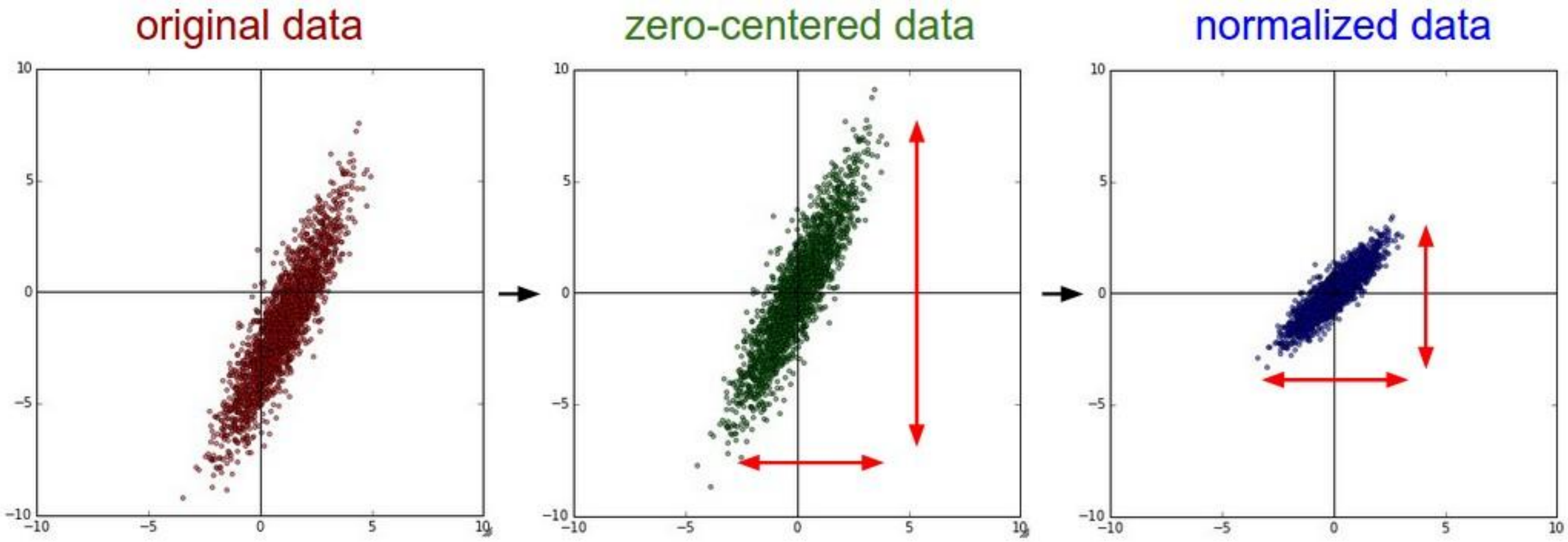
Babysitting the Learning Process

Step 1: Preprocess the data



Mean subtract

Divide by standard deviation

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

Notebook

https://github.com/stencilman/CS763_Spring2017/blob/master/Lec3%2C4/CrossEntropy-Linear.ipynb

2. Data Preprocessing: We compute the mean and standard deviation 'images' and then subtract and divide by the same respectively (like AlexNet). We also visualize them.

In [3]:

```
x_mean = torch.mean(tr_x:float(), 1)
x_std = torch.std(tr_x:float(), 1)
itorch.image(x_mean)
itorch.image(x_std)
```

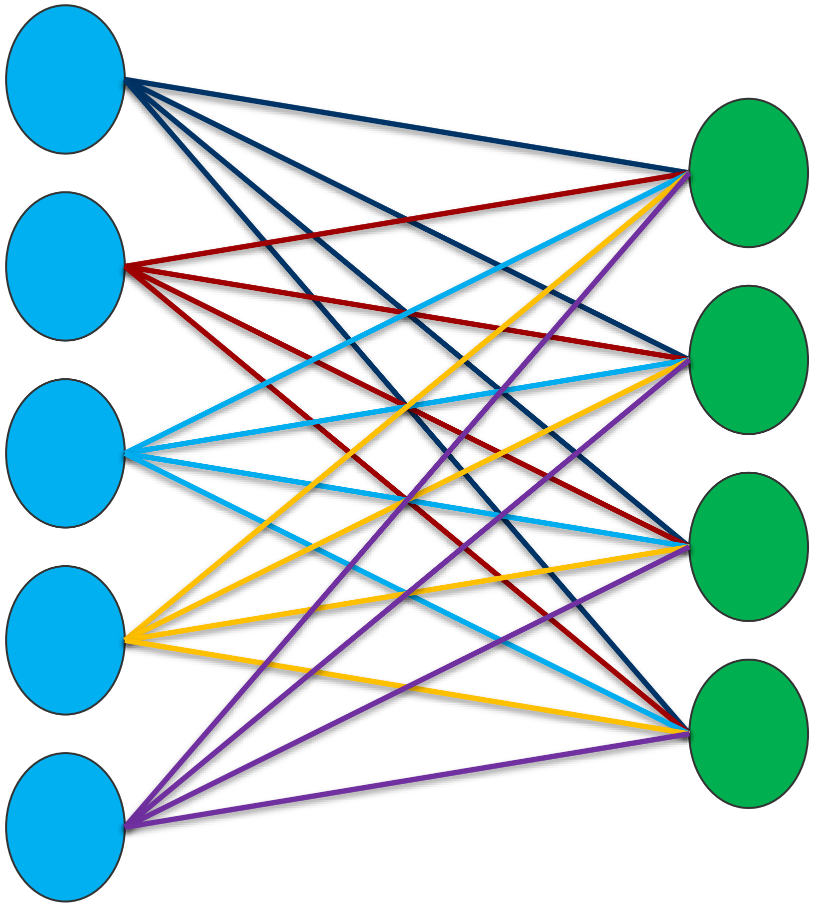
In [7]:

```
function get_xi(data_x, idx)
    xi = (data_x[idx]:float() - x_mean)
    xi = xi:cdiv(x_std)
    xi = xi:reshape(3*32*32)
    return xi
end
```

Step 2: Choose the architecture:

Say we start with single layer network:

input layer



output layer

10 output neurons, one per class

CIFAR-10 images, **3072** numbers

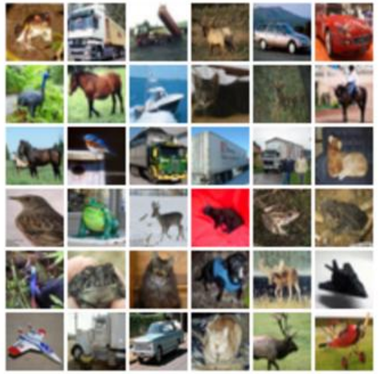
1. Data Loading: Let us load the training and the test data and check the size of the tensors. Let us also display the first few images from the training set.

In [1]:
```
-- load trainin images
tr_x = torch.load('cifar10/tr_data.bin')
-- load trainin labels
tr_y = torch.load('cifar10/tr_labels.bin'):double() + 1
-- load test images
te_x = torch.load('cifar10/te_data.bin')
-- load test labels
te_y = torch.load('cifar10/te_labels.bin'):double() + 1
print(tr_x:size())
print(tr_y:size())
```

Out[1]:
```
 50000
     3
    32
    32
[torch.LongStorage of size 4]

 50000
[torch.LongStorage of size 1]
```

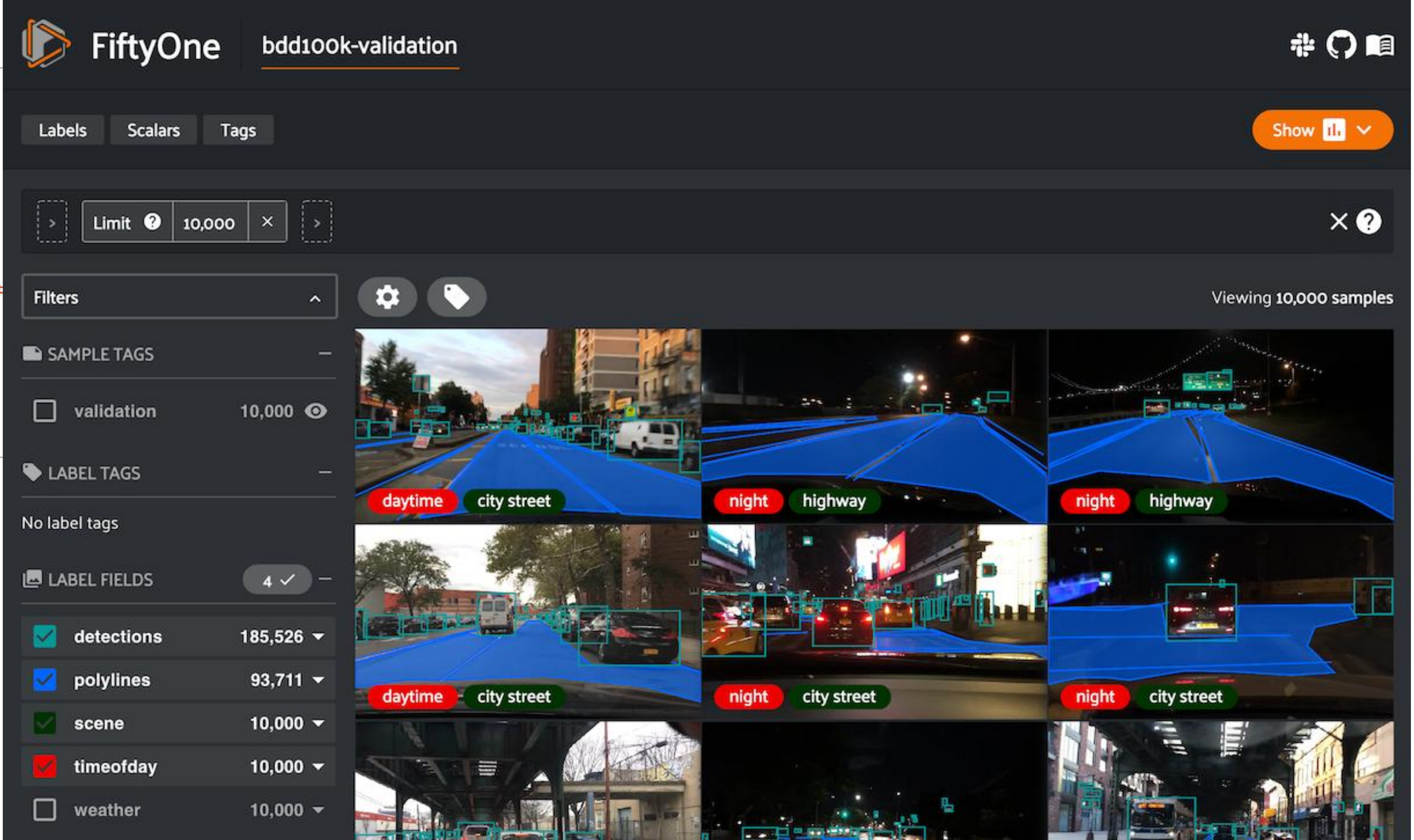
In [2]:
```
-- display the first 36 training set images
require 'image';
itorch.image(tr_x[{{1,36},{},{},{}}])
```



