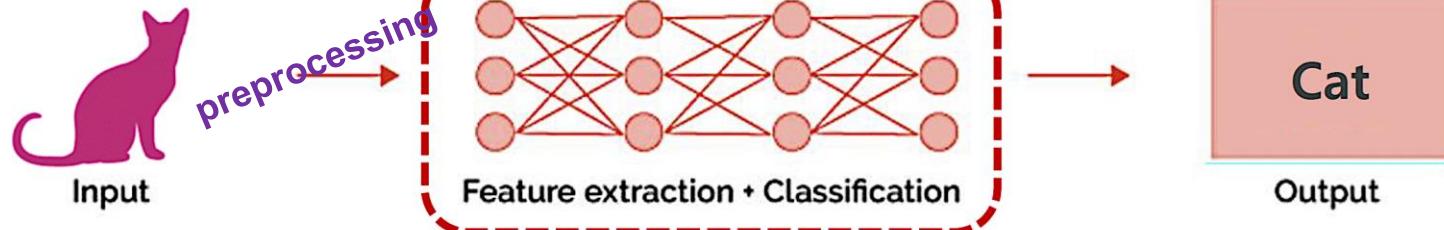


Pipeline

Machine Learning

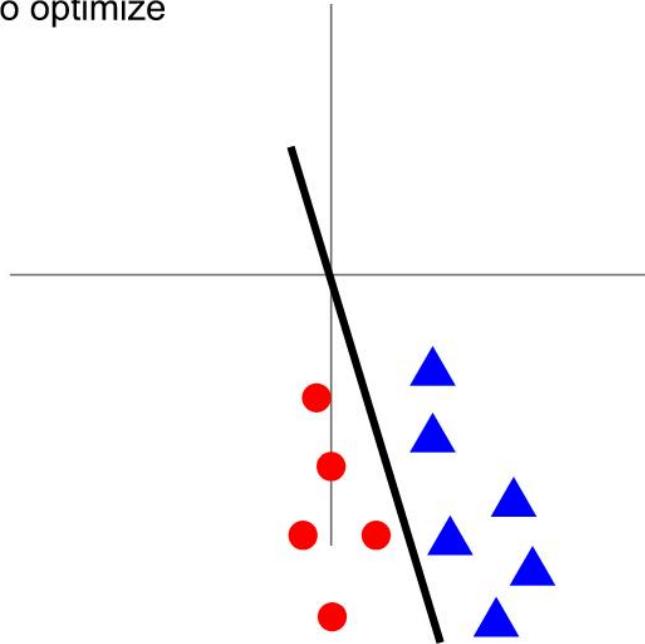


Deep Learning

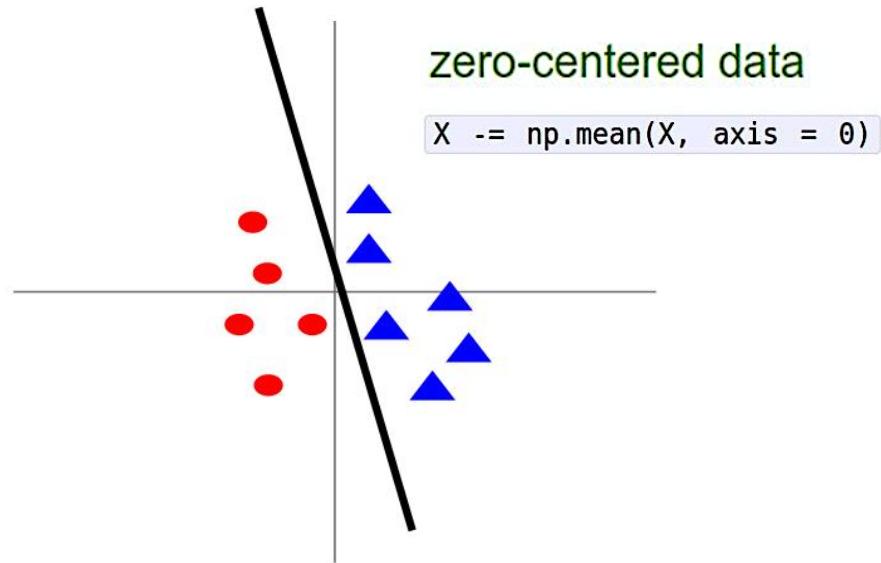


Data Preprocessing

Before normalization: classification loss very sensitive to changes in weight matrix; hard to optimize



After normalization: less sensitive to small changes in weights; easier to optimize



Convergence of Gradient Descent

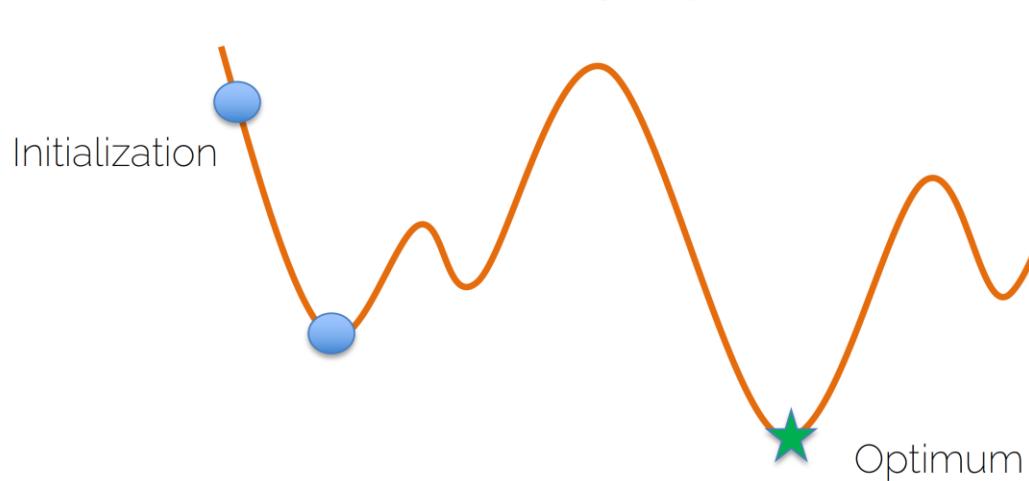
- Neural networks are non-convex
 - many (different) local minima
 - no (practical) way to say which is globally optimal



Convergence of Gradient Descent

Initialization is Extremely Important

$$x^* = \arg \min f(x)$$

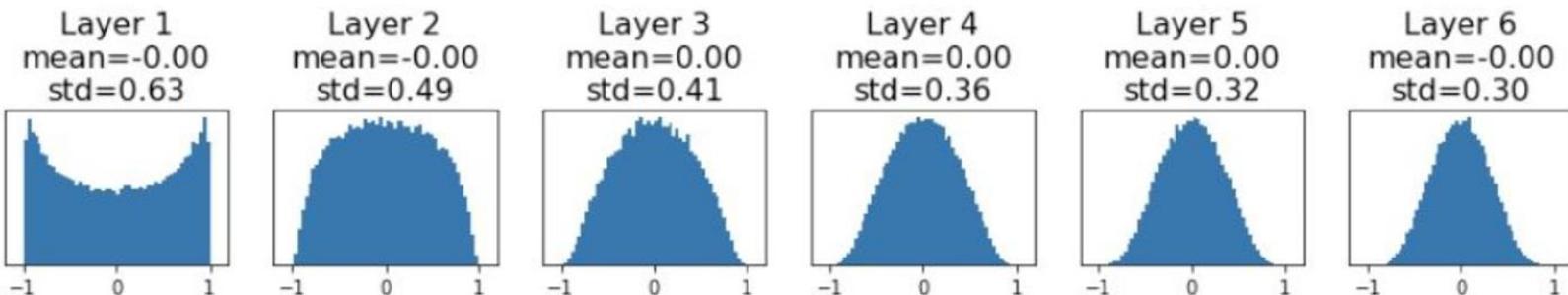


Weight Initialization: “Xavier” Initialization

```
dims = [4096] * 7          "Xavier" initialization:  
hs = []                      std = 1/sqrt(Din)  
x = np.random.randn(16, dims[0])  
for Din, Dout in zip(dims[:-1], dims[1:]):  
    W = np.random.randn(Din, Dout) / np.sqrt(Din)  
    x = np.tanh(x.dot(W))  
    hs.append(x)
```

“Just right”: Activations are nicely scaled for all layers!

For conv layers, Din is $\text{filter_size}^2 * \text{input_channels}$

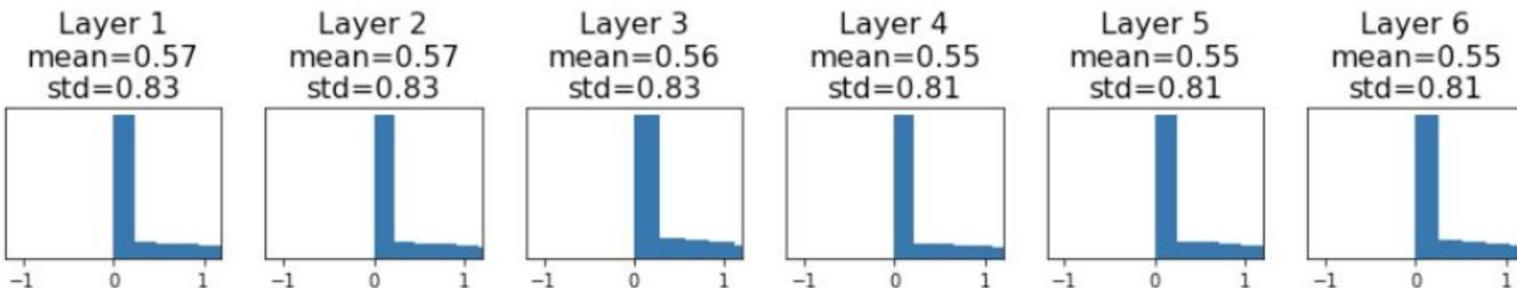


Glorot and Bengio, “Understanding the difficulty of training deep feedforward neural networks”, AISTAT 2010

Weight Initialization: Kaiming / MSRA Initialization

```
dims = [4096] * 7      ReLU correction: std = sqrt(2 / Din)
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) * np.sqrt(2/Din)
    x = np.maximum(0, x.dot(W))
    hs.append(x)
```

“Just right”: Activations are nicely scaled for all layers!



He et al, “Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification”, ICCV 2015

Batch Normalization

“you want zero-mean unit-variance activations? just make them so.”

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_1 \dots m\}$;

Parameters to be learned: γ, β

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

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

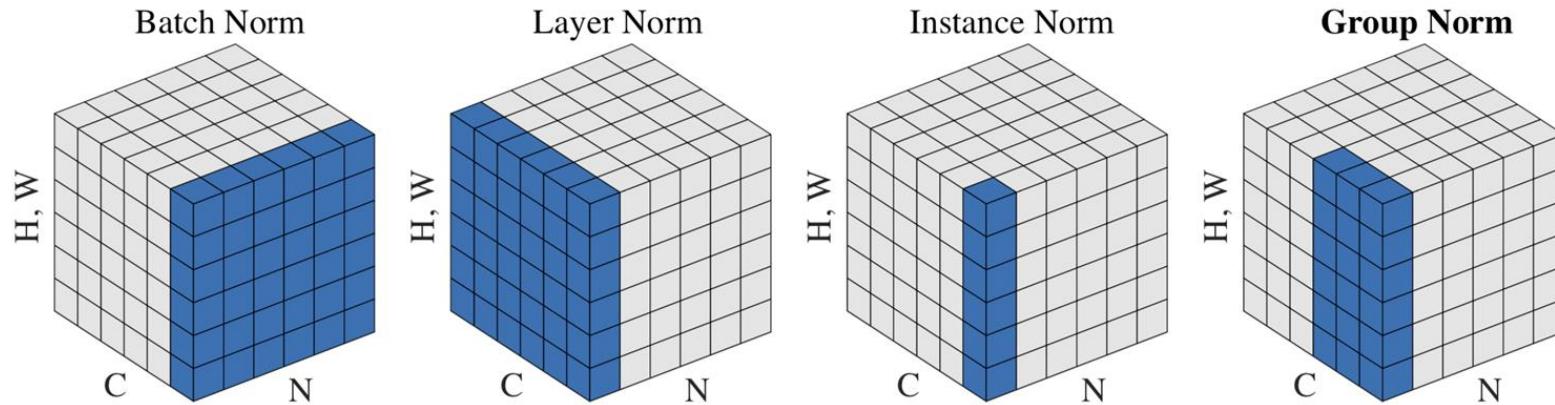
$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2 \quad // \text{mini-batch variance}$$

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

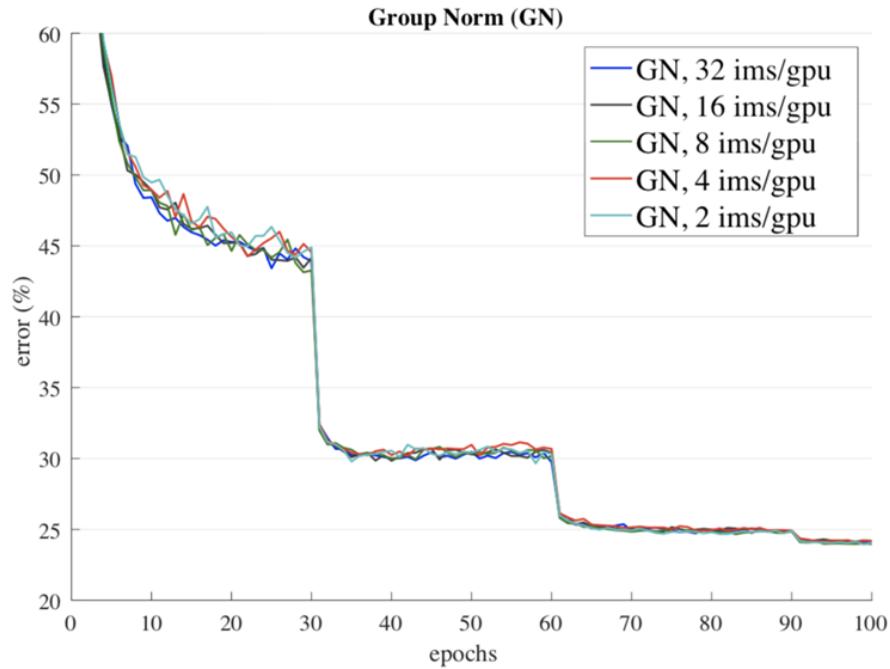
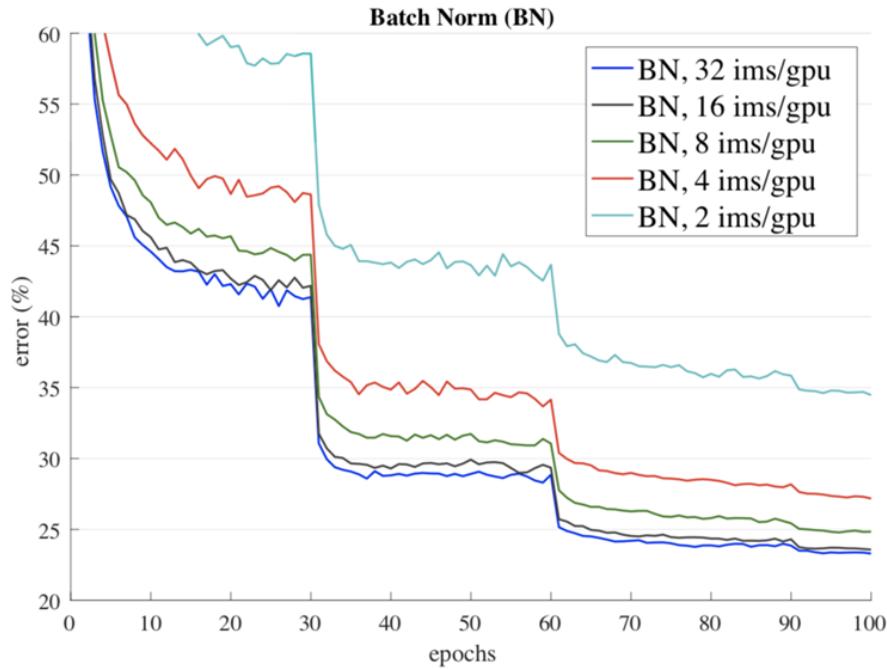
$$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma, \beta}(x_i) \quad // \text{scale and shift}$$

