CS60010: Deep Learning

Sudeshna Sarkar

Spring 2018

22 Jan 2018

TODO

Look at:

- Parameter update schemes
- Learning rate schedules
- Gradient Checking
- Regularization (Dropout etc)
- Evaluation (Ensembles etc)

Batch Gradient Descent

Algorithm 1 Batch Gradient Descent at Iteration k

Require: Learning rate ϵ_k

Require: Initial Parameter θ

1: **while** stopping criteria not met **do**

2: Compute gradient estimate over N examples:

$$\hat{g} \leftarrow +\frac{1}{N}\nabla_\theta \sum_i L\left(f\left(x^{(i)};\theta\right), y^{(i)}\right)$$

4: Apply Update: $\theta = \theta - \epsilon\hat{g}$

5: **end while**

Positive: Gradient estimates are stable

Negative: Need to compute gradients over the entire training for one update

Stochastic Gradient Descent

Algorithm 2 Stochastic Gradient Descent at Iteration k

Require: Learning rate ϵ_k

Require: Initial Parameter θ

1. while stopping criteria not met do
2. Sample example $(x^{(i)}, y^{(i)})$ from training set
3. Compute gradient estimate:
4. $\hat{g} \leftarrow +\nabla_\theta \sum_i L\left(f(x^{(i)}; \theta), y^{(i)}\right)$
5. Apply Update: $\theta = \theta - \epsilon\hat{g}$
6. end while

ϵ_k is learning rate at step k

Sufficient condition to guarantee convergence:

$$\sum_{k=1}^{\infty} \epsilon_k = \infty \text{ and} \sum_{k=1}^{\infty} {\epsilon_k}^2 = \infty$$

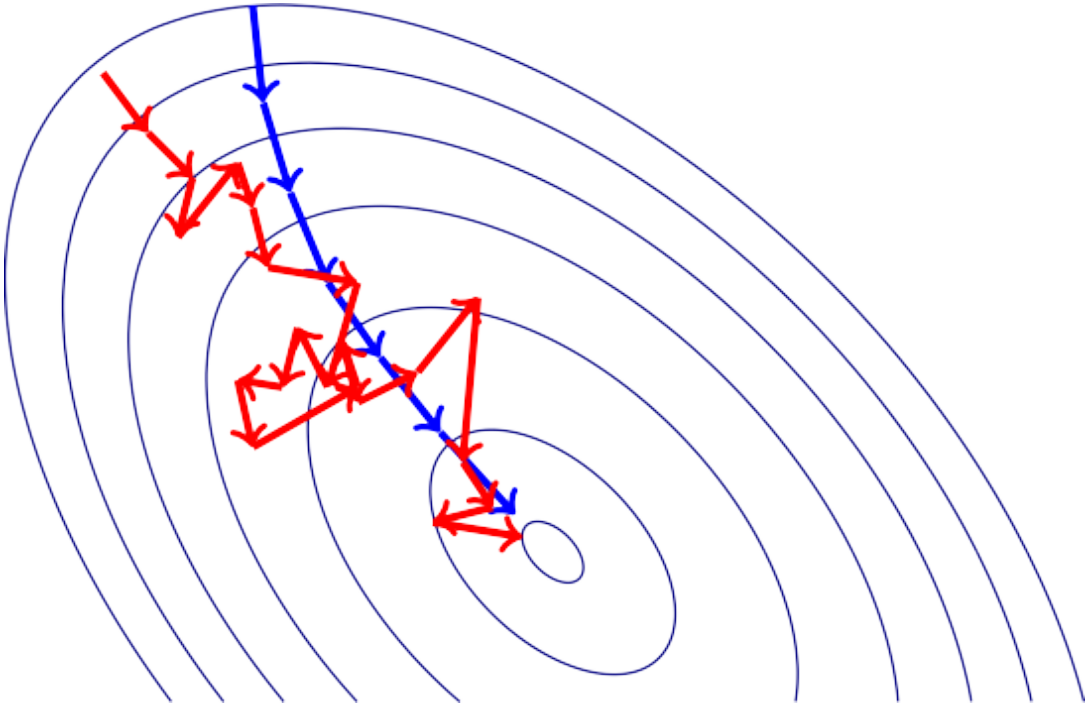
Learning Rate Schedule

- In practice the learning rate is decayed linearly till iteration τ

$$\epsilon_k = (1 - \alpha)\epsilon_0 + \alpha\epsilon_\tau \text{ with } \alpha = \frac{k}{\tau}$$

- τ is usually set to the number of iterations needed for a large number of passes through the data
- ϵ_τ should roughly be set to 1% of ϵ_0
- How to set ϵ_0 ?

Stochastic Gradient Descent



NN TRAINING

Overview

1. **One time setup**
   activation functions, preprocessing, weight initialization, regularization, gradient checking

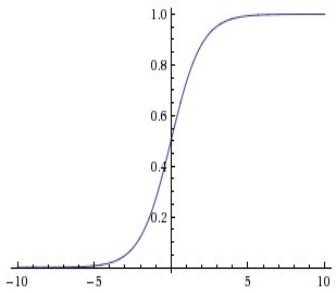
2. **Training dynamics**
   Training process, parameter updates, hyperparameter optimization

3. **Evaluation**
   model ensembles

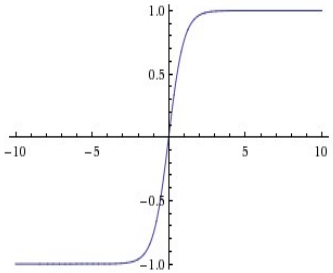
Activation Functions

Sigmoid

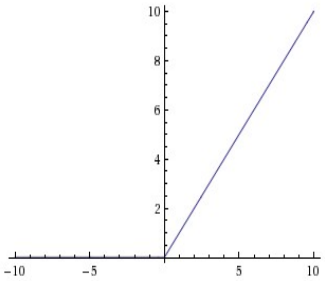
$$\sigma(x) = 1/(1+e^{-x})$$



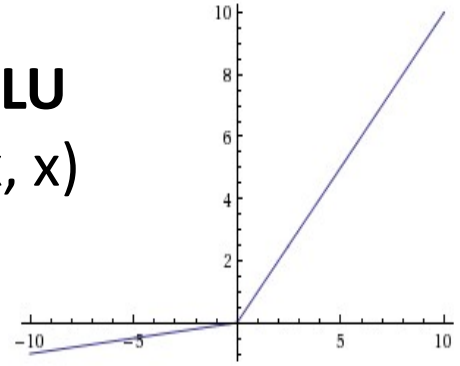
tanh tanh(x)



ReLU max(0,x)



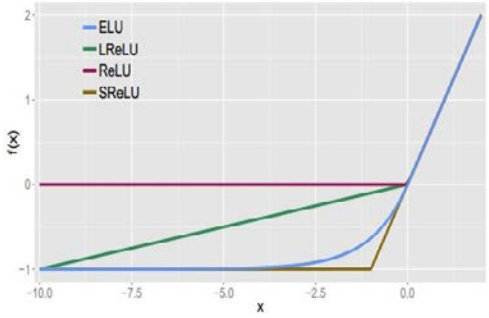
Leaky ReLU
max(0.1x, x)