In

Out

In


FiftyOne
bdd100k-validation
Labels
Scalars
Tags
Show
Limit
10,000
Filters
Viewing 10,000 samples
SAMPLE TAGS
validation 10,000
LABEL TAGS
No label tags
LABEL FIELDS
4
detections 185,526
polylines 93,711
scene 10,000
timeofday 10,000
weather 10,000
daytime
city street
night
highway
night
highway
daytime
city street
night
city street
night
city street

```lua
function train_and_test_loop(no_iterations, lr, lambda)
    for i = 0, no_iterations do

        -- trainin input and target
        xi = get_xi(tr_x, i)
        ti = tr_y[i]

        -- Train
        op = model:forward(xi)
        loss_tr = criterion:forward(op, ti)
        dl_do = criterion:backward(op, ti)
        model:backward(xi, dl_do)

        -- Test
        idx = shuffle_te[mod(i, te_x:size(1)) + 1]
        xi = get_xi(te_x, idx)
        ti = te_y[idx]
        -- Compute loss
        op = model:forward(xi)
        loss_te = criterion:forward(op, ti, model, lambda)

        -- udapte model weights
        gradient_descent(model, lr)

        err = evaluate(model, tr_x, tr_y)
        if (err < besterr) then
            besterr = err
            bestmodel:copy(model)
        end

    end
    return (1 - besterr)*100 -- Accuracy
end
```

Double check that the loss is reasonable:

```
op = model:forward(xi)
loss_tr = criterion:forward(op, ti)
print(loss_tr)
```

```
-- run it
lr = 0.00001
lambda = 0.0
train_and_test_loop(1, lr, lambda)
```

disable regularization

```
Out[11]: 2.2656910718829
```

loss ~2.3.
“correct ” for
10 classes

Print Loss

Double check that the loss is reasonable:

```
op = model:forward(xi)
loss_tr = criterion:forward(op, ti,
print(loss_tr)
```

```
-- run it
lr = 0.00001
lambda = 1e3
train_and_test_loop(1, lr, lambda)
```

Crank it way up regularization

```
Out[12]: 12.582525307612
```

Print Loss

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 CIFAR-10
- turn off regularization (reg = 0.0)
- use simple vanilla 'sgd'

```
tr_x = tr_x[{{1,20},{},{},{}}]
te_x = tr_x[{{1,20},{},{},{}}]
tr_y = tr_y[{{1,20}}]
print(tr_x:size())
print(tr_y:size())
```

```
Out[14]:  20
           3
          32
          32
         [torch.LongStorage of size 4]

          20
         [torch.LongStorage of size 1]
```

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)
```

```
Out[54]: iter: 0, accuracy: 100%  Loss: 0.023342480719671
          -- best accuracy achieved: 20%

Out[54]: iter: 500, accuracy: 20%  Loss: 6.4891701533306
          -- best accuracy achieved: 100%

Out[54]: iter: 1000, accuracy: 100%  Loss: 3.363490690347

Out[54]: iter: 1500, accuracy: 100%  Loss: 2.399597567 7242

Out[54]: iter: 2000, accuracy: 100%  Loss: 1.8909617506362

Out[54]: iter: 2500, accuracy: 100%  Loss: 1.5617572159784

Out[54]: iter: 3000, accuracy: 100%  Loss: 1.3375534142717

Out[54]: iter: 3500, accuracy: 100%  Loss: 1.1668484200641

Out[54]: iter: 4000, accuracy: 100%  Loss: 1.0398030826978

Out[54]: iter: 98000, accuracy: 100%  Loss: 0.075056174474324

Out[54]: iter: 98500, accuracy: 100%  Loss: 0.074695131101785

Out[54]: iter: 99000, accuracy: 100%  Loss: 0.074675841566382

Out[54]: iter: 99500, accuracy: 100%  Loss: 0.074908872365756

Out[54]: iter: 100000, accuracy: 100%  Loss: 0.074439254969025
```

Very small loss, train accuracy 100, nice!

Lets try to train now…

I like to start with small regularization and find learning rate that makes the loss go down.

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

Lets try to train now…

I like to start with small regularization and find learning rate that makes the loss go down.

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

```
Out[18]: iter: 0, accuracy: 10%  Loss: 0.023248429529449
          -- best accuracy achieved: 10%

Out[18]: iter: 500, accuracy: 10%  Loss: 11.522416713458

Out[18]: iter: 1000, accuracy: 10%  Loss: 11.517536122735

Out[18]: iter: 1500, accuracy: 11%  Loss: 11.508510566527
          -- best accuracy achieved: 11%

Out[18]: iter: 2000, accuracy: 13%  Loss: 11.510842908524
          -- best accuracy achieved: 13%

Out[18]: iter: 2500, accuracy: 13%  Loss: 11.501224886344

Out[18]: iter: 3000, accuracy: 14%  Loss: 11.49398984774
          -- best accuracy achieved: 14%

Out[18]: iter: 3500, accuracy: 16%  Loss: 11.487628759524
          -- best accuracy achieved: 16%

Out[18]: iter: 4000, accuracy: 17%  Loss: 11.492140238992
          -- best accuracy achieved: 17%
```

Loss barely changing: Learning rate is probably too low

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

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

```
Out[18]: iter: 0, accuracy: 10%  Loss: 0.023248429529449
          -- best accuracy achieved: 10%
Out[18]: iter: 500, accuracy: 10%  Loss: 11.522416713458
Out[18]: iter: 1000, accuracy: 10%  Loss: 11.517536122735
Out[18]: iter: 1500, accuracy: 11%  Loss: 11.508510566527
          -- best accuracy achieved: 11%
Out[18]: iter: 2000, accuracy: 13%  Loss: 11.510842908524
          -- best accuracy achieved: 13%
Out[18]: iter: 2500, accuracy: 13%  Loss: 11.501224886344
Out[18]: iter: 3000, accuracy: 14%  Loss: 11.49398984774
          -- best accuracy achieved: 14%
Out[18]: iter: 3500, accuracy: 16%  Loss: 11.487628759524
          -- best accuracy achieved: 16%
Out[18]: iter: 4000, accuracy: 17%  Loss: 11.492140238992
          -- best accuracy achieved: 17%
```

Loss barely changing: Learning rate is probably too low

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

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

```
Out[18]: iter: 0, accuracy: 10%  Loss: 0.023248429529449
          -- best accuracy achieved: 10%
Out[18]: iter: 500, accuracy: 10%  Loss: 11.522416713458
Out[18]: iter: 1000, accuracy: 10%  Loss: 11.517536122735
Out[18]: iter: 1500, accuracy: 11%  Loss: 11.508510566527
          -- best accuracy achieved: 11%
Out[18]: iter: 2000, accuracy: 13%  Loss: 11.510842908524
          -- best accuracy achieved: 13%
Out[18]: iter: 2500, accuracy: 13%  Loss: 11.501224886344
Out[18]: iter: 3000, accuracy: 14%  Loss: 11.49398984774
          -- best accuracy achieved: 14%
Out[18]: iter: 3500, accuracy: 16%  Loss: 11.487628759524
          -- best accuracy achieved: 16%
Out[18]: iter: 4000, accuracy: 17%  Loss: 11.49214023899
          -- best accuracy achieved: 17%
```

Notice train/val accuracy goes to 17% 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)
```

I like to start with small regularization and find learning rate that makes the loss go down.

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

loss not going down:
learning rate too low

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

```
Out[19]: iter: 0, accuracy: 11%  Loss: 0.023115084740835
          -- best accuracy achieved: 11%

Out[19]: iter: 500, accuracy: 13%  Loss: nan
          -- best accuracy achieved: 13%

Out[19]: iter: 1000, accuracy: 13%  Loss: nan

Out[19]: iter: 1500, accuracy: 13%  Loss: nan

Out[19]: iter: 2000, accuracy: 13%  Loss: nan
```

cost: NaN almost 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: 0.0235711978869
          -- 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
```

3e-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

I like to 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

note it’s best to optimize in log space!

