CS5489
Lecture 10.2: Neural Networks and Deep Learning Part IV: Regularizations

Kede Ma

City University of Hong Kong (Dongguan)

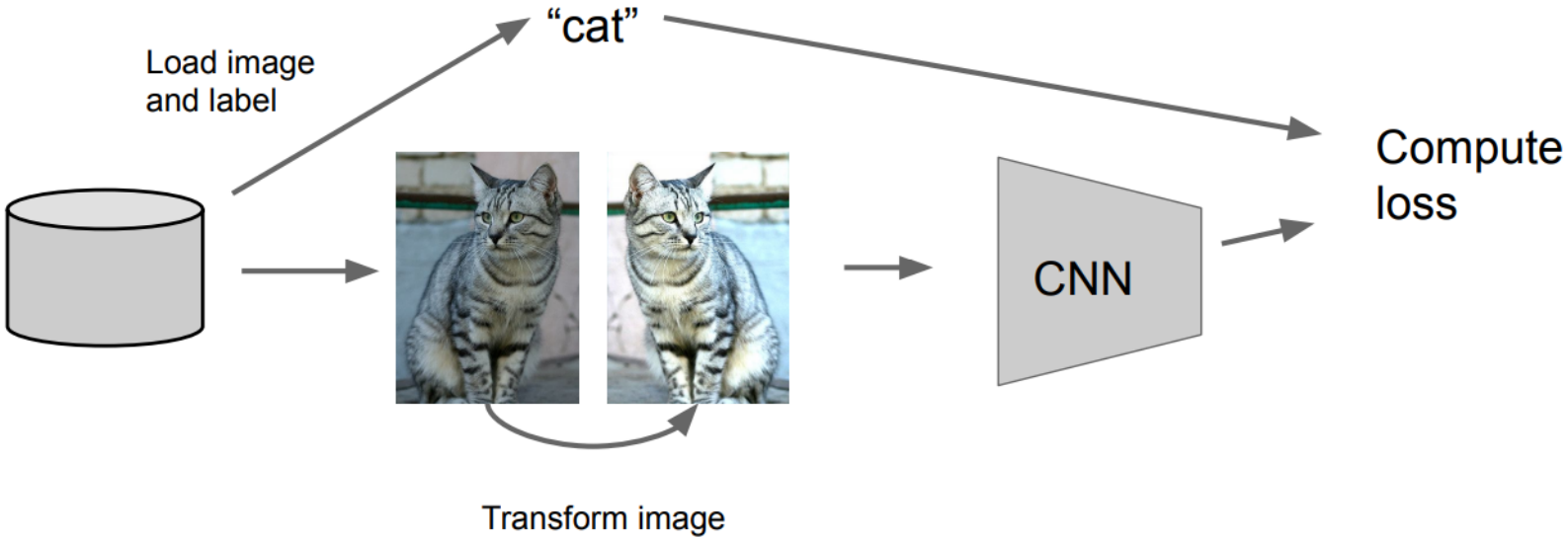
Slide template by courtesy of Benjamin M. Marlin

Part of slides are borrowed from the Stanford University CS231n course

Outline

Regularization: Data Augmentation

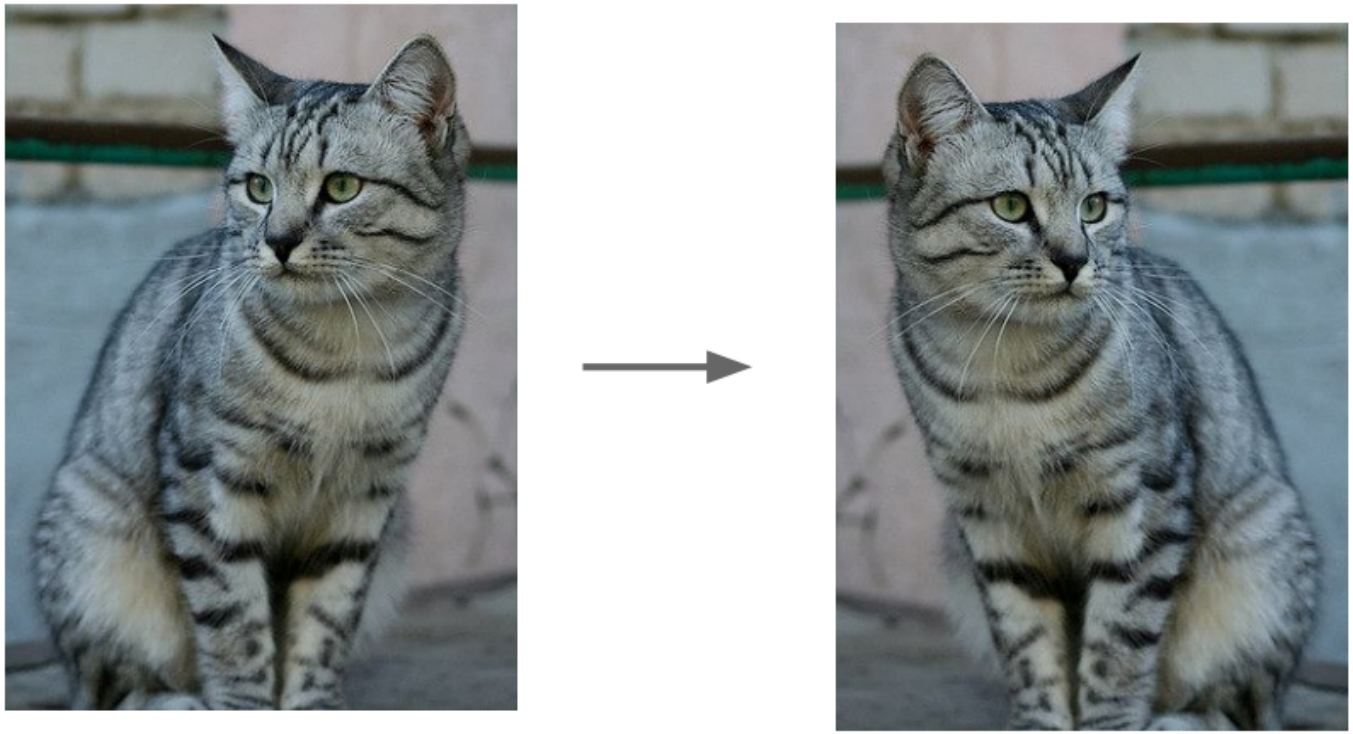
- Artificially permute the data to increase the dataset size
  - Goal: make the network invariant to the permutations



- Be careful about the transformation applied → **label preserving**
  - Example: classifying ‘b’ and ‘d’; ‘6’ and ‘9’

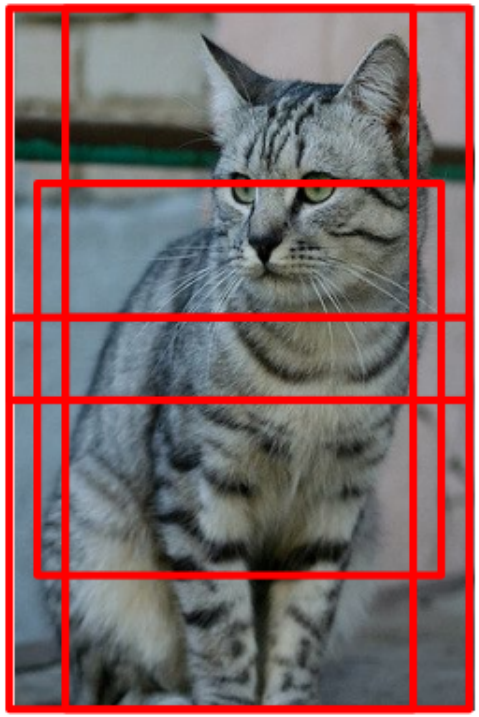
Data Augmentation for Image Classification

- Horizontal flips



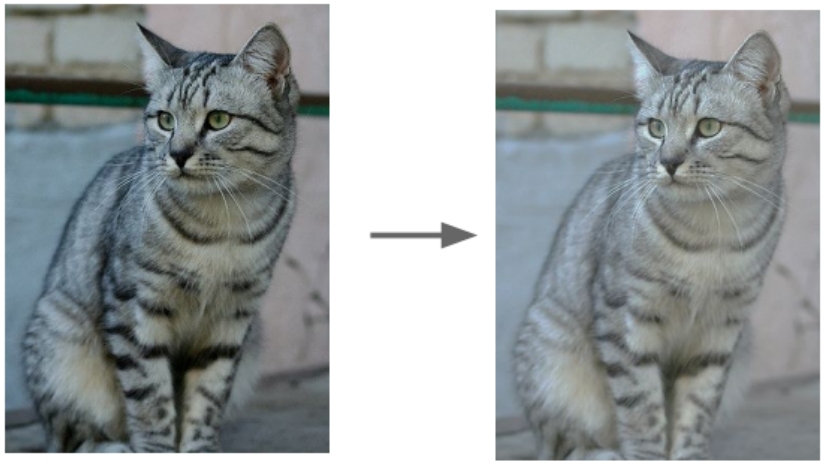
Data Augmentation for Image Classification

