CS 4644-DL / 7643-A: LECTURE 12
DANFEI XU

Topics:

- Training Neural Networks (Part 3)

Administrative

- HW2 Due today + 2 late days
- Proposal due today (no late day)
- HW3 will be out
- Milestone will be out

SGD + Momentum:

continue moving in the general direction as the previous iterations

SGD

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

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

SGD+Momentum

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

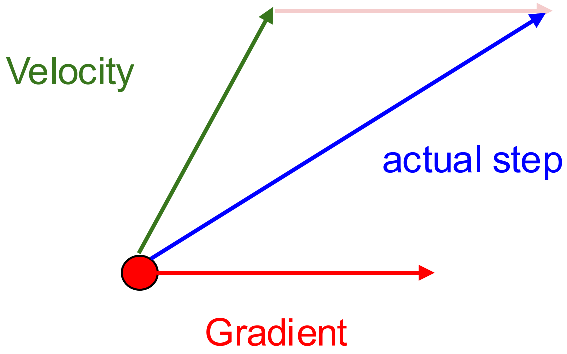
```
vx = 0
while True:
  dx = compute_gradient(x)
  vx = rho * vx + dx
  x -= learning_rate * vx
```

- Build up “velocity” as a running mean of gradients
- Rho gives “friction”; typically rho=0.9 or 0.99

Sutskever et al, “On the importance of initialization and momentum in deep learning”, ICML 2013

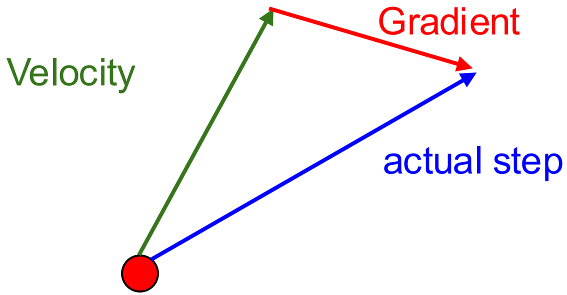
Nesterov Momentum

Momentum update:



Combine gradient at current point with velocity to get step used to update weights

Nesterov Momentum



"Look ahead" to the point where updating using velocity would take us; compute gradient there and mix it with velocity to get actual update direction

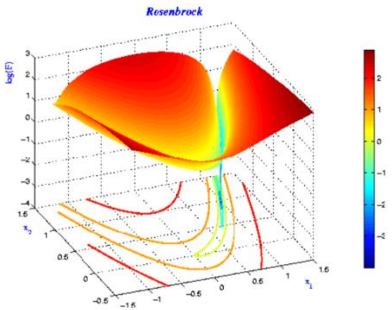
Nesterov, "A method of solving a convex programming problem with convergence rate O(1/k^2)", 1983
Nesterov, "Introductory lectures on convex optimization: a basic course", 2004
Sutskever et al, "On the importance of initialization and momentum in deep learning", ICML 2013

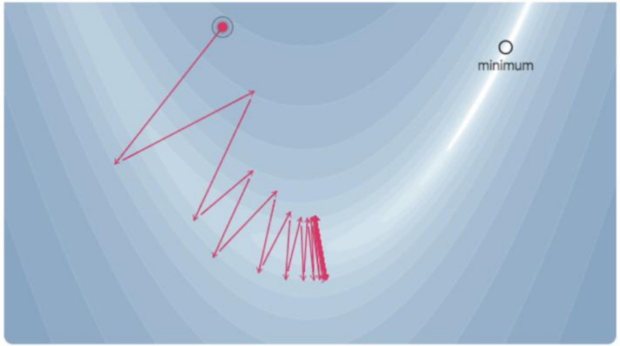
Optimization: Problem #3 with SGD

What if loss changes quickly in one direction and slowly in another?

Very slow progress along shallow dimension, jitter along steep direction

Long, narrow ravines:





https://www.cs.toronto.edu/~rgrosse/courses/csc421_2019/slides/lec07.pdf

Loss function has high **condition number**: ratio of largest to smallest eigen value ($\lambda_{max}/\lambda_{min}$) of the Hessian matrix of a loss function is large

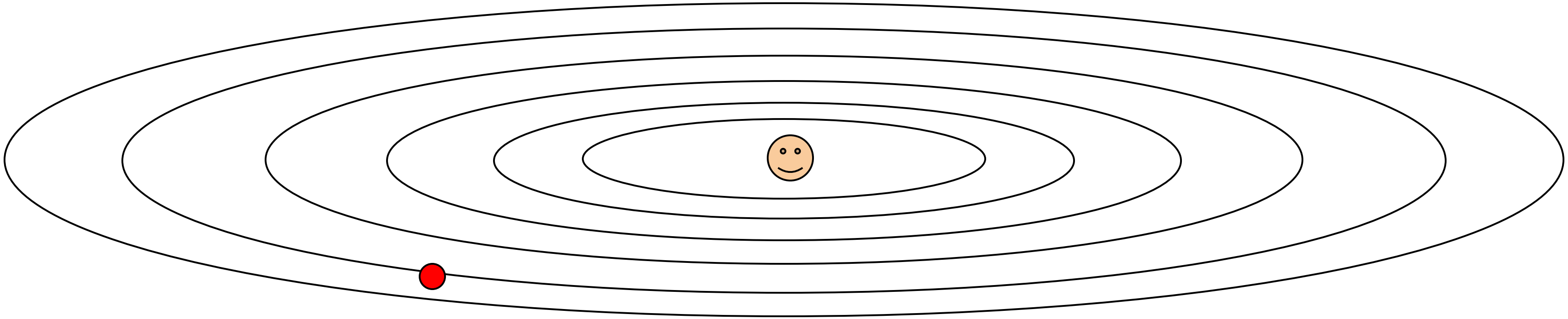
Small condition number in loss Hessian -> circular contour

Large condition number in loss Hessian -> skewed contour

Can we enable SGD to adapt to this skew-ness?

AdaGrad

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



Q2: What happens to the step size over long time? Decays to zero

RMSProp: “Leaky AdaGrad”

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

Adam (full form)

```
first_moment = 0
second_moment = 0
for t in range(1, 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))
```

Momentum

Bias correction

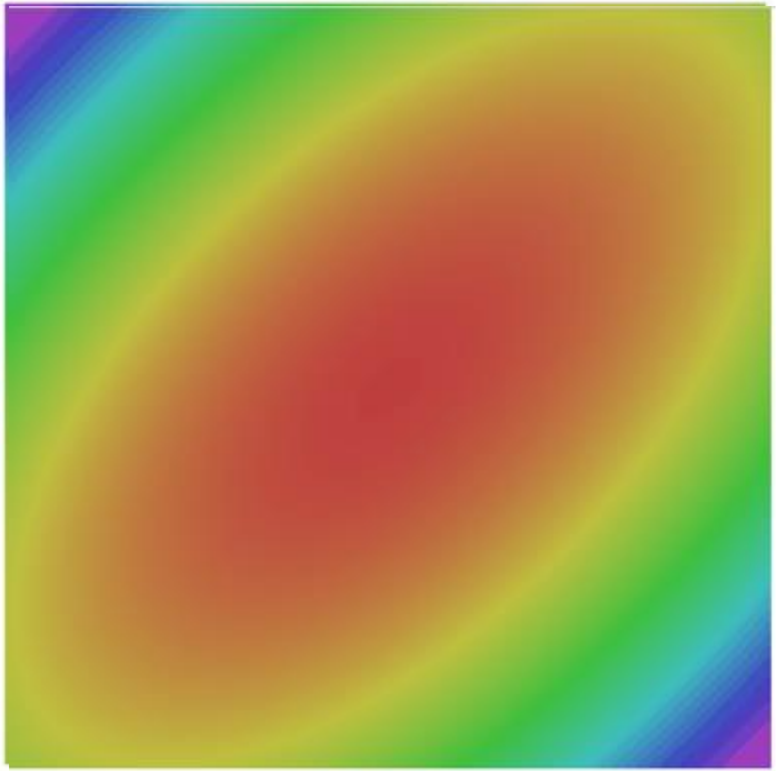
AdaGrad / RMSProp

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!

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

Adam



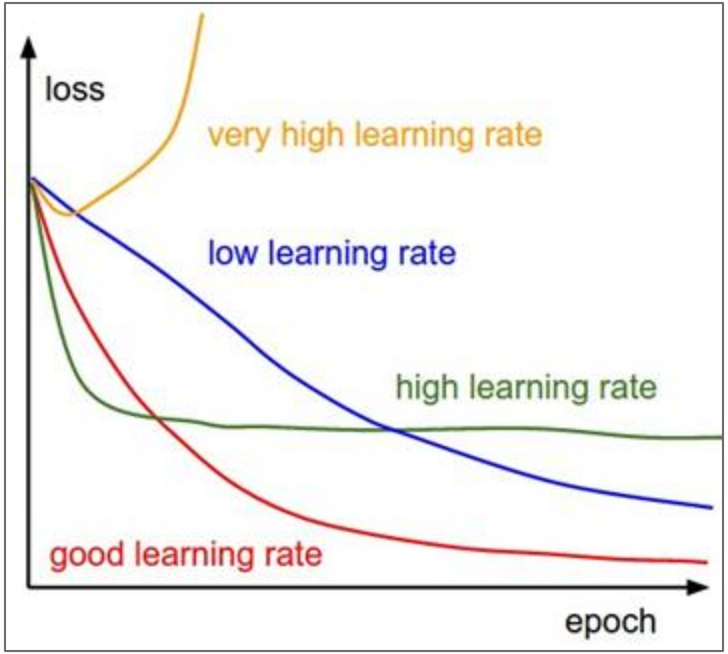
SGD

SGD+Momentum

RMSProp

Adam

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



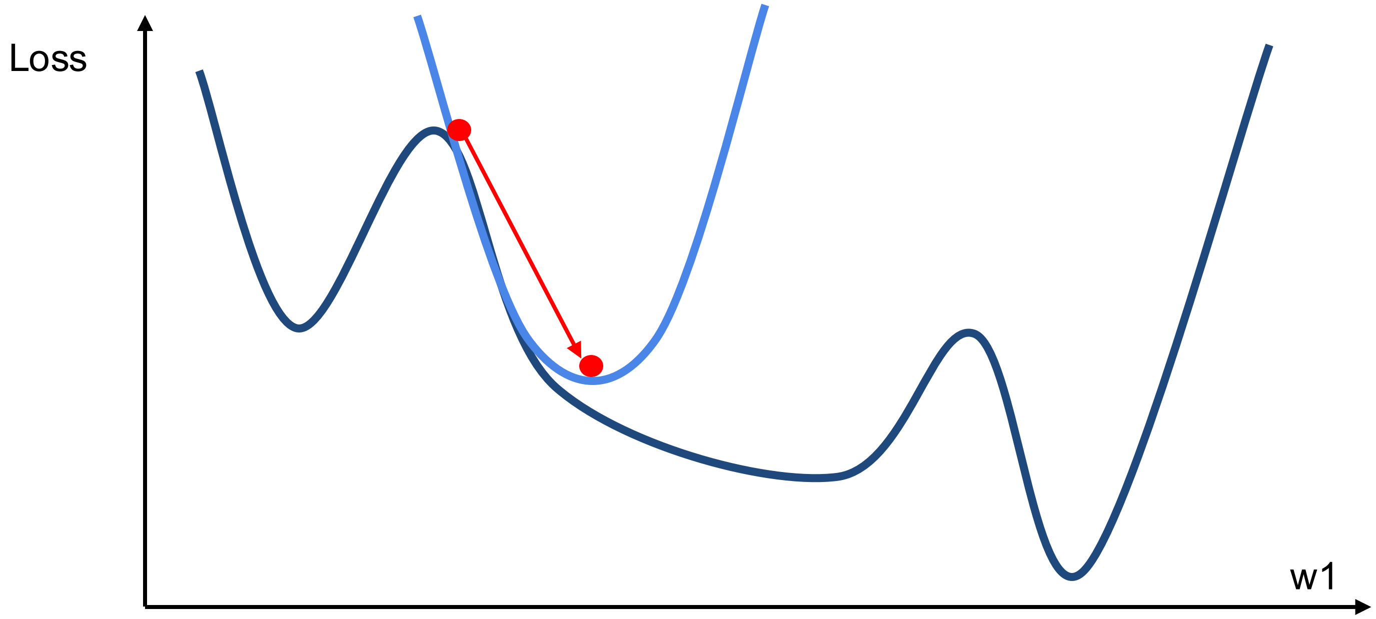
Q: Which one of these learning rates is best to use?

