greatlearning
Learning for Life

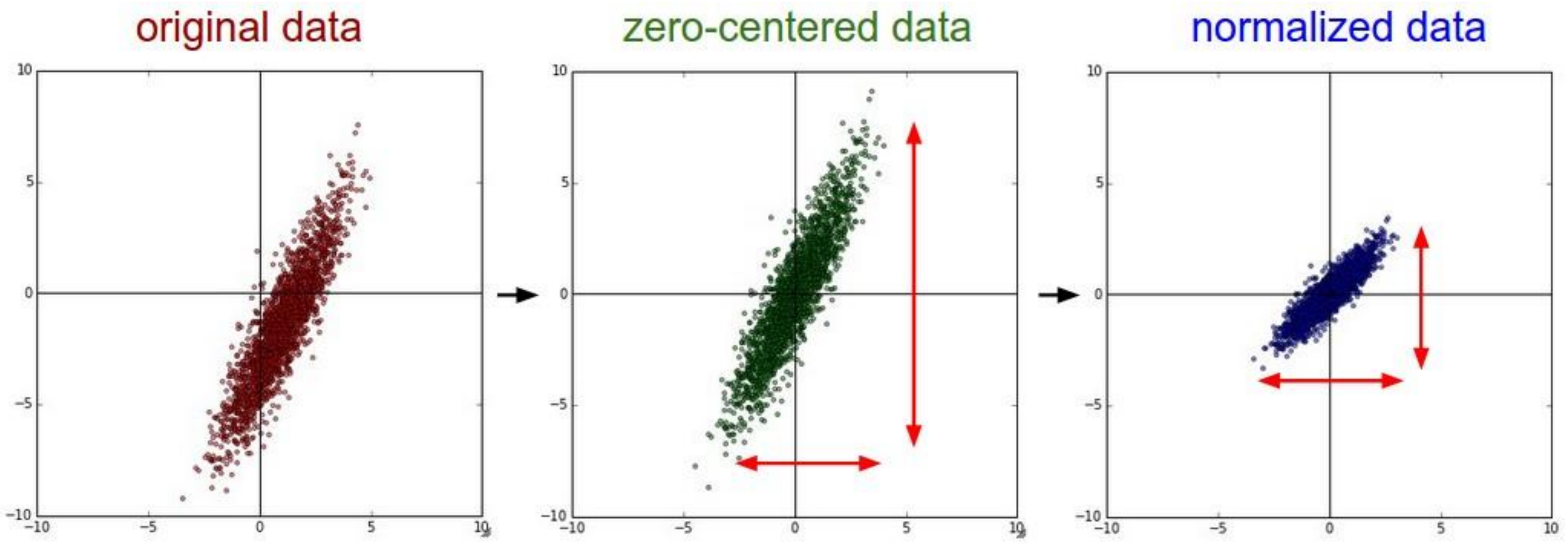
Deep Learning (for Computer Vision)

Arjun Jain

Babysitting the Learning Process

Step 1: Preprocess the data

Arjun: this will remain same?



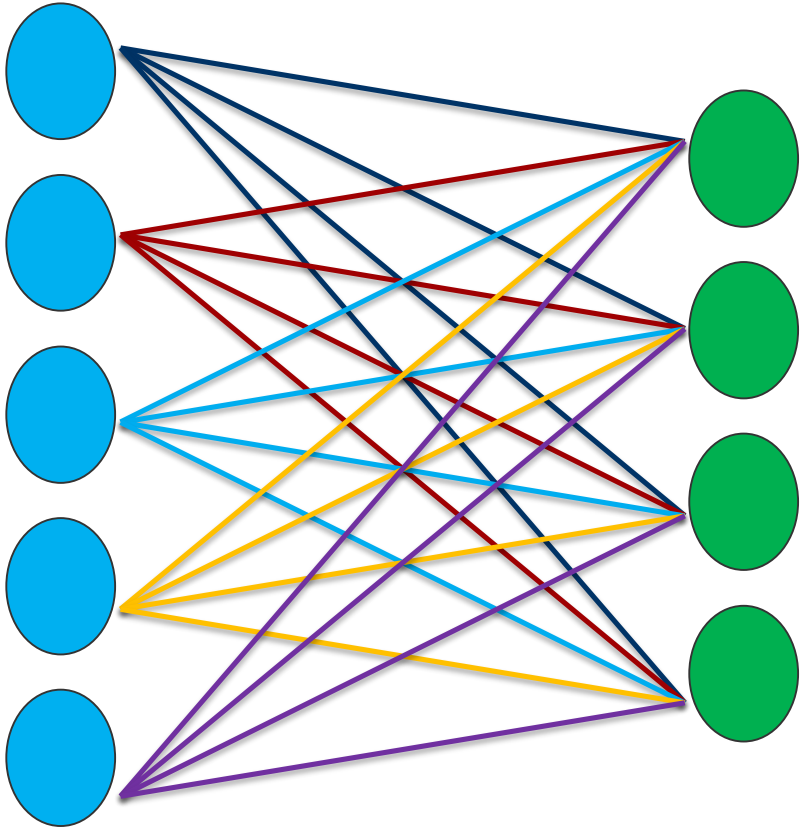
Mean subtract

Divide by standard deviation

(Assume X [NxD] is data matrix, each example in a row)

Step 2: Choose the architecture:

Say we start with single layer network:



output layer

10 output neurons, one per class

input layer

MNIST images, **784** numbers

1. **Data Loading: Let us load the training and the test data and check the size of the tensors. Let us also display the first few images from the training set.**

```
from keras.datasets import mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

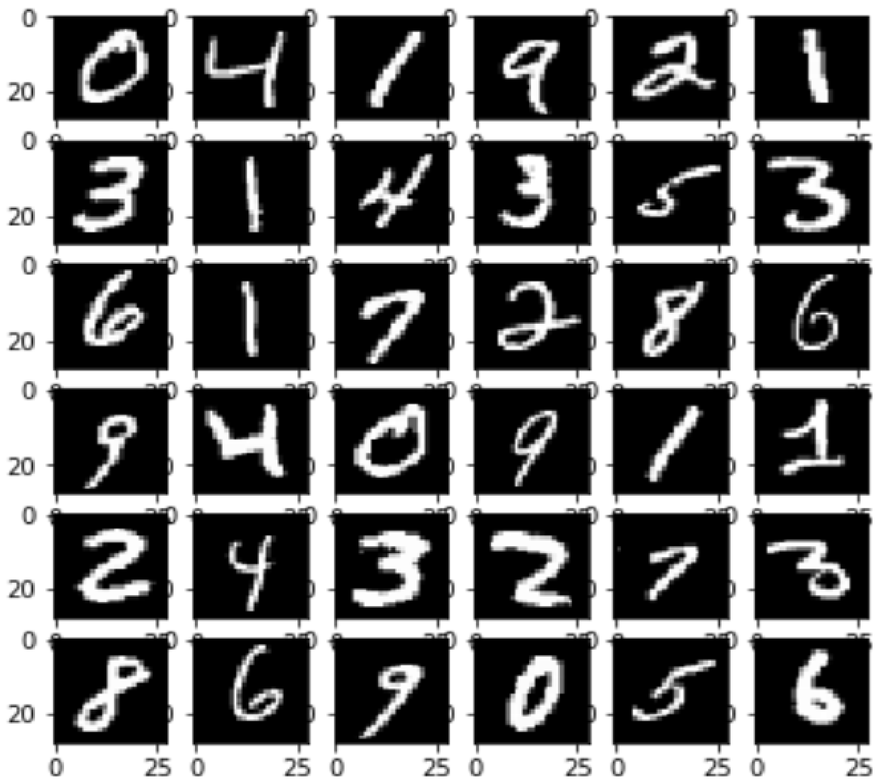
```
print(x_train.shape)
```

```
(60000, 28, 28)
```

```
print(y_train.shape)
```

```
(60000,)
```

```
import matplotlib.pyplot as plt
fig=plt.figure(figsize=(6, 6))
columns = 6
rows = 6
for i in range(1, columns*rows +1):
    img = x_train[i]
    fig.add_subplot(rows, columns, i)
    plt.imshow(img, cmap='gray')
plt.show()
```