Maxout $\max(w_1^T x + b_1, w_2^T x + b_2)$

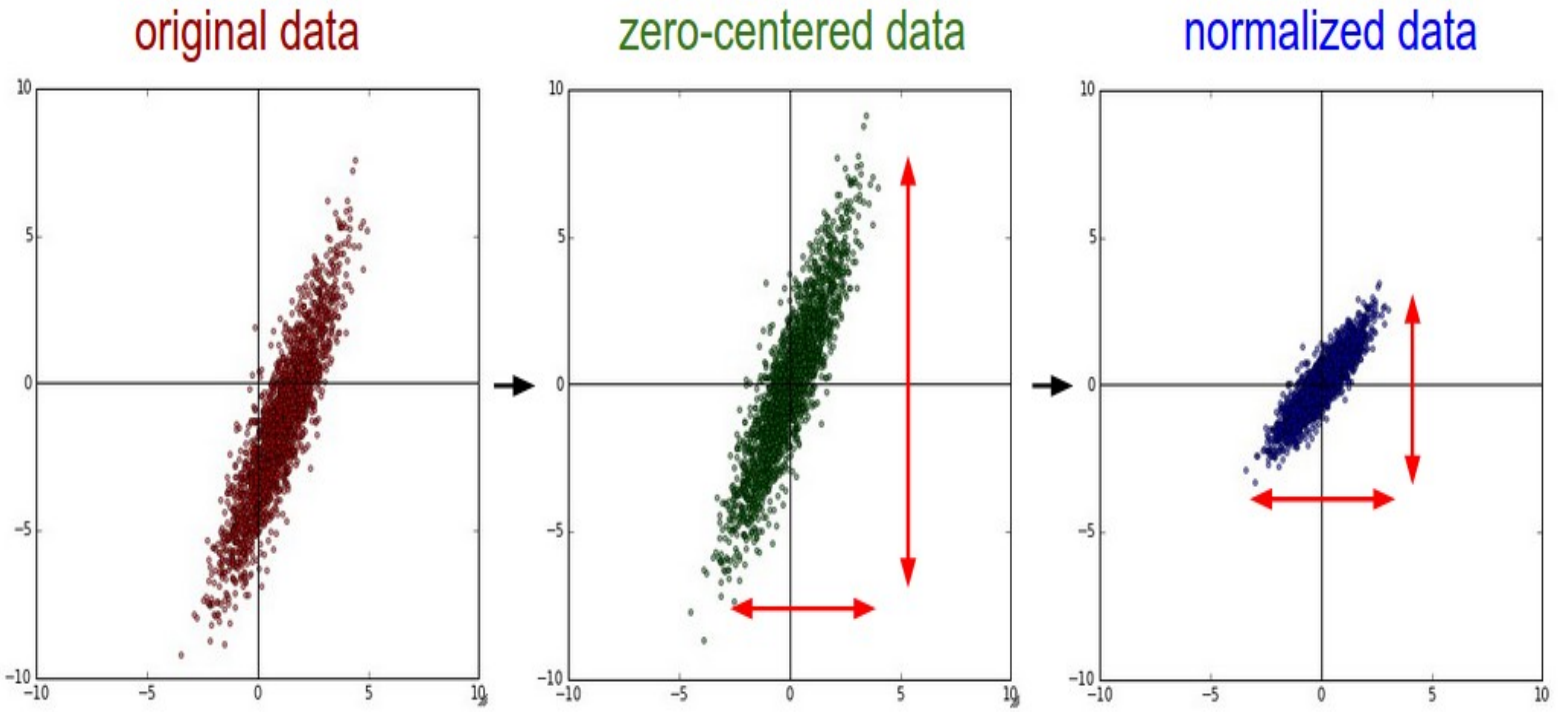
ELU

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha\,(\exp(x) - 1) & \text{if } x \leq 0 \end{cases}$$



Data Preprocessing

Step 1: Preprocess the data

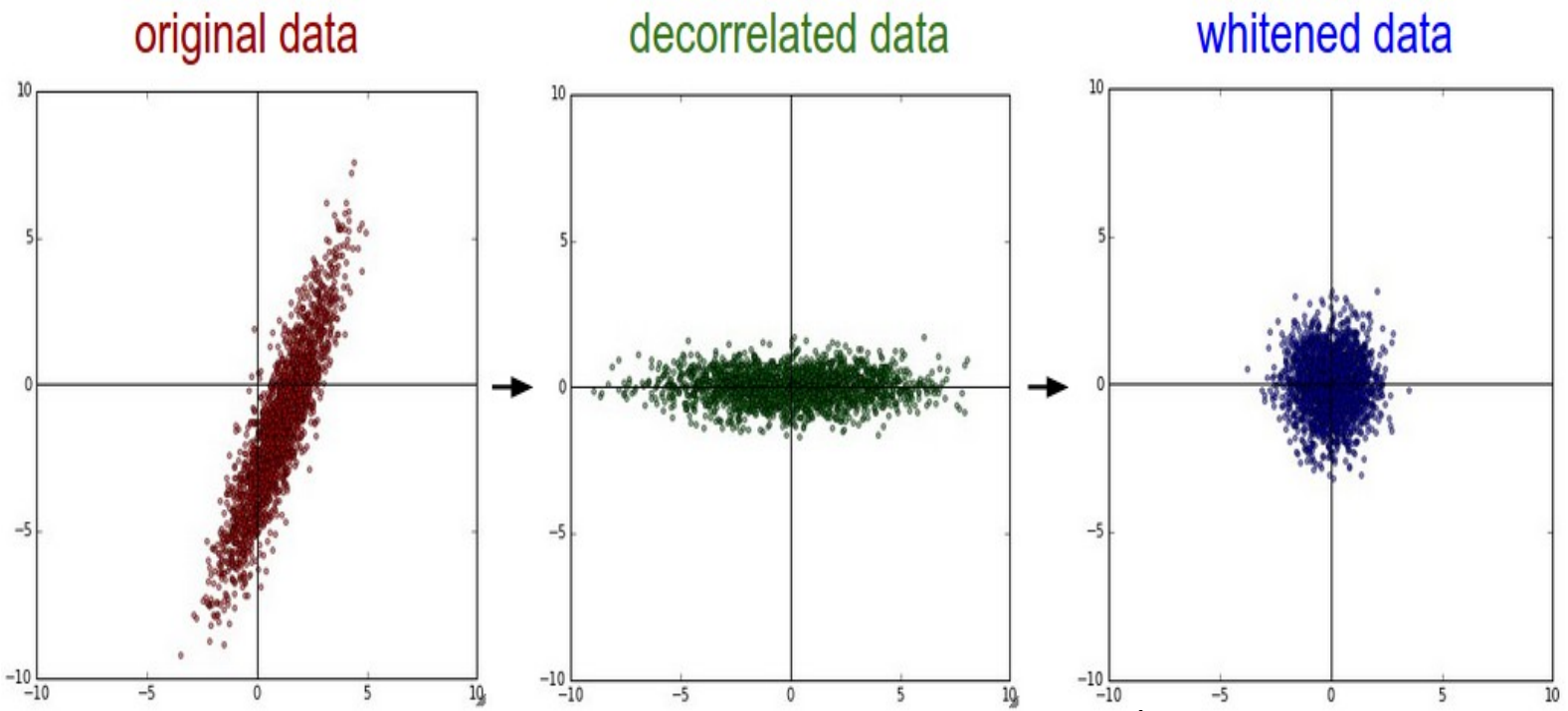


(Assume X [NxD] is data matrix,
each example in a row)

