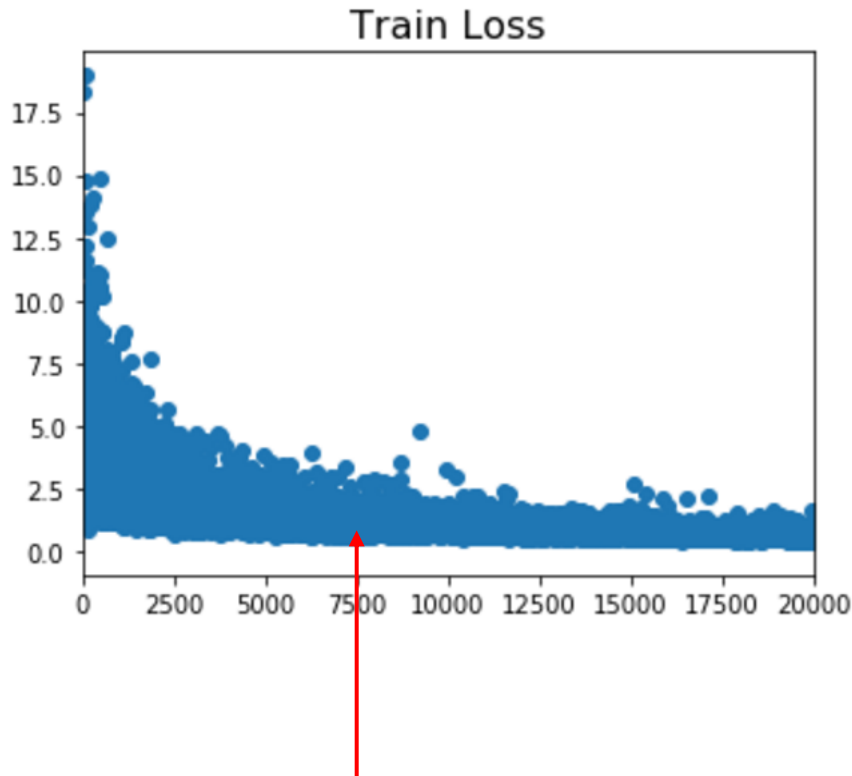


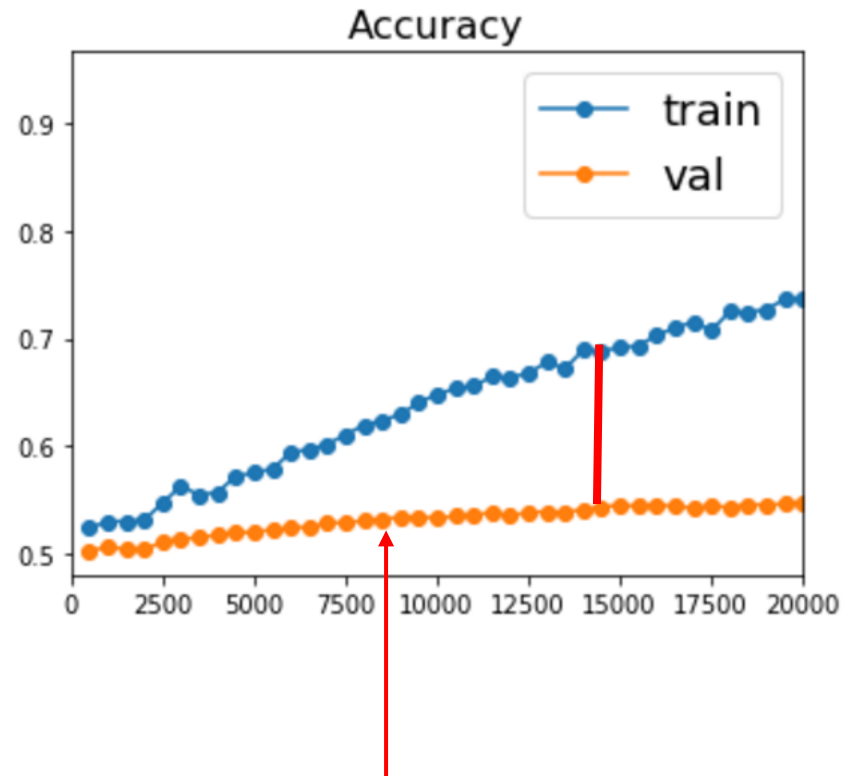
Pattern Recognition
Lecture 06-2
Deep Learning Visualization

Prof. Jongwon Choi
Chung-Ang University
Fall 2022

Beyond Training Error



Better optimization algorithms help reduce training loss



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

Model Ensembles

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

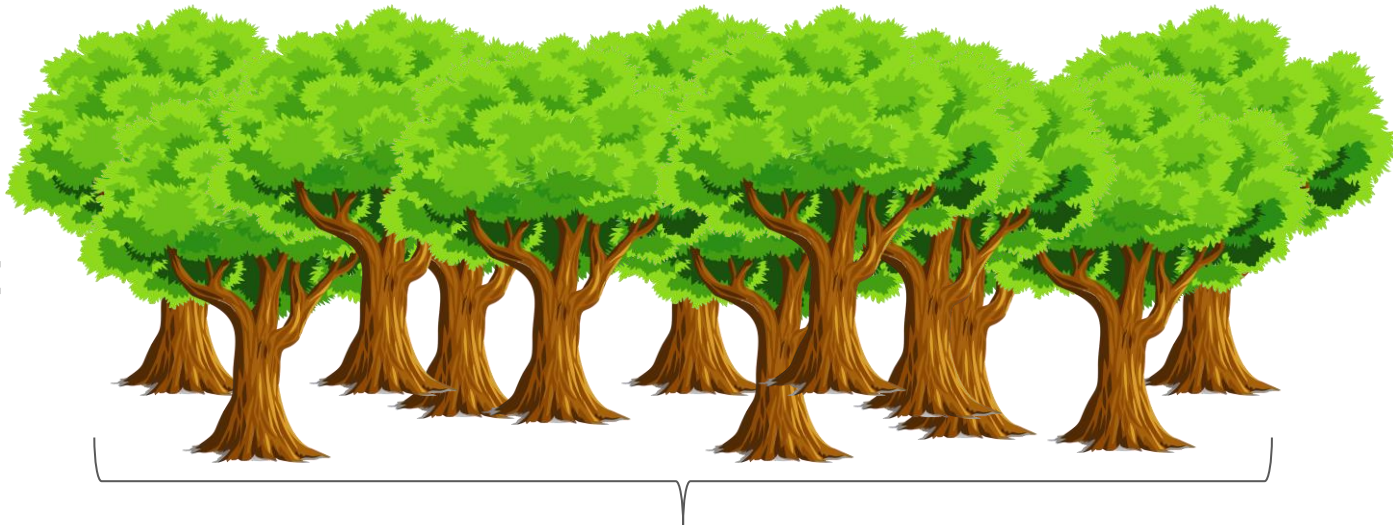
Enjoy 2% extra performance

Model Ensembles

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

Enjoy 2% extra performance

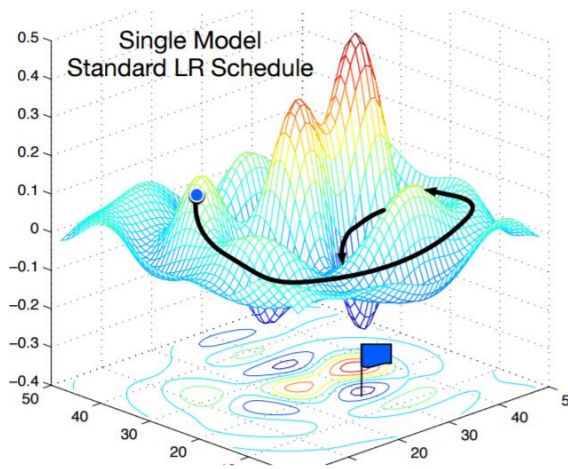
Recall:



Accuracy = **0.983**

Model Ensembles: Tips and Tricks

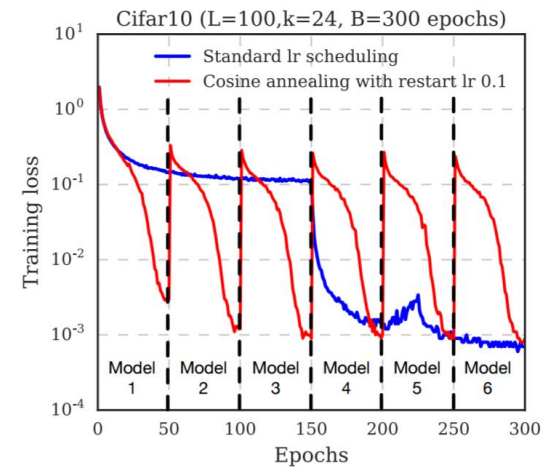
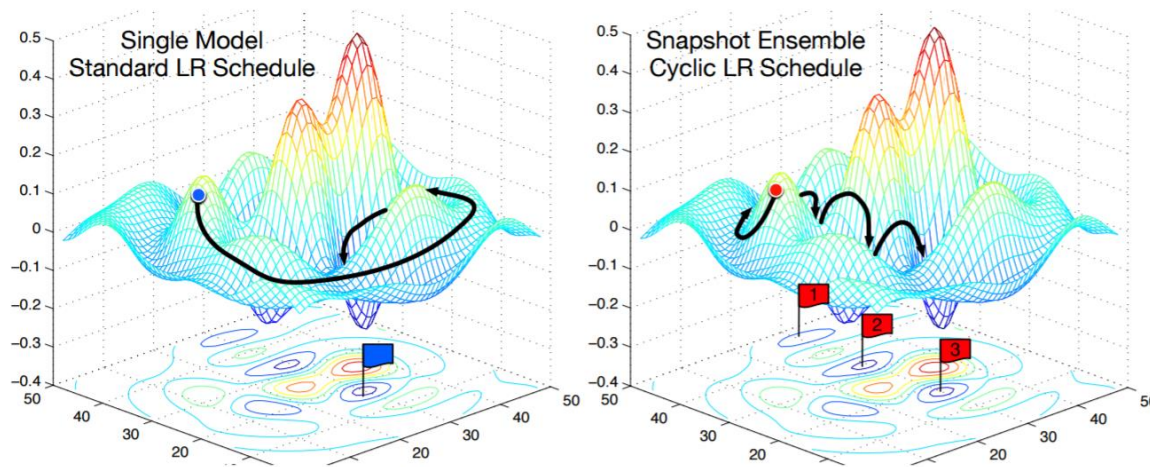
Instead of training independent models, use multiple snapshots of a single model during training!



Loshchilov and Hutter, "SGDR: Stochastic gradient descent with restarts", arXiv 2016
Huang et al, "Snapshot ensembles: train 1, get M for free", ICLR 2017
Figures copyright Yixuan Li and Geoff Pleiss, 2017. Reproduced with permission.

Model Ensembles: Tips and Tricks

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



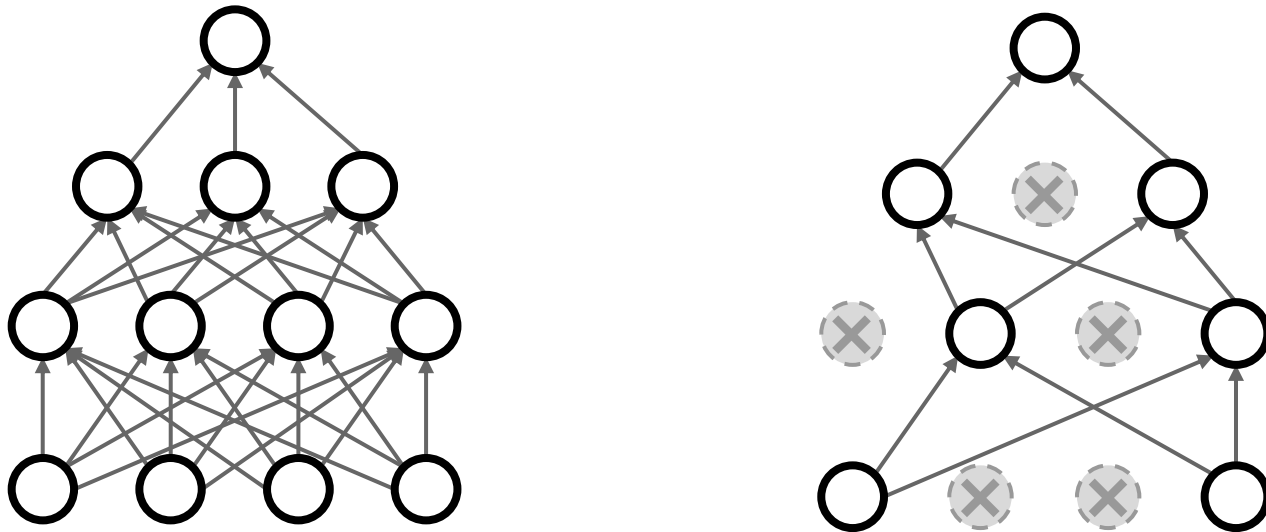
Cyclic learning rate schedules can make this work even better!

Quite new, try it out yourself!

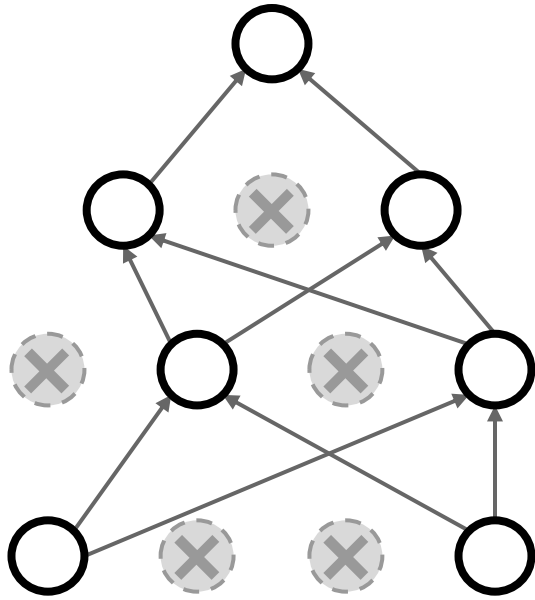
Loshchilov and Hutter, "SGDR: Stochastic gradient descent with restarts", arXiv 2016
Huang et al, "Snapshot ensembles: train 1, get M for free", ICLR 2017
Figures copyright Yixuan Li and Geoff Pleiss, 2017. Reproduced with permission.