- Random crops and scales
  - **Training**: sample random crops / scales (with input resolution $\sim 512 \times 512$)
    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
    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 for Image Classification

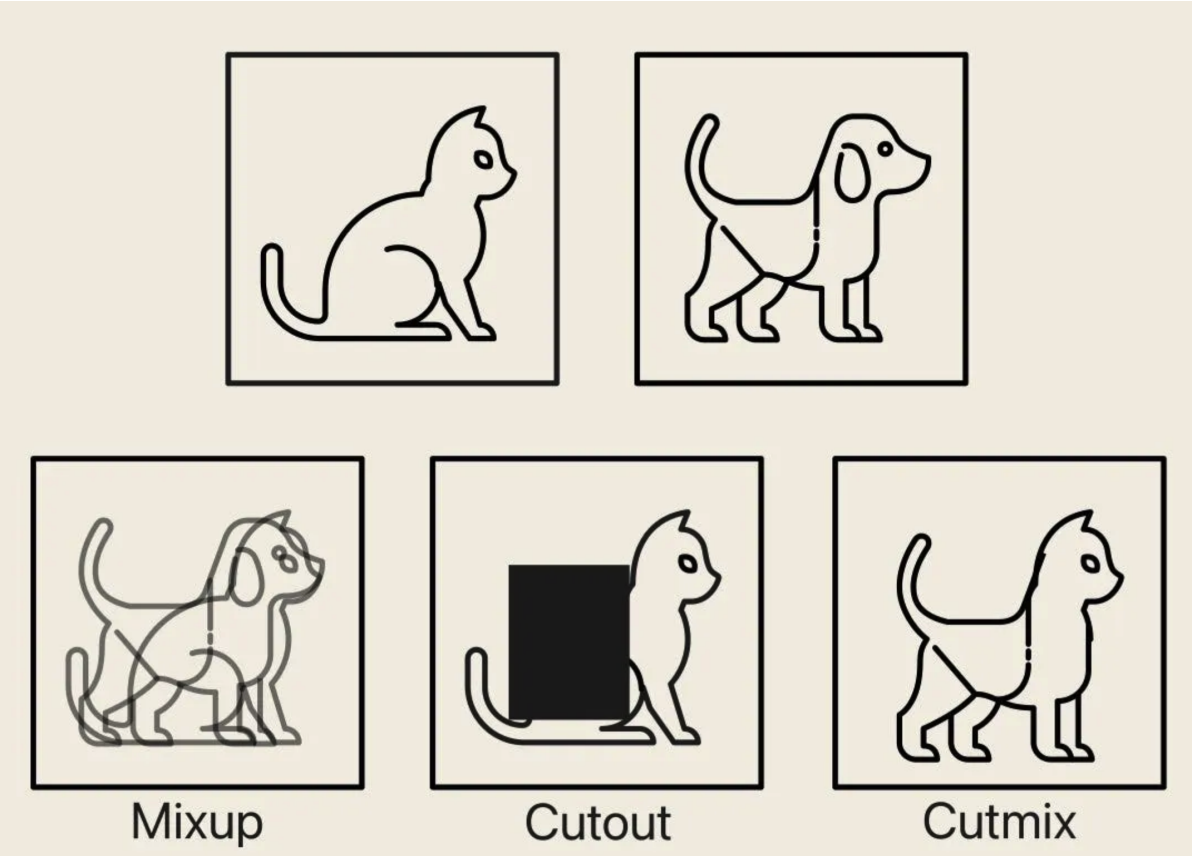
- Color jitter
  - E.g., randomize contrast and brightness



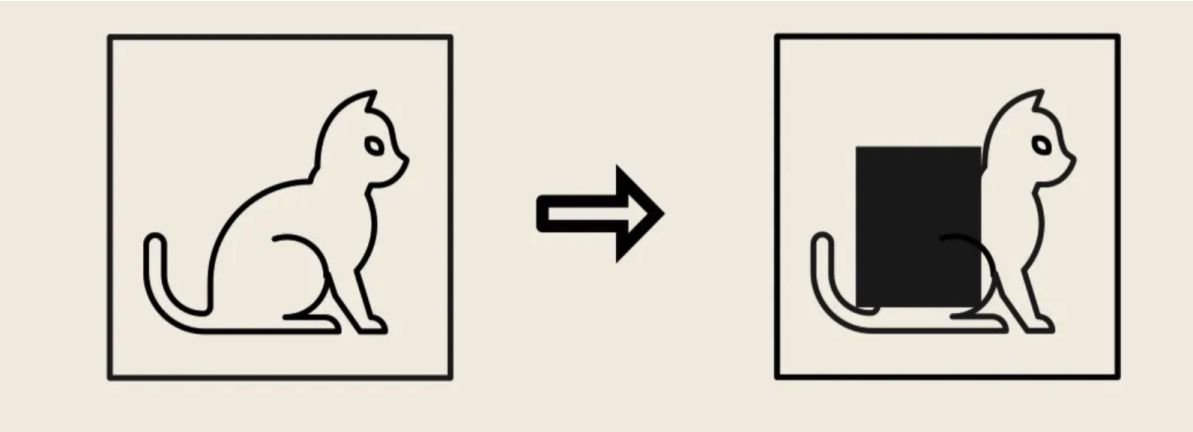
Data Augmentation for Image Classification

- The list goes on and on...
  - Random mix/combinations of:
    - Translation
    - Rotation
    - Stretching
    - Shearing
    - Lens distortions
    - Add noise to pixels (as a form of "dequantization")
    - ...
  - There are methods to *automatically* augment training data to boost the final performance!

Emerging Data Augmentation: Cutout, Mixup, Cutmix

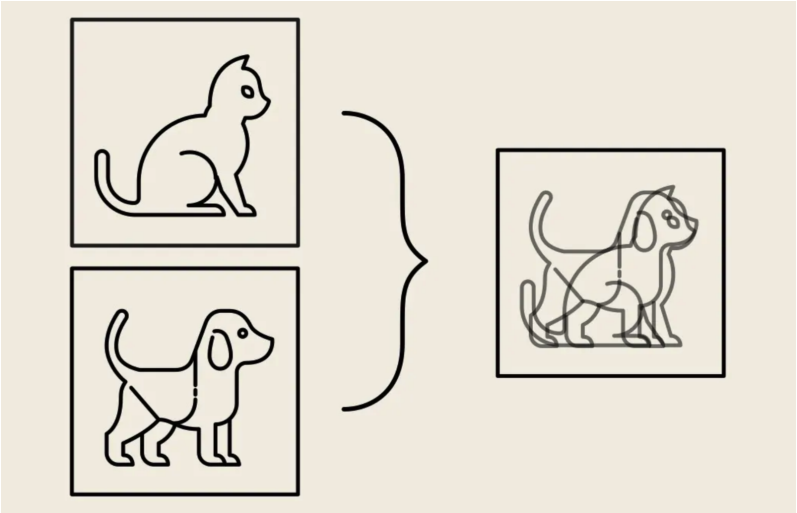


Cutout



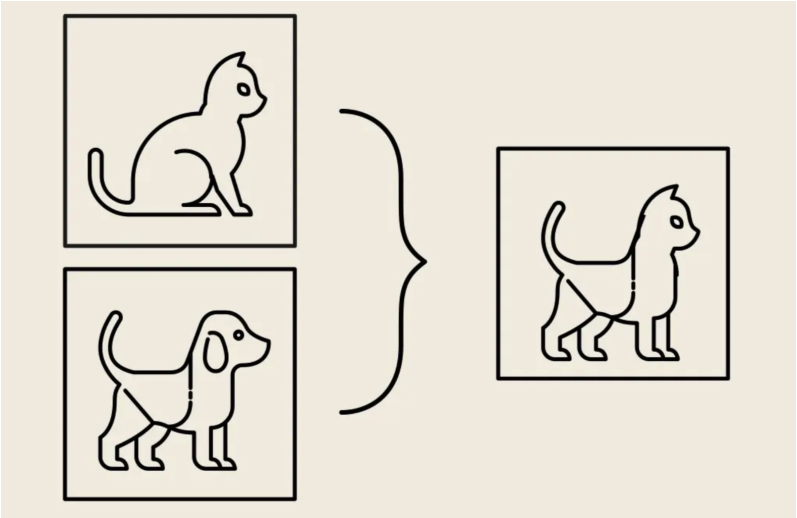
- Idea: Randomly remove a square region of pixels in an image during training
- Pros: Can remove noise, e.g., background
- Cons: Can also remove important features

