NEURAL NETWORK FOR MACHINE LEARNING 2

한재근 과장 | jahan@nvidia.com
유현곤 부장 | hryu@nvidia.com / 양한별 과장 | hanbyuly@nvidia.com

NVIDIA

AGENDA

Training Neural Network

Stochastic Gradient Descent

Learning Rate Control

Data Preprocessing

Weight Initialization

Data Augmentation

Regularization & Dropout

Hyper parameter Exploration

Training of Neural Networks

1. Divide data on 2 sets: training set and validation set
2. Define network and training procedure
   - network architecture
   - loss function
   - preprocessing and data augmentation
   - training algorithm and parameters (batch size, initialization,…)
3. Training:
   for (t = 0; t < T; t++)
   train Net(W) on training set
4. test Net(W) on validation set
   If (not state-of-the art) goto 2

Training of Neural Networks is difficult

A lot of hyper-parameters to choose:

- Loss function
- Network architecture: # of layers, and # of channels/layer
- Weight initialization
- SGD algorithm and learning rate policy
- Data preprocessing: scaling, augmentation …

Optimization Algorithms for CNN training

Batch Gradient Descent

We want to minimize loss over training set with N samples (x_n, y_n):

$$L(w) = \frac{1}{N}\sum_{n=1}^{N} E(f(x_n, w), y_n)$$

Batch optimization:

1. accumulate gradients over all samples in training set

$$\frac{\partial E}{\partial y_{l-1}} = \frac{\partial E}{\partial y_l} \times \frac{\partial y_l(w, y_{l-1})}{\partial y_{l-1}}\ ; \qquad \frac{\partial E}{\partial w_l} = \frac{\partial E}{\partial y_l} \times \frac{\partial y_l(w, y_{l-1})}{\partial w_l}$$

2. update W:

$$W(t+1) = W(t) - \lambda * \frac{1}{N}\sum_{n=1}^{N} \frac{\partial E}{\partial w}((x_n, w), y_n)$$

Issue:

Imagenet has 10^6 images→ gradient computation for whole set is expensive

Stochastic Gradient Descent

Stochastic Gradient Descent (on-line learning):

1. Randomly choose sample (x_k, y_k):
2. $W(t+1) = W(t) - \lambda * \frac{\partial E}{\partial w}((x_k, w), y_k)$

Stochastic Gradient Descent with mini-batches:

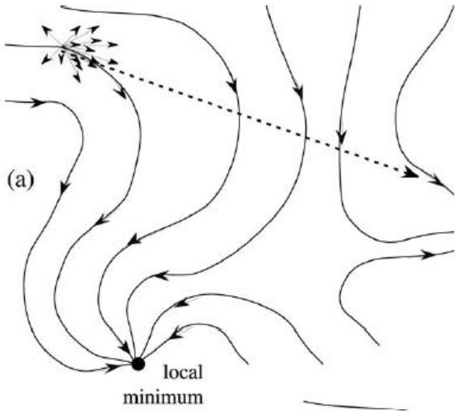
1. divide the dataset into small mini-batches, choose samples from different classes
2. compute the gradient using a single m-batch, make an update
3. move to the next mini-batch …

Don’t forget to shuffle / shift data between epochs!

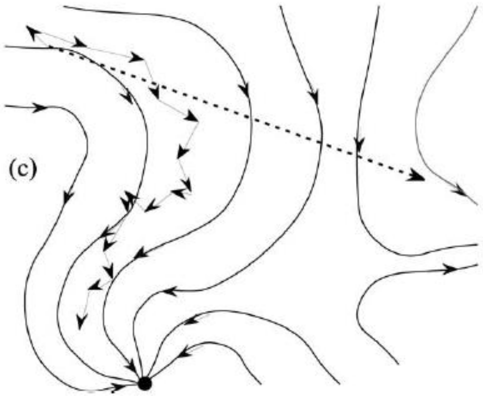
Stochastic Gradient Descent

N = batch: gradient computation is heavy, step is small

N =1 (on-line training): gradient is very noisy, zig-zags around “true” gradient



batch



mini-batch

Mini-batch training follows the curve of gradient:

the expected value of the weight change for on-line training is continuously pointing in the direction of the gradient at the current point in weight space.

Wilson, ‘On The general inefficiency of batch training for gradient descent learning, 2003

Batch gradient descent

$$J_{train}(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2$$

Repeat {

$$\theta_j := \theta_j - \alpha \underbrace{\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}}_{\frac{\partial}{\partial\theta_j} J_{train}(\theta)}$$

(for every $j = 0, \ldots, n$)

}

Stochastic gradient descent

$$cost(\theta, (x^{(i)}, y^{(i)})) = \frac{1}{2}(h_\theta(x^{(i)}) - y^{(i)})^2$$

$$J_{train}(\theta) = \frac{1}{m}\sum_{i=1}^{m} cost(\theta, (x^{(i)}, y^{(i)}))$$

1. Randomly shuffle dataset.

2. Repeat {

for $i=1, \ldots, m$ {

$$\theta_j := \theta_j - \alpha \underbrace{(h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}}_{\frac{\partial}{\partial\theta_j} cost(\theta, (x^{(i)}, y^{(i)}))}$$

(for $j=0, \ldots, n$)

}

}

$(x^{(i)}, y^{(i)})$.

Mini-batch Gradient Descent

SGD with mini-batch:

1. faster than batch
2. more robust to redundant data
3. Behaves better in local minima/ saddle.

"Use stochastic gradient descent when training time is the bottleneck"
Leon Bottou, Stochastic Gradient Descent Tricks

Mini-batch size and Learning Rate

How to change learning rate when we change the batch size?

“Classical ML” rule: on-line / mini-batch training with large learning rate is much more stable than batch training with the same learning rate.

Convolutional NN (for problems with large number of classes)

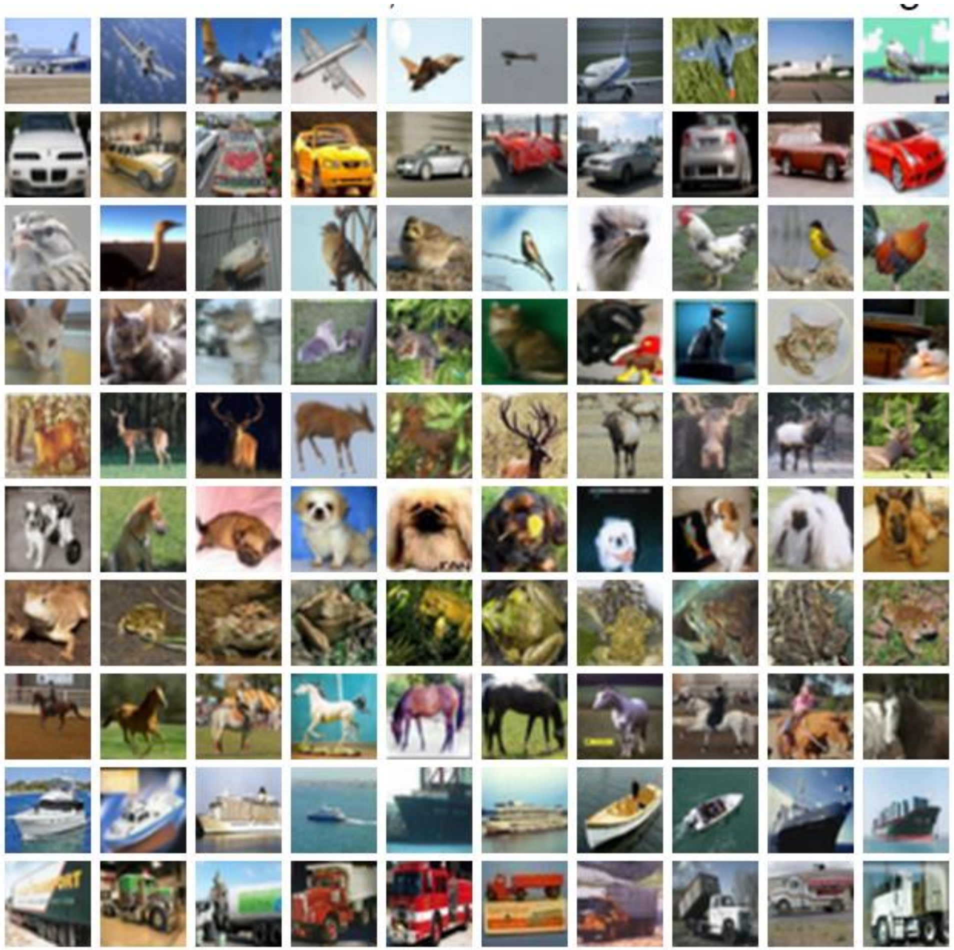
Alex Krizhevsky (“One weird trick on parallelizing CNN”):

1. **Theory:** multiply the learning rate by $k^{1/2}$ when increase the batch size by K to keep the variance in the gradient expectation constant.
2. **Practice:** to multiply the learning rate by k when multiplying the batch size by k.

WARNING:
This rule does not work when mini-batch size become too large !

Example: CIFAR0-10

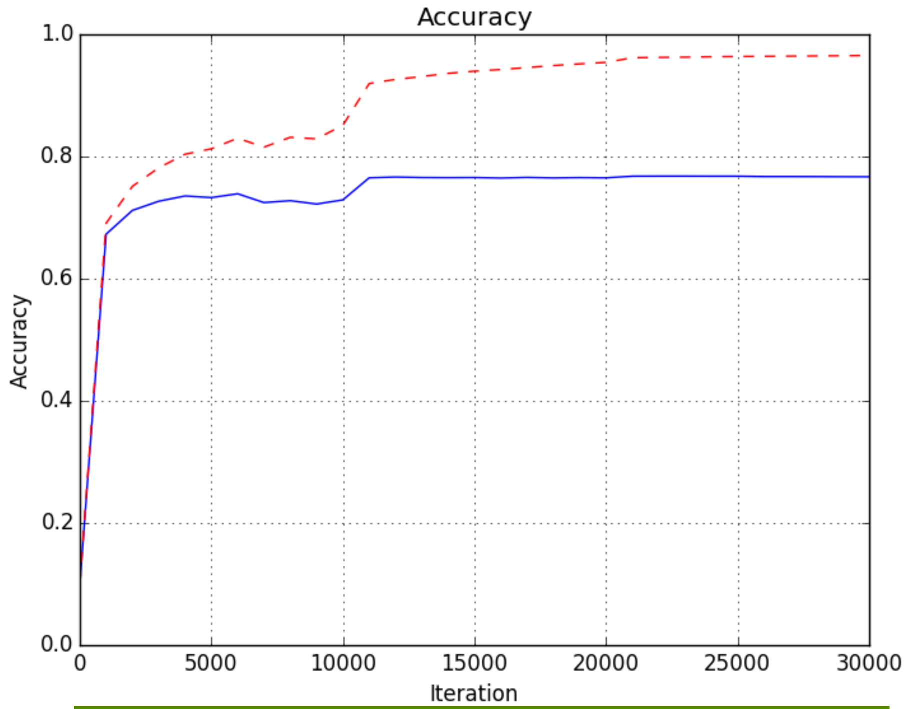
airplane
automobile
bird
cat
deer
dog
frog
horse
ship
truck