Regularization: Dropout

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



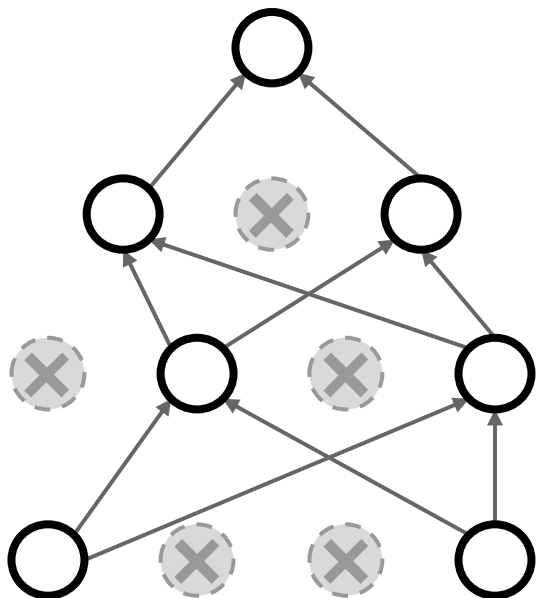
Regularization: Dropout



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



Regularization: Dropout



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...

Regularization: Dropout

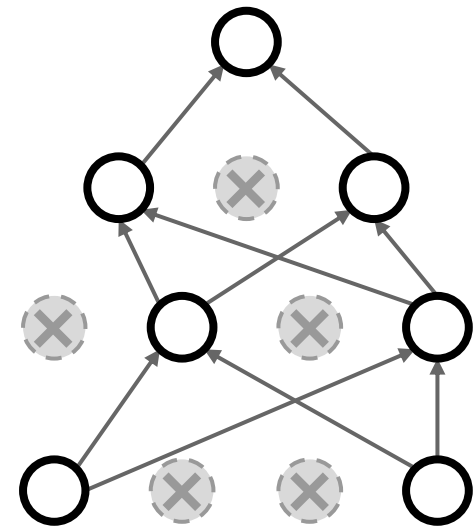
```
p = 0.5 # probability of keeping a unit active. higher = less dropout

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 at **test time**

Dropout makes our output random! $\overset{\text{Output (label)}}{\boxed{y}} = f_W(\overset{\text{Input (image)}}{\boxed{x}}, \overset{\text{Random mask}}{\boxed{z}})$

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

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

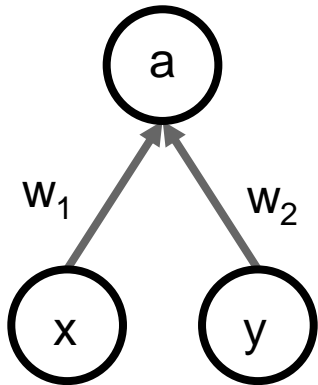
But this integral seems hard ...

Regularization: Dropout at 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: $E[a] = \frac{1}{4}(w_1x + w_2y) + \frac{1}{4}(w_1x + 0y) + \frac{1}{4}(0x + 0y) + \frac{1}{4}(0x + w_2y)$
 $= \frac{1}{2}(w_1x + w_2y)$

At test time, **multiply**
by dropout probability

Regularization: Dropout at 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

Regularization: Dropout (Vanilla)

```
""" 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 at train time

scale at test time

Regularization: Dropout (Inverted)

```
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
```

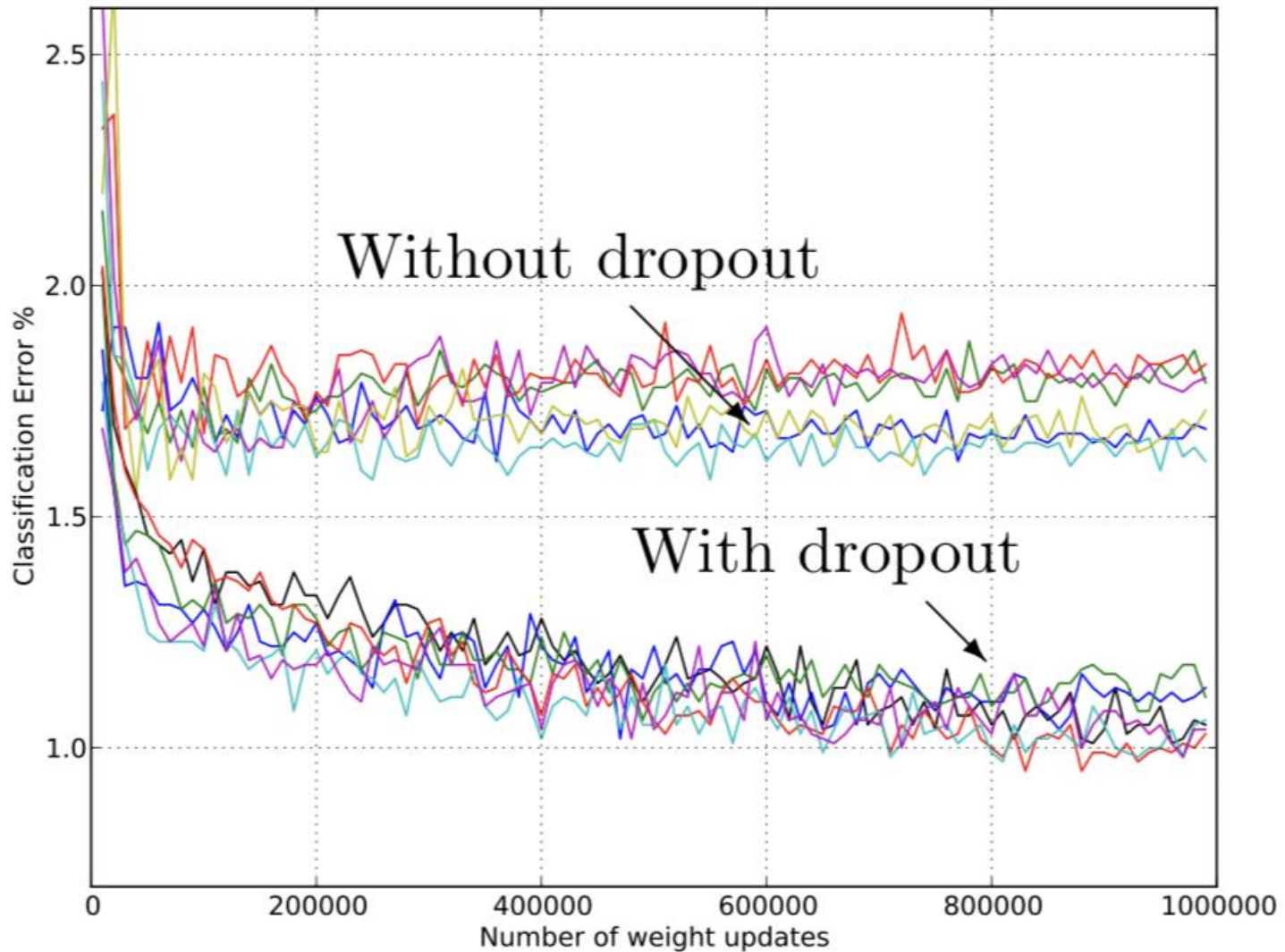
```
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
```

```
    out = np.dot(W3, H2) + b3
```

drop & scale at train time

test time is unchanged!