```
for i = 1, 100 do
    init_model()
    lr = math.pow(10, torch.uniform(-7.0, -3.0))
    lambda = math.pow(10, torch.uniform(-5, 5))
    best_acc = train_and_test_loop(2000, lr, lambda)
    print(string.format("Try %d/%d Best val accuracy: %d, lr: %f, lambda: %f",i, 100, best_acc, lr, lambda))
end
```

```
Out[10]: Try 1/100 Best val accuracy: 16, lr: 0.000045, lambda: 4996.489302
Out[10]: Try 2/100 Best val accuracy: 31, lr: 0.000003, lambda: 0.001315
Out[10]: Try 3/100 Best val accuracy: 25, lr: 0.000001, lambda: 0.000012
Out[10]: Try 4/100 Best val accuracy: 24, lr: 0.000002, lambda: 216.397129
Out[10]: Try 5/100 Best val accuracy: 26, lr: 0.000007, lambda: 0.000012
Out[10]: Try 6/100 Best val accuracy: 29, lr: 0.000009, lambda: 275.964597
Out[10]: Try 7/100 Best val accuracy: 30, lr: 0.000021, lambda: 0.000253
Out[10]: Try 8/100 Best val accuracy: 13, lr: 0.000809, lambda: 4.339235
Out[10]: Try 9/100 Best val accuracy: 26, lr: 0.000003, lambda: 0.000062
Out[10]: Try 10/100 Best val accuracy: 27, lr: 0.000095, lambda: 18.288190
Out[10]: Try 11/100 Best val accuracy: 14, lr: 0.000000, lambda: 1333.400659
Out[10]: Try 12/100 Best val accuracy: 8, lr: 0.000311, lambda: 0.000020
Out[10]: Try 13/100 Best val accuracy: 8, lr: 0.000617, lambda: 0.000050
Out[10]: Try 14/100 Best val accuracy: 34, lr: 0.000013, lambda: 0.124955
Out[10]: Try 15/100 Best val accuracy: 17, lr: 0.000013, lambda: 5262.631955
```

nice

Now run finer search...

```
for i = 1, 100 do
    init_model()
    lr = math.pow(10, torch.uniform(-7.0, -3.0))
    lambda = math.pow(10, torch.uniform(-5, 5))
    best_acc = train_and_test_loop(2000, lr, lambda)
    print(string.format("Try %d/%d Best val accuracy: %d, lr
end
```

adjust range

```
for i = 1, 100 do
    init_model()
    lr = math.pow(10, torch.uniform(-6.0, -4.0))
    lambda = math.pow(10, torch.uniform(-3, 1))
    best_acc = train_and_test_loop(2000, lr, lambda)
    print(string.format("Try %d/%d Best val accuracy: %d,
end
```

```
Out[11]: Try 1/100 Best val accuracy: 35, lr: 0.000055, lambda: 0.002026
Out[11]: Try 2/100 Best val accuracy: 28, lr: 0.000001, lambda: 1.994656
Out[11]: Try 3/100 Best val accuracy: 32, lr: 0.000003, lambda: 0.483409
Out[11]: Try 4/100 Best val accuracy: 37, lr: 0.000032, lambda: 1.981563
Out[11]: Try 5/100 Best val accuracy: 27, lr: 0.000003, lambda: 0.004578
Out[11]: Try 6/100 Best val accuracy: 28, lr: 0.000004, lambda: 0.082862
Out[11]: Try 7/100 Best val accuracy: 34, lr: 0.000020, lambda: 0.003083
Out[11]: Try 8/100 Best val accuracy: 28, lr: 0.000054, lambda: 0.064499
Out[11]: Try 9/100 Best val accuracy: 31, lr: 0.000003, lambda: 0.004361
Out[11]: Try 10/100 Best val accuracy: 32, lr: 0.000004, lambda: 0.001610
Out[11]: Try 11/100 Best val accuracy: 31, lr: 0.000006, lambda: 0.300821
```

37% - relatively good for a 1-layer neural net and only 2000 iterations

Now run finer search...

```
for i = 1, 100 do
    init_model()
    lr = math.pow(10, torch.uniform(-7.0, -3.0))
    lambda = math.pow(10, torch.uniform(-5, 5))
    best_acc = train_and_test_loop(2000, lr, lambda)
    print(string.format("Try %d/%d Best val accuracy: %d, lr
end
```

adjust range

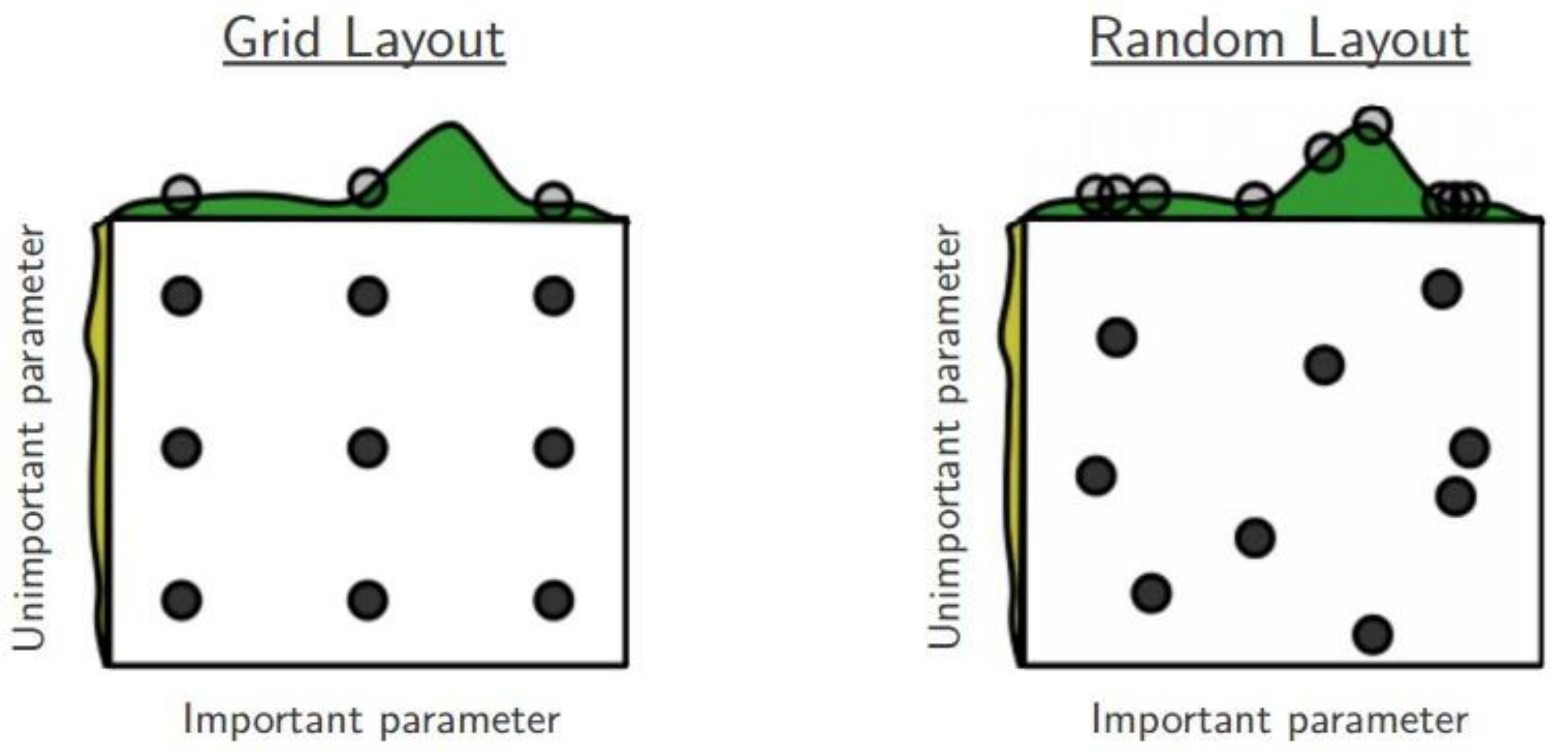
```
for i = 1, 100 do
    init_model()
    lr = math.pow(10, torch.uniform(-6.0, -4.0))
    lambda = math.pow(10, torch.uniform(-3, 1))
    best_acc = train_and_test_loop(2000, lr, lambda)
    print(string.format("Try %d/%d Best val accuracy: %d,
end
```

