Training Neural Networks
Regularization

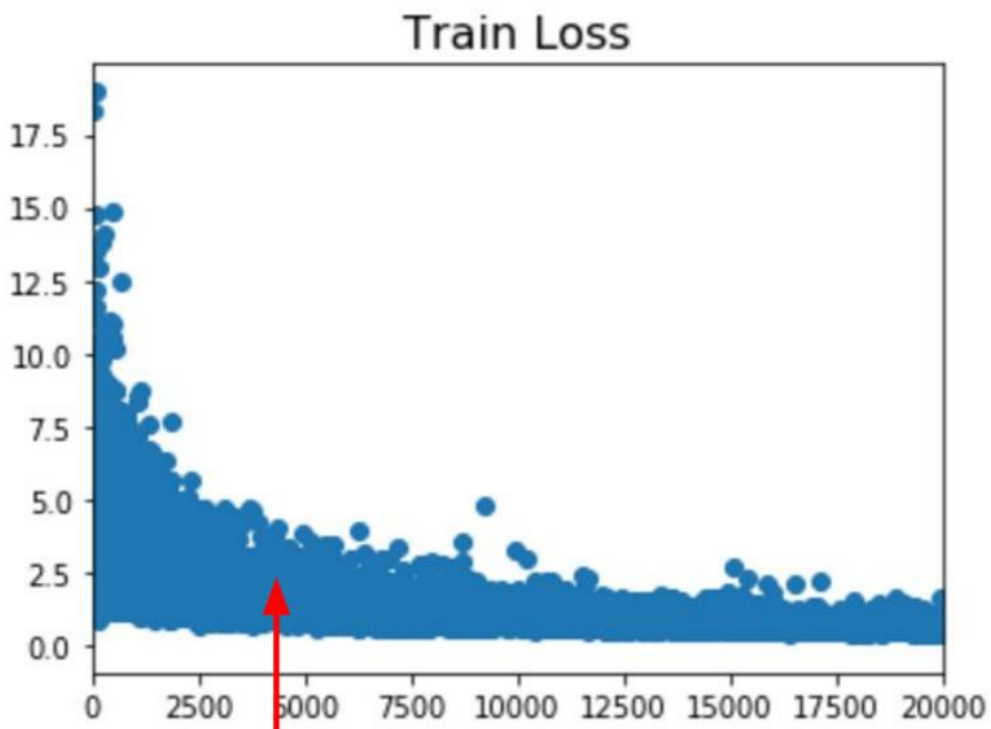
M. Soleymani
Sharif University of Technology
Fall 2017

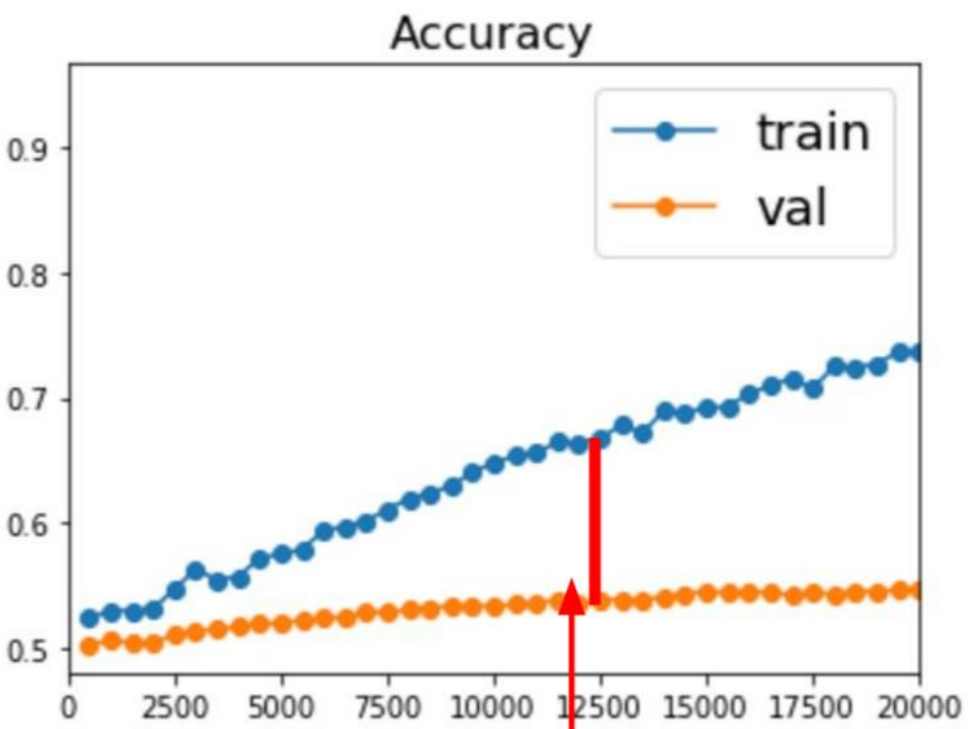
Most slides have been adapted from Fei Fei Li and colleagues lectures, cs231n, Stanford 2017, some from Andrew Ng lectures, “Deep Learning Specialization”, coursera, 2017,

Outline

- Regularization
- Drop-out
- Data Augmentation
- Early stopping
- Model ensembles
- Hyper-parameter selection