Step 1: Preprocess the data

PCA and Whitening:

1. After centring the data compute the covariance matrix. $cov = \frac{np.dot(X.T,X)}{X.shape[0]}$
2. Compute the SVD factorization of the data covariance matrix: $U, S, V = np.linalg.svd(cov)$
3. De-correlate the data: project the original zero-centered data into the eigenbasis: $Xrot = np.dot(X, U)$



(data has diagonal covariance matrix)

(covariance matrix is the identity matrix)

TLDR: In practice for Images: center only

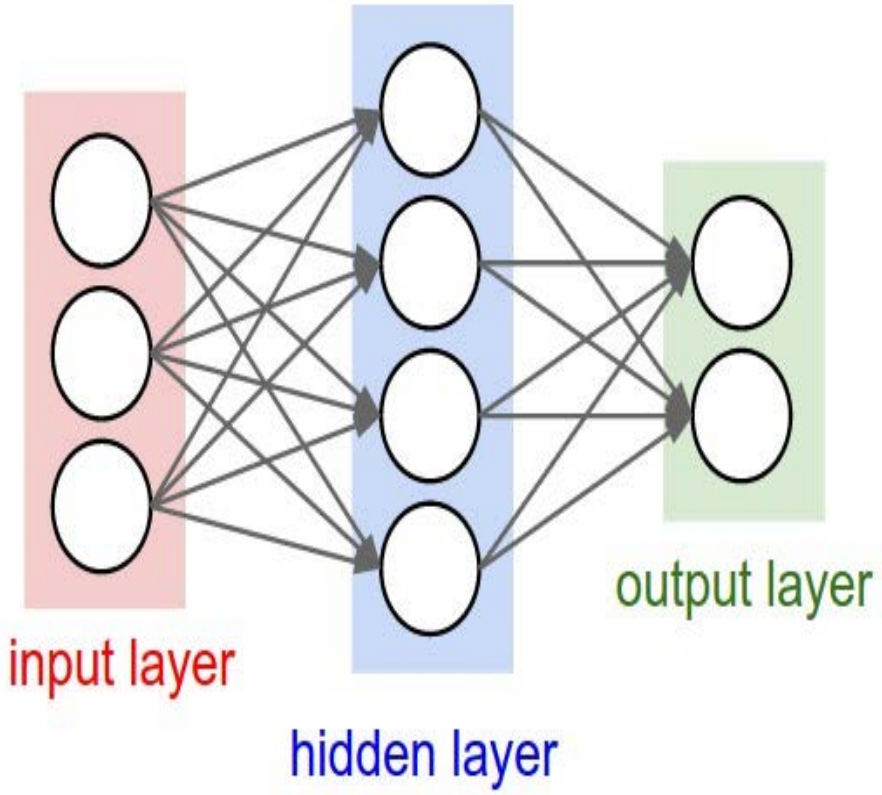
e.g. consider CIFAR-10 example with [32,32,3] images

- 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

Weight Initialization

- Q: what happens when W=0 init is used?



if every neuron in the network computes the same output, they will also all compute the same gradients during backpropagation and undergo the exact same parameter updates. In other words, there is no source of asymmetry between neurons.

Weight Initialization

Small random numbers
(gaussian with zero mean and 1e-2 standard deviation)

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

$$w_{ij} = \mathcal{N}(\mu, \sigma)$$

Works ~okay for small networks, but can lead to non-homogeneous distributions of activations across the layers of a network.

"Xavier initialization" [Glorot et al., 2010]

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

$$\sigma = \frac{1}{\sqrt{n_{\text{in}}}}$$

Batch Normalization

[Ioffe and Szegedy, 2015]

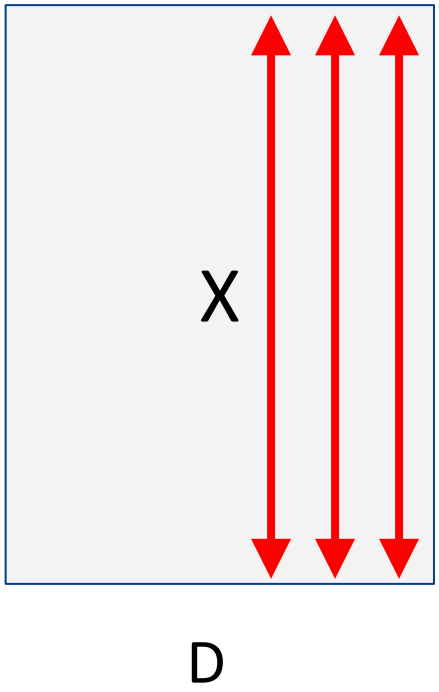
- Explicitly forcing the activations throughout a network to take on a unit gaussian distribution at the beginning of the training.

- Consider a batch of activations at some layer. To make each dimension unit gaussian, apply:

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

Batch Normalization

[Ioffe and Szegedy, 2015]



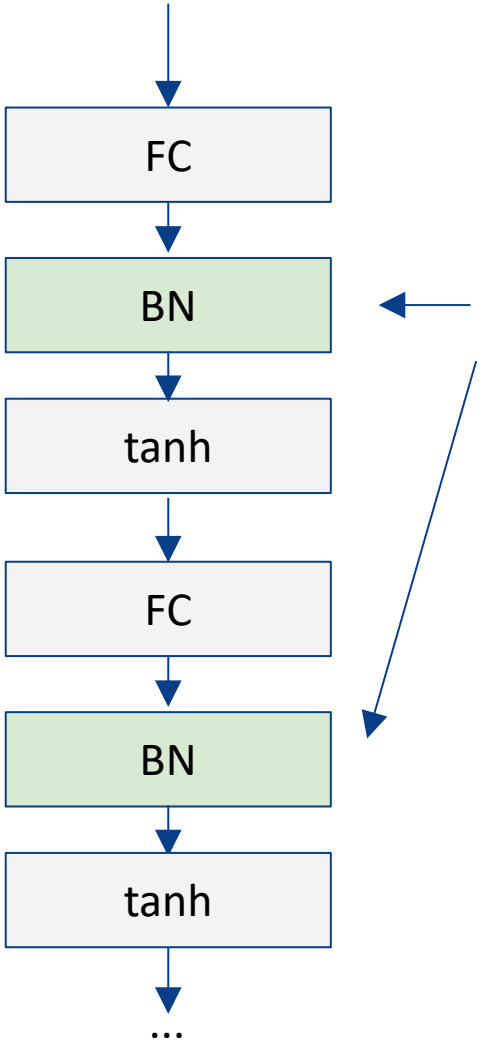
1. compute the empirical mean and variance independently for each dimension.

2. Normalize

$$\hat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

Batch Normalization

[Ioffe and Szegedy, 2015]



Usually inserted after Fully Connected / (or Convolutional) layers, and before nonlinearity.

Batch Normalization

[Ioffe and Szegedy, 2015]

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

- Improves gradient flow through the network
- Allows higher learning rates
- Reduces the strong dependence on initialization
- Acts as a form of regularization in a funny way, and slightly reduces the need for dropout, maybe

Regularization

- To prevent overfitting or help the optimization

- “Regularization is any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error.”