Regularization: Dropout



Figures copyright JLMR, 2014. Reproduced for educational purposes.

Check that your loss is reasonable

```
def init_two_layer_model(input_size, hidden_size, output_size):  
    # initialize a model  
    model = {}  
    model['W1'] = 0.0001 * np.random.randn(input_size, hidden_size)  
    model['b1'] = np.zeros(hidden_size)  
    model['W2'] = 0.0001 * np.random.randn(hidden_size, output_size)  
    model['b2'] = np.zeros(output_size)  
    return model
```

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes  
loss, grad = two_layer_net(X_train, model, y_train, 0.0) # disable regularization  
print loss
```

returns the loss and the
gradient for all parameters

Q: What should your
loss values be?

$$L_i = -\log\left(\frac{e^{sy_i}}{\sum_j e^{s_j}}\right)$$

Check that your loss is reasonable

```
def init_two_layer_model(input_size, hidden_size, output_size):  
    # initialize a model  
    model = {}  
    model['W1'] = 0.0001 * np.random.randn(input_size, hidden_size)  
    model['b1'] = np.zeros(hidden_size)  
    model['W2'] = 0.0001 * np.random.randn(hidden_size, output_size)  
    model['b2'] = np.zeros(output_size)  
    return model
```

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes  
loss, grad = two_layer_net(X_train, model, y_train, 0.0) # disable regularization  
print loss
```

2.30261216167

returns the loss and the
gradient for all parameters

For cross entropy,
loss $\sim 2.3 = \log_{10}$
“correct” for 10 classes

Check that your loss is reasonable

```
def init_two_layer_model(input_size, hidden_size, output_size):  
    # initialize a model  
    model = {}  
    model['W1'] = 0.0001 * np.random.randn(input_size, hidden_size)  
    model['b1'] = np.zeros(hidden_size)  
    model['W2'] = 0.0001 * np.random.randn(hidden_size, output_size)  
    model['b2'] = np.zeros(output_size)  
    return model
```

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes  
loss, grad = two_layer_net(X_train, model, y_train, 1e3) # crank up regularization  
print loss
```

3.06859716482

← loss went up, good. (sanity check)

Training

Let's try to train now...

Tip: Make sure that you can overfit very small portion of the training data

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
X_tiny = X_train[:20] # take 20 examples
y_tiny = y_train[:20]
best_model, stats = trainer.train(X_tiny, y_tiny, X_tiny, y_tiny,
                                  model, two_layer_net,
                                  num_epochs=200, reg=0.0,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = False,
                                  learning_rate=1e-3, verbose=True)
```

The above code:

- take the first 20 examples from CIFAR-10
- turn off regularization (reg = 0.0)
- use simple vanilla 'sgd'

Training

Let's try to train now...

Tip: Make sure that you can overfit very small portion of the training data

Very small loss, train accuracy 1.00, nice!

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
X_tiny = X_train[:20] # take 20 examples
y_tiny = y_train[:20]
best_model, stats = trainer.train(X_tiny, y_tiny, X_tiny, y_tiny,
                                  model, two_layer_net,
                                  num_epochs=200, reg=0.0,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = False,
                                  learning_rate=1e-3, verbose=True)
```

```
Finished epoch 1 / 200: cost 2.302603, train: 0.400000, val 0.400000, lr 1.000000e-03
Finished epoch 2 / 200: cost 2.302258, train: 0.450000, val 0.450000, lr 1.000000e-03
Finished epoch 3 / 200: cost 2.301849, train: 0.600000, val 0.600000, lr 1.000000e-03
Finished epoch 4 / 200: cost 2.301196, train: 0.650000, val 0.650000, lr 1.000000e-03
Finished epoch 5 / 200: cost 2.300044, train: 0.650000, val 0.650000, lr 1.000000e-03
Finished epoch 6 / 200: cost 2.297864, train: 0.550000, val 0.550000, lr 1.000000e-03
Finished epoch 7 / 200: cost 2.293595, train: 0.600000, val 0.600000, lr 1.000000e-03
Finished epoch 8 / 200: cost 2.285096, train: 0.550000, val 0.550000, lr 1.000000e-03
Finished epoch 9 / 200: cost 2.268094, train: 0.550000, val 0.550000, lr 1.000000e-03
Finished epoch 10 / 200: cost 2.234787, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 11 / 200: cost 2.173187, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 12 / 200: cost 2.076862, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 13 / 200: cost 1.974090, train: 0.400000, val 0.400000, lr 1.000000e-03
Finished epoch 14 / 200: cost 1.895885, train: 0.400000, val 0.400000, lr 1.000000e-03
Finished epoch 15 / 200: cost 1.820876, train: 0.450000, val 0.450000, lr 1.000000e-03
Finished epoch 16 / 200: cost 1.737430, train: 0.450000, val 0.450000, lr 1.000000e-03
Finished epoch 17 / 200: cost 1.642356, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 18 / 200: cost 1.535239, train: 0.600000, val 0.600000, lr 1.000000e-03
Finished epoch 19 / 200: cost 1.421527, train: 0.600000, val 0.600000, lr 1.000000e-03
Finished epoch 20 / 200: cost 1.297760, train: 0.650000, val 0.650000, lr 1.000000e-03
...
Finished epoch 195 / 200: cost 0.002694, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 196 / 200: cost 0.002674, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 197 / 200: cost 0.002655, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 198 / 200: cost 0.002635, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 199 / 200: cost 0.002617, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 200 / 200: cost 0.002597, train: 1.000000, val 1.000000, lr 1.000000e-03
finished optimization. best validation accuracy: 1.000000
```

Training

Let's try to train now...

Start with small regularization and find learning rate that makes the loss go down.

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = True,
                                  learning_rate=1e-6, verbose=True)
```

Training

Let's try to train now...

Start with small regularization and find learning rate that makes the loss go down.

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = True,
                                  learning_rate=1e-6, verbose=True)
```

Finished epoch 1 / 10:	cost 2.302576,	train: 0.080000, val 0.103000, lr 1.000000e-06
Finished epoch 2 / 10:	cost 2.302582,	train: 0.121000, val 0.124000, lr 1.000000e-06
Finished epoch 3 / 10:	cost 2.302558,	train: 0.119000, val 0.138000, lr 1.000000e-06
Finished epoch 4 / 10:	cost 2.302519,	train: 0.127000, val 0.151000, lr 1.000000e-06
Finished epoch 5 / 10:	cost 2.302517,	train: 0.158000, val 0.171000, lr 1.000000e-06
Finished epoch 6 / 10:	cost 2.302518,	train: 0.179000, val 0.172000, lr 1.000000e-06
Finished epoch 7 / 10:	cost 2.302466,	train: 0.180000, val 0.176000, lr 1.000000e-06
Finished epoch 8 / 10:	cost 2.302452,	train: 0.175000, val 0.185000, lr 1.000000e-06
Finished epoch 9 / 10:	cost 2.302459,	train: 0.206000, val 0.192000, lr 1.000000e-06
Finished epoch 10 / 10:	cost 2.302420,	train: 0.190000, val 0.192000, lr 1.000000e-06
finished optimization.	best validation	accuracy: 0.192000

Loss barely changing

Training

Let's try to train now...

Start with small regularization and find learning rate that makes the loss go down.

loss not going down:
learning rate too low

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = True,
                                  learning_rate=1e-6, verbose=True)
```