Beyond Training Error

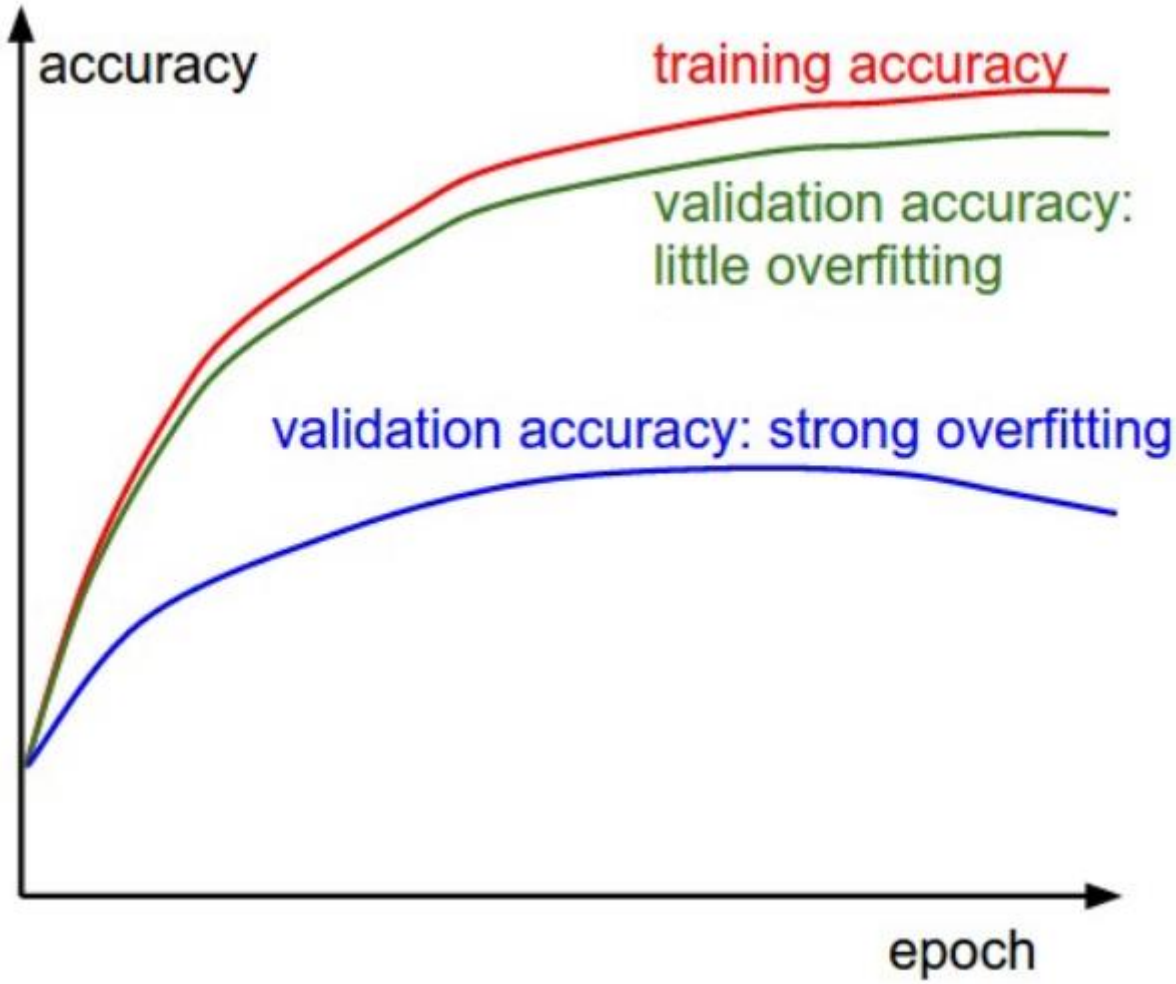




Better optimization algorithms help reduce training loss

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

Beyond Training Error



Regularization: Add term to loss

$$J(W) = \frac{1}{N}\sum_{n=1}^{N} L^{(n)}(W) + \lambda R(W)$$

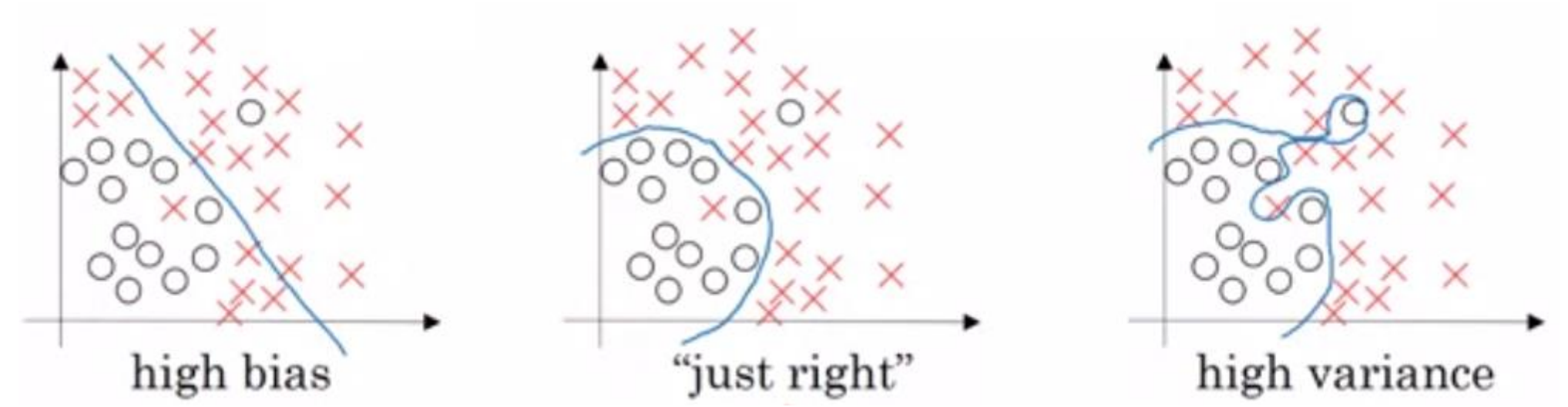
In common use:

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 help to reduce overfitting



[Andrew Ng, Deep Learning Specialization, 2017]

Why L2 regularization called weight decay

$$J(W) = L(W) + \lambda R(W)$$

$$J(W) = L(W) + \lambda \sum_{l=1}^{L} W^{[l]^T} W^{[l]}$$

$$W^{[l]} \leftarrow W^{[l]} - \alpha \nabla_{W^{[l]}} L(W) - 2\lambda W^{[l]}$$

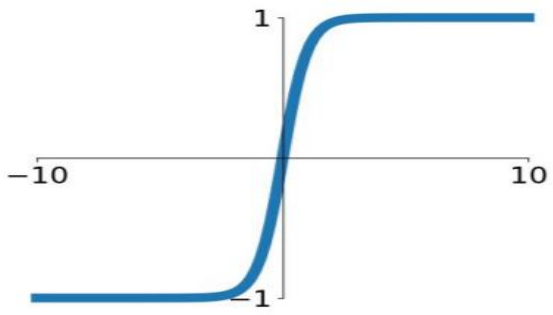
$$W^{[l]} \leftarrow W^{[l]}(1 - 2\lambda) - \alpha \nabla_{W^{[l]}} L(W)$$

Weight decay

With regularization, smoother cost vs. #epoch curve is obtained.

Regularization can prevent overfitting

- Small W linear regime of tanh
- A deep network with small W can also act as a near linear function

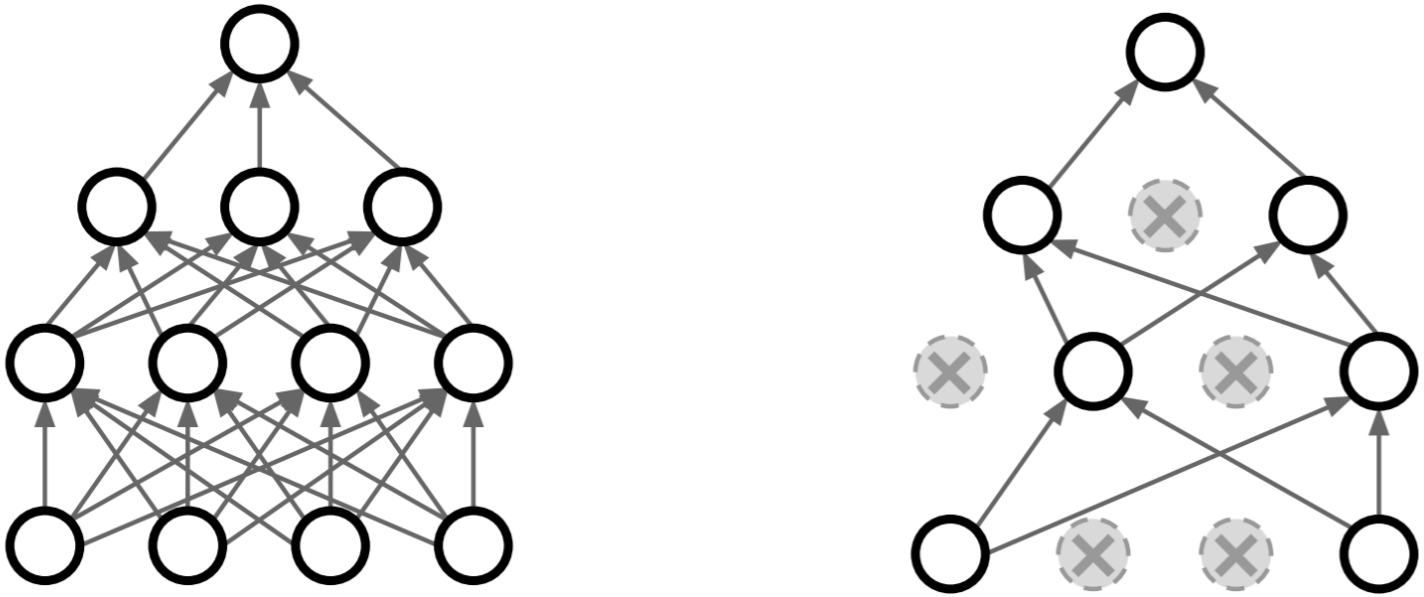


tanh(x)