- Improves gradient flow through the network
- Allows higher learning rates
- Reduces the strong dependence on initialization
- Acts as a form of regularization in a funny way, and slightly reduces the need for dropout, maybe

ImageNet Classification (BN-LN-IN-GN)



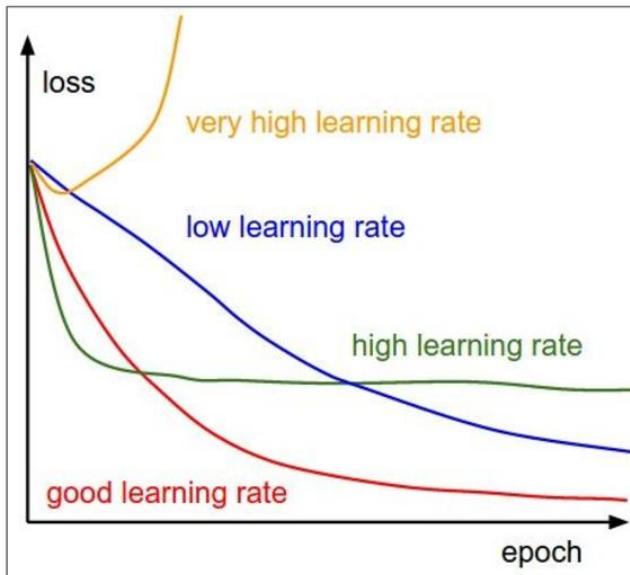
ImageNet Classification (BN vs GN)



Contents

- Learning Rate Scheduling
- Bias–Variance Tradeoff
- Regularization
- Hyperparameter Tuning
- Transfer Learning

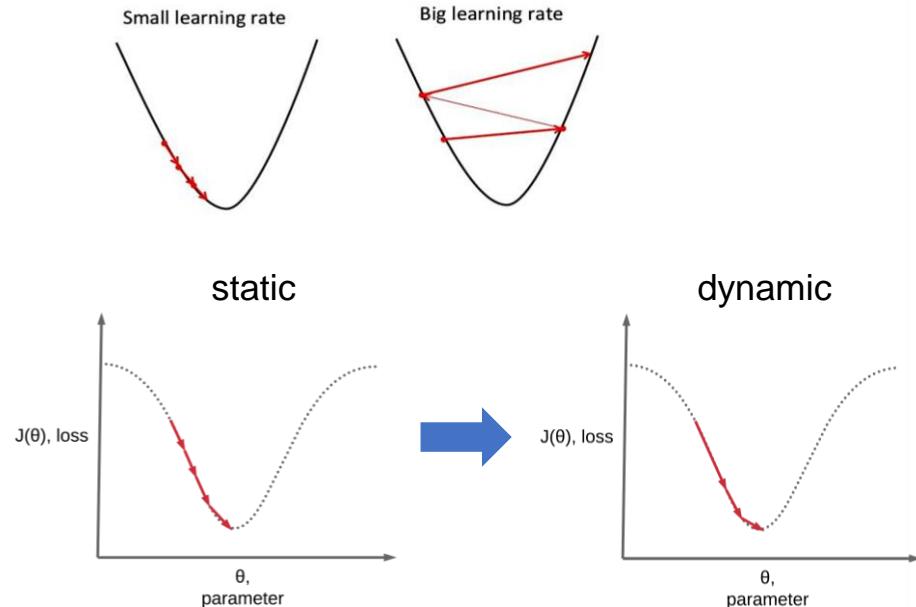
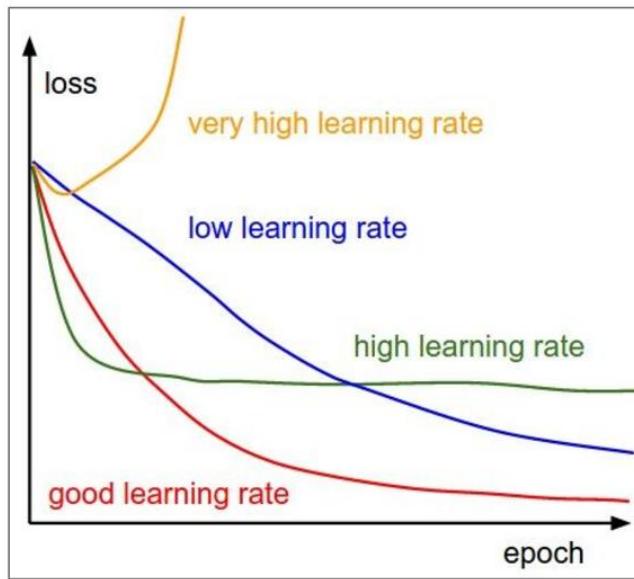
SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyperparameter.



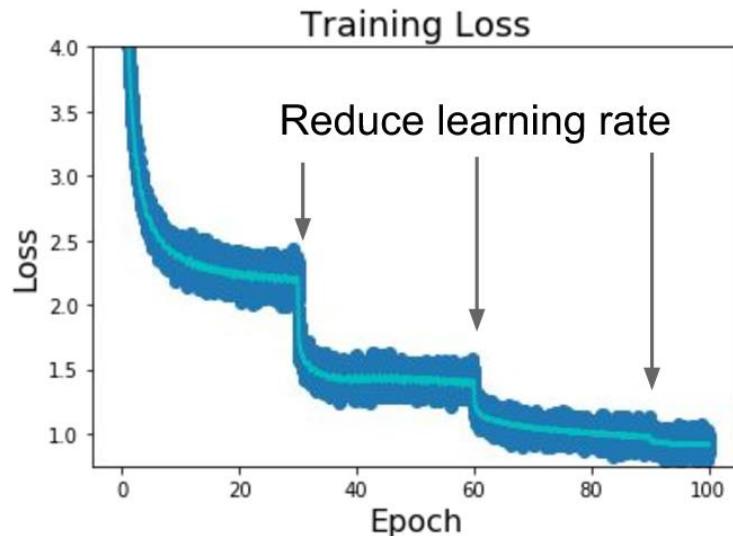
Q: Which one of these learning rates is best to use?

A: All of them! Start with large learning rate and decay over time

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

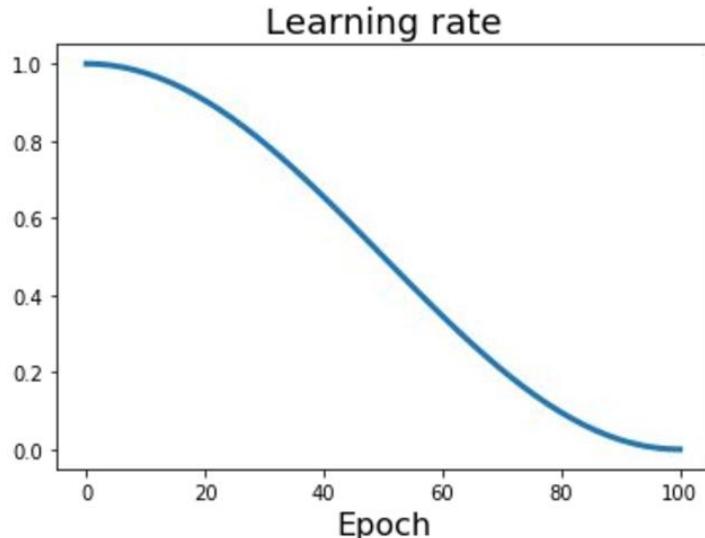


Learning Rate Decay



Step: Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

Learning Rate Decay



Step: Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

Cosine: $\alpha_t = \frac{1}{2}\alpha_0 (1 + \cos(t\pi/T))$

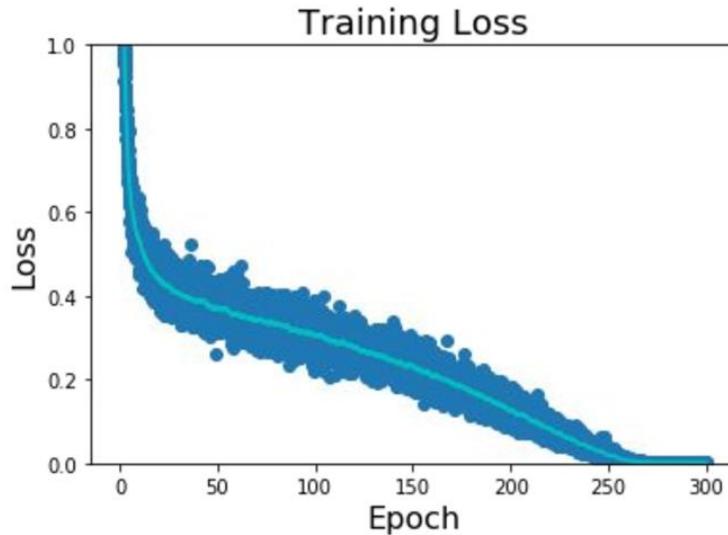
Loshchilov and Hutter, "SGDR: Stochastic Gradient Descent with Warm Restarts", ICLR 2017
Radford et al, "Improving Language Understanding by Generative Pre-Training", 2018
Feichtenhofer et al, "SlowFast Networks for Video Recognition", arXiv 2018
Child et al, "Generating Long Sequences with Sparse Transformers", arXiv 2019

α_0 : Initial learning rate

α_t : Learning rate at epoch t

T : Total number of epochs

Learning Rate Decay



Step: Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

Cosine: $\alpha_t = \frac{1}{2}\alpha_0 (1 + \cos(t\pi/T))$

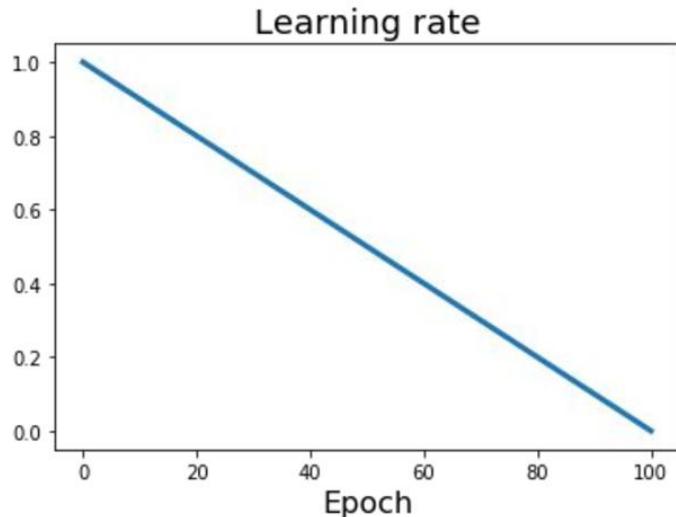
Loshchilov and Hutter, "SGDR: Stochastic Gradient Descent with Warm Restarts", ICLR 2017
Radford et al, "Improving Language Understanding by Generative Pre-Training", 2018
Feichtenhofer et al, "SlowFast Networks for Video Recognition", arXiv 2018
Child et al, "Generating Long Sequences with Sparse Transformers", arXiv 2019