L2 parameter regularization

- also known as ridge regression or Tikhonov regularization
- For every w add $\frac{1}{2}\lambda w^2$ to the objective function.
- Gradient of this term: λw
- Weight decay: Encourages small weights

Weight Decay as Constrained Optimization

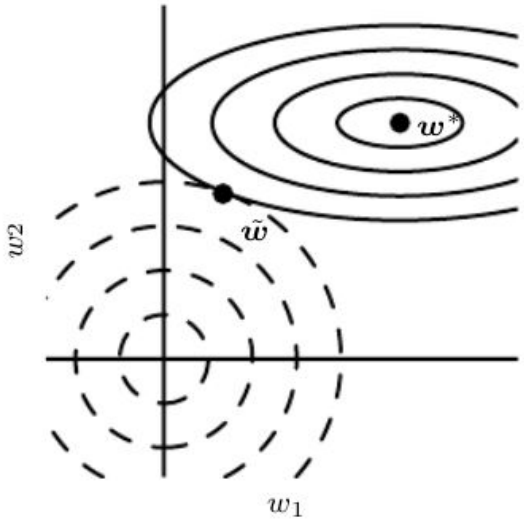


Figure 7.1: An illustration of the effect of L^2 (or weight decay) regularization on the value of the optimal $\boldsymbol{w}$. The solid ellipses represent contours of equal value of the unregularized objective. The dotted circles represent contours of equal value of the L^2 regularizer. At the point $\tilde{\boldsymbol{w}}$, these competing objectives reach an equilibrium. In the first dimension, the eigenvalue of the Hessian of J is small. The objective function does not increase much when moving horizontally away from $\boldsymbol{w}^*$. Because the objective function does not express a strong preference along this direction, the regularizer has a strong effect on this axis. The regularizer pulls w_1 close to zero. In the second dimension, the objective function is very sensitive to movements away from $\boldsymbol{w}^*$. The corresponding eigenvalue is large, indicating high curvature. As a result, weight decay affects the position of w_2 relatively little.

L1 regularization

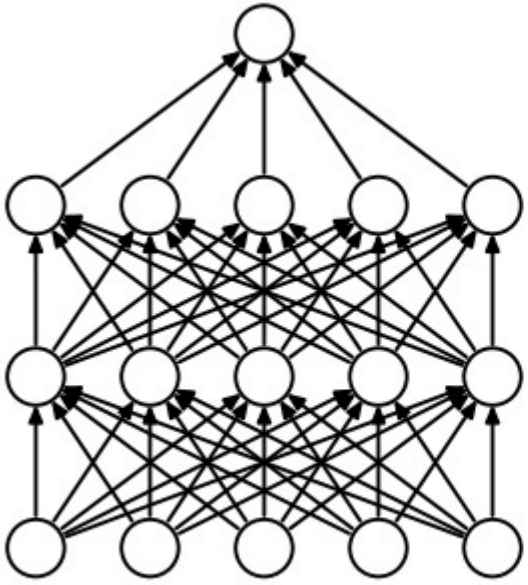
- For every w add $\lambda|w|$ to the objective function.

- it leads the weight vectors to become sparse during optimization

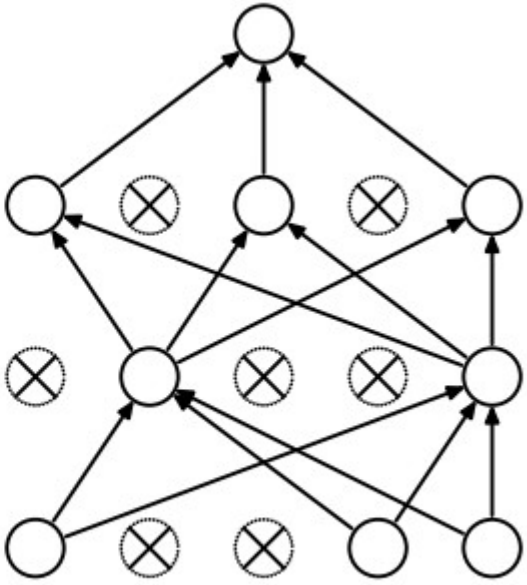
Regularization (dropout)

Regularization by Dropout

"randomly set some neurons to zero in the forward pass"



(a) Standard Neural Net

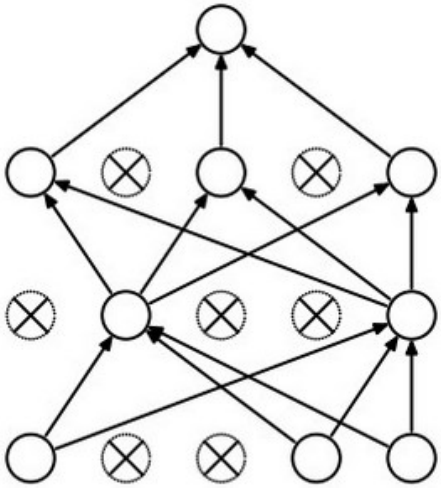


(b) After applying dropout.

