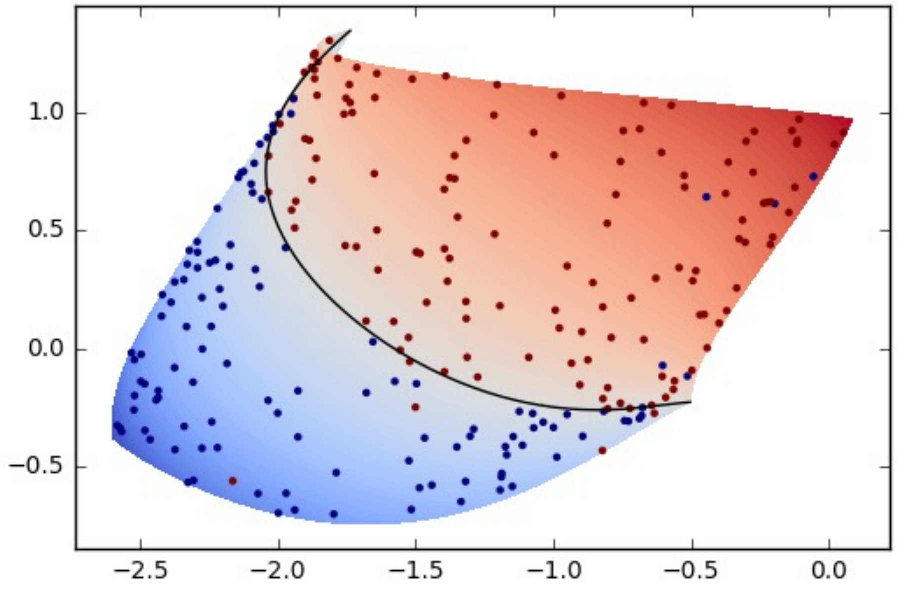
Lecture 14: Neural Networks Part 3



Gavin Kerrigan

Spring 2023

Some materials courtesy Padhraic Smyth, Alex Ihler.

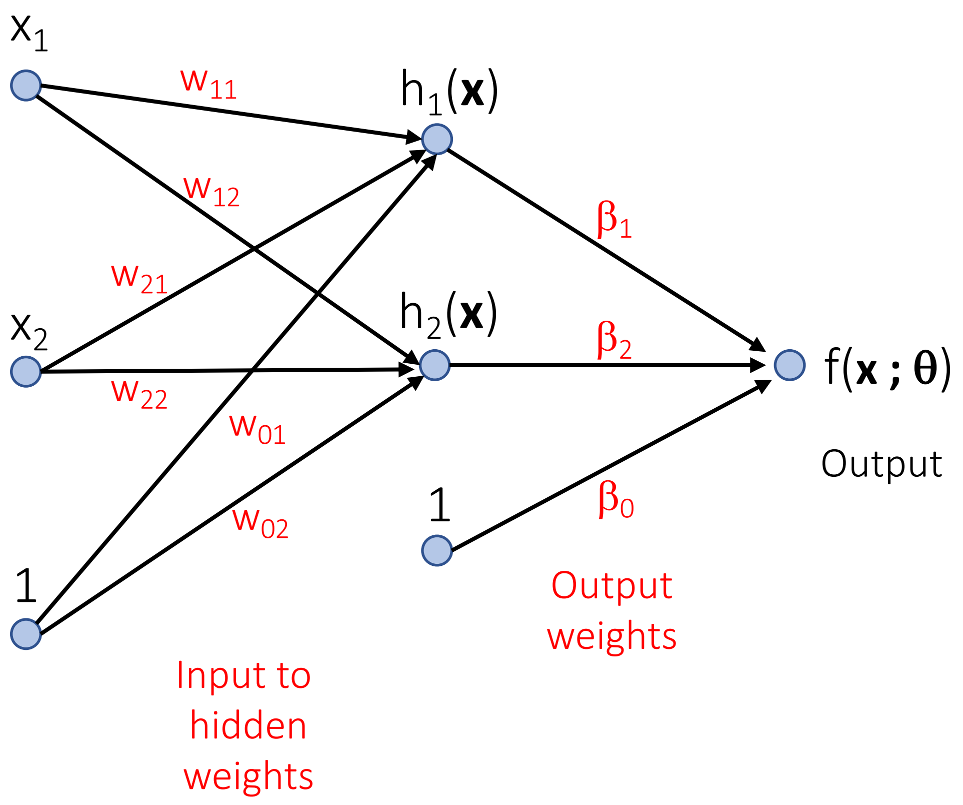
Announcements

- Midterm Friday
  - Recommendation: read entire exam before attempting
  - Typo in sample exam – kNN Problem 3
  - Updated version on Canvas
- Homework 2 grades posted
  - Mean score: 89%
  - Regrade requests available until Weds. 5/10
- Homework 3 – due in ~2 weeks (Friday 5/12)
  - Available now
  - Shorter HW due to exam
  - Focused on neural networks