α_0 : Initial learning rate

α_t : Learning rate at epoch t

T : Total number of epochs

Learning Rate Decay



Step: Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

Cosine: $\alpha_t = \frac{1}{2}\alpha_0 (1 + \cos(t\pi/T))$

Linear: $\alpha_t = \alpha_0(1 - t/T)$

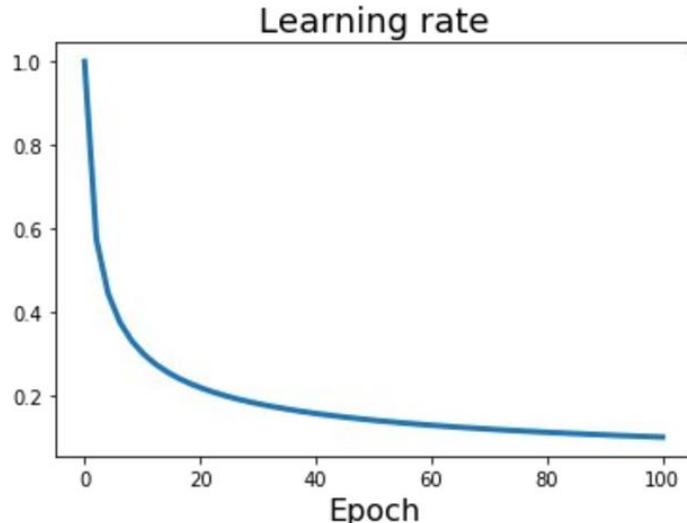
α_0 : Initial learning rate

α_t : Learning rate at epoch t

T : Total number of epochs

Devlin et al, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding", 2018

Learning Rate Decay



Step: Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

Cosine: $\alpha_t = \frac{1}{2}\alpha_0 (1 + \cos(t\pi/T))$

Linear: $\alpha_t = \alpha_0(1 - t/T)$

Inverse sqrt: $\alpha_t = \alpha_0/\sqrt{t}$

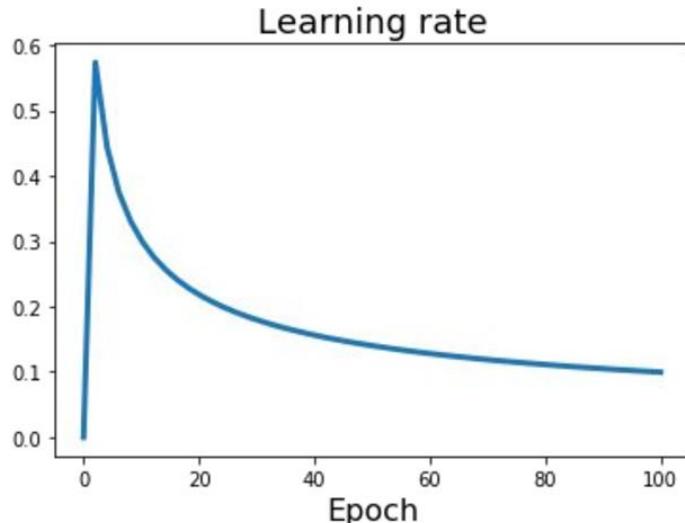
α_0 : Initial learning rate

α_t : Learning rate at epoch t

T : Total number of epochs

Vaswani et al, "Attention is all you need", NIPS 2017

Learning Rate Decay: Linear Warmup

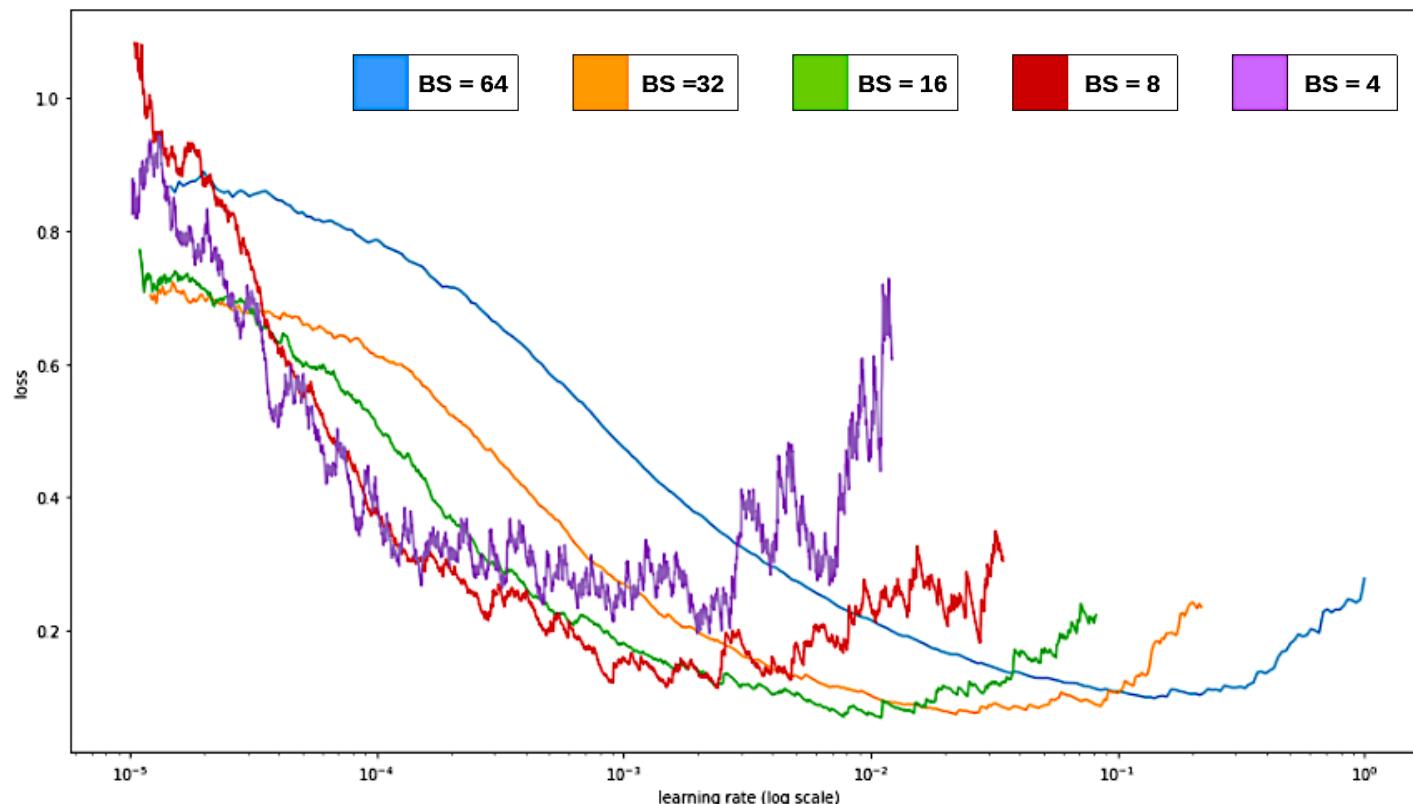


High initial learning rates can make loss explode; linearly increasing learning rate from 0 over the first ~5000 iterations can prevent this

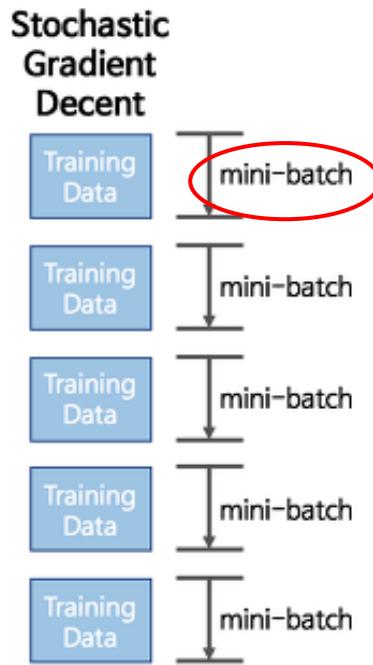
Empirical rule of thumb: If you increase the batch size by N , also scale the initial learning rate by N

Goyal et al, "Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour", arXiv 2017

Optimal Learning Rate for Different Batch sizes



Optimal Learning Rate for Different Batch sizes



$$\theta_{k+1} = \theta_k + \eta \Delta \theta_k \quad \Delta \theta_k = -\frac{1}{m} \sum_{i=1}^m \nabla L_i(\theta)$$

$$\mathbb{E}[\eta \Delta \theta] = -\eta \mathbb{E}[\nabla L_i(\theta)] \quad \text{per iteration}$$

$$\frac{1}{m} \mathbb{E}[\eta \Delta \theta] = -\frac{\eta}{m} \mathbb{E}[\nabla L_i(\theta)] \quad \text{per iteration & per sample}$$

maintaining a constant learning rate $\tilde{\eta}$

$$\tilde{\eta} = \eta \frac{1}{m} \quad \text{e.g.}
batch size=32, lr=0.001
batch size=64, lr=0.002$$

Masters, Dominic, and Carlo Luschi. "Revisiting small batch training for deep neural networks." *arXiv preprint arXiv:1804.07612* (2018).

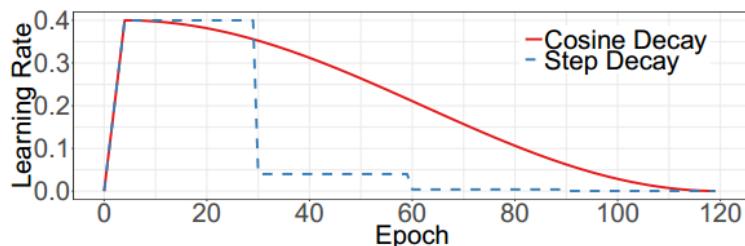
In practice:

Large Batch

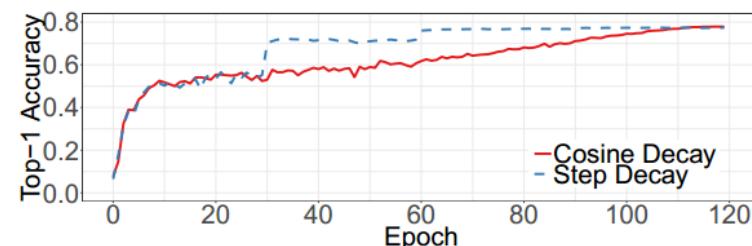
- Accurate estimate of the gradient (low variance)
- High computation cost per iteration
- High availability of parallelism (fast training)

Small Batch

- Noisy estimate of the gradient (high variance)
- Low computation cost per iteration
- Low availability of parallelism (slow training)



(a) Learning Rate Schedule



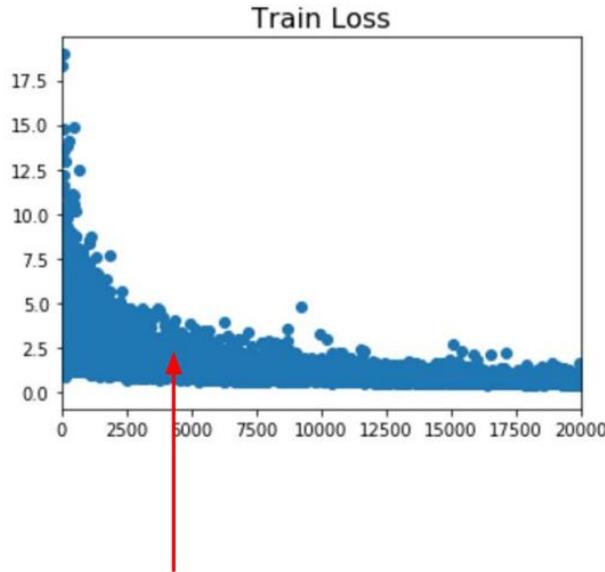
(b) Validation Accuracy

He, Tong, et al. "Bag of tricks for image classification with convolutional neural networks."
Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 2019.

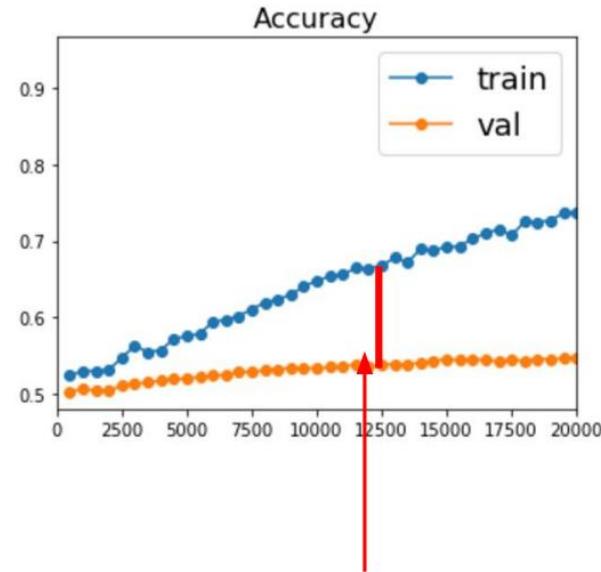
Contents

- Learning Rate Scheduling
- **Bias–Variance Tradeoff**
- Regularization
- Hyperparamter Tuning
- Transfer Learning

Beyond Training Error

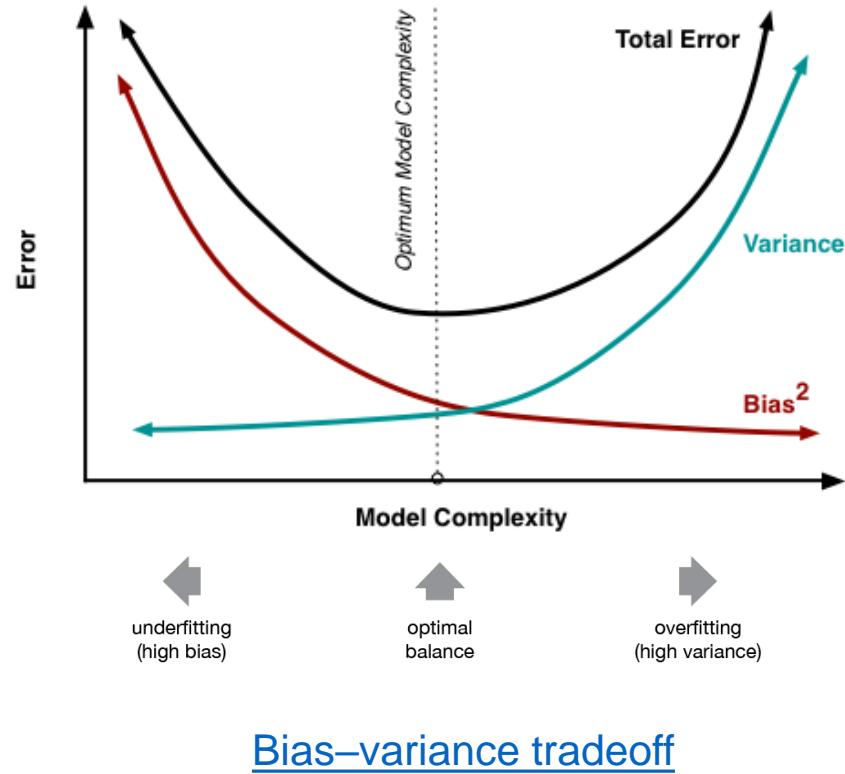
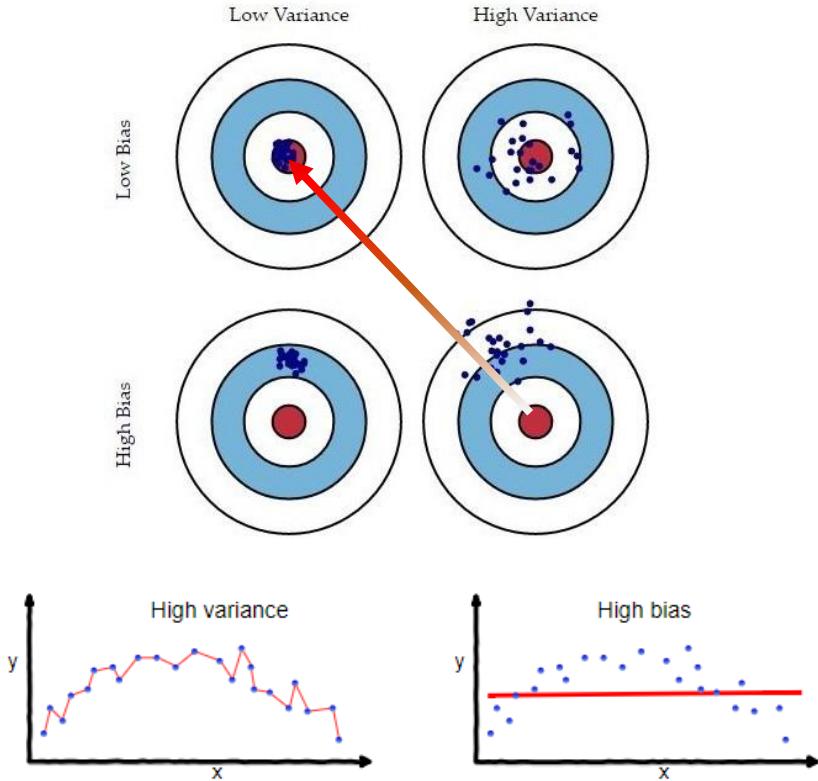


Better optimization algorithms
help reduce training loss

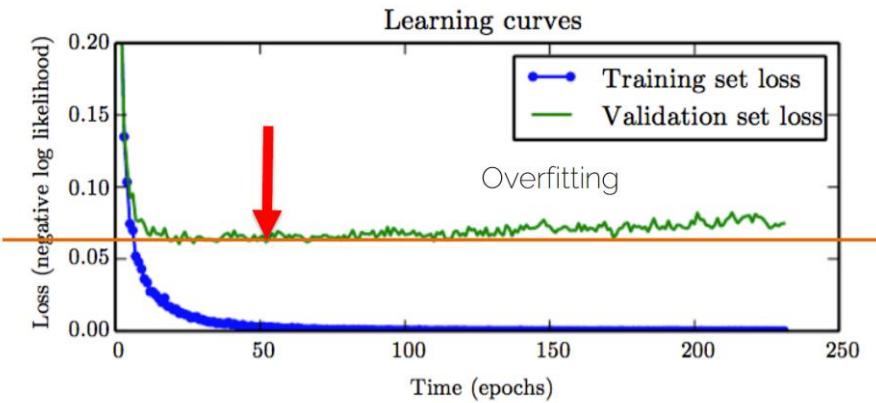


But we really care about error on
new data - how to reduce the gap?