A: In reality, all of these are good learning rates.

Need finer adjustment closer to convergence, so we want to reduce learning rate over time to keep making progress.

Second-Order Optimization

(1) Use gradient **and Hessian** to form **quadratic** approximation
(2) Step to the **minima** of the approximation



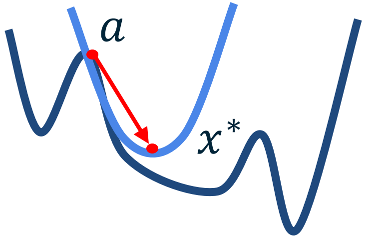
Second-Order Optimization

second-order Taylor Expansion of $f(x)$ at a:

$$f(x) = f(a) + \frac{f'(a)}{1!}(x-a) + \frac{f''(a)}{2!}(x-a)^2$$

Newton's method for optimization: solving for the critical point $f'(x) = 0$, we obtain the Newton update rule

$$f'(x) = f'(a) + f''(a)(x-a) = 0$$
$$x^* = a - \frac{1}{f''(a)} f'(a)$$



Think of a as the current params, x^* as the updated params

Second-Order Optimization

second-order Taylor expansion:

$$f(\boldsymbol{x}) = f(\boldsymbol{a}) + (x-a)^T \nabla f \; + \frac{1}{2}(x-a)^T H (x-a)$$

Solving for the critical point we obtain the Newton parameter update:

$$x^* = a - H^{-1} \nabla f$$

Q: Why is this unsuitable for deep learning?

Hessian has O(N^2) elements for N->1 functions
Inverting takes O(N^3), N = Millions

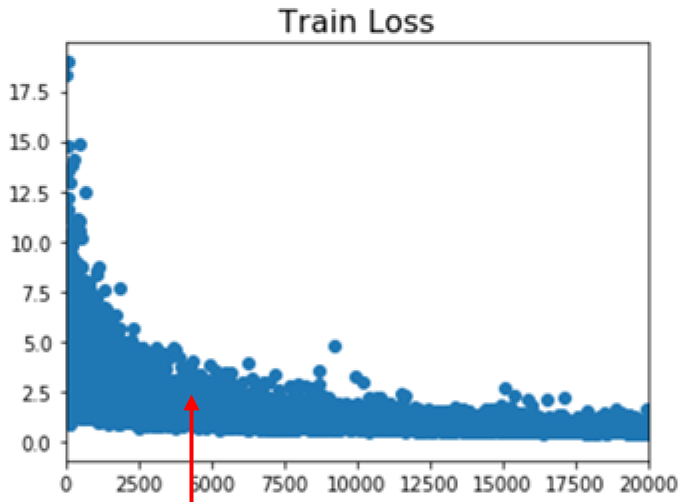
This Time:

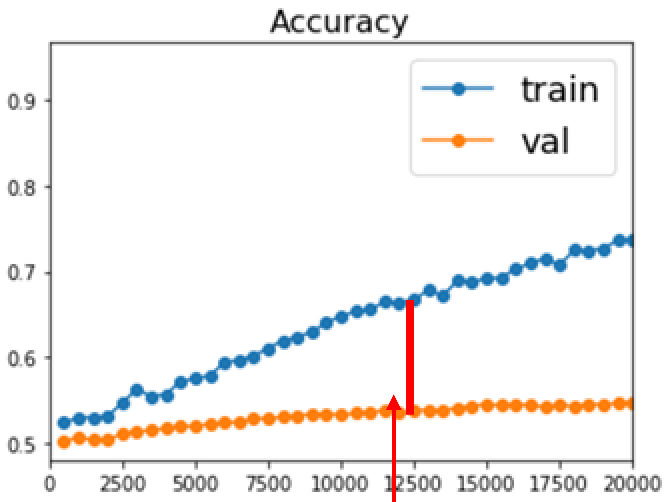
Training Deep Neural Networks

- Details of the non-linear activation functions
- Data normalization
- Weight Initialization
- Batch Normalization
- Advanced Optimization
- Regularization
- Data Augmentation
- Transfer learning
- Hyperparameter Tuning

Regularization

Beyond Training Error

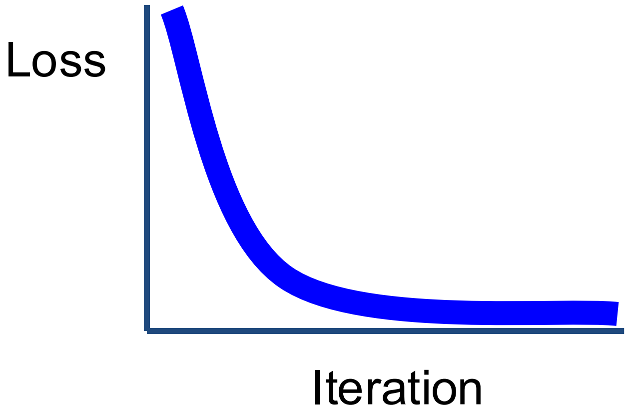


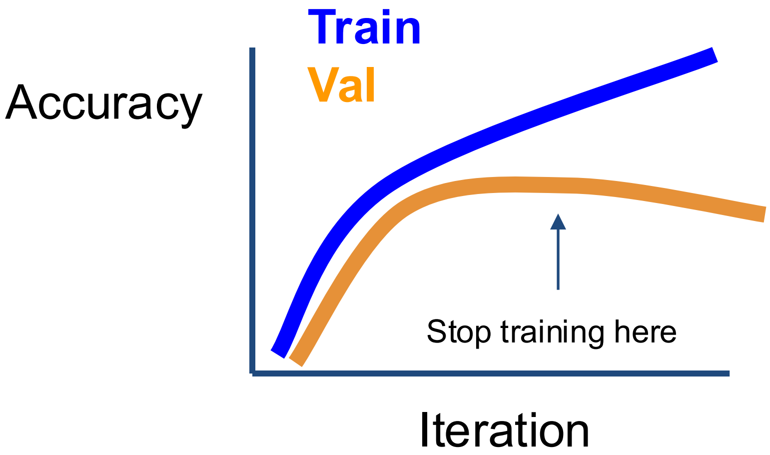


Better optimization algorithms help reduce training loss

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

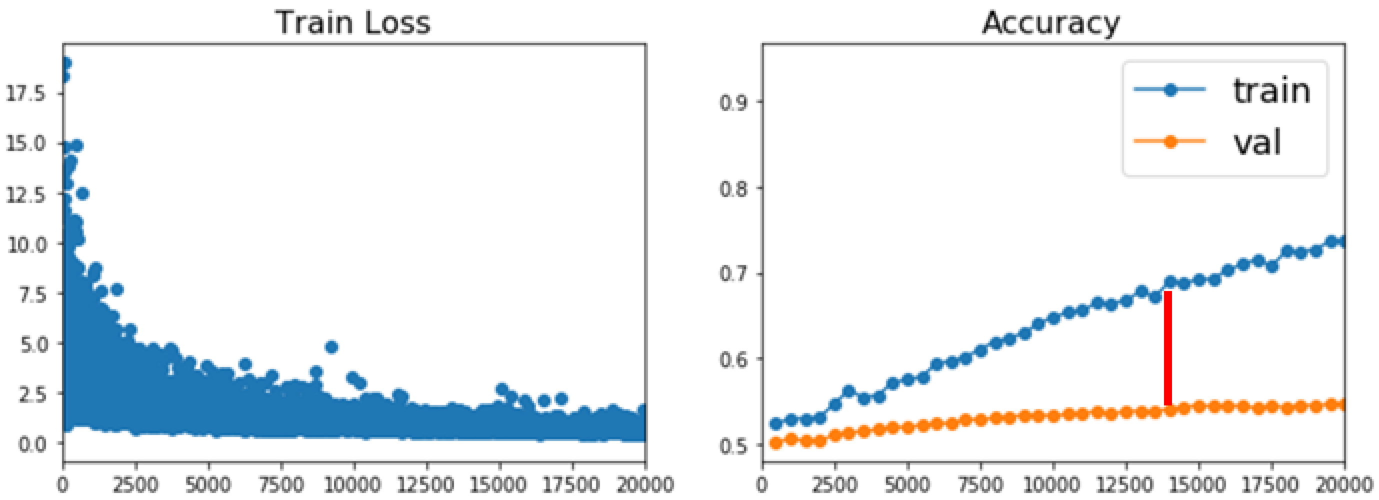
Early Stopping: Always do this





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

How to improve generalization?



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

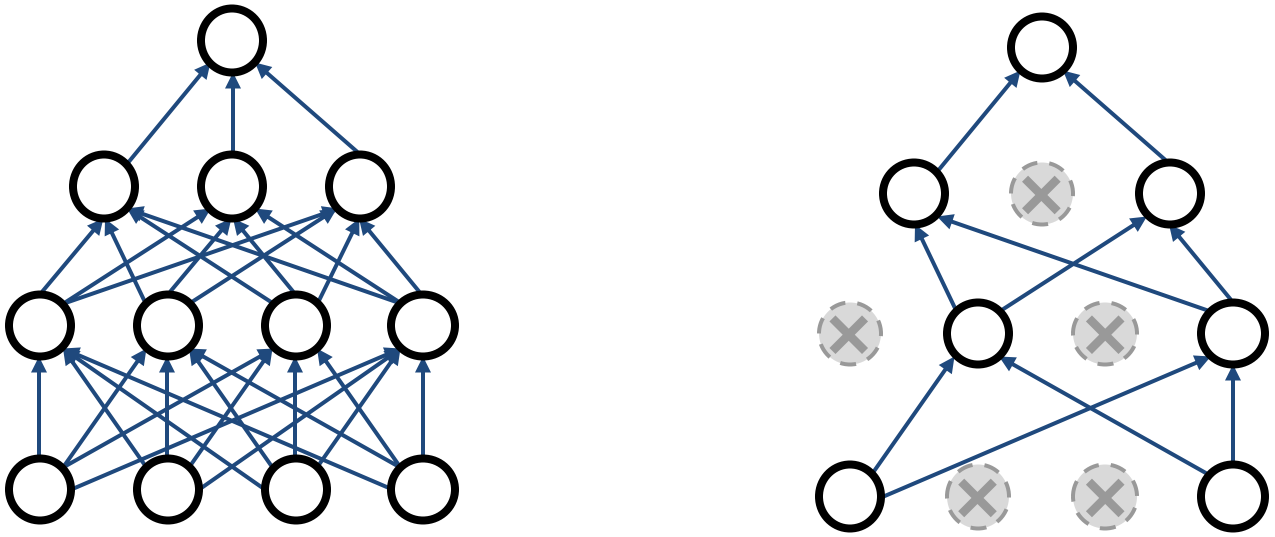
L1 regularization $R(W) = \sum_k \sum_l |W_{k,l}|$