```
def train_and_test_loop(no_iterations, lr, Lambda):


        graph = tf.Graph()
        with graph.as_default():

            # Input data.
            tf_train_dataset = tf.constant(train_dataset[:train_subset, :])
            tf_train_labels = tf.constant(train_labels[:train_subset])
            tf_test_dataset = tf.constant(test_dataset)

            tf_train_dataset = tf.cast(tf_train_dataset,dtype=tf.float32)
            tf_test_dataset = tf.cast(tf_test_dataset,dtype=tf.float32)
            tf_train_labels = tf.cast(tf_train_labels,dtype=tf.float32)

            # Variables
            # They are variables we want to update and optimize.
            weights = tf.Variable(tf.truncated_normal([image_size * image_size, num_labels]))
            biases = tf.Variable(tf.zeros([num_labels]))

            # Training computation.
            logits = tf.matmul(tf_train_dataset, weights) + biases
            # Original loss function
            loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits= logits, labels=tf_train_labels) )
            # Loss function using L2 Regularization
            regularizer = tf.nn.l2_loss(weights)
            loss = tf.reduce_mean(loss + Lambda * regularizer)

            # Optimizer.
            optimizer = tf.train.GradientDescentOptimizer(lr).minimize(loss)

            # Predictions for the training and test data.
            train_prediction = tf.nn.softmax(logits)
            test_prediction = tf.nn.softmax(tf.matmul(tf_test_dataset, weights) + biases)

        with tf.Session(graph=graph) as session:
            tf.initialize_all_variables().run()
            print('Initialized')
            for step in range(num_steps):
                _, l, predictions = session.run([optimizer, loss, train_prediction])
                if (step % 100 == 0):
                    print('Loss at step {}: {}'.format(step, l))
                    print('Training accuracy: {:.1f}'.format(accuracy(predictions,train_labels[:train_subset, :])))
            print('Test accuracy: {:.1f}'.format(accuracy(test_prediction.eval(), test_labels)))
```

Double check that the loss is reasonable:

```
# Training computation.
logits = tf.matmul(tf_train_dataset, weights) + biases
# Original loss function
loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits= logits, labels=tf_train_labels) )
# Loss function using L2 Regularization
regularizer = tf.nn.l2_loss(weights)
loss = tf.reduce_mean(loss + Lambda * regularizer)
```

```
## run it
lr = 0.00001
Lambda = 0.0
train_and_test_loop(1,lr,Lambda)
```

disable regularization

```
Initialized
Loss at step 0: 3822.80810547
Training accuracy: 8.0
Test accuracy: 7.6
```

Print Loss

Double check that the loss is reasonable:

```
# Training computation.
logits = tf.matmul(tf_train_dataset, weights) + biases
# Original loss function
loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits= logits, labels=tf_train_labels) )
# Loss function using L2 Regularization
regularizer = tf.nn.l2_loss(weights)
loss = tf.reduce_mean(loss + Lambda * regularizer)
```

```
## run it
lr = 0.00001
Lambda = 1e3
train_and_test_loop(1,lr,Lambda)
```

```
Initialized
Loss at step 0: 3058726.25
Training accuracy: 9.4
Test accuracy: 8.8
```

Crank it way up regularization

loss went up, good. (sanity check)

Lets try to train now…

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

```
## run it
lr = 0.0001
Lambda = 0
train_and_test_loop(100000,lr,Lambda)
```

The above code:

- take the first 20 examples from MNIST
- turn off regularization (reg = 0.0)
- use simple vanilla ‘sgd’

```
train_subset = 20
tf_train_dataset = tf.constant(train_dataset[:train_subset, :])
tf_train_labels = tf.constant(train_labels[:train_subset])
print(tf_train_dataset.shape)
print(tf_train_labels.shape)
```

```
(20, 784)
(20, 10)
```

Lets try to train now…

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

Very small loss, train accuracy 100,
nice!

```
In [53]: ## run it
         lr = 0.0001
         Lambda = 0
         train_and_test_loop(100000,lr,Lambda)
```

```
Initialized
Loss at step 0: 2720.238525390625
Training accuracy: 4.5
Loss at step 500: 538.8995971679688
Training accuracy: 53.8
Loss at step 1000: 325.397524902344
Training accuracy: 69.3
Loss at step 1500: 250.513626098328
Training accuracy: 74.8
Loss at step 2000: 207.04647827148438
Training accuracy: 77.8
Loss at step 2500: 176.4746551513672
Loss at step 97500: 6.917799328221008e-05
Training accuracy: 100.0
Loss at step 98000: 6.884850154165179e-05
Training accuracy: 100.0
Loss at step 98500: 6.852139631519094e-05
Training accuracy: 100.0
Loss at step 99000: 6.819446571171284e-05
Training accuracy: 100.0
Loss at step 99500: 6.7878958361689e-05
Training accuracy: 100.0
Test accuracy: 77.5
```

Lets try to train now…

```
## run it
lr = 0.0001
Lambda = 0
train_and_test_loop(100000,lr,Lambda)
```

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

Lets try to train now...

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

```
In [*]:  ## run it
         lr = 1e-7
         Lambda = 1e-7
         train_and_test_loop(10000,lr,Lambda)
```

```
Initialized
Loss at step 0: 2947.041015625
Training accuracy: 10.8
Loss at step 500: 2931.934814453125
Training accuracy: 10.8
Loss at step 1000: 2917.03930664062 5
Training accuracy: 10.8
Loss at step 1500: 2902.32470703125
Training accuracy: 10.8
Loss at step 2000: 2887.82495117187 5
Training accuracy: 10.8
Loss at step 2500: 2873.548828125
Training accuracy: 10.7
Loss at step 3000: 2859.4697265625
Training accuracy: 10.7
Loss at step 3500: 2845.53393554687 5
Training accuracy: 10.7
Loss at step 4000: 2831.77270507812 5
Training accuracy: 10.8
```

Loss barely changing: Learning rate is probably too low

Lets try to train now…

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

loss not going down:
learning rate too low

```
In [*]: ## run it
lr = 1e-7
Lambda = 1e-7
train_and_test_loop(10000,lr,Lambda)
```

```
Initialized
Loss at step 0: 2947.041015625
Training accuracy: 10.8
Loss at step 500: 2931.934814453125
Training accuracy: 10.8
Loss at step 1000: 2917.03930640625
Training accuracy: 10.8
Loss at step 1500: 2902.32470703125
Training accuracy: 10.8
Loss at step 2000: 2887.824951171875
Training accuracy: 10.8
Loss at step 2500: 2873.548828125
Training accuracy: 10.7
Loss at step 3000: 2859.4697265625
Training accuracy: 10.7
Loss at step 3500: 2845.533935546875
Training accuracy: 10.7
Loss at step 4000: 2831.772705078125
Training accuracy: 10.8
```

Loss barely changing: Learning rate is probably too low

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

```
In [*]: ## run it
        lr = 1e-7
        Lambda = 1e-7
        train_and_test_loop(10000,lr,Lambda)
```

```
Initialized
Loss at step 0: 2947.041015625
Training accuracy: 10.8
Loss at step 500: 2931.934814453125
Training accuracy: 10.8
Loss at step 1000: 2917.039306640625
Training accuracy: 10.8
Loss at step 1500: 2902.32470703125
Training accuracy: 10.8
Loss at step 2000: 2887.824951171875
Training accuracy: 10.8
Loss at step 2500: 2873.548828125
Training accuracy: 10.7
Loss at step 3000: 2859.4697265625
Training accuracy: 10.7
Loss at step 3500: 2845.533935546875
Training accuracy: 10.7
Loss at step 4000: 2831.772705078125
Training accuracy: 10.8
```

Notice train/val accuracy goes to 10.8% though, what's up with that? (remember this is softmax)

Loss barely changing: Learning rate is probably too low

Lets try to train now…

