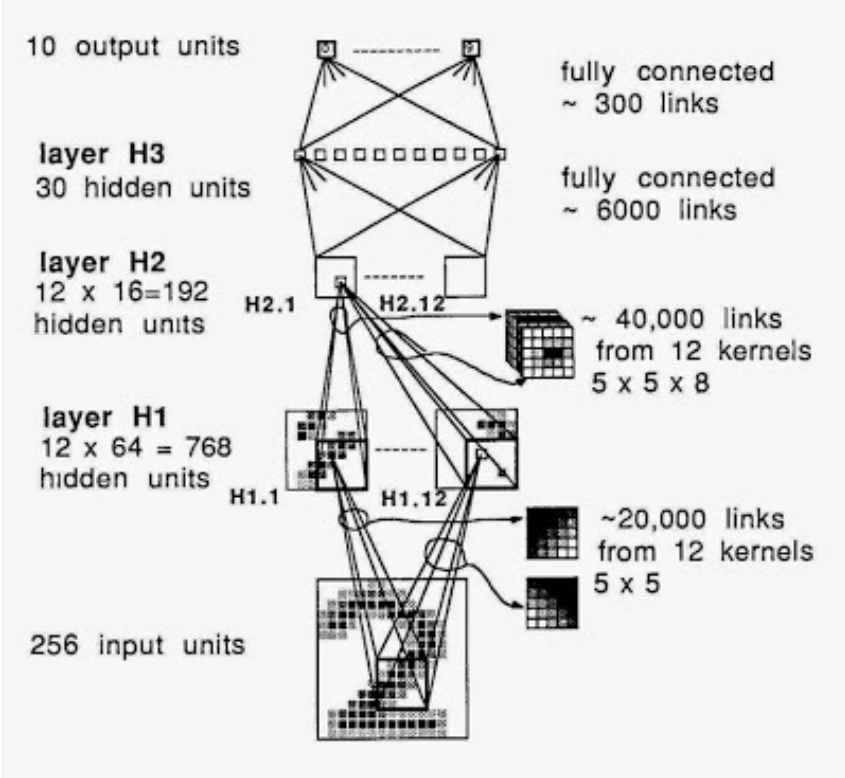
Deep Neural Network

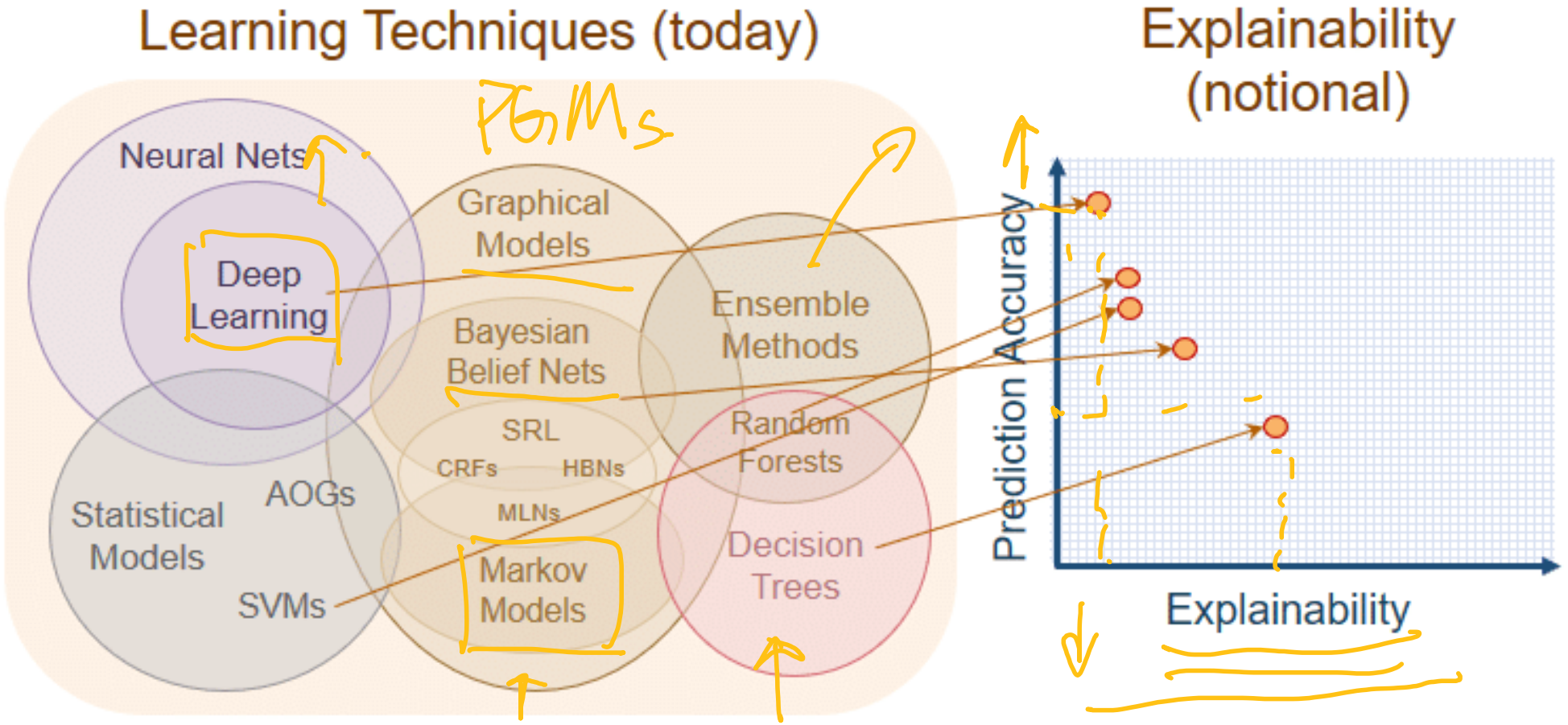
Deep Learning (Data-Driven Machines)

- Learning from examples and today's state-of-the-art recognition techniques
- Deep Learning emphasizes the network architecture of today's most successful machine learning approaches.



A Deep Network from 1989

Deep Learning vs. Other Method



Deep Learning Revolution

Is deep learning a revolution in artificial intelligence?

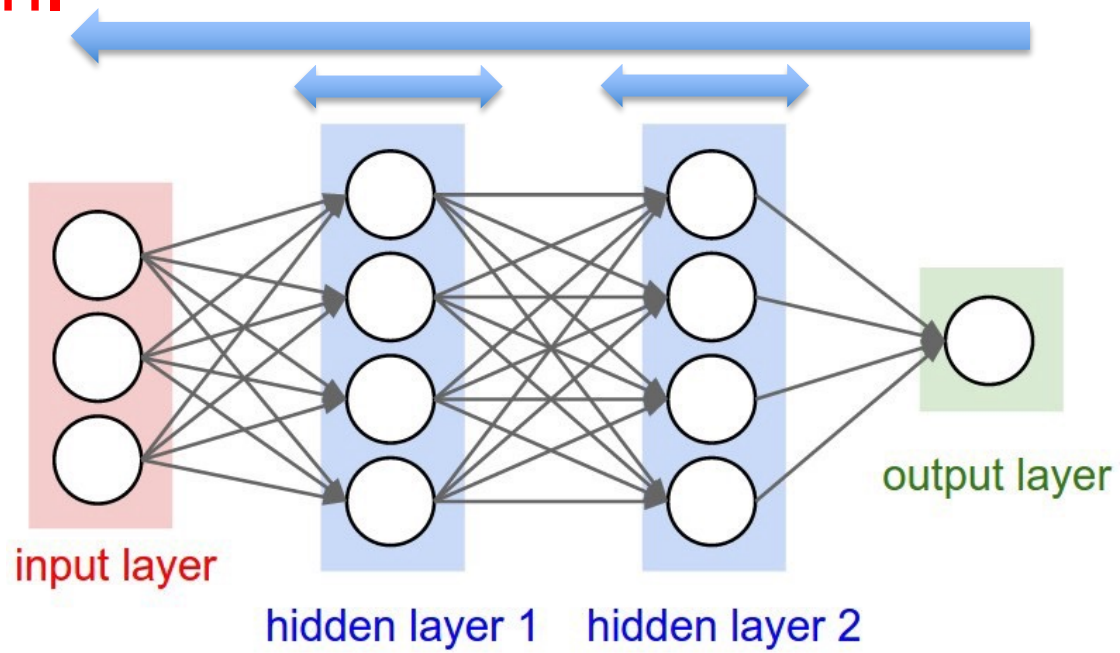
- Apple's Siri virtual personal assistant
- Google's Street View & Self-Driving Car
- Google/Facebook/Tweeter/Yahoo
- Google's DeepMind AlphaGO

Deep Neural Network

- 'Deep Learning' means using a neural network with several layers of nodes between input and output
- Why is it generally better than other methods on image, speech and certain other types of data?
  - the series of layers between input & output do feature identification and processing in a series of stages, just as our brains seem to.
- Multilayer neural networks have been around for 35 years. What's actually new?
  - We have always had good algorithms for learning the weights in networks with 1 hidden layer. But these algorithms are not good at learning the weights for networks with more hidden layers
  - What's new is: algorithms for training many-layer networks

Training a Deep Network

- Weights are learned layer by layer via unsupervised learning.
- Final layer is learned as a supervised_neural network.
- All weights are fine-tuned using supervised back propagation.

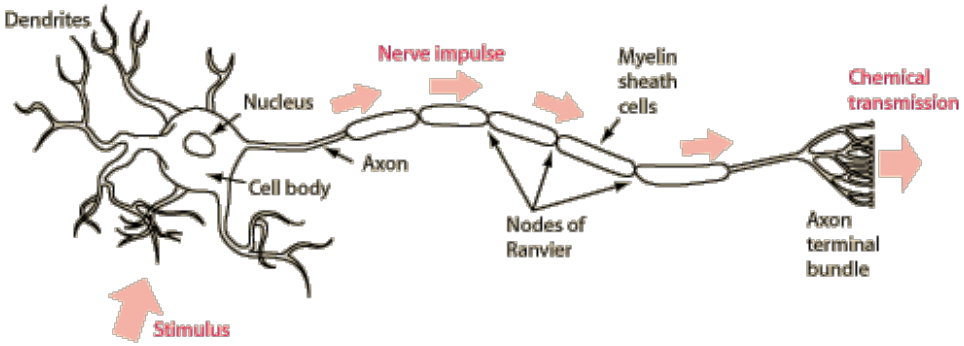


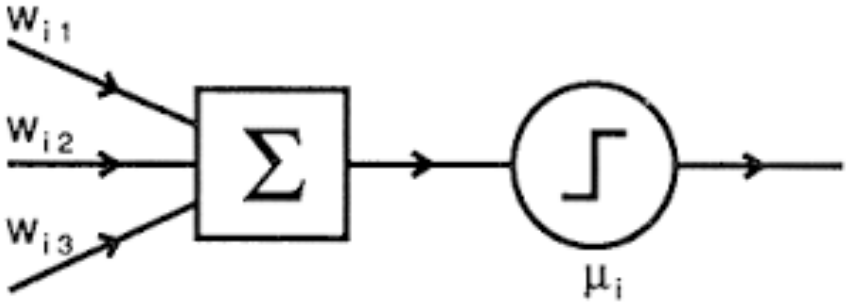
Deep Learning

- Deep Neural Network Model
  - Activation Functions
  - Hyperparameter
- Training Deep Neural Network
  - Data preprocessing
  - Initial weights
  - Optimization solvers
  - Practical Techniques

Activation Functions

$$y = \varphi\left(\sum_{i=1}^{m} x_i - b\right)$$

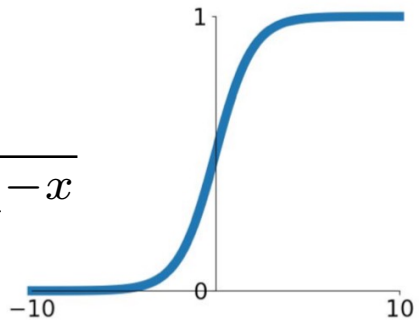




How to Choose Activation Functions?

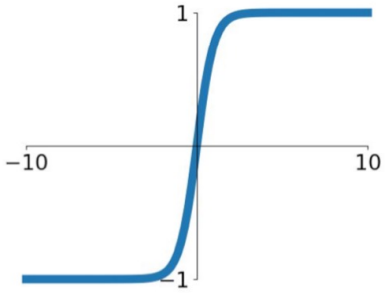
- Sigmoid

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



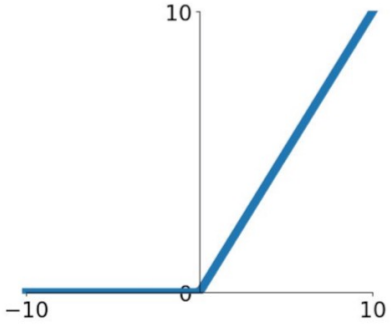
- Tanh

$$\tanh(x)$$



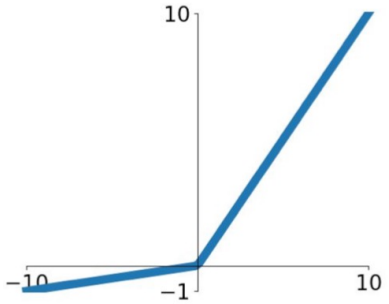
- ReLU

$$\max(0, x)$$



- Leaky ReLU

$$\max(0.1x, x)$$

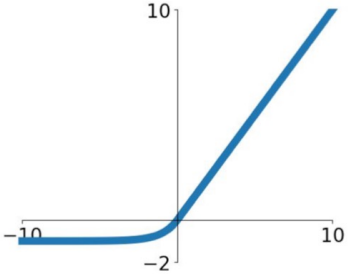


- Maxout

$$\max(\omega_1^T x + b_1, \omega_2^T x + b_2)$$

- ELU - Exponential Linear Unit

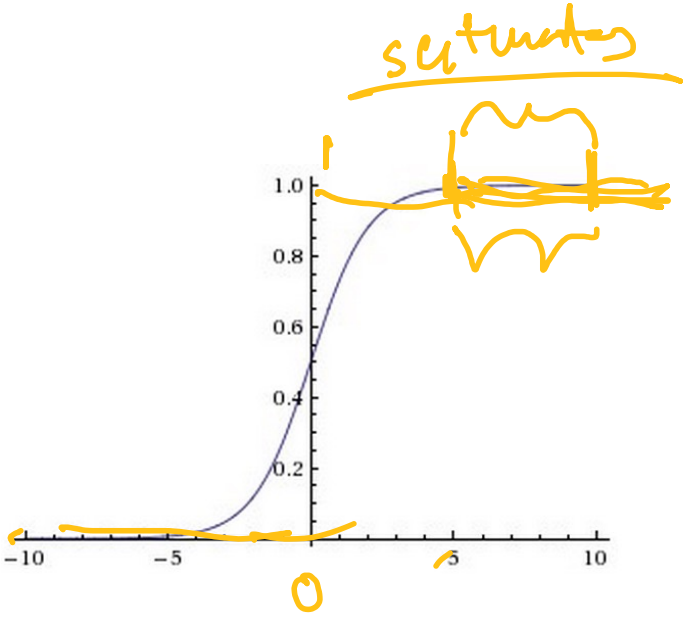
$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Sigmoid