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

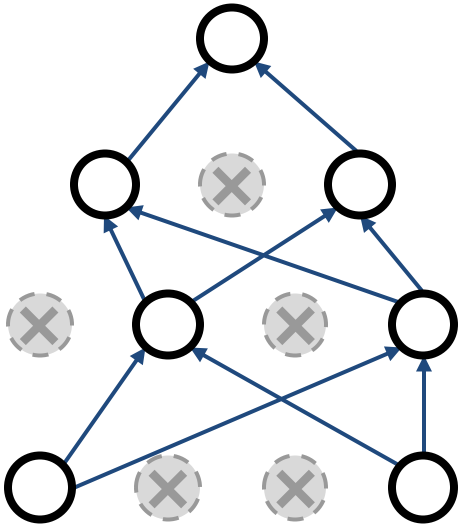


Srivastava et al, “Dropout: A simple way to prevent neural networks from overfitting”, JMLR 2014

Regularization: Dropout

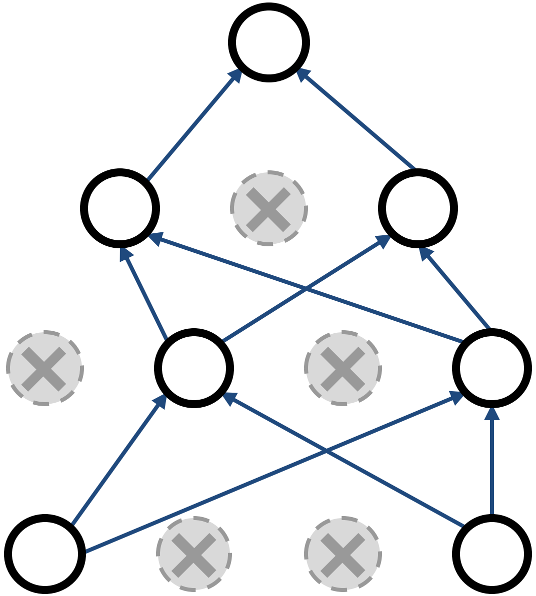
```
p = 0.5 # probability of keeping a unit active. higher = less dropout

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

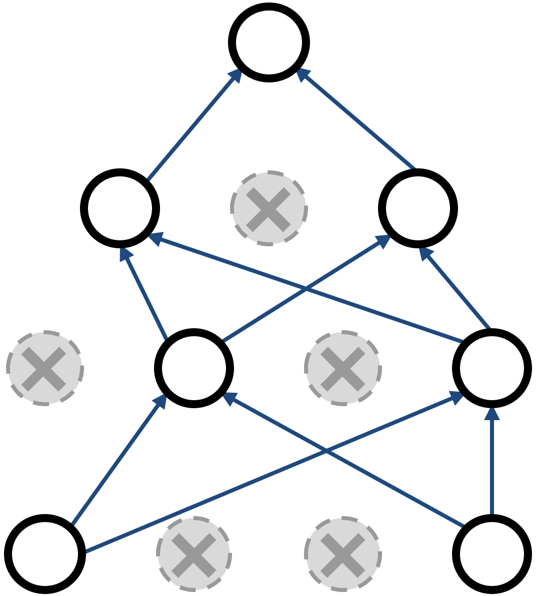
How can this possibly be a good idea?



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!
Only $\sim 10^{82}$ atoms in the universe...

Dropout: Test time

Dropout makes our output random!

Output (label) Input (image) Random mask

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

Test-time behavior should be deterministic

Want to "average out" the randomness at test-time

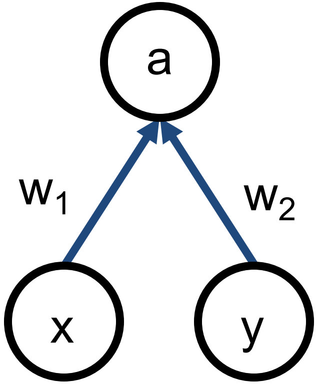
$$y = f(x) = E_z[f(x,z)] = \int p(z) f(x,z) dz$$

Dropout: Test time

Compute the expectation

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

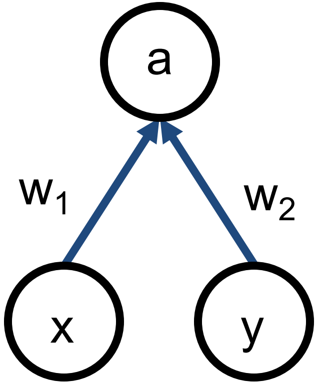
Consider a single neuron.



Dropout: Test time

Compute the expectation

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



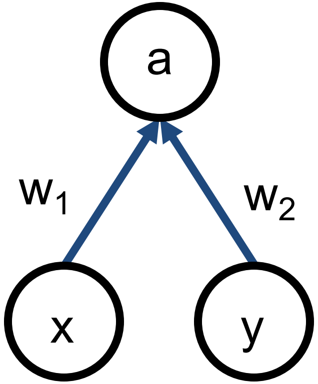
Consider a single neuron.

Without dropout: $E\left[a\right] = w_1x + w_2y$

Dropout: Test time

Compute the expectation

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



Consider a single neuron.

Without dropout: $E\big[a\big] = w_1 x + w_2 y$

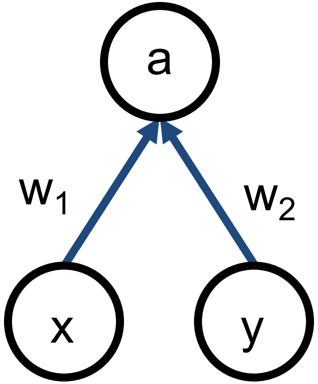
With dropout we have:

$$\begin{aligned} E\big[a\big] &= \frac{1}{4}(w_1 x + w_2 y) + \frac{1}{4}(w_1 x + 0y) \\ &\quad + \frac{1}{4}(0x + 0y) + \frac{1}{4}(0x + w_2 y) \\ &= \frac{1}{2}(w_1 x + w_2 y) \end{aligned}$$

Dropout: Test time

Compute the expectation

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



Consider a single neuron.

Without dropout: $E\left[a\right] = w_1 x + w_2 y$

With dropout we have:

$$\begin{aligned} E\left[a\right] =& \frac{1}{4}(w_1 x + w_2 y) + \frac{1}{4}(w_1 x + 0y) \\ &+ \frac{1}{4}(0x + 0y) + \frac{1}{4}(0x + w_2 y) \\ =& \frac{1}{2}(w_1 x + w_2 y) \end{aligned}$$

At test time, **scale activation** by dropout probability

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:
<u>output at test time</u> = <u>expected output at training time</u>

Dropout Summary

```python
""" 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

Regularization: A common strategy

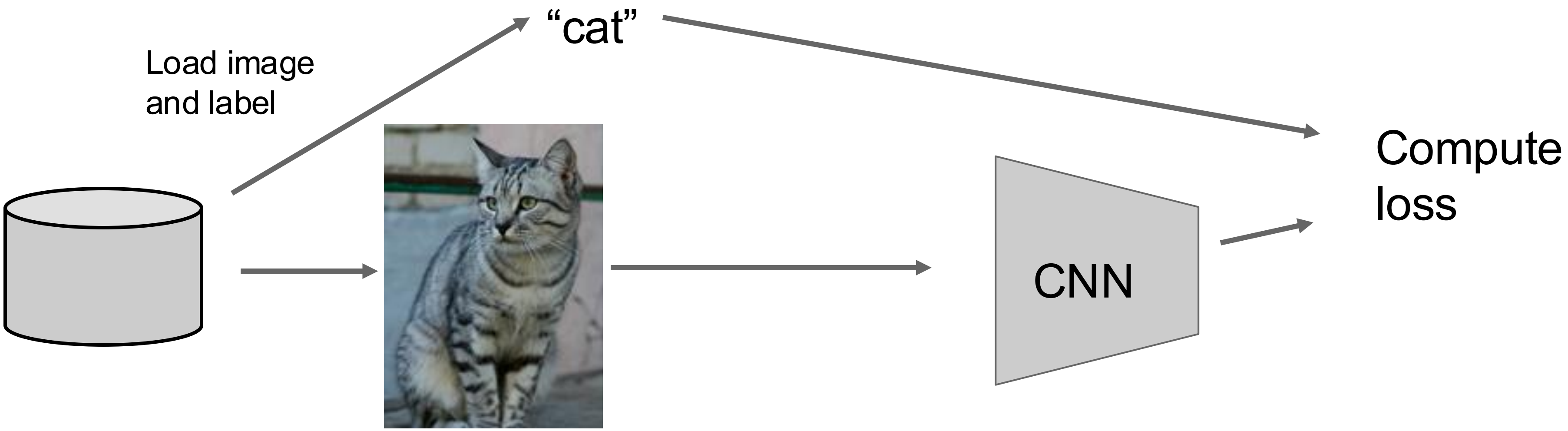
Training: Add some kind of randomness

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

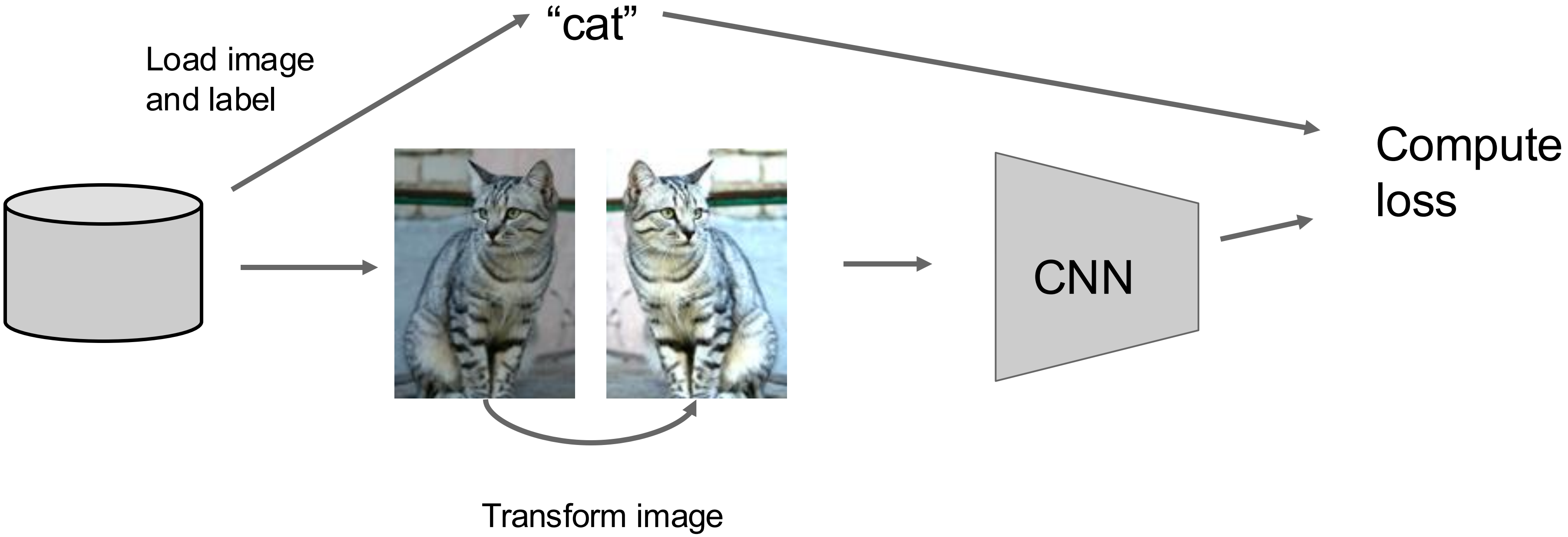
Testing: Average out randomness (sometimes approximate)