```
-- run it
lr = 1e6
lambda = 1e-7
train_and_test_loop(10000, lr, lambda)
```

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

loss not going down:
learning rate too low

Okay now lets try learning rate 1e6. What could possibly go wrong?

Lets try to train now...

I like to 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

In [*]:

```
## run it
lr = 1e6
Lambda = 1e-7
train_and_test_loop(10000,lr,Lambda)
```

```
Initialized
Loss at step 0: 2789.831298828125
Training accuracy: 11.9
Loss at step 500: 175791833088.0
Training accuracy: 50.2
Loss at step 1000: 147257933824.0
Training accuracy: 39.8
Loss at step 1500: 102461349888.0
Training accuracy: 58.3
Loss at step 2000: 141665763328.0
Training accuracy: 47.0
Loss at step 2500: 149215477760.0
Training accuracy: 45.6
Loss at step 3000: 169182396416.0
Training accuracy: 43.2
Loss at step 3500: 161494515712.0
Training accuracy: 53.4
Loss at step 4000: 141815939072.0
Training accuracy: 61.4
Loss at step 4500: 125380009984.0
```

cost: Very high

always means high learning rate...

Lets try to train now…

I like to 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

```
## run it
lr = 1e-3
Lambda = 1e-7
train_and_test_loop(3000,lr,Lambda)
```

```
Out[29]: iter: 0, accuracy: 20%  Loss: 0.02357119788693
          -- best accuracy achieved: 20%

Out[29]: iter: 500, accuracy: 13%  Loss: nan

Out[29]: iter: 1000, accuracy: 13%  Loss: nan

Out[29]: iter: 1500, accuracy: 13%  Loss: nan

Out[29]: iter: 2000, accuracy: 13%  Loss: nan

Out[29]: iter: 2500, accuracy: 13%  Loss: nan

Out[29]: iter: 3000, accuracy: 13%  Loss: nan
```

1e-3 is still too high. Cost explodes….

=> Rough range for learning rate we should be cross-validating is somewhere [1e-3 … 1e-7]

Hyperparameter Optimization

Cross-validation Strategy

Do **coarse -> fine** cross-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 * original cost, break out early

For example: run coarse search for 2000 iterations

In [*]:
```
import math
for i in range(1,100):
    lr = math.pow(10, np.random.uniform(-7.0,-3.0))
    Lambda = math.pow(10, np.random.uniform(-5,5))
    best_acc = train_and_test_loop(2000, lr, Lambda)
    print("Try {}/{} Best val accuracy: {}, lr: {}, Lambda: {}\n".format(i, 100, best_acc, lr, Lambda))
```

```
Try 1/100 Best val accuracy: 66.27, lr: 5.531150836907919e-05, Lambda: 0.0006478249956731675

Try 2/100 Best val accuracy: 12.73, lr: 4.251692668247112e-07, Lambda: 0.0001841192310560464

Try 3/100 Best val accuracy: 15.51, lr: 0.0007719701966206582, Lambda: 1131.7733448763438

Try 4/100 Best val accuracy: 33.43, lr: 1.0601119140629175e-05, Lambda: 0.00020118004362275232

Try 5/100 Best val accuracy: 78.65, lr: 0.00021657871041910485, Lambda: 5.513146508193506

Try 6/100 Best val accuracy: 15.11, lr: 4.010246952402139e-06, Lambda: 7.223431540581708e-05

Try 7/100 Best val accuracy: 12.92, lr: 1.5941265490509437e-06, Lambda: 0.07303345671033763

Try 8/100 Best val accuracy: 9.82, lr: 4.275465039058684e-05, Lambda: 30996.335786164895

Try 9/100 Best val accuracy: 8.62, lr: 1.3149674932203763e-07, Lambda: 0.0006327196882522297

Try 10/100 Best val accuracy: 10.66, lr: 5.034624026089686e-07, Lambda: 1.0511011782956448

Try 11/100 Best val accuracy: 9.8, lr: 0.00011217505415998534, Lambda: 45154.64994211267

Try 12/100 Best val accuracy: 9.8, lr: 0.0006070229598245868, Lambda: 7165.444545998027

Try 13/100 Best val accuracy: 50.55, lr: 2.0834833520853556e-05, Lambda: 7.277345954108924

Try 14/100 Best val accuracy: 9.74, lr: 0.0001253131724867973, Lambda: 13557.016063816893

Try 15/100 Best val accuracy: 10.48, lr: 3.169391909114394e-07, Lambda: 0.019504801963701995
```

note it's best to optimize in log space!

nice

Now run finer search...

```
import math
for i in range(1,100):
    lr = math.pow(10, np.random.uniform(-7.0,-3.0))
    Lambda = math.pow(10, np.random.uniform(-5,5))
    best_acc = train_and_test_loop(2000, lr, Lambda)
    print("Try {}/{} Best val accuracy: {}, lr: {}, Lambda: {}\n".format(i, 100, best_acc, lr, Lambda))
```

```
import math
for i in range(1,100):
    lr = math.pow(10, np.random.uniform(-6.0,-4.0))
    Lambda = math.pow(10, np.random.uniform(-3,1))
    best_acc = train_and_test_loop(2000, lr, Lambda)
    print("Try {}/{} Best val accuracy: {}, lr: {}, Lambda: {}\n".format(i, 100, best_acc, lr, Lambda))
```

```
Try 1/100 Best val accuracy: 19.46, lr: 5.417270002123785e-06, Lambda: 3.451835448987154

Try 2/100 Best val accuracy: 25.99, lr: 6.501495775369341e-06, Lambda: 0.002069915669820317

Try 3/100 Best val accuracy: 28.83, lr: 9.733200731691926e-06, Lambda: 0.0010868181177409722

Try 4/100 Best val accuracy: 46.14, lr: 1.9776169441074813e-05, Lambda: 3.270957369966795

Try 5/100 Best val accuracy: 21.39, lr: 5.961062639977423e-06, Lambda: 0.1529955422819221

Try 6/100 Best val accuracy: 18.88, lr: 6.628154271108286e-06, Lambda: 0.012851871729300968

Try 7/100 Best val accuracy: 71.17, lr: 9.336138820699605e-05, Lambda: 2.162547220278807

Try 8/100 Best val accuracy: 54.48, lr: 2.8884720344700566e-05, Lambda: 0.09526284705480523

Try 9/100 Best val accuracy: 56.18, lr: 3.0369445932033802e-05, Lambda: 2.80437535834822

Try 10/100 Best val accuracy: 17.49, lr: 3.4704138264445587e-06, Lambda: 0.0020283269549467999

Try 11/100 Best val accuracy: 13.01, lr: 1.3029539640800816e-06, Lambda: 0.09323405990107717
```

adjust range

71% - relatively good for a 1-layer neural net and only 2000 iterations

Now run finer search...

```
import math
for i in range(1,100):
    lr = math.pow(10, np.random.uniform(-7.0,-3.0))
    Lambda = math.pow(10, np.random.uniform(-5,5))
    best_acc = train_and_test_loop(2000, lr, Lambda)
    print("Try {}/{} Best val accuracy: {}, lr: {}, Lambda: {}\n".format(i, 100, best_acc, lr, Lambda))
```