Bias–variance tradeoff



Early Stopping: Always do this



```
# Additional information
EPOCH = 5
PATH = "model.pt"
LOSS = 0.4

torch.save({
    'epoch': EPOCH,
    'model_state_dict': net.state_dict(),
    'optimizer_state_dict': optimizer.state_dict(),
    'loss': LOSS,
}, PATH)
```

checkpoint

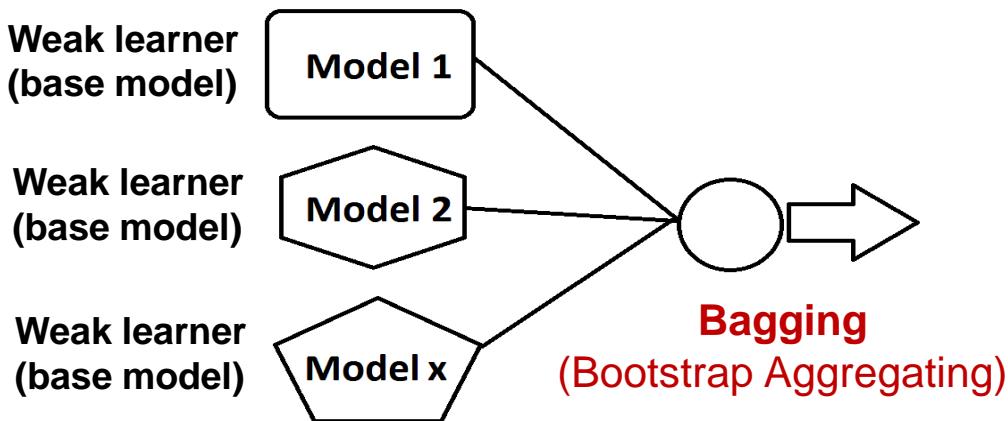
Stop training the model when accuracy on the validation set decreases
Or train for a long time, but always keep track of the model snapshot
that worked best on val

Model Ensembles

Enjoy 2% extra performance

1. Train multiple independent models
2. At test time average their results

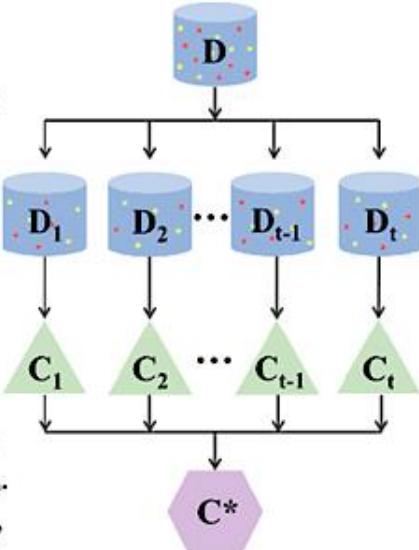
(Take average of predicted probability distributions, then choose argmax)



Model Ensembles

(A) bagging

step 1
create multiple data sets through random sampling with replacement



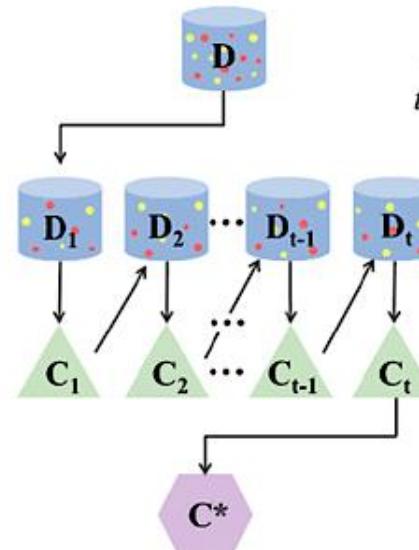
step 2
build multiple learners in parallel

step 3
combine all learners using an averaging or majority-vote strategy

e.g. Random Forest, DNN

(B) boosting

step 1
create multiple data sets through random sampling with replacement over weighted data

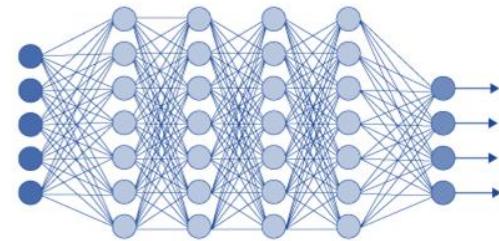
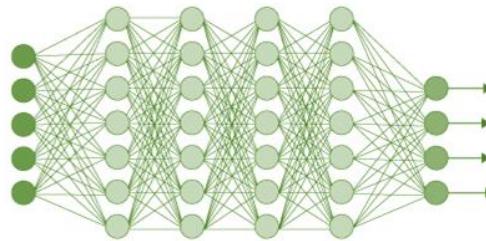
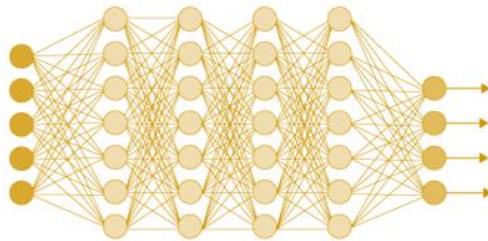


step 2
build learners sequentially

step 3
combine all learners using a weighted-averaging strategy

e.g. AdaBoost, GBM, XGBoost

Model Ensembles in Deep Learning

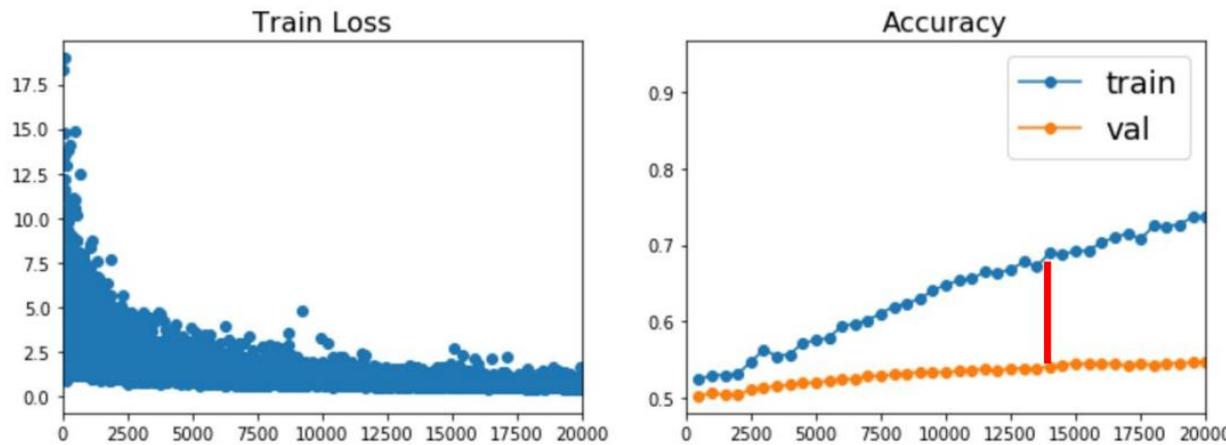


- Same model but different **initializations**
- Same model but different **optimization/objective function**
- Same model but different **datasets**
- **Top models** discovered during cross-validation
- Different **checkpoints** (i.e. iteration) of a single model
- Running **average of parameters** during training

Contents

- Learning Rate Scheduling
- Bias–Variance Tradeoff
- **Regularization**
- Hyperparameter Tuning
- Transfer Learning

How to improve single-model performance?



Regularization

Regularization: Add term to loss

$$L = \frac{1}{N} \sum_{i=1}^N \sum_{j \neq y_i} \max(0, f(x_i; W)_j - f(x_i; W)_{y_i} + 1) + \boxed{\lambda R(W)}$$

In common use:

L2 regularization

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

L1 regularization

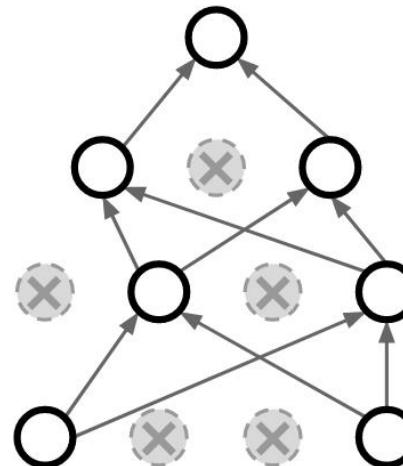
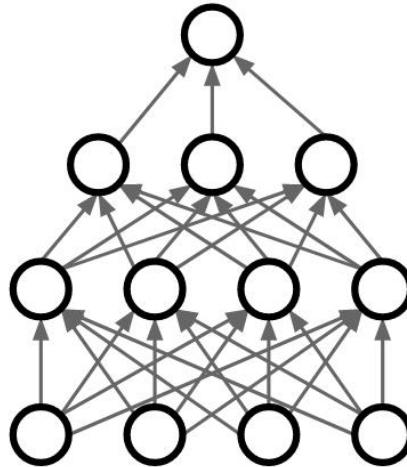
$$R(W) = \sum_k \sum_l |W_{k,l}|$$

Elastic net (L1 + L2)

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

Regularization: Dropout

In each forward pass, randomly set some neurons to zero
Probability of dropping is a hyperparameter; 0.5 is common



Srivastava et al., "Dropout: A simple way to prevent neural networks from overfitting", JMLR 2014

Regularization: Dropout

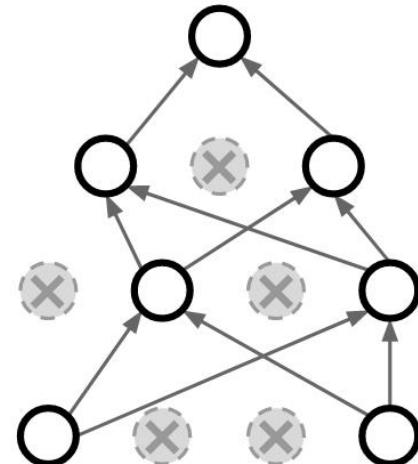
```
p = 0.5 # probability of keeping a unit active. higher = less dropout

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

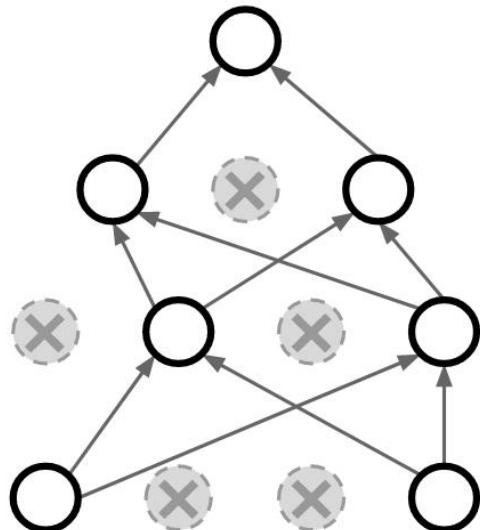
    # backward pass: compute gradients... (not shown)
    # perform parameter update... (not shown)
```

Example forward pass with a 3-layer network using dropout



Regularization: Dropout

How can this possibly be a good idea?

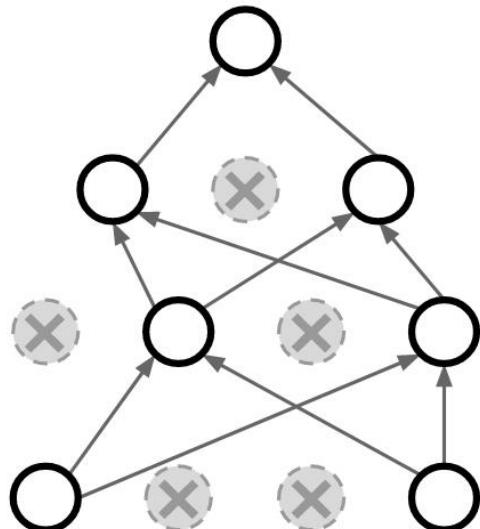


Forces the network to have a redundant representation;
Prevents co-adaptation of features



Regularization: Dropout

How can this possibly be a good idea?



Another interpretation:

Dropout is training a large **ensemble** of models (that share parameters).

Each binary mask is one model

An FC layer with 4096 units has
 $2^{4096} \sim 10^{1233}$ possible masks!

Dropout: Test time

Dropout makes our output random!

$$\boxed{y} = f_W(\boxed{x}, \boxed{z})$$

Output (label) Input (image) Random mask

Want to “average out” the randomness at test-time

$$y = f(x) = E_z [f(x, z)] = \int p(z) f(x, z) dz$$

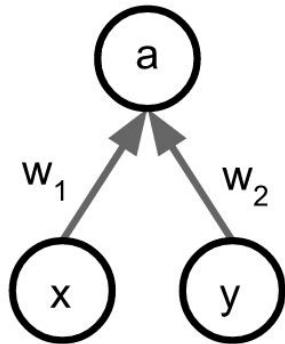
But this integral seems hard ...

Dropout: Test time

Want to approximate
the integral

$$y = f(x) = E_z[f(x, z)] = \int p(z)f(x, z)dz$$

Consider a single neuron.