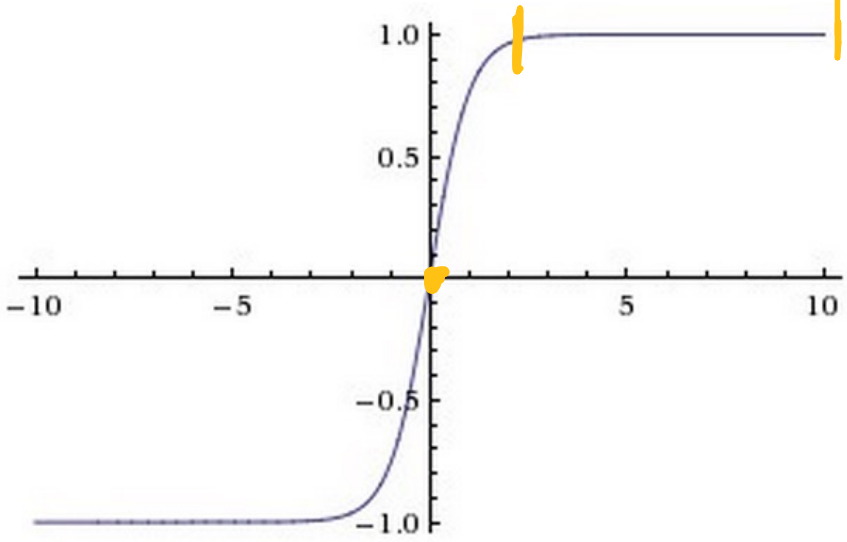
- Sigmoid takes a real-valued number and "squashes" it into range between 0 and 1.
- **Nice interpretation** of the firing mechanism of a neuron
- Not commonly used
  - Sigmoid saturates and kill gradients
  - Sigmoid outputs are not zero-centered
  - exp() computation expensive

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



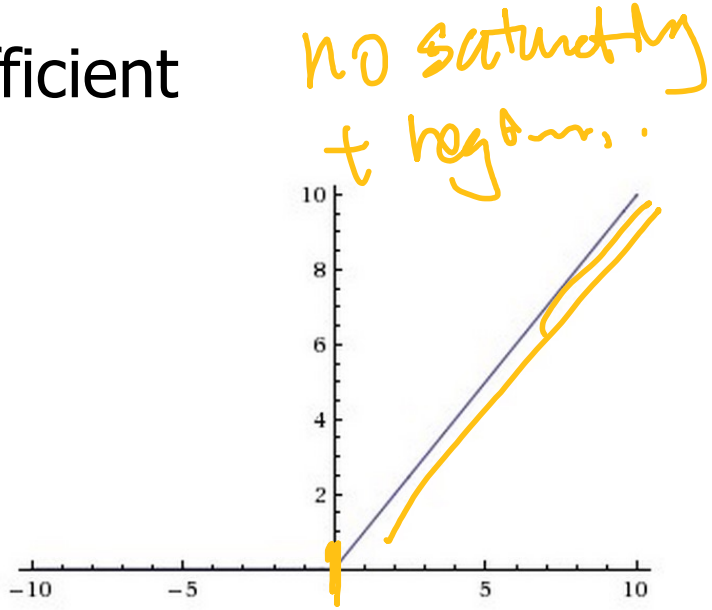
Tanh

- tanh(x) squashes a real-valued number to the range [-1, 1]
  Nice interpretation of the firing mechanism of a neuron
- tanh(x) saturate and kill gradients
- The outputs of tanh(x) are zero-centered
- tanh non-linearity is always preferred to the sigmoid nonlinearity



Rectified Linear Unit (ReLU)

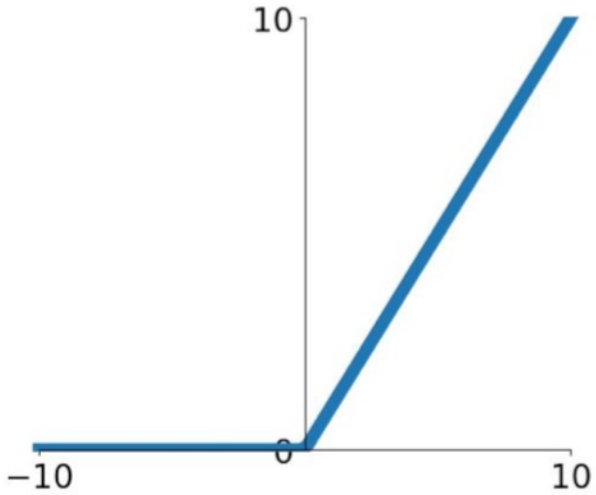
- Rectified Linear Unit (ReLU) computes $\max(0,x)$ which is simply threshold at zero
- ReLU has become very popular in the last few years.
- Pros
  - Easy to implement and computation efficient
  - Without saturating in +region
  - Greatly accelerate the convergence of stochastic gradient descent (much faster than sigmoid/tanh)
- Cons
  - Not zero-centered output
  - ReLU units can be fragile during training and can "die"



Leaky ReLU

- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practice!
- Will not "die"

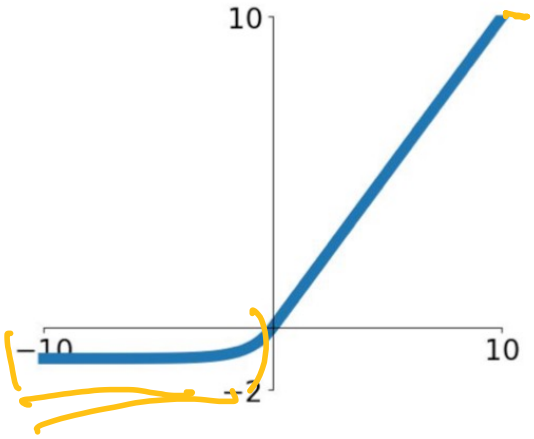
$$\max(0.1x, x)$$



Exponential Linear Units (ELU)

- All benefits of ReLU
- Closer to zero mean outputs
- Negative saturation regime
- Add some robustness to noise (compared with Leaky ReLU)
- Computation requires exp()

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Maxout Neuron

- Does not have the basic form of dot product: nonlinearity
- Generalizes ReLU and Leaky ReLU
- Linear regime; does not saturate; does not die
- The number of parameters doubles.

$$\max(\omega_1^T x + b_1, \omega_2^T x + b_2)$$

$w_1^T x + b_1 > w_2^T x + b_2$.

$\max(w_1^T x + b_1, w_1^T x + b_2) = w_1^T x + b_1$.

General Rules for Activation Functions

- Use ReLU. Choose a proper learning rate.
- Try out Leaky ReLU / Maxout / ELU
- Try out tanh but don't expect much
- Don't use sigmoid in hidden layers

Hyperparameter Optimization

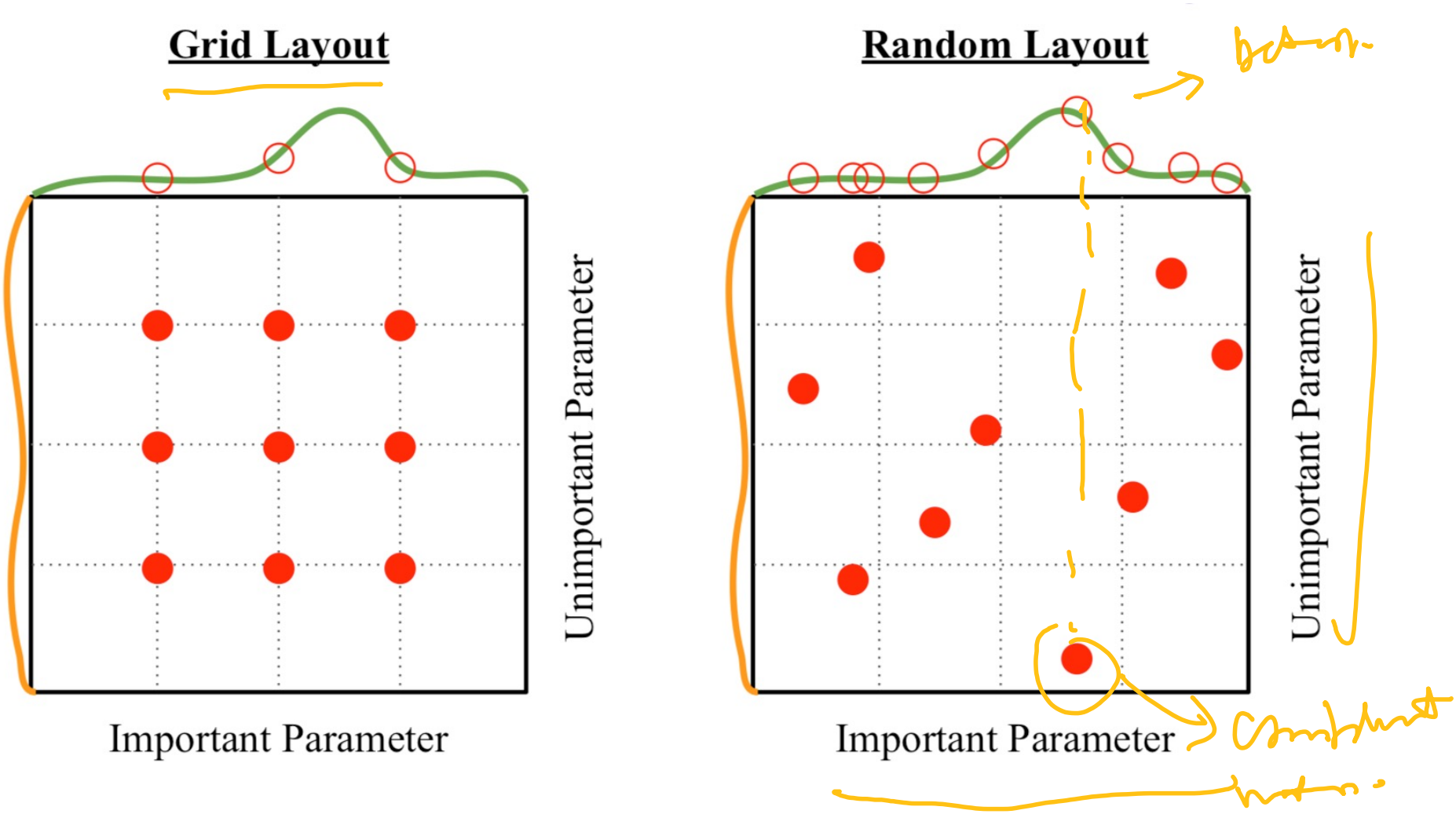
What Hyperparameters to Optimize?

- Network architecture
- Learning rate; its decay schedule; update type
- Regularization (L2/Dropout strength)

Cross-Validation Strategy (Manual Trials)

- Coarse -> fine: cross-validation in stages
  - First stage: only a few epochs to get rough idea of what parameters work
  - Second stage: longer running time, finer search
  - Going on as needed
- Tip for detecting explosions in the solver: If the cost is ever > 3 * original cost, break out early
- Best to optimize in log space (to the power of 10)