```
Out[11]: Try 1/100 Best val accuracy: 35, lr: 0.000055, lambda: 0.002026
Out[11]: Try 2/100 Best val accuracy: 28, lr: 0.000001, lambda: 1.994656
Out[11]: Try 3/100 Best val accuracy: 32, lr: 0.000003, lambda: 0.483409
Out[11]: Try 4/100 Best val accuracy: 37, lr: 0.000032, lambda: 1.981563
Out[11]: Try 5/100 Best val accuracy: 27, lr: 0.000003, lambda: 0.004578
Out[11]: Try 6/100 Best val accuracy: 28, lr: 0.000004, lambda: 0.082862
Out[11]: Try 7/100 Best val accuracy: 34, lr: 0.000020, lambda: 0.003083
Out[11]: Try 8/100 Best val accuracy: 28, lr: 0.000054, lambda: 0.064499
Out[11]: Try 9/100 Best val accuracy: 31, lr: 0.000003, lambda: 0.004361
Out[11]: Try 10/100 Best val accuracy: 32, lr: 0.000004, lambda: 0.001610
Out[11]: Try 11/100 Best val accuracy: 31, lr: 0.000006, lambda: 0.300821
```

37% - 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



Random Search for Hyper-Parameter Optimization

Bergstra and Bengio, 2012

http://www.jmlr.org/papers/volume13/bergstra12a/bergstra12a.pdf mlflow 2.2.2 Experiments Models GitHub Docs

Experiments

Search Experiments

Default

Default

Provide Feedback

Share

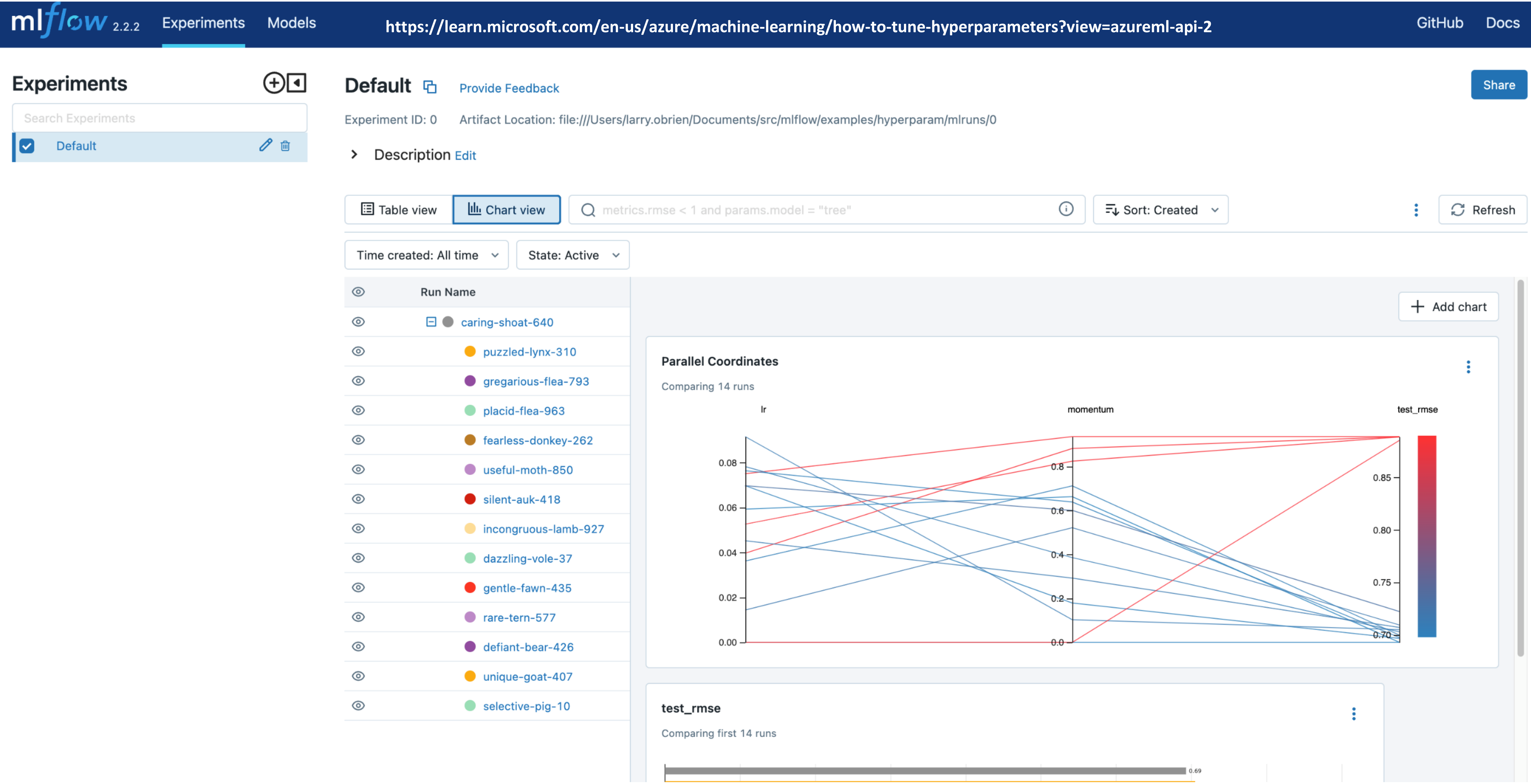
Experiment ID: 0   Artifact Location: file:///Users/larry.obrien/Documents/src/mlflow/examples/hyperparam/mlruns/0

Description Edit

Table view | Chart view | metrics.rmse < 1 and params.model = "tree" | Sort: Created | Columns | Refresh

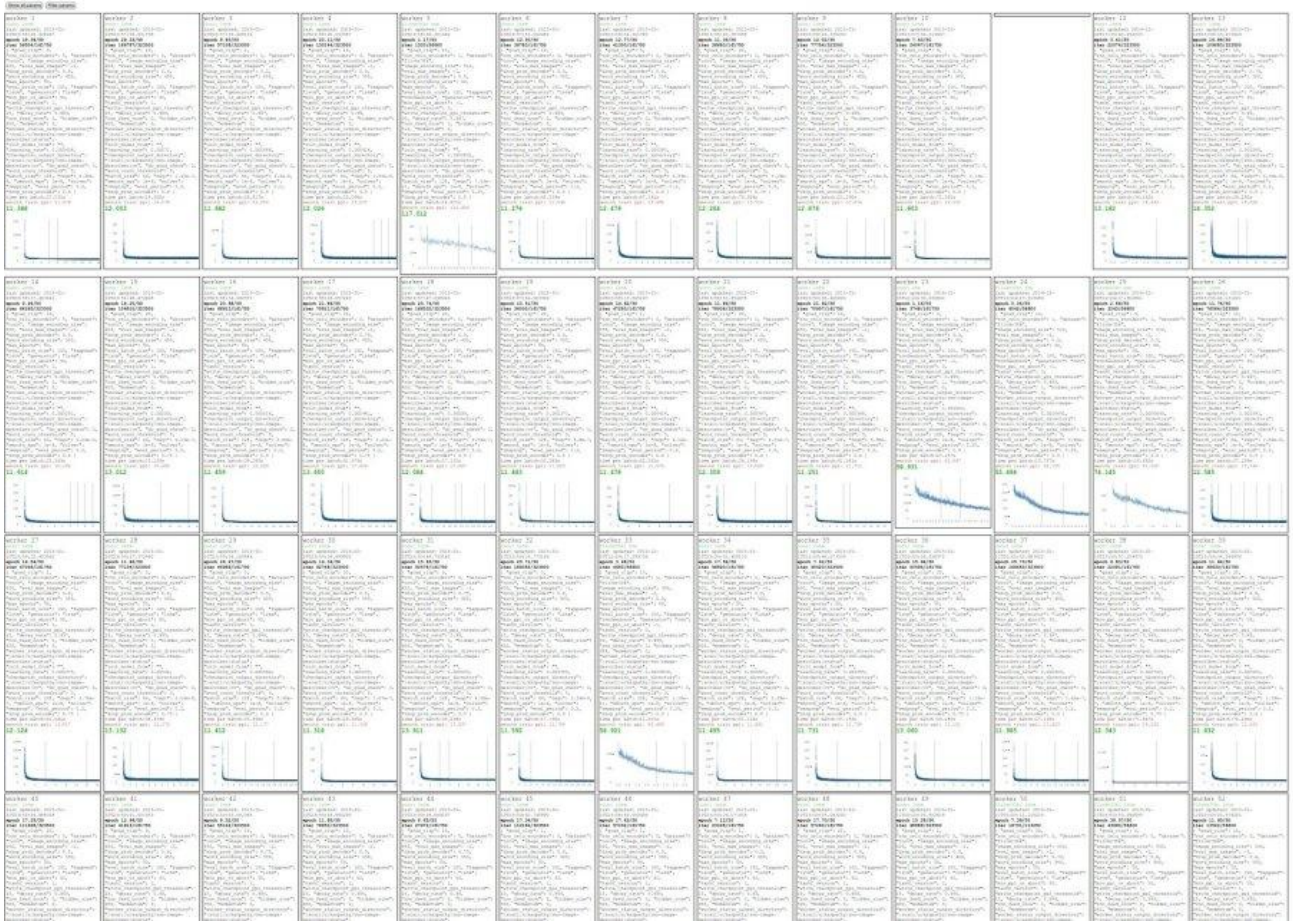
Time created: All time | State: Active

						Metrics	Parameters	
Run Name	Created	Duration	Source	Models		test_rmse	lr	momentum
caring-shoat-640	5 minutes ago	4.7min	hyperpa...	-		0.693	-	-
puzzled-lynx-310	1 minute ago	21.9s	search_...	tensorflow		0.705	0.0915459...	0.1026761...