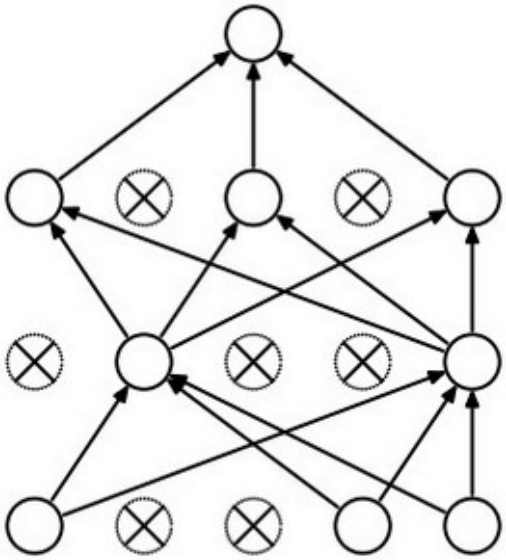
[Srivastava et al., 2014]

```python
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

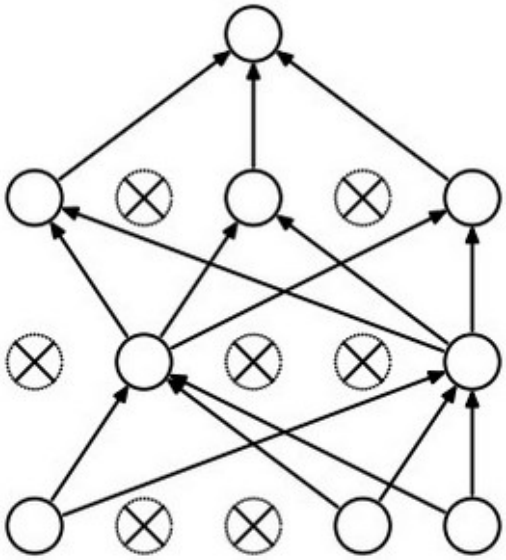


How could this possibly be a good idea?



Forces the network to have a redundant representation.

How could this possibly be a good idea?

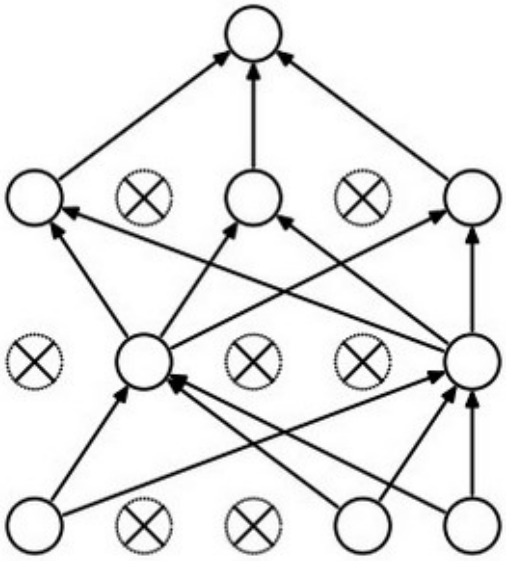


Another interpretation:

Dropout is training a large ensemble of models (that share parameters).

Each binary mask is one model, gets trained on only ~one datapoint.

At test time….



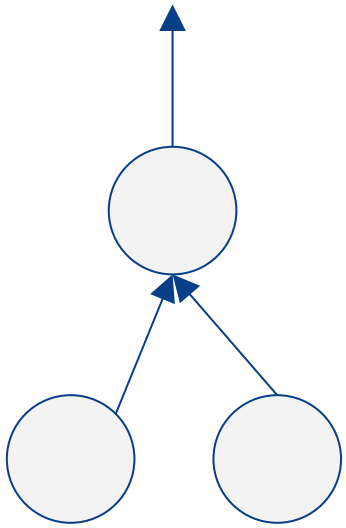
Ideally:
want to integrate out all the noise

Monte Carlo approximation:
do many forward passes with different dropout masks, average all predictions

At test time….

Can in fact do this with a single forward pass! (approximately)

Leave all input neurons turned on (no dropout).

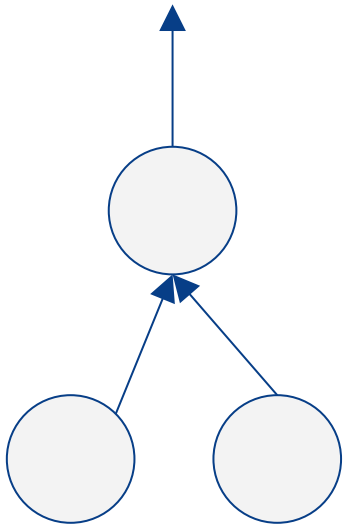


(this can be shown to be an approximation to evaluating the whole ensemble)

At test time….

Can in fact do this with a single forward pass! (approximately)

Leave all input neurons turned on (no dropout).



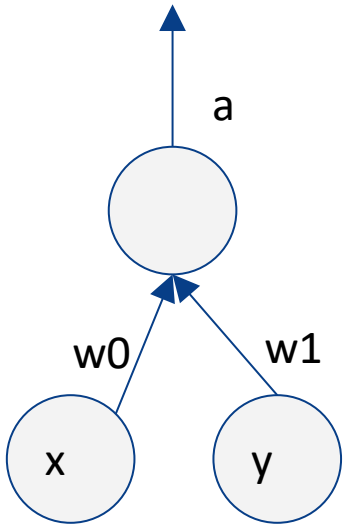
Q: Suppose that with all inputs present at test time the output of this neuron is x.

What would its output be during training time, in expectation? (e.g. if p = 0.5)

At test time….

Can in fact do this with a single forward pass! (approximately)

Leave all input neurons turned on (no dropout).



during test: **a = w0*x + w1*y**

during train:

E[a] = ¼ * (w0*0 + w1*0
+ w0*0 + w1*y
+ w0*x + w1*0
+ w0*x + w1*y)

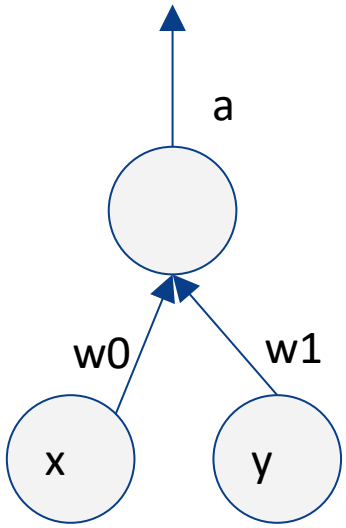
= ¼ * (2 w0*x + 2 w1*y)

= **½ * (w0*x + w1*y)**

At test time….

Can in fact do this with a single forward pass! (approximately)

Leave all input neurons turned on (no dropout).



during test: **a = w0*x + w1*y**

during train:

E[a] = ¼ * (w0*0 + w1*0

w0*0 + w1*y

w0*x + w1*0

w0*x + w1*y)

= ¼ * (2 w0*x + 2 w1*y)

= ½ * (w0*x + w1*y)

With p=0.5, using all inputs in the forward pass would inflate the activations by 2x from what the network was "used to" during training!

=> Have to compensate by scaling the activations back down by ½

We can do something approximate analytically

```
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

Dropout Summary

```
""" 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 in forward pass

scale at test time

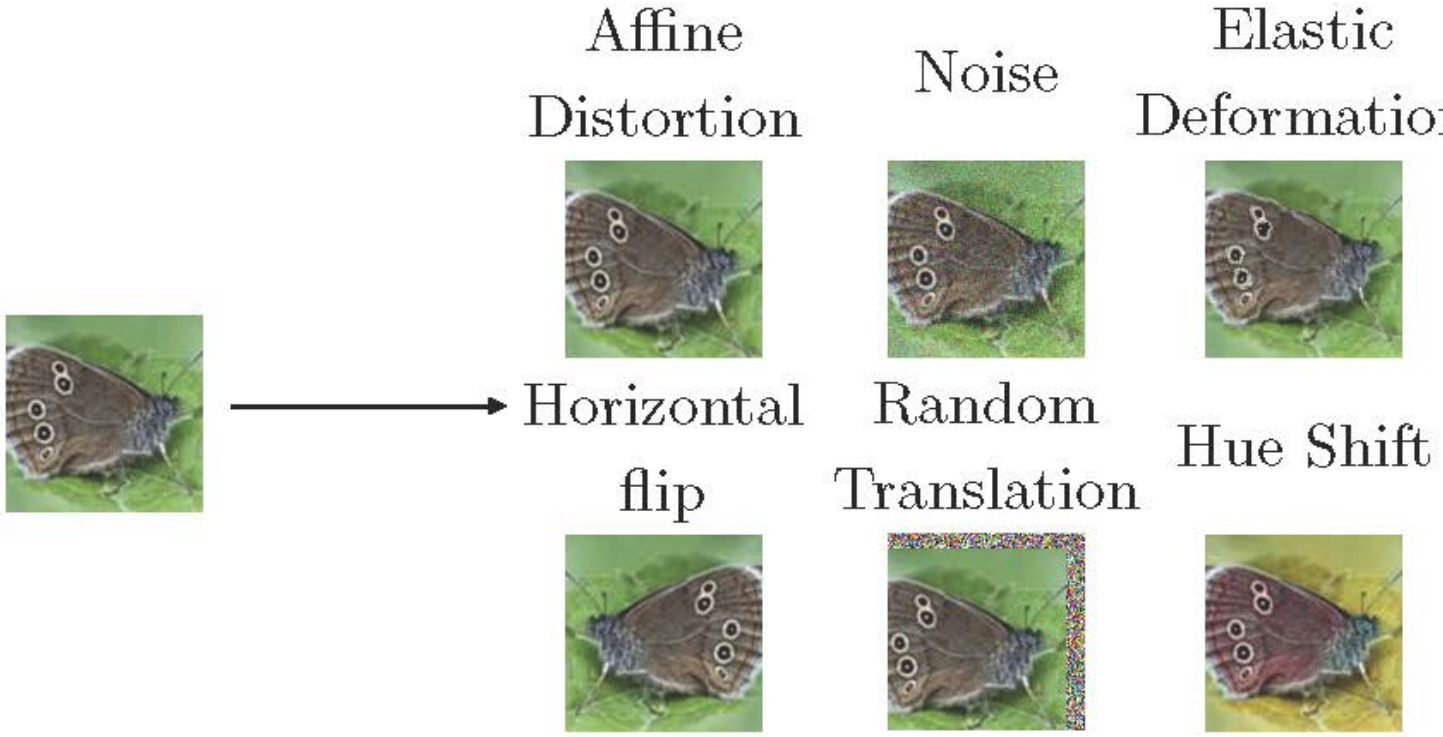
More common: “Inverted dropout”