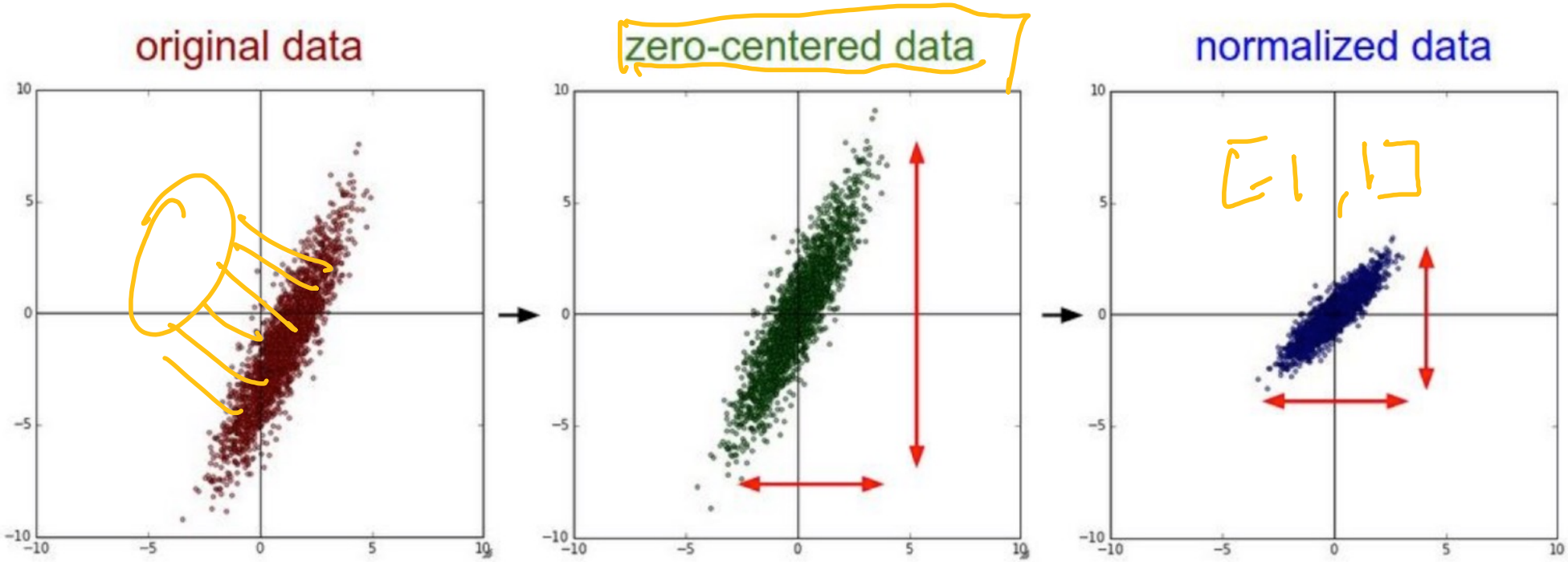
Random Search vs. Grid Search



Data Preprocessing

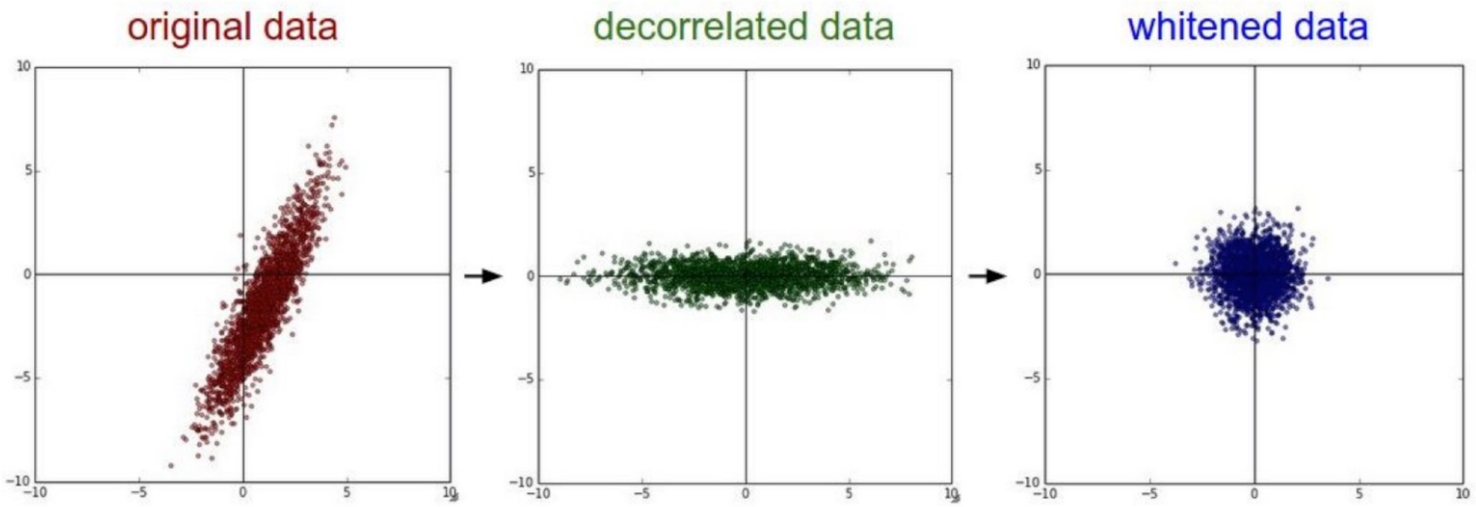
Data Preprocessing: Recommended

- Data preprocessing is a must!
- Zero-center: put the mean at zero. Why?
- Normalize: assume data follow normal distribution



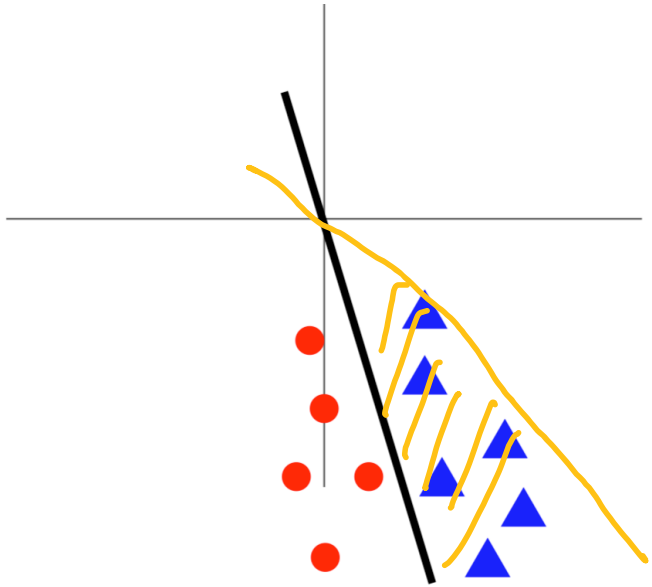
Data Preprocessing: Optional

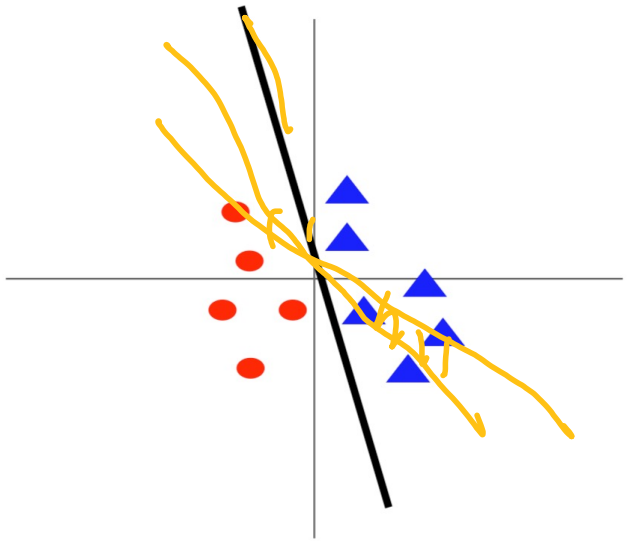
- Decorrelate: project the original zero-centered data into the eigenbasis
- Whitening: takes the data in the eigenbasis and divides every dimension by the eigenvalue to normalize the scale



Why Data Normalization?

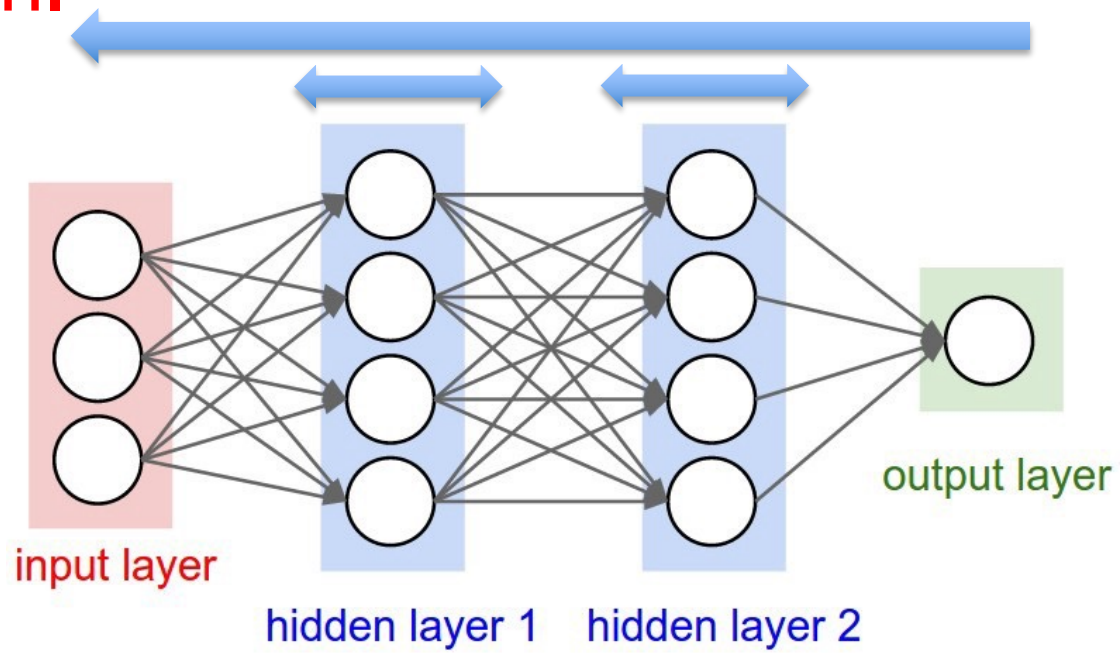
- **Before normalization**: classification loss very sensitive to changes in weight matrix; hard to optimize
- **After normalization**: less sensitive to small changes in weights; easier to optimize





Recap: Training a Deep Network

- Weights are learned layer by layer via unsupervised learning.
- Final layer is learned as a supervised_neural network.
- All weights are fine-tuned using supervised back propagation.



Recap: Deep Learning Challenges

- Deep Neural Network Model
  - Activation Functions
  - Hyperparameter
- Training Deep Neural Network
  - Data preprocessing
  - Initial weights
  - Optimization solvers
  - Practical Techniques

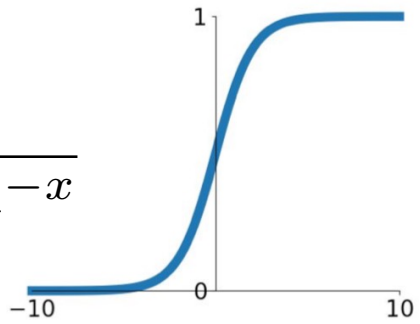
Recall General Rules for Activation Functions

- Use ReLU. Choose a proper learning rate.
- Try out Leaky ReLU / Maxout / ELU
- Try out tanh but don’t expect much
- Don’t use sigmoid in hidden layers

Recap Activation Function

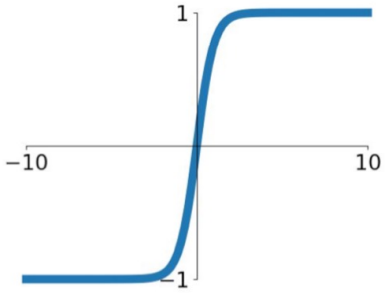
- Sigmoid

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



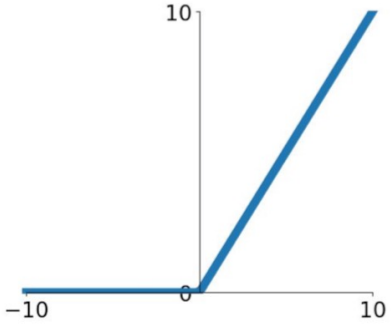
- Tanh

$$\tanh(x)$$



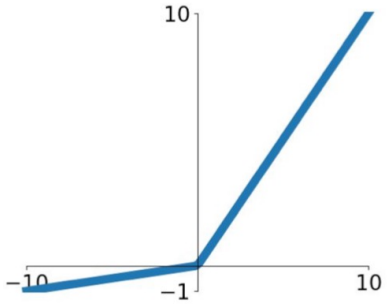
- ReLU

$$\max(0, x)$$



- Leaky ReLU

$$\max(0.1x, x)$$

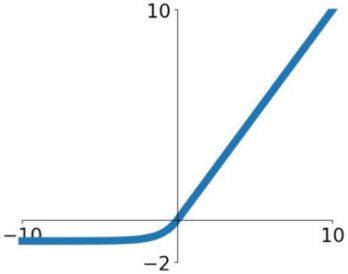


- Maxout

$$\max(\omega_1^T x + b_1, \omega_2^T x + b_2)$$

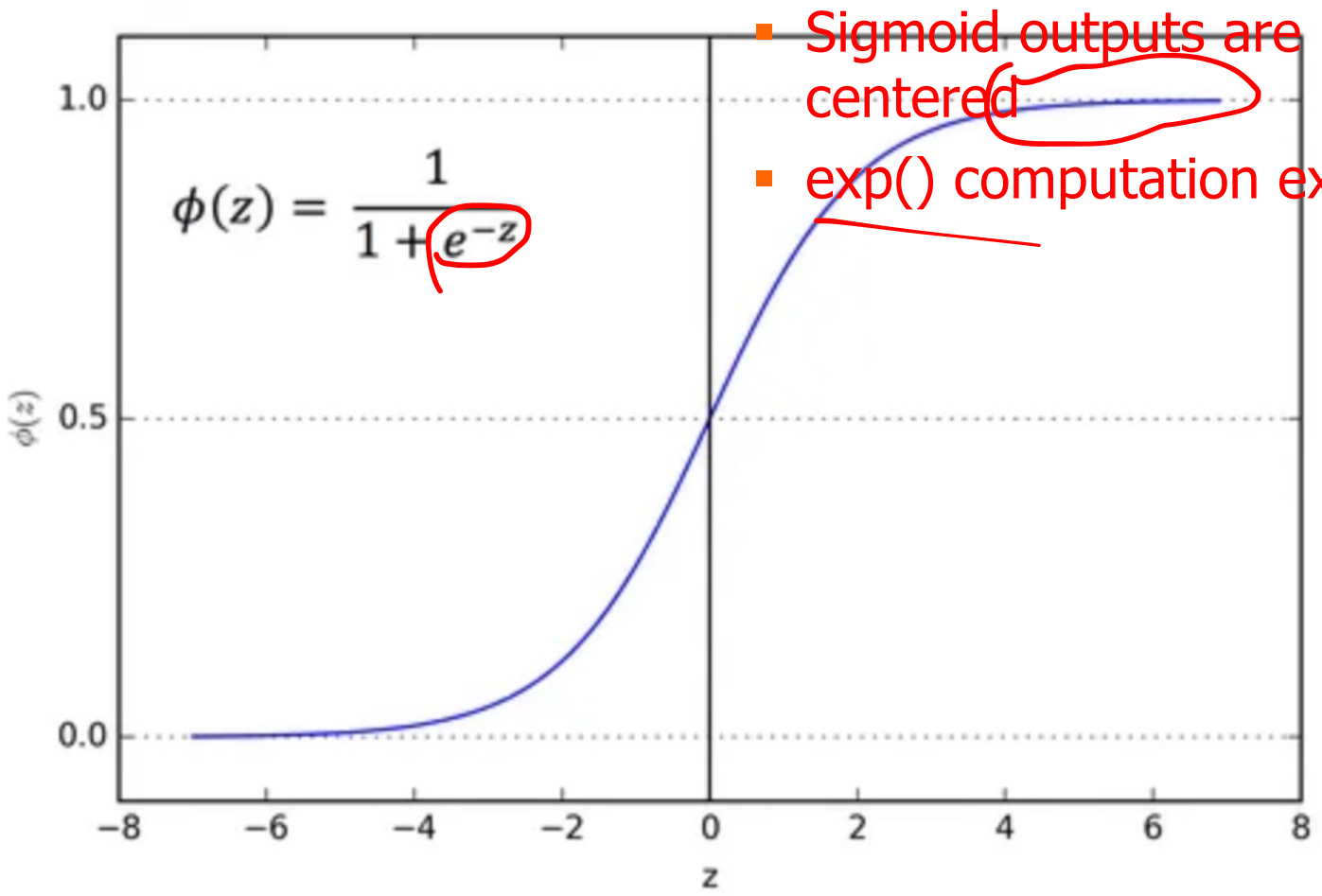
- ELU - Exponential Linear Unit

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Recap Sigmoid

- [0,1] good for modeling probability

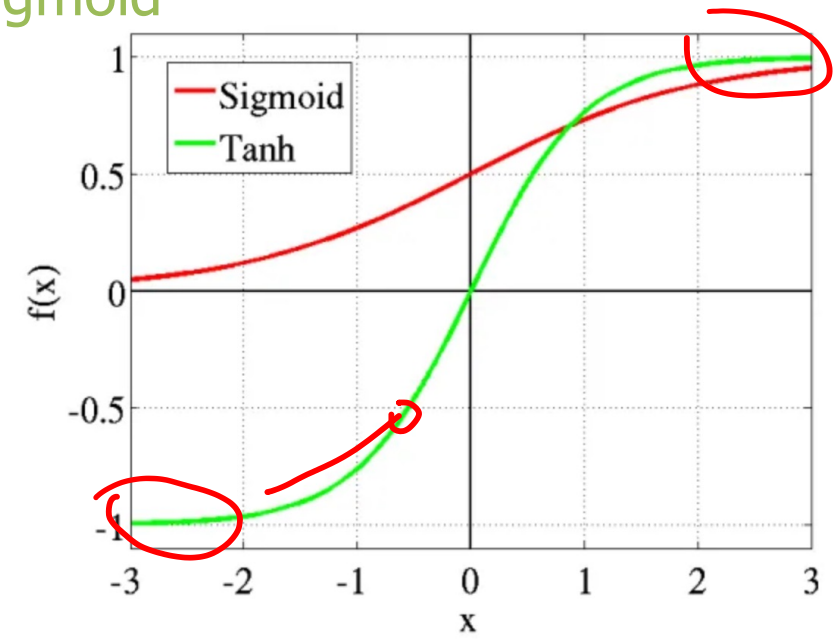


- Sigmoid saturates and kill gradients
- Sigmoid outputs are not zero-centered
- exp() computation expensive