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

Regularization: Data Augmentation



Regularization: Data Augmentation



Data Augmentation
Horizontal Flips

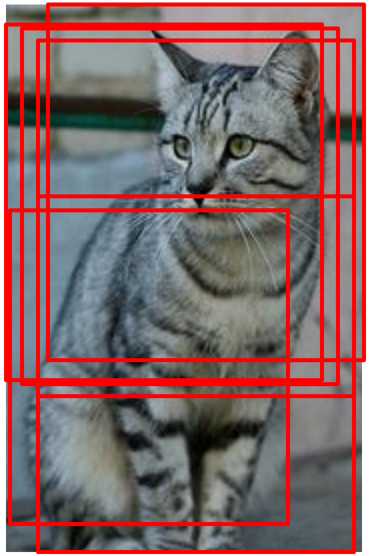
Data Augmentation

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 x 224 patch

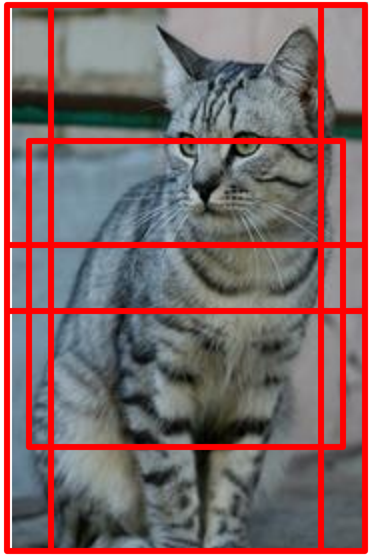


Data Augmentation
Random crops and scales

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 x 224 patch



Testing (test-time augmentation):

take votes / average from a fixed set of crops

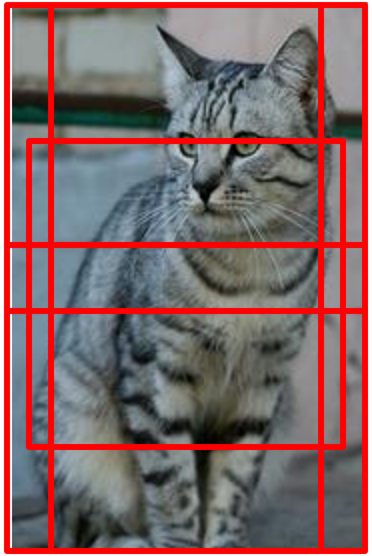
1. Resize image at 5 scales: {224, 256, 384, 480, 640}
2. For each size, use 10 224 x 224 crops: 4 corners + center, + flips
3. Make prediction on all crops, use the majority vote as the final output.

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 x 224 patch



Testing (deterministic):

- Take a center crop of 224 by 224.
- Or crop by longer dimension and resize.

Data Augmentation
Color Jitter

Simple: Randomize contrast and brightness

Data Augmentation
Color Jitter

Simple: Randomize contrast and brightness

More Complex:

1. Apply PCA to all [R, G, B] pixels in training set
2. Sample a "color offset" along principal component directions
1. Add offset to all pixels of a training image

(As seen in *[Krizhevsky et al. 2012],* ResNet, etc)

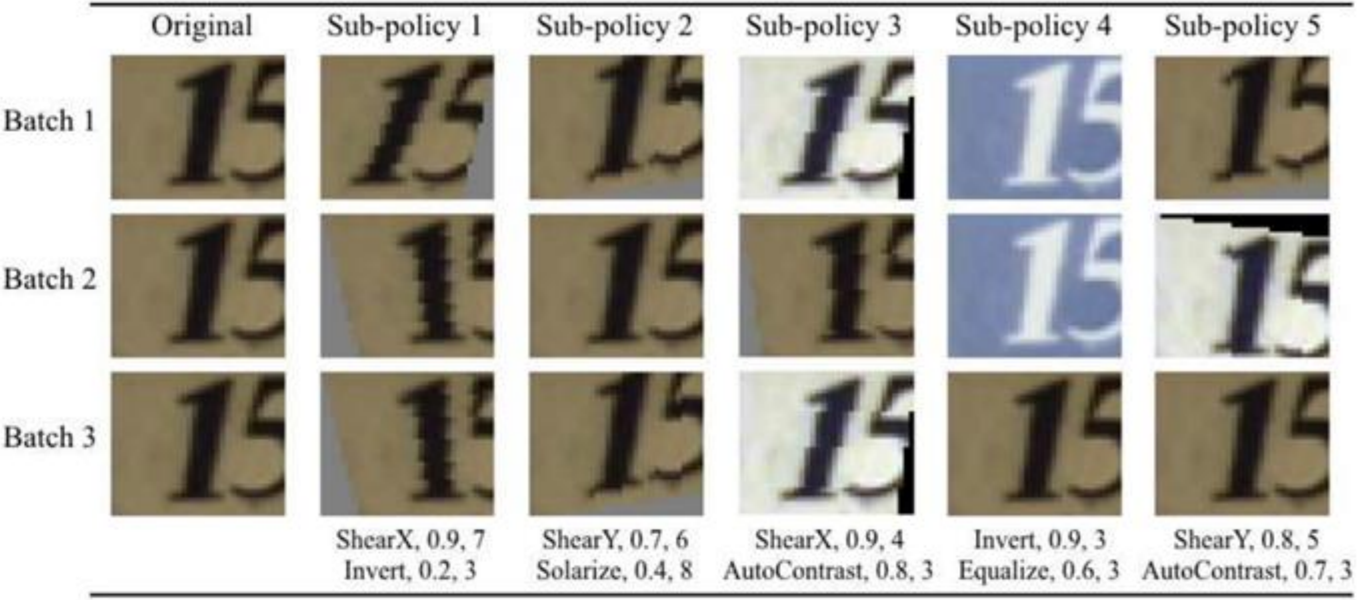
Data Augmentation

Get creative for your problem!

Examples of data augmentations:

- translation
- rotation
- stretching
- shearing,
- chromatic aberration
- lens distortions, … (go crazy)

Automatic Data Augmentation



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

Gradient clipping: prevent large gradient step

Large gradient step will likely destabilize training (gradients are noisy!)
Large gradient update can be caused by many issues, e.g., large weights, large input, bad loss function / activation function, …
Should always first try to fix the root cause (normalization, better loss / activation function, etc.)

But if all things fail … just clip the gradient

$$g_{new} = \min\left(1, \frac{\lambda}{||g||}\right) \times g$$

g: original gradient
λ: clipping threshold
If $||g|| \leq \lambda$, no effect

```
# Zero the gradients.
optimizer.zero_grad()

# Perform forward pass.
outputs = model(inputs)

# Compute the loss.
loss = loss_function(outputs, targets)

# Perform backward pass (compute gradients).
loss.backward()

# Clip the gradients.
torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)

# Update the model parameters.
optimizer.step()
```

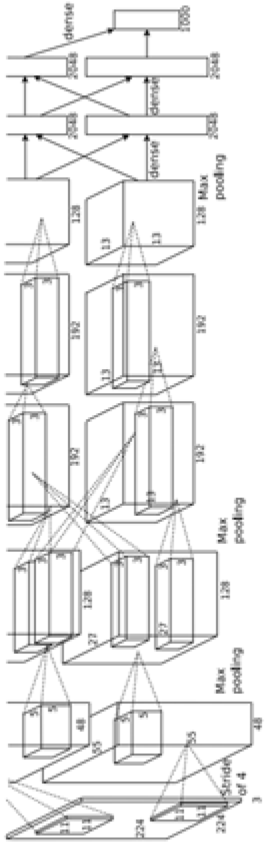
Transfer learning / Pretraining

“You need a lot of a data if you want to train/use deep neural networks”

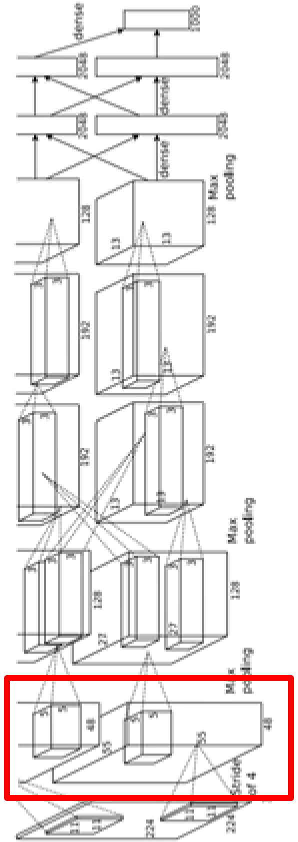
“You need a lot of a data if you want to train/use deep neural networks”

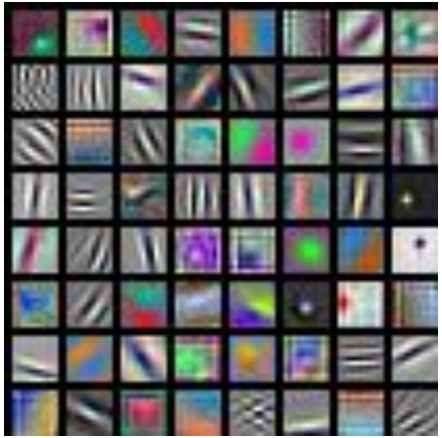
BUSTED

Transfer Learning with CNNs



Transfer Learning with CNNs

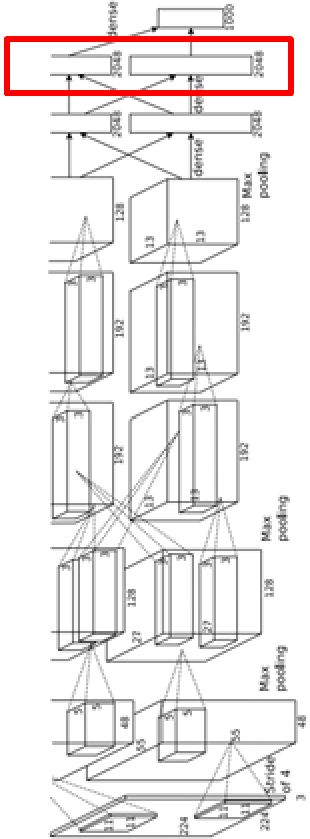




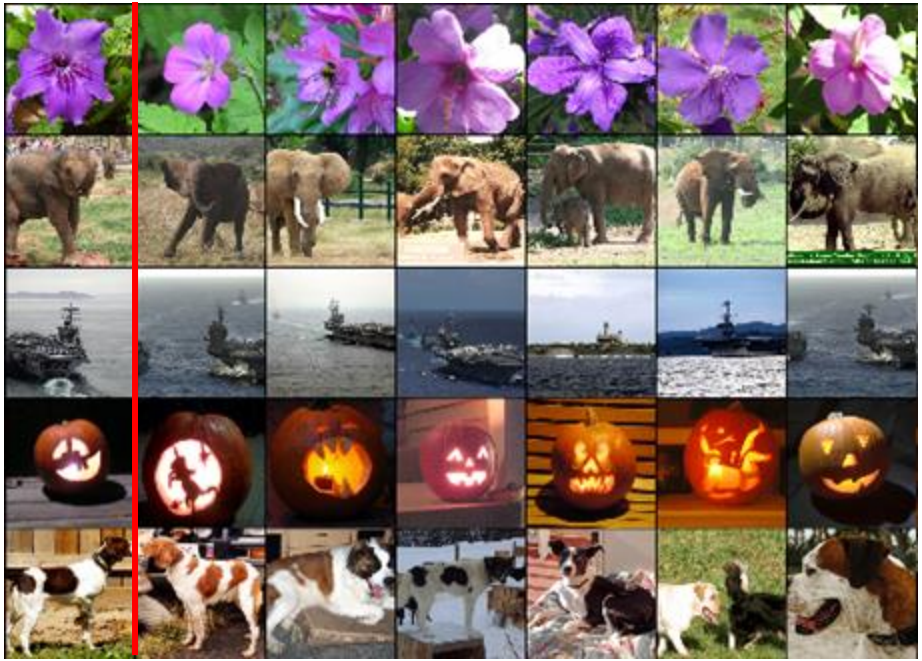
AlexNet:
64 x 3 x 11 x 11

(More on this in Lecture 13)