More on Neural Networks

Training Neural Networks

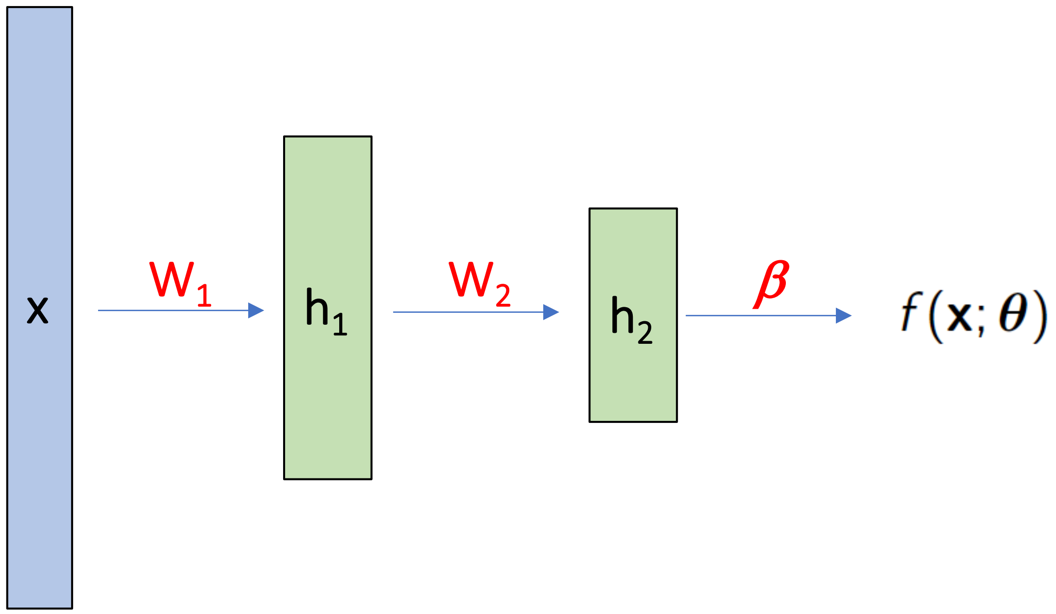
Example of a Neural Network with 1 Hidden Layer



In general, to compute values of nodes:

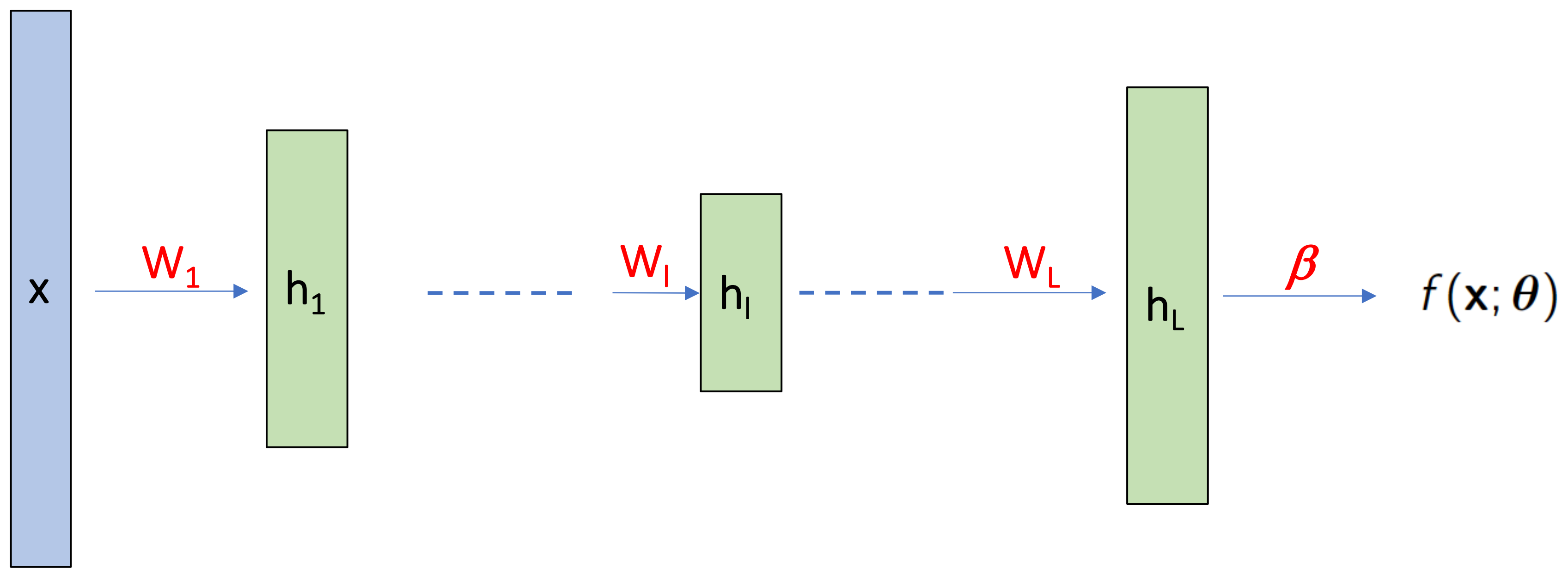
1. Take weighted linear combination of features
    - Each node has its own weights (on edges)
2. Apply nonlinearity (e.g. sigmoid)