Regularization: Dropout

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

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



- Thus, a smaller network is trained for each sample
  - Smaller network provides a regularization effect

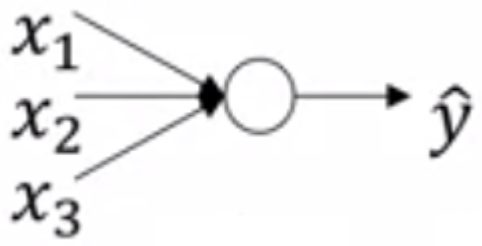
Implementing dropout (Inverted dropout)

- for l=1,...L
  - $d^{[l]} \leftarrow I\left(rand(\#neurons^{[l]}, 1) < keep\_prob\right)$
  - $a^{[l]} \leftarrow a^{[l]} .* d^{[l]}$

- for l=1,...L
  - $d^{[l]} \leftarrow I\left(rand(\#neurons^{[l]}, 1) < keep\_prob\right)$
  - $a^{[l]} \leftarrow a^{[l]} .* d^{[l]}$
  - $a^{[l]} \leftarrow a^{[l]} / keep\_prob$

Why does drop-out work?

- Intuition: cannot rely on any one feature, so have to spread out weights

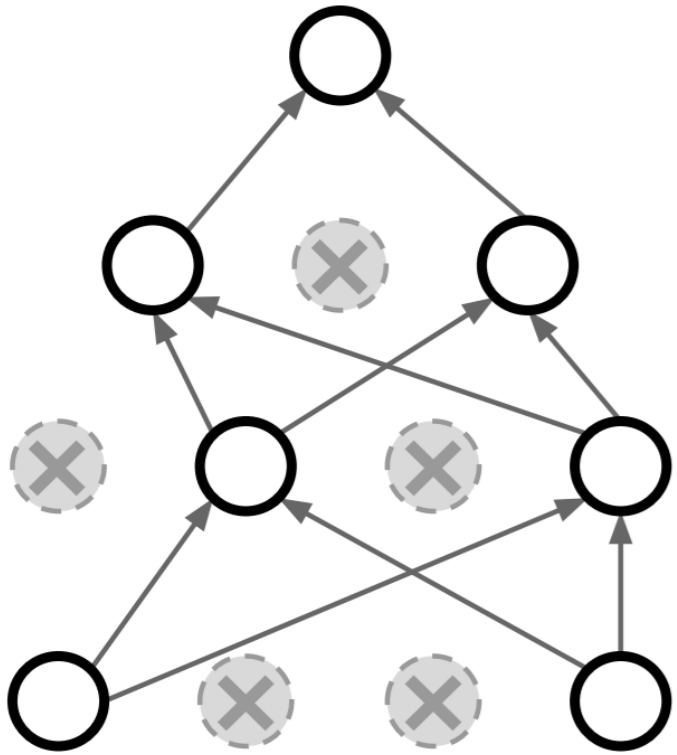


- Thus, shrinking the weights ($W^T W$) and shows similar effect to the L_2 regularization

[Andrew Ng, Deep Learning Specialization, 2017]

Regularization: Dropout

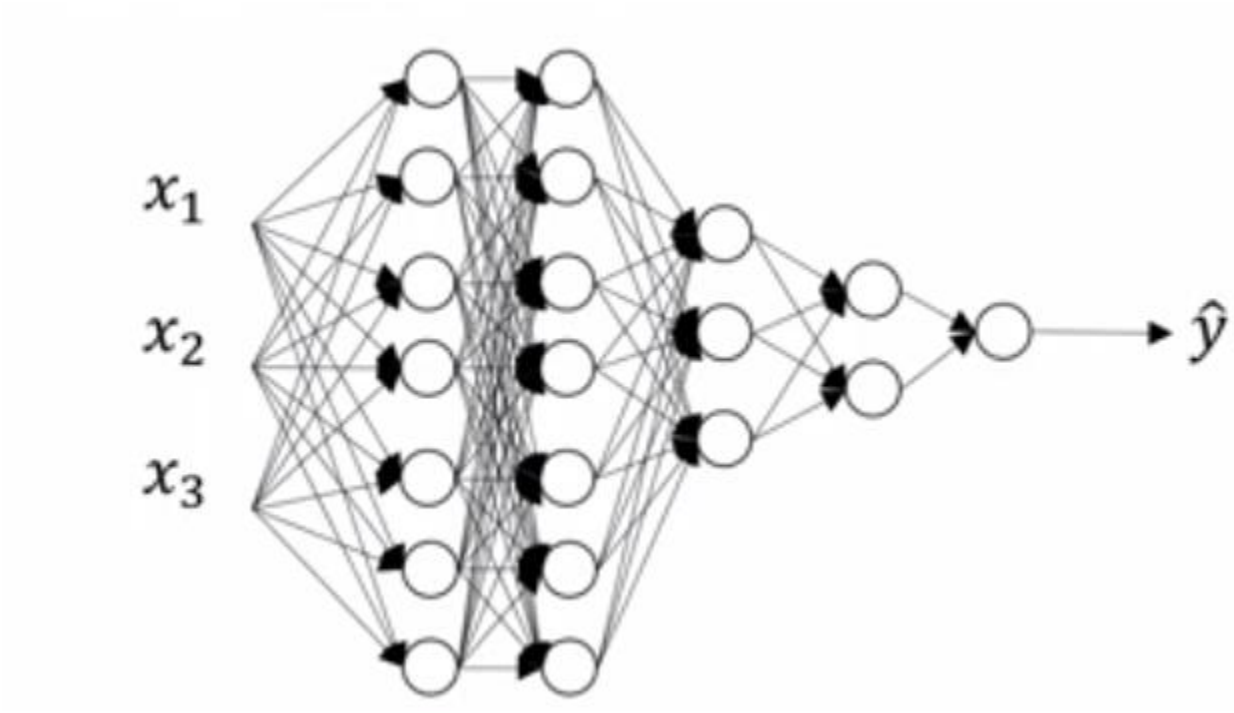
- How can dropout be a good idea?