Transfer Learning with CNNs



Test image L2 Nearest neighbors in feature space

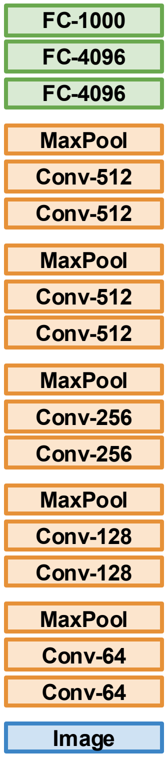


(More on this in Lecture 13)

Transfer Learning with CNNs

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

1. Train on Imagenet

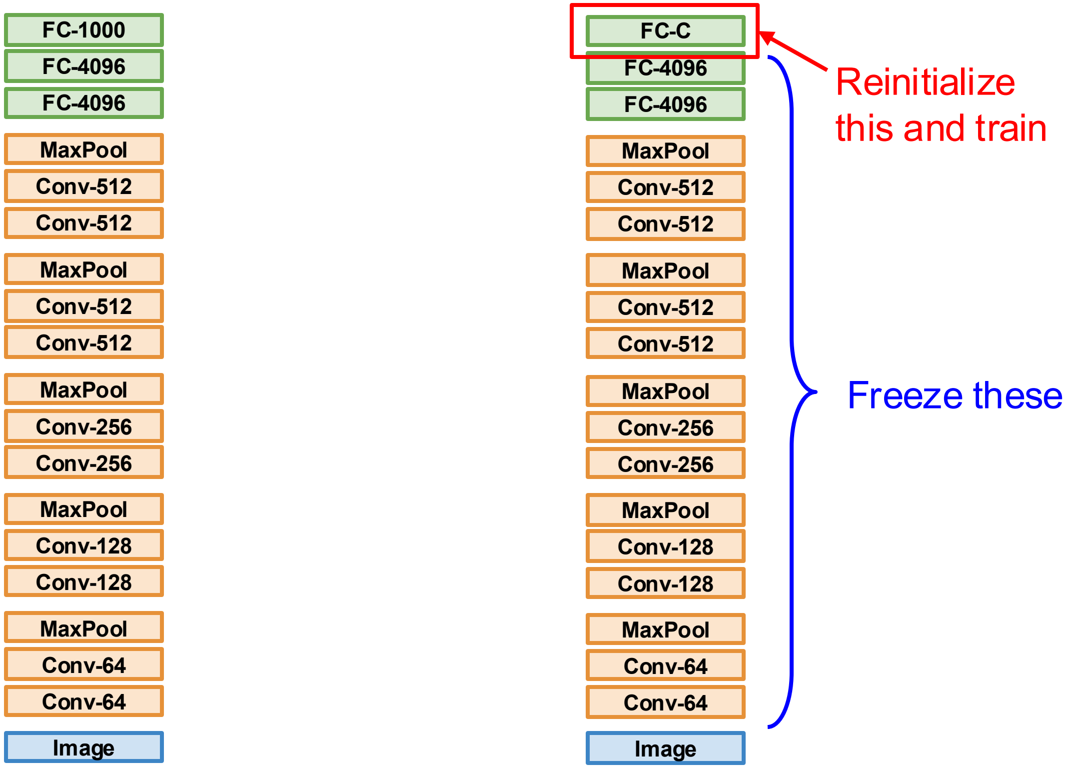


Transfer Learning with CNNs

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

1. Train on Imagenet

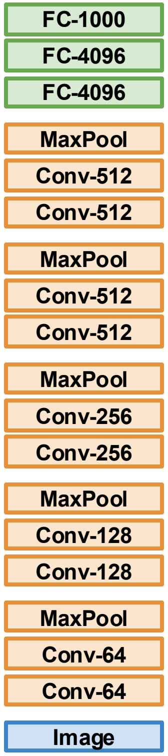
2. Small Dataset (C classes)



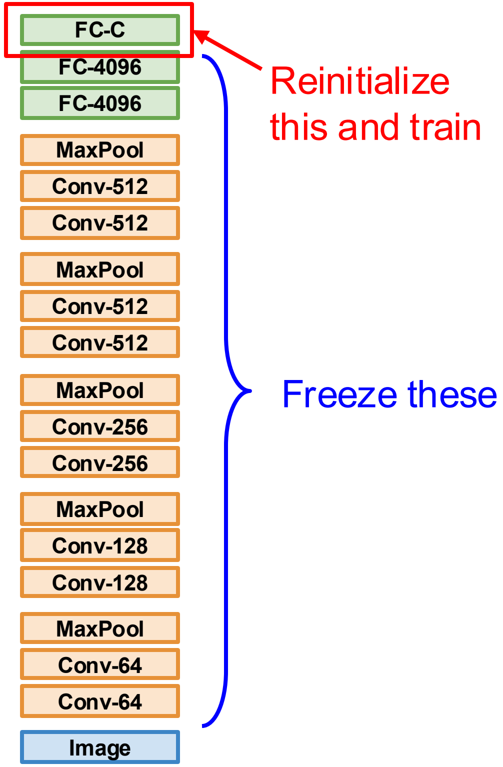
Transfer Learning with CNNs

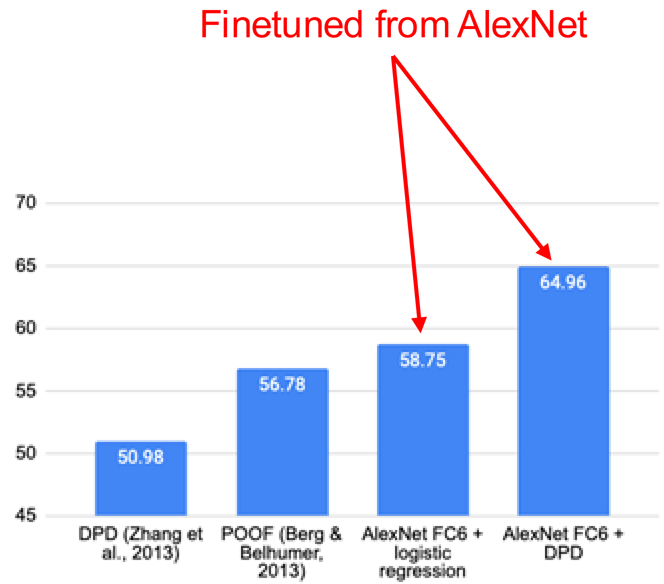
Donahue et al, "DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition", ICML 2014
Razavian et al, "CNN Features Off-the-Shelf: An Astounding Baseline for Recognition", CVPR Workshops 2014

1. Train on Imagenet



2. Small Dataset (C classes)



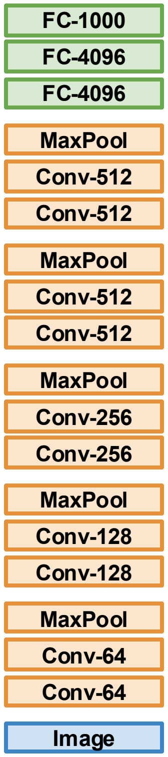


Donahue et al, "DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition", ICML 2014