Recap Tanh

- Range [-1, 1], good for classification problems.
- The outputs of tanh(x) are zero-centered.
- Tanh non-linearity is always preferred to the sigmoid nonlinearity

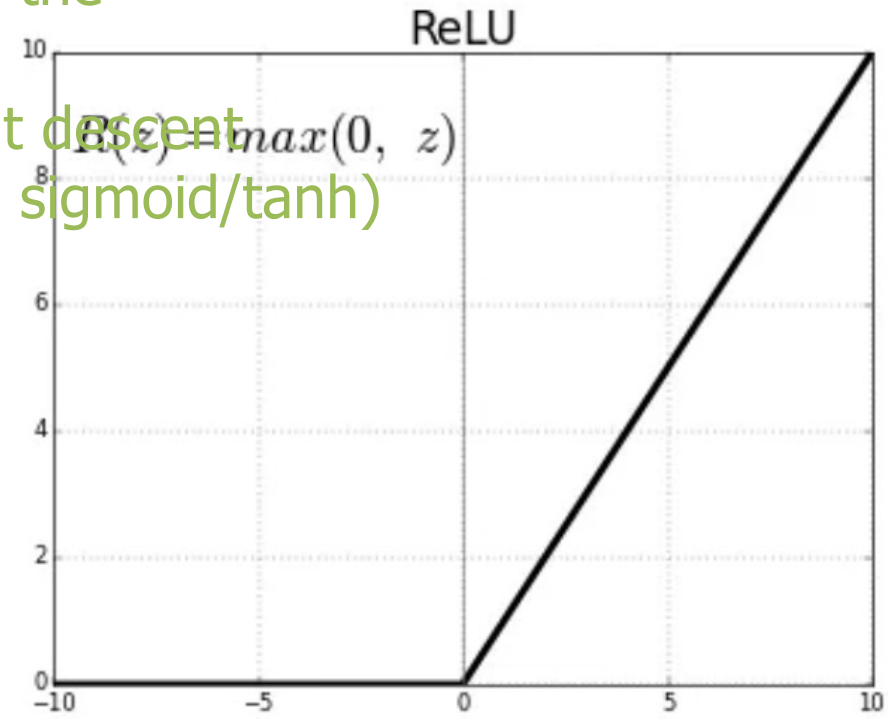
- Tanh(x) saturate and kill gradients



Recall ReLU

- The most used activation function
- Easy to implement and computation efficient
- Without saturating in +region
- Greatly accelerate the convergence of stochastic gradient descent (much faster than sigmoid/tanh)

- Negative values below 0
- Die ReLU problem:
  - A "dead" ReLU always outputs the same value (zero as it happens, but that is not important) for any input.

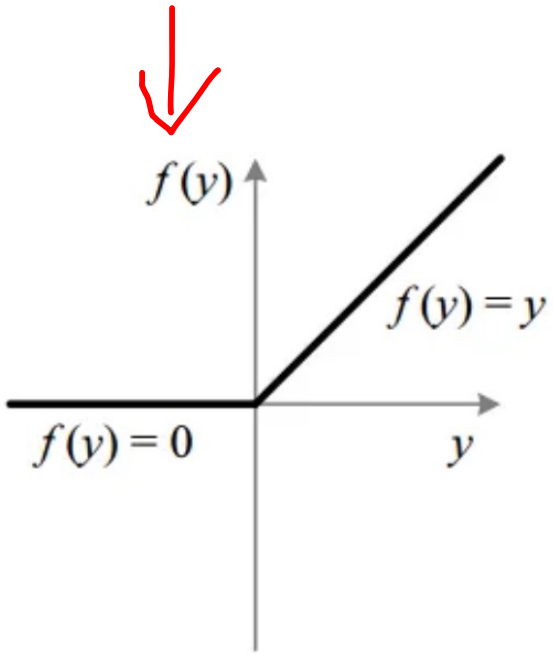


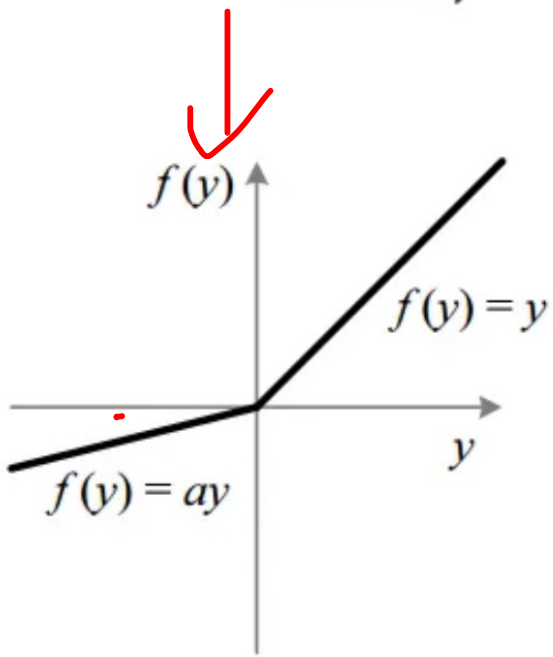
Recall Leaky ReLU

- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practice!
- Will not "die"

- Sensitive to noise

$$f'(x) = \begin{cases} 1, & x > 0 \\ 0.01, & x \leq 0 \end{cases}$$





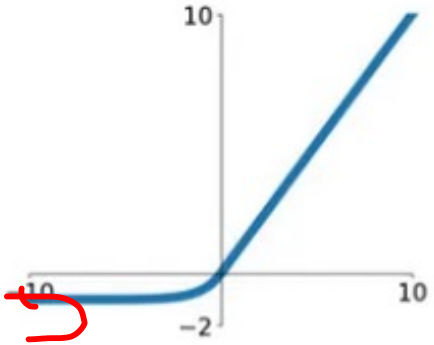
Recall Exponential Linear Unit (ELU)

- All benefits of ReLU
- Closer to zero mean outputs (Faster convergence)
- Add some robustness to noise (compared with Leaky ReLU)

- Negative saturation regime
- Computation requires exp()

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Recall Maxout Neurons

- Does not have the basic form of dot product: nonlinearity
- Generalizes ReLU and Leaky ReLU
- Linear regime; does not saturate; does not die

- Computationally expensive as it doubles the number of
- Parameters for each neuron.

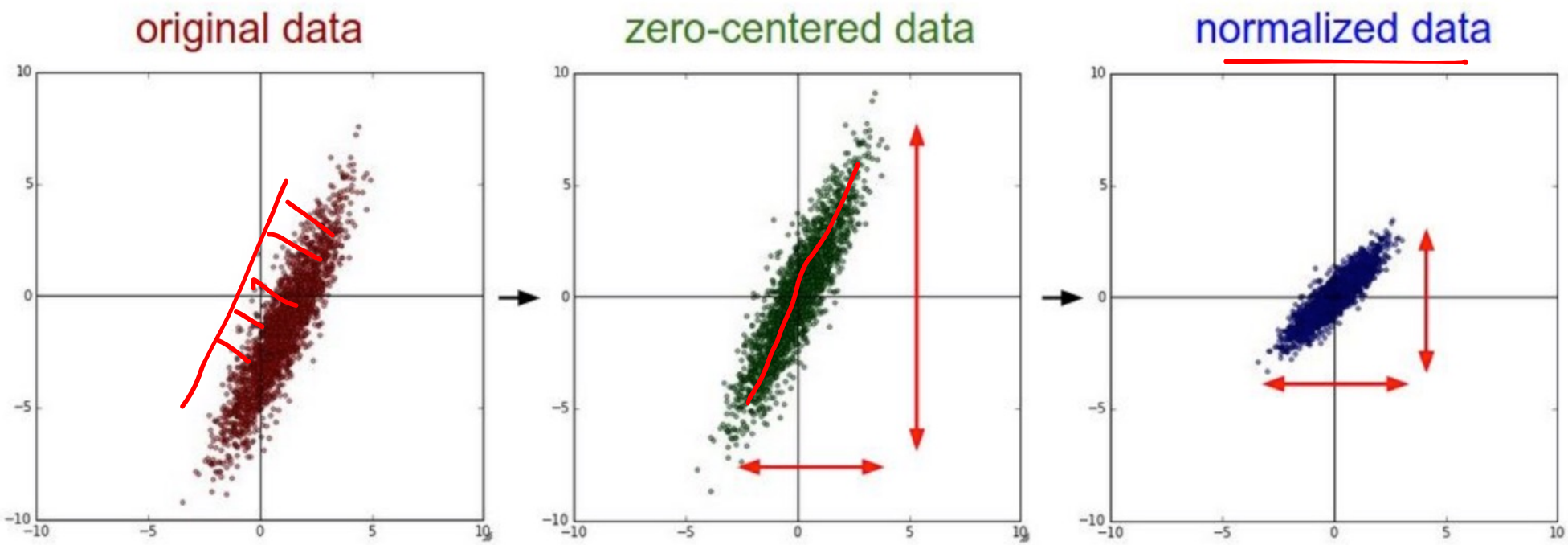
$$\max(\omega_1^T x + b_1, \omega_2^T x + b_2)$$

Recap What Hyperparameters to Optimize?

- Network architecture
- Learning rate; its decay schedule; update type
- Regularization (L2/Dropout strength)

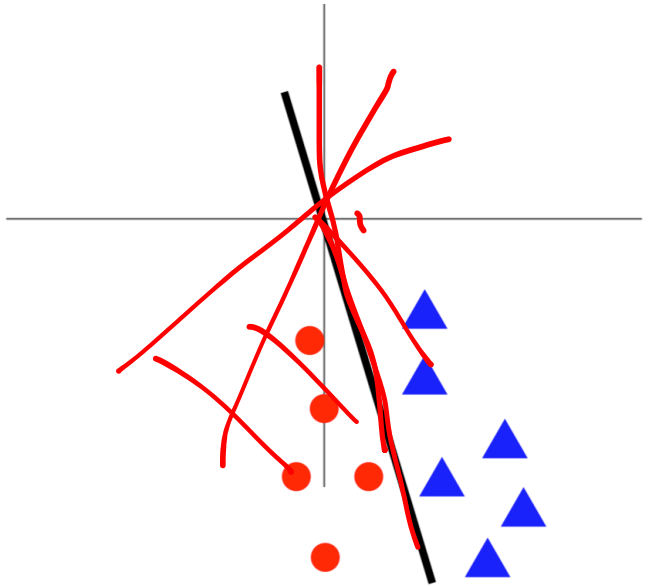
Recap Data Preprocessing: Recommended

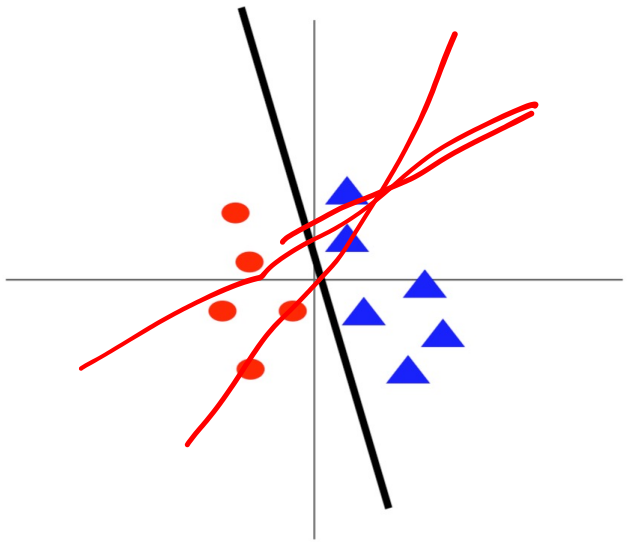
- Data preprocessing is a must!
- Zero-center: put the mean at zero. Why?
- Normalize: assume data follow normal distribution



Recap Why Data Normalization?