Mixup



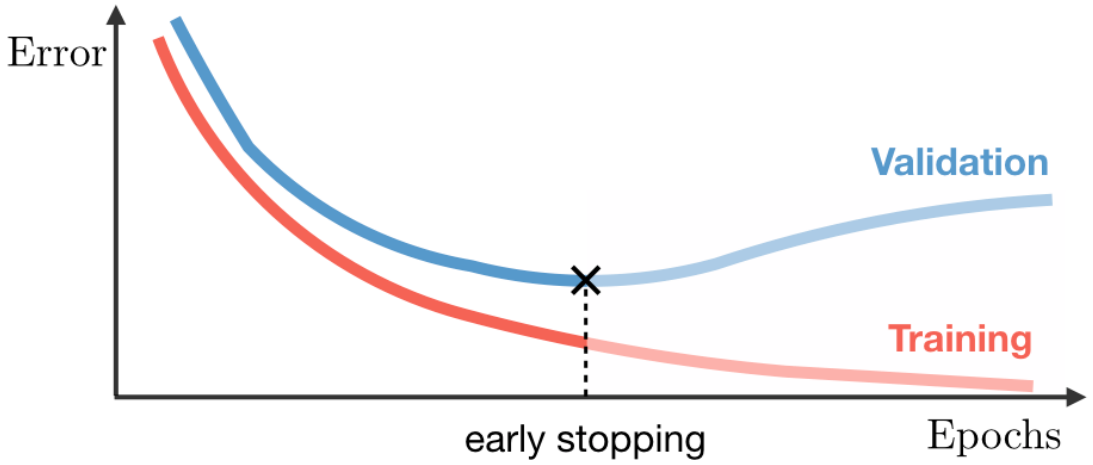
- Idea: Linearly mix two random images and their labels during training
- Pros: Increase diversity in the training data
- Cons: Can create blurred images, especially for images with complex textures

Cutmix



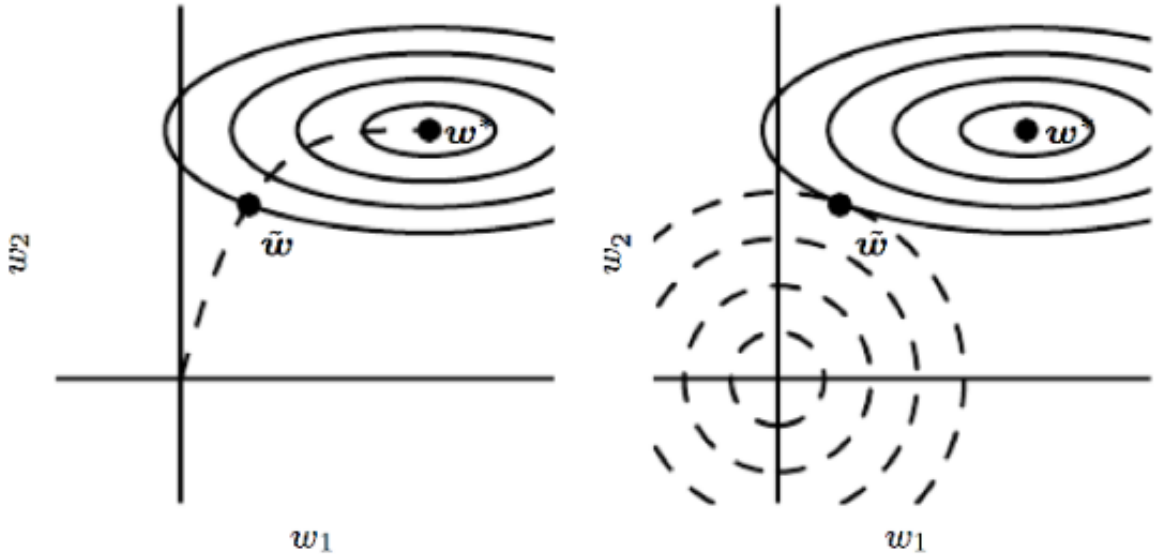
- Idea: Randomly select two images during training, cut a random patch of pixels from one image and paste it to the other, and then mix their labels proportionally to the area of the patch
- Pros: Increase diversity in the training data
- Cons: Can create unrealistic images; Can remove important features

Early Stopping



- Idea: Don't train the network to too small training error
- Recall overfitting: with a larger model complexity, it is easier to fit training samples
- Prevent overfitting: Use validation error to decide when to stop

Early Stopping as Regularization

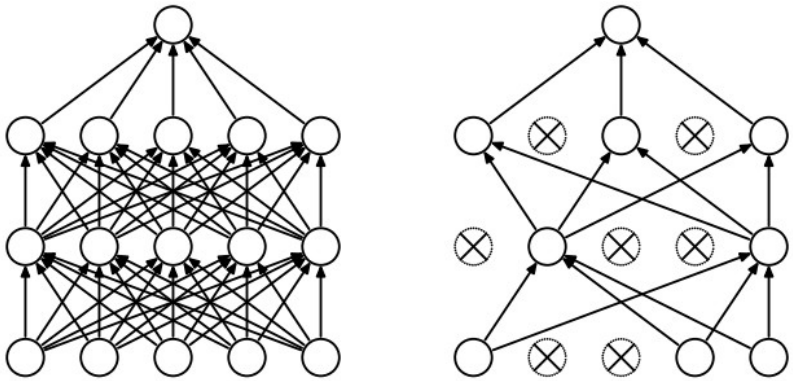


- The *effective* complexity of the network starts out small
  - As the weights are initialized small
- The effective complexity grows during the training process
  - As the weights get larger and larger
- Early stopping then represents a way of limiting the effective network complexity

Early Stopping

- In practice, when training, also output validation error
- Every time validation error improves, store a copy of the network weights
- When validation error plateaus for some time, stop
- Return the copy of the weights stored
- The number of training steps (*i.e.*, iterations) is also a hyperparameter

Dropout



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

- In each update step, randomly sample a different **binary** mask to all the input and hidden units
- Multiple the mask with the units and do the update as usual
- Typical dropout probability: 0.8/0.5 for input/hidden units
- Useful for fully connected layers, less for convolution layers

Dropout

- How can this possibly be a good idea?
  - Forces the network to have a redundant representation
    - Prevents co-adaptation of features, in which a feature is only helpful in the context/presence of several other specific features
  - Trains a large ensemble of models (that share parameters)
    - Each binary mask corresponds to one model
- How to vote to make a final prediction?
  - At test time all neurons are active always (Used more often)
    - Scale the activations so that for each neuron: output at test time = *expected* output at training time
  - Mask sampling (Used less often):
    - Randomly sample some (typically, $10 \sim 20$) masks
    - For each mask, apply it to the trained model to get a prediction
    - Take the majority vote (i.e., average) the predictions

Dropout at Test Time

- Dropout makes our output random!
  - $\mathbf{y} = f_{\mathbf{W}}(\mathbf{x}, \mathbf{z})$, where $\mathbf{z}$ is a random mask
- We want to "average out" the randomness at test-time
  - $\mathbf{y} = f_{\mathbf{W}}(\mathbf{x}) = \mathbb{E}_{\mathbf{z}}[f_{\mathbf{W}}(\mathbf{x}, \mathbf{z})] = \int_{\mathbf{z}} f_{\mathbf{W}}(\mathbf{x}, \mathbf{z}) d\mathbf{z}$
- Consider a single neuron: $a = w_1 x_1 + w_2 x_2$
  - At test time, we have: $\mathbb{E}[a] = w_1 x_1 + w_2 x_2$
  - At training time (with a dropout probability of 0.5), we have

$$\begin{aligned}\mathbb{E}[a] =& \frac{1}{4}(w_1 x_1 + w_2 x_2) + \frac{1}{4}(w_1 x_1 + w_2 0) \\ &+ \frac{1}{4}(w_1 0 + w_2 x_2) + \frac{1}{4}(w_1 0 + w_2 0) = \frac{1}{2}(w_1 x_1 + w_2 x_2)\end{aligned}$$

- At test time, multiply by dropout probability

Dropout at Test 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

Dropout at Test Time

- More common: "inverted dropout"

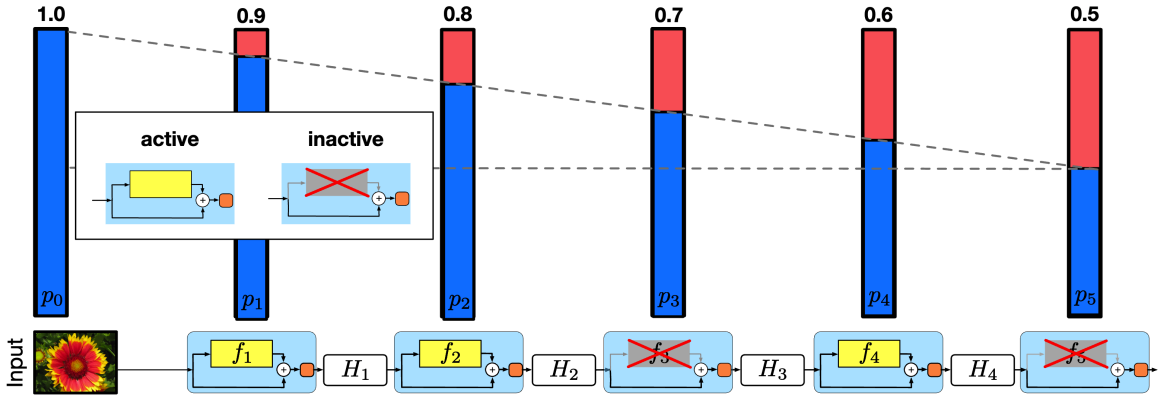
```
p = 0.5 # probability of keeping a unit active. higher = less dropout

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
```