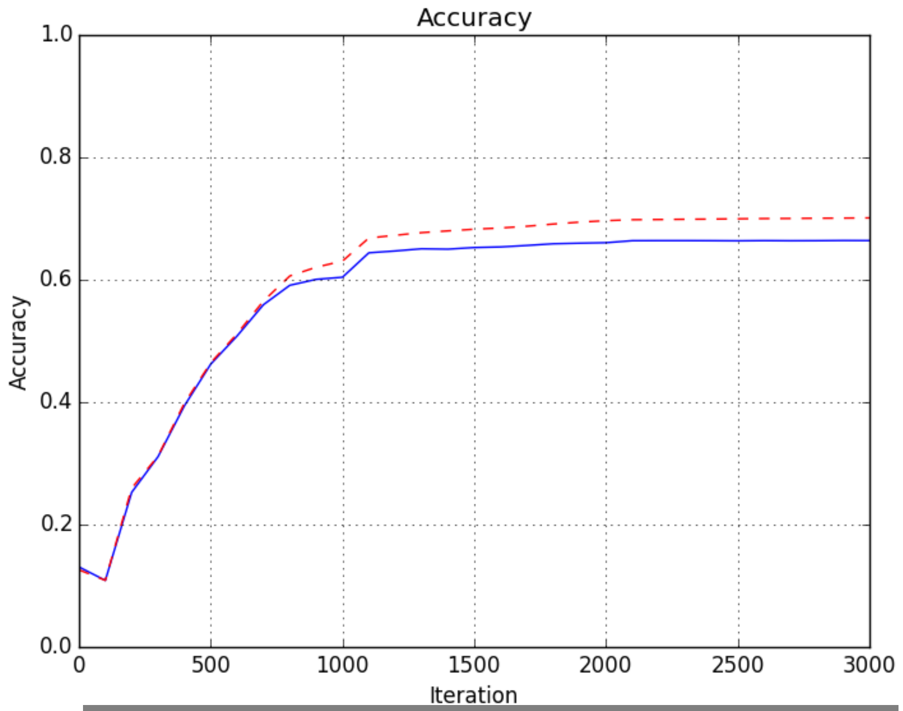
Learning Rate & its control

Example: CIFAR-10 training

Training and testing accuracy



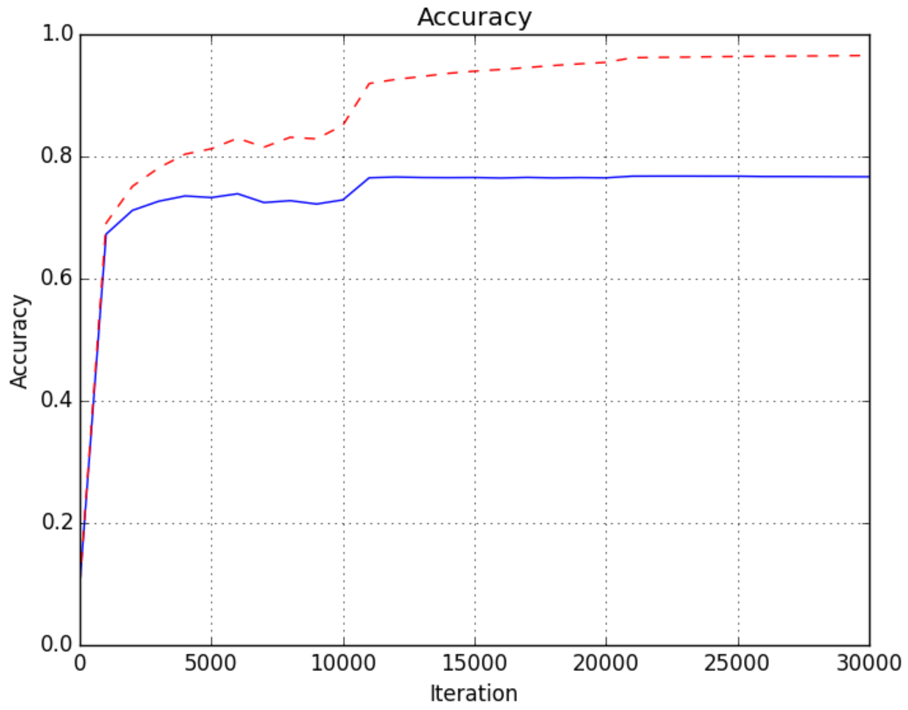
Batch =100, initial lr =0.001



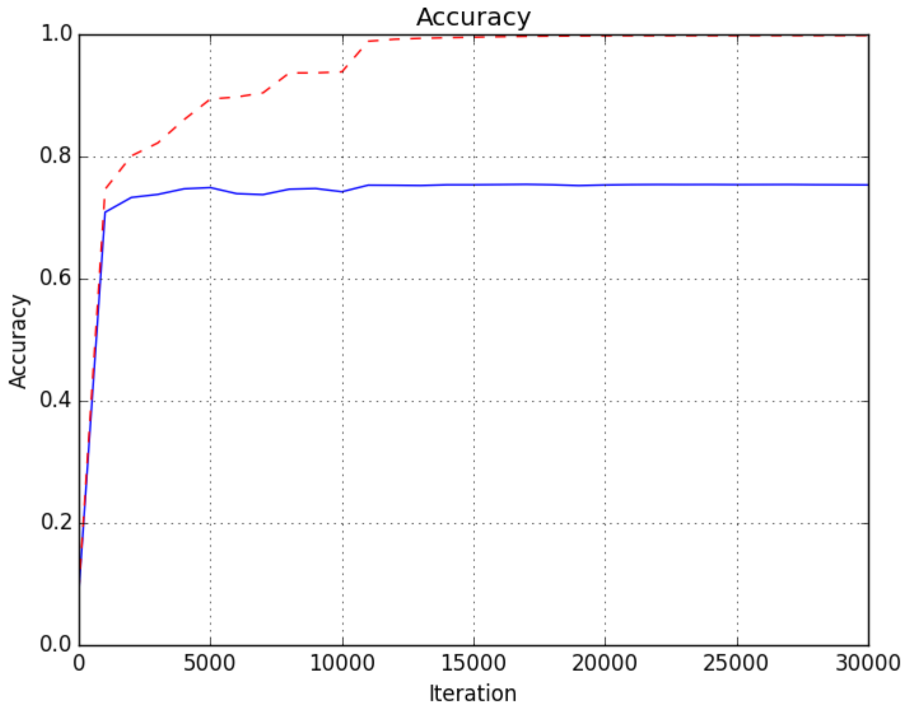
Batch =1000, initial **lr =0.01**

Example: CIFAR-10

Training and testing accuracy



Batch =100,initial lr =0.001



Batch =1000, intial lr =0.001

Learning Rate Adaptation

$$W(t+1) = W(t) - \lambda(t) * \frac{\partial E}{\partial w}$$

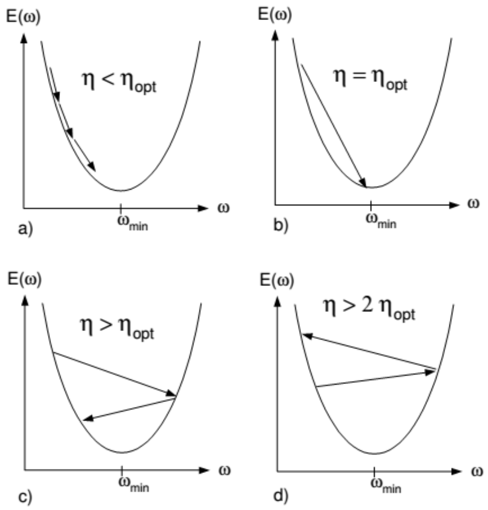
Classical learning rate annealing:

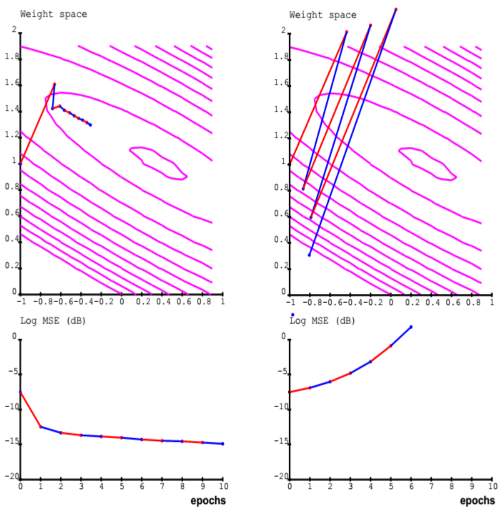
$$\sum_{t=1}^{\infty} \frac{1}{\lambda^2(t)} < \infty \text{ and } \sum_{t=1}^{\infty} \frac{1}{\lambda(t)} = \infty \,,$$

e.g. $\lambda(t) = \frac{c}{t}$;

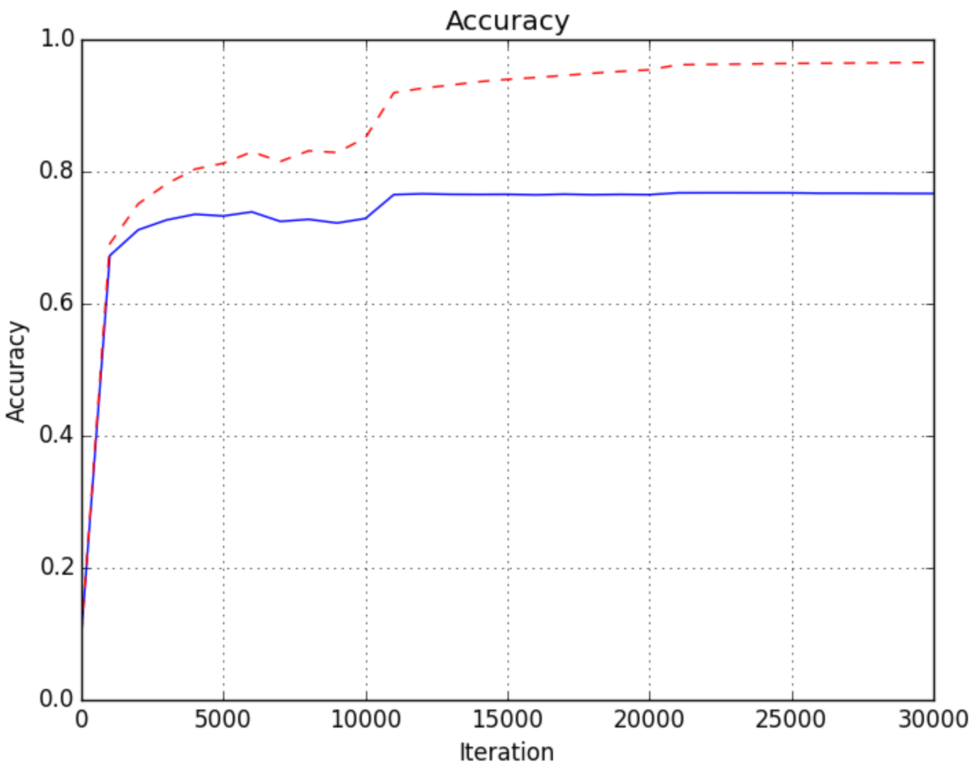
Other learning rate policies:

1. Manual: $\lambda = const$
2. exp: $\lambda_n = \lambda_0 * \gamma^n$
3. step : $\lambda_n = \lambda_0 * \gamma^{\left[\frac{n}{step}\right]}$
4. inverse: $\lambda_n = \lambda_0 * (1 + \gamma * n)^{-c}$