```
import math
for i in range(1,100):
    lr = math.pow(10, np.random.uniform(-6.0,-4.0))
    Lambda = math.pow(10, np.random.uniform(-3,1))
    best_acc = train_and_test_loop(2000, lr, Lambda)
    print("Try {}/{} Best val accuracy: {}, lr: {}, Lambda: {}\n".format(i, 100, best_acc, lr, Lambda))
```

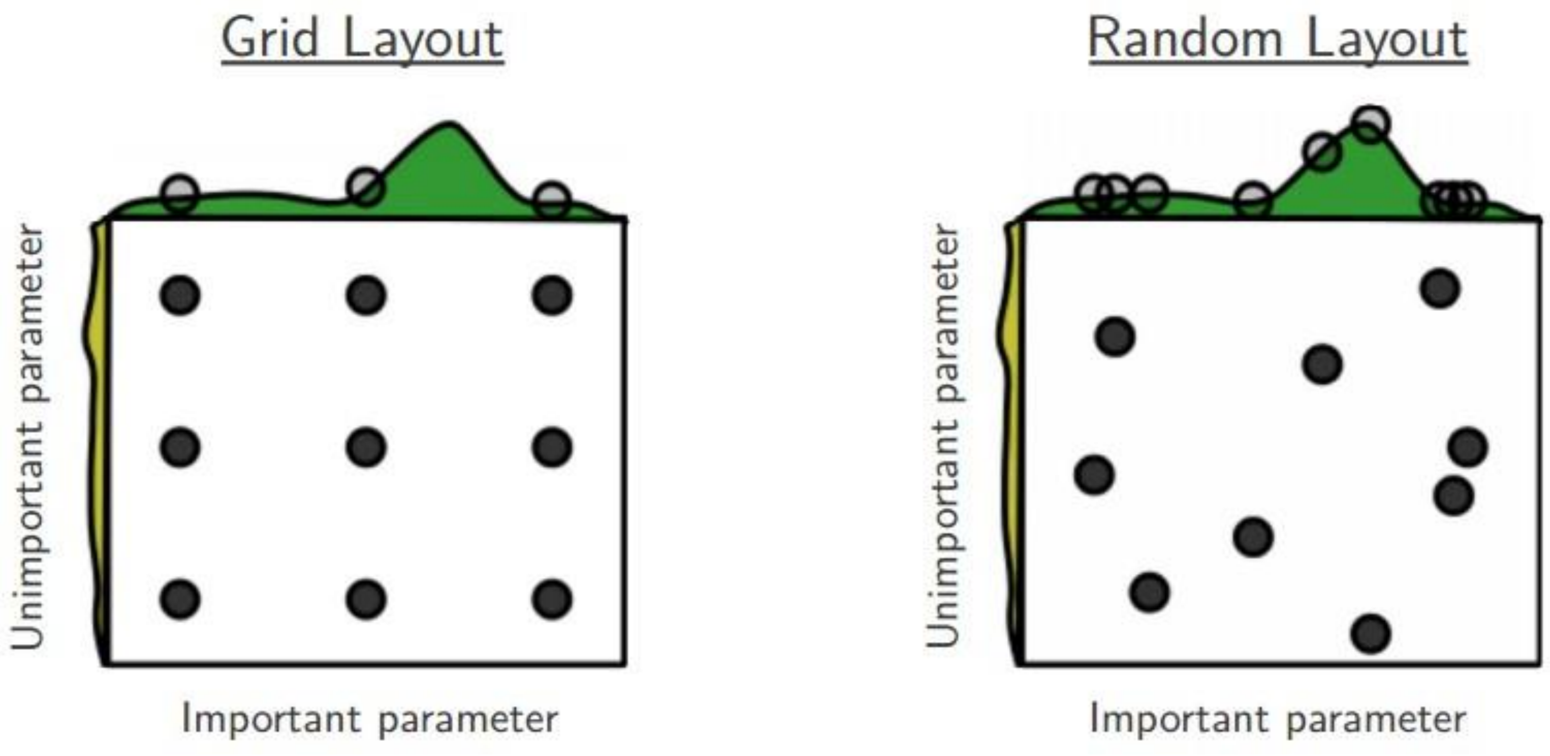
adjust range

```
Try 1/100 Best val accuracy: 19.46, lr: 5.417270002123785e-06, Lambda: 3.451835448987154

Try 2/100 Best val accuracy: 25.99, lr: 6.501495775369341e-06, Lambda: 0.002069915669820317

Try 3/100 Best val accuracy: 28.83, lr: 9.733200731691926e-06, Lambda: 0.0010868181177409722

Try 4/100 Best val accuracy: 46.14, lr: 1.9776169441074813e-05, Lambda: 3.270957369966795

Try 5/100 Best val accuracy: 21.39, lr: 5.961062639977423e-06, Lambda: 0.1529955422819221

Try 6/100 Best val accuracy: 18.88, lr: 6.628154271108286e-06, Lambda: 0.012851871729300968

Try 7/100 Best val accuracy: 71.17, lr: 9.336138820699605e-05, Lambda: 2.162547220278807

Try 8/100 Best val accuracy: 54.48, lr: 2.8884720344700566e-05, Lambda: 0.09526284705480523

Try 9/100 Best val accuracy: 56.18, lr: 3.0369445932033802e-05, Lambda: 2.80437535834822

Try 10/100 Best val accuracy: 17.49, lr: 3.4704138264445587e-06, Lambda: 0.002028326954946799

Try 11/100 Best val accuracy: 13.01, lr: 1.3029539640800816e-06, Lambda: 0.09323405990107717
```

71% - relatively good for a 1-layer neural net and only 2000 iterations

Make sure the best ones are not on the boundary

Random Search vs. Grid Search



Sourced from: Random Search for Hyper-Parameter Optimization by Bergstra and Bengio , et.al 2012

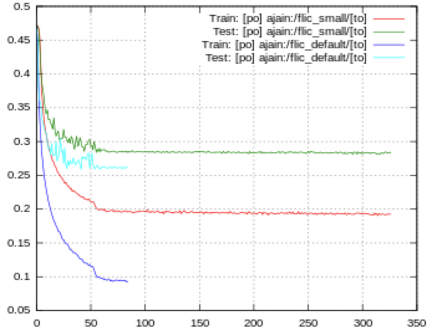
Hyperparameters to play with

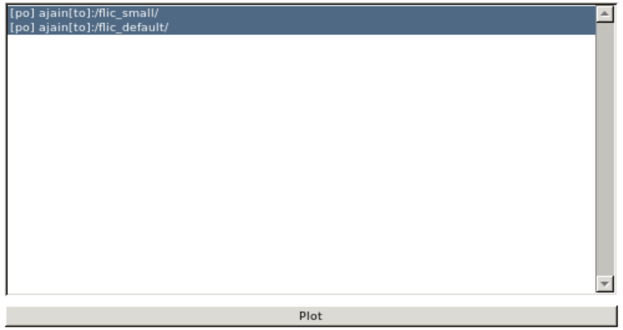
- network architecture
- learning rate, its multiplier schedule
- regularization (L2/Dropout strength)

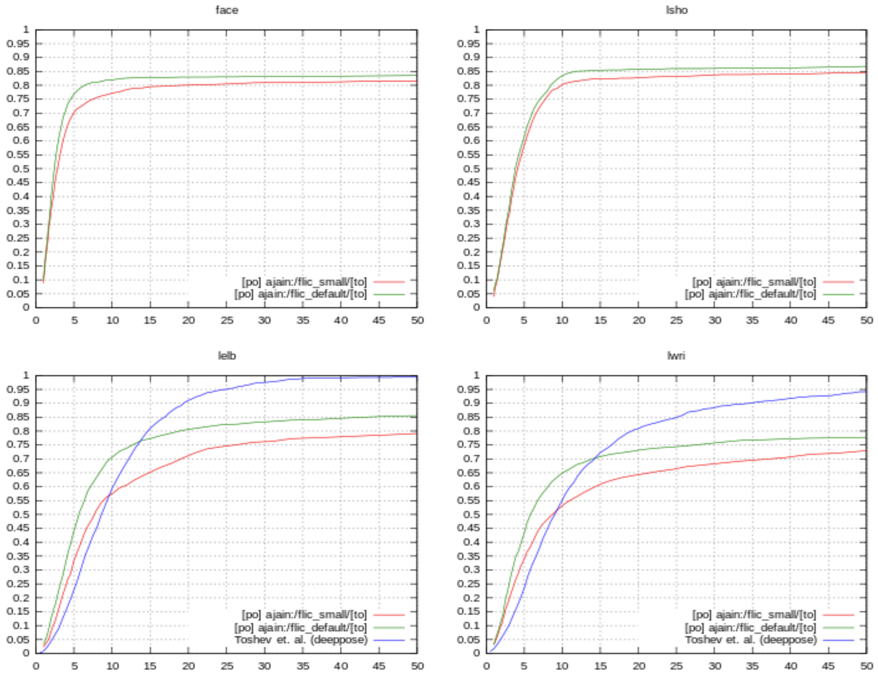
neural networks practitioner
music = loss function

Karpathy’s cross-validation “command center”

My cross-validation “command center”







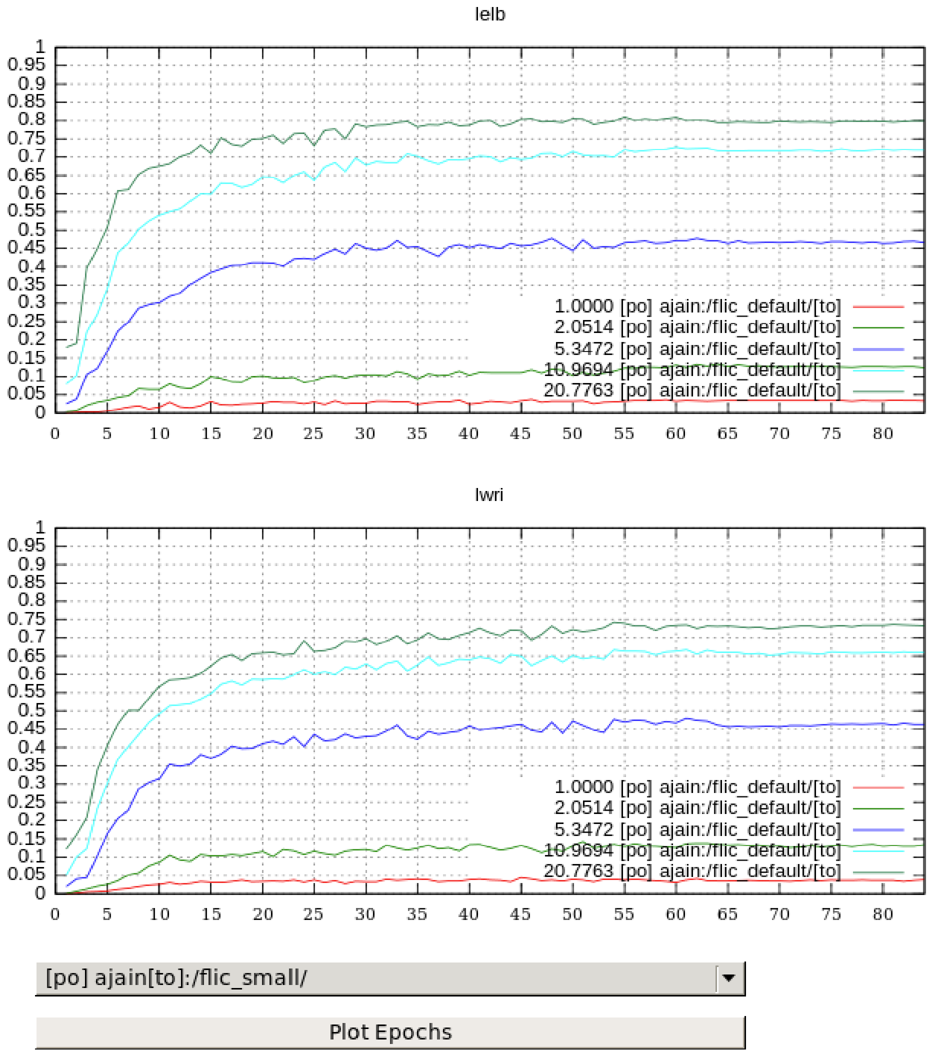
[po] ajain:/flic_small/[to]