Finished epoch 1 / 10:	cost 2.302576,	train: 0.080000, val 0.103000, lr 1.000000e-06
Finished epoch 2 / 10:	cost 2.302582,	train: 0.121000, val 0.124000, lr 1.000000e-06
Finished epoch 3 / 10:	cost 2.302558,	train: 0.119000, val 0.138000, lr 1.000000e-06
Finished epoch 4 / 10:	cost 2.302519,	train: 0.127000, val 0.151000, lr 1.000000e-06
Finished epoch 5 / 10:	cost 2.302517,	train: 0.158000, val 0.171000, lr 1.000000e-06
Finished epoch 6 / 10:	cost 2.302518,	train: 0.179000, val 0.172000, lr 1.000000e-06
Finished epoch 7 / 10:	cost 2.302466,	train: 0.180000, val 0.176000, lr 1.000000e-06
Finished epoch 8 / 10:	cost 2.302452,	train: 0.175000, val 0.185000, lr 1.000000e-06
Finished epoch 9 / 10:	cost 2.302459,	train: 0.206000, val 0.192000, lr 1.000000e-06
Finished epoch 10 / 10:	cost 2.302420,	train: 0.190000, val 0.192000, lr 1.000000e-06
finished optimization.	best validation	accuracy: 0.192000

Loss barely changing: Learning rate is probably too low

Training

Let's try to train now...

Start with small regularization and find learning rate that makes the loss go down.

loss not going down:
learning rate too low

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = True,
                                  learning_rate=1e-6, verbose=True)
```

Epoch	Cost	Train Accuracy	Val Accuracy	Learning Rate
1	2.302576	0.080000	0.103000	1.000000e-06
2	2.302582	0.121000	0.124000	1.000000e-06
3	2.302558	0.119000	0.138000	1.000000e-06
4	2.302519	0.127000	0.151000	1.000000e-06
5	2.302517	0.158000	0.171000	1.000000e-06
6	2.302518	0.179000	0.172000	1.000000e-06
7	2.302466	0.180000	0.176000	1.000000e-06
8	2.302452	0.175000	0.185000	1.000000e-06
9	2.302459	0.206000	0.192000	1.000000e-06
10	2.302420	0.190000	0.192000	1.000000e-06

finished optimization. best validation accuracy: 0.192000

Loss barely changing: Learning rate is probably too low

Notice train/val accuracy goes to 20% though, what's up with that? (remember this is softmax)

Training

Lets try to train now...

Start with small regularization and find learning rate that makes the loss go down.

loss not going down:
learning rate too low

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = True,
                                  learning_rate=1e6, verbose=True)
```

Now let's try learning rate 1e6.

Training

Lets try to train now...

Start with small regularization and find learning rate that makes the loss go down.

loss not going down:
learning rate too low
loss exploding:
learning rate too high

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = True,
                                  learning_rate=1e6, verbose=True)
```

```
/home/karpathy/cs231n/code/cs231n/classifiers/neural_net.py:50: RuntimeWarning: divide by zero en
countered in log
  data_loss = -np.sum(np.log(probs[range(N), y])) / N
/home/karpathy/cs231n/code/cs231n/classifiers/neural_net.py:48: RuntimeWarning: invalid value enc
ountered in subtract
  probs = np.exp(scores - np.max(scores, axis=1, keepdims=True))
Finished epoch 1 / 10: cost nan, train: 0.091000, val 0.087000, lr 1.000000e+06
Finished epoch 2 / 10: cost nan, train: 0.095000, val 0.087000, lr 1.000000e+06
Finished epoch 3 / 10: cost nan, train: 0.100000, val 0.087000, lr 1.000000e+06
```

cost: NaN almost
always means high
learning rate...

Training

Lets try to train now...

Start with small regularization and find learning rate that makes the loss go down.

loss not going down:
learning rate too low
loss exploding:
learning rate too high

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = True,
                                  learning_rate=3e-3, verbose=True)
```

```
Finished epoch 1 / 10: cost 2.186654, train: 0.308000, val 0.306000, lr 3.000000e-03
Finished epoch 2 / 10: cost 2.176230, train: 0.330000, val 0.350000, lr 3.000000e-03
Finished epoch 3 / 10: cost 1.942257, train: 0.376000, val 0.352000, lr 3.000000e-03
Finished epoch 4 / 10: cost 1.827868, train: 0.329000, val 0.310000, lr 3.000000e-03
Finished epoch 5 / 10: cost inf, train: 0.128000, val 0.128000, lr 3.000000e-03
Finished epoch 6 / 10: cost inf, train: 0.144000, val 0.147000, lr 3.000000e-03
```

3e-3 is still too high. Cost explodes....

=> Rough range for learning rate we should be cross-validating is somewhere [1e-3 ... 1e-5]

Hyperparameter Optimization

The pain and the glory

Validation strategy

coarse -> fine validation in stages

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

Tip for detecting explosions in the solver:

If the cost is ever $> 3 * \text{original cost}$, break out early

Example: 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



Example: fine search

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



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


Example: fine search

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



```
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.

Example: fine search

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



```
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 validation result is worrying. Why?