Example: CIFAR-10 training

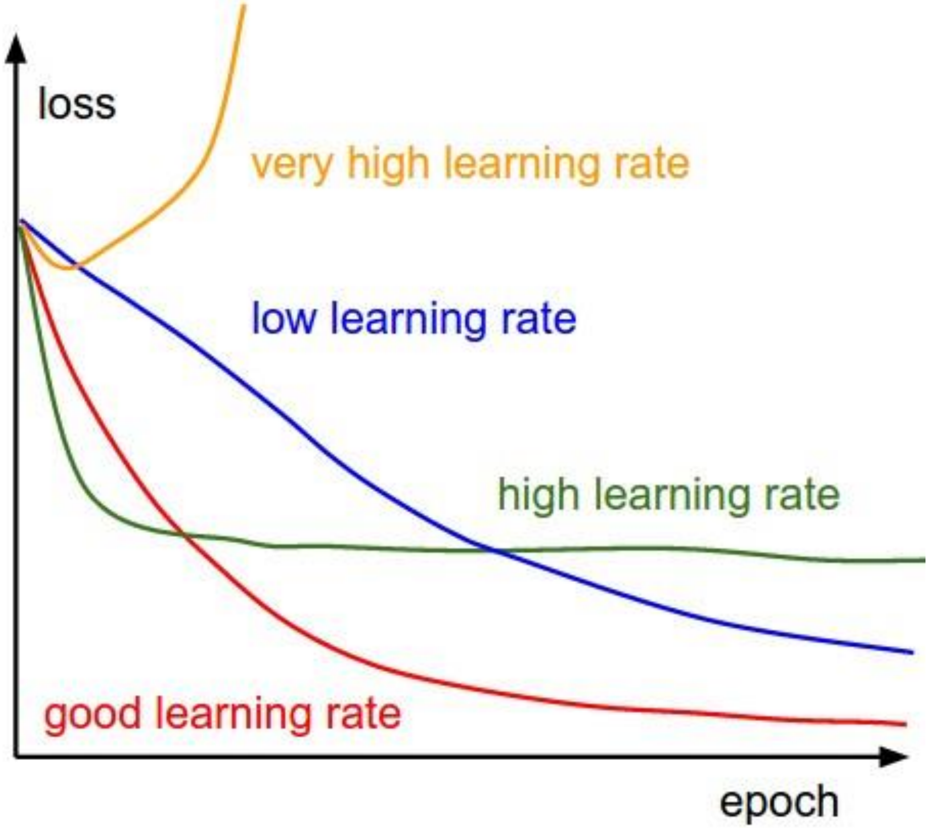


Initial lr = 0.001, decrease lr by 10x for 10,000 and 20,000 iterations

Learning Rate

No silver bullet for learning rate

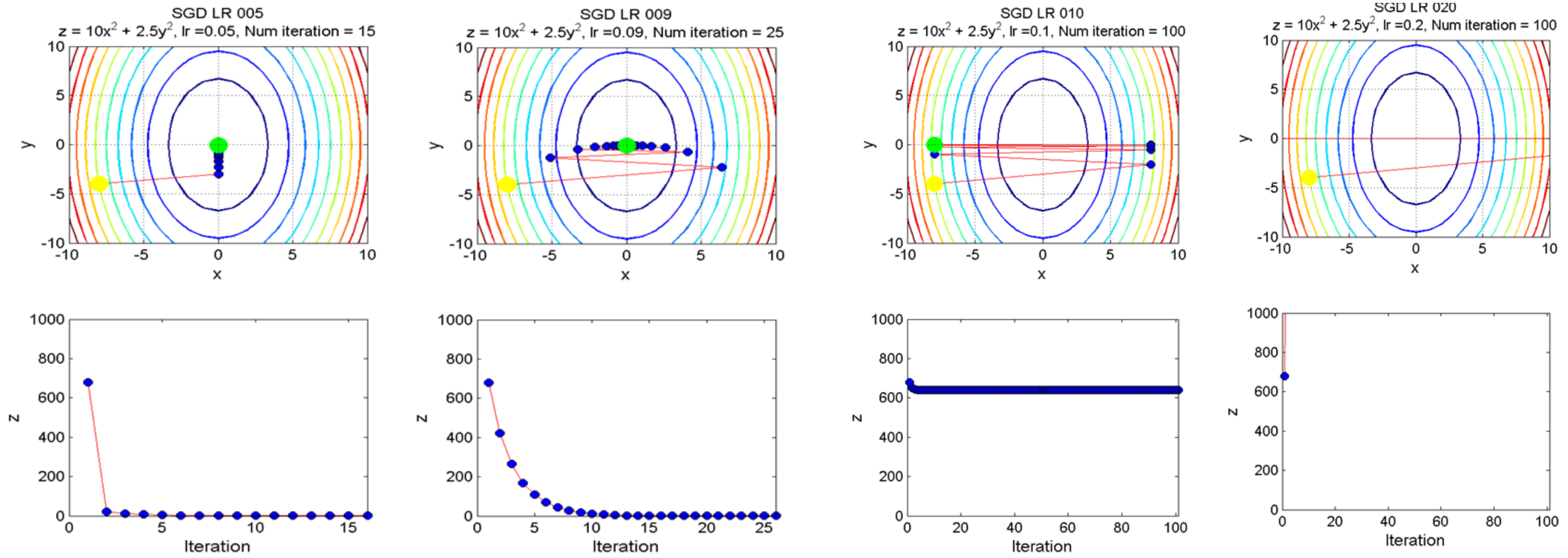
Choose proper size with explaration



Why accuracy improves when we decrease lr? Motivational example

Let's take a quadratic function $z = 10 * x^2 + 2.5 * y^2$ and use SGD to find minimum.

Basic SGD converges if learning rate $< \frac{1}{10}$.



lr=0.05

lr=0.09

lr=0.10

lr=0.20

Adaptive learning rate

Adapt

1. learning rate per weight or per layer
   - Use only the sign of the gradient
   - Divide the learning rate for a weight by a running average of the magnitudes of recent gradients for that weight (Adagrad, AdaDelta, RMSPROP,..)
2. learning rate and momentum
   - Natural gradient
   - ADAM (*Knigmna and BA)*

SGD with weight decay

$$W(t+1) = W(t) - \lambda * \left(\frac{\partial E}{\partial w} + \boldsymbol{\theta} * \boldsymbol{W(t)} \right)$$

Weight decay works to keep weights size under control ("regularization").

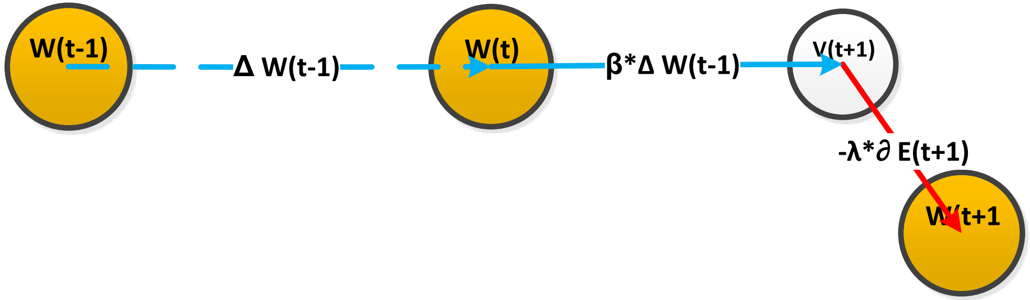
This is equivalent to adding penalty on weights to loss function:

$$E'(W) = E(W) + \frac{\theta}{2} * W^2$$

SGD with momentum

$$W(t+1) = W(t) + \Delta W(t+1)$$

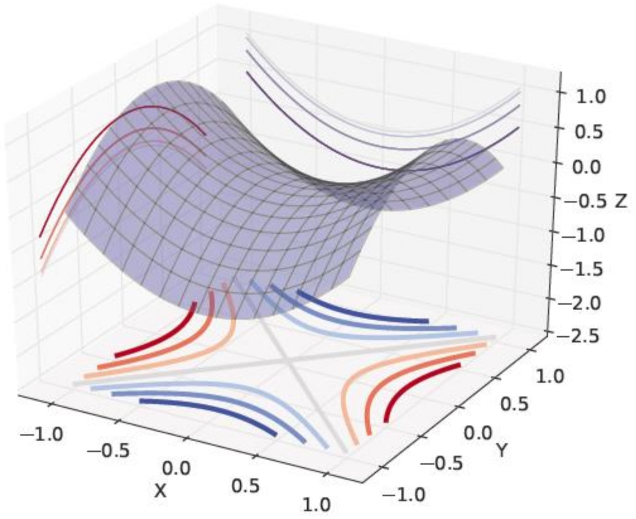
$$\boldsymbol{\Delta W(t+1) = \beta * \Delta W(t) - \lambda * \frac{\partial E}{\partial w}}$$

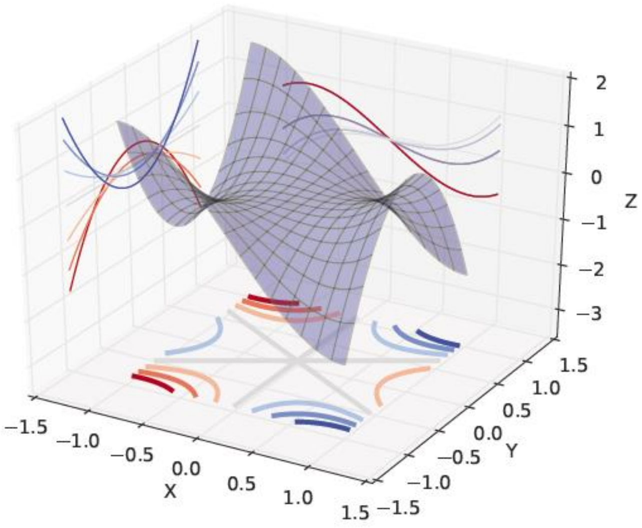


Momentum works as weighted average of gradients .

$$\Delta W(t+1) = -\lambda * \left(\sum_{k=1}^{t+1} \beta^{t+1-k} * \frac{\partial E}{\partial w}(k) \right)$$

Optimization near Saddle Points





“The problem with convnets cost functions is not local min, but local saddle points. How SGD methods behave near saddle point?”

R. Pascanu, “On the saddle point problem for non-convex optimization”, http://arxiv.org/abs/1405.4604

Dauphin, ”Identifying and attacking the saddle point problem in high-dimensional non-convex optimization”, http://arxiv.org/pdf/1406.2572v1.pdf

Advanced Optimization Algorithms

A lot of interesting optimization algorithms to speed-up training and get better results:

- Nesterov Accelerated Gradient
- Adagrad
- Adadelta
- RMSPROP/RPROP
- Varience-based SGD
- Averaged SGD
- ADAM

AdaGrad, AdaDelta, and ADAM

Adagrad: adapt learning rate for each weight

$$\Delta W_{ij}(t+1) = -\frac{\gamma}{\sqrt{\sum_1^{t+1}\left(\frac{\partial E}{\partial w_{ij}}(\tau)\right)^2}} * \frac{\partial E}{\partial w_{ij}}(t+1)$$

AdaDelta: accumulate the denominator over last k gradients (sliding window):

$$\alpha(t+1) = \sum_{t-k+1}^{t+1}\left(\frac{\partial E}{\partial w_{ij}}(\tau)\right)^2$$

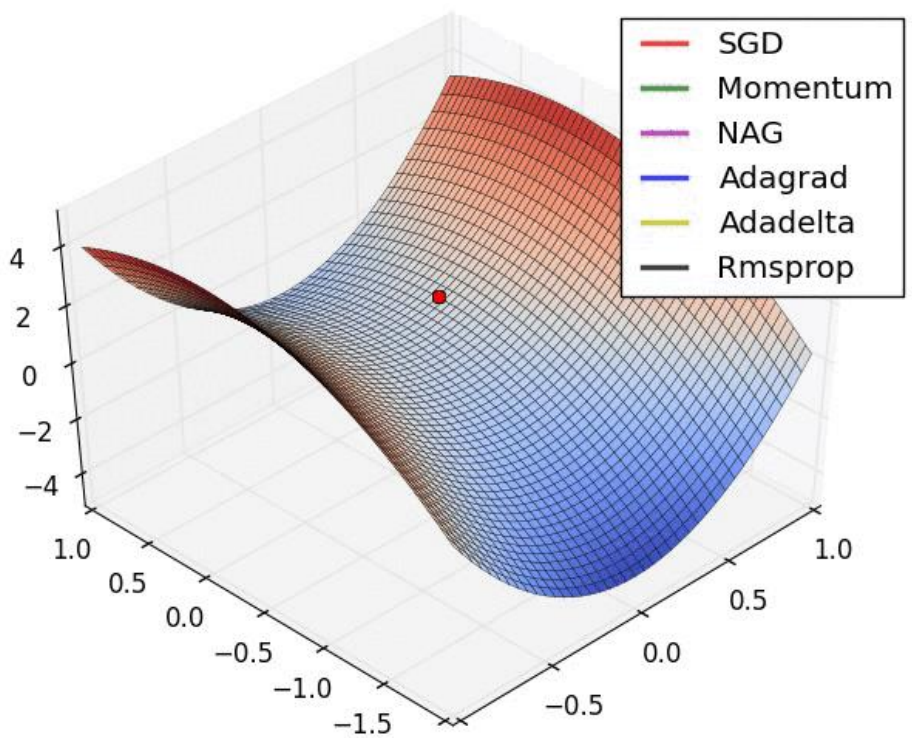
$$\Delta W_{ij}(t+1) = -\frac{\gamma}{\sqrt{\alpha(t+1)}} * \frac{\partial E}{\partial w_{ij}}(t+1)$$