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

Dropout on layers

- Different keep-prob can be used for different layers
  - However, more hyper-parameters are required



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)

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

Random mask

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

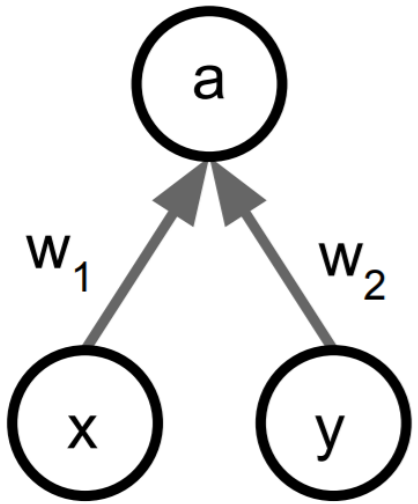
$$y = f(x) = E_z\left[f(x,z)\right] = \int p(z) f(x,z) dz$$

- But this integral seems hard …

Dropout: test time

Want to approximate the integral

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



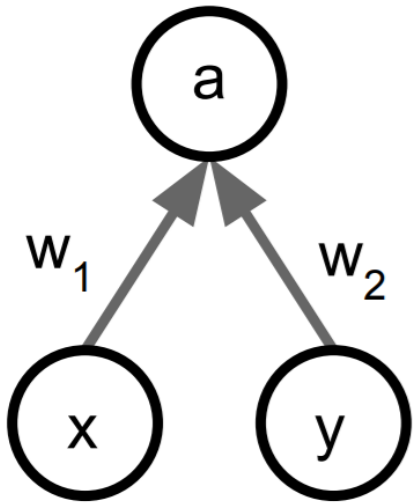
Consider a single neuron.

At test time we have: $E\left[a\right] = w_1 x + w_2 y$

Dropout: test time

Want to approximate the integral

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



Consider a single neuron.

At test time we have: $E\left[a\right] = w_1 x + w_2 y$

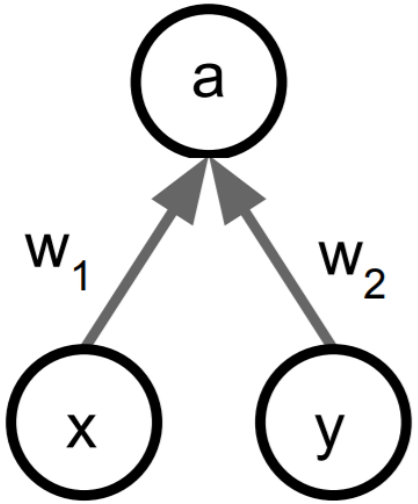
During training we have:

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

Dropout: test time

Want to approximate the integral

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



Consider a single neuron.

At test time we have: $E\left[a\right] = w_1 x + w_2 y$

During training 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, **multiply** by dropout probability