- **Before normalization**: classification loss very sensitive to changes in weight matrix; hard to optimize
- **After normalization**: less sensitive to small changes in weights; easier to optimize



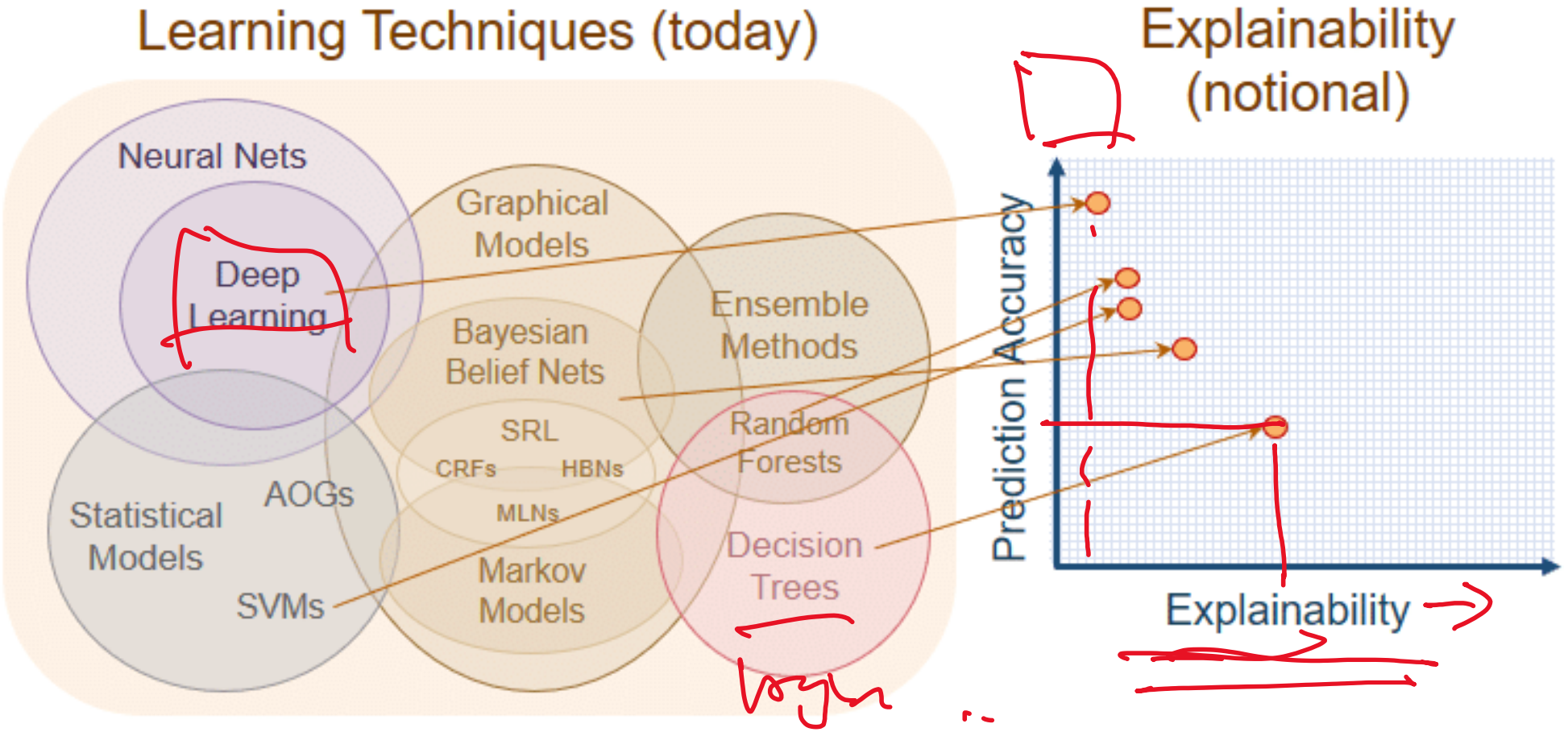


Initial Weights

How to Initialize the Weights?

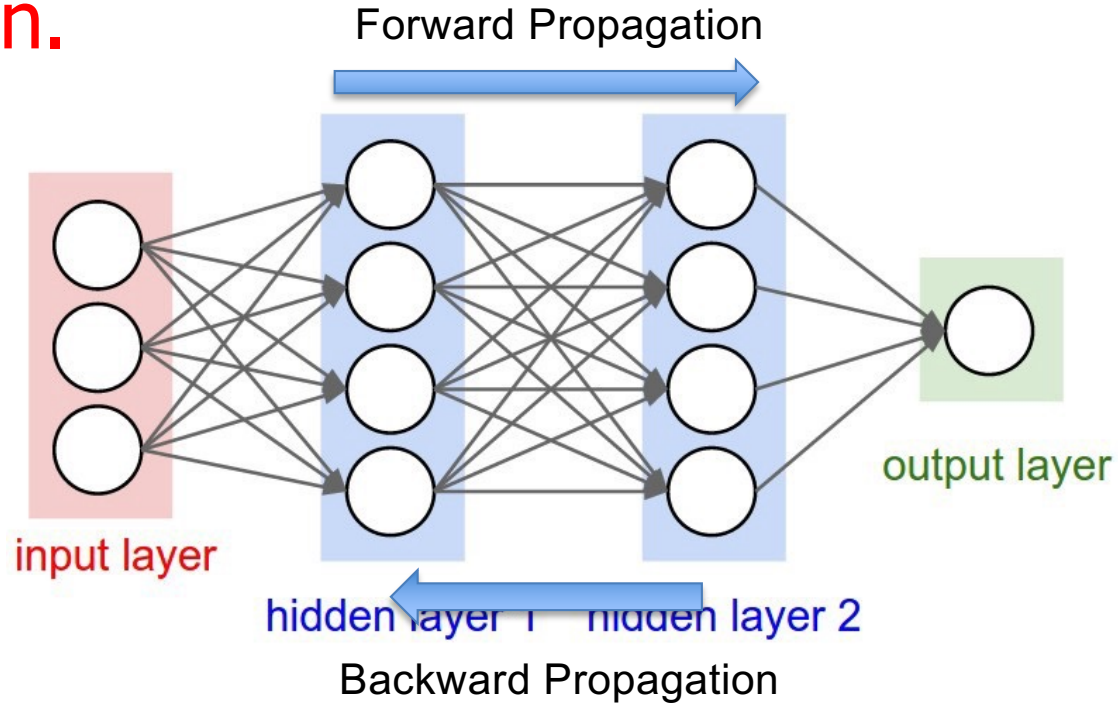
- All zero Initialization
- Initialization with small random numbers
- Calibrating the variances: normalize the variance of each neuron's output to 1
- Latest (best?): the variance of neurons in the network should be $2.0/n$ where n is the number of weights

Recap: Deep Learning vs. Other Method



Recap: Training a Deep Network

- Weights are learned layer by layer via unsupervised learning. Final layer is learned as a supervised_neural network.
- All weights are fine-tuned using supervised back propagation.



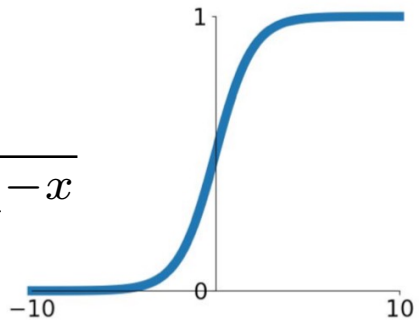
Recap: Deep Learning

- Deep Neural Network Model
  - Activation Functions
  - Hyperparameter
- Training Deep Neural Network
  - Data preprocessing
  - Initial weights
  - Optimization solvers
  - Practical Techniques

Recap Activation Function

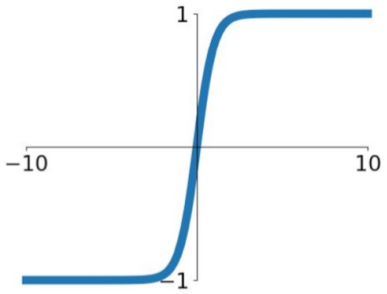
- Sigmoid

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



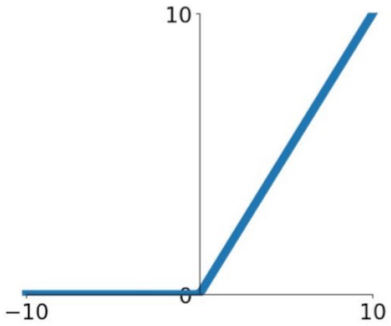
- Tanh

$$\tanh(x)$$



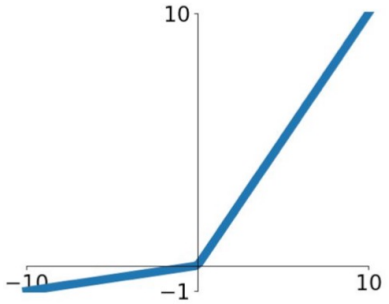
- ReLU

$$\max(0, x)$$



- Leaky ReLU

$$\max(0.1x, x)$$



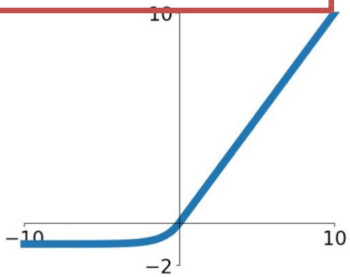
- Maxout

Normally take longer training time, due to double parameters.

$$\max(\omega_1^T x + b_1, \omega_2^T x + b_2)$$

- ELU - Exponential Linear Unit

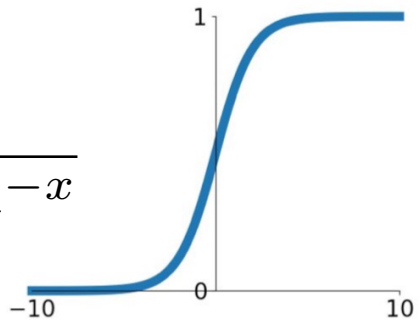
$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Recap Activation Function

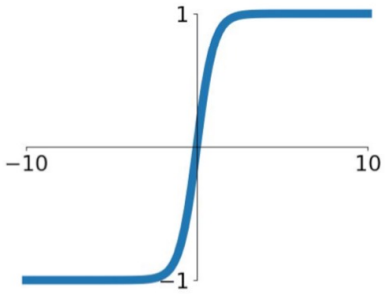
- Sigmoid

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



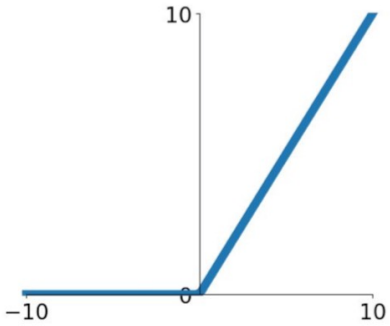
- Tanh

$$\tanh(x)$$



- ReLU

$$\max(0, x)$$

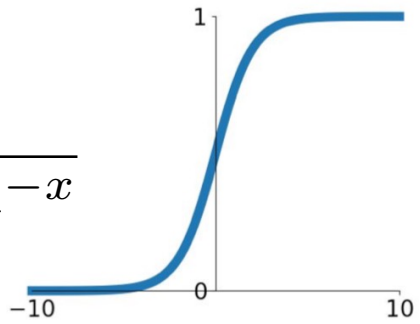


- [0,1] good for modeling probability
- Usually used in the last layer (output layer)
- Sigmoid saturates and kill gradients
- Sigmoid outputs are not zero-centered
- exp() computation expensive

Recap Activation Function

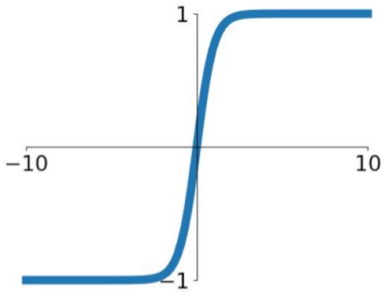
- Sigmoid

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



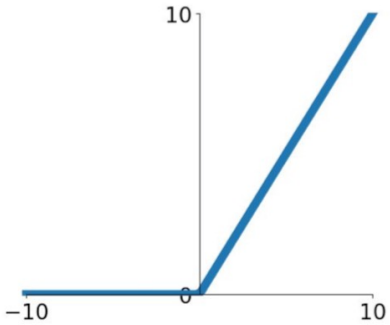
- Tanh

$$\tanh(x)$$



- ReLU

$$\max(0, x)$$

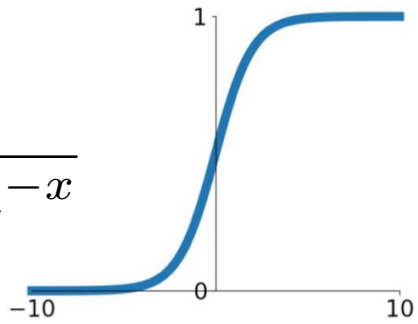


- Range [-1, 1], good for classification problems.
- Usually used in the last layer (output layer)
- Tanh(x) saturate and kill gradients

Recap Activation Function

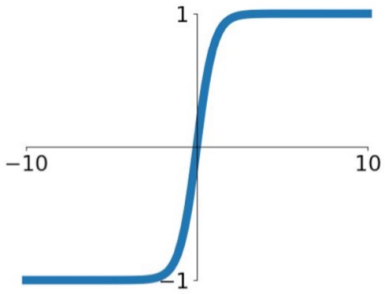
- Sigmoid

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



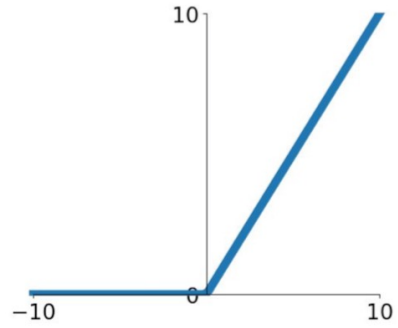
- Tanh

$$\tanh(x)$$



- ReLU

$$\max(0, x)$$



- The most used activation function
- Easy to implement and computation efficient
- Without saturating in +region
- Greatly accelerate the convergence of stochastic gradient descent (much faster than sigmoid/tanh)
- Negative values below 0
- Die ReLU problem:
  - A "dead" ReLU always outputs the same value (zero as it happens, but that is not important) for any input.
- Try Leaky ReLU or ELU

Recall General Rules for Activation Functions

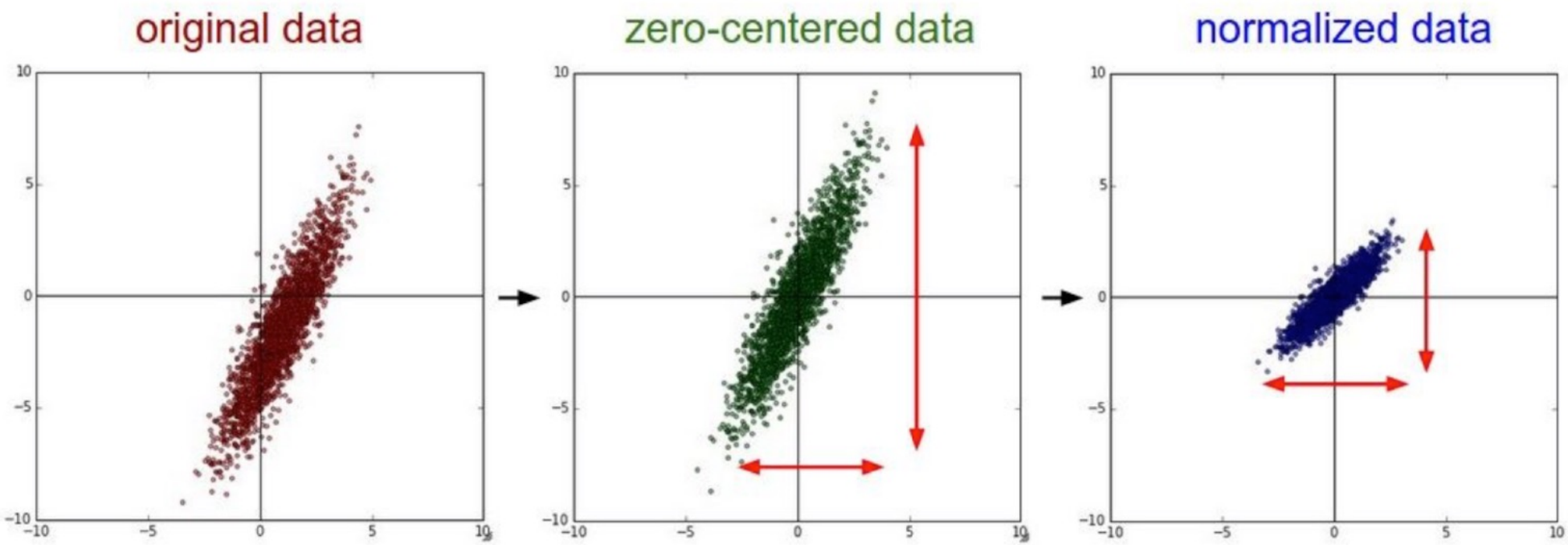
- Use ReLU. Choose a proper learning rate.
- Try out Leaky ReLU / Maxout / ELU
- Try out tanh but don’t expect much
- Don’t use sigmoid in hidden layers

Recap: Hyperparameters to Optimize?