```python
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
```

Dataset Augmentation

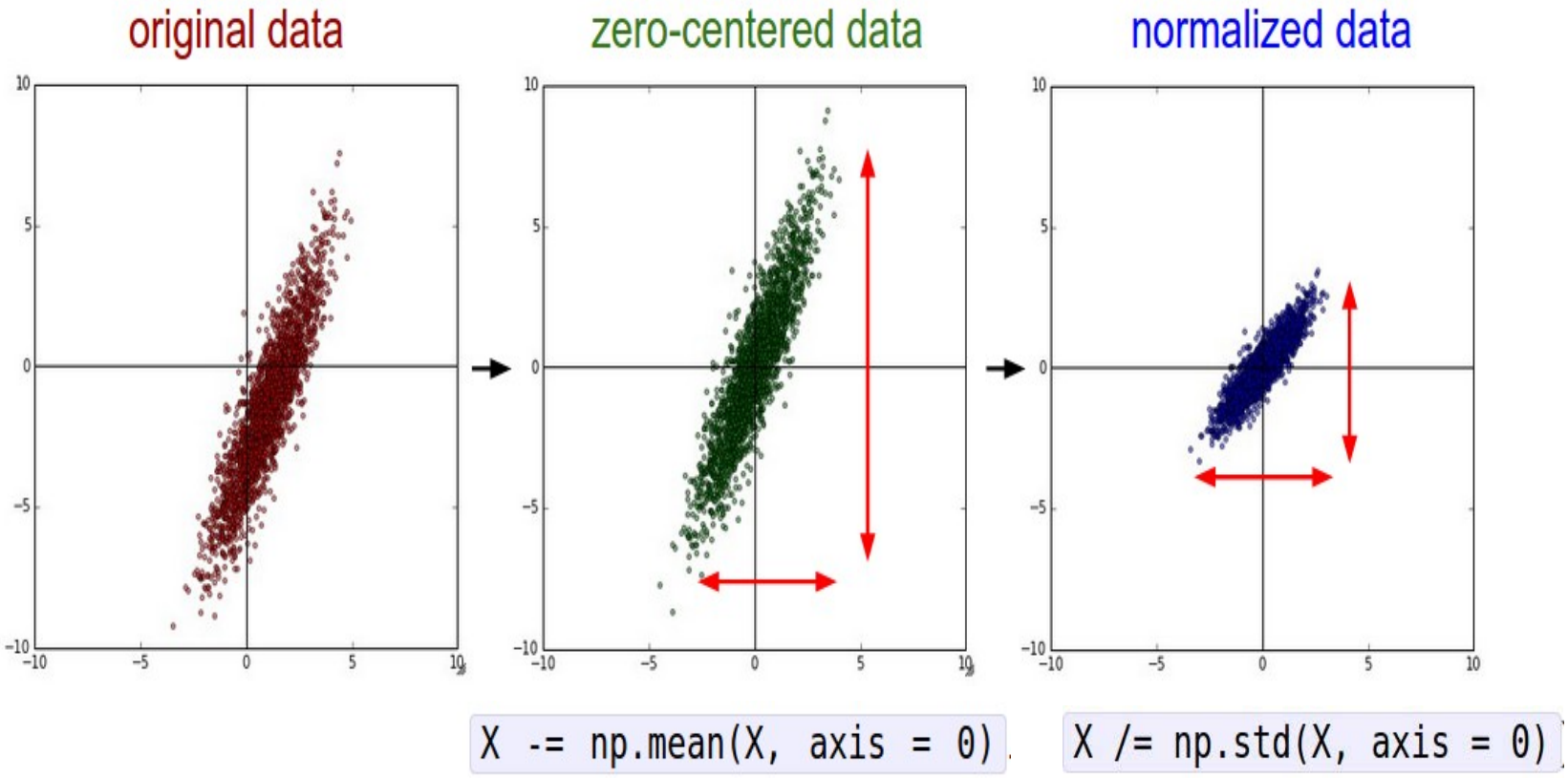
- to make a machine learning model generalize better is to train it on more data.
- create fake data
- We can generate new (x,y) pairs for classification just by transforming the x inputs in our training set.
  - Image (object classification,), speech
- Injecting noise in the input to a neural network can also be seen as a form of data augmentation.
- One way to improve the robustness of neural networks is simply to train them with random noise applied to their inputs.