Making prediction at test time

- No dropout is used during test time

- With dropout, the expected output of neuron will become $p \times a + (1$

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
```

Dropout Summary

drop in forward pass

scale at test time

Inverted dropout

- Since test-time performance is so critical, it is always preferable to use inverted dropout,
  - which performs the scaling at train time, leaving the forward pass at test time untouched.
  - Just divide the activation of each layer by keep-prob

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!

Dropout issues

- If your NN is significantly overfitting, dropout usually help must to prevent overfitting
- Training takes longer time but provides better generalization
  - If your NN is not overfitting, bigger NN with dropout can help

- The cost function is no longer well defined.
- Gradcheck by turning off the dropout

Regularization: A common pattern

- Training: Add some kind of randomness Testing:

$$\hat{y} = f(x, z; W)$$

- Average out randomness (sometimes approximate)

$$\hat{y} = E_z[f(x, z; W)] = \int p(z) f(x, z; W) dz$$

Regularization: A common pattern

- Training: Add some kind of randomness Testing:

$$\hat{y} = f(x, z; W)$$

- Average out randomness (sometimes approximate)

$$\hat{y} = E_z[f(x, z; W)] = \int p(z) f(x, z; W) dz$$

Example: Batch Normalization

Training: Normalize using stats from random mini-batches

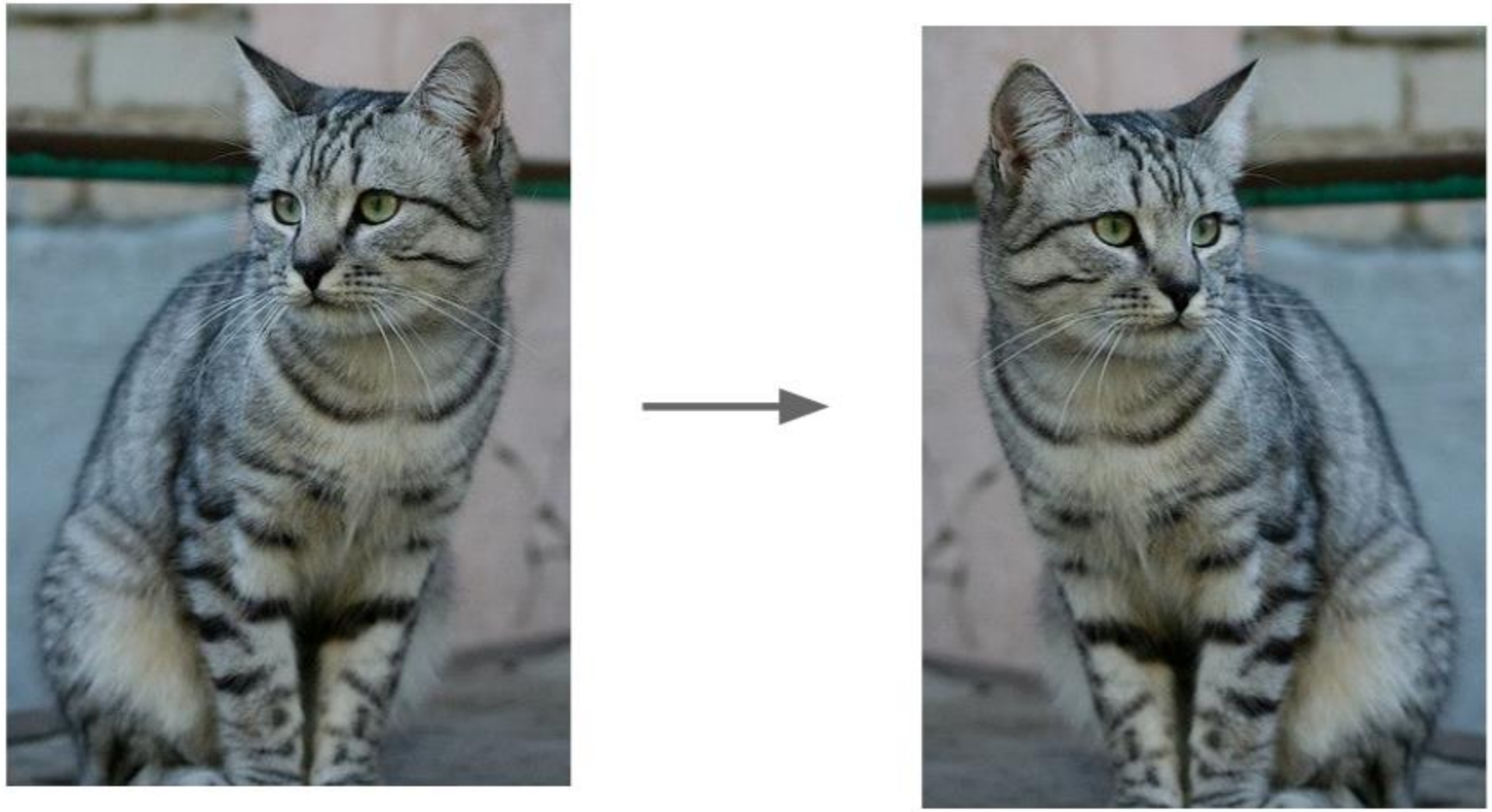
Testing: Use fixed stats to normalize

Data augmentation

- Getting more training data can be expensive