At test time we have: $E[a] = w_1x + w_2y$

During training we have:

$$\begin{aligned} E[a] &= \frac{1}{4}(w_1x + w_2y) + \frac{1}{4}(w_1x + 0y) \\ &\quad + \frac{1}{4}(0x + 0y) + \frac{1}{4}(0x + w_2y) \\ &= \frac{1}{2}(w_1x + w_2y) \end{aligned}$$

At test time, multiply
by dropout probability

Dropout: Test time

```
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

Dropout Summary

```
""" Vanilla Dropout: Not recommended implementation (see notes below) """
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)

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
```

drop in train time

scale at test time

More common: “Inverted dropout”

```
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!



Regularization: A common pattern

Training: Add some kind
of randomness

$$y = f_W(x, z)$$

Testing: Average out randomness
(sometimes approximate)

$$y = f(x) = E_z[f(x, z)] = \int p(z)f(x, z)dz$$

Regularization: A common pattern

Training: Add some kind of randomness

$$y = f_W(x, z)$$

Testing: Average out randomness (sometimes approximate)

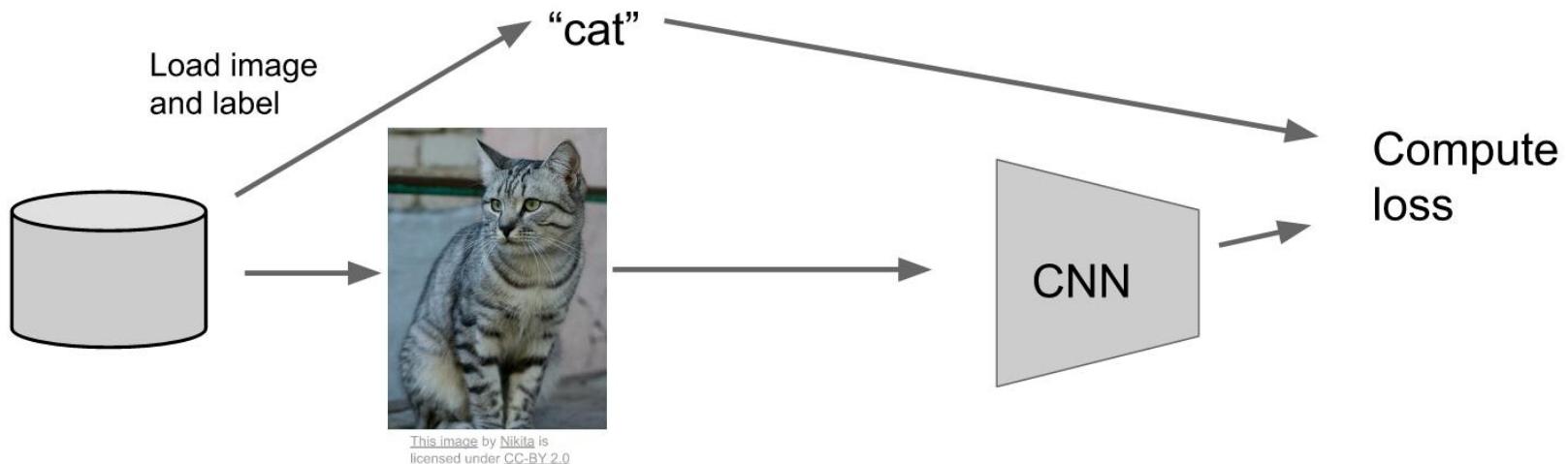
$$y = f(x) = E_z[f(x, z)] = \int p(z)f(x, z)dz$$

Example: Batch Normalization

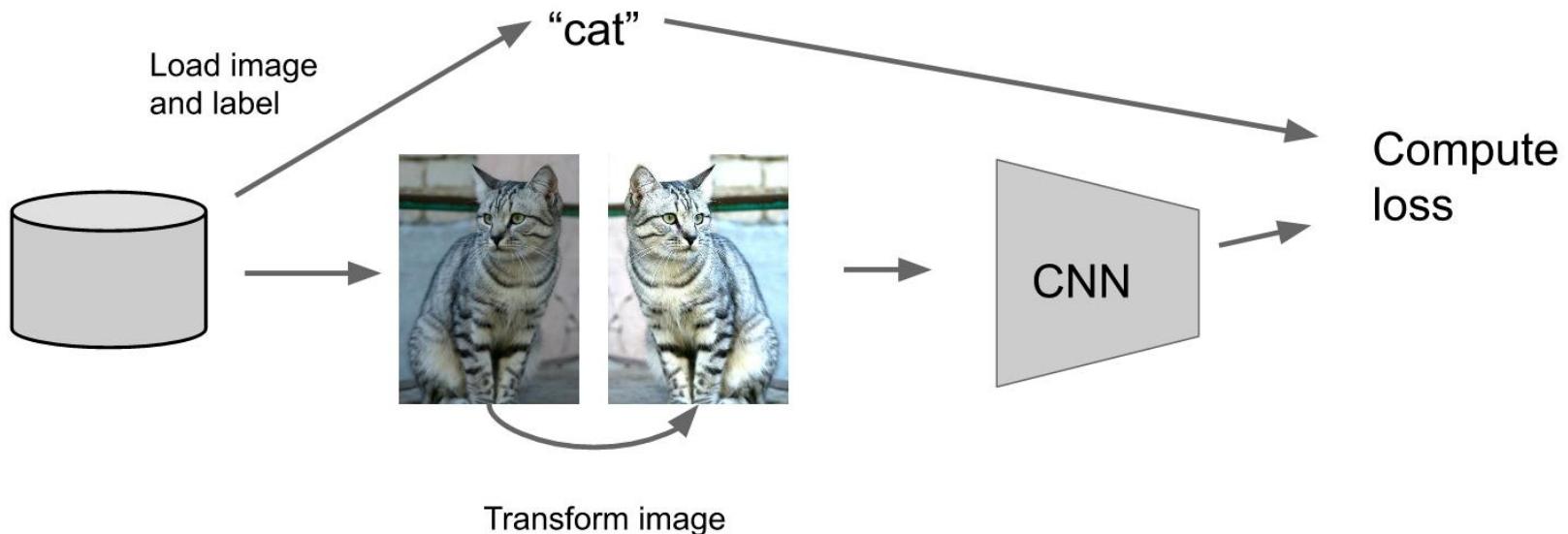
Training: Normalize using stats from random minibatches

Testing: Use fixed stats to normalize

Regularization: Data Augmentation

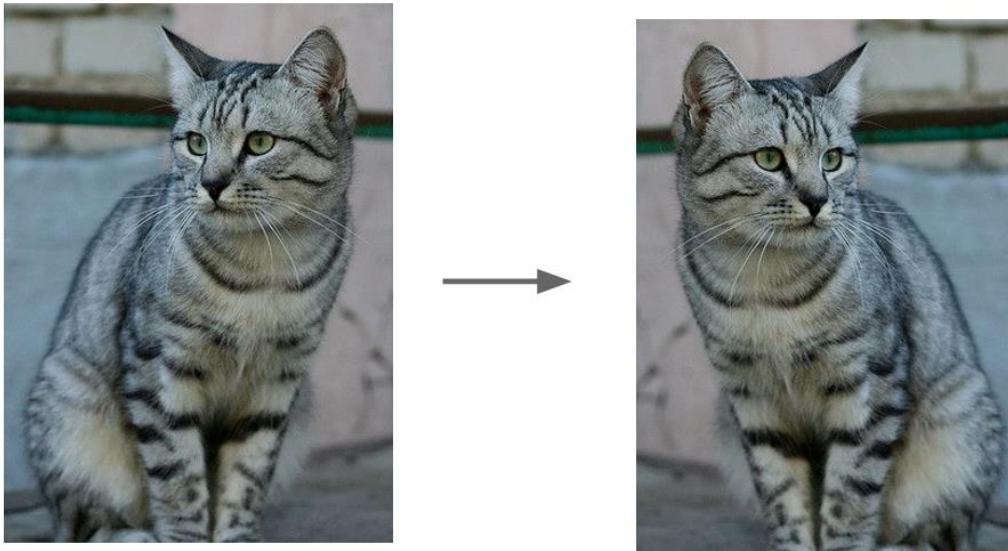


Regularization: Data Augmentation



Data Augmentation

Horizontal Flips



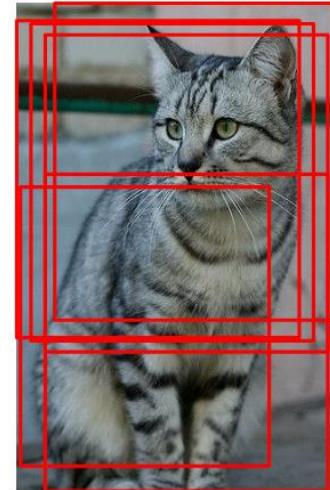
Data Augmentation

Random crops and scales

Training: sample random crops / scales

ResNet:

1. Pick random L in range [256, 480]
2. Resize training image, short side = L
3. Sample random 224×224 patch



Data Augmentation

Random crops and scales

Training: sample random crops / scales

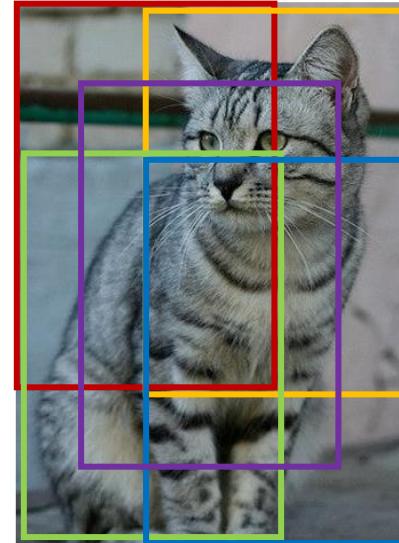
ResNet:

1. Pick random L in range [256, 480]
2. Resize training image, short side = L
3. Sample random 224×224 patch

Testing: average a fixed set of crops

ResNet:

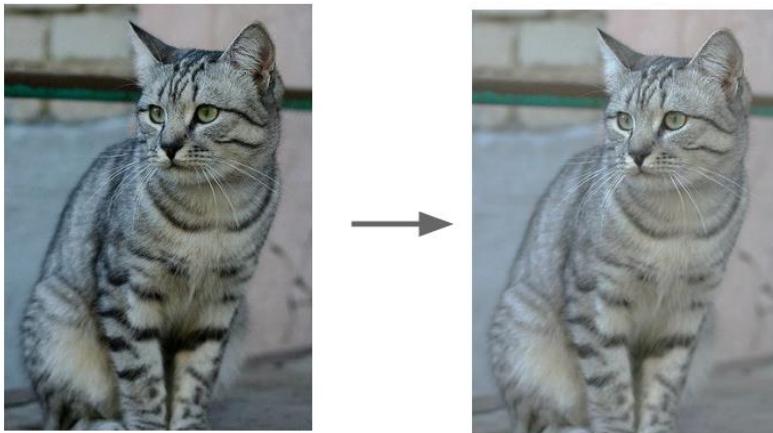
1. Resize image at 5 scales: {224, 256, 384, 480, 640}
2. For each size, use 10 224×224 crops: 4 corners + center, + flips



Data Augmentation

Color Jitter

Simple: Randomize
contrast and brightness

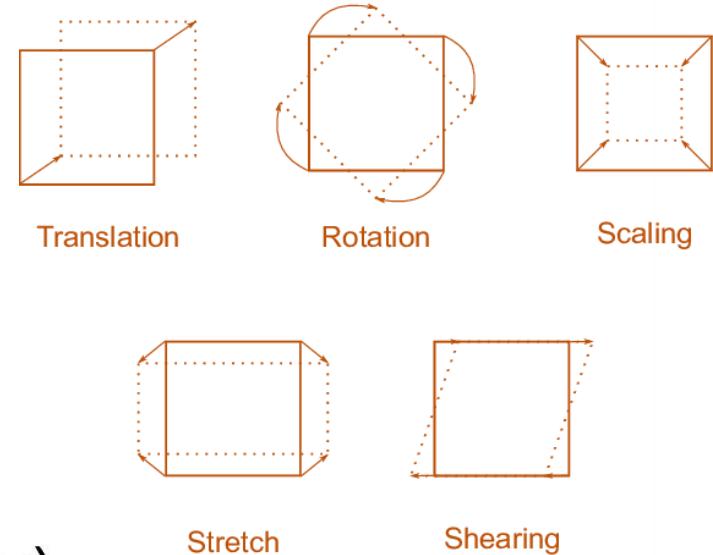


Data Augmentation

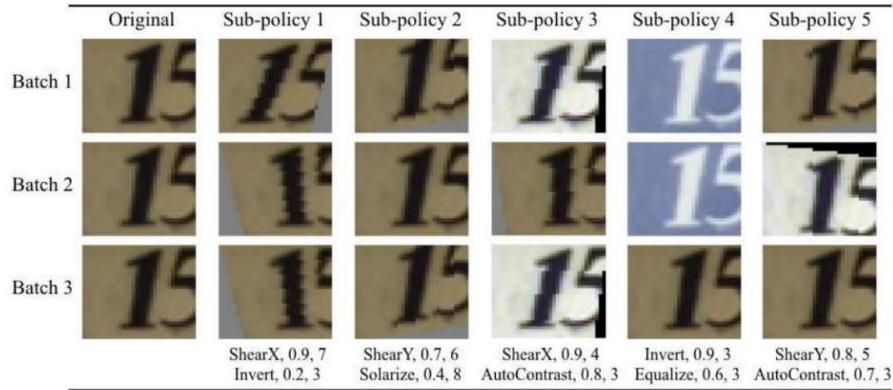
Get creative for your problem!