Stochastic Depth



- Similar to Dropout, in each iteration, Stochastic Depth randomly drops a subset of layers with some survival probabilities and bypasses them with the identity function
- At test time, all layers are activated and re-calibrated by multiplying the corresponding survival probability
- Reduces training time substantially and improves generalization as an implicit model ensemble

Batch Normalization

- Idea: adjust activations to lie within a desired operating range, while maintaining their relative values
- Given a mini-batch $\mathcal{B} = \{(\mathbf{x}^{(i)}, y^{(i)}), i = 1, \ldots, M\}$
- Compute per-channel (*i.e.*, feature) mean: $\mu_j = \frac{1}{M} \sum_{i=1}^{M} x_j^{(i)}$
- Compute per-channel variance: $\sigma_j^2 = \frac{1}{M} \sum_{i=1}^{M} (x_j^{(i)} - \mu_j)^2$
- Normalize $\mathbf{x}^{(i)}$ across channel: $\hat{x}_j^{(i)} = \frac{x_j^{(i)} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$
- Issue: what if zero-mean, unit variance is too hard of a constraint?
- Incorporate learnable scale and shift parameters γ and β :

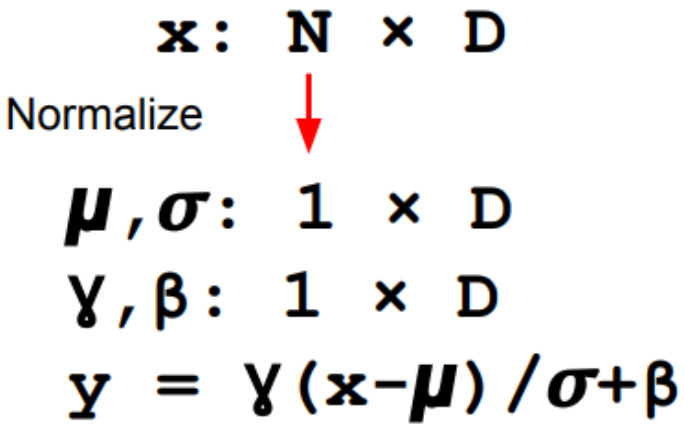
$$z_j^{(i)} = \gamma_j \hat{x}_j^{(i)} + \beta_j$$

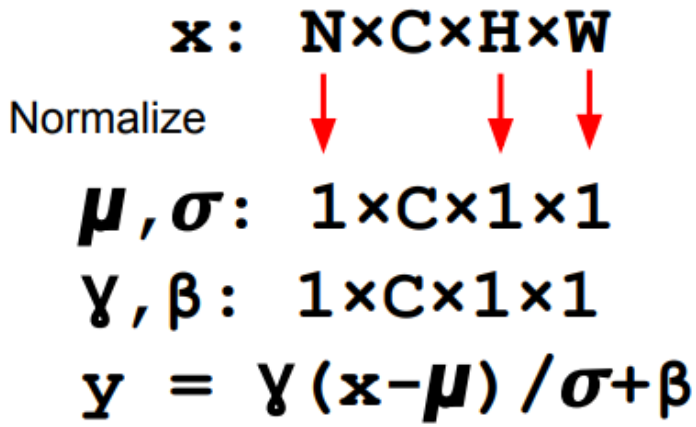
Batch Normalization: Test Time

- Mean and variance estimates depend on mini-batch at train time; can't do this at test-time
- Solution: keep exponentially decaying moving average of values seen during training and use them for testing
- During testing, batch normalization becomes a linear operator; can be fused with the previous convolution layer
- Usually inserted after convolution layers and before nonlinearity
- Behaves **differently** during training and testing: this is a very common source of bugs!

Batch Normalization for Convolution Layers

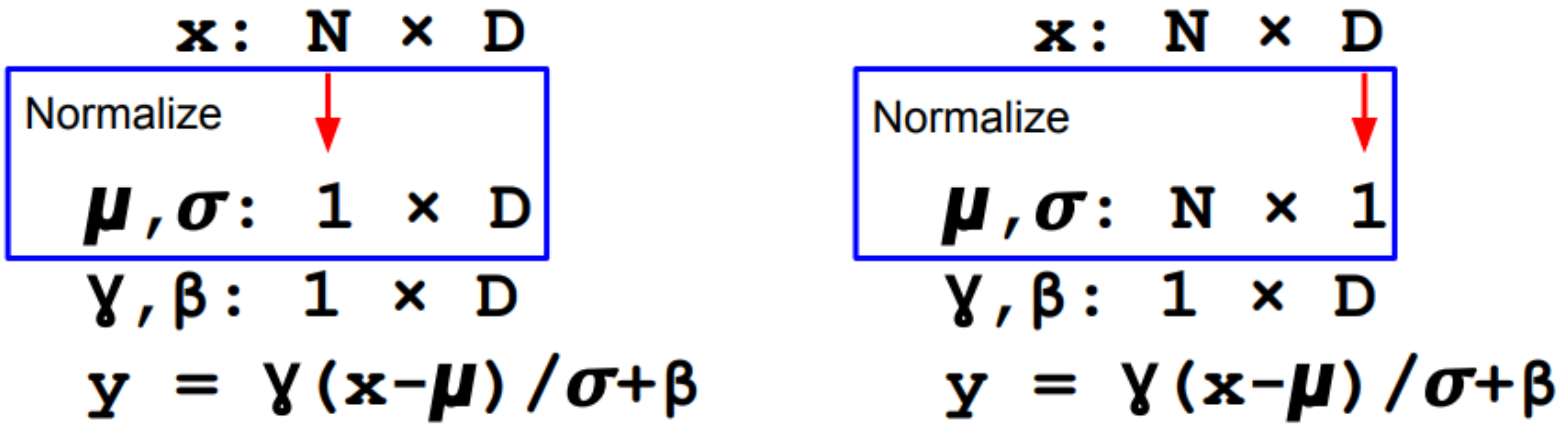
- Normalize along both batch and spatial dimensions





Layer Normalization

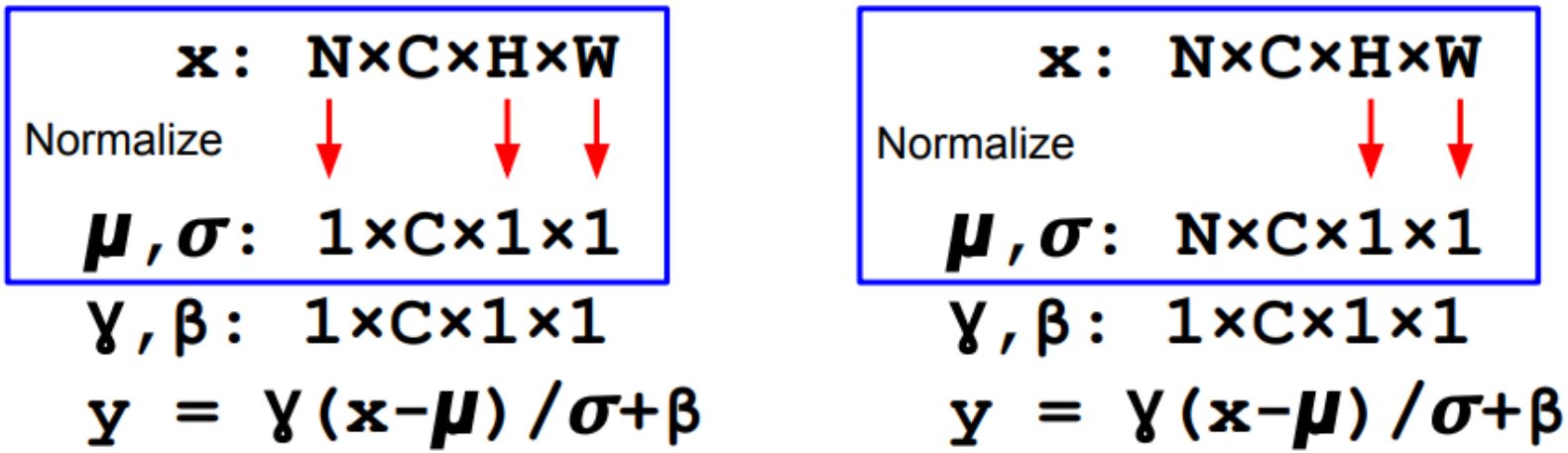
- Normalize along the channel dimension
- Same behavior at train and test!