- But, sometimes we can generate more training examples from the datasets

Data augmentation

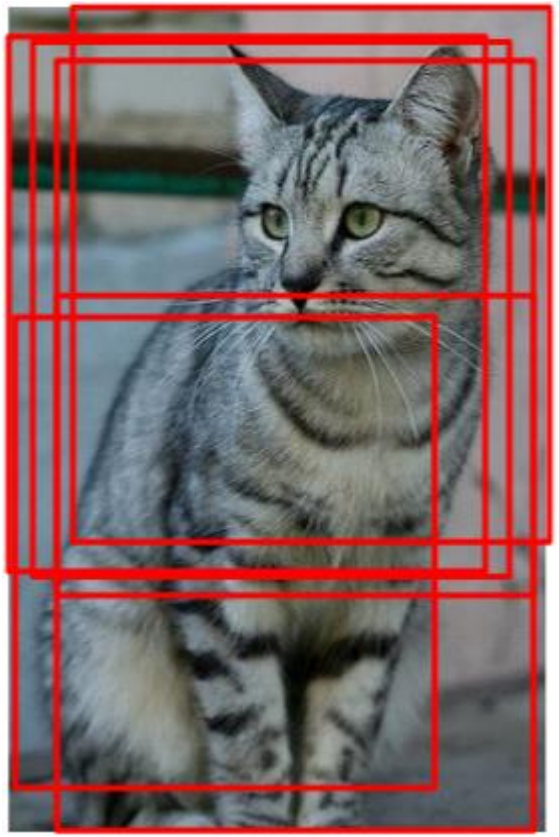
- Horizontal flip



Data augmentation

- Random crops and scales

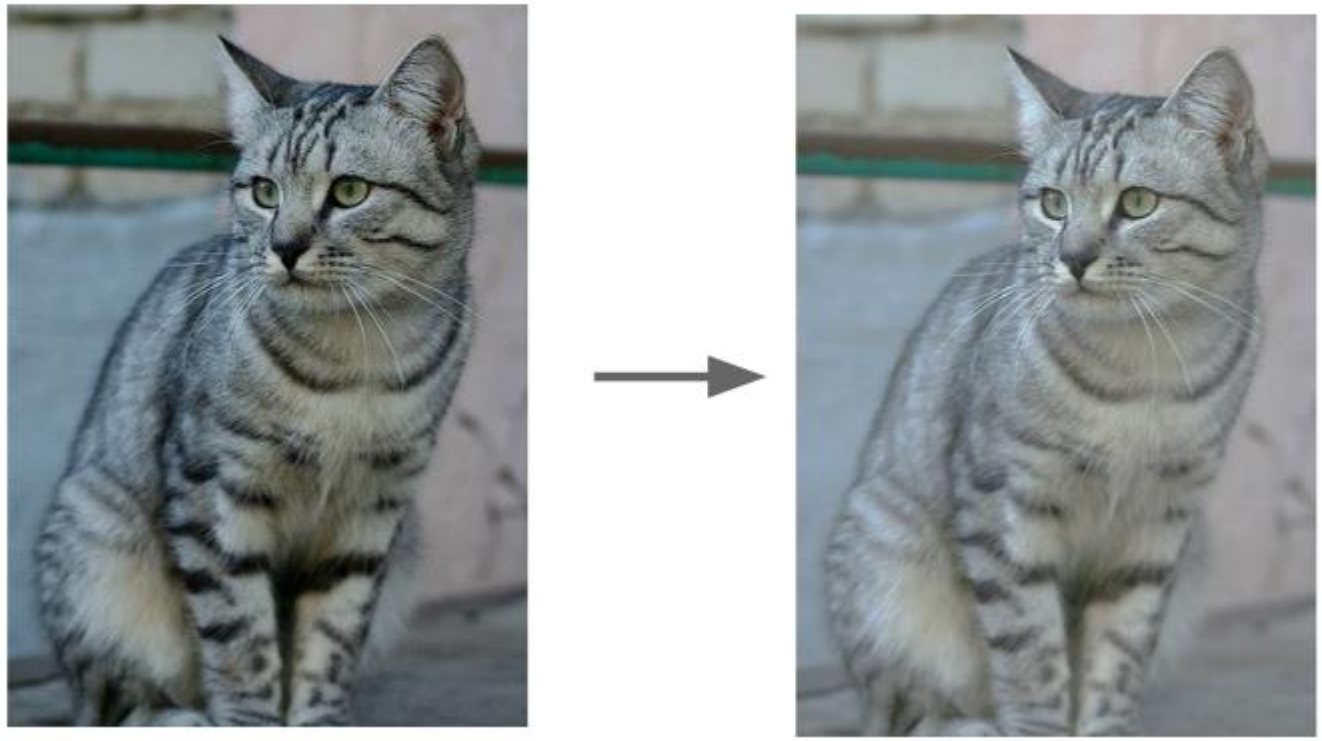
- Training: sample random crops / scales
  - ResNet:
    - 1. Pick random L in range [256, 480]
    - 2. Resize training image, short side = L
    - 3. Sample random 224 x 224 patch
- Testing: average a fixed set of crops
  - ResNet:
    - 1. Resize image at 5 scales: {224, 256, 384, 480, 640}
    - 2. For each size, use 10 224 x 224 crops: 4 corners + center, + flips



Data augmentation

- Color Jitter

Simple: Randomize contrast and brightness



Data augmentation

- Random mix/combinations of :
  - Translation
  - Rotation
  - Stretching
  - Shearing
  - …

Regularization: A common pattern

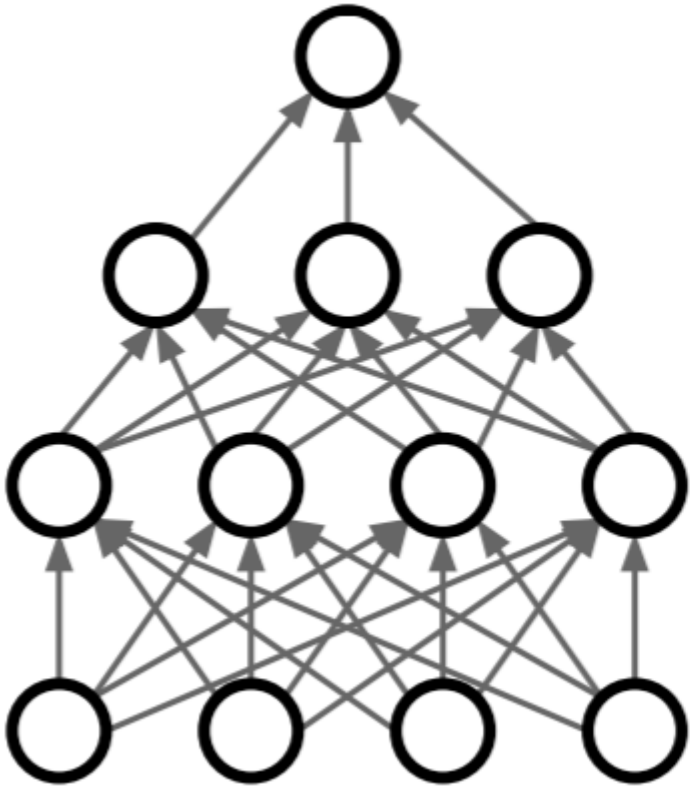
- Training: Add random noise
- Testing: Marginalize over the noise

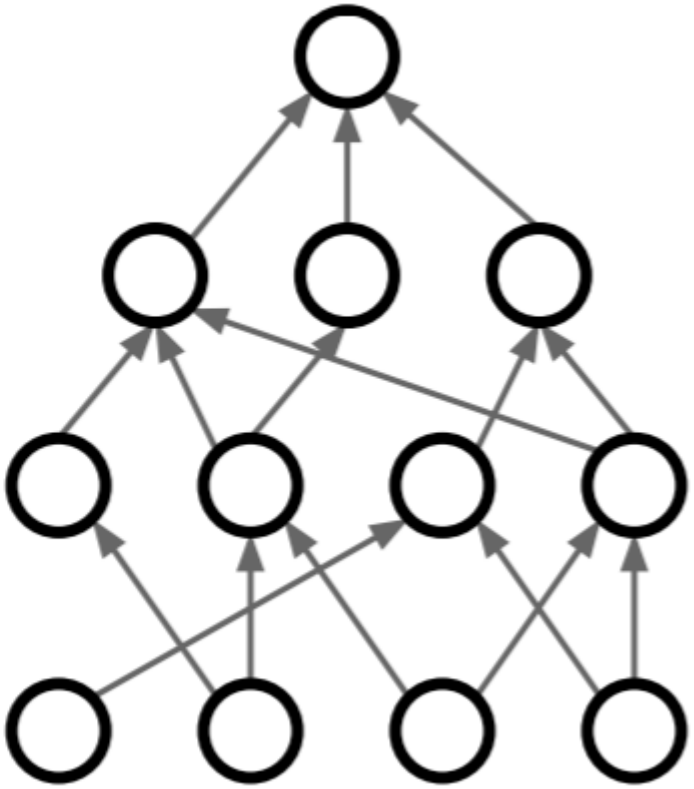
- Examples:
  - Dropout
  - Batch Normalization
  - Data Augmentation

Regularization: A common pattern

- Training: Add random noise
- Testing: Marginalize over the noise

- Examples:
  - Dropout
  - Batch Normalization
  - Data Augmentation
  - Dropconnect

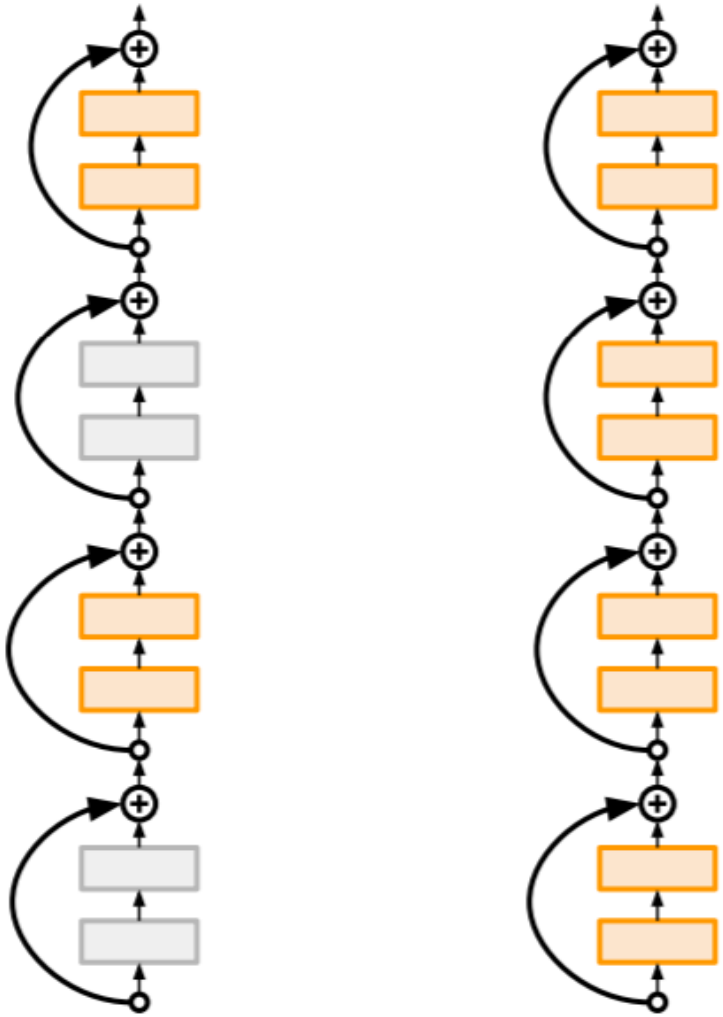




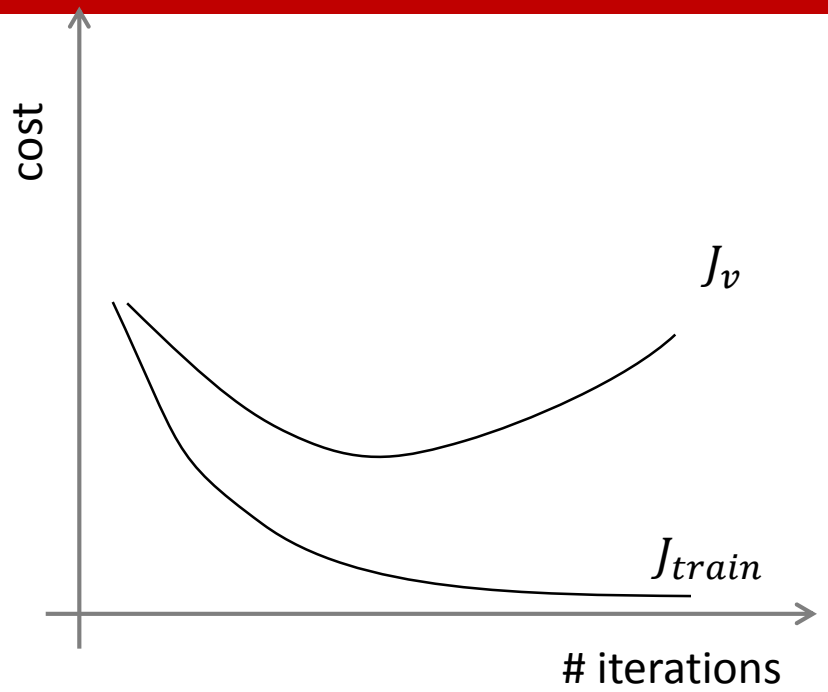
Regularization: A common pattern

- Training: Add random noise
- Testing: Marginalize over the noise

- Examples:
  - Dropout
  - Batch Normalization
  - Data Augmentation
  - Dropconnect
  - Stochastic depth