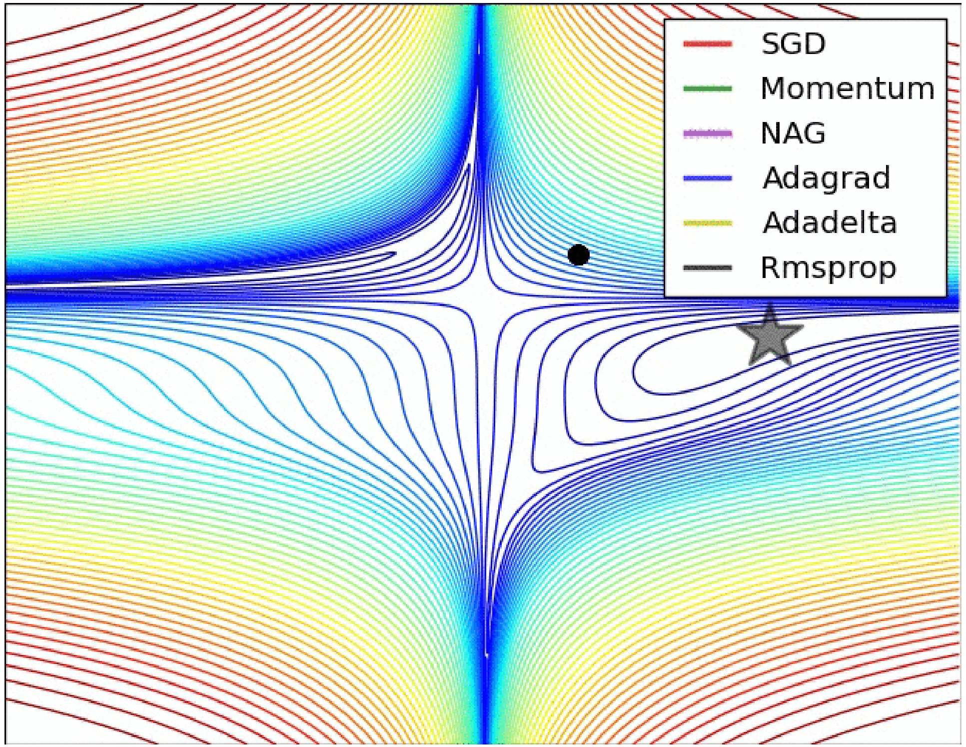
This requires to keep k gradients. Instead we can use simpler formula:

$$\beta(t+1) = \rho * \beta(t) + (1-\rho) * \left(\frac{\partial E}{\partial w_{ij}}(t+1)\right)^2$$

$$\Delta W_{ij}(t+1) = -\frac{\gamma}{\sqrt{\beta(t+1)+\epsilon}} * \frac{\partial E}{\partial w_{ij}}(t+1)$$

Performance near Saddle Points

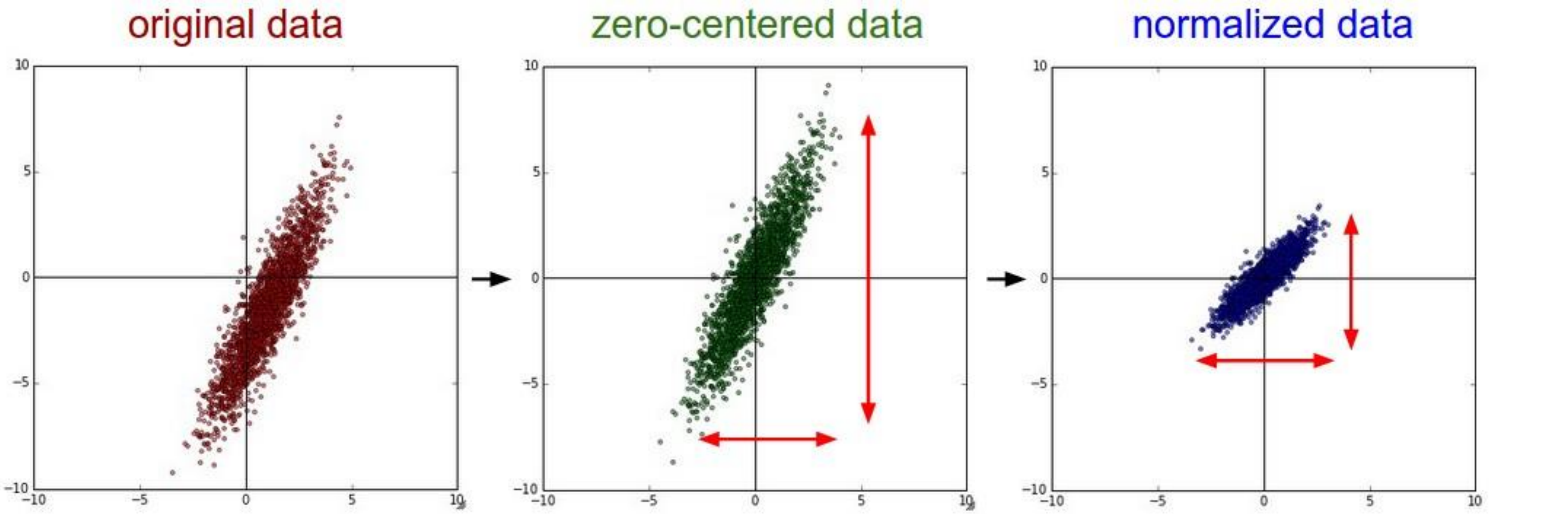




by Alec Radford alecradford http://imgur.com/a/Hqolp

Data Preprocessing

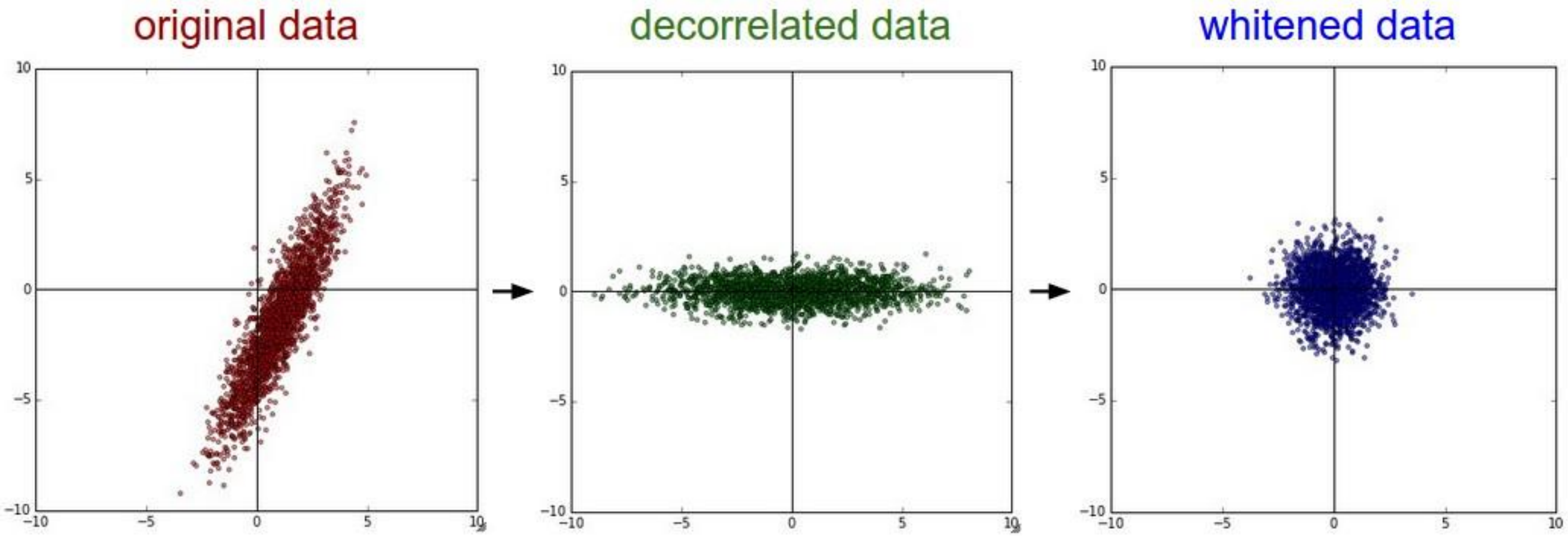
Preprocessing the data




```
X -= np.mean(X, axis = 0)
X /= np.std(X, axis = 0)
```

Dimension Reduction

In practice, you may also see **PCA** and **Whitening** of the data



(data has diagonal covariance matrix)

(covariance matrix is the identity matrix)

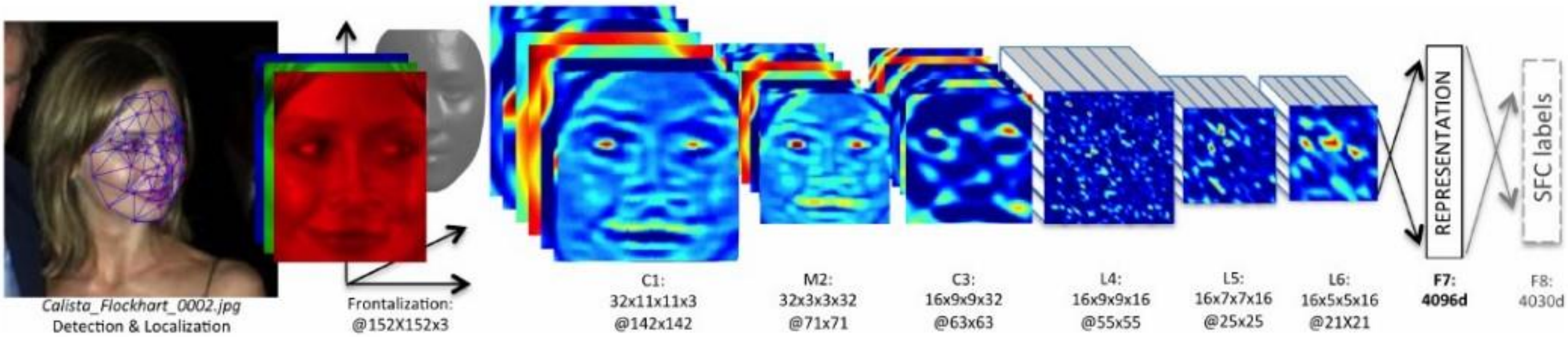
TLDR: In practice for Images: center only

e.g. consider CIFAR-10 example with [32,32,3] images

- Subtract the mean image (e.g. AlexNet)
  (mean image = [32,32,3] array)
- Subtract per-channel mean (e.g. VGGNet)
  (mean along each channel = 3 numbers)