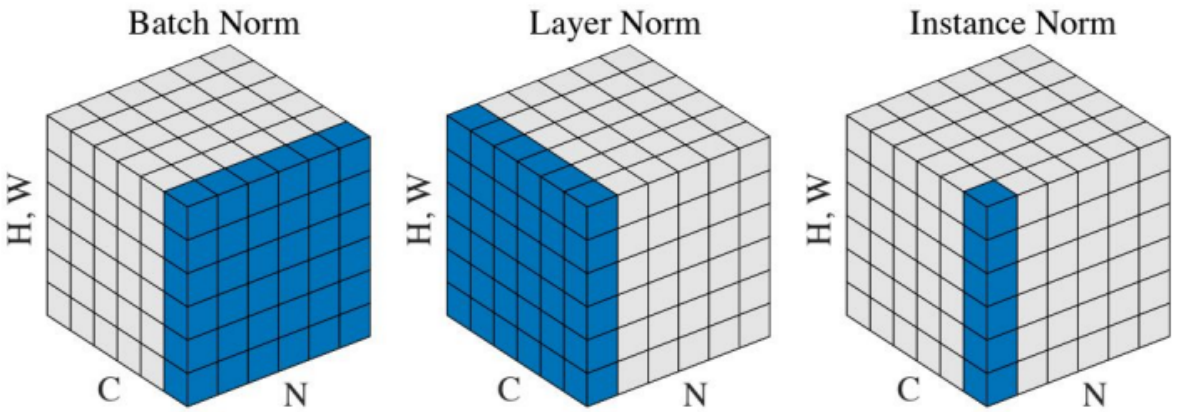
- Layer normalization for convolution layers normalizes along both channel and spatial dimensions

Instance Normalization for Convolution Layers

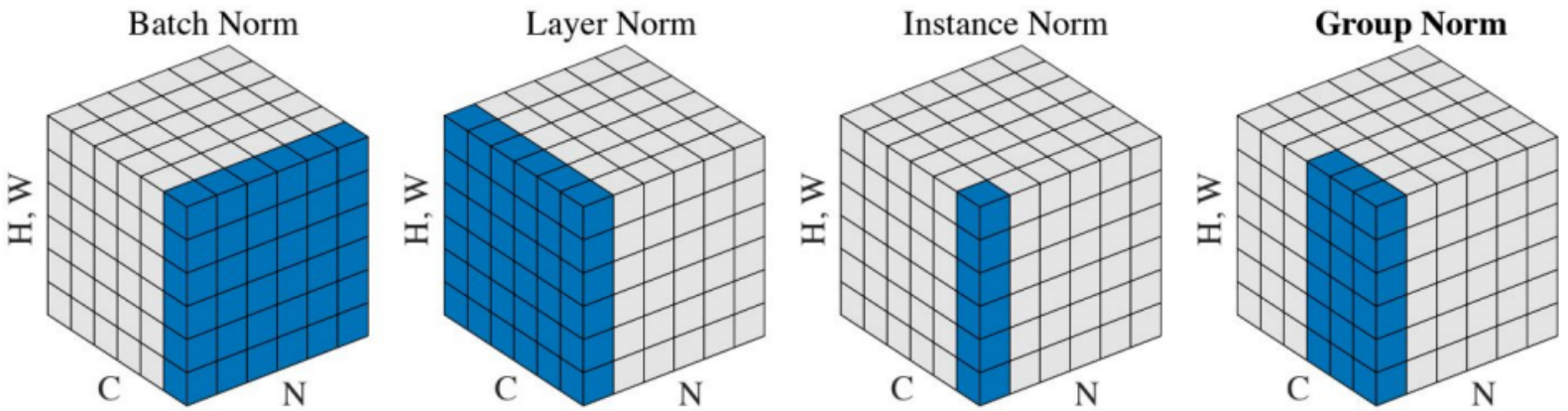
- Normalize along the spatial dimension
- Same behavior at train and test!



Comparison of Normalization Layers



Group Normalization

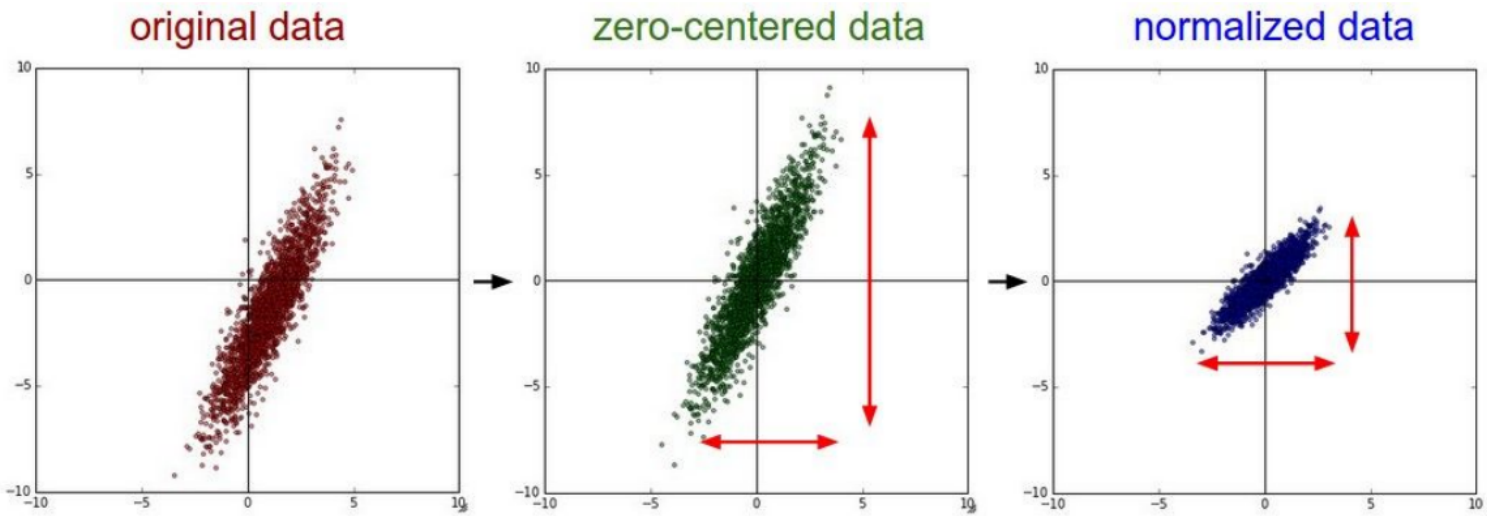


- Group normalization divides the channels into groups and computes within each group the mean and variance for normalization

Outline

Data Pre-Processing

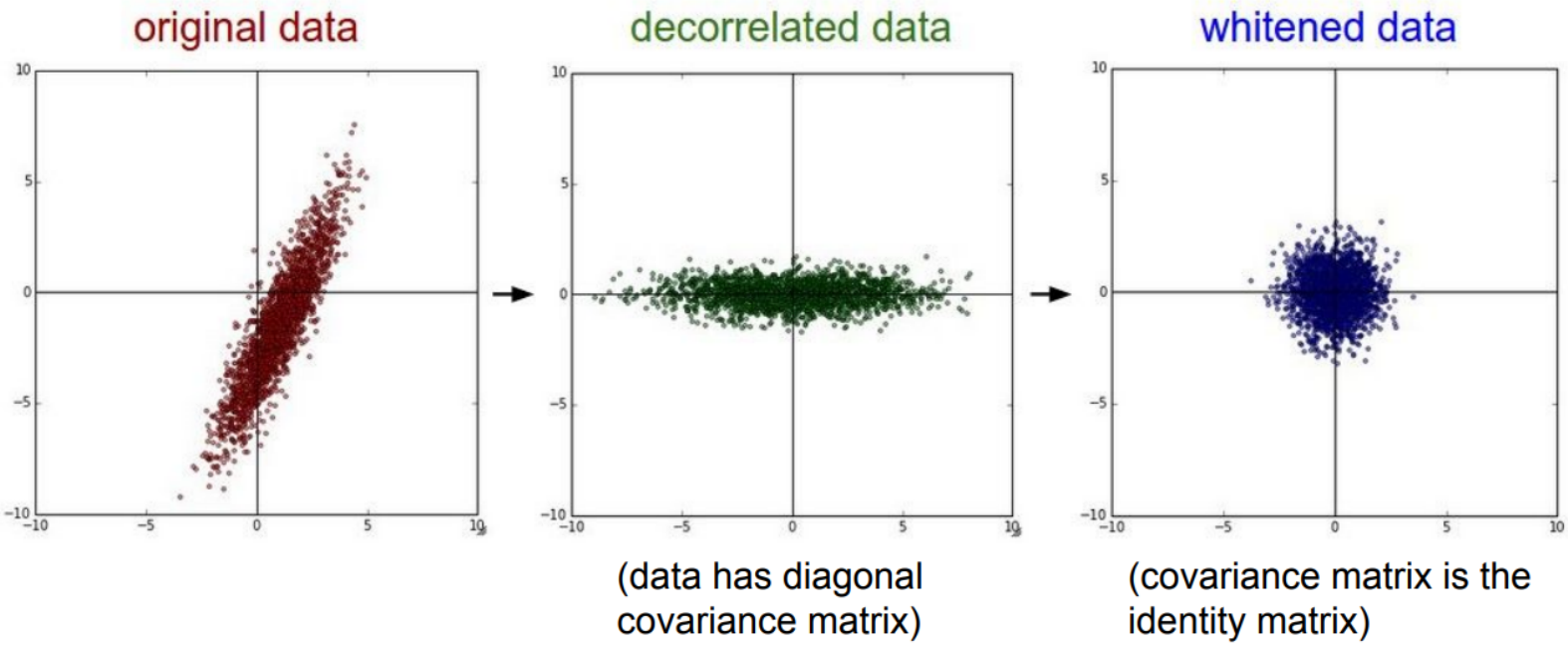
- Assume $\mathbf{X} \in \mathbb{R}^{M \times N}$ is data matrix (each example in a row)