Networks with Two Hidden Layers



Each hidden unit layer can have different numbers of hidden units

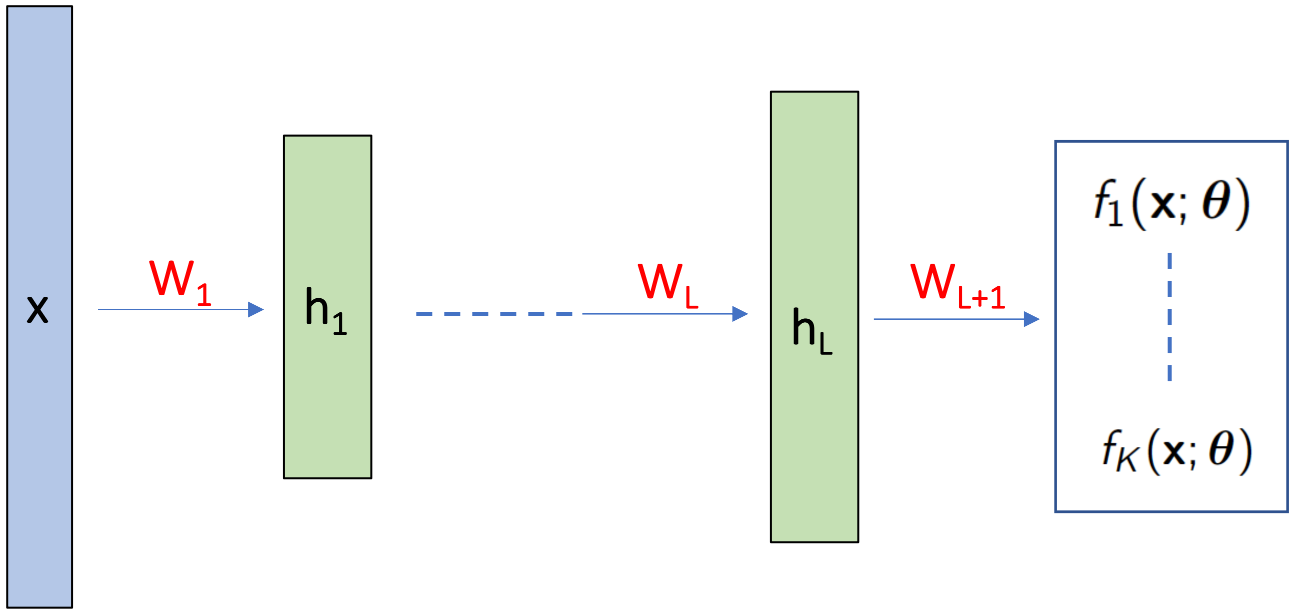
Networks with L Hidden Layers



The network can have an arbitrary number of layers, each with an arbitrary number of hidden units

Networks with multiple hidden layers are referred to as "deep"

Networks with Multiple Outputs


x
W_1
h_1
W_L
h_L
W_{L+1}
$f_1(\mathbf{x}; \boldsymbol{\theta})$
$f_K(\mathbf{x}; \boldsymbol{\theta})$

K different outputs, normalized by softmax function to sum to 1
(same softmax function as for K-ary logistic classifier)

Can interpret kth output as P(y = k | x), i.e., probability of class k

Notation warning:
We are using K here to denote the number of classes (instead of C)

Equations for General Network

$$\mathbf{h}_1 = g(\mathbf{W}_1\mathbf{x} + \mathbf{w}_{10})$$

Computation of vector of hidden unit values in first layer

$$\mathbf{h}_l = g(\mathbf{W}_l\mathbf{h}_{l-1} + \mathbf{w}_{l0})$$

Recursive computation of vector of hidden unit values in each layer from hidden units in previous layer, l = 2,… ,L