Early stopping



- Similar to L2 regularization

☹ Thinks about approximation and generalization tasks separately

- By stopping gradient descent early breaks whatever you are doing to optimize cost function

☺ Advantage is that it needs running gradient descent process just once

Model ensembles

1. Train multiple independent models

2. At test time average their results

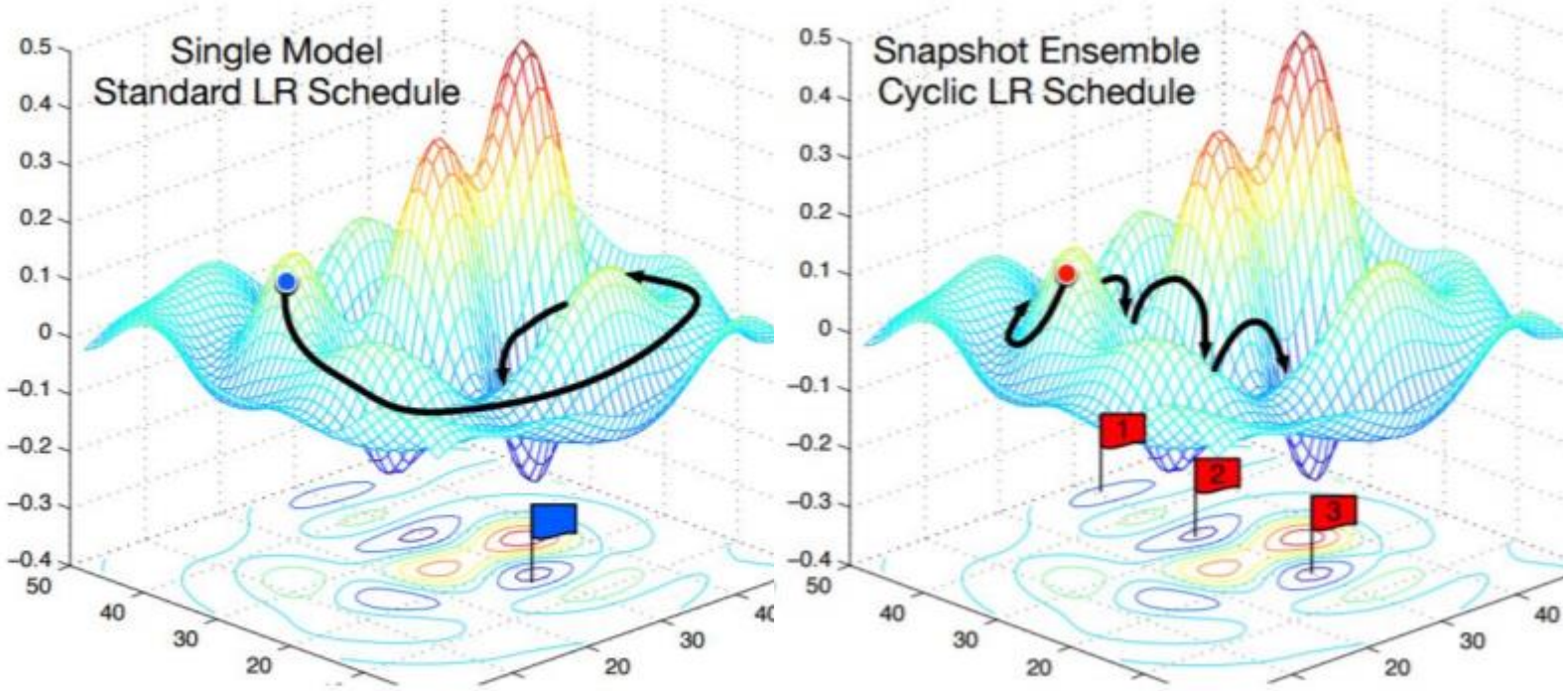
Model ensembles

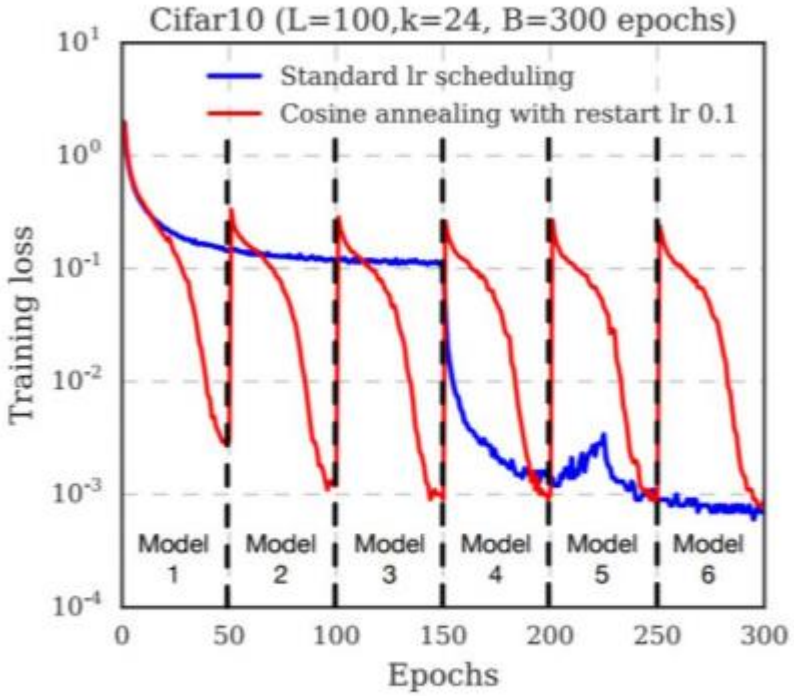
- Train multiple independent models, and at test time average their predictions.
  - improving the performance by a few percent

- Candidates
  - **Same model, different initializations** (after selecting the best hyperparameters**)**
  - **Top models discovered during cross-validation**
  - **Different checkpoints of a single model** (If training is very expensive)
    - maybe limited success to form an ensemble.
  - **Running average of parameters during training**
    - maintains an exponentially decaying sum of previous weights during training.

Model ensembles: Tips and tricks

- Instead of training independent models, use multiple snapshots of a single model during training!





Cyclic learning rate schedules can make this work even better!

Loshchilov and Hutter, "SGDR: Stochastic gradient descent with restarts", arXiv 2016
Huang et al, "Snapshot ensembles: train 1, get M for free", ICLR 2017

Model ensembles: Tips and tricks

- Instead of using actual parameter vector, keep a moving average of the parameter vector and use that at test time (Polyak averaging)