- Network architecture
- Learning rate; its decay schedule; update type
- Regularization (L2/Dropout strength)

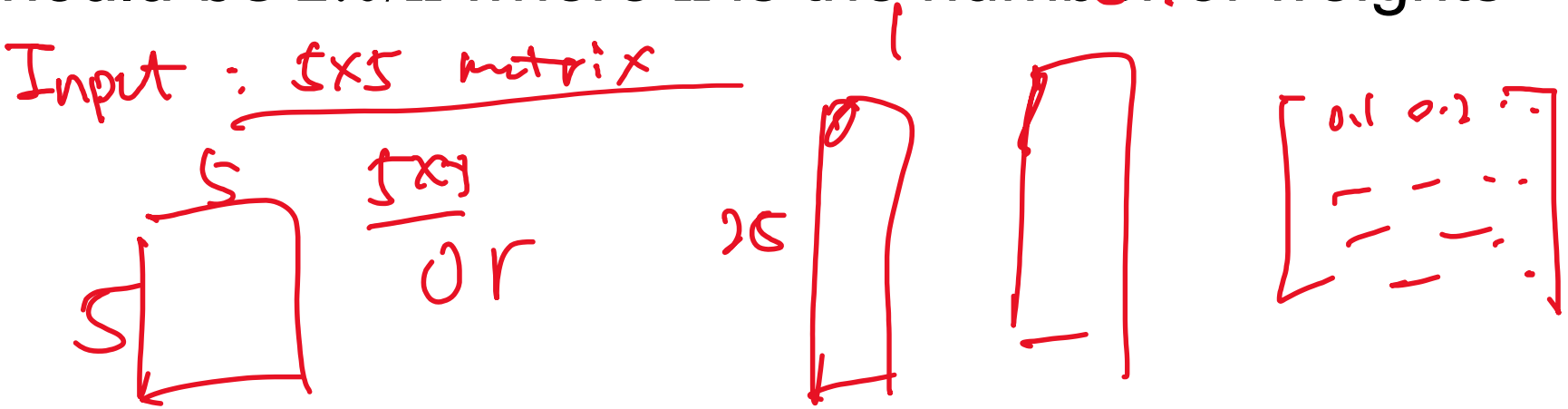
Data Preprocessing: Recommended

- Data preprocessing is a must!
- Zero-center: put the mean at zero. Why?
- Normalize: assume data follow normal distribution



Recap: How to Initialize the Weights?

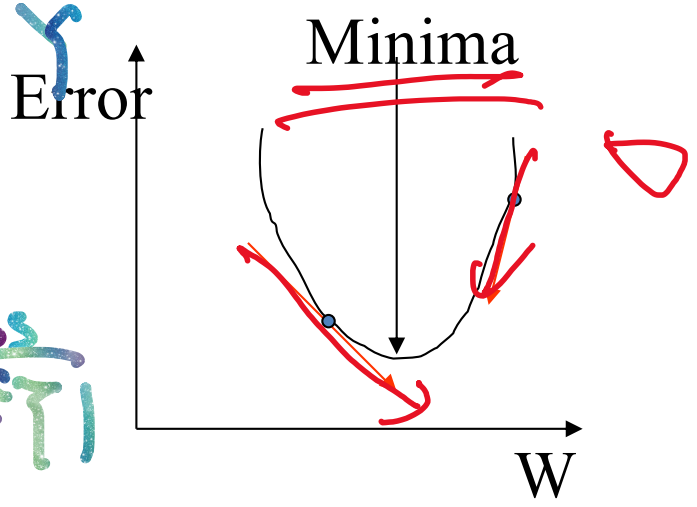
- All zero Initialization
- **Initialization with small random numbers**
- Calibrating the variances: normalize the variance of each neuron's output to 1
- Latest (best?): the variance of neurons in the network should be $2.0/n$ where n is the number of weights

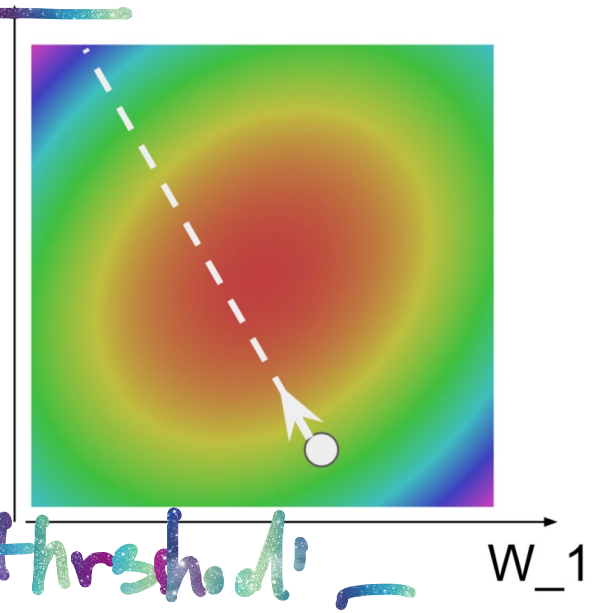


Optimization Solver

Gradient Descent Optimization Algorithms

- **Initialize** weights *w*
- **Repeat**
  - For each data point *x*, do the following:
  - Forward propagation: compute outputs and activations
  - Backward propagation: compute errors for each output units and hidden units. Compute gradient for each weight.
  - Update weight $w \mathrel{*}= w - \eta\,(\partial E / \partial w)$
- **Until** a number of iterations or errors drops below a threshold.

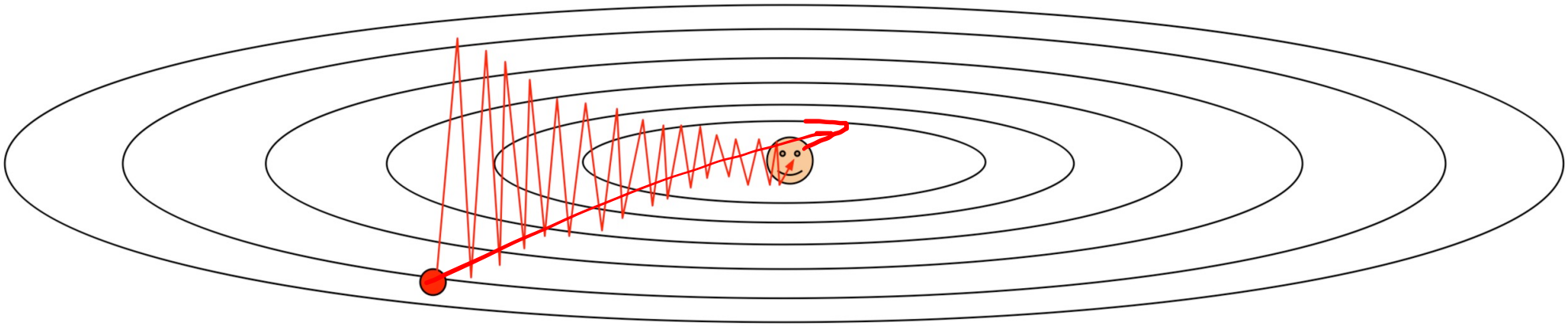




Problems with Gradient Descent

What if loss changes quickly in one direction and slowly in another?
What does gradient descent do?

Very slow progress along shallow dimension, jitter along steep direction

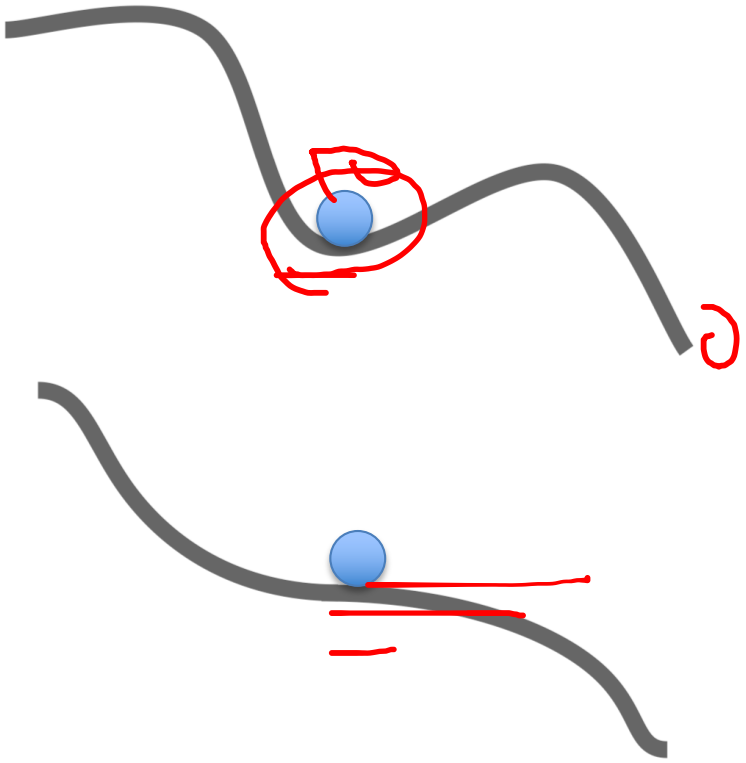


Loss function has high condition number: ratio of largest to smallest singular value of the Hessian matrix is large

Problems with Gradient Descent

- What if the loss function has a **local minima** or **saddle point**?
- Saddle points much more common in high dimension

Zero gradient
gradient descent gets
stuck



Challenges

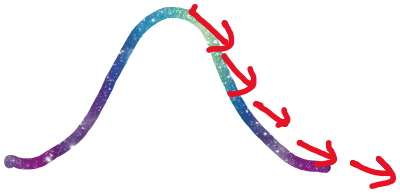
- Choosing a proper **learning rate** can be difficult.
  - A learning rate that is too small leads to painfully slow convergence
  - A learning rate that is too large can hinder convergence and cause the loss function to fluctuate around the minimum or even to diverge.
- Learning rate schedules try to adjust the learning rate during training, but it has to be determined before the training.
- The same learning rate applies to all parameter updates, but some parameters are more sensitive.
- Another key challenge of minimizing highly non-convex error functions common for neural networks is avoiding getting trapped in their numerous suboptimal local minima.

Batch Gradient Descent

- Batch gradient descent, aka Vanilla gradient descent, computes the gradient of the cost function w.r.t. to the parameters for the entire training dataset

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta)$$

- As we need to calculate the gradients for the whole dataset to perform just *one* update, batch gradient descent can be very slow and is intractable for datasets that don't fit in memory.
- Batch gradient descent also doesn't allow us to update our model *online.*

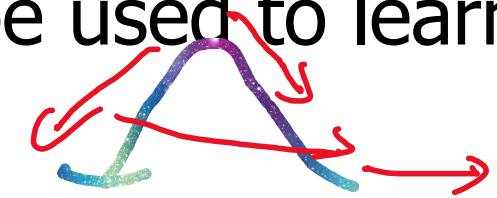


Stochastic Gradient Descent

- Stochastic gradient descent (SGD) in contrast performs a parameter update for *each* training example

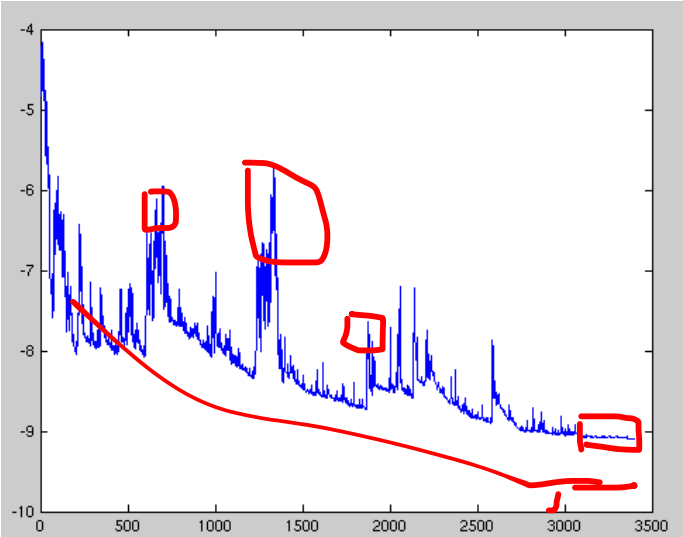
$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i)}; y^{(i)})$$

- Batch gradient descent performs redundant computations for large datasets, as it recomputes gradients for similar examples before each parameter update.
- SGD does away with this redundancy by performing one update at a time. It is therefore usually much faster and can also be used to learn online.



Stochastic Gradient Descent

- SGD performs frequent updates with a high variance that cause the objective function to fluctuate heavily.



- + SGD's fluctuation, on the one hand, enables it to jump to new and potentially better local minima.
- - this ultimately complicates convergence to the exact minimum, as SGD will keep overshooting

Mini-Batch Gradient Descent

- Mini-batch gradient descent finally takes the best of both worlds and performs an update for every mini-batch of n training examples

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)})$$

n = selected data.

$1 < n < \text{Total}$

- + Reduces the variance of the parameter updates, which can lead to more stable convergence
- + can make use of highly optimized matrix optimizations common to state-of-the-art deep learning libraries
- Common mini-batch sizes range between 50 and 256, but can vary for different applications

Stochastic Gradient Decent + Momentum

- SGD has trouble navigating ravines, i.e. areas where the surface curves much more steeply in one dimension than in another, which are common around local optima.
- Momentum is a method that helps accelerate SGD in the relevant direction and dampens oscillations.