  - Makes sense only when data attributes have approximately equal importance but different scales

Data Pre-Processing

- In practice, you may also want to PCA your data
  - The data is first centered, then projected into the eigenbasis, followed by dividing every dimension by the corresponding eigenvalue



PCA as Whitening

- Eigendecomposition:
  $\mathbf{C} = \frac{1}{M}\mathbf{X}^T\mathbf{X} = \frac{1}{M}\sum_{i=1}^{M}\mathbf{x}^{(i)}(\mathbf{x}^{(i)})^T = \mathbf{U}\mathbf{\Sigma}^2\mathbf{U}^T$
- Defining $\hat{\mathbf{x}}^{(i)} = \mathbf{\Sigma}^{-1}\mathbf{U}^T\mathbf{x}^{(i)}$, we have

$$\begin{aligned}\hat{\mathbf{C}} &= \frac{1}{M}\sum_{i=1}^{M}\hat{\mathbf{x}}^{(i)}(\hat{\mathbf{x}}^{(i)})^T = \frac{1}{M}\sum_{i=1}^{M}\mathbf{\Sigma}^{-1}\mathbf{U}^T\mathbf{x}^{(i)}(\mathbf{\Sigma}^{-1}\mathbf{U}^T\mathbf{x}^{(i)})^T \\ &= \mathbf{\Sigma}^{-1}\mathbf{U}^T\left(\frac{1}{M}\sum_{i=1}^{M}\mathbf{x}^{(i)}(\mathbf{x}^{(i)})^T\right)\mathbf{U}\mathbf{\Sigma}^{-1} \\ &= \mathbf{\Sigma}^{-1}\mathbf{U}^T\mathbf{U}\mathbf{\Sigma}^2\mathbf{U}^T\mathbf{U}\mathbf{\Sigma}^{-1} = \mathbf{I}\end{aligned}$$

  - This is termed as *whitening* in signal processing

Data Pre-Processing

- In practice for color images: center only
  - Not common to do PCA whitening
- Three variants:
  - Subtract the mean image
    - Mean image $= [3, \text{height}, \text{width}]$ array
  - Subtract per-channel mean
    - Mean along each channel $= 3$ numbers
  - Subtract per-channel mean and divide by per-channel std
    - Mean and std along each channel $= 2 \times 3$ numbers

Data Pre-Processing

- It is typical to train your network on training data several times
  - One complete pass through the data is called an *epoch*
- Never forget to **shuffle** your training data for each epoch!

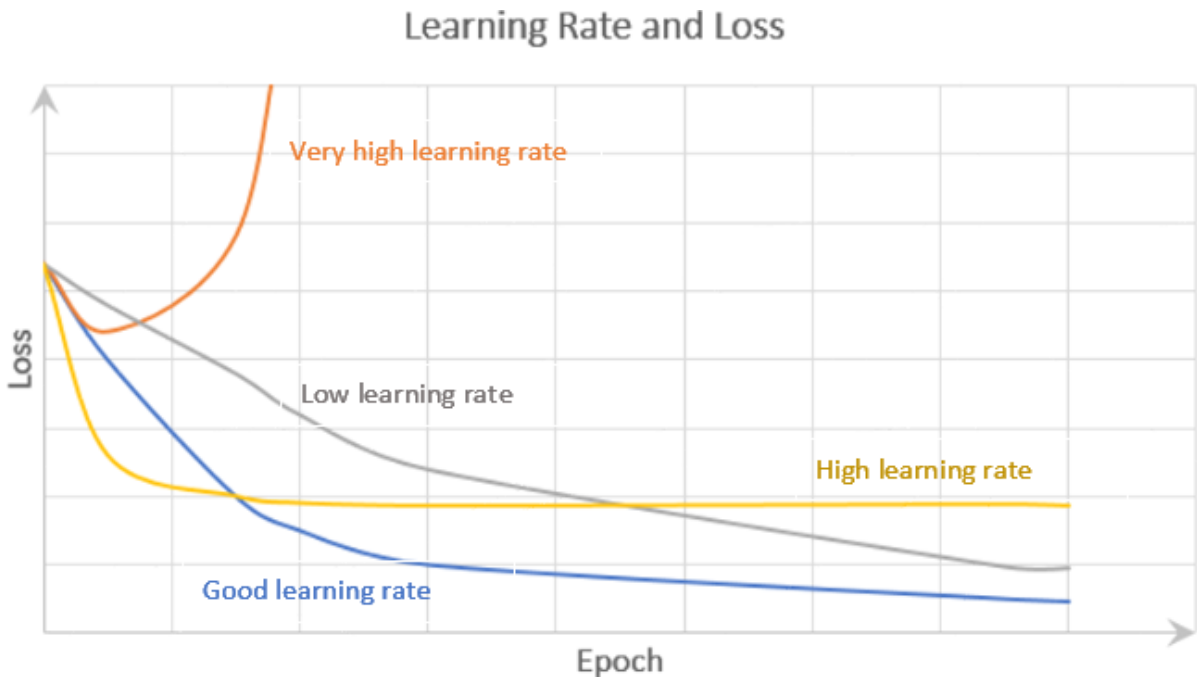
Weight Initialization

- Constant (including all zero) initialization: terribly wrong!
  - If every neuron in the network computes the same output, then all of them will also compute the same gradients during backpropagation and undergo the exact same parameter updates
- Small random initialization
  - Works okay for small networks, but not for deeper networks
- Current recommendation for initializing CNNs with ReLU:

$$w = \text{np.random.randn}(n) \times \text{sqrt}(2.0/n)$$

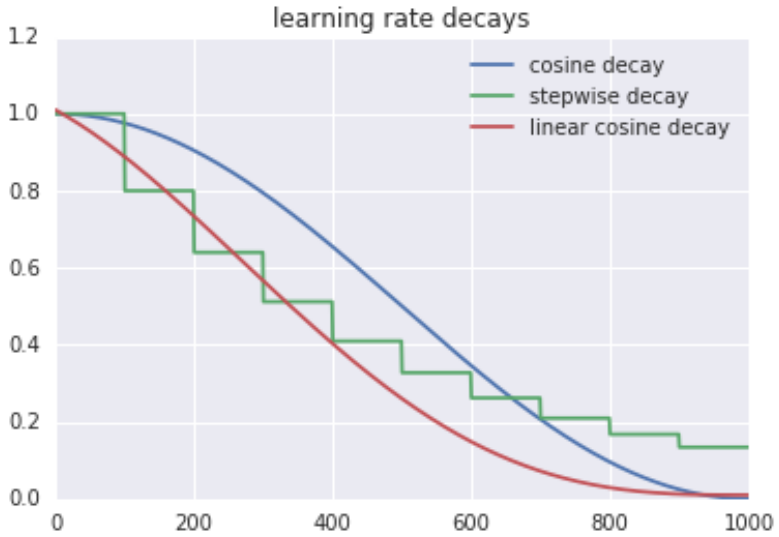
  - randn: Gaussian
  - n: the number of input channels for the current layer
- Proper initialization is an active area of research

Learning Rate Decay



- SGD, AdaGrad, RMSProp, and Adam all have learning rate as a hyperparameter
- Which one of these learning rates is best to use?
  - Answer: All of them! Start with large learning rates and decay over time