gregarious-flea-793	1 minute ago	21.4s	search_...	tensorflow		0.889	0.0750214...	0.9362895...
placid-flea-963	2 minutes ago	22.9s	search_...	tensorflow		0.697	0.0763516...	0.6388090...
fearless-donkey-262	2 minutes ago	21.8s	search_...	tensorflow		0.889	0.0396457...	0.8823604...
useful-moth-850	2 minutes ago	22.4s	search_...	tensorflow		0.693	0.0592645...	0.6627415...
silent-auk-418	3 minutes ago	22.3s	search_...	tensorflow		0.889	0.0525756...	0.8246142...
incongruous-lamb-927	3 minutes ago	21.8s	search_...	tensorflow		0.709	0.0143746...	0.5214591...
dazzling-vole-37	3 minutes ago	23.0s	search_...	tensorflow		0.703	0.0782039...	0.3855263...
gentle-fawn-435	4 minutes ago	21.8s	search_...	tensorflow		0.697	0.0697095...	0.1790176...
rare-tern-577	4 minutes ago	22.5s	search_...	tensorflow		0.722	0.0696752...	0.6004828...
defiant-bear-426	5 minutes ago	23.2s	search_...	tensorflow		0.707	0.0452154...	0.2921335...
unique-goat-407	5 minutes ago	23.4s	search_...	tensorflow		0.7	0.0361839...	0.7121900...
selective-pig-10	5 minutes ago	8.4s	search_...	-		0.885	0	0

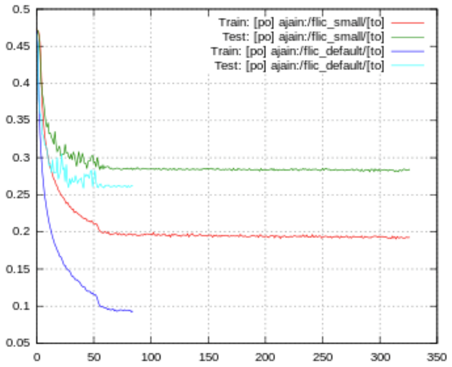


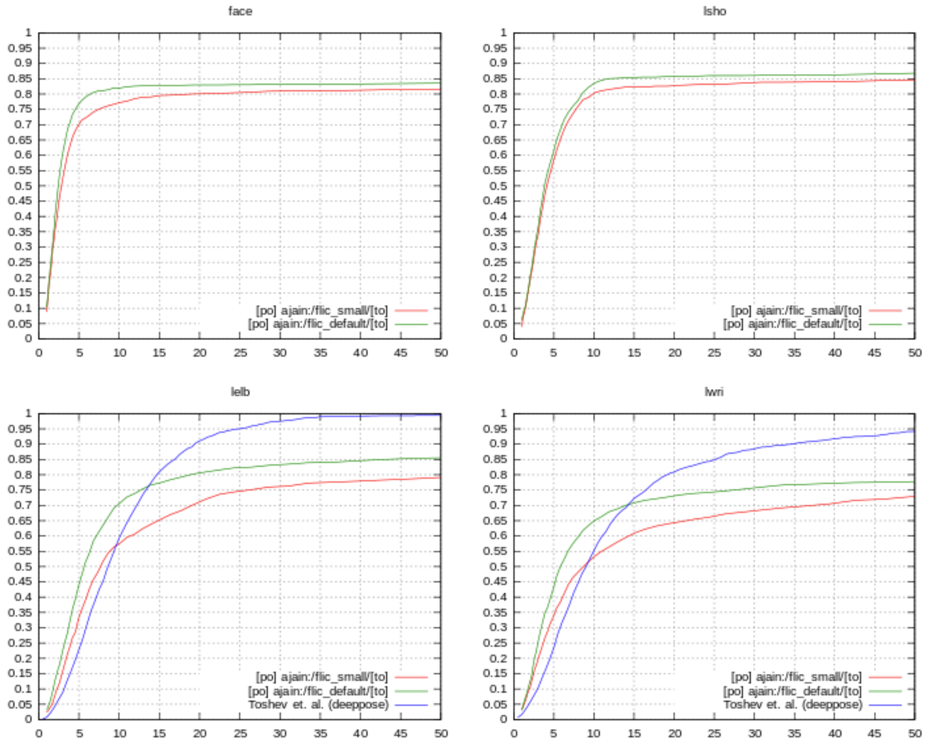
IIT Bombay, Perceptive Code LLC

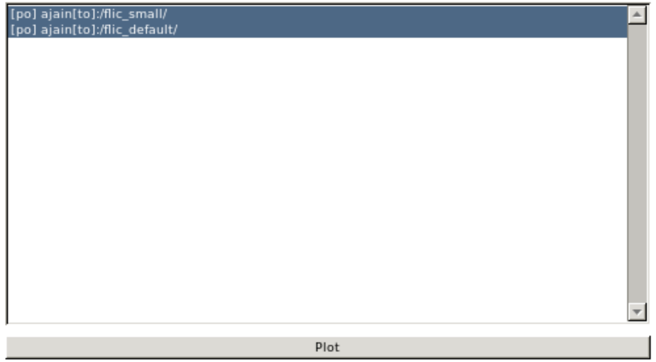
Karpathy’s cross-validation “command center”



My cross-validation “command center”







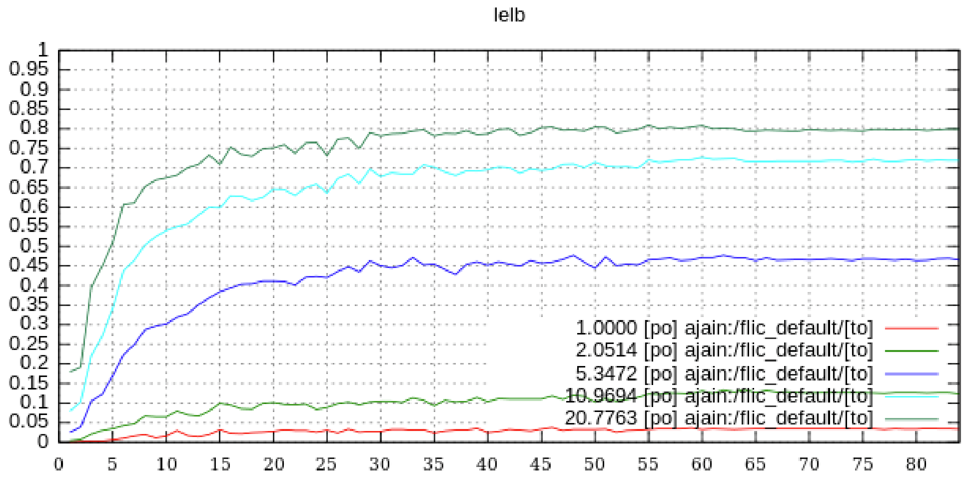
[po] ajain:/flic_small/[to]

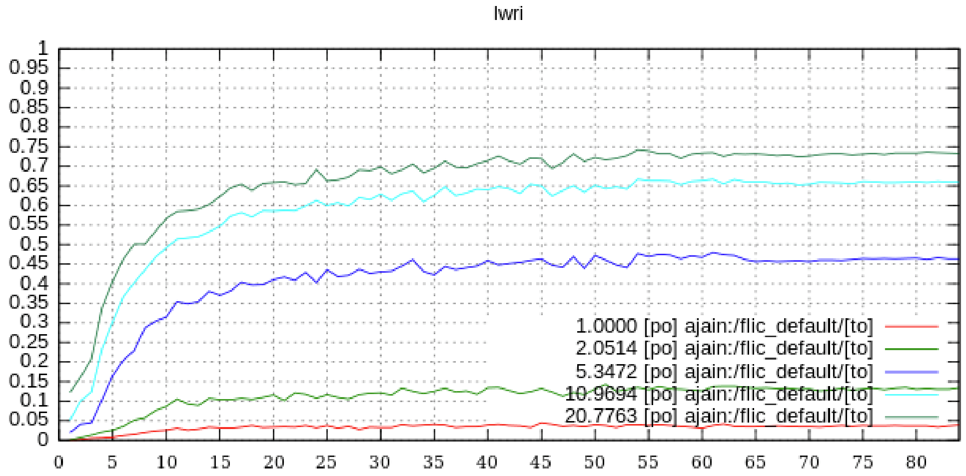
```
 2  batch_normalization = false
 3  batch_size = 16
 4  batch_size_per_gpu = 16
 5  big_model = false
 6  body_scale_range = {}
 7  compress_src_model =
 8  conv_1x1_size = 1
 9  conv_nfeats = { 1 = { 1 = 16, 2 = 16, 3 = 16, }, 2 = { 1 = 16, 2 = 16, 3 = 16, }, 3 = { 1 = 16, 2 = 16, 3 = 16, },}
10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