Not common to normalize variance, to do PCA or whitening

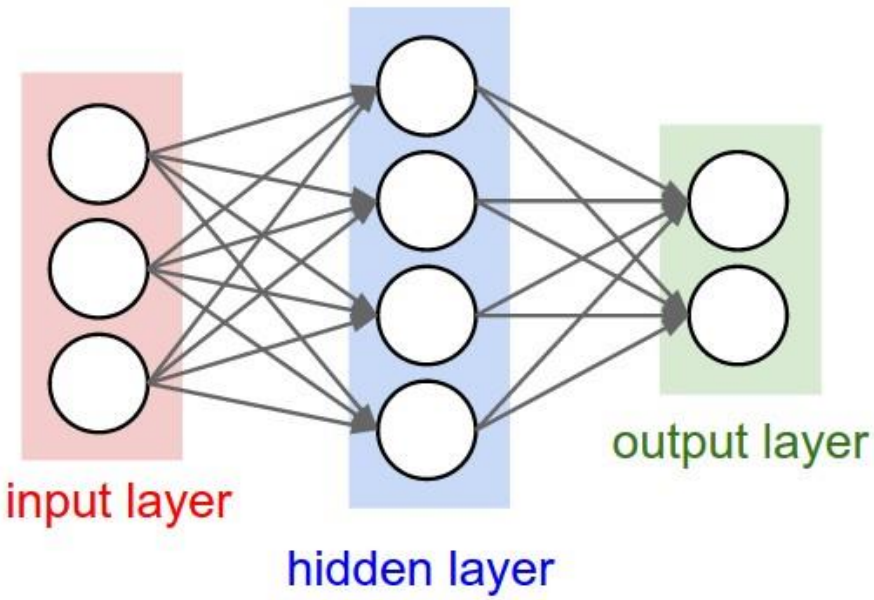
Case: DeepFace Architecture



Yaniv Taigman, etc (Facebook) . DeepFace: Closing the Gap to Human-Level Performance in Face Verification, CVPR 2014

Weight Initialization

Initialization for Neural Network

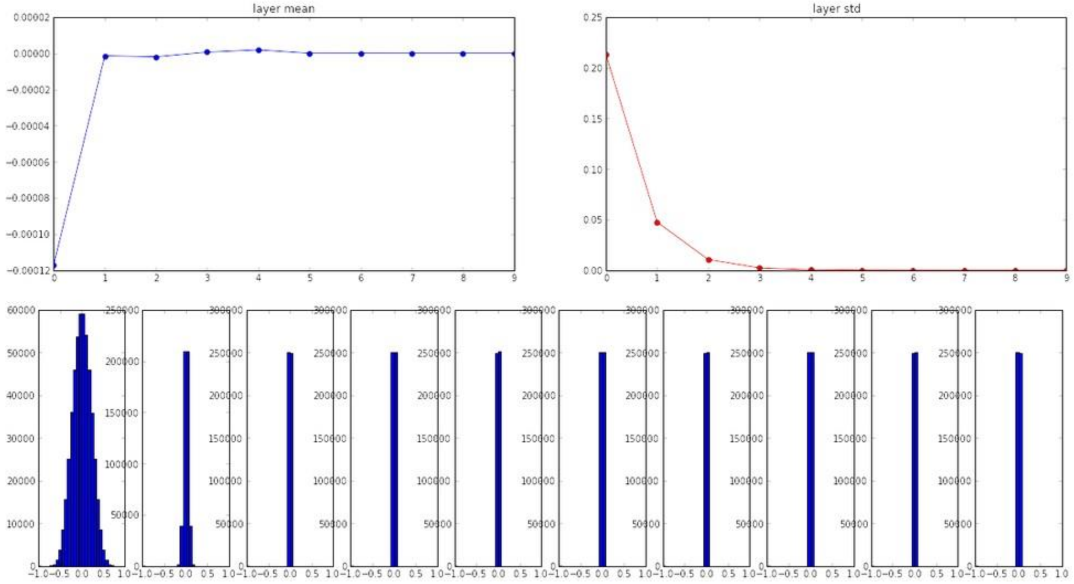



```
W = 0.01* np.random.randn(D,H)
```

Gradient banishment

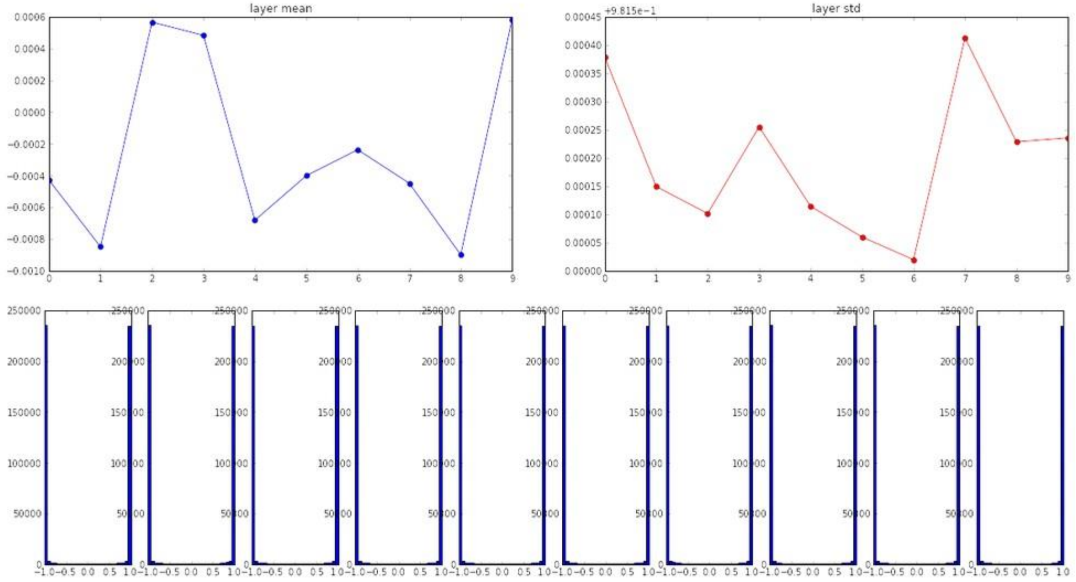
```
W = 0.01* np.random.randn(D,H)
```

```
input layer had mean 0.000927 and std 0.998388
hidden layer 1 had mean -0.000117 and std 0.213081
hidden layer 2 had mean -0.000001 and std 0.047551
hidden layer 3 had mean -0.000002 and std 0.010630
hidden layer 4 had mean 0.000001 and std 0.002378
hidden layer 5 had mean 0.000002 and std 0.000532
hidden layer 6 had mean -0.000000 and std 0.000119
hidden layer 7 had mean 0.000000 and std 0.000026
hidden layer 8 had mean -0.000000 and std 0.000006
hidden layer 9 had mean 0.000000 and std 0.000001
hidden layer 10 had mean -0.000000 and std 0.000000
```




```
W = np.random.randn(fan_in, fan_out) * 1.0 # layer initialization
```

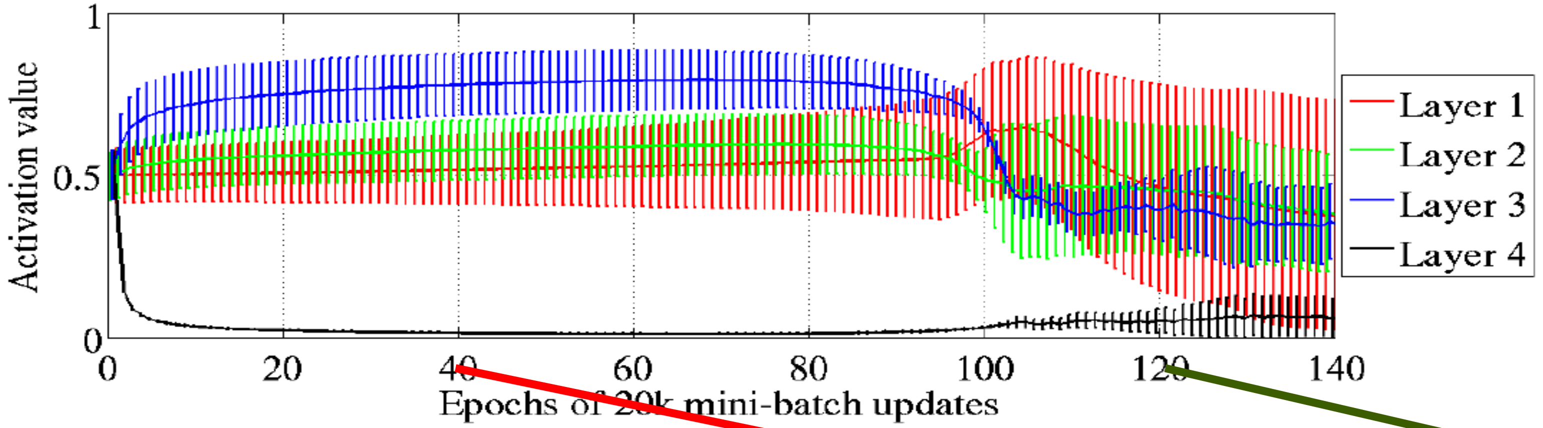
```
input layer had mean 0.001800 and std 1.001311
hidden layer 1 had mean -0.000430 and std 0.981879
hidden layer 2 had mean -0.000849 and std 0.981649
hidden layer 3 had mean 0.000566 and std 0.981601
hidden layer 4 had mean 0.000483 and std 0.981755
hidden layer 5 had mean -0.000682 and std 0.981614
hidden layer 6 had mean -0.000401 and std 0.981560
hidden layer 7 had mean -0.000237 and std 0.981520
hidden layer 8 had mean -0.000448 and std 0.981913
hidden layer 9 had mean -0.000899 and std 0.981728
hidden layer 10 had mean 0.000584 and std 0.981736
```

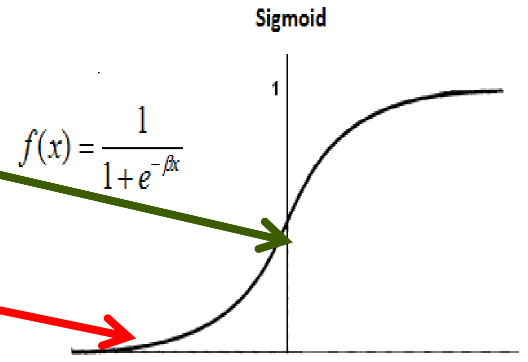


Understanding the difficulty of training convolutional networks

Example: MLP with 4 fully connected layers, with sigmoid non-linear function.

Measure mean and standard deviation of the activation (output of the sigmoid) for 4 hidden layers