```
2  batch_normalization = false
3  batch_size = 16
4  batch_size_per_gpu = 16
5  big_model = false
6  body_scale_range = {}
7  compress_src_model =
8  conv_1x1_size = 1
9  conv_nfeats = { 1 = { 1 = 16, 2 = 16, 3 = 16, }, 2 = { 1 = 16, 2 = 16, 3 = 16, }, 3 = { 1 = 16, 2 = 16, 3 = 16, },}
10 conv_pool = { 1 = { 1 = 2, 2 = 2, 3 = 1, }, 2 = { 1 = 2, 2 = 2, 3 = 1, }, 3 = { 1 = 2, 2 = 2, 3 = 1, },}
11 conv_size = { 1 = { 1 = 5, 2 = 5, 3 = 5, }, 2 = { 1 = 5, 2 = 5, 3 = 5, }, 3 = { 1 = 5, 2 = 5, 3 = 5, },}
12 crop_gradOutput = false
```

[po] ajain:/flic_default/[to]

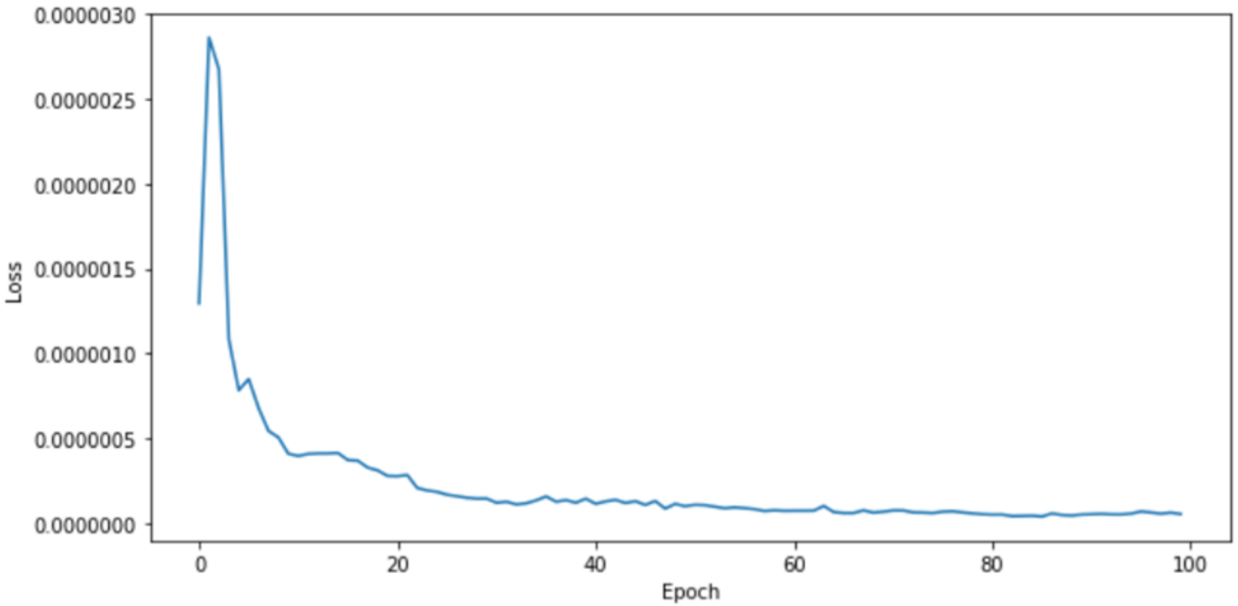
```
2  batch_normalization = false
3  batch_size = 16
4  batch_size_per_gpu = 16
5  big_model = true
6  body_scale_range = {}