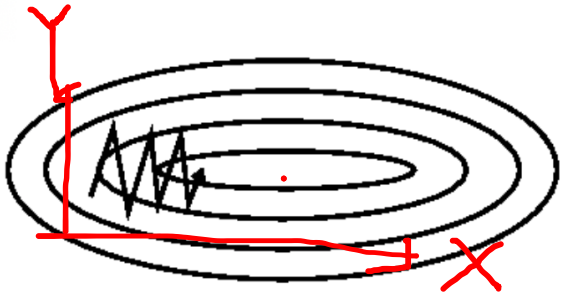


Image 2: SGD without momentum

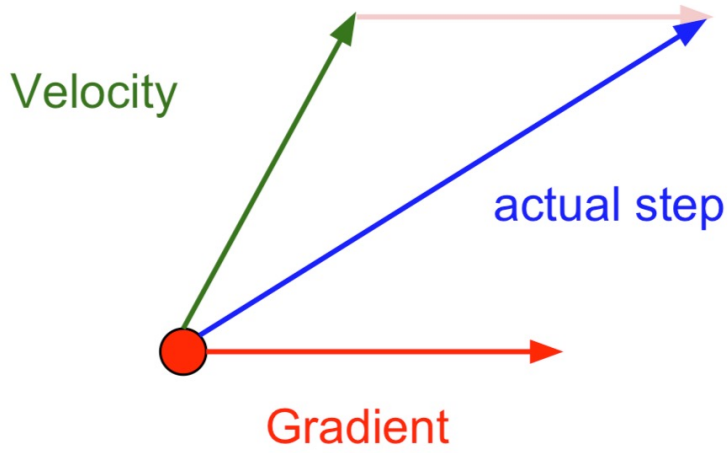
Image 3: SGD with momentum

SGD+ Momentum

- It does this by adding a fraction of the update vector of the past time step to the current update vector.

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$$
$$\theta = \theta - v_t$$

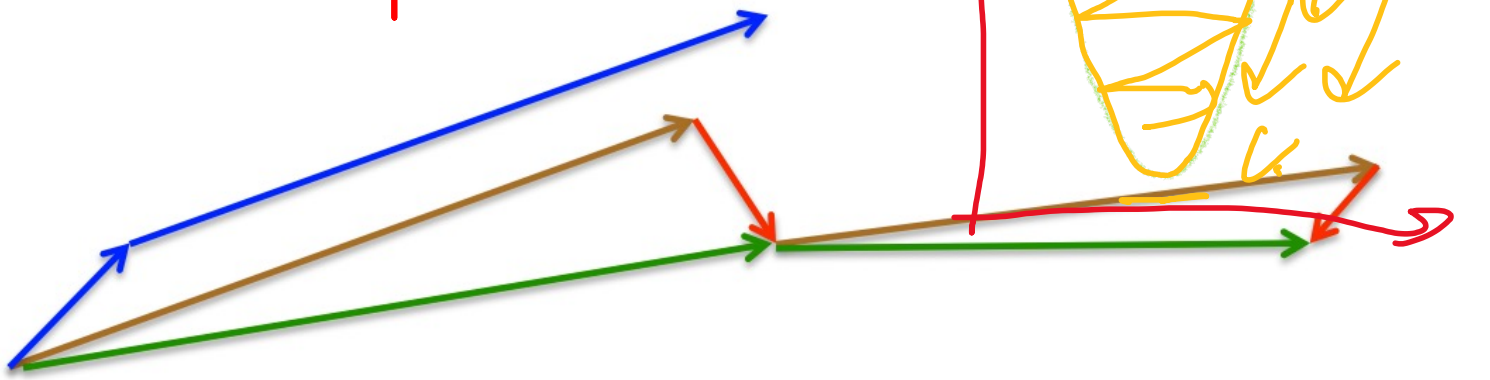
- Some implementations exchange the signs in the equations. The momentum term is usually set to 0.9 or a similar value.



Nesterov Accelerated Gradient (NAG)

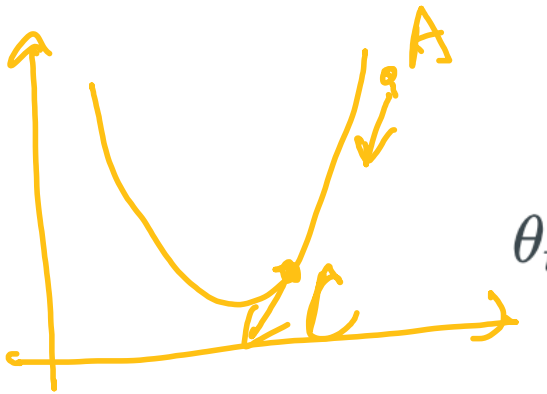
- Nesterov accelerated gradient (NAG): use an approximation of the next position of the parameters (the gradient is missing for the full update), a rough idea where our parameters are going to be.

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1})$$
$$\theta = \theta - v_t$$



Adagrad

- It adapts the learning rate to the parameters, performing smaller updates for parameters associated with frequently occurring features, and larger updates for parameters associated with infrequent features.
- It is well-suited for dealing with sparse data.



$$g_{t,i} = \nabla_\theta J(\theta_{t,i})$$

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i}$$

where G_t is a diagonal matrix where each diagonal element is the sum of the squares of the gradients w.r.t. θ_i

RMSprop (Root Mean Squared Propagation)

- RMSprop as well divides the learning rate by an exponentially decaying average of squared gradients

$$E[g^2]_t = 0.9E[g^2]_{t-1} + 0.1g_t^2$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

Adaptive Moment Estimation (Adam)

- The decaying averages of past and past squared gradients

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

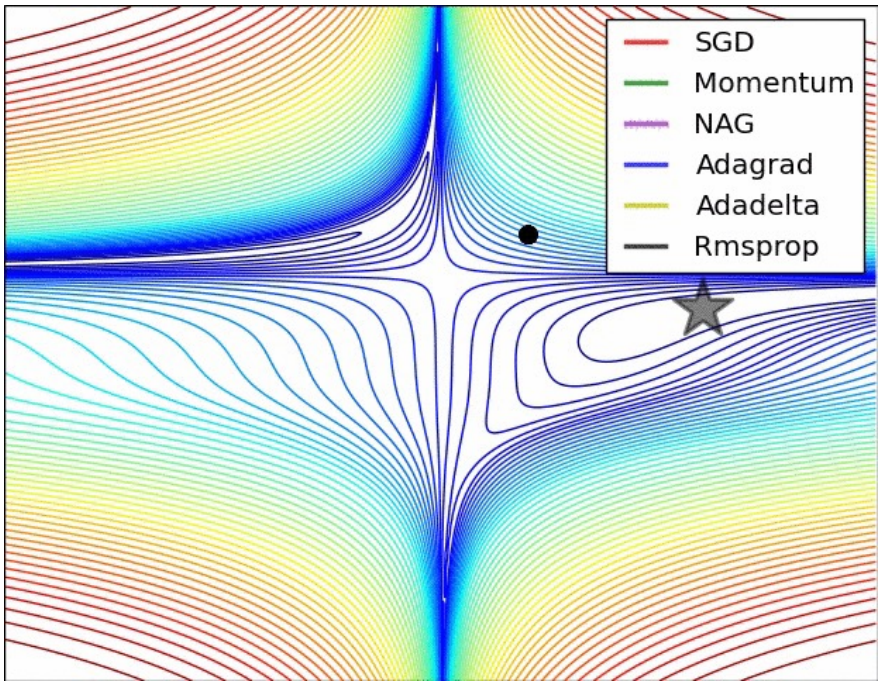
  where $\mathrm{m_t}$ and $\mathrm{v_t}$ are estimates of the first moment (the mean) and the second moment (the uncentered variance) of the gradients respectively

- Bias-corrected first and second moment, and update

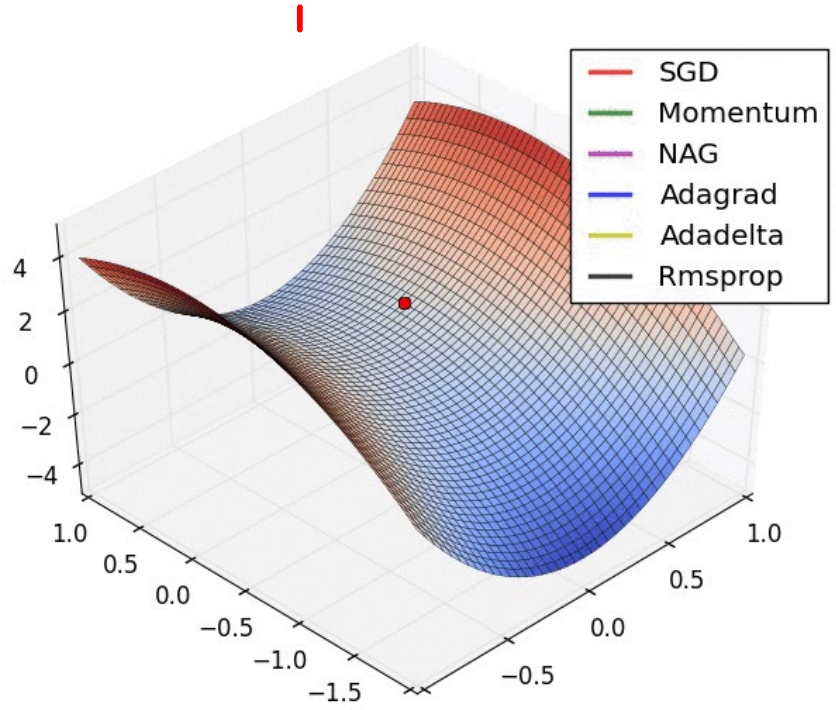
$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$
$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}} \hat{m}_t$$

Visualization of Algorithms



contours of a loss surface



a saddle point

Adadelta is a more robust extension of Adagrad that adapts learning rates based on a moving window of gradient updates, instead of accumulating all past gradients.

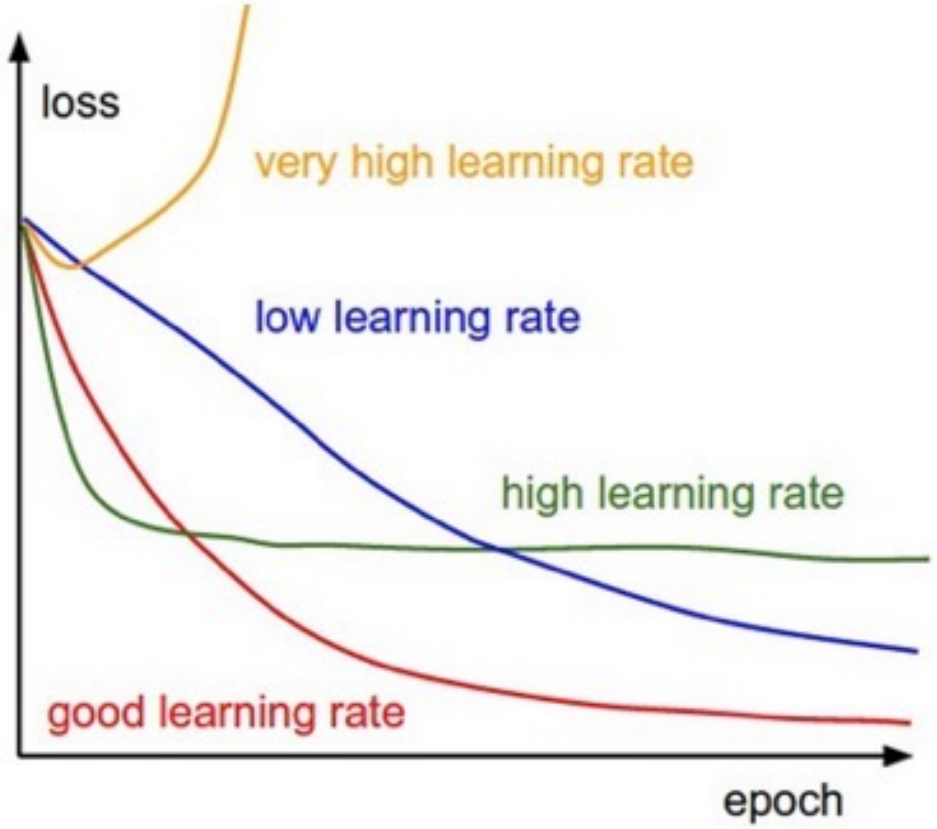
What Optimizer to Use?

- If your input data is sparse, then you likely achieve the best results using one of the adaptive learning-rate methods (no need to tune learning rates). Sparse?
- Insofar, RMSprop, Adadelta, and Adam are very similar algorithms that do well in similar circumstances. Adam might be the best overall choice.
- If you care about fast convergence and train a deep or complex neural network, you should choose one of the adaptive learning rate methods.

Practical Issues

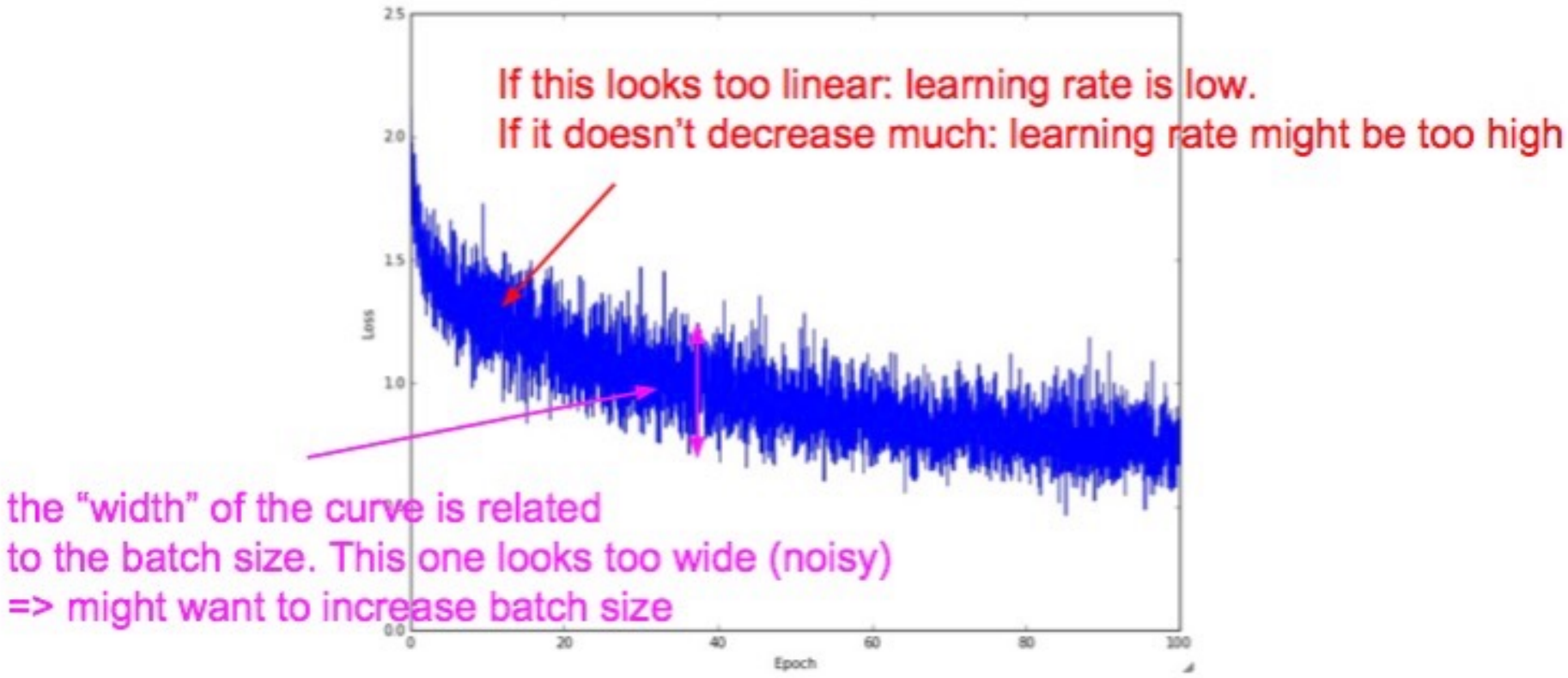
What Does the Training Process Tell Us?