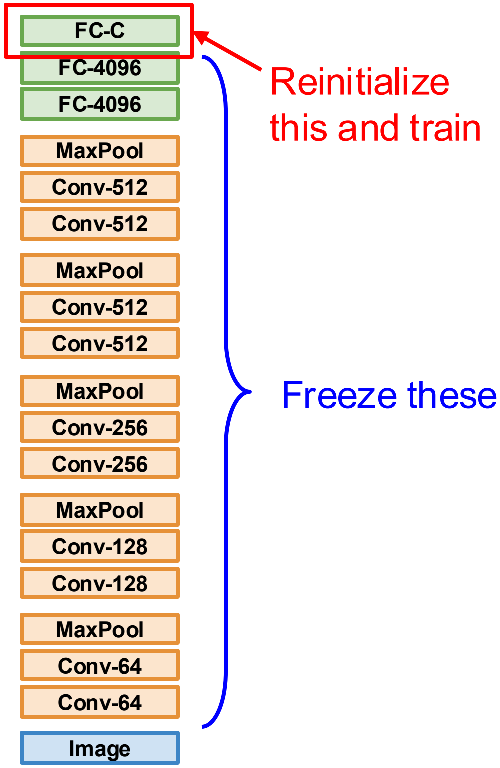
Transfer Learning with CNNs

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

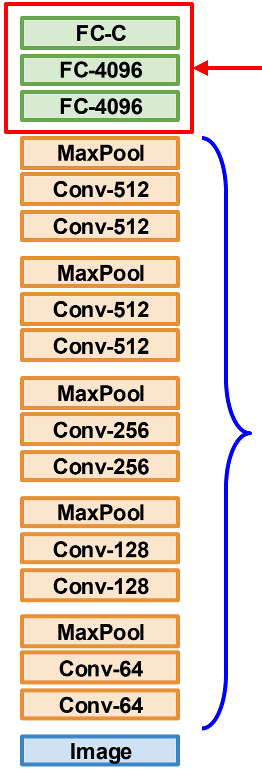
1. Train on Imagenet



2. Small Dataset (C classes)



3. Bigger dataset

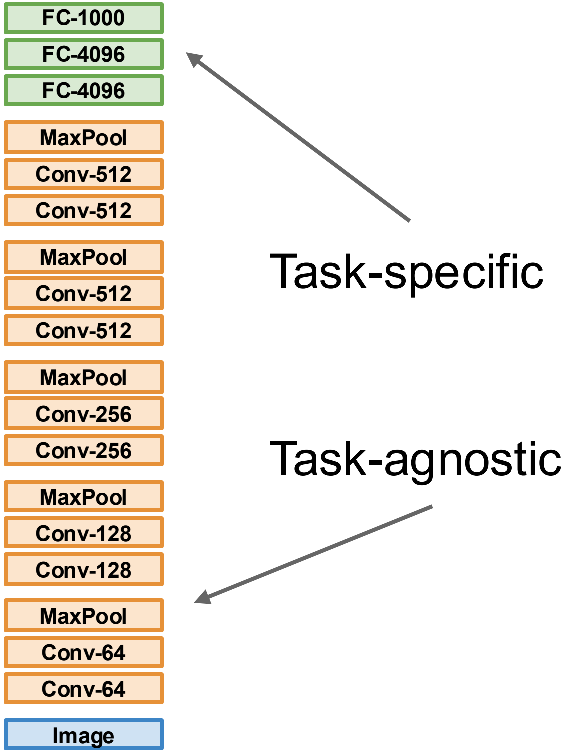


Train these

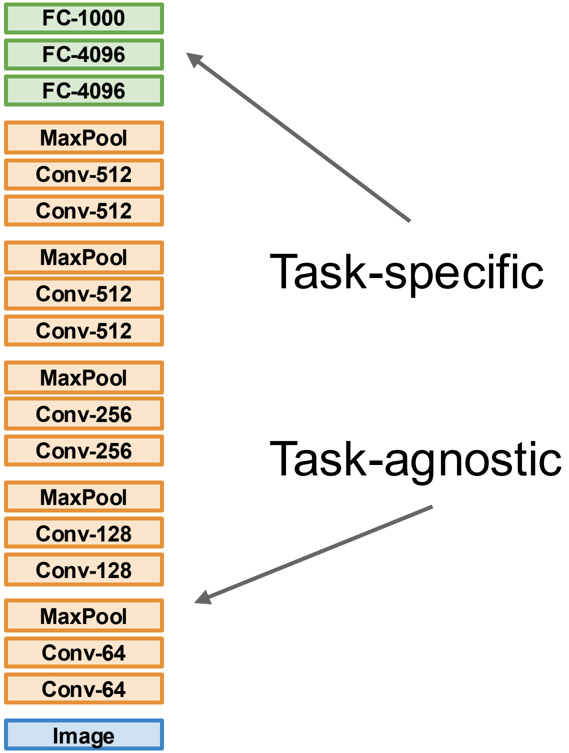
With bigger dataset, train more layers

Freeze these

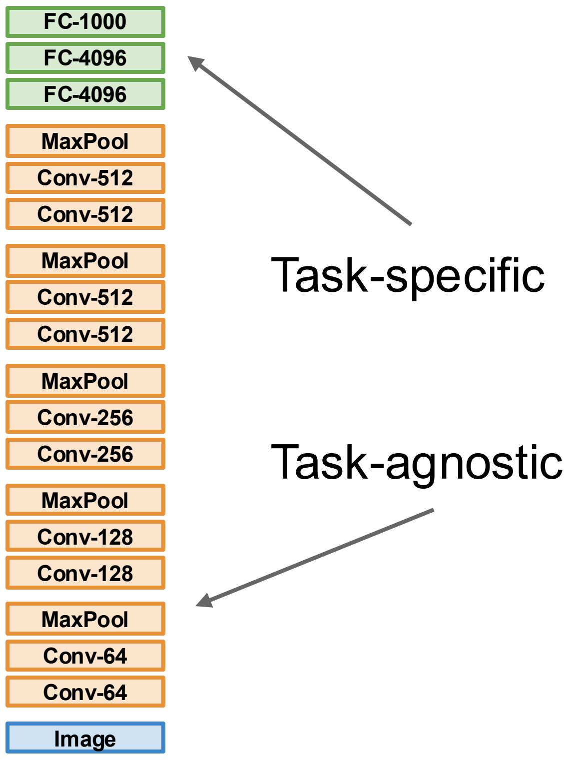
Lower learning rate when finetuning; 1/10 of original LR is good starting point



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



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



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

Transfer learning is pervasive…

(it's the norm, not an exception)

Object Detection
(Fast R-CNN)

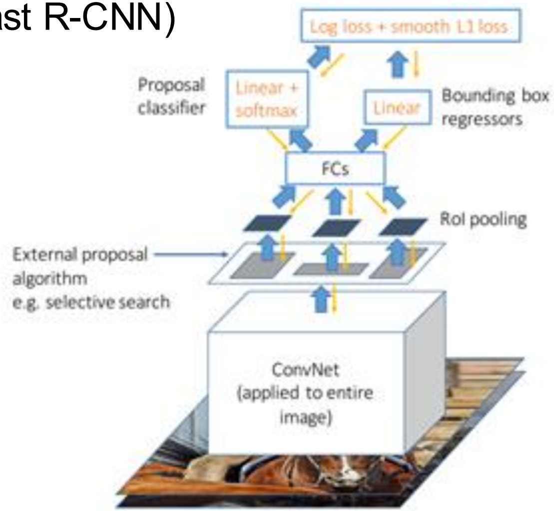
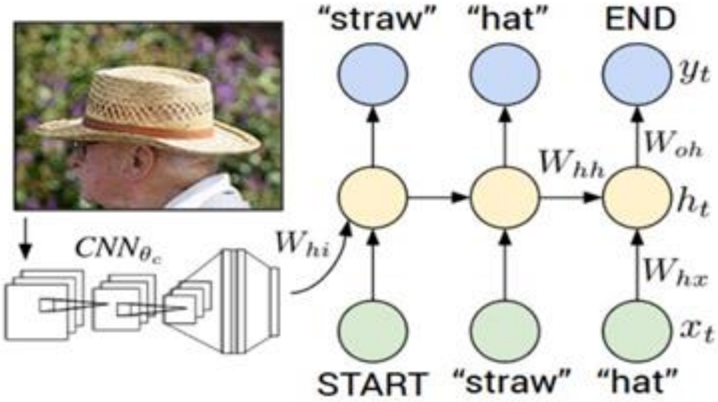


Image Captioning: CNN + RNN

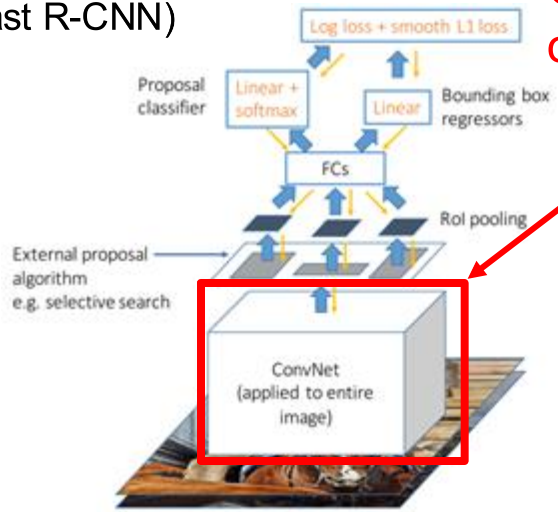


Girshick, "Fast R-CNN", ICCV 2015

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

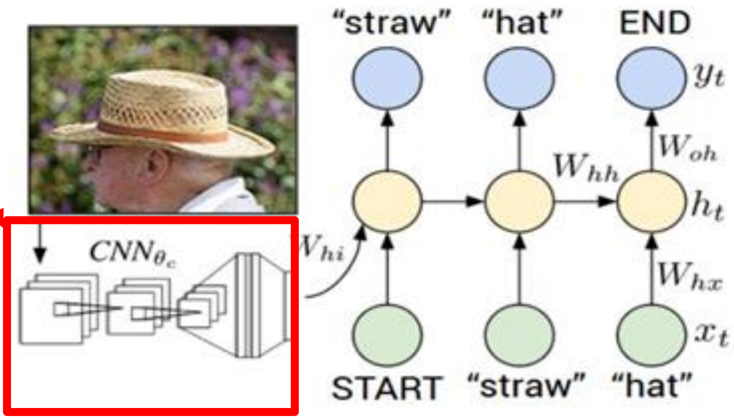
Transfer learning is pervasive…
(it’s the norm, not an exception)

Object Detection
(Fast R-CNN)



CNN pretrained on ImageNet

Image Captioning: CNN + RNN

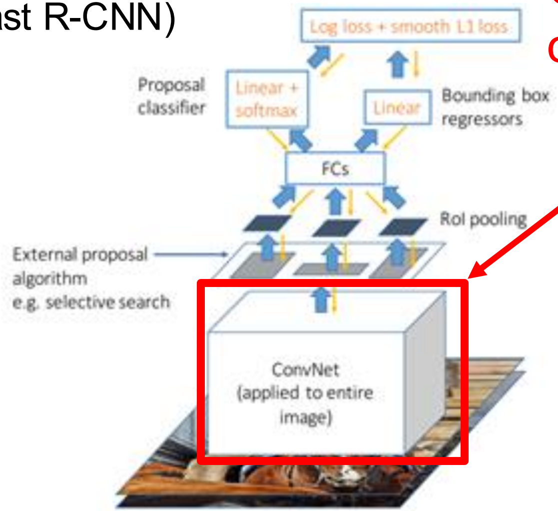


Girshick, “Fast R-CNN”, ICCV 2015

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

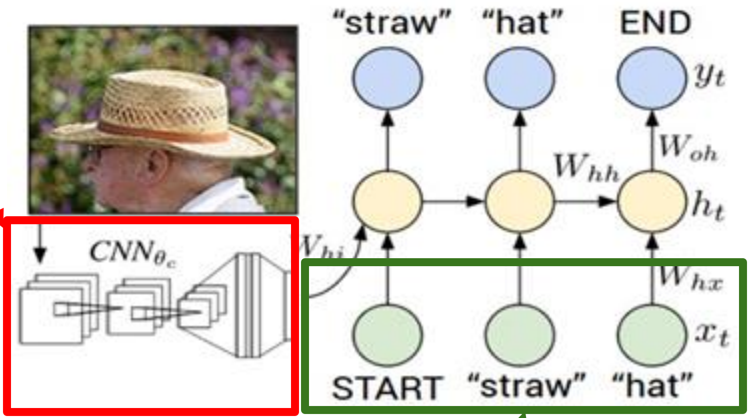
Transfer learning is pervasive…
(it's the norm, not an exception)

Object Detection
(Fast R-CNN)



CNN pretrained on ImageNet

Image Captioning: CNN + RNN

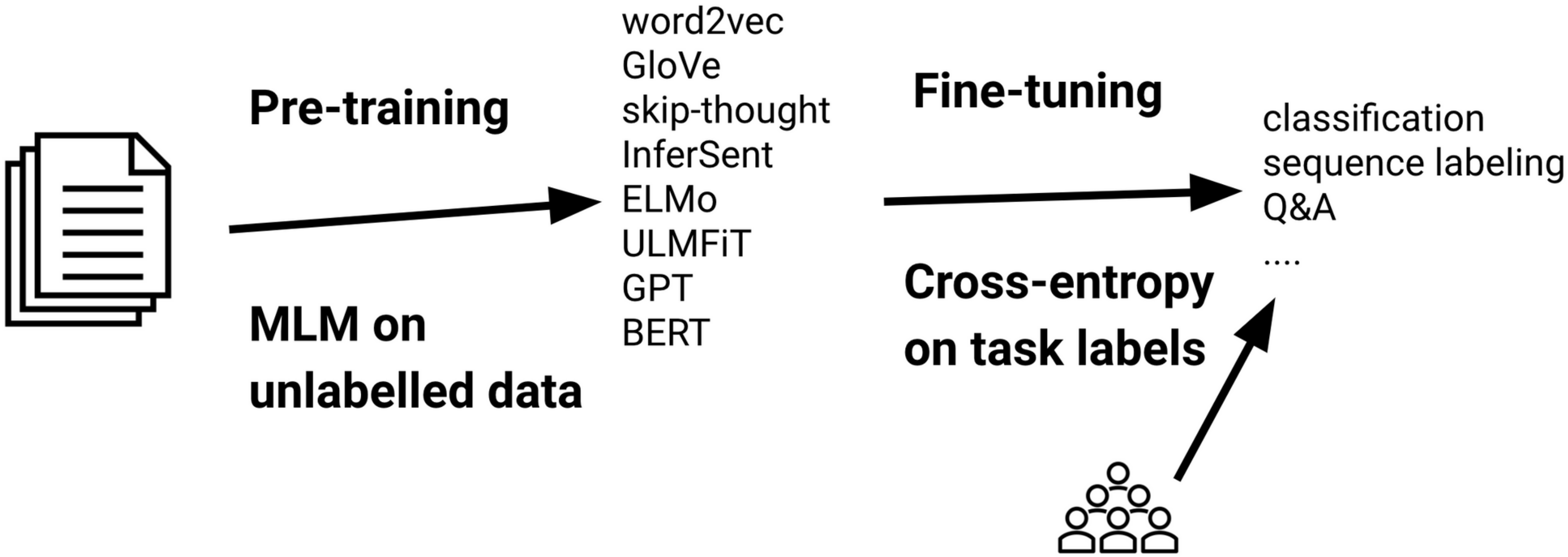


Word vectors pretrained
with word2vec

Girshick, "Fast R-CNN", ICCV 2015

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

Transfer learning is pervasive…
(it’s the norm, not an exception)