7  compress_src_model =
8  conv_1x1_size = 1
9  conv_nfeats = { 1 = { 1 = 128, 2 = 128, 3 = 128, }, 2 = { 1 = 128, 2 = 128, 3 = 128, }, 3 = { 1 = 128, 2 = 128, 3 = 128, },}
10 conv_pool = { 1 = { 1 = 2, 2 = 2, 3 = 1, }, 2 = { 1 = 2, 2 = 2, 3 = 1, }, 3 = { 1 = 2, 2 = 2, 3 = 1, },}
11 conv_size = { 1 = { 1 = 5, 2 = 5, 3 = 5, }, 2 = { 1 = 5, 2 = 5, 3 = 5, }, 3 = { 1 = 5, 2 = 5, 3 = 5, },}
12 crop_gradOutput = false
```

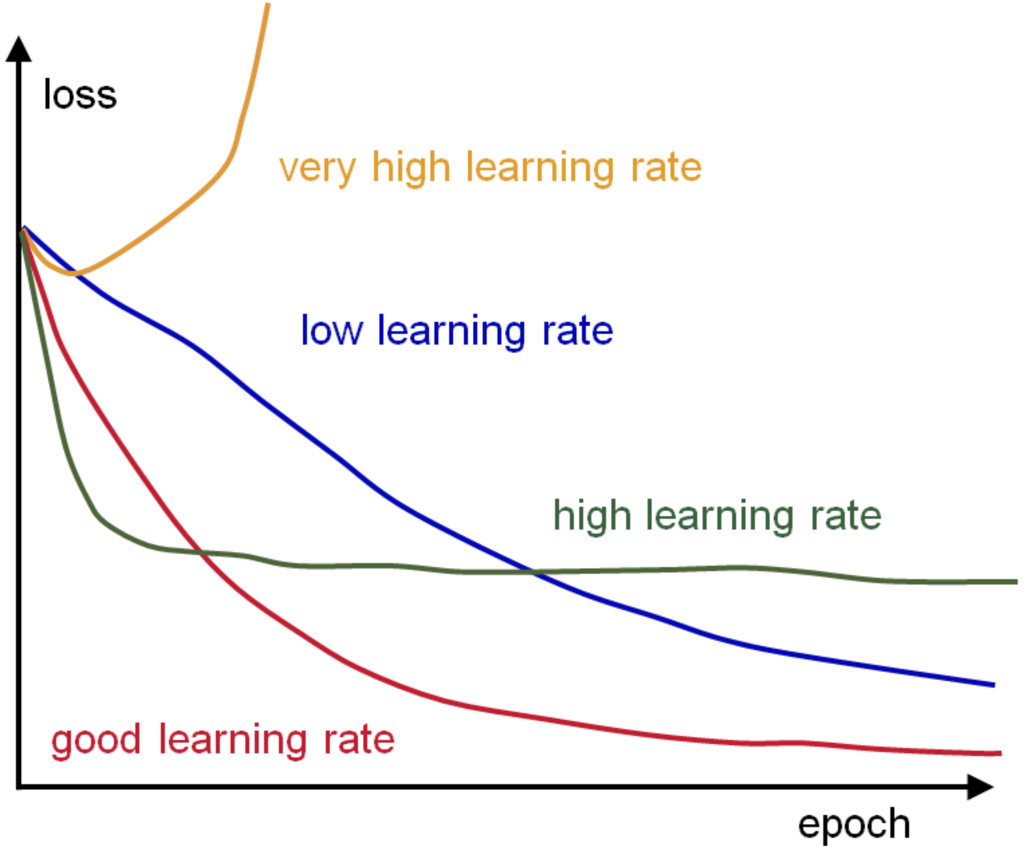
My cross-validation "command center"

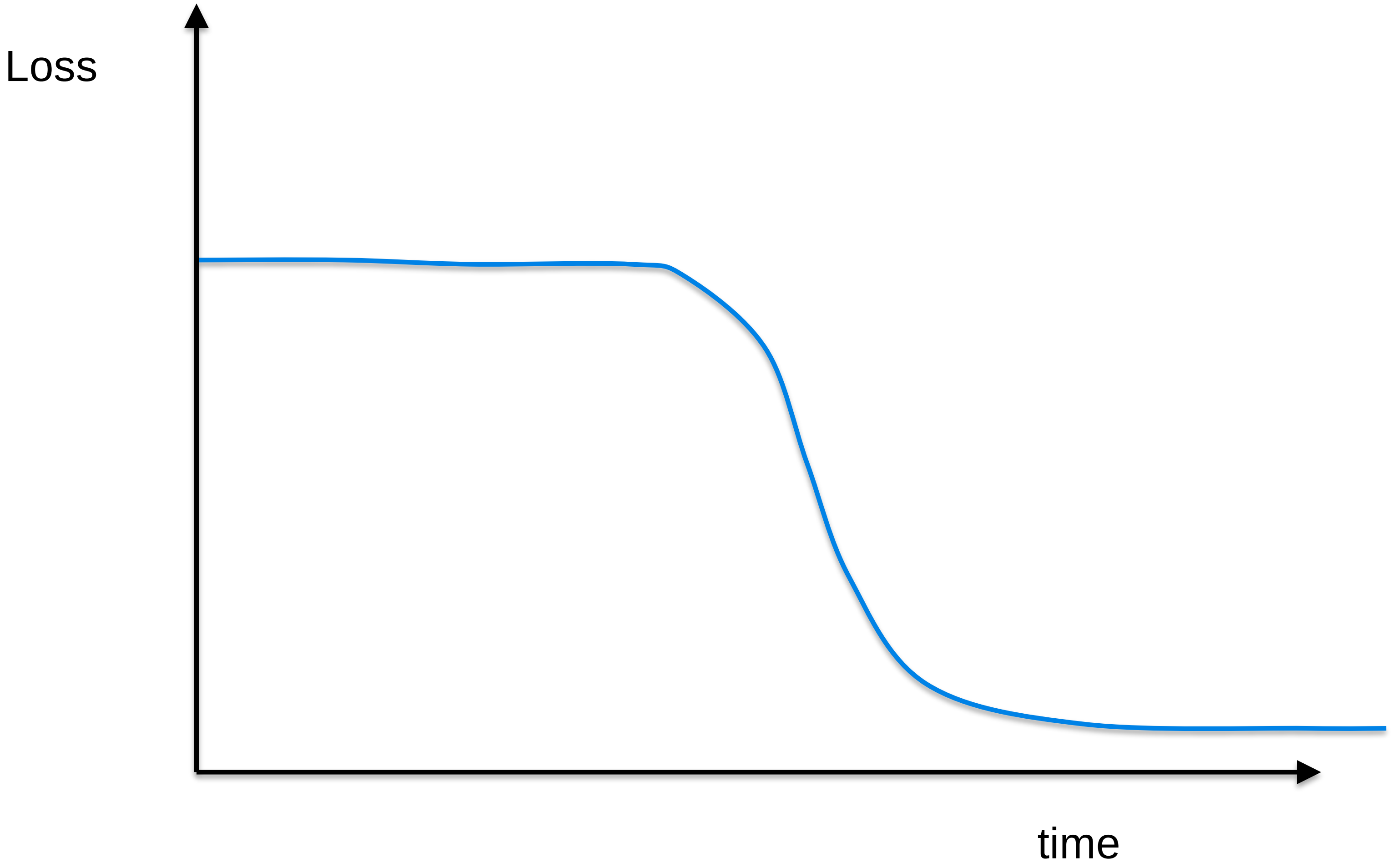


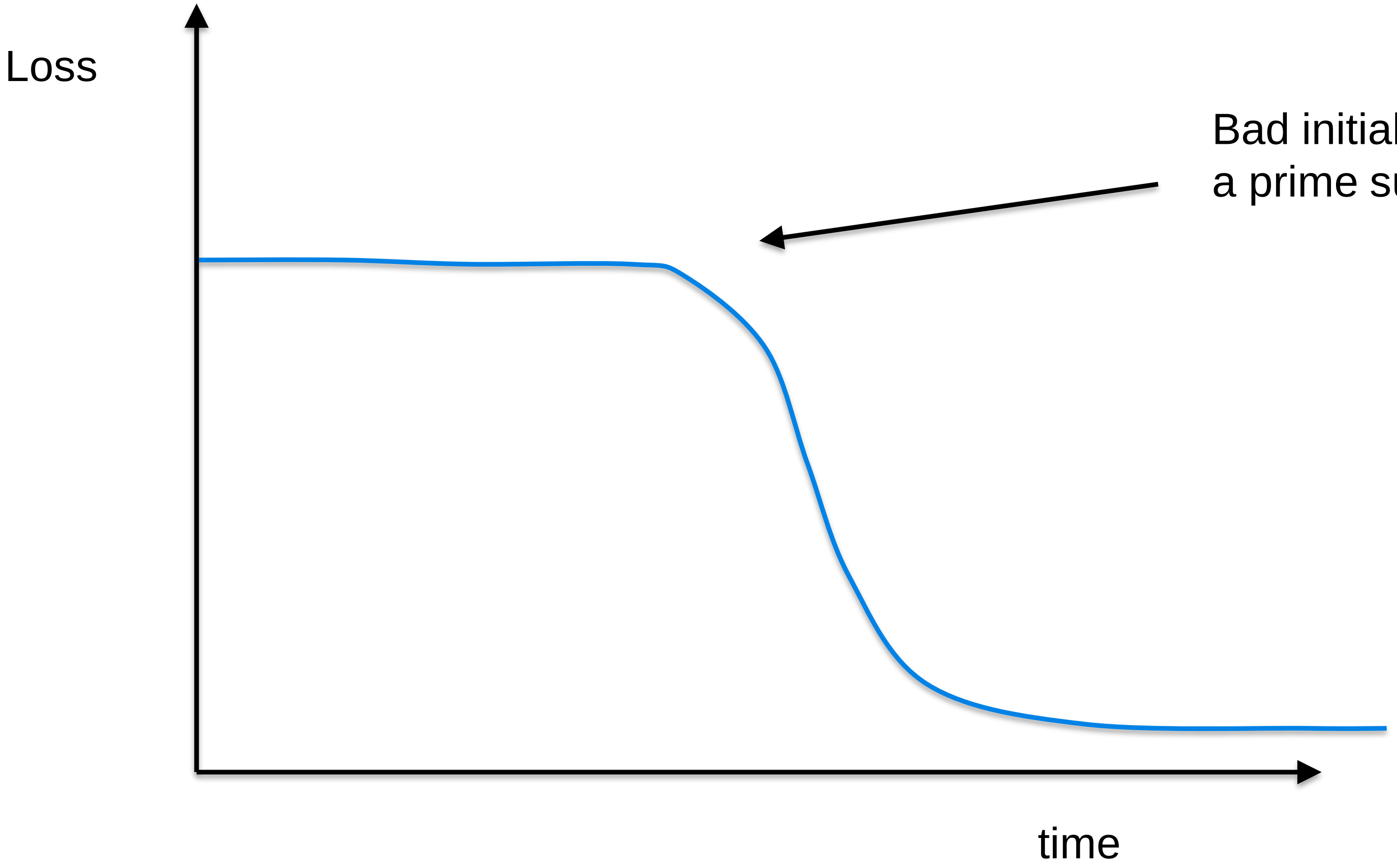
My cross-validation
“command center”

Monitor and visualize the loss curve

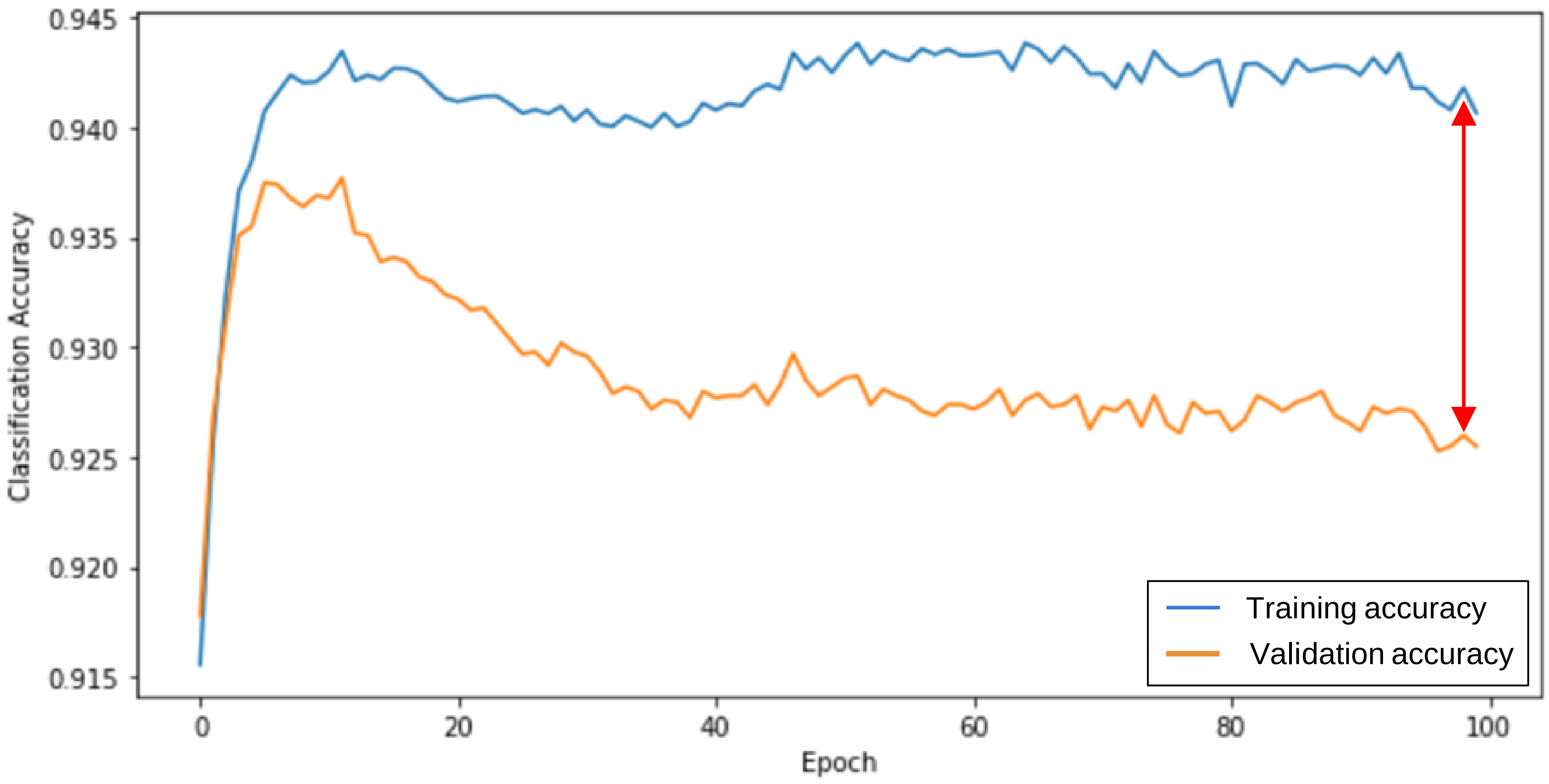





Loss
time


Loss
Bad initialization