```
while True:
  data_batch = dataset.sample_data_batch()
  loss = network.forward(data_batch)
  dw = network.backward()
  w += - learning_rate * dw
  w_test = 0.995*w_test + 0.005*w # use for test set
```

Hyperparameter selection

- Finding a good set of hyperparameters to provide better convergence
  - initial learning rate α
  - regularization strength (L2 penalty, dropout strength)
  - # of hidden units and # of layers
  - mini-batch size
  - decay schedule (such as the decay constant), update type

  - parameters of optimization algorithms (momentum, adam, …)
    - These are usually fixed to $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\epsilon = 10^{-8}$ or 10^{-7}

Hyperparameter search

- Larger Neural Networks typically require a long time to train
  - so performing hyperparameter search can take many days/weeks

- A single validation set of respectable size substantially simplifies the code base, without the need for cross-validation with multiple folds

Cross-validation strategy

- **coarse -> fine** cross-validation in stages
  - Zoom in to smaller region of hyperparameters and sample very densely in them

- **First stage**: only a few epochs to get rough idea of what params work
- **Second stage**: longer running time, finer search … (repeat as necessary)

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)
```

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

Careful with best values on border

Random search vs. grid search

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

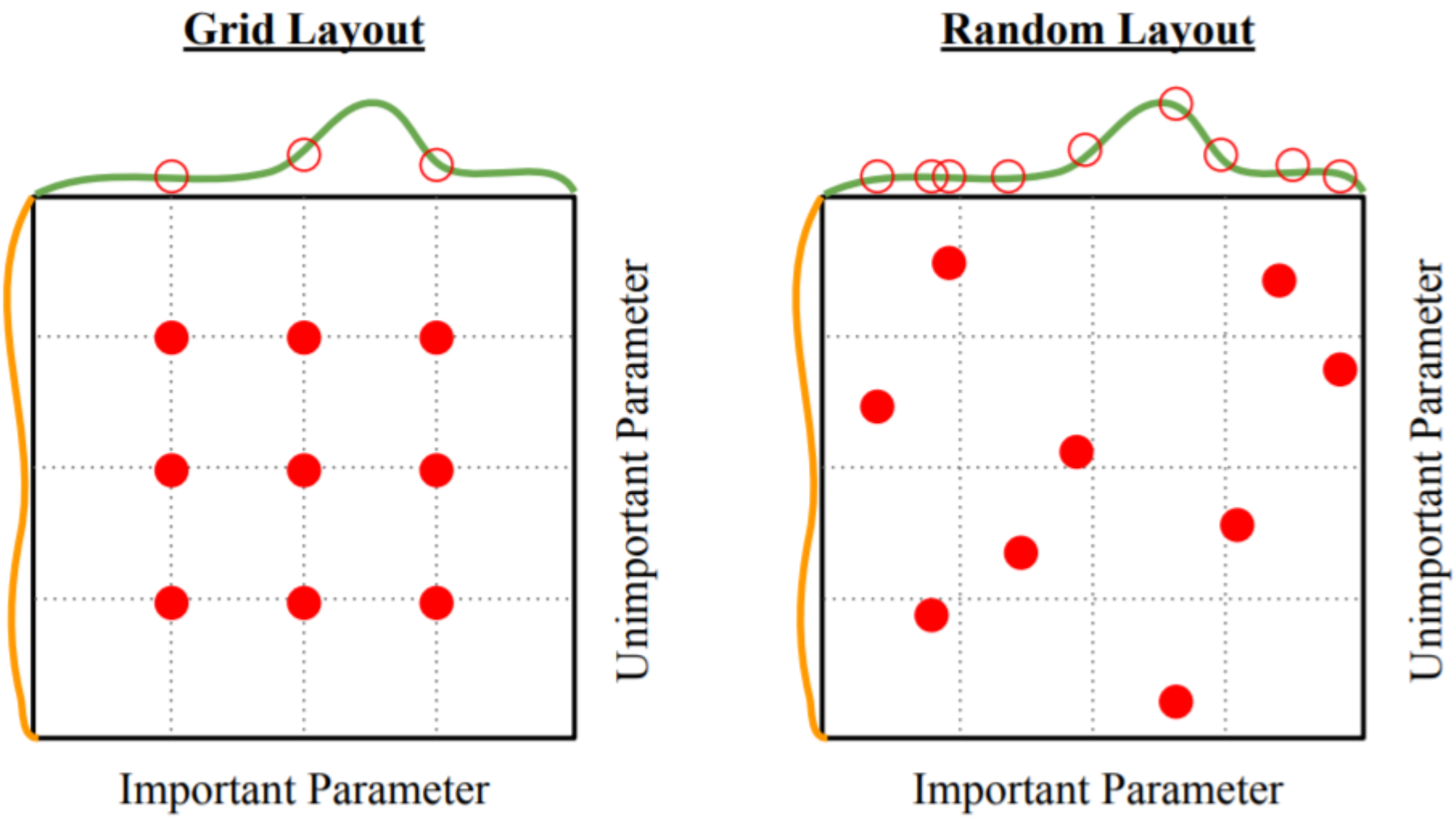


Illustration of Bergstra et al., 2012 by Shayne Longpre, copyright CS231n 2017

Random search

- Random search is more effective and richly exploring when:
  - The performance may not be such sensitive to the value of all hyperparameters (some parameters are actually much more important)
    - We don’t know it in advance
    - More distinct values of the more important hyperparameter are tried.
    - Especially when the number of hyperparameters becomes larger

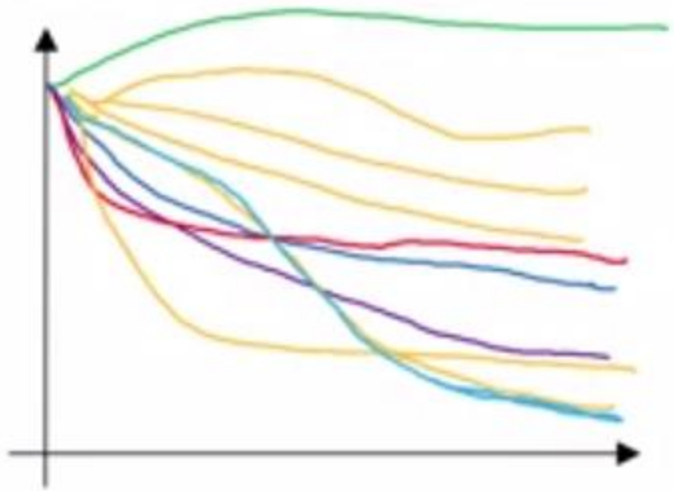
Babysitting one model vs. training models in parallel

- When we do not have sufficient computational resources to train a lot of models
  - Watching performance of one model during the time and tune its parameters by nudging them up and down

- Train many different models in parallel (with different hyperparameetrs) and just pick the one that works best

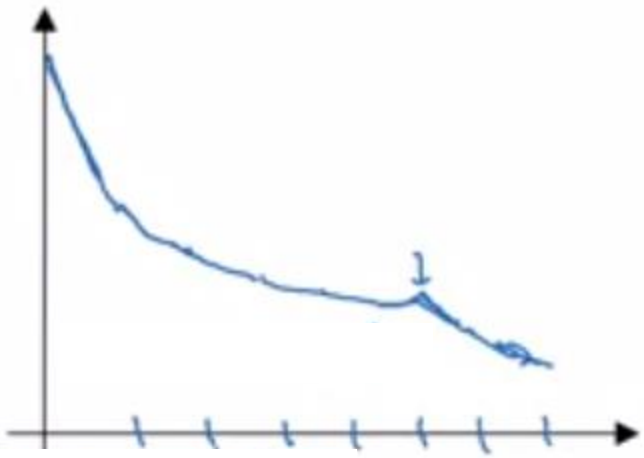
- Babysitting one model called Panda while training many models in parallel called Cavier approach for selecting hyperparameters

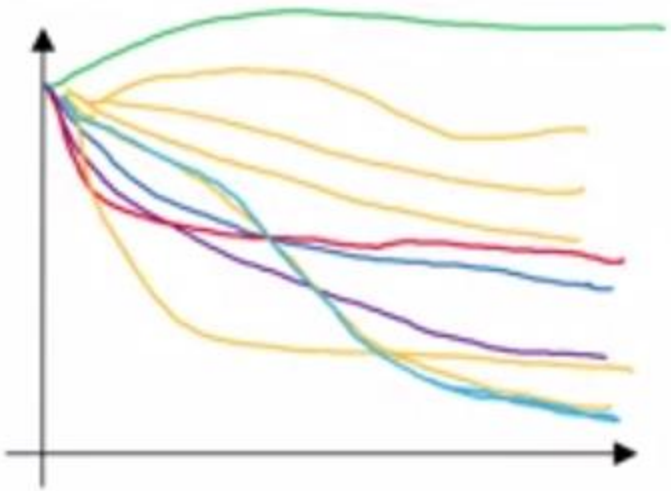
Babysitting one model vs. training models in parallel



[Andrew Ng, Deep Learning Specialization, 2017]

Babysitting one model vs. training models in parallel





Panda

Caviar

[Andrew Ng, Deep Learning Specialization, 2017]

Resources

- Deep Learning Book, Chapter 8.
- Please see the following note:
  - http://cs231n.github.io/neural-networks-3/