Random mix/combinations of :

- translation
- rotation
- stretching
- shearing,
- lens distortions, ... (go crazy)



Automatic Data Augmentation



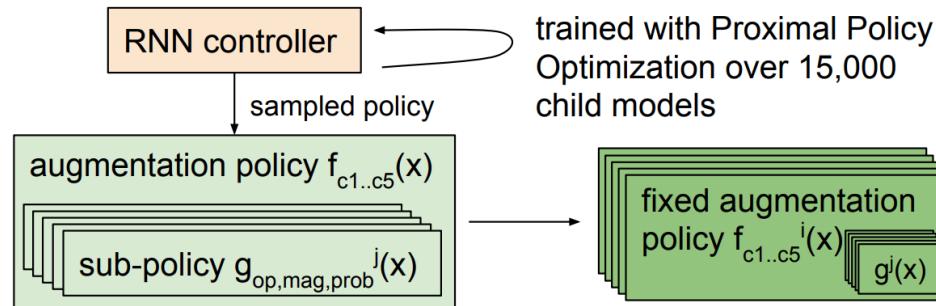
Size of Search Space

- 16 Data Augmentation Operations
 - 10 Magnitudes, (uniformly spaced)
 - 11 Values for Probability of Applying Operation, (uniformly spaced)
 - 5 Sub-policies, Each Containing 2 Operations
-
- $(16 \times 10 \times 11)^{10} \sim 2.9 \times 10^{32}$ possibilities

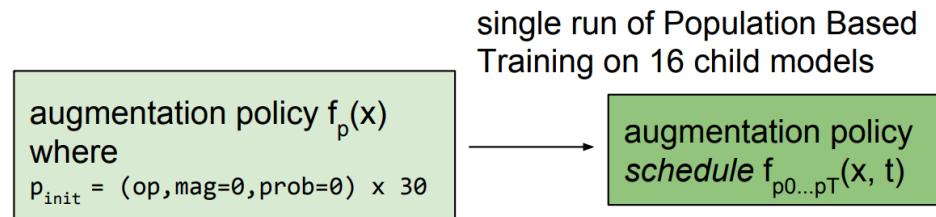
Cubuk et al., "AutoAugment: Learning Augmentation Strategies from Data", CVPR 2019

Automatic Data Augmentation

(a) AutoAugment

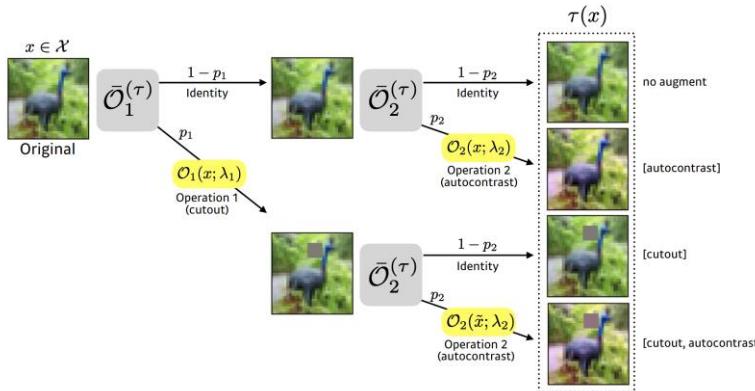


(b) Population Based Augmentation



Ho, Daniel, et al. "Population based augmentation: Efficient learning of augmentation policy schedules." *ICML* 2019

Automatic Data Augmentation

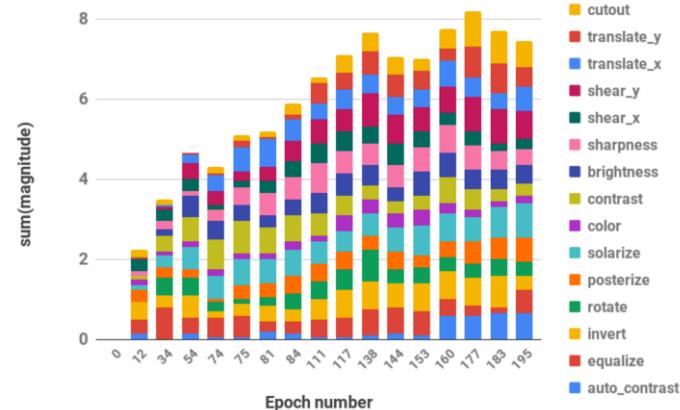


Model	Baseline	AutoAugment [3]	Fast AutoAugment
ResNet-50	23.7 / 6.9	22.4 / 6.2	22.4 / 6.3
ResNet-200	21.5 / 5.8	20.00 / 5.0	19.4 / 4.7

Table 5: Validation set Top-1 / Top-5 error rate (%) on ImageNet.

Lim, Sungbin, et al. "Fast autoaugment." *NeurIPS* 2019

Dataset	Value	Previous Best	AA	PBA
CIFAR-10	GPU Hours	-	5000	5
	Test Error	2.1	1.48	1.46
CIFAR-100	GPU Hours	-	0*	0*
	Test Error	12.2	10.7	10.9
SVHN	GPU Hours	-	1000	1
	Test Error	1.3	1.0	1.1



Ho, Daniel, et al. "Population based augmentation: Efficient learning of augmentation policy schedules." *ICML* 2019

Regularization: A common pattern

Training: Add random noise

Testing: Marginalize over the noise

Examples:

Dropout

Data Augmentation

Regularization: DropConnect

Training: Drop connections between neurons (set weights to 0)

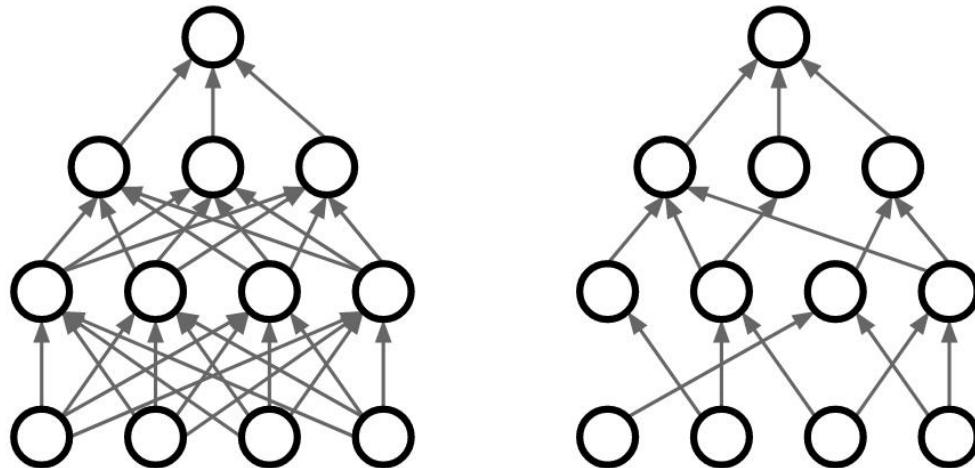
Testing: Use all the connections

Examples:

Dropout

Data Augmentation

DropConnect



Wan et al, "Regularization of Neural Networks using DropConnect", ICML 2013

Regularization: Cutout

Training: Set random image regions to zero

Testing: Use full image

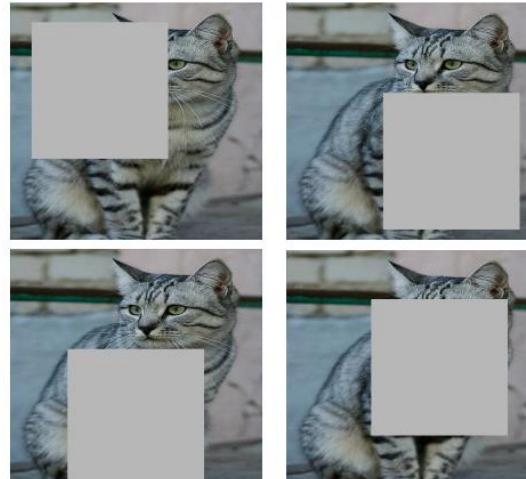
Examples:

Dropout

Data Augmentation

DropConnect

Cutout / Random Crop



Works very well for small datasets like CIFAR,
less common for large datasets like ImageNet

DeVries and Taylor, "Improved Regularization of
Convolutional Neural Networks with Cutout", arXiv 2017

Regularization: Mixup

Training: Train on random blends of images

Testing: Use original images

Examples:

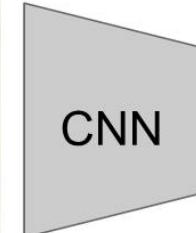
Dropout

Data Augmentation

DropConnect

Cutout / Random Crop

Mixup



Target label:
cat: 0.4
dog: 0.6

Randomly blend the pixels
of pairs of training images,
e.g. 40% cat, 60% dog

Zhang et al, "mixup: Beyond Empirical Risk Minimization", ICLR 2018

Regularization: Cutmix

Training: Train on random blends of images

Testing: Use original images

Examples:

Dropout

Data Augmentation

DropConnect

Cutout / Random Crop

Mixup

Cutmix

	Image	ResNet-50	Mixup	Cutout	CutMix
Label		Dog 1.0	Dog 0.5 Cat 0.5	Dog 1.0	Dog 0.6 Cat 0.4
					

Yun, Sangdoo, et al. "Cutmix: Regularization strategy to train strong classifiers with localizable features." /ICCV 2019

Regularization - In practice

Training: Add random noise

Testing: Marginalize over the noise

Examples:

Dropout

Data Augmentation

DropConnect

Cutout / Random Crop

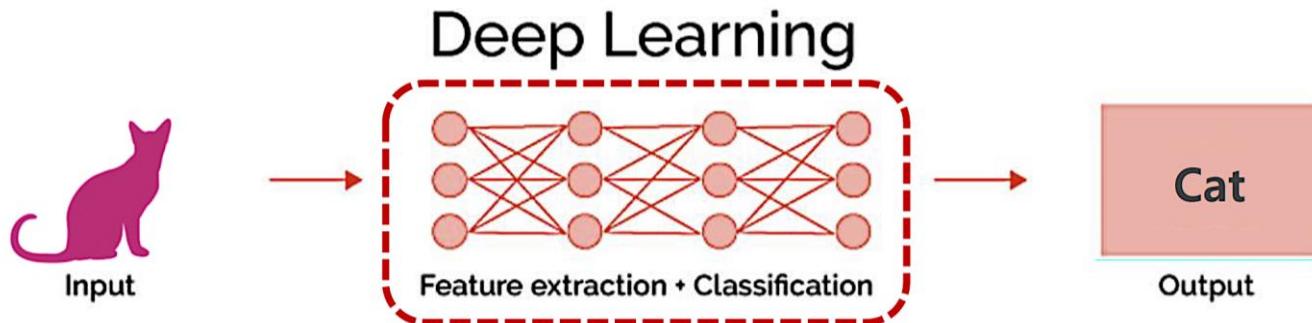
Mixup

Cutmix

- Consider dropout for large fully-connected layers
- Batch normalization and data augmentation almost always a good idea
- Try cutout and mixup especially for small classification datasets
- Try Cutmix if possible

Contents

- Learning Rate Scheduling
- Bias–Variance Tradeoff
- Regularization
- **Hyperparameter Tuning**
- Transfer Learning



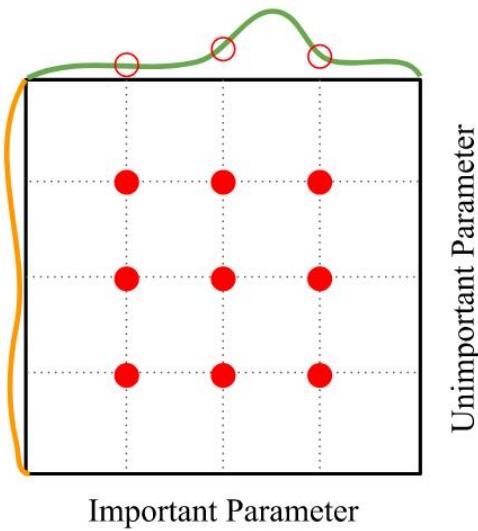
Hyperparameters

- Network architecture (e.g., num layers, #weights)
- Number of iterations
- Learning rate(s) (i.e., solver parameters, decay, etc.)
- Regularization (more later next lecture)
- Batch size

Random Search vs. Grid Search

*Random Search for
Hyper-Parameter Optimization
Bergstra and Bengio, 2012*

Grid Layout



Random Layout

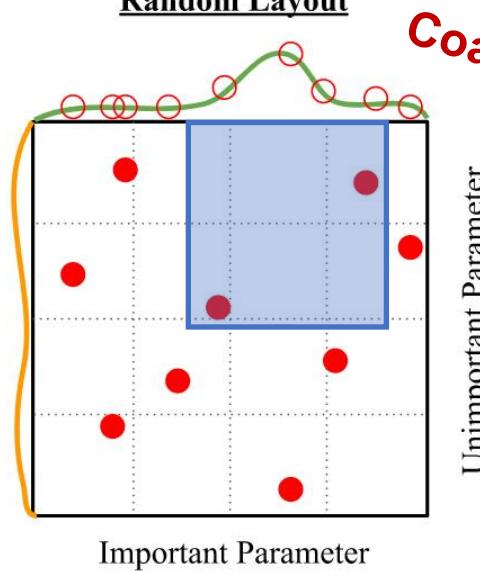


Illustration of Bergstra et al., 2012 by Shayne
Longpre, copyright CS231n 2017

For example: run coarse search for 5 epochs

```
max_count = 100
for count in xrange(max_count):
    reg = 10**uniform(-5, 5)
    lr = 10**uniform(-3, -6) ←
        trainer = ClassifierTrainer()
        model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
        trainer = ClassifierTrainer()
        best_model_local, stats = trainer.train(X_train, y_train, X_val, y_val,
                                                model, two_layer_net,
                                                num_epochs=5, reg=reg,
                                                update='momentum', learning_rate_decay=0.9,
                                                sample_batches = True, batch_size = 100,
                                                learning rate=lr, verbose=False)
```

note it's best to optimize
in log space!

```
val acc: 0.412000, lr: 1.405206e-04, reg: 4.793564e-01, (1 / 100)
val acc: 0.214000, lr: 7.231888e-06, reg: 2.321281e-04, (2 / 100)
val acc: 0.208000, lr: 2.119571e-06, reg: 8.011857e+01, (3 / 100)
val acc: 0.196000, lr: 1.551131e-05, reg: 4.374936e-05, (4 / 100)
val acc: 0.079000, lr: 1.753300e-05, reg: 1.200424e+03, (5 / 100)
val acc: 0.223000, lr: 4.215128e-05, reg: 4.196174e+01, (6 / 100)
val acc: 0.441000, lr: 1.750259e-04, reg: 2.110807e-04, (7 / 100)
val acc: 0.241000, lr: 6.749231e-05, reg: 4.226413e+01, (8 / 100)
val acc: 0.482000, lr: 4.296863e-04, reg: 6.642555e-01, (9 / 100)
val acc: 0.079000, lr: 5.401602e-06, reg: 1.599828e+04, (10 / 100)
val acc: 0.154000, lr: 1.618508e-06, reg: 4.925252e-01, (11 / 100)
```

nice

Now run finer search...

```
max_count = 100
for count in xrange(max_count):
    reg = 10**uniform(-5, 5)
    lr = 10**uniform(-3, -6)
```

adjust range

```
max_count = 100
for count in xrange(max_count):
    reg = 10**uniform(-4, 0)
    lr = 10**uniform(-3, -4)
```

```
val_acc: 0.527000, lr: 5.340517e-04, reg: 4.097824e-01, (0 / 100)
val_acc: 0.492000, lr: 2.279484e-04, reg: 9.991345e-04, (1 / 100)
val_acc: 0.512000, lr: 8.680827e-04, reg: 1.349727e-02, (2 / 100)
val_acc: 0.461000, lr: 1.028377e-04, reg: 1.220193e-02, (3 / 100)
val_acc: 0.460000, lr: 1.113730e-04, reg: 5.244309e-02, (4 / 100)
val_acc: 0.498000, lr: 9.477776e-04, reg: 2.001293e-03, (5 / 100)
val_acc: 0.469000, lr: 1.484369e-04, reg: 4.328313e-01, (6 / 100)
val_acc: 0.522000, lr: 5.586261e-04, reg: 2.312685e-04, (7 / 100)
val_acc: 0.530000, lr: 5.808183e-04, reg: 8.259964e-02, (8 / 100)
val_acc: 0.489000, lr: 1.979168e-04, reg: 1.010889e-04, (9 / 100)
val_acc: 0.490000, lr: 2.036031e-04, reg: 2.406271e-03, (10 / 100)
val_acc: 0.475000, lr: 2.021162e-04, reg: 2.287807e-01, (11 / 100)
val_acc: 0.460000, lr: 1.135527e-04, reg: 3.905040e-02, (12 / 100)
val_acc: 0.515000, lr: 6.947668e-04, reg: 1.562808e-02, (13 / 100)
val_acc: 0.531000, lr: 9.471549e-04, reg: 1.433895e-03, (14 / 100)
val_acc: 0.509000, lr: 3.140888e-04, reg: 2.857518e-01, (15 / 100)
val_acc: 0.514000, lr: 6.438349e-04, reg: 3.033781e-01, (16 / 100)
val_acc: 0.502000, lr: 3.921784e-04, reg: 2.707126e-04, (17 / 100)
val_acc: 0.509000, lr: 9.752279e-04, reg: 2.850865e-03, (18 / 100)
val_acc: 0.500000, lr: 2.412048e-04, reg: 4.997821e-04, (19 / 100)
val_acc: 0.466000, lr: 1.319314e-04, reg: 1.189915e-02, (20 / 100)
val_acc: 0.516000, lr: 8.039527e-04, reg: 1.528291e-02, (21 / 100)
```

53% - relatively good
for a 2-layer neural net
with 50 hidden neurons.