a prime suspect
time

Monitor and visualize the accuracy:



big gap = overfitting
=> increase regularization strength?

no gap
=> increase model capacity?

Track the ratio of weight updates / weight magnitudes:

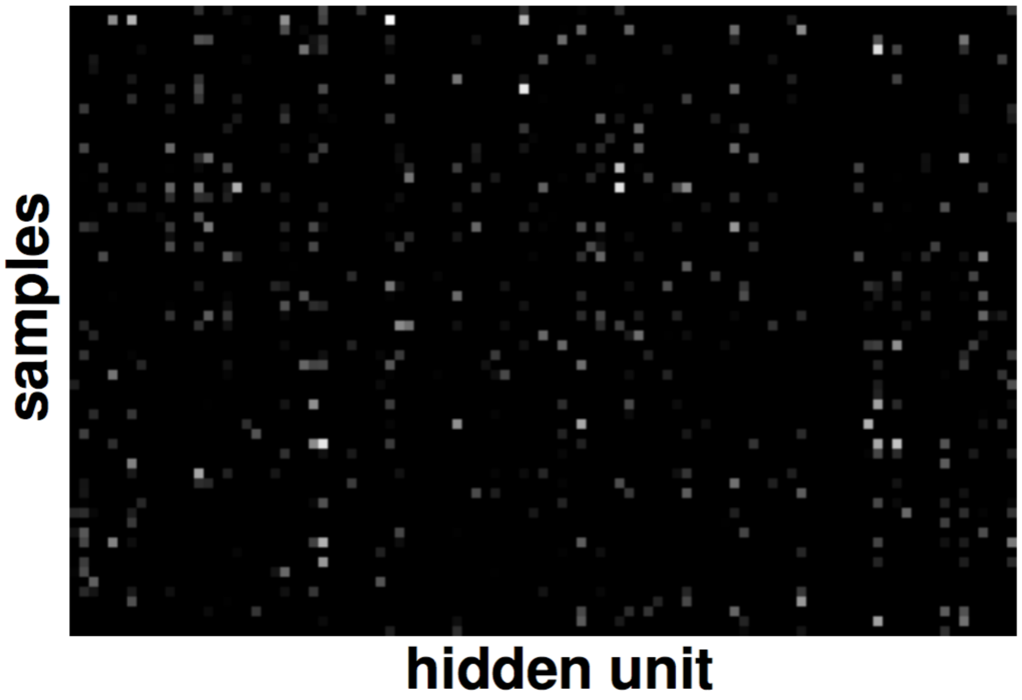
```
# weight vector W and its gradient vector dW
w_scale = np.linalg.norm(W.ravel())
update = -learning_rate*dW # simple SGD update
update_scale = np.linalg.norm(update.ravel())
W += update # the actual update
print update_scale / w_scale # want ~1e-3
```

ratio between the values and updates: ~ 0.0001 / 0.88 = 0.0001 (about okay)

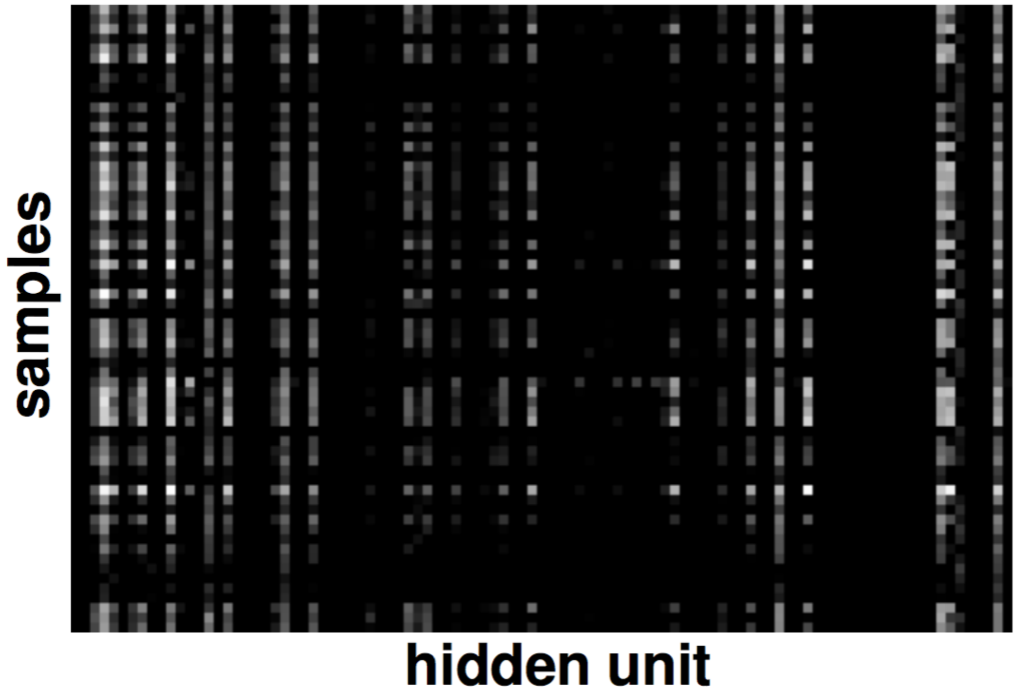
want this to be somewhere around 0.0001 or so

Visualize Activations