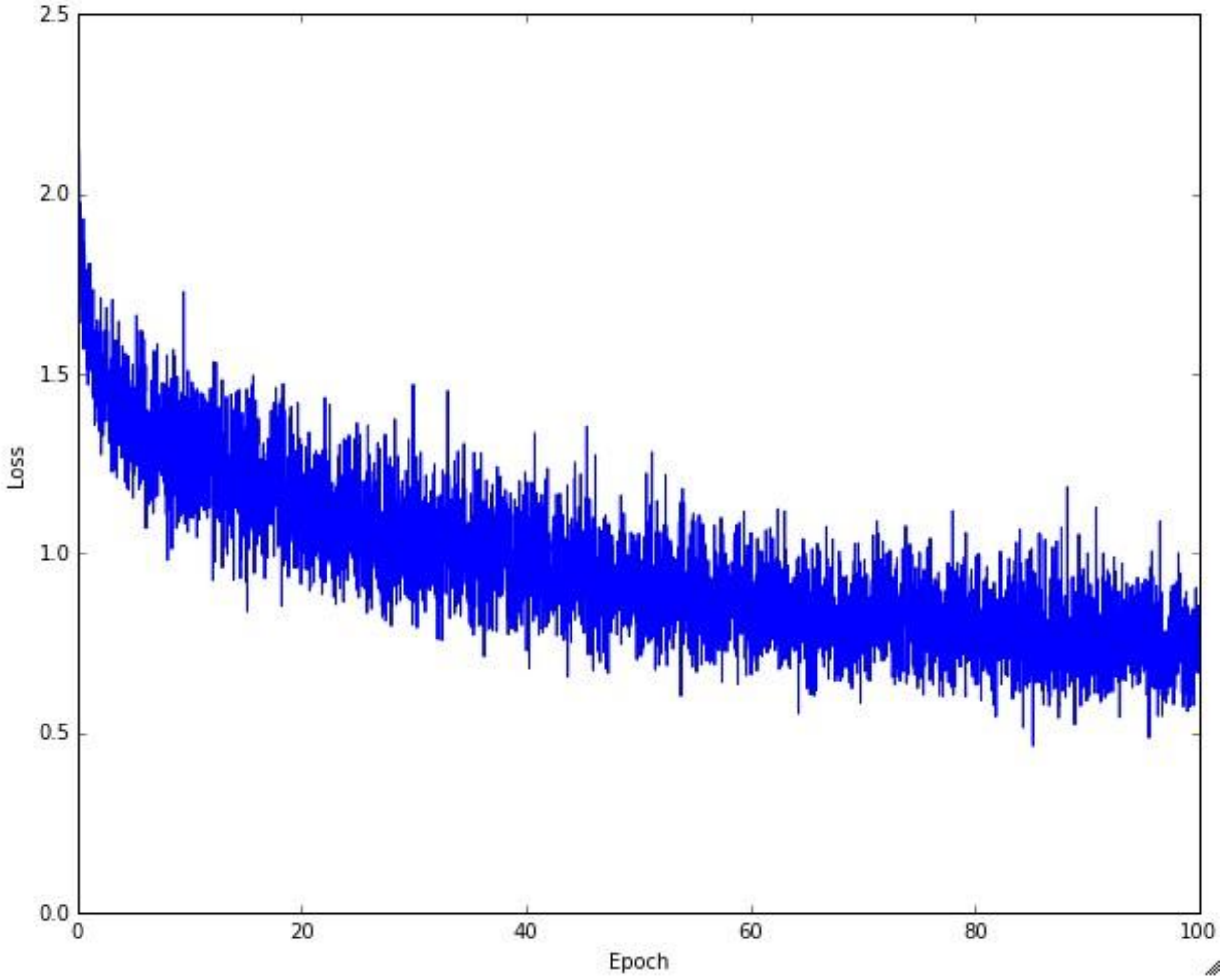


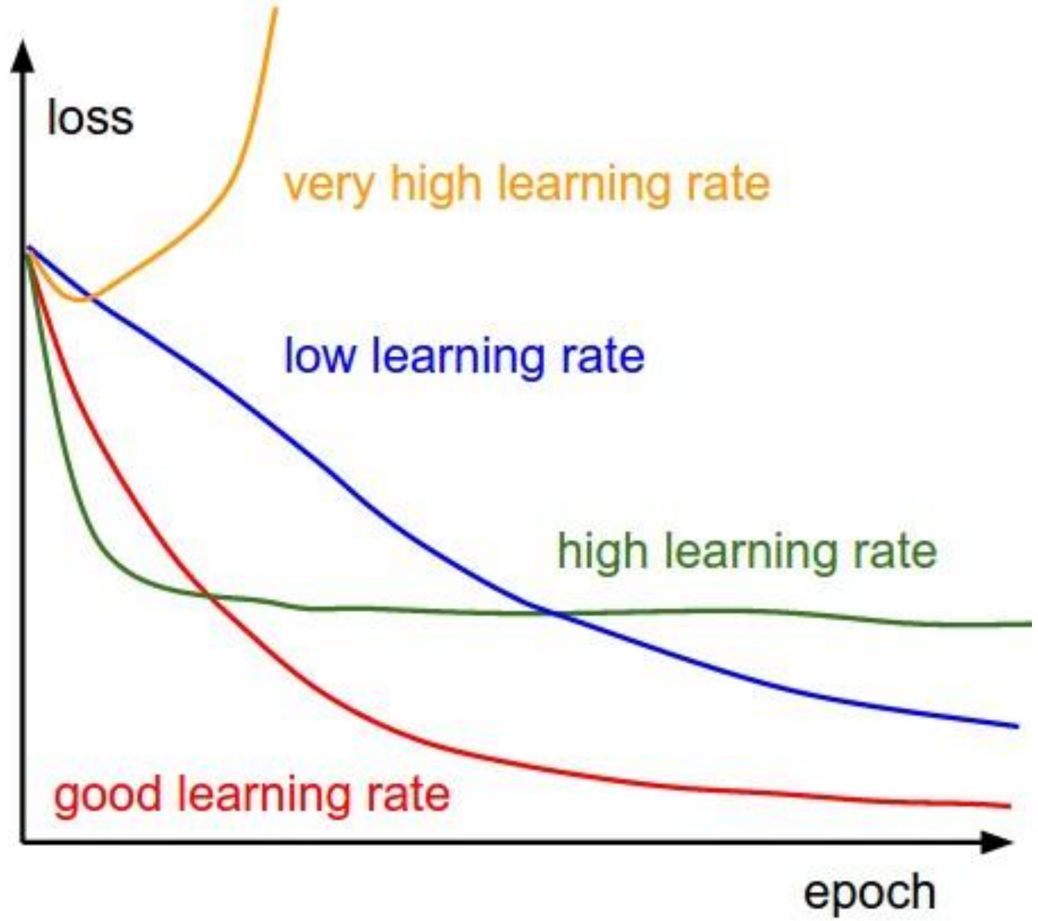
[po] ajain[to]:/flic_small/

Plot Epochs

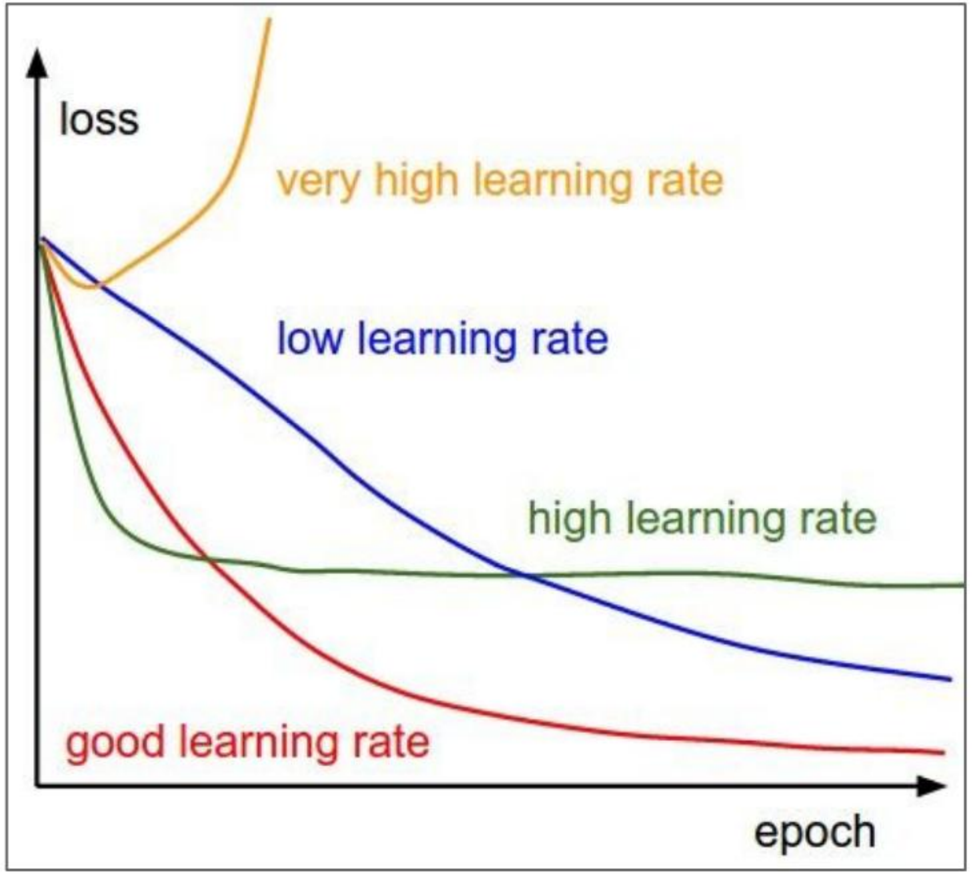
My cross-validation "command center"

Monitor and visualize the loss curve





Use Learning Rate Decay



=> Learning rate decay over time!

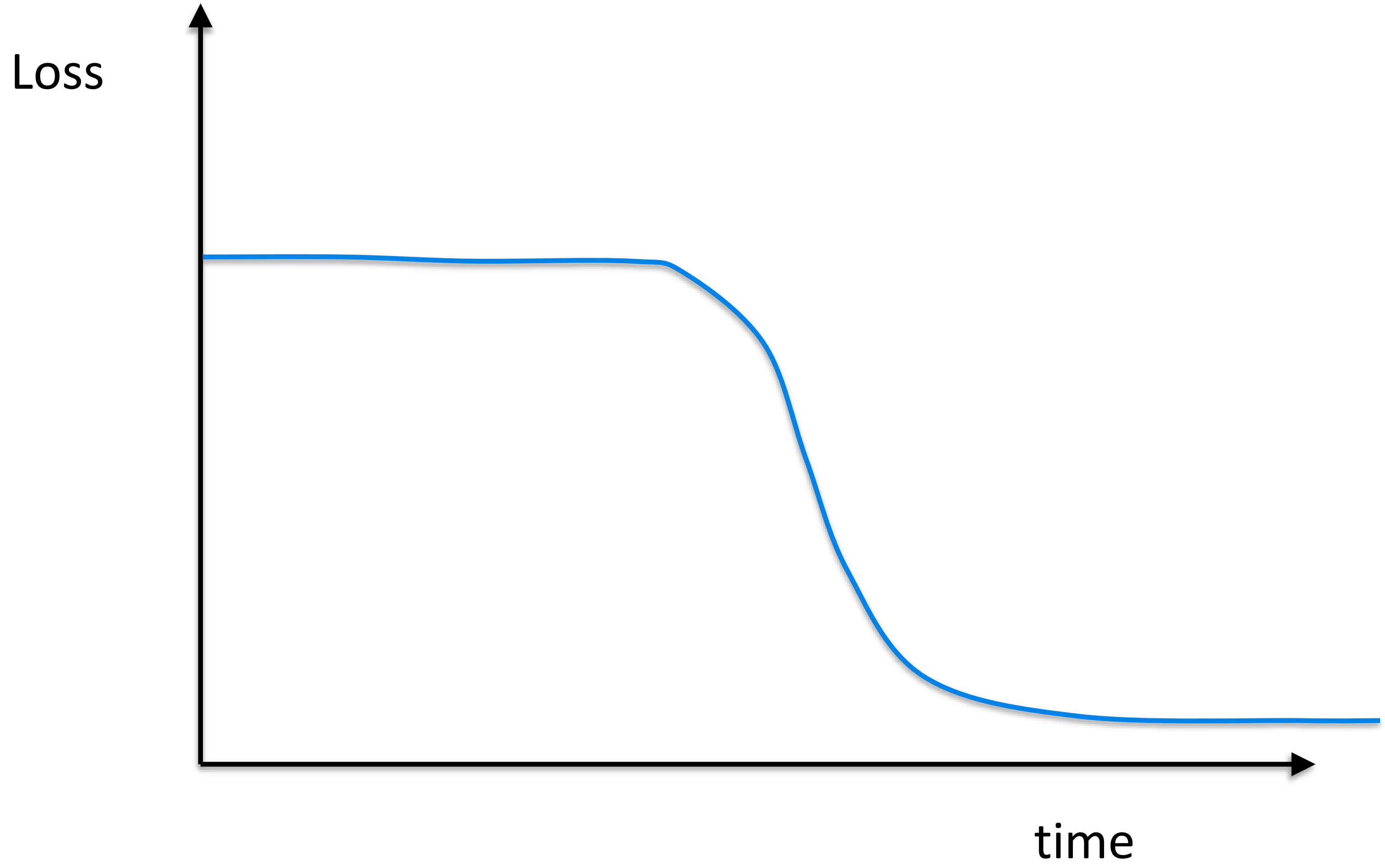
step decay:
e.g. decay learning rate by half every few epochs.