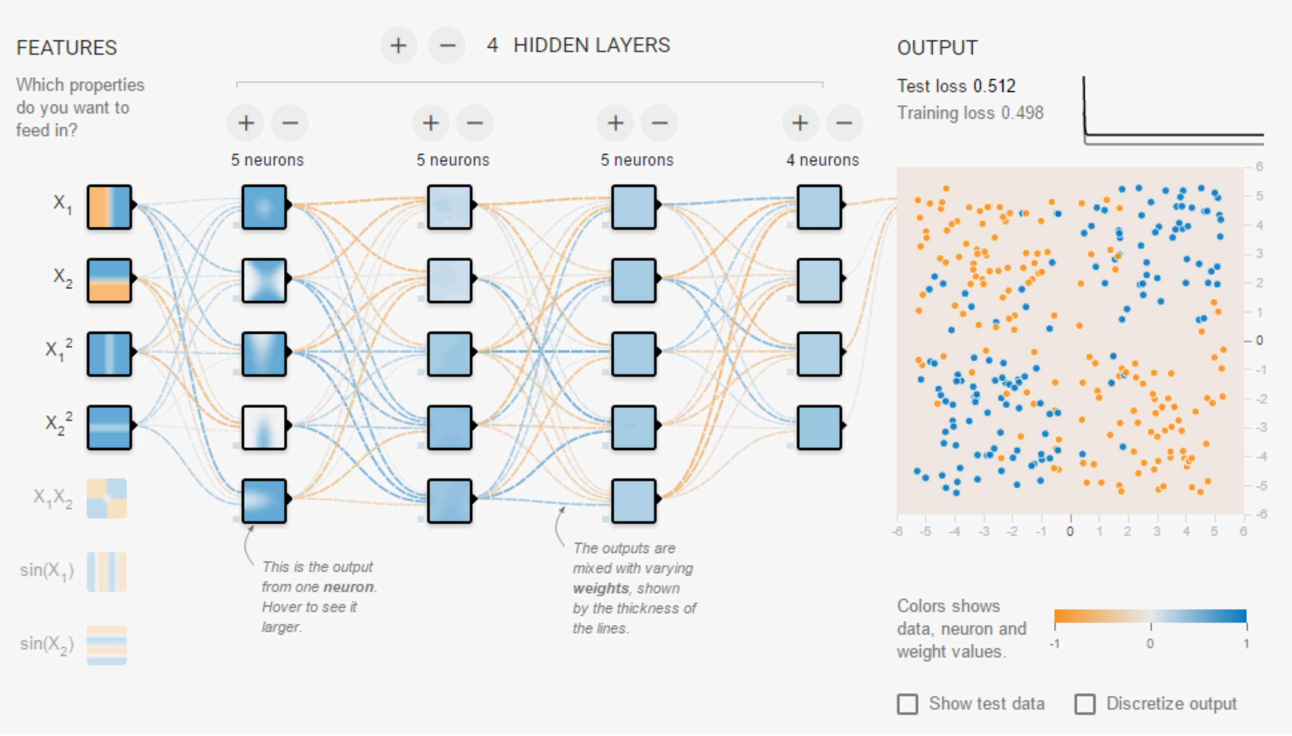


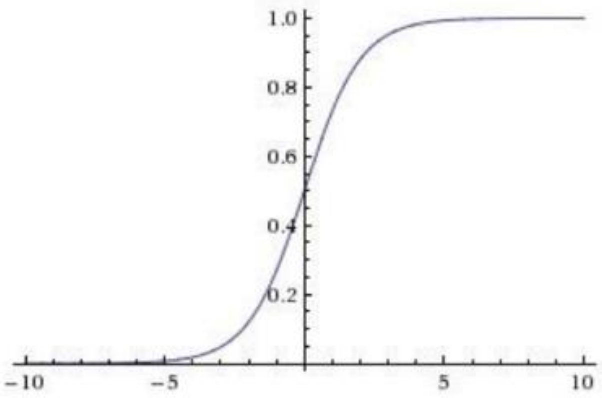


The top hidden layer quickly saturates at 0. It drives the grdaient dE/dy multiplied by 0 and therefore causing the gradient descent to stall, slowing down all learning. Near epoch 100 it slowly de-saturates releasing other layers

X. Glorot ,Y. Bengio, *Understanding the difficulty of training deep feedforward neural networks*

Why?





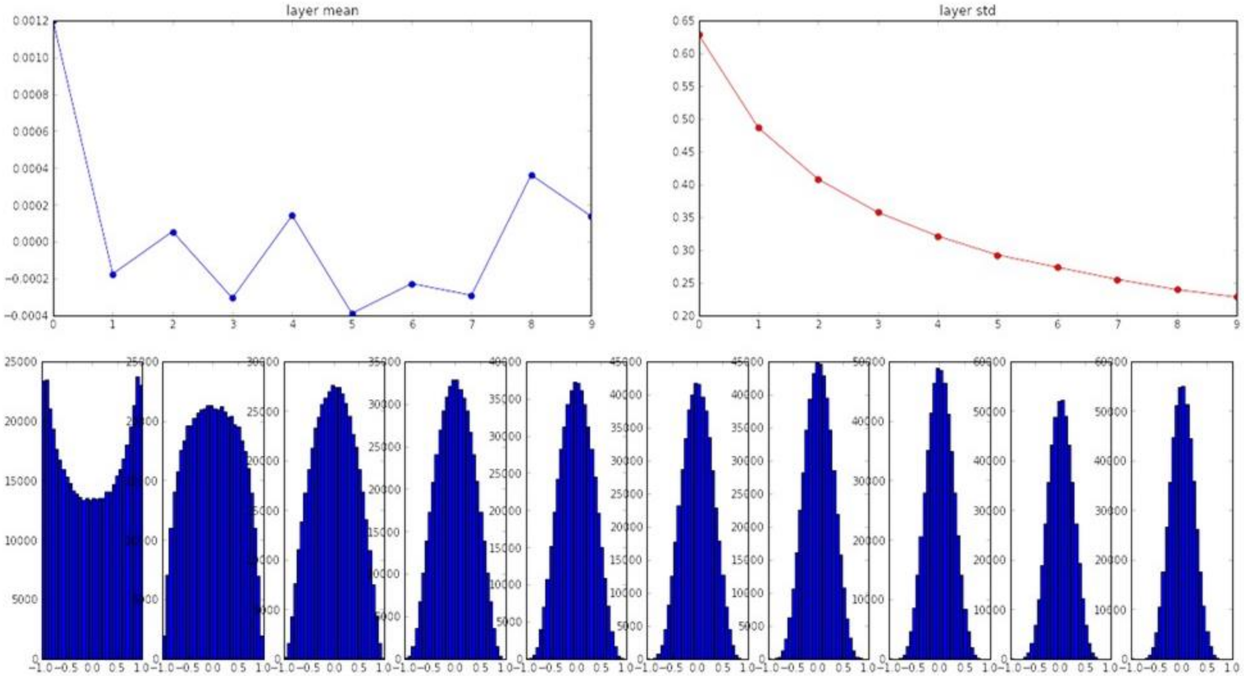
Sigmoid

All neurons will be all zero during back-propagation

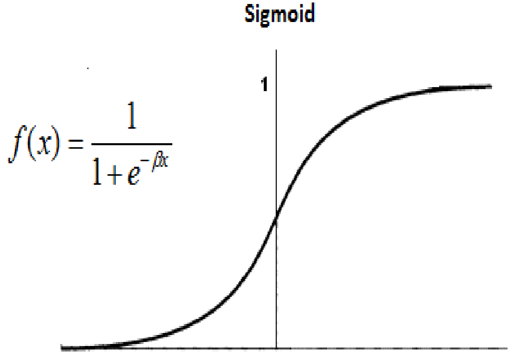
playground.tensorflow.org

Xavier initialization

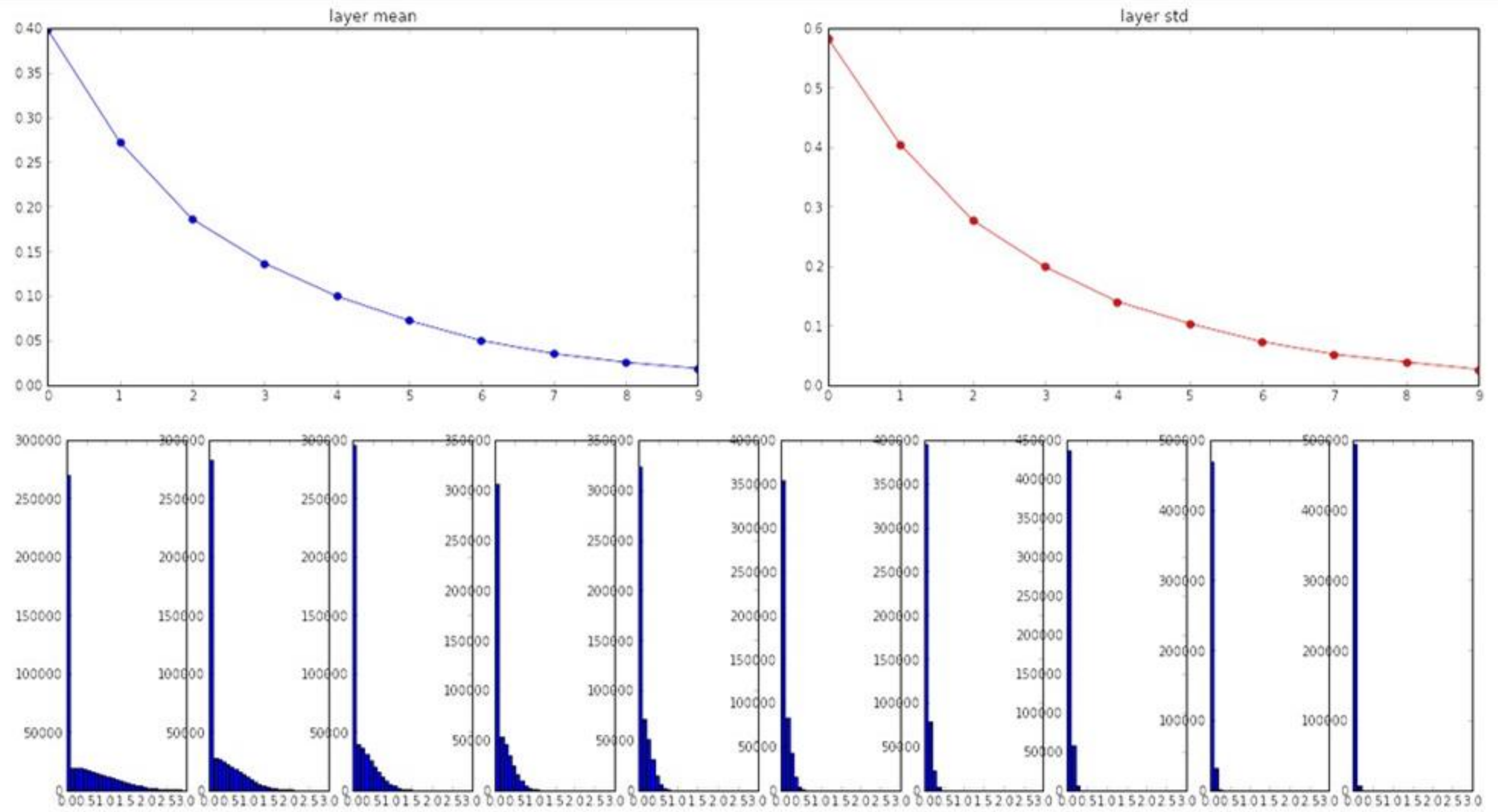
```
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in) # layer initialization
```



Reasonable initialization.
(Mathematical derivation assumes linear activations)

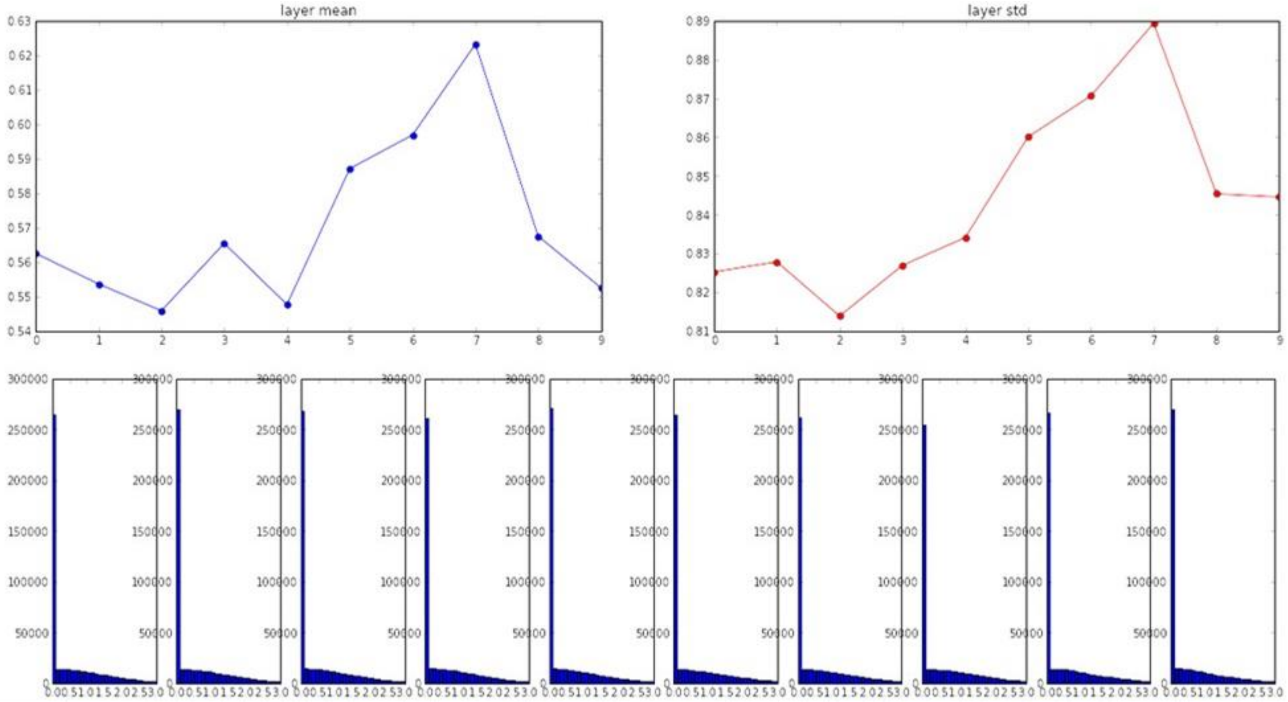


Xavier Initialization with ReLU



He initialization

```
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in/2) # layer initialization
```