- Visualize features (feature maps need to be uncorrelated) and have high variance.



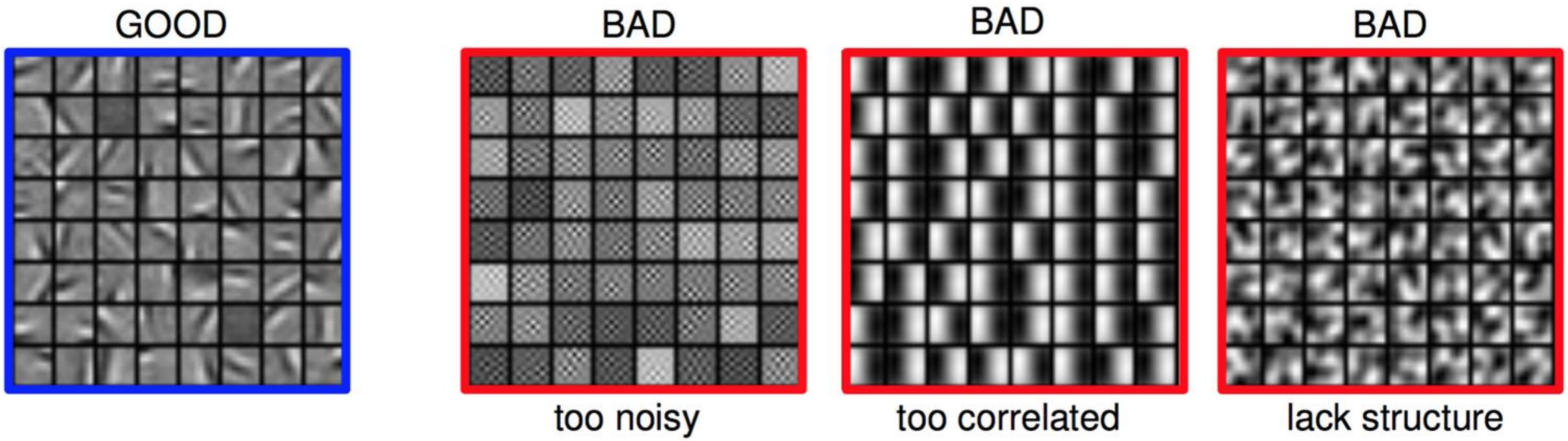
Good training: hidden units are sparse across samples and across features.



Bad training: many hidden units ignore the input and/or exhibit strong correlations.

Visualize (initial) Convolution Layer Weights

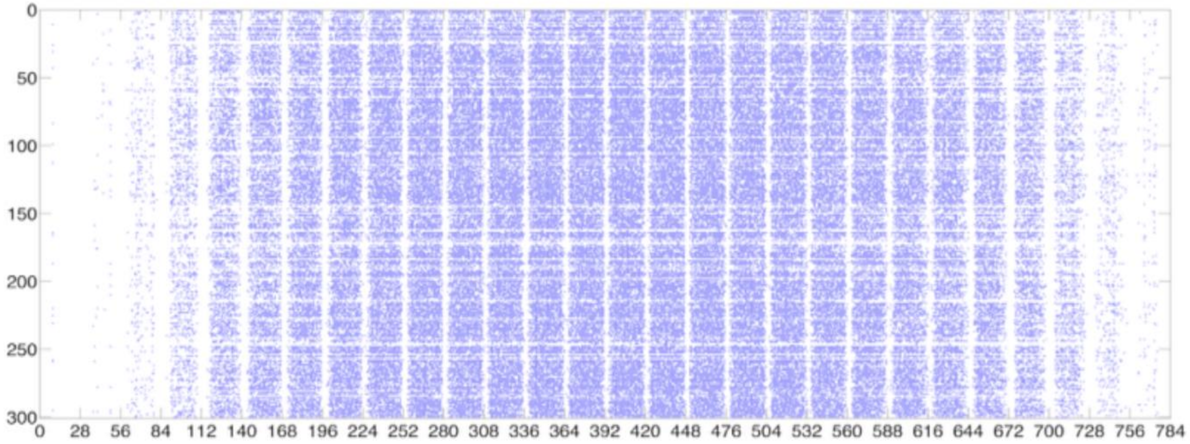
- Visualize features (feature maps need to be uncorrelated) and have high variance.



Good training: learned filters exhibit structure and are uncorrelated.

Visualize Linear(Fully Connected)Weights

- Visualization of Linear layer weights for some networks
- It has a banded structure repeated 28 times (Why?!) Hint: Images are 28x28
- Thus, looking at the weights we get some intuition



Thank you!