Random Search vs. Grid Search

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

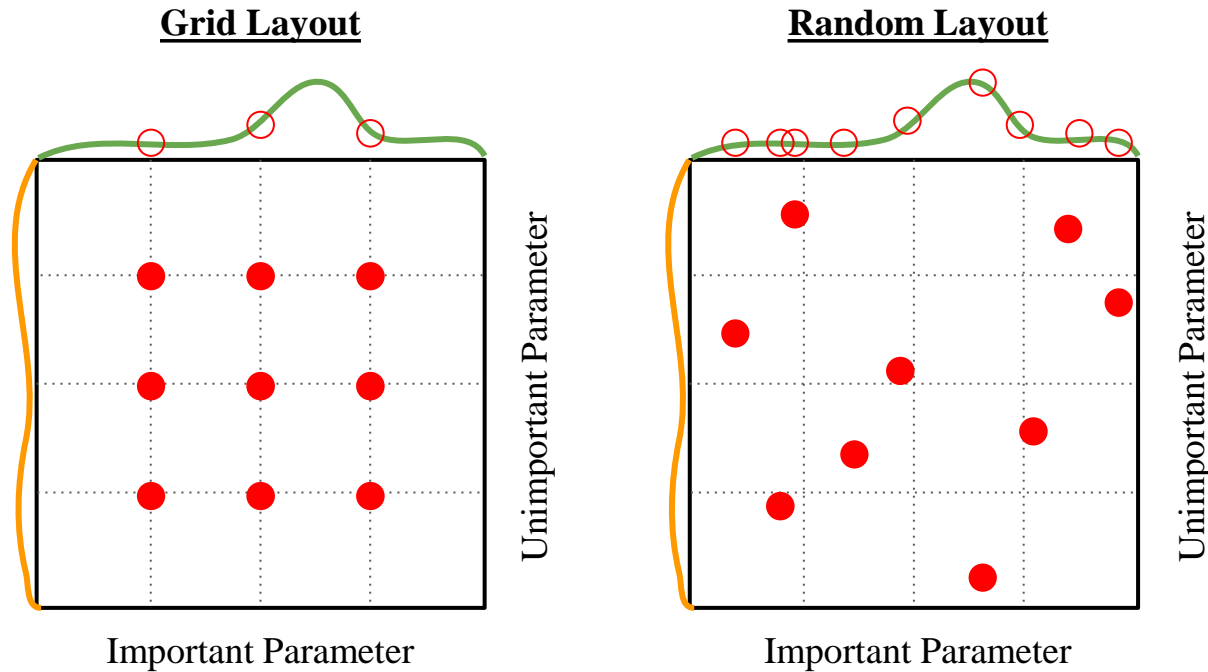
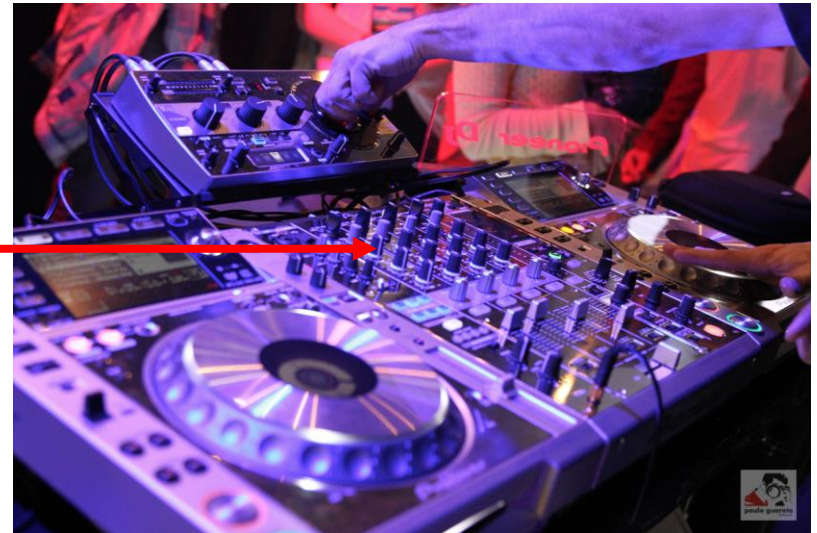


Illustration of Bergstra et al., 2012 by Shayne Longpre, copyright CS231n 2017

Typical hyperparameters

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

neural networks practitioner
music = loss function

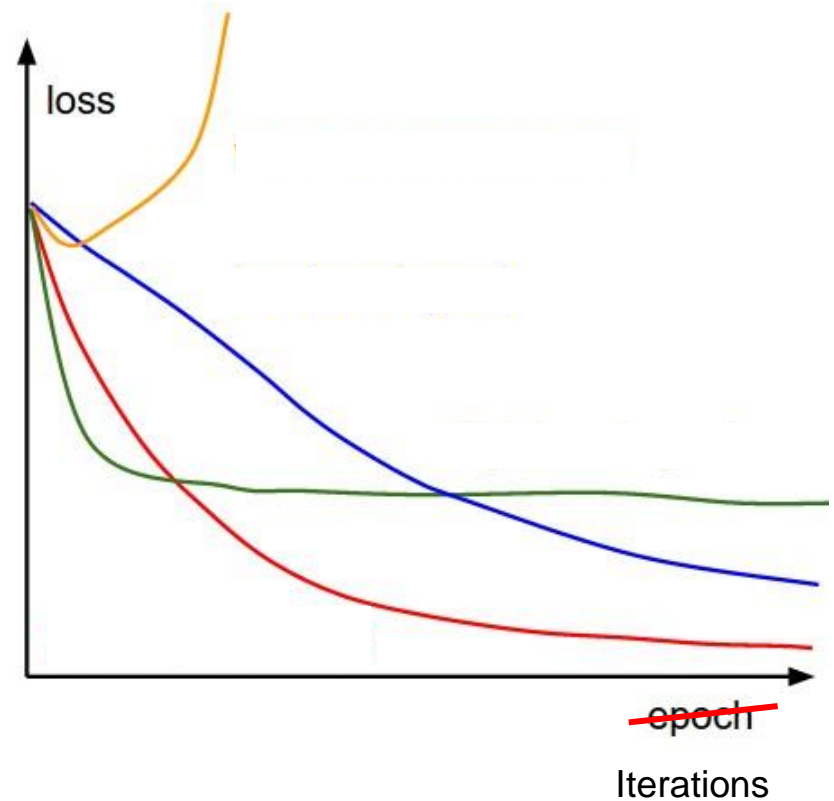
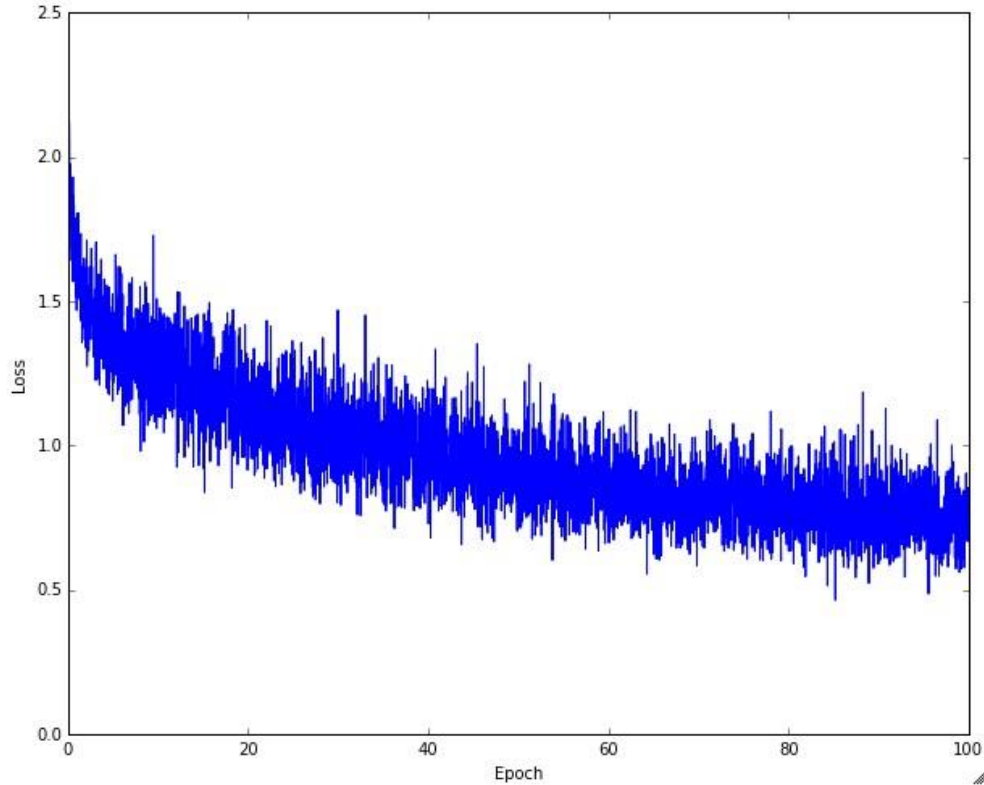