- Dropout a process of constructing new inputs by multiplying by noise

Dataset Augmentation



Learning Process

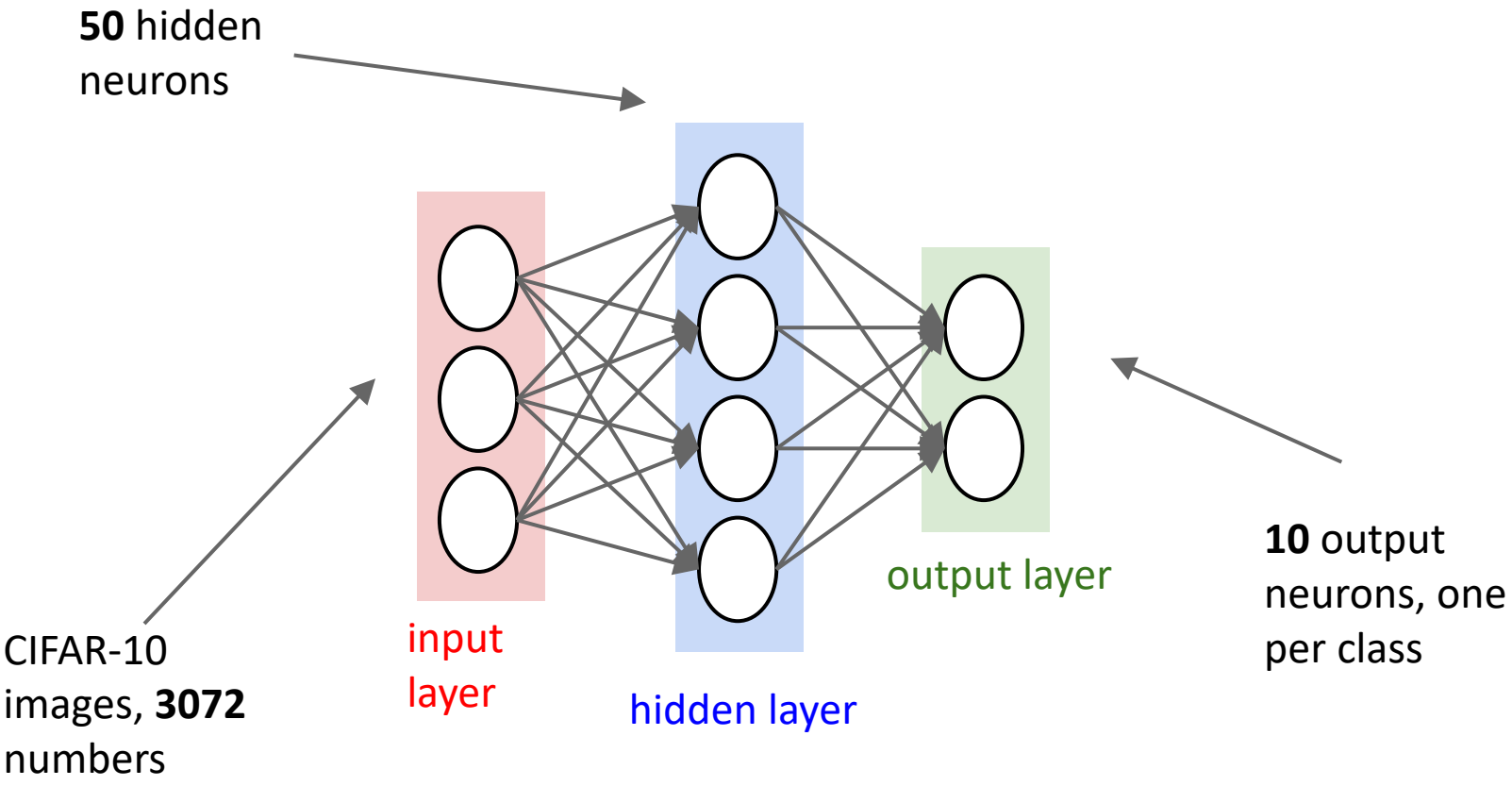
Step 1: Preprocess the data



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

Step 2: Choose the architecture:

say we start with one hidden layer of 50 neurons:



Hyperparameter Optimization

Hyperparameters to play with:

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

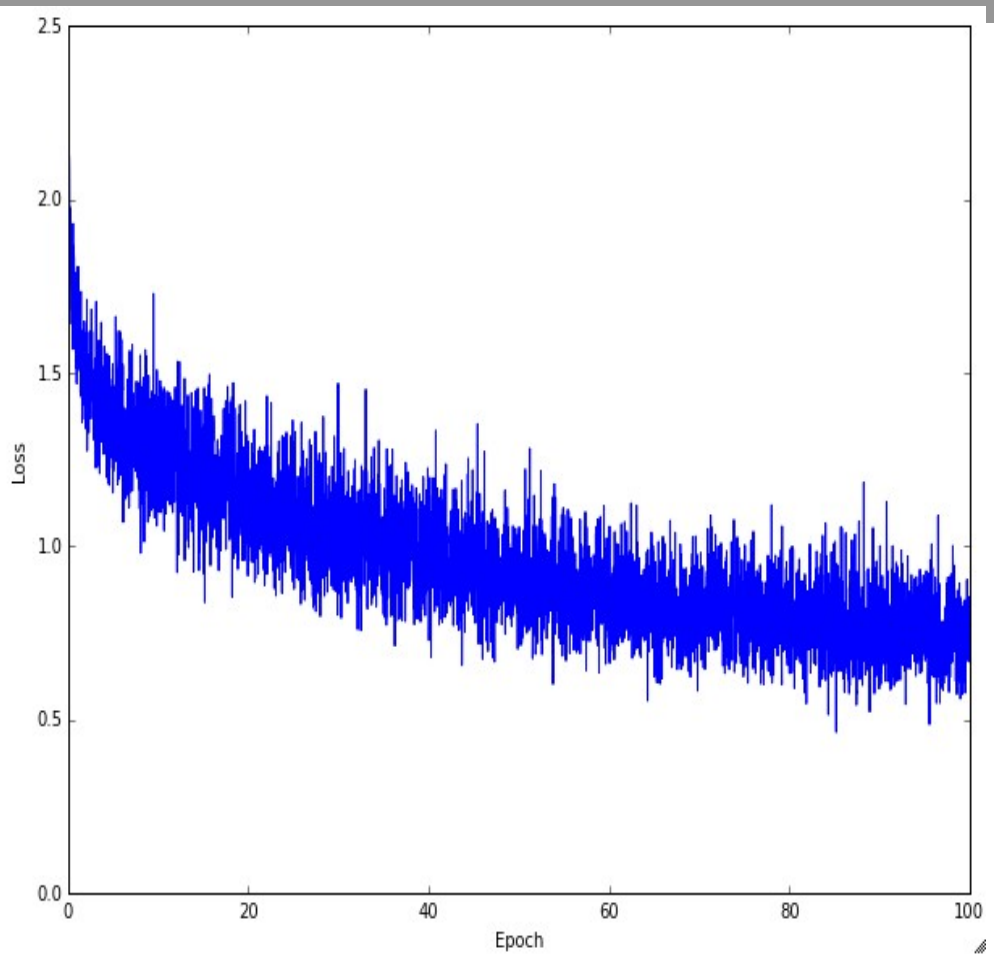
neural networks practitioner
music = loss function

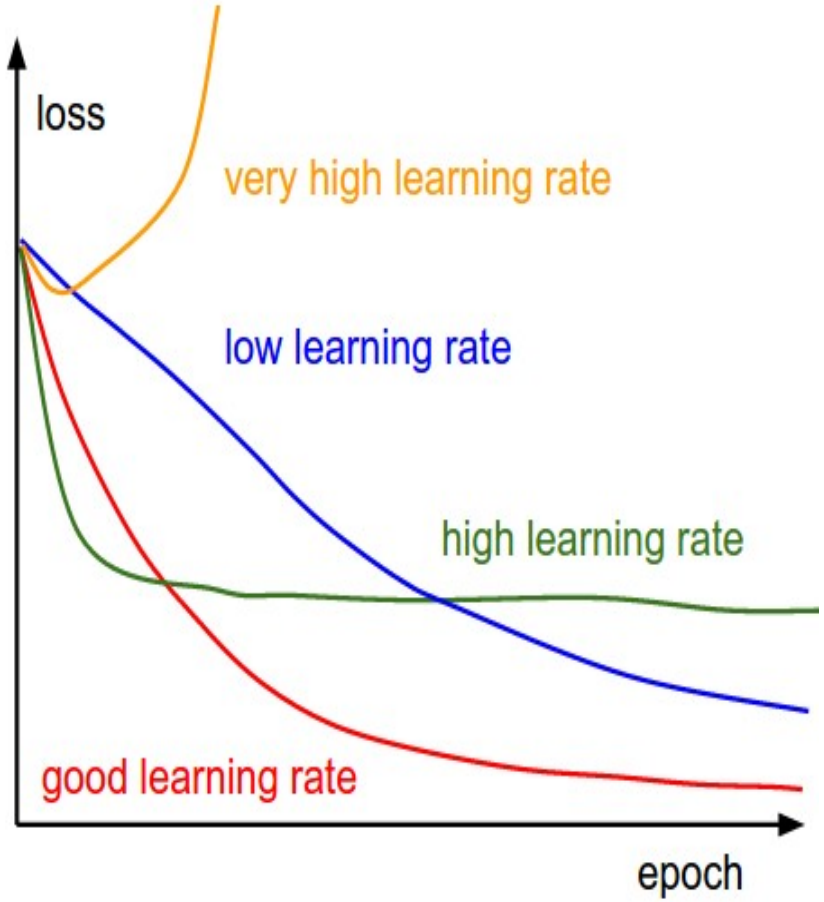
Training

1. Check that loss is reasonable.

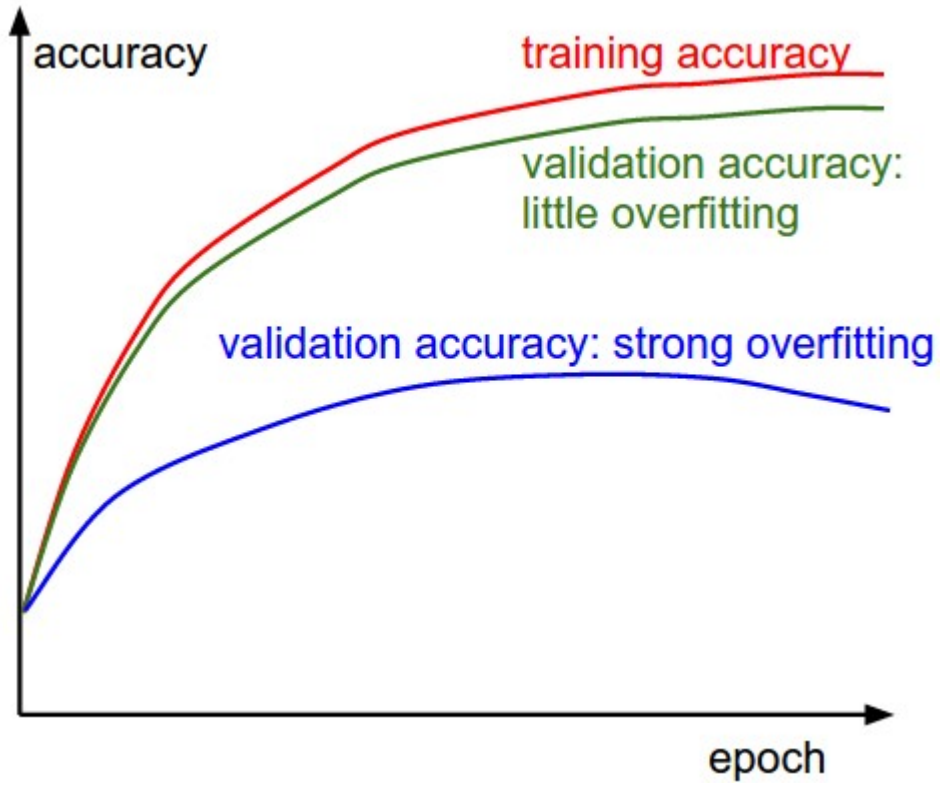