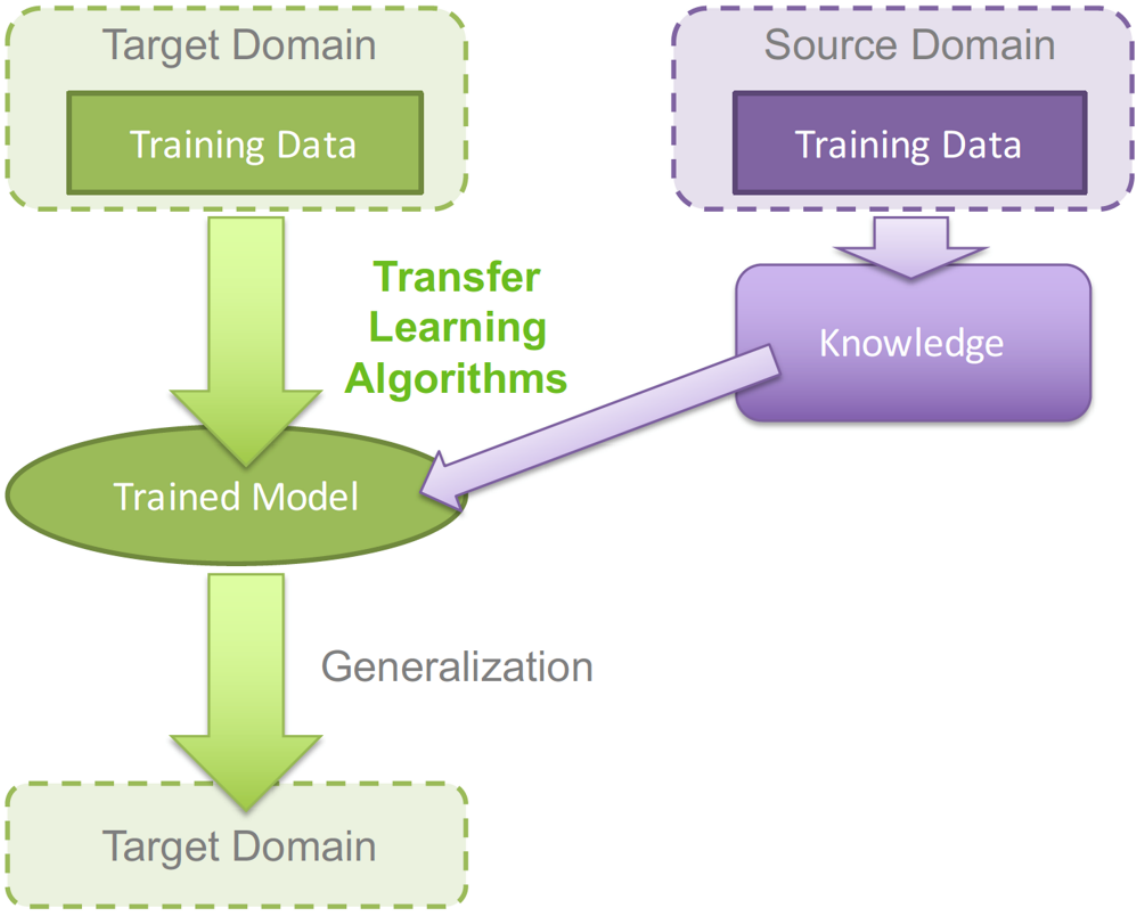
Learning Rate Decay



- Stepwise: Reduce learning rate at a few fixed points
  - E.g., multiply α by 0.1 after epochs 30, 60, and 90
- Cosine: $\alpha^{(t)} = \frac{1}{2}\alpha^{(0)}(1 + \cos(t\pi/T))$, t: epoch index
- Linear: $\alpha^{(t)} = \alpha^{(0)}(1 - t/T)$

Transfer Learning

- Improve learning new tasks by learned tasks



Fine-Tuning Pre-Trained Model

- Deep features are fairly transferable, and open-source pre-trained models are now everywhere

	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 large number of layers

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 ($\sim$ 5-10 minibatches)
    - Fiddle with architecture, learning rate, weight initialization
  - Loss not going down?
    - Learning rate too low, bad initialization
  - Loss explodes to Inf or NaN?
    - Learning rate too high, bad initialization

Choosing Hyperparameters

- Step 1: Check initial loss
- Step 2: Overfit a small sample
- Step 3: Find learning rate 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 $\sim$ 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 learning rate that makes loss go down
- Step 4: Coarse grid, train for $\sim$ 1-5 epochs
  - Choose a few values of learning rate and weight decay around what worked from Step 3, train a few models for $\sim$ 1-5 epochs
  - Good weight decay to try: $1e^{-4}$, $1e^{-5}$, 0

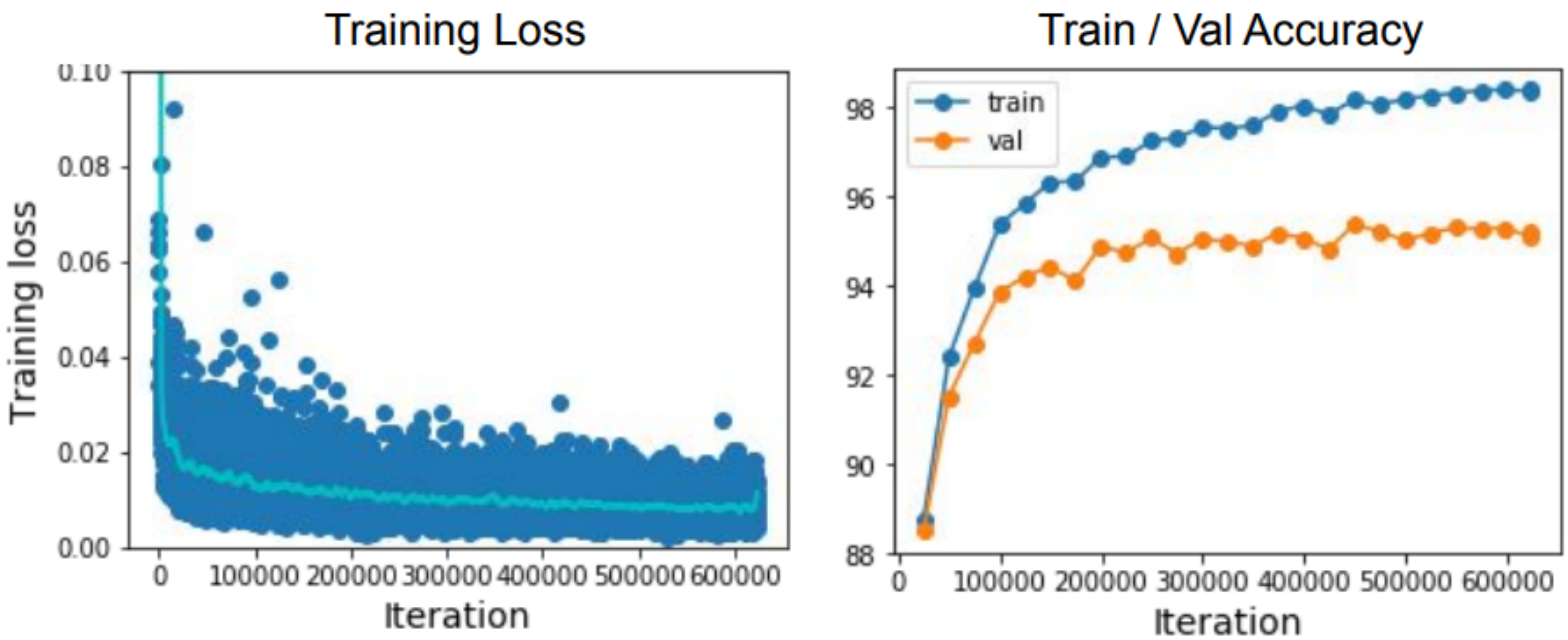
Choosing Hyperparameters

- Step 1: Check initial loss
- Step 2: Overfit a small sample
- Step 3: Find learning rate that makes loss go down
- Step 4: Coarse grid, train for $\sim$ 1-5 epochs
- Step 5: Refine grid, train longer
  - Pick best models from Step 4, train them for longer ($\sim$ 10-20 epochs) without learning rate decay