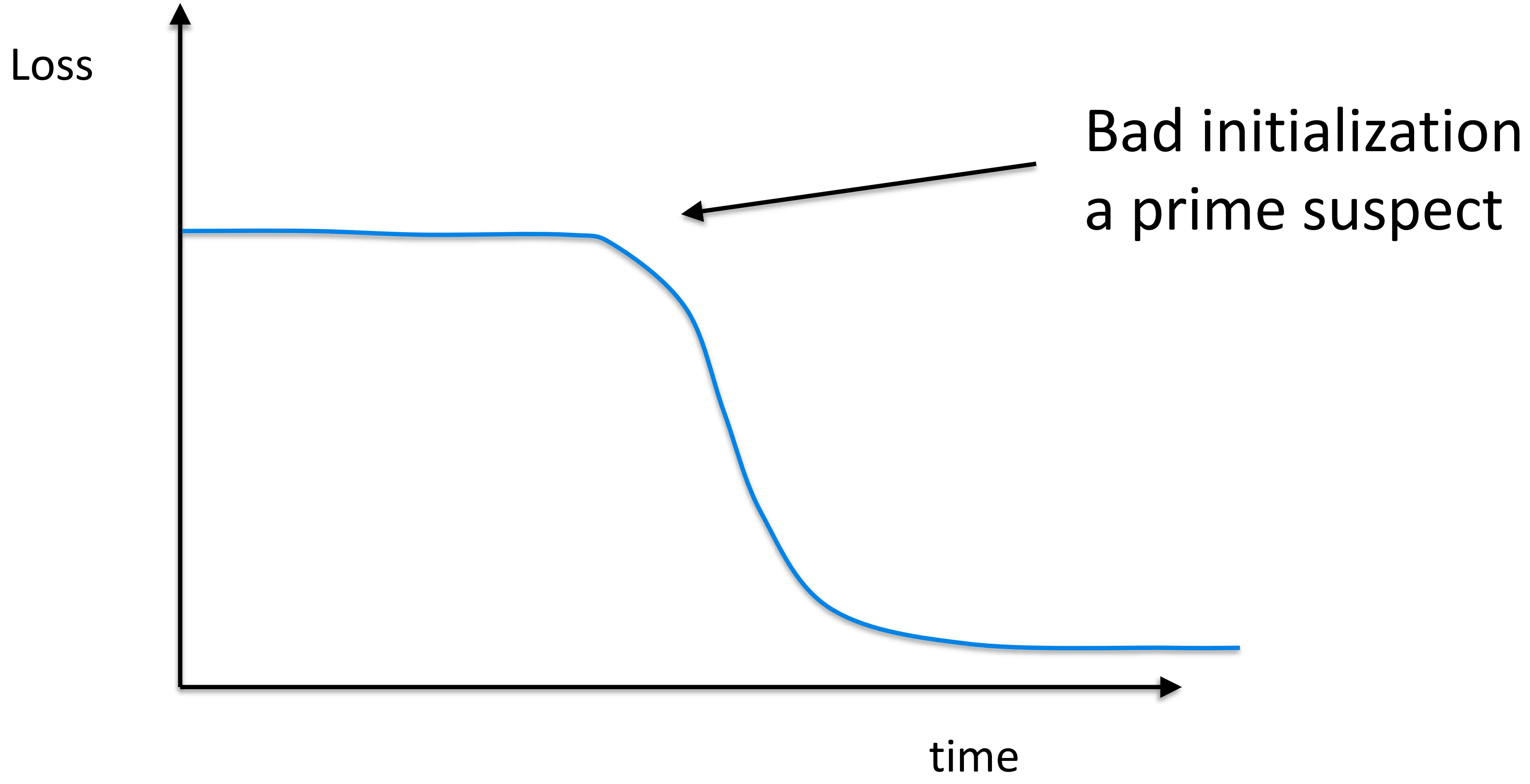
exponential decay:

$$\alpha = \alpha_0 e^{-kt}$$

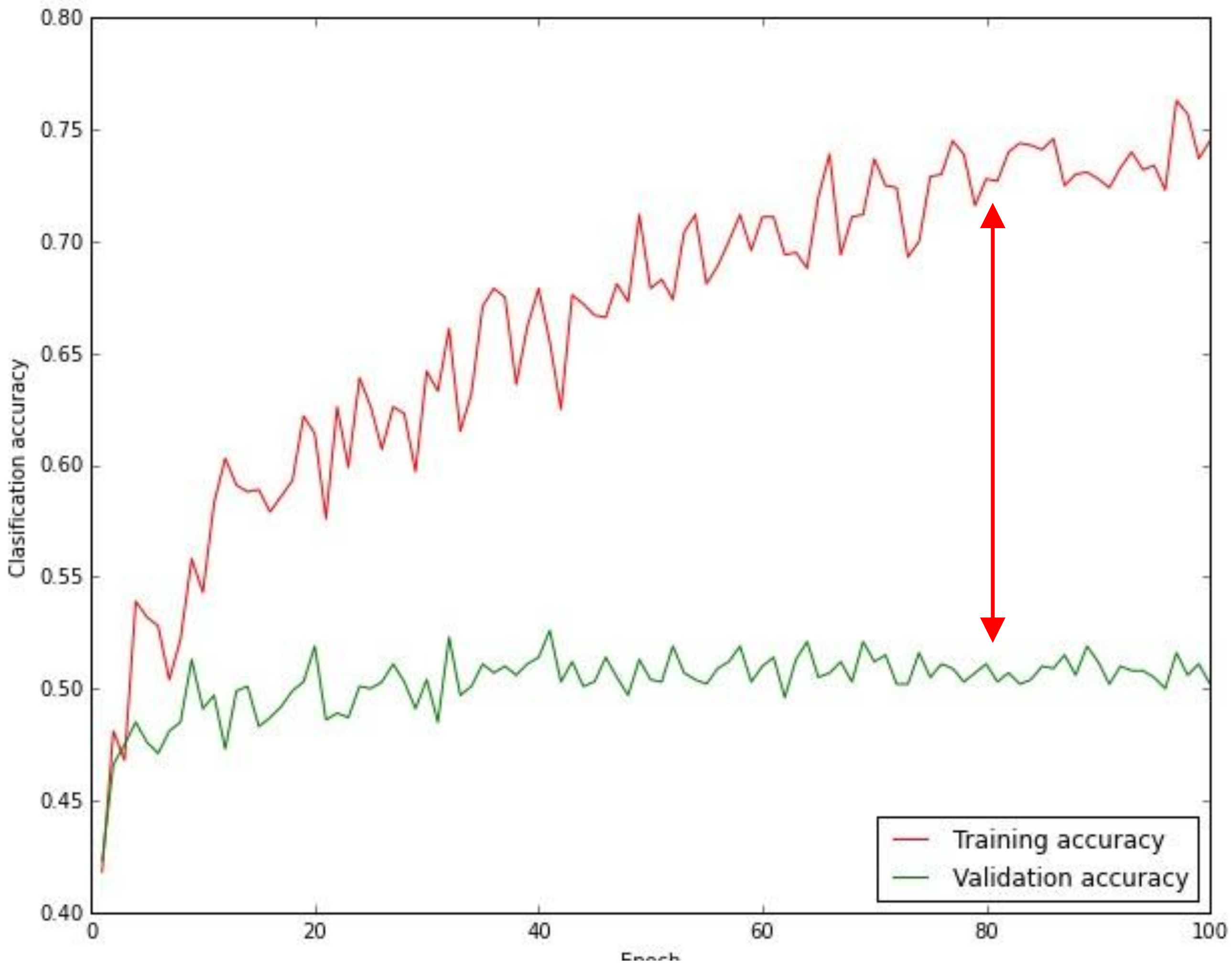
1/t decay:

$$\alpha = \alpha_0 / (1 + kt)$$


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:

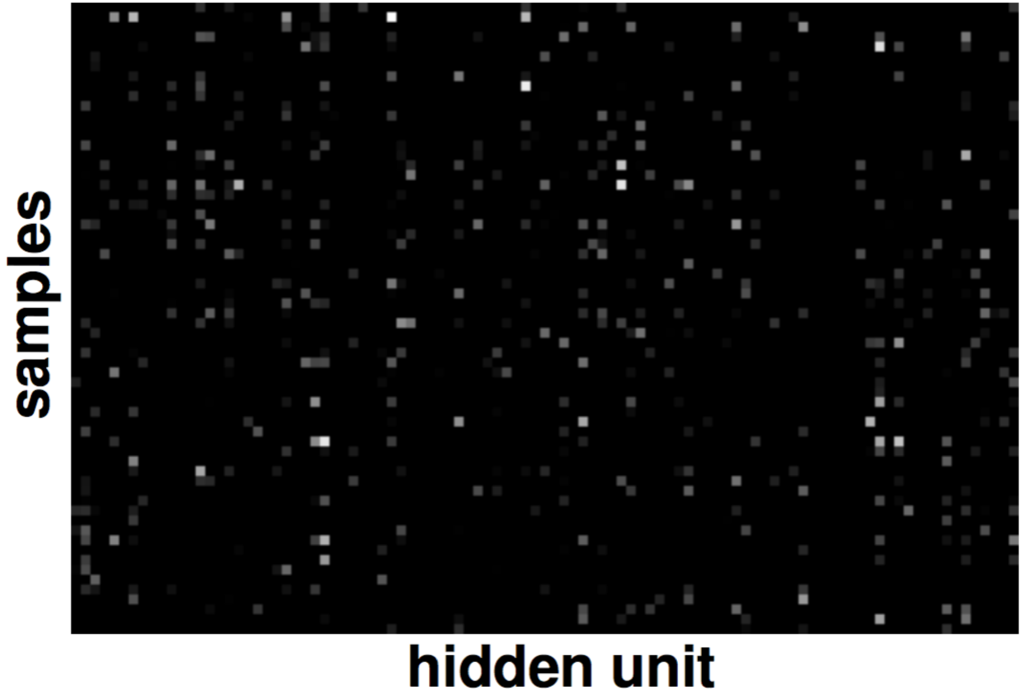
```
function gradient_descent(model, lr)
    w_scale = torch.norm(model.W:view(model.W:nElement()), 2, 1)
    update_scale = torch.norm(lr * model.gradW:view(model.gradW:nElement()), 2, 1)
    model.W = model.W + lr * model.gradW
    model.b = model.b + lr * model.gradb
    print(update_scale/w_scale) -- Want ~1e-3
end
```

ratio between the values and updates: ~ 0.0002 / 0.02 = 0.01 (about okay)