[This image](#) by Paolo Guereta is licensed under [CC-BY 2.0](#)

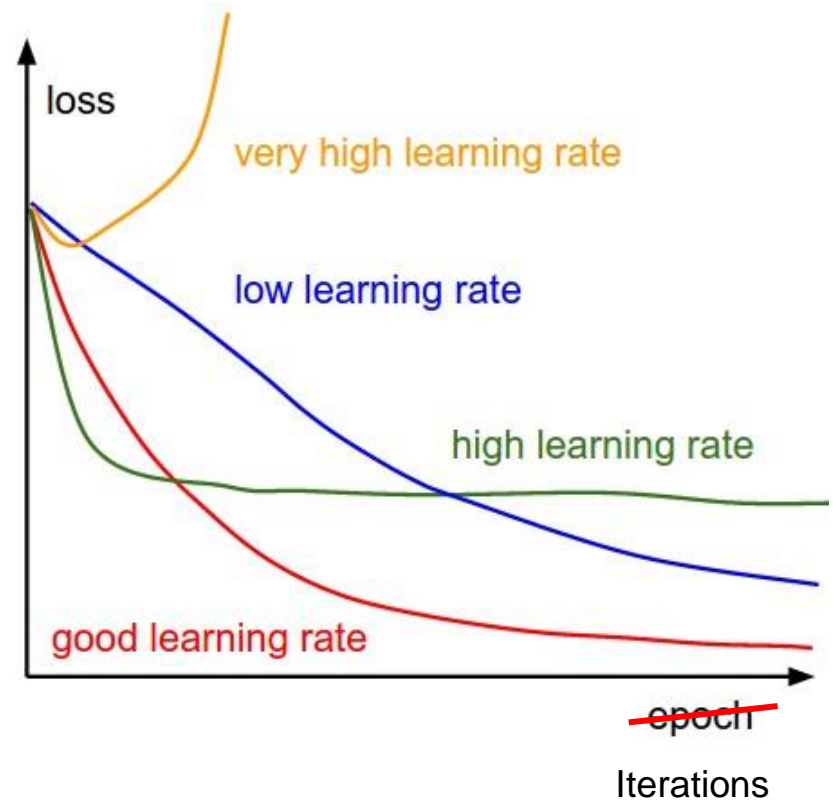
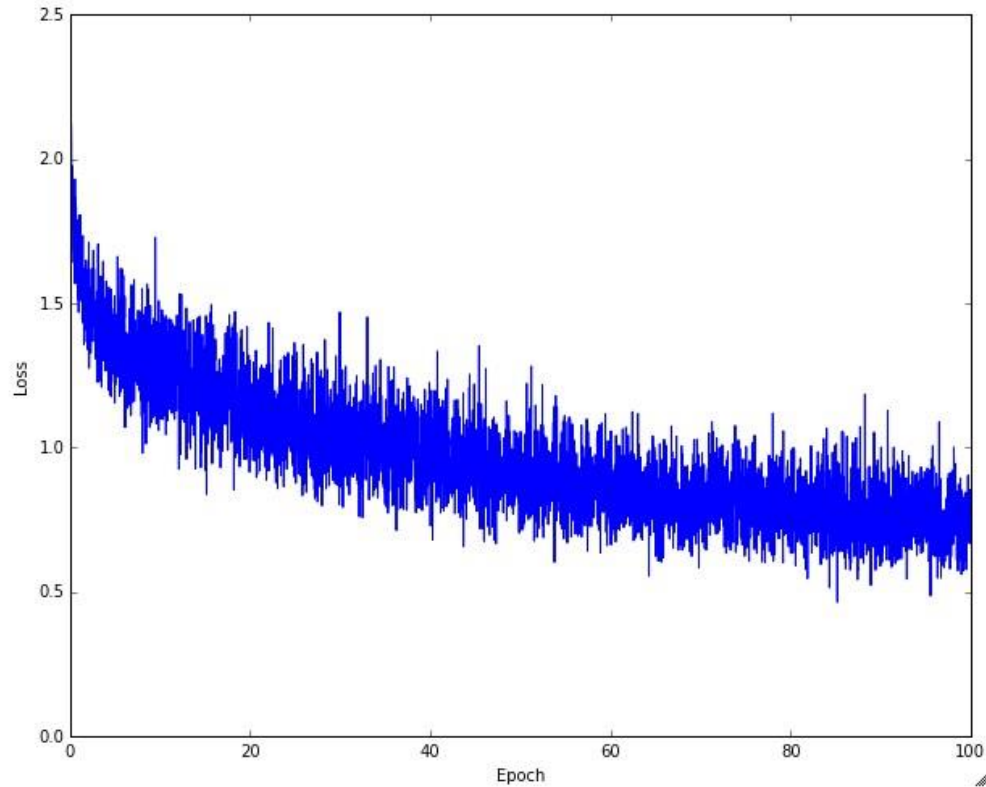
Typical situation