Choosing Hyperparameters

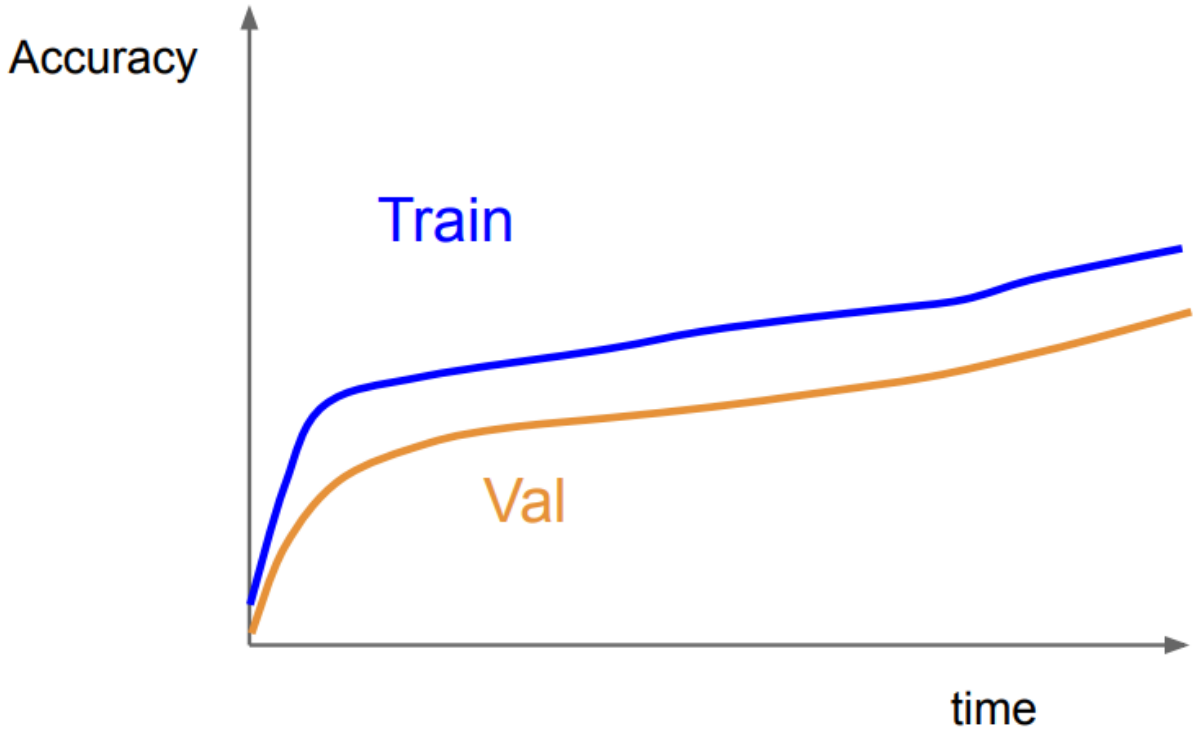
- Step 1: Check initial loss
- Step 2: Overfit a small sample
- Step 3: Find learning rate that makes loss go down
- Step 4: Coarse grid, train for $\sim$ 1-5 epochs
- Step 5: Refine grid, train longer
- Step 6: Look at loss curves

Look at Learning Curves



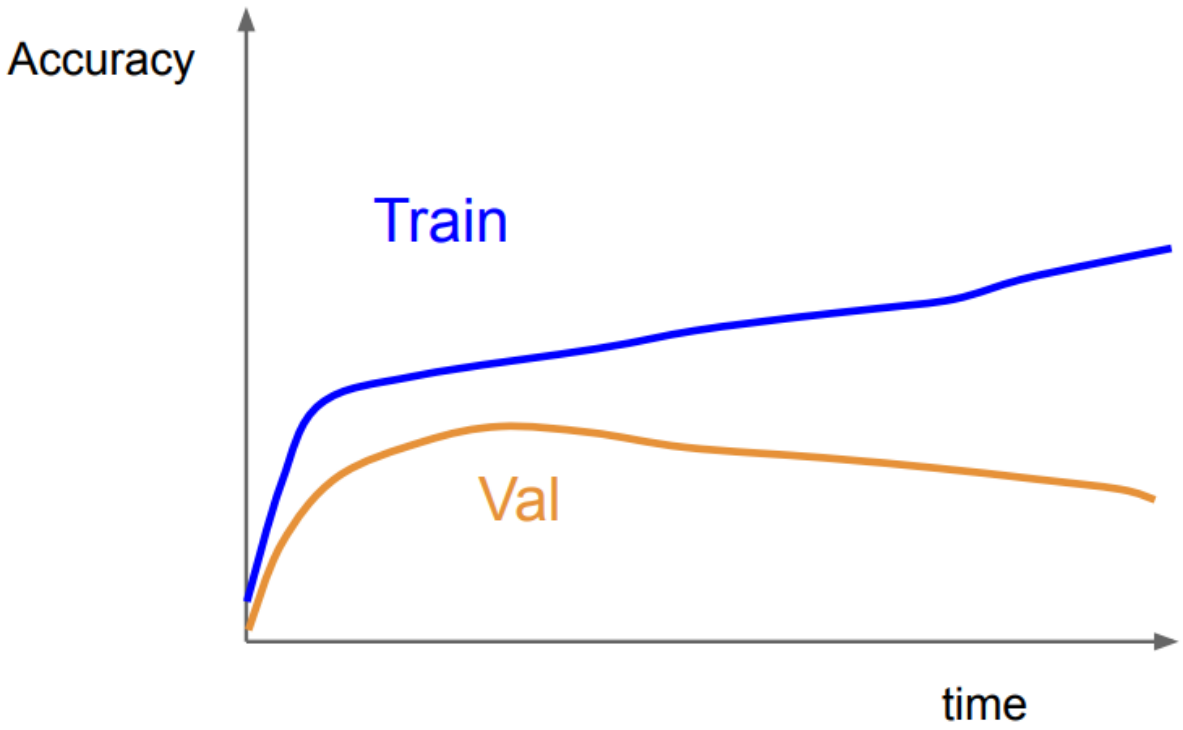
- Losses may be noisy
  - Use a scatter plot
  - And also plot exponentially decaying moving average to see trends better

Look at Learning Curves



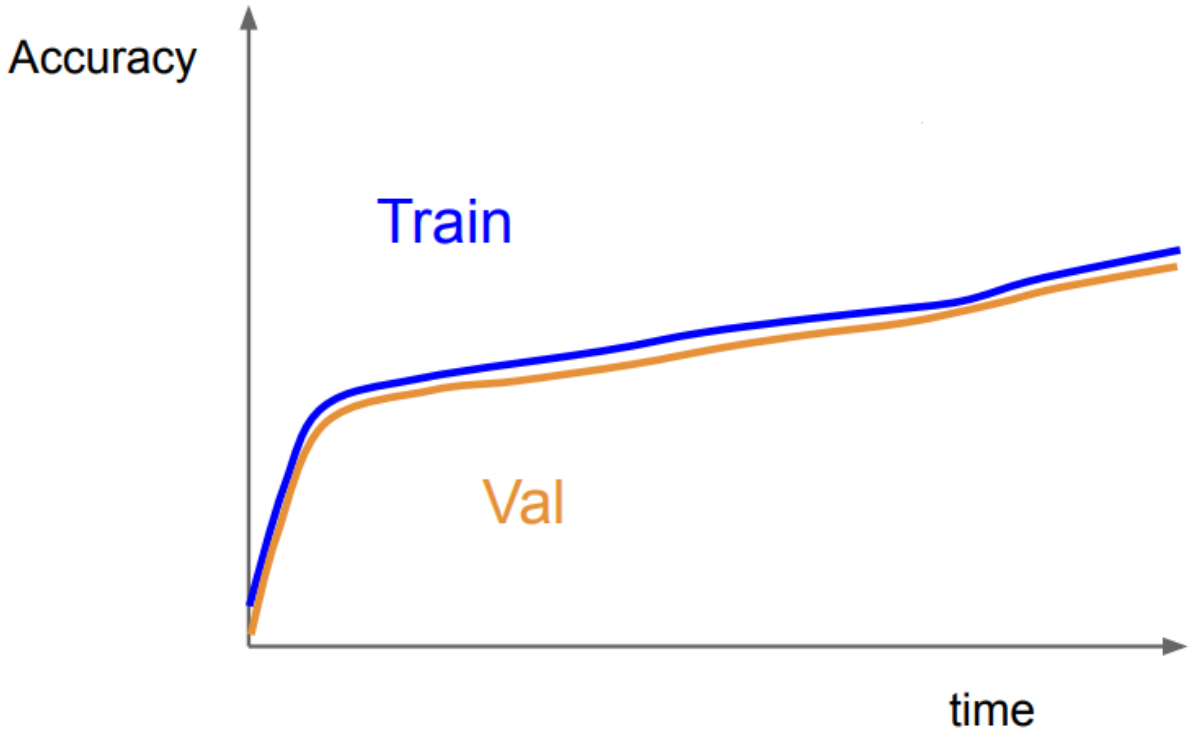
- Accuracy still going up, you need to train longer

Look at Learning Curves



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

Look at Learning Curves



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

Choosing Hyperparameters

- Step 1: Check initial loss
- Step 2: Overfit a small sample
- Step 3: Find learning rate that makes loss go down
- Step 4: Coarse grid, train for $\sim$ 1-5 epochs
- Step 5: Refine grid, train longer
- Step 6: Look at loss curves
- Step 7: Go to Step 5

Summary

- Improve your training error:
  - Network architectures
  - Initializations
  - Optimizers
  - Learning rate schedulers
- Improve your test error:
  - Regularization
  - Choosing hyperparameters