Now run finer search...

```
max_count = 100
for count in xrange(max_count):
    reg = 10**uniform(-5, 5)
    lr = 10**uniform(-3, -6)
```

adjust range

```
max_count = 100
for count in xrange(max_count):
    reg = 10**uniform(-4, 0)
    lr = 10**uniform(-3, -4)
```

```
val_acc: 0.527000, lr: 5.340517e-04, reg: 4.097824e-01, (0 / 100)
val_acc: 0.492000, lr: 2.279484e-04, reg: 9.991345e-04, (1 / 100)
val_acc: 0.512000, lr: 8.680827e-04, reg: 1.349727e-02, (2 / 100)
val_acc: 0.461000, lr: 1.028377e-04, reg: 1.220193e-02, (3 / 100)
val_acc: 0.460000, lr: 1.113730e-04, reg: 5.244309e-02, (4 / 100)
val_acc: 0.498000, lr: 9.477776e-04, reg: 2.001293e-03, (5 / 100)
val_acc: 0.469000, lr: 1.484369e-04, reg: 4.328313e-01, (6 / 100)
val_acc: 0.522000, lr: 5.586261e-04, reg: 2.312685e-04, (7 / 100)
val_acc: 0.530000, lr: 5.808183e-04, reg: 8.259964e-02, (8 / 100)
val_acc: 0.489000, lr: 1.979168e-04, reg: 1.010889e-04, (9 / 100)
val_acc: 0.490000, lr: 2.036031e-04, reg: 2.406271e-03, (10 / 100)
val_acc: 0.475000, lr: 2.021162e-04, reg: 2.287807e-01, (11 / 100)
val_acc: 0.460000, lr: 1.135527e-04, reg: 3.905040e-02, (12 / 100)
val_acc: 0.515000, lr: 6.947668e-04, reg: 1.562808e-02, (13 / 100)
val_acc: 0.531000, lr: 9.471549e-04, reg: 1.433895e-03, (14 / 100) ←
val_acc: 0.509000, lr: 3.140888e-04, reg: 2.857518e-01, (15 / 100)
val_acc: 0.514000, lr: 6.438349e-04, reg: 3.033781e-01, (16 / 100)
val_acc: 0.502000, lr: 3.921784e-04, reg: 2.707126e-04, (17 / 100)
val_acc: 0.509000, lr: 9.752279e-04, reg: 2.850865e-03, (18 / 100)
val_acc: 0.500000, lr: 2.412048e-04, reg: 4.997821e-04, (19 / 100)
val_acc: 0.466000, lr: 1.319314e-04, reg: 1.189915e-02, (20 / 100)
val_acc: 0.516000, lr: 8.039527e-04, reg: 1.528291e-02, (21 / 100)
```

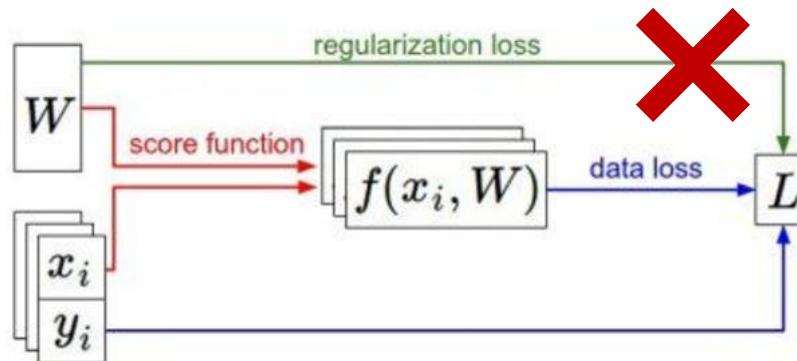
53% - relatively good
for a 2-layer neural net
with 50 hidden neurons.

But this best
cross-validation result is
worrying. Why?

Choosing Hyperparameters

Step 1: Check initial loss

Turn off weight decay, sanity check loss at initialization
e.g. $\log(C)$ for softmax with C classes



Choosing Hyperparameters

Step 1: Check initial loss

Step 2: Overfit a small sample

Try to train to 100% training accuracy on a small sample of training data (~5-10 minibatches); fiddle with architecture, learning rate, weight initialization

Loss not going down? LR too low, bad initialization

Loss explodes to Inf or NaN? LR too high, bad initialization

Choosing Hyperparameters

Step 1: Check initial loss

Step 2: Overfit a small sample

Step 3: Find LR that makes loss go down

Use the architecture from the previous step, use all training data, turn on small weight decay, find a learning rate that makes the loss drop significantly within ~100 iterations

Good learning rates to try: 1e-1, 1e-2, 1e-3, 1e-4

Choosing Hyperparameters

Step 1: Check initial loss

Step 2: Overfit a small sample

Step 3: Find LR that makes loss go down

Step 4: Coarse grid, train for ~1-5 epochs

Choose a few values of learning rate and weight decay around what worked from Step 3, train a few models for ~1-5 epochs.

Good weight decay to try: 1e-4, 1e-5, 0

Choosing Hyperparameters

Step 1: Check initial loss

Step 2: Overfit a small sample

Step 3: Find LR that makes loss go down

Step 4: Coarse grid, train for ~1-5 epochs

Step 5: Refine grid, train longer

Pick best models from Step 4, train them for longer (~10-20 epochs) without learning rate decay