- Loss goes up as you increase regularization.
- Make sure that you can overfit very small portion of the training data
- Start with small regularization and find learning rate that makes the loss go down.
  - loss not going down: means learning rate too low
  - loss exploding: learning rate too high

Monitor and visualize the loss curve

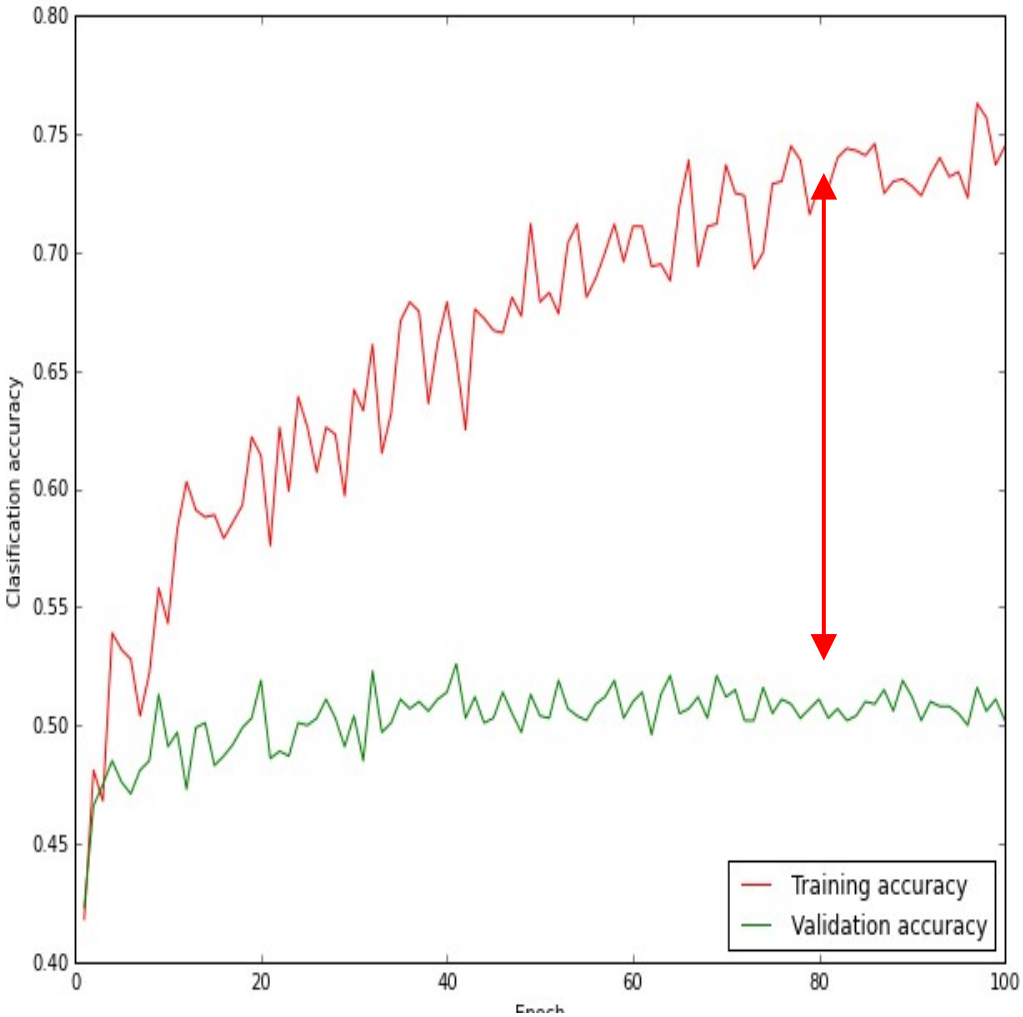




Train/ Val accuracy



Monitor and visualize the accuracy:



big gap = overfitting

=> increase regularization strength?

no gap

=> increase model capacity?

Cross-validation strategy

May 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)