Regularization | **Data Augmentation**

Neural Network Regularization

- Empirical
  - Dropout
  - DropConnect
  - Stochastic pooling
  - Artificial Data
- Explicit
  - Early Stopping
  - Limiting Number of Parameters
  - Weight Decay
    - L1/L2 regularization
  - Max norm constraints

https://en.wikipedia.org/wiki/Regularization_(mathematics)

Data Augmentation

The common method to “enlarge” training set:

- image translations
- re-scale (both up and down) before crop
- horizontal and vertical reflections (“flip”)
- elastic deformation with random interpolations ((bilinear, area, nearest neighbor and cubic, with equal probability) (*Simard, 2003*)
- photometric distortion and altering the intensities of the RGB channels in training images (A.G. Howard., 2013)

Data Augmentation (1)

1. Horizontal Filps

2. Random crops/scales

Training: sample random crops / scales

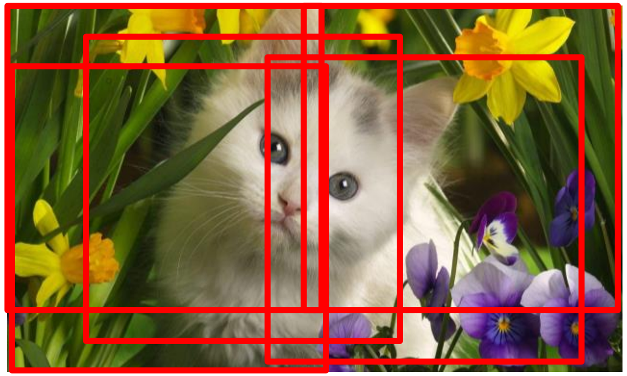
ResNet:

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

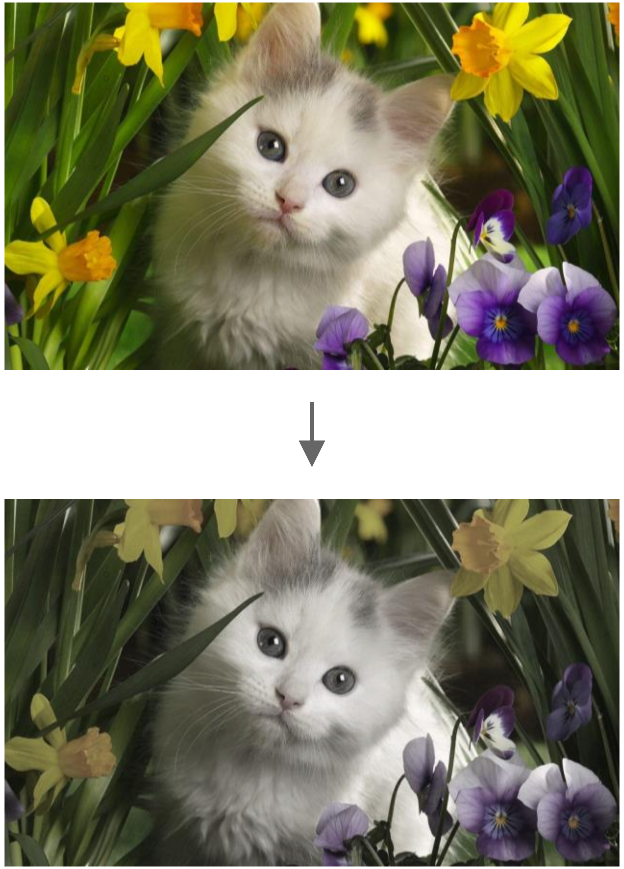
ResNet:

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

3. Color Jitter



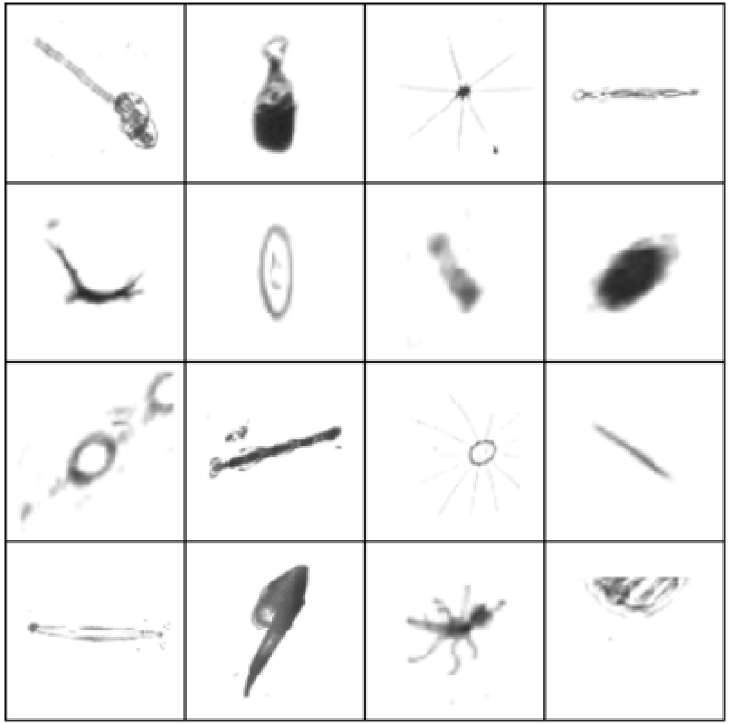
4. Others

Random mix/combinations of :

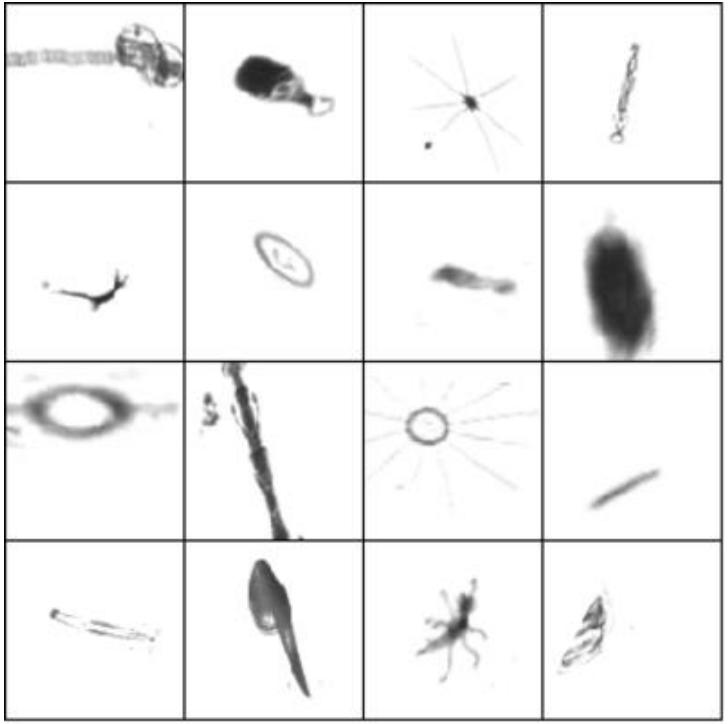
- translation
- rotation
- stretching
- shearing,
- lens distortions, … (go crazy)

Data augmentation (Plankton competition)

Pre-processed image



Augmented image

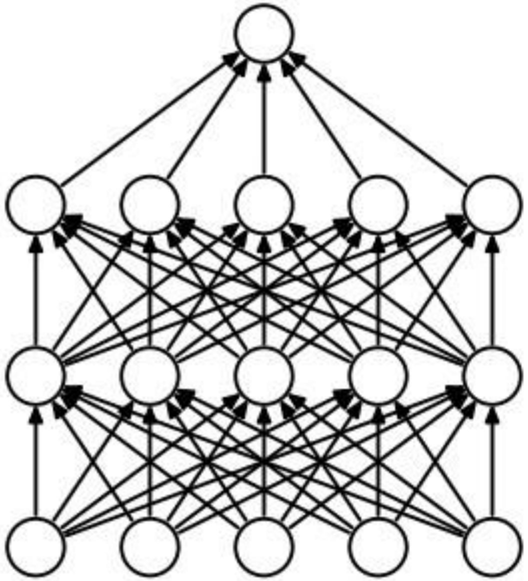


http://benanne.github.io/2015/03/17/plankton.html

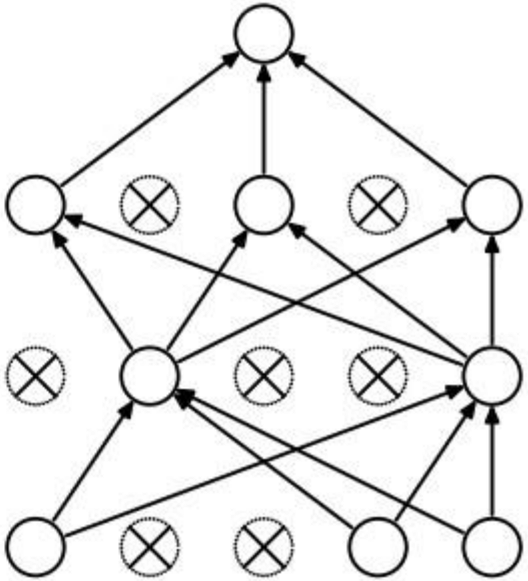
Regularization **Dropout**

Dropout Layer

randomly set some neurons to zero in the forward pass



(a) Standard Neural Net



(b) After applying dropout.