- Learning rate



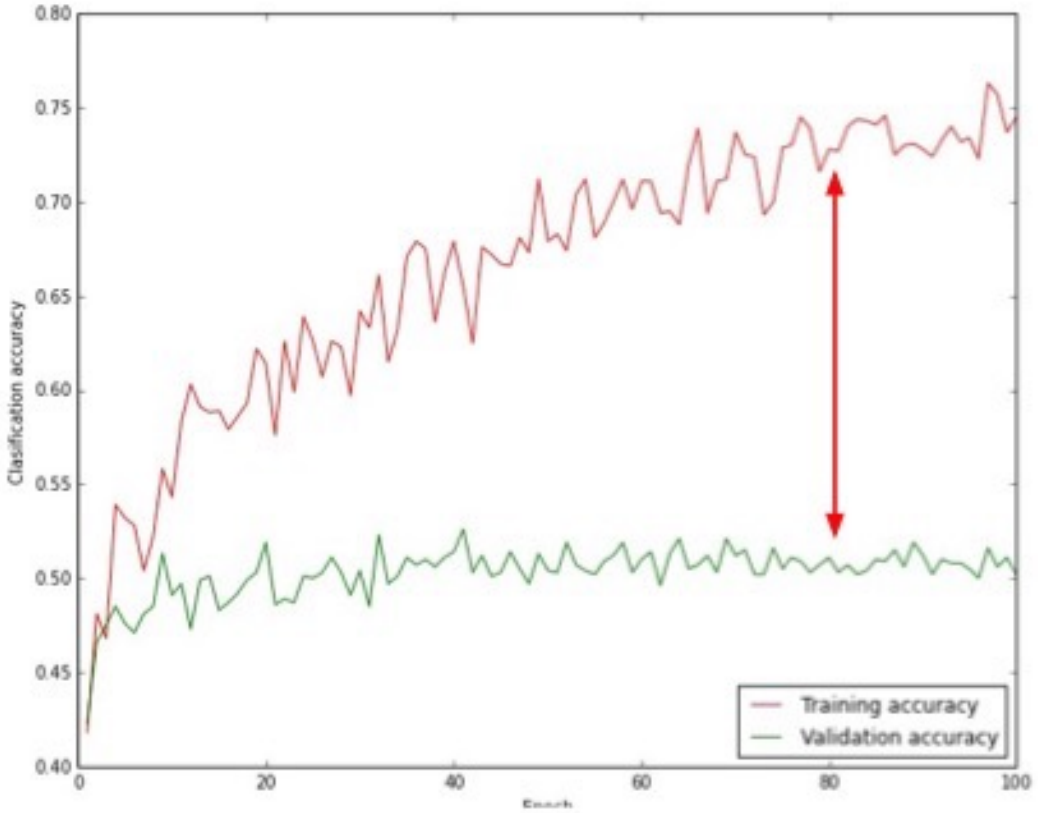
What Does the Training Process Tell Us?

- Loss



What Does the Training Process Tell Us?

- Accuracy curve



big gap = overfitting

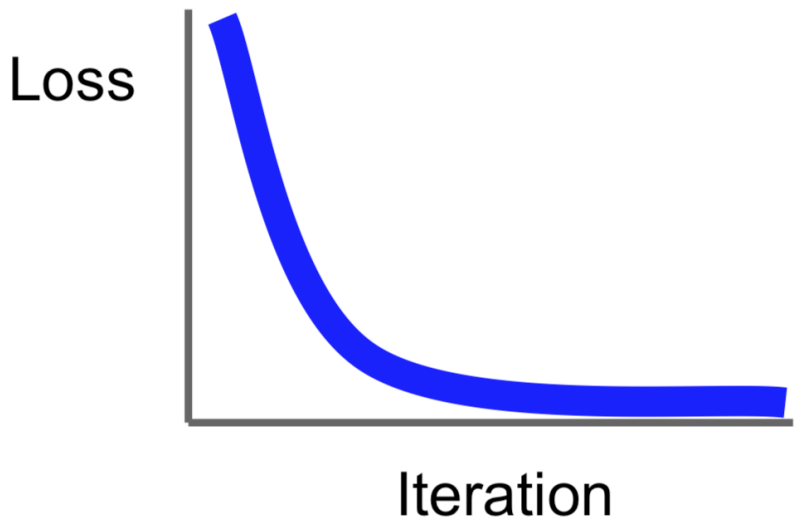
=> increase regularization strength

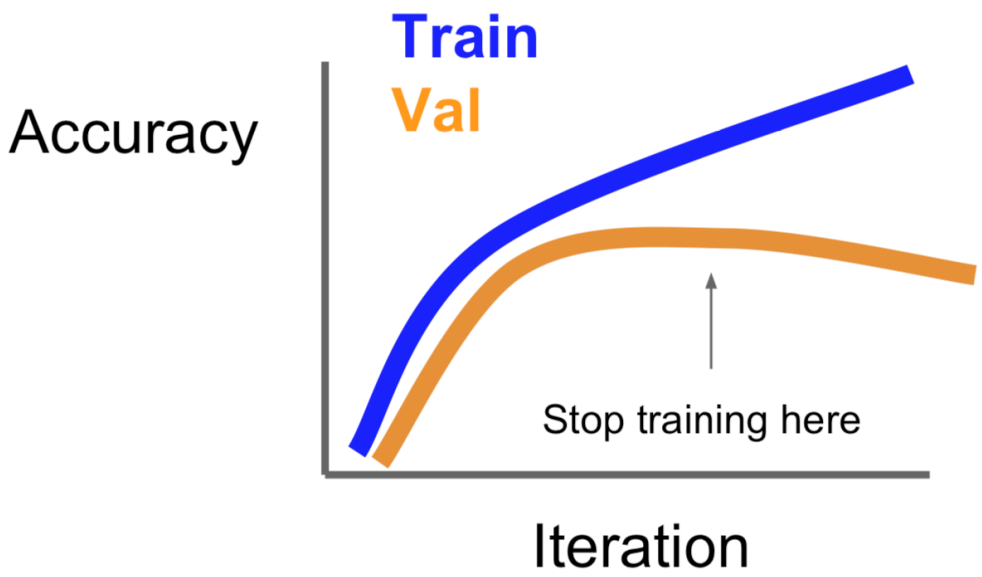
no gap

=> increase model capacity

Reduce the Error on New Data

- Stop training the model when accuracy on the validation set decreases
- Or train for a long time, but always keep track of the model snapshot that worked best on validation





Model Ensembles

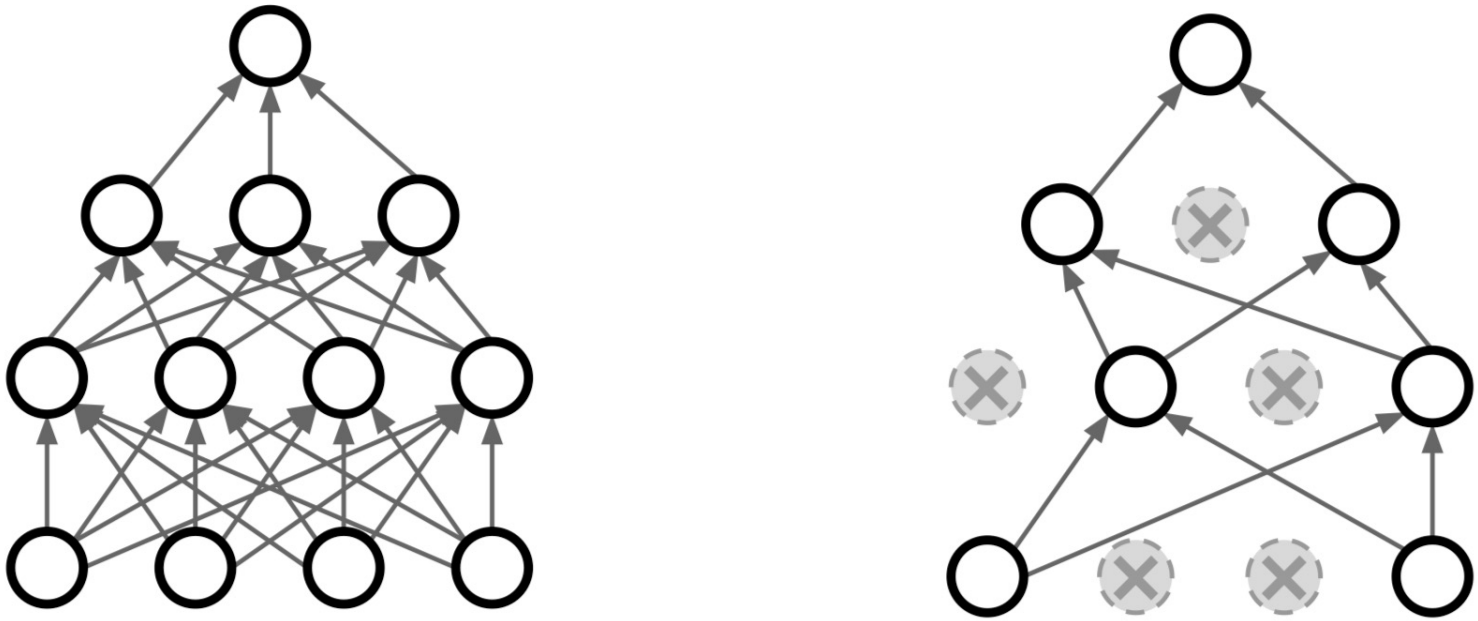
- Train multiple independent models
- At test time average their results (average of predicted probability distributions then argmax)
- Usually 2% extra performance
- Instead of training independent models, use multiple snapshots of a single model during training.
- Instead of using actual parameter vector, keep a moving average of the parameter vector and use that as test time.

How to Avoid Overfitting?

- Regularization
  - L2 regularization
  - L1 regularization
  - Max norm
- Dropout
  - During training, dropout samples a Neural Network within the full Neural Network, and only updates the parameters of the sampled network based on the input data.
  - During testing there is no dropout applied
  - Extremely effective, simple; use it!
- Max Pooling
  - Take the local maximum to represent all

Regularization - Dropout

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



Why Dropout Works?

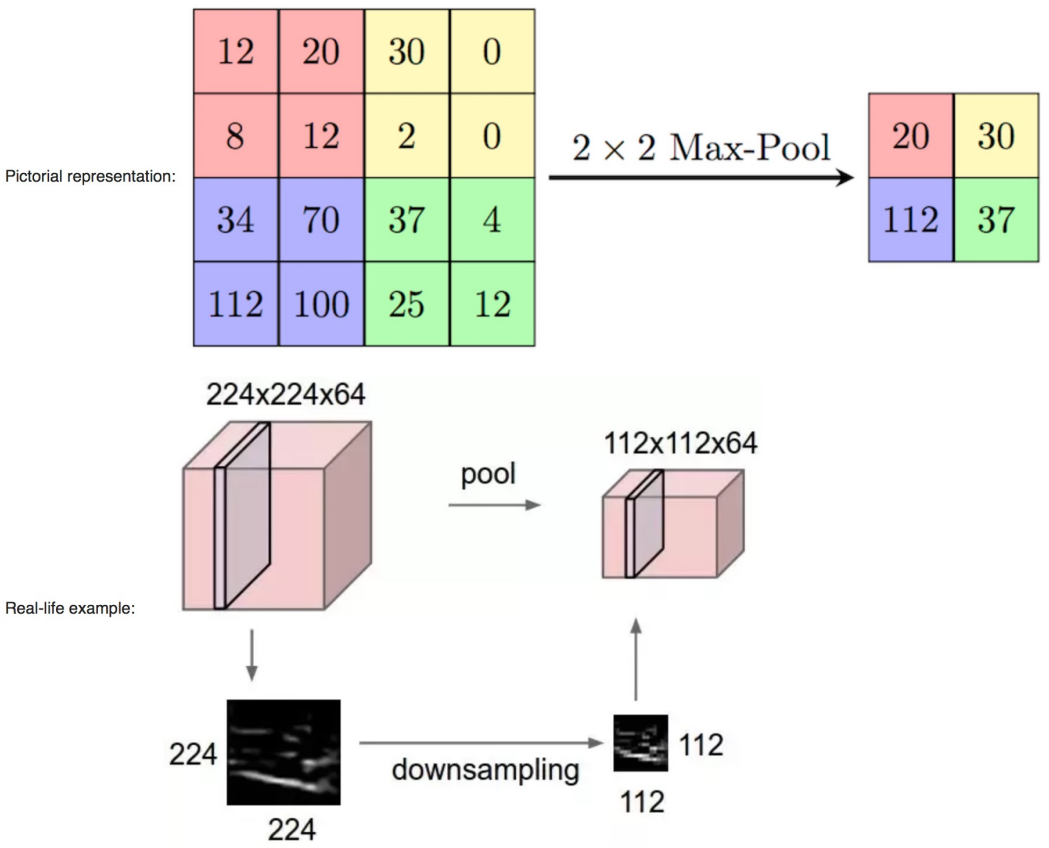
- Forces the network to have a redundant representation
- Prevents co-adaptation of features

Dropout at Test Time

- 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
  - Drop in training -> scale at test time
  - Drop in training and scaled -> test time is unchanged

Max-Pooling

- Max pooling is a **sample-based discretization process**.
- The objective is to down-sample an input representation (image, hidden-layer output matrix, etc.), reducing its dimensionality and allowing for assumptions to be made about features contained in the sub-regions binned.



What if We Don't Have Many Data?

- Image data augmentation
  - Flipping, crop, and rotation
  - Combination: translation, rotation, stretching, shearing, lens distortions
  - Color alternation
- Other data?

Training Accuracy Decreases

- Reduce your learning rate to a very small number like 0.001 or even 0.0001.
- Provide more data.
- Set Dropout rates to a number like 0.2. Keep them uniform across the network.
- Try decreasing the batch size.
- Using appropriate optimizer: You may need to experiment a bit on this. Use different optimizers on the same network, and select an optimizer which gives you the least loss.
- Accuracy curve

Summary

- Deep neural network model
- Model training
- Practical issues