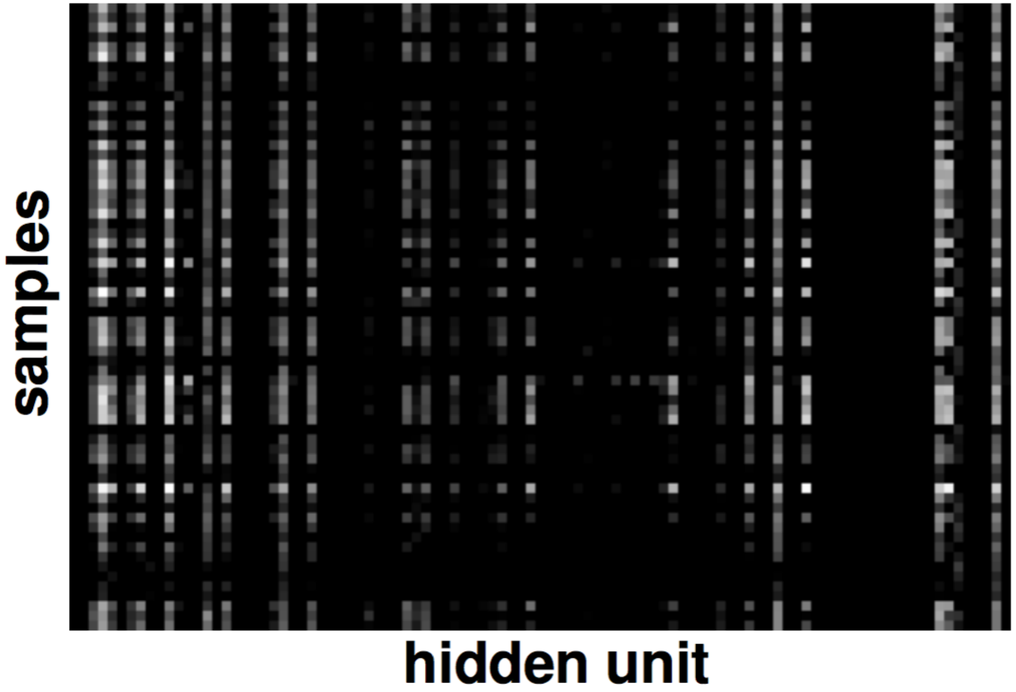
want this to be somewhere around 0.001 or so

Visualize Activations

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



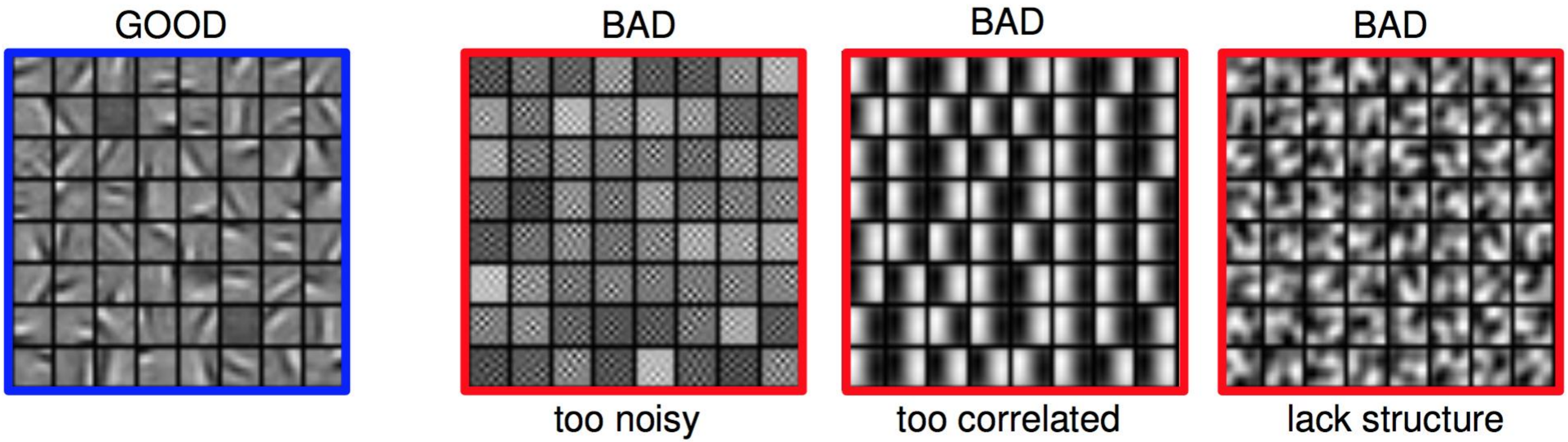
Good training: hidden units are sparse across samples and across features.



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

Visualize (initial) Convolution Layer Weights

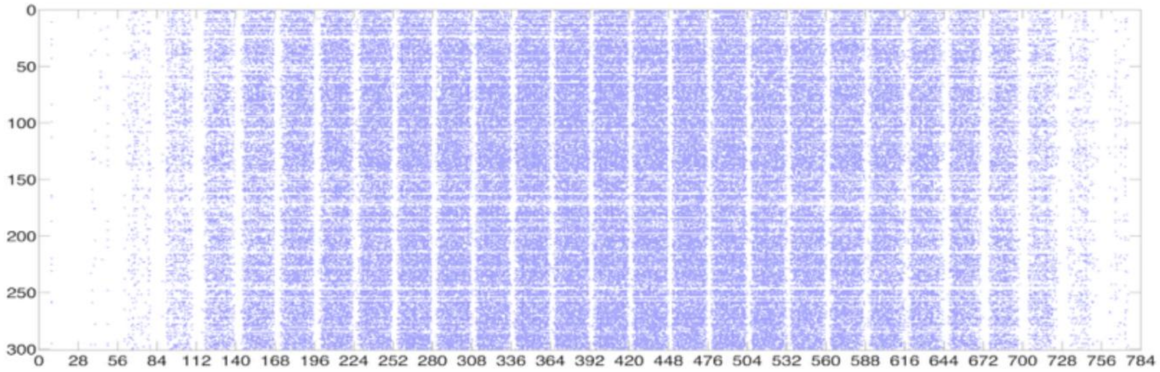
- Visualize features (feature maps need to be uncorrelated) and have high variance.



Good training: learned filters exhibit structure and are uncorrelated.

Visualize Linear Layer (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



why your model is not working? - data issues

- Check your input data (all zeros, using same batch over and over)
- try random input (if error behaves same for random data, your network is turning real data into garbage)
- is there too much noise in the dataset? (bad labels)
- shuffle the dataset (ordered by label could -ively impact learning)
- reduce class imbalance (balance your loss function)
- have enough data
- reduce batch size (huge batch size can reduce generalization ability)
- Don’t use too much data augmentation (augmentation has regularization effect, but too much of it combined with other regularizers will underfit)
- check preprocessing of your pretrained model
- check the preprocessing for train/val/test set (any preprocessing stats must only be computed on train set then applied to test set. Don’t compute on whole data and divide the dataset into train/val/test)

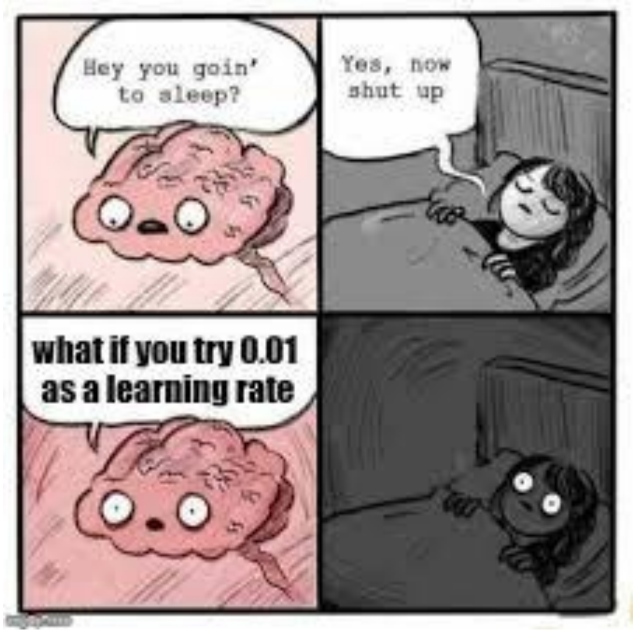
why your model is not working? - implementation issues

- try solving a simpler version of the problem (if the target output is class and coordinates, try limiting the prediction to class only)
- look for correct loss "at chance" (if we have 10 classes, at chance means we will get the correct class 10% of the time, and the Softmax loss is the negative log probability of the correct class so: -ln(0.1) = 2.302)
- check your loss function (for your custom loss function, add unit tests)
- test any custom layers
- check for "frozen" layers (check if you unintentionally disabled gradient updates for some layers)
- check for hidden dimension errors

why your model is not working? - training issues

- Solve for really small dataset
- check weights initialization (if unsure, use Xavier or He)
- change your hyperparameters
- reduce regularization (too much can underfit badly. See “Practical Deep Learning for coders” for more details)
- give it time
- visualize the training (monitor activations, weights and updates. Use tensorboard or crayon.)
- check for exploding or vanishing gradients (check layer updates, as very large values can indicate exploding gradients. Check layer activations, “A good standard deviation for the activations is on the order of 0.5 to 2.0”.)
- Overcoming NaNs (reduce LR, avoid division by zero or ln(0). Check Russell Stewart’s “How to deal with NaNs”.)

General thoughts - Trial & error



"It is ironic to say that, we moved from feature engineering to feature learning to avoid hand-crafting parameters. But, we ended up fine tuning more hyperparameters to train our neural network..."

- Anonymous Reddit user (probably a frustrated Grad student)