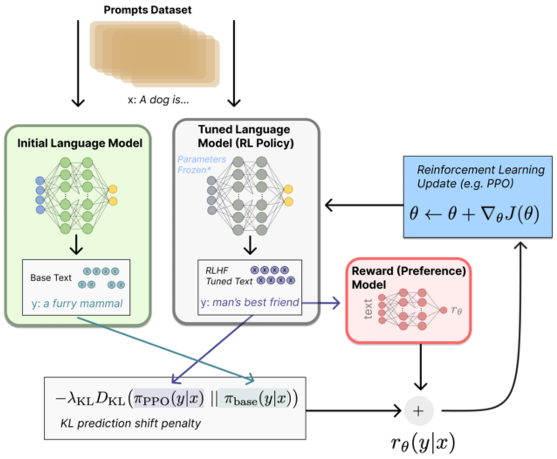


Generic Language Model **Train with Task-specific Labels**

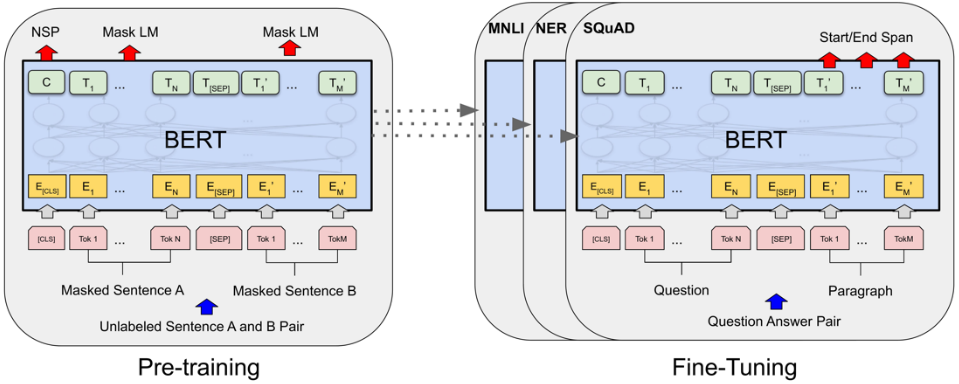
https://ruder.io/recent-advances-lm-fine-tuning/

Preview: Pretrained Language Models

“Generative Pretrained Transformer”



https://huggingface.co/blog/rlhf



Devlin et al. in BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding, 2019

Preview: Self-Supervised Pretraining
(pretraining tasks that do not need labels)

Example: learn to predict image transformations / complete corrupted images

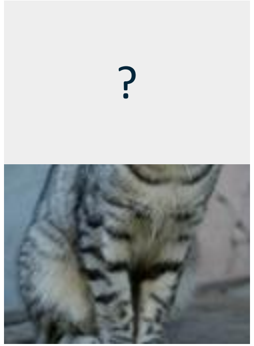
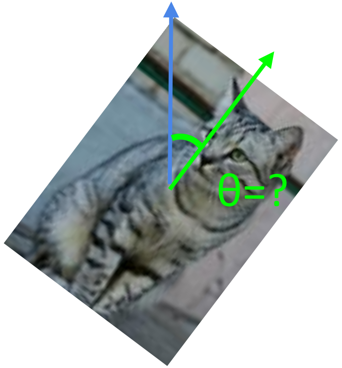


image completion



rotation prediction

"jigsaw puzzle"

colorization

1. Solving the pretext tasks allow the model to learn good features.
2. We can automatically generate labels for the pretext tasks.

Preview: Low-rank finetuning (LORA)

quickly finetune a billion-parameter model

Problem: finetuning still takes a lot of data, especially if the model is huge and/or the domain gap is large.
Fact: finetuning is just adding a W_δ to the existing weight matrix W, i.e., $W^* = W + W_\delta$
Hypothesis: W_δ is *low-rank*, meaning that W_δ can be decomposed into two smaller matrices A and B, i.e., $W_\delta = A^T B$.
So what?: A and B have a lot fewer parameters than the full W**.** Requires less data and faster to train.

Hu, Edward J., et al. "Lora: Low-rank adaptation of large language models.", 2021

Takeaway for your projects and beyond:

1. Find a very large dataset that has similar data, train a big model there (or start with a pretrained model)
2. Transfer learn to your dataset
3. Try LORA (low-rank finetuning) if necessary

Deep learning frameworks provide a "Model Zoo" of pretrained models so you don't need to train your own
TensorFlow: https://github.com/tensorflow/models
PyTorch (Vision): https://github.com/pytorch/vision
PyTorch (NLP): https://github.com/pytorch/text

Diagnose your training

(without tons of GPUs)

Diagnose your training

Step 1: Check initial loss

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

Reminder: $L = -\log p = -\log(1/C) = \log(C)$

Diagnose your training

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, bug in code or errors in training labels
Loss explodes to Inf or NaN? LR too high, bad initialization, bug in code

Diagnose your training

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-3, 3e-4, 1e-4

Diagnose your training

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

Diagnose your training

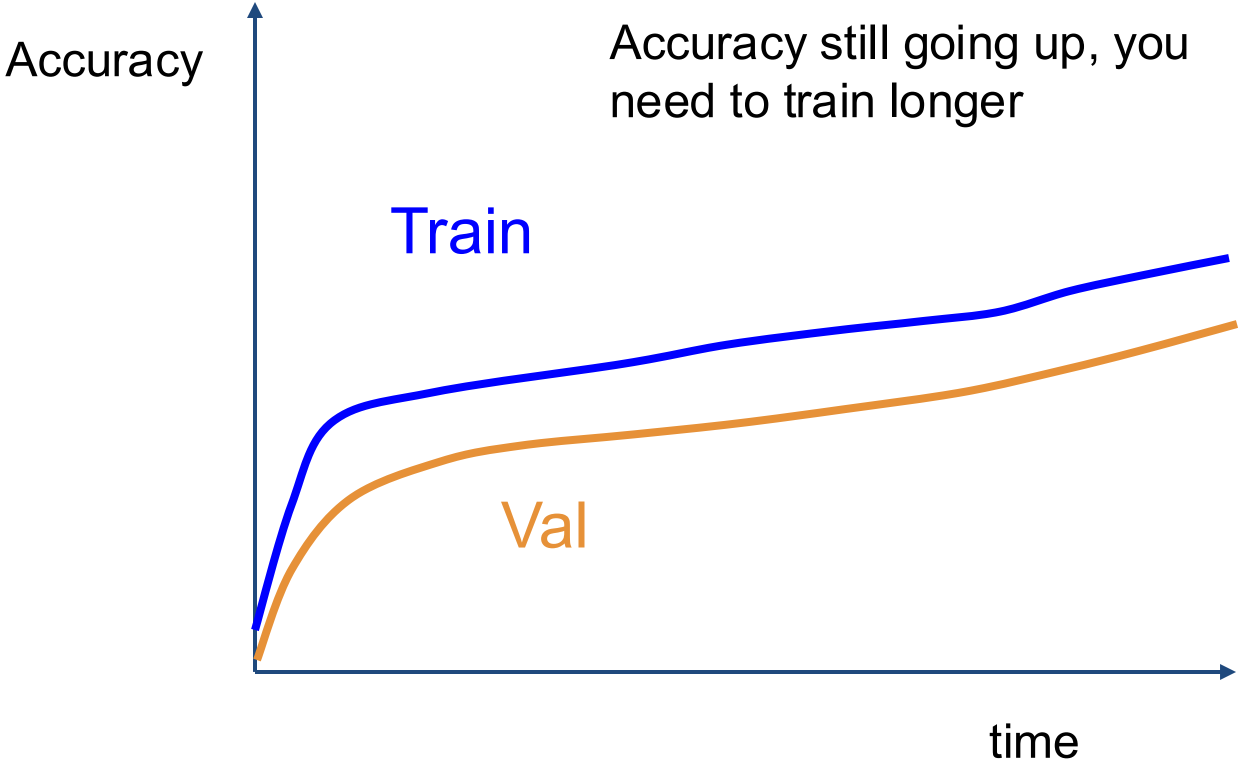
Step 1: Check initial loss
Step 2: Overfit a small sample
Step 3: Find LR that makes loss go down
Step 4: Coarse grid, train for ~1-5 epochs
Step 5: Refine grid, train longer