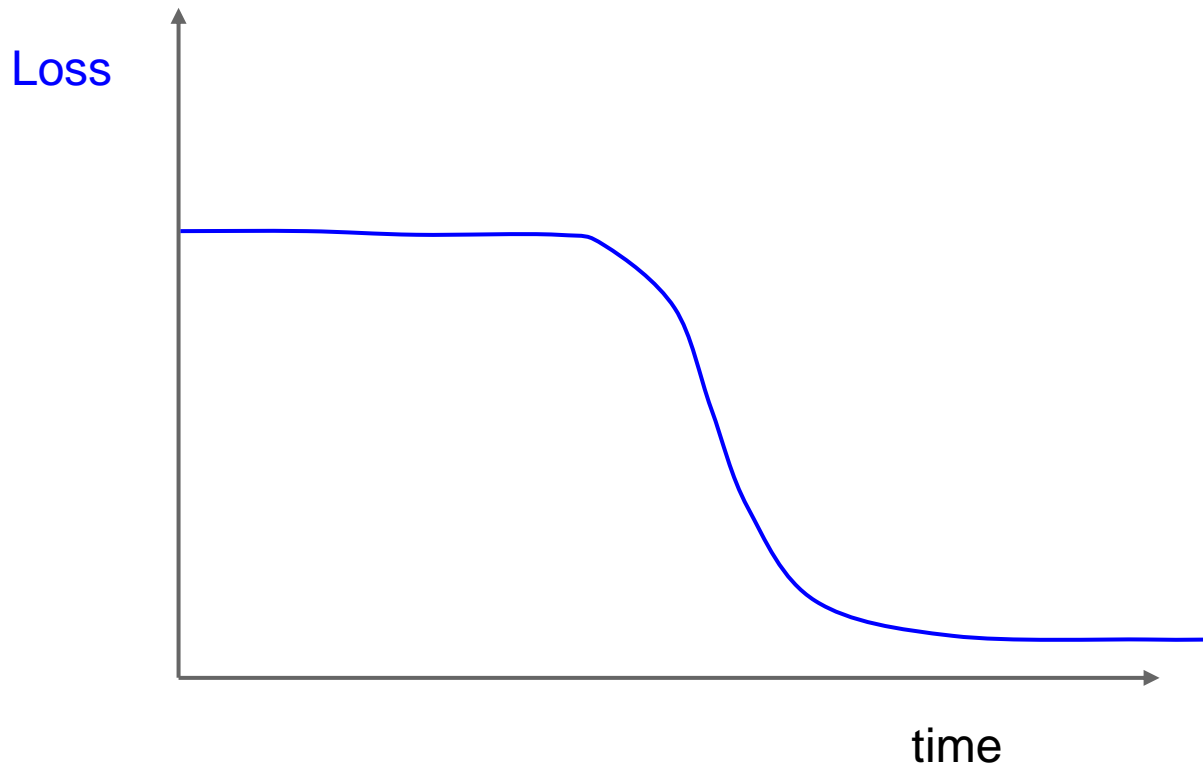
Visualize VISUALIZE VISUALIZE



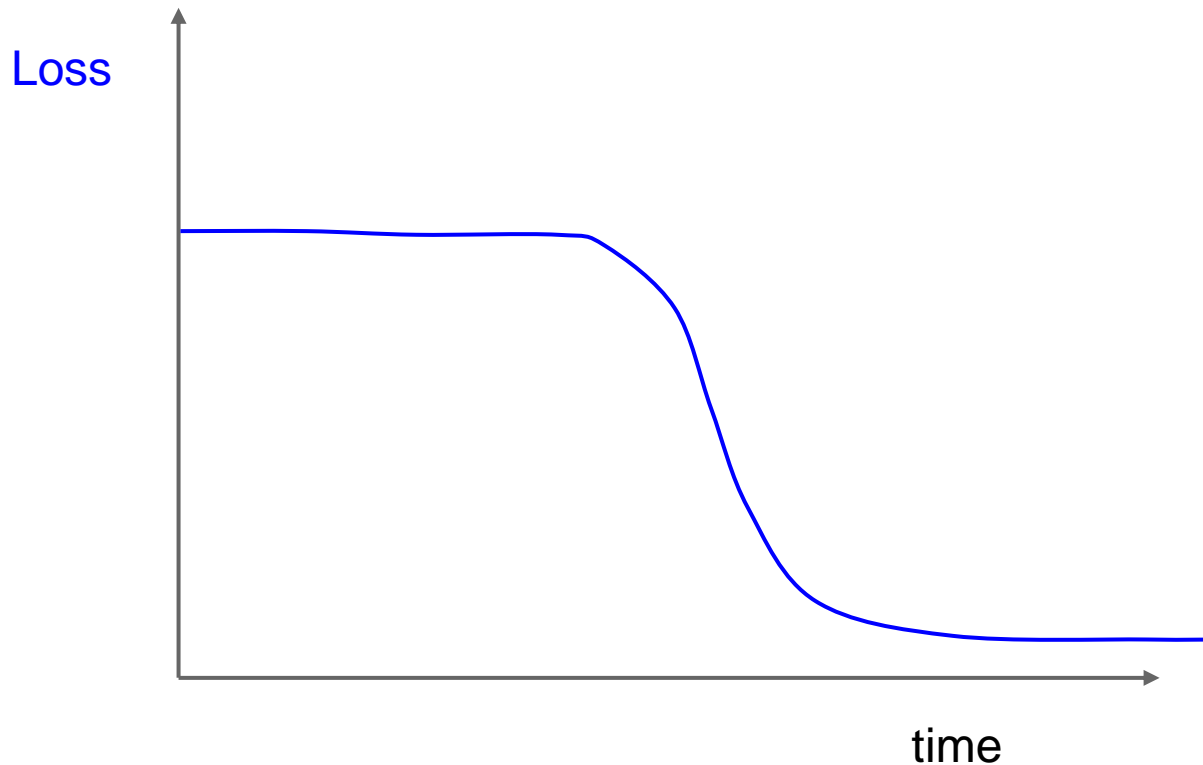
Visualize VISUALIZE VISUALIZE



What do you think is wrong here?

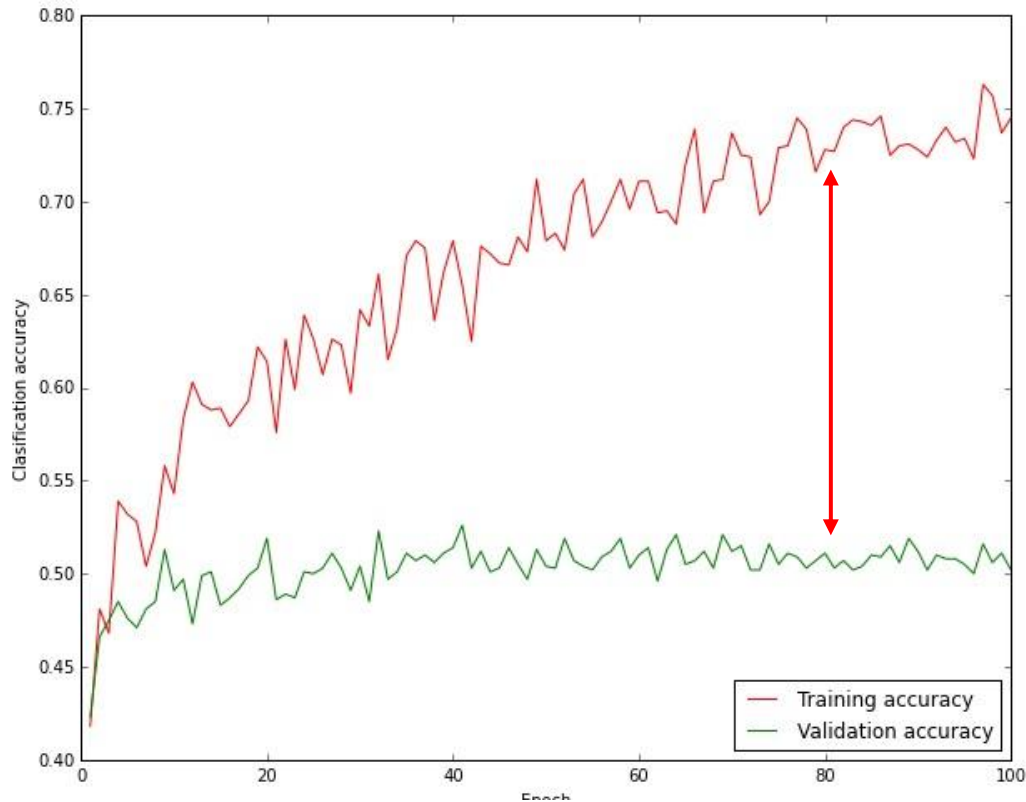


What do you think is wrong here?



The network is not being trained until it reaches a point. Recall that this would be when we are at the “flat” region in our activated responses.

Monitor and visualize the metric



big gap = overfitting
=> drop out? augmentation?

no gap
=> increase model capacity?

Track the effective learning rate

```
# assume parameter vector W and its gradient vector dW
param_scale = np.linalg.norm(W.ravel())
update = -learning_rate*dW # simple SGD update
update_scale = np.linalg.norm(update.ravel())
W += update # the actual update
print update_scale / param_scale # want ~1e-3
```

ratio between the updates and values: $\sim 0.0002 / 0.02 = 0.01$ (about okay)
want this to be somewhere around 0.001 or so

Training Neural Networks

- Check your input data
- Find a reasonable range for your hyperparameters
 - Try log-scale
- Launch your hyperparameter search
- VISUALIZE EVERYTHING