Choosing Hyperparameters

Step 1: Check initial loss

Step 2: Overfit a small sample

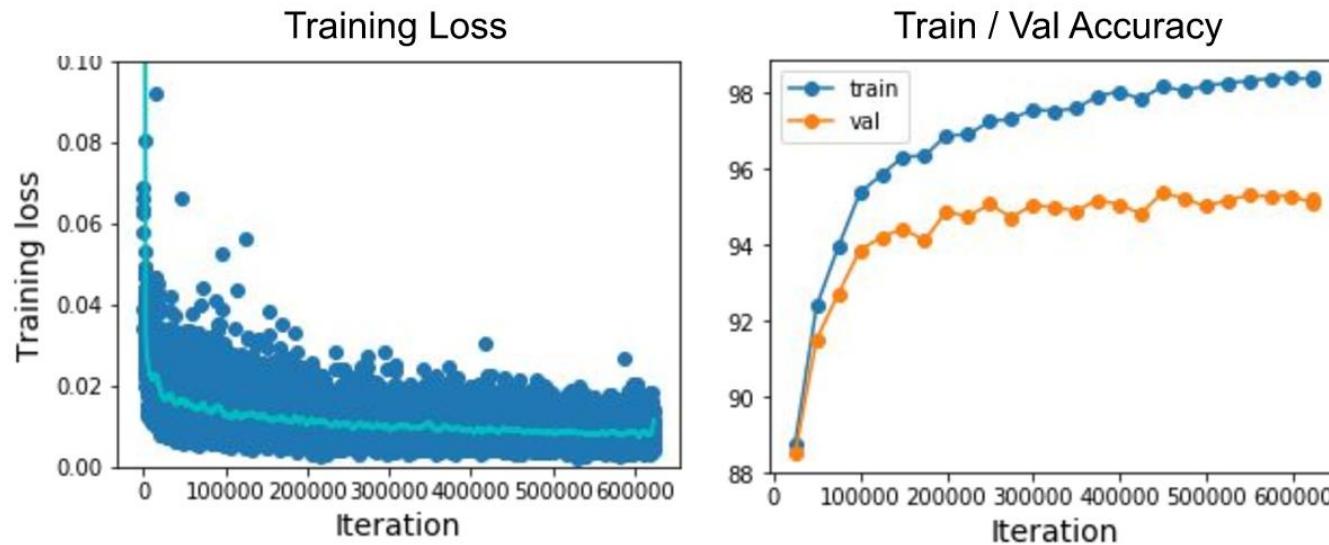
Step 3: Find LR that makes loss go down

Step 4: Coarse grid, train for ~1-5 epochs

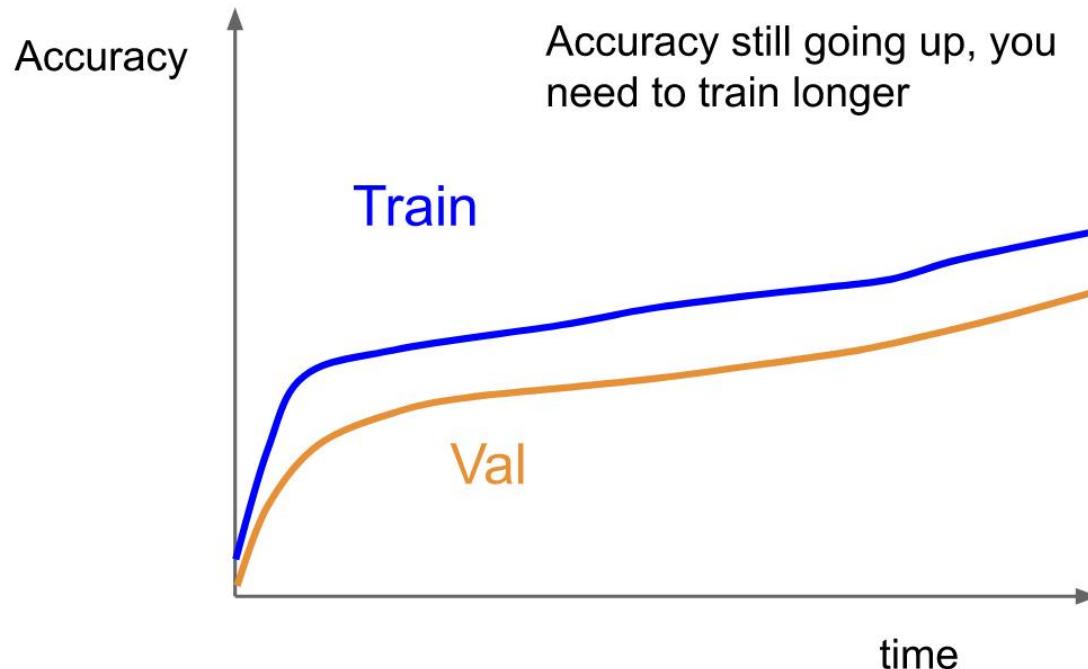
Step 5: Refine grid, train longer

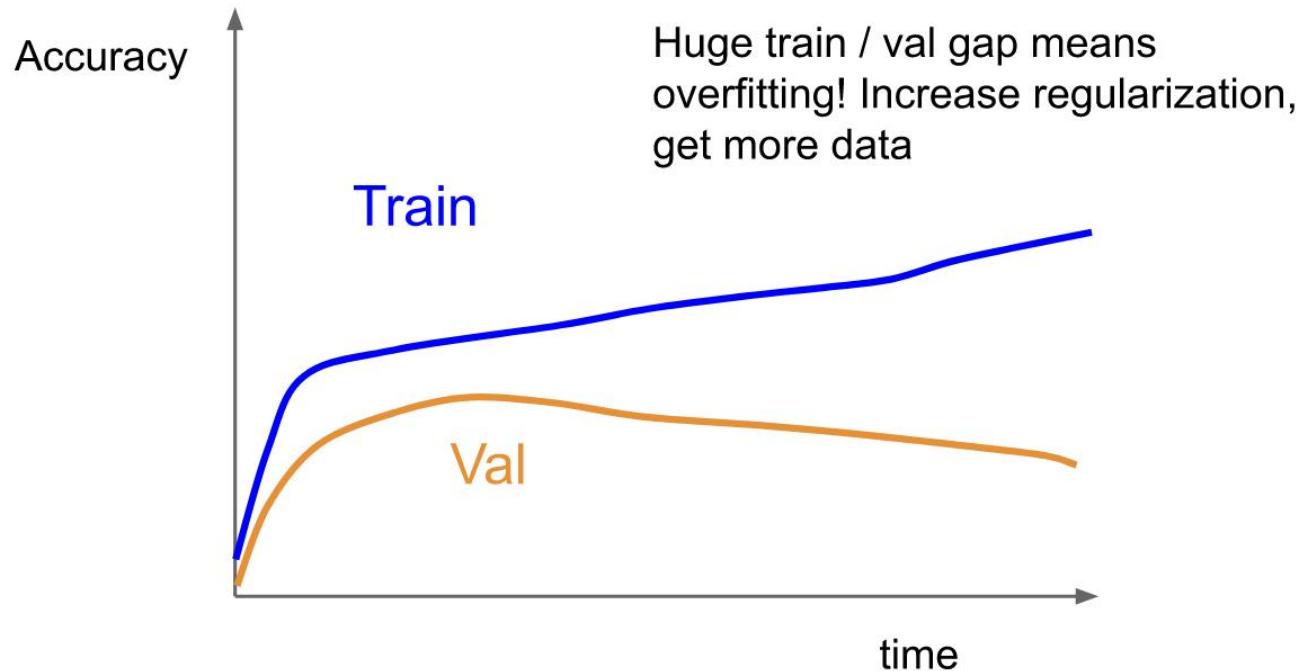
Step 6: Look at loss curves

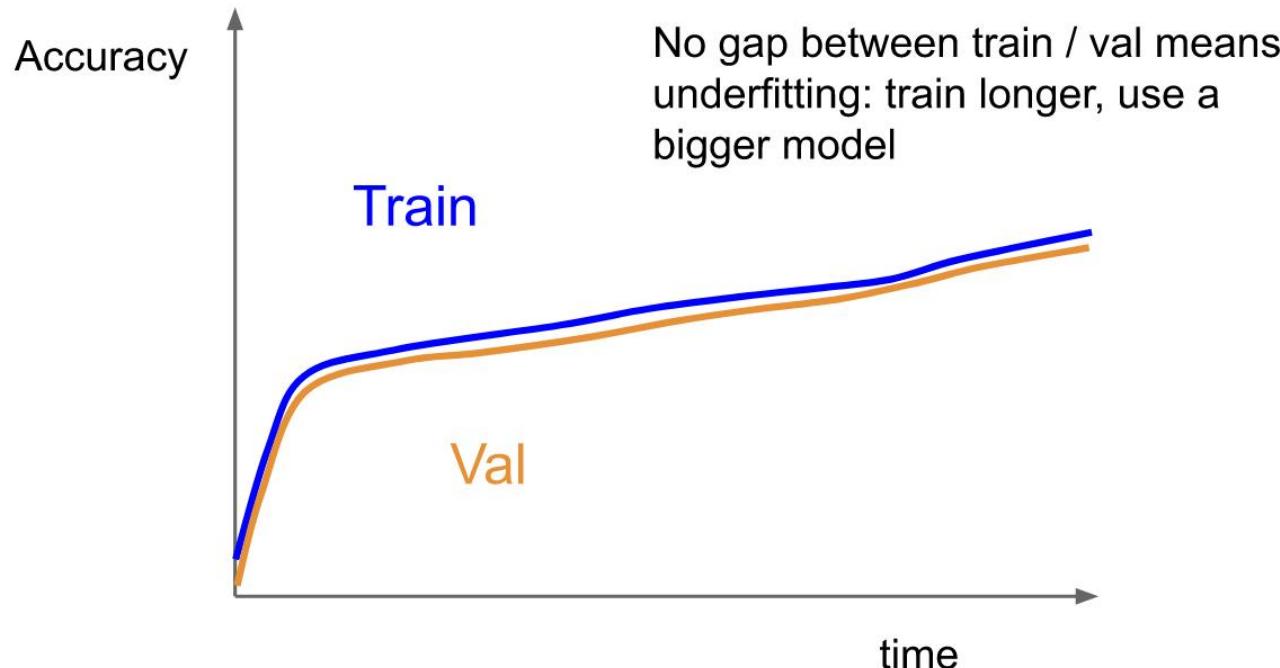
Look at learning curves!



Losses may be noisy, use a scatter plot and also plot moving average to see trends better







Choosing Hyperparameters

Step 1: Check initial loss

Step 2: Overfit a small sample

Step 3: Find LR that makes loss go down

Step 4: Coarse grid, train for ~1-5 epochs

Step 5: Refine grid, train longer

Step 6: Look at loss curves

Step 7: GOTO step 5



Hyperparameters to play with:

- network architecture
- learning rate, its decay schedule, update type
- regularization (L2/Dropout strength)

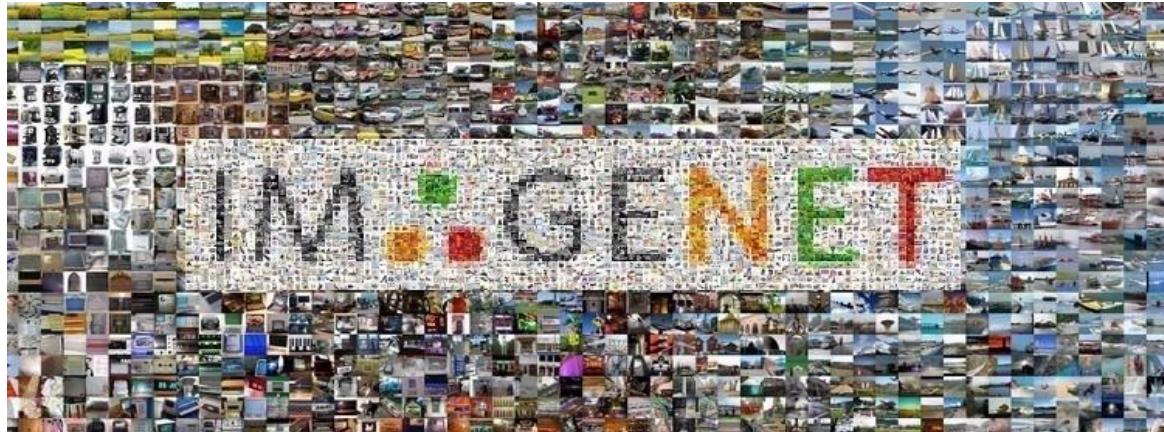
Contents

- Learning Rate Scheduling
- Bias–Variance Tradeoff
- Regularization
- Hyperparameter Tuning
- Transfer Learning

Transfer Learning

1. Train on Imagenet

FC-1000
FC-4096
FC-4096
MaxPool
Conv-512
Conv-512
MaxPool
Conv-512
Conv-512
MaxPool
Conv-256
Conv-256
MaxPool
Conv-128
Conv-128
MaxPool
Conv-64
Conv-64
Image



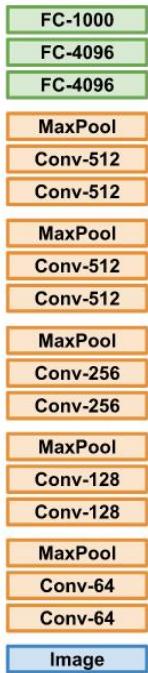
14 million images

ImageNet is a large database or dataset of over 14 million images. It was designed by academics intended for computer vision research. It was the first of its kind in terms of scale. Images are organized and labelled in a hierarchy. Jul 3, 2019

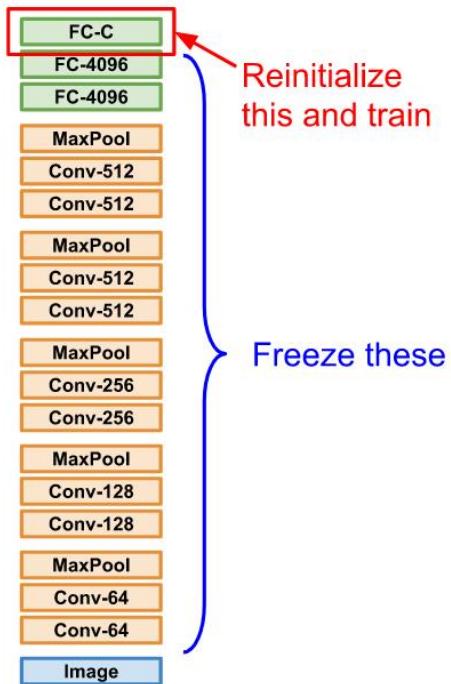
Donahue et al, "DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition", ICML 2014
Razavian et al, "CNN Features Off-the-Shelf: An Astounding Baseline for Recognition", CVPR Workshops 2014

Transfer Learning

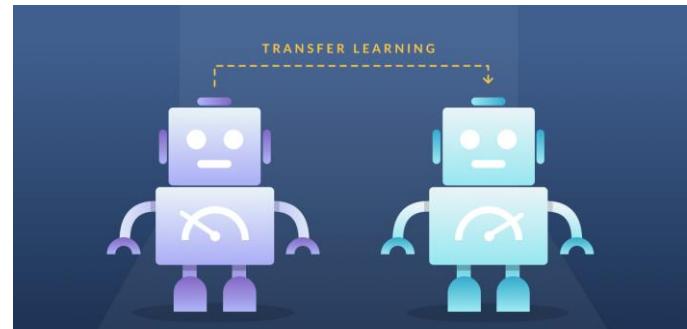
1. Train on Imagenet



2. Small Dataset (C classes)

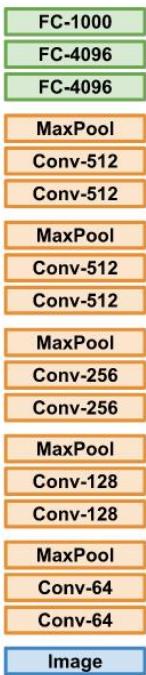


Donahue et al, "DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition", ICML 2014
Razavian et al, "CNN Features Off-the-Shelf: An Astounding Baseline for Recognition", CVPR Workshops 2014



Transfer Learning

1. Train on Imagenet

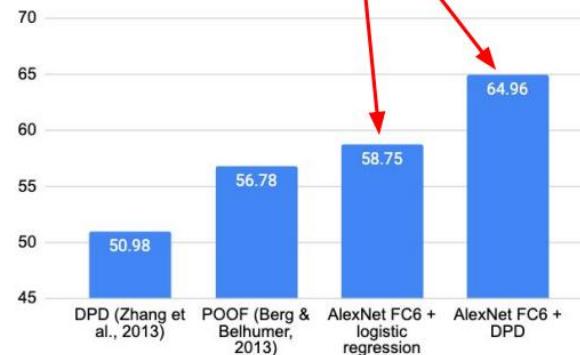


2. Small Dataset (C classes)



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

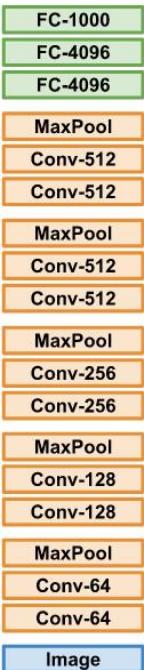
Finetuned from AlexNet



Donahue et al, "DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition", ICML 2014

Transfer Learning

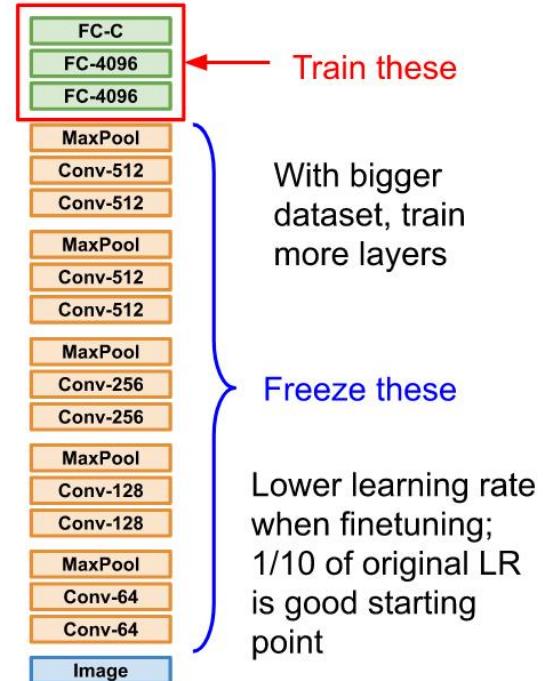
1. Train on Imagenet



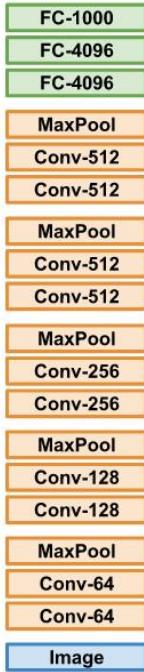
2. Small Dataset (C classes)



3. Bigger dataset



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



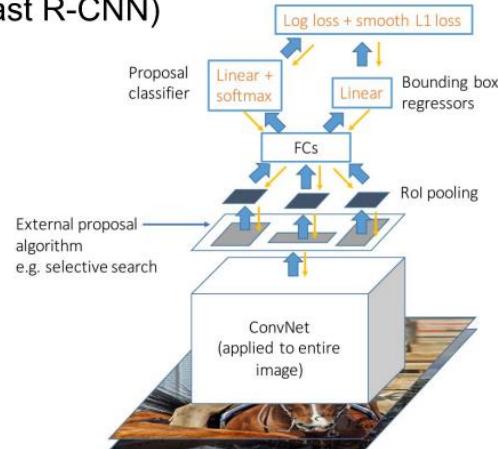
More specific

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

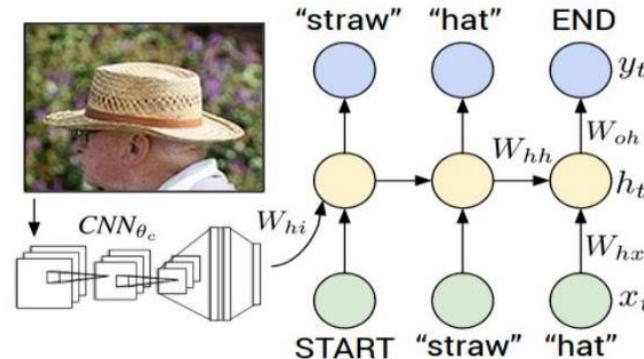
Transfer learning with CNNs is pervasive... (it's the norm, not an exception)

Object Detection (Fast R-CNN)



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

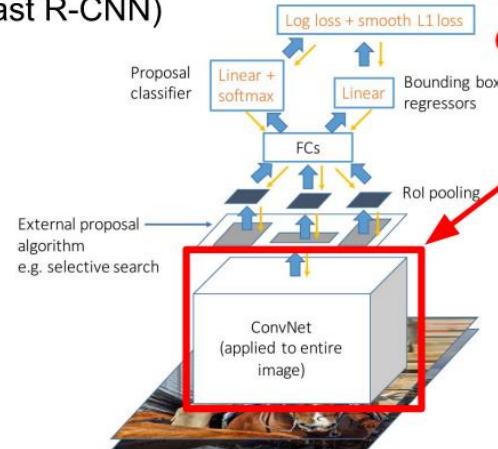
Image Captioning: CNN + RNN



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

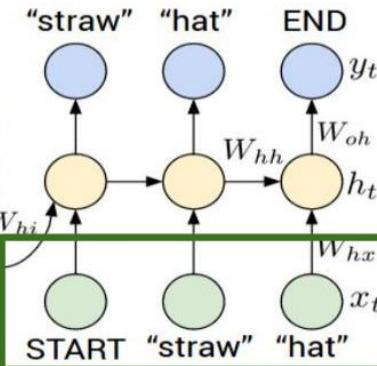
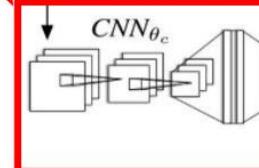
Transfer learning with CNNs is pervasive... (it's the norm, not an exception)

Object Detection
(Fast R-CNN)



CNN pretrained
on ImageNet

Image Captioning: CNN + RNN



Word vectors pretrained
with word2vec

Girshick, "Fast R-CNN", ICCV 2015
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.