Srivastava et al., 2014

Code example

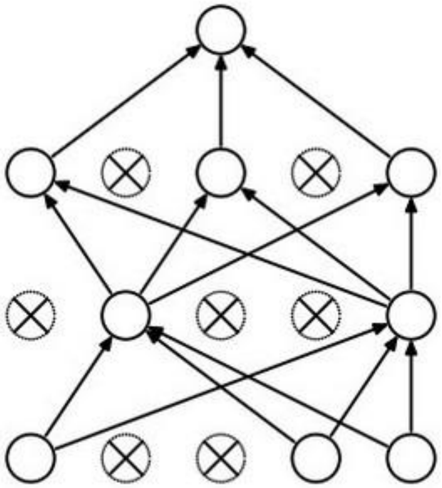
```
p = 0.5 # probability of keeping a unit active. higher = less dropout

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)
```

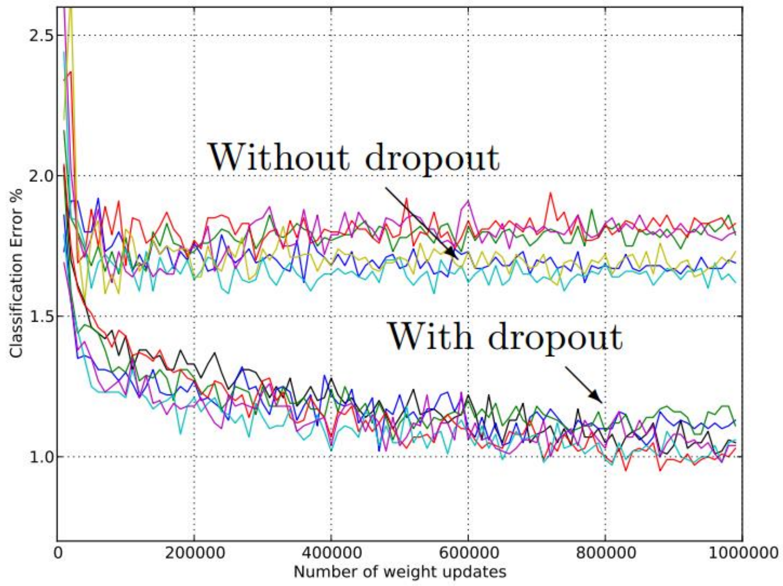
Example forward pass with a 3-layer network using dropout

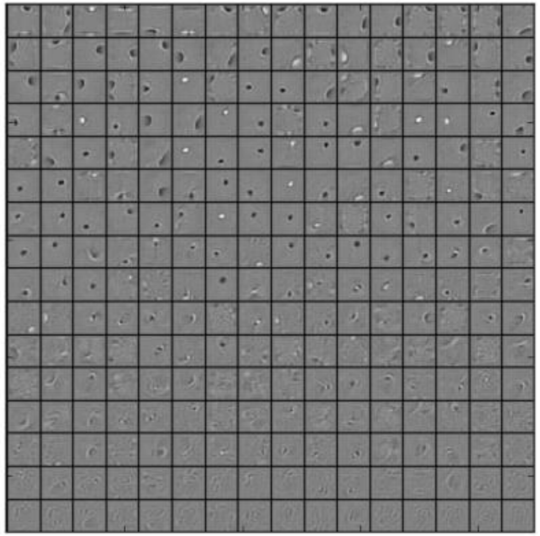


cs231n

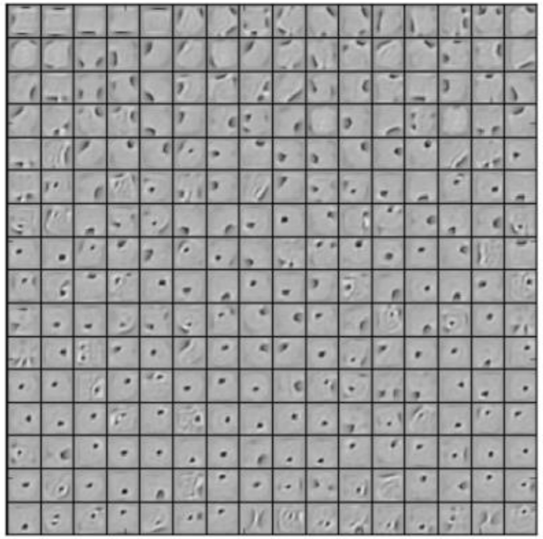
Why this is good?

It helps higher accuracy



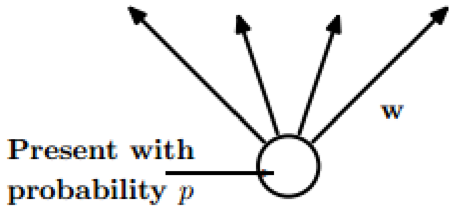


(a) Without dropout



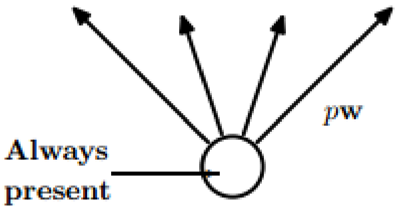
(b) Dropout with $p = 0.5$.

Using for Training / Inference



(a) At training time

```
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
```



(b) At test time

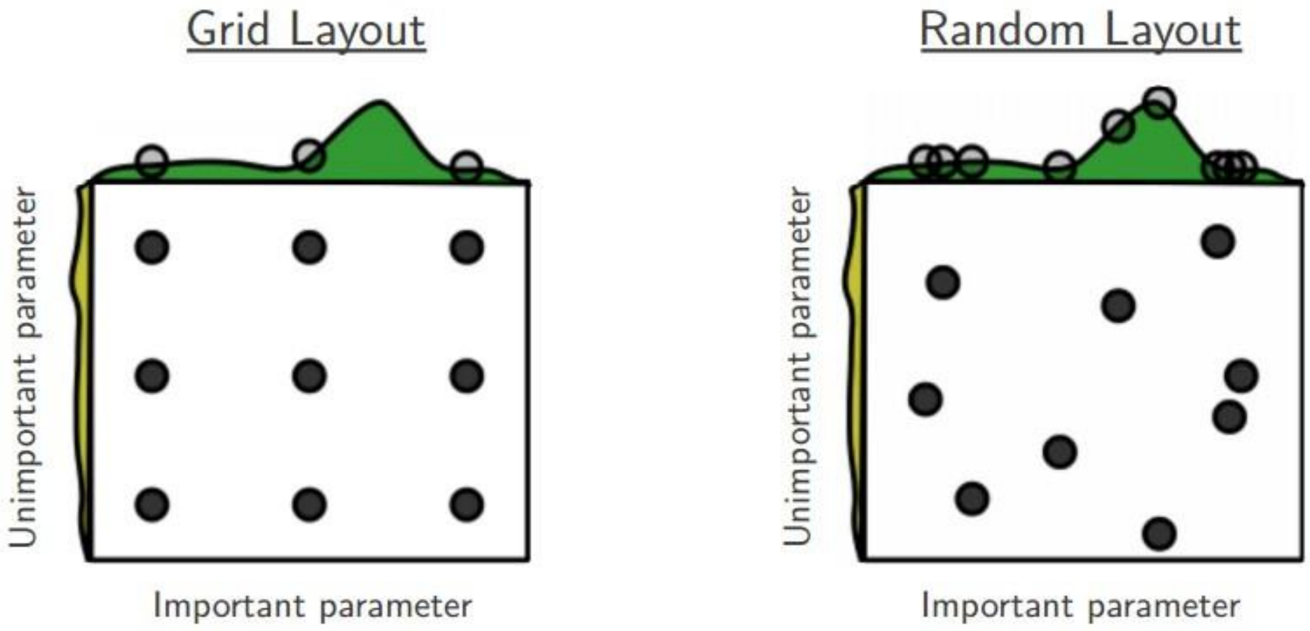
```
def predict(X):
  # ensembled forward pass
  H1 = np.maximum(0, np.dot(W1, X) + b1) # no scaling necessary
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  out = np.dot(W3, H2) + b3
```

Hyperparameter Optimizations

Parameters affecting training

Learning rate
Regularizations
Filter size
Model depth
Stride
Dropout
Model architecture
…

Random Search vs. Grid Search



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

Search optimal hyper-parameters

example

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

adjust range →

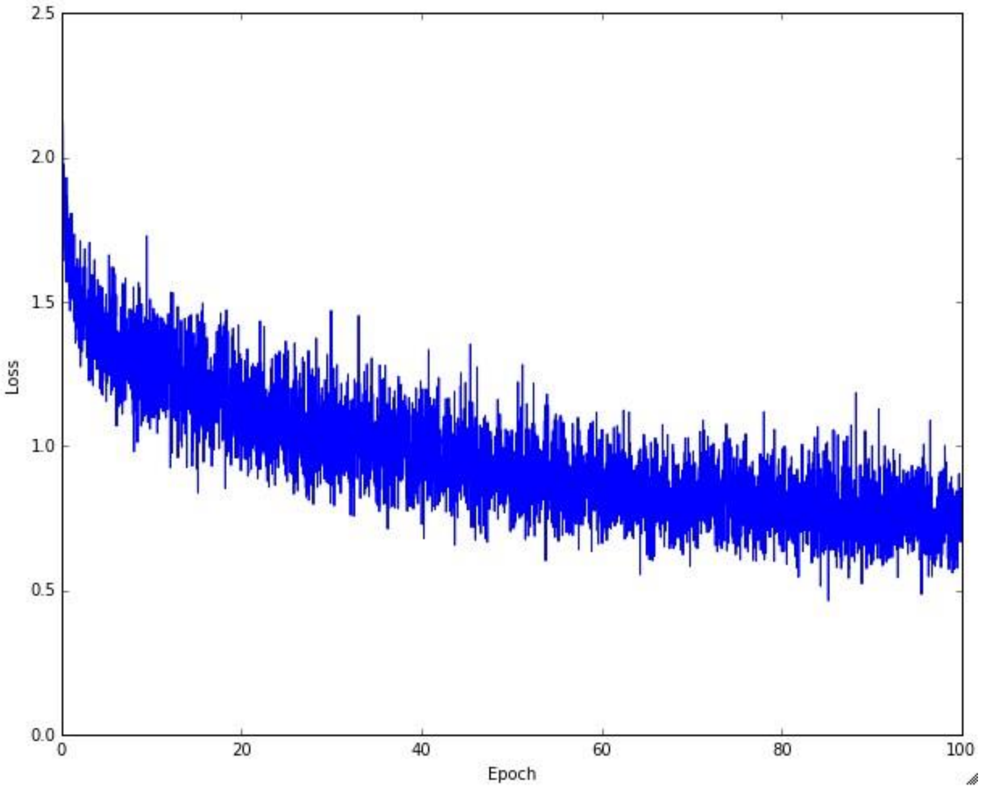
```
max_count = 100
for count in xrange(max_count):
      reg = 10**uniform(-4, 0)
      lr = 10**uniform(-3, -4)
```

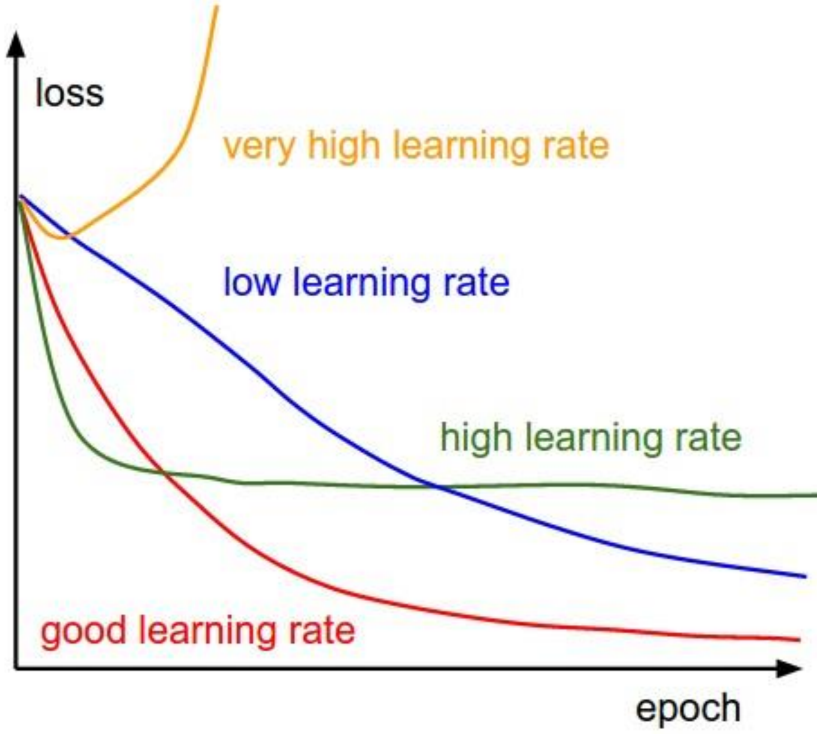
```
val_acc: 0.527000, lr: 5.340517e-04, reg: 4.097824e-01, (0 / 100)
val_acc: 0.492000, lr: 2.279484e-04, reg: 9.991345e-04, (1 / 100)
val_acc: 0.512000, lr: 8.680827e-04, reg: 1.349727e-02, (2 / 100)
val_acc: 0.461000, lr: 1.028377e-04, reg: 1.220193e-02, (3 / 100)
val_acc: 0.460000, lr: 1.113730e-04, 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)
```

53% - relatively good for a 2-layer neural net with 50 hidden neurons.

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

Monitor and visualize the loss curve





NVIDIA