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

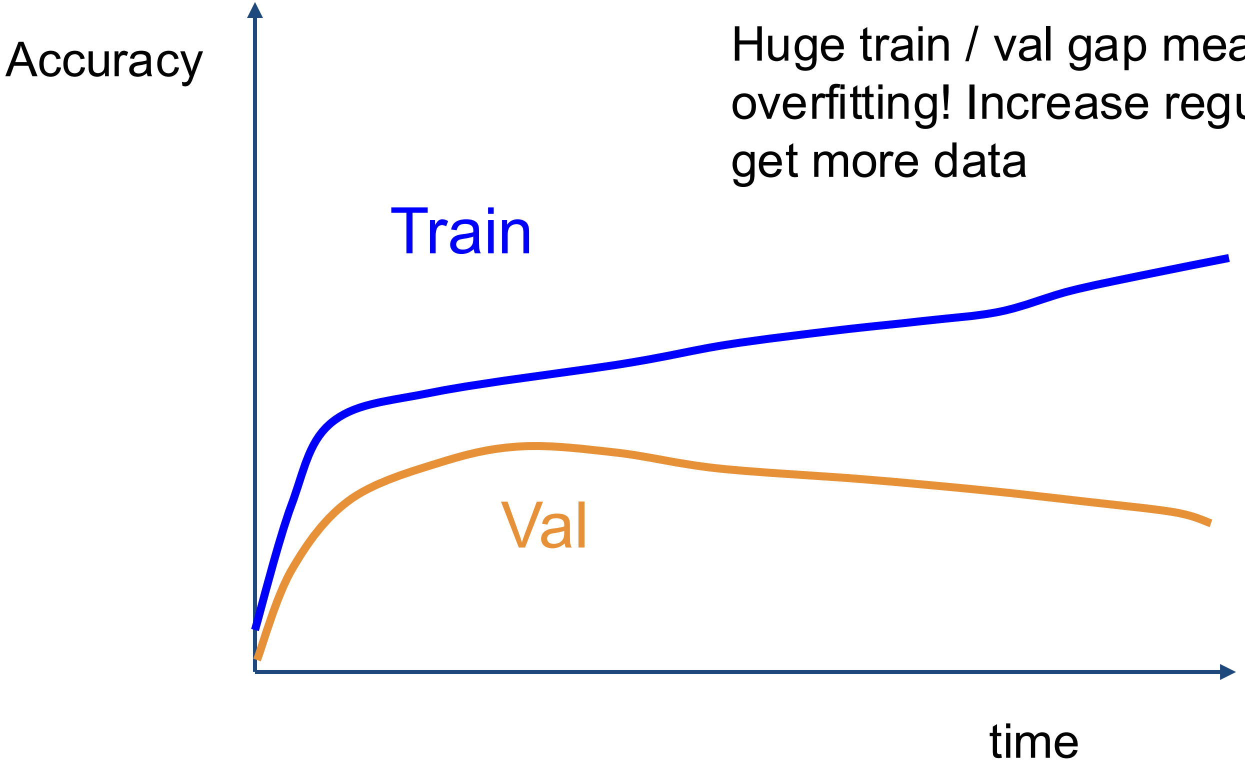
Diagnose your training

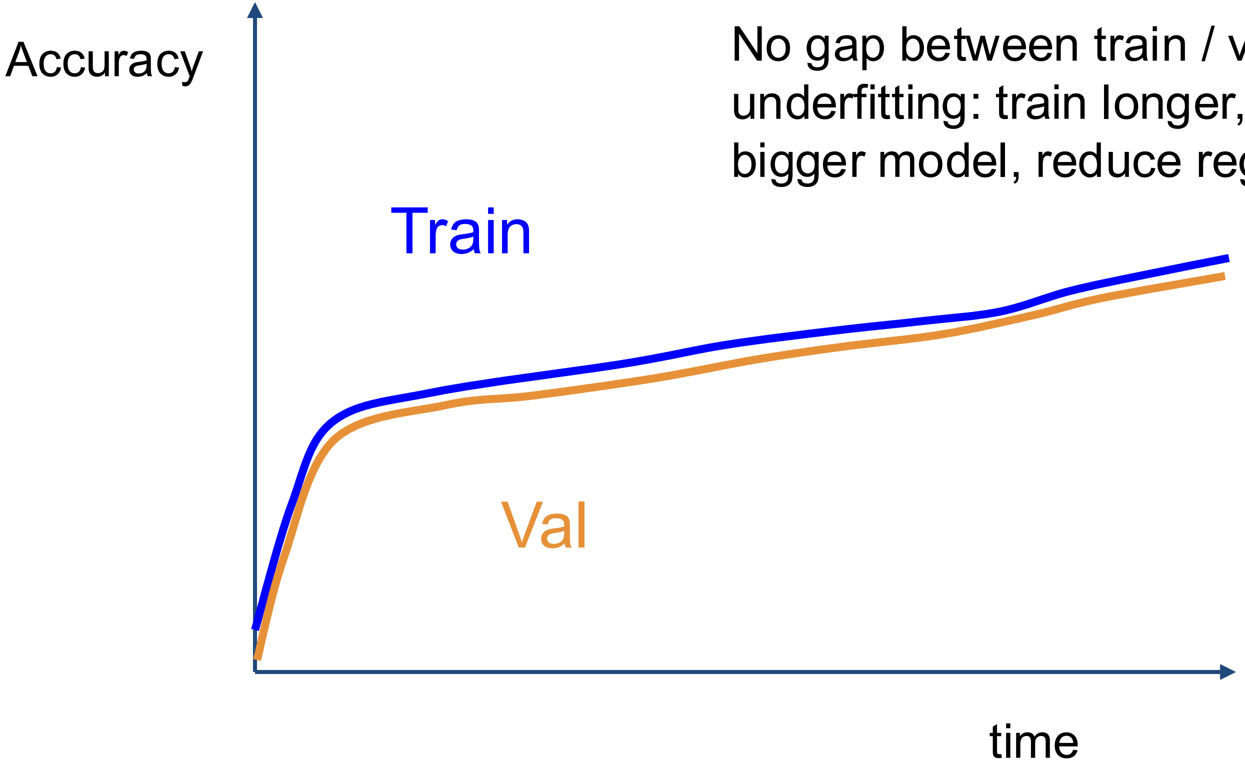
Step 1: Check initial loss
Step 2: Overfit a small sample
Step 3: Find LR that makes loss go down
Step 4: Coarse grid, train for ~1-5 epochs
Step 5: Refine grid, train longer
Step 6: Look at loss and accuracy curves

Accuracy still going up, you need to train longer



Huge train / val gap means overfitting! Increase regularization, get more data

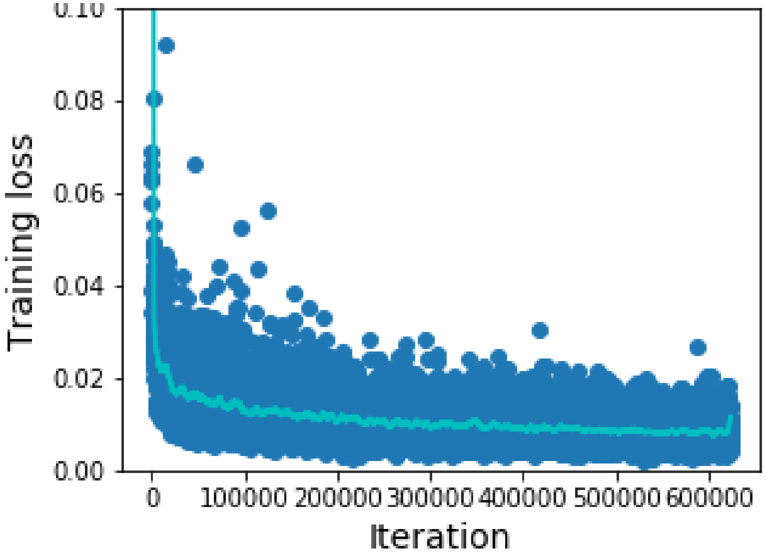




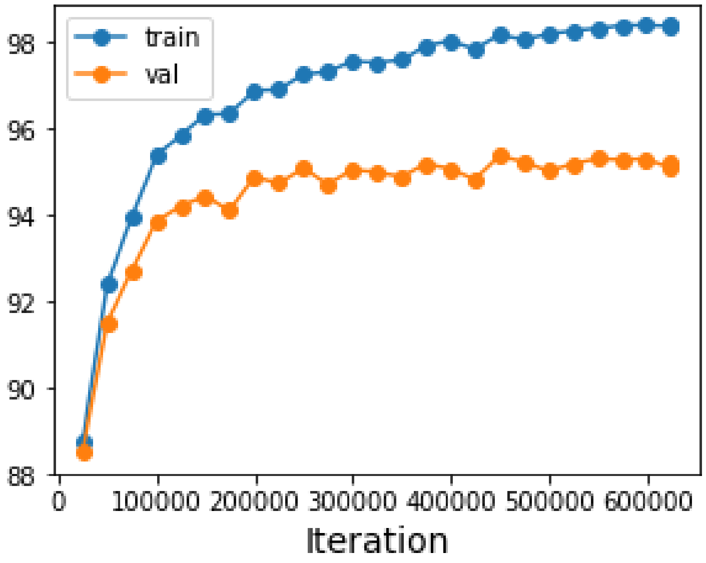
No gap between train / val means underfitting: train longer, use a bigger model, reduce regularization

Look at learning curves!

Training Loss



Train / Val Accuracy



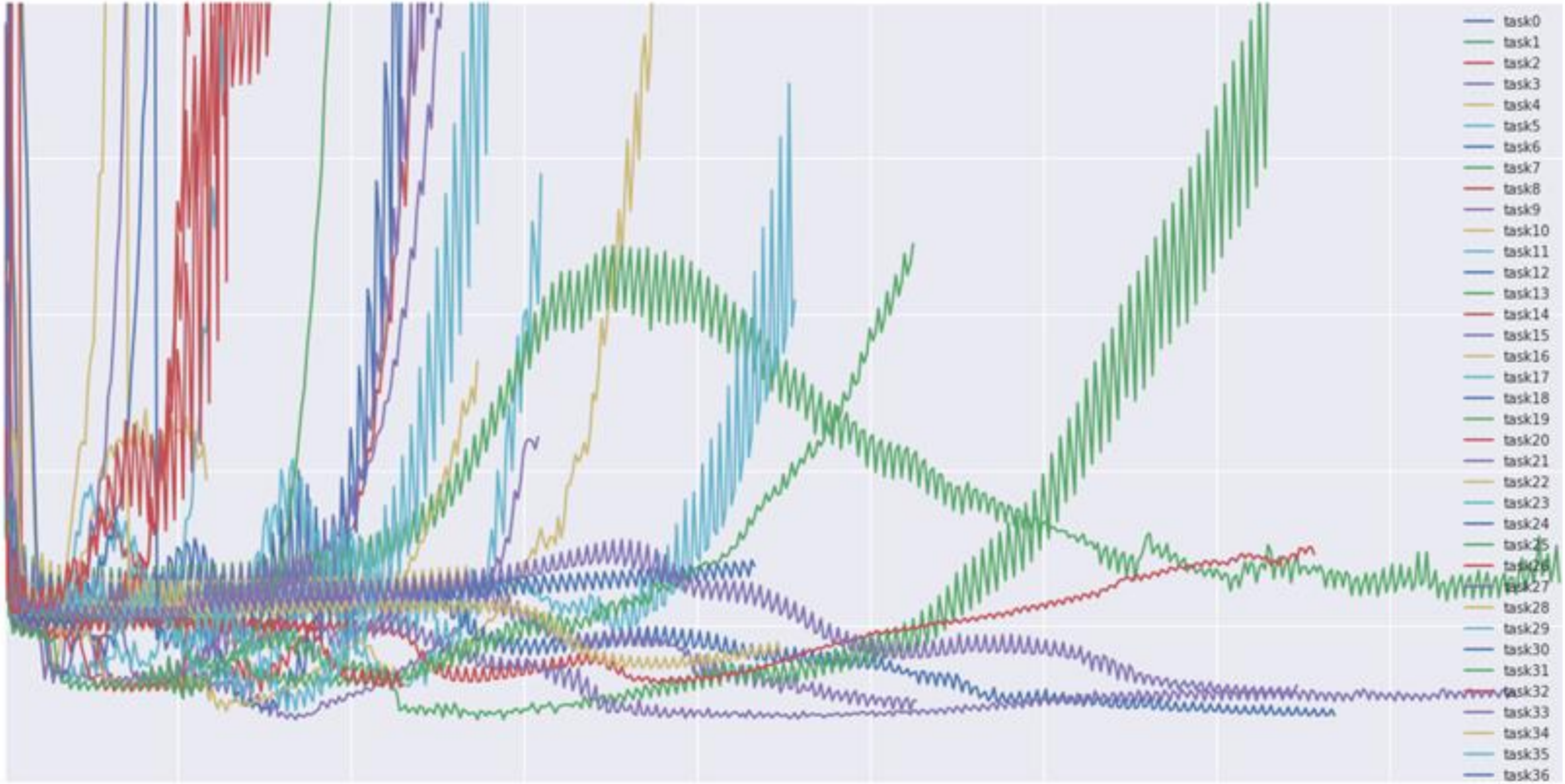
Losses may be noisy, use a scatter plot and also plot moving average to see trends better

Cross-validation

We develop "command centers" to visualize all our models training with different hyperparameters

check out [weights and biases]()

You can plot all your loss curves for different hyperparameters on a single plot



Don't look at accuracy or loss curves for too long!

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 and accuracy curves
Step 7: GOTO step 5

Hyperparameters to play with:

- network architecture
- learning rate, its decay schedule, update type
- regularization (L1/L2/Dropout strength)

Summary

- Improve your training error:
  - Optimizers
  - Learning rate schedules
- Improve your test error:
  - Regularization
  - Choosing Hyperparameters

Summary

Training Deep Neural Networks

- Details of the non-linear activation functions
- Data normalization
- Weight Initialization
- Batch Normalization
- Advanced Optimization
- Regularization
- Data Augmentation
- Transfer learning
- Hyperparameter Tuning

Next three lectures:

How to learn from sequence data?

- Recurrent Neural Networks
- Long-short Term Memory
- Transformers