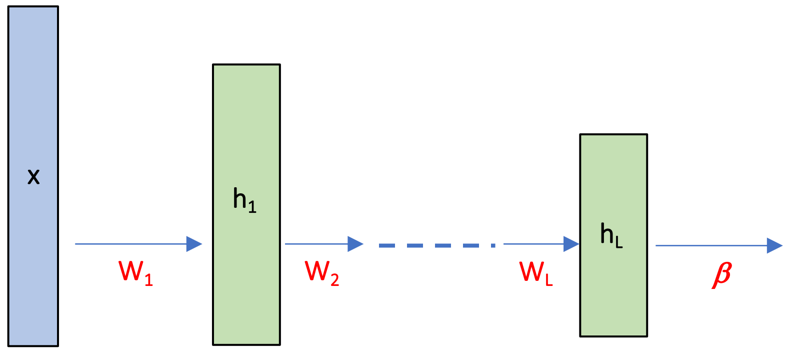
$$f_k(\mathbf{x};\boldsymbol{\theta}) = \text{softmax}\left(\boldsymbol{\beta}_k\mathbf{h} + \beta_{0k}\right)$$

Softmax over outputs to produce kth output, corresponding to P(class = k | x)

Number of Parameters in L-Layer Network

Say the network has:

- d-dimensional feature inputs
- L layers of hidden units
- K classes



Assume for simplicity that each hidden layer has M hidden units and lets ignore bias terms

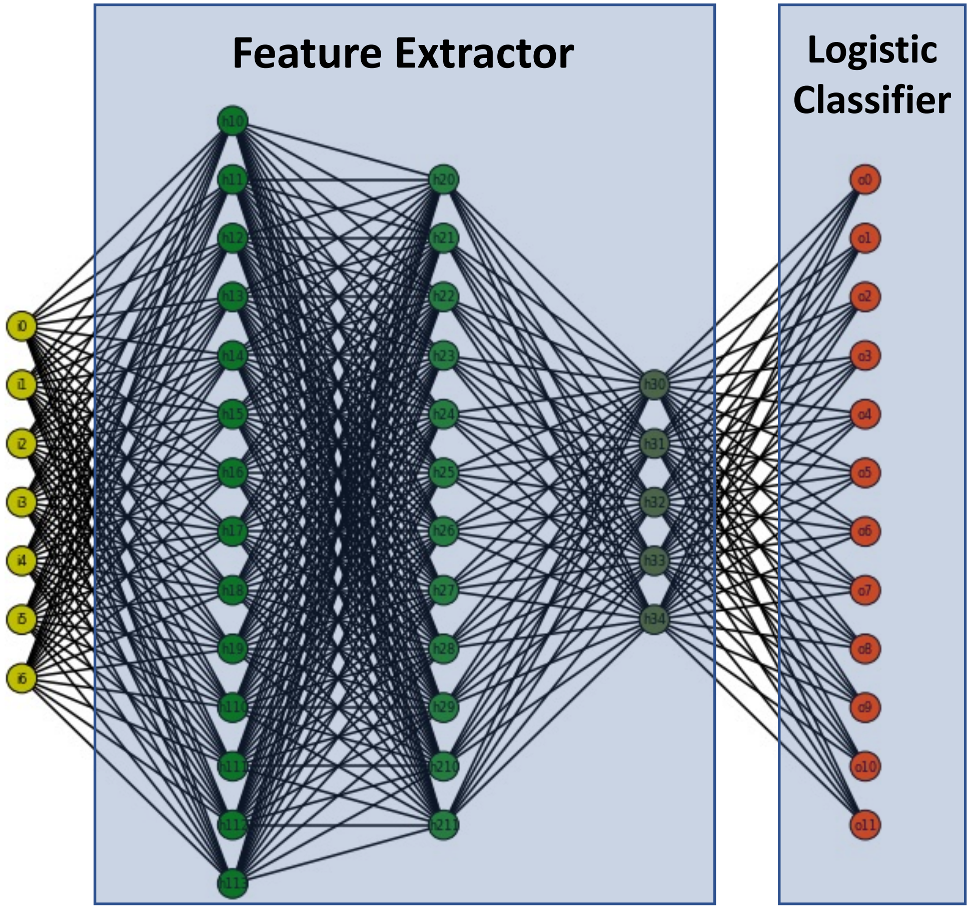
Number of parameters p is roughly: $d H + (L-1)H^2 + H K$

e.g., d = 100 x 100 = 104 pixels, H = 300, K = 1000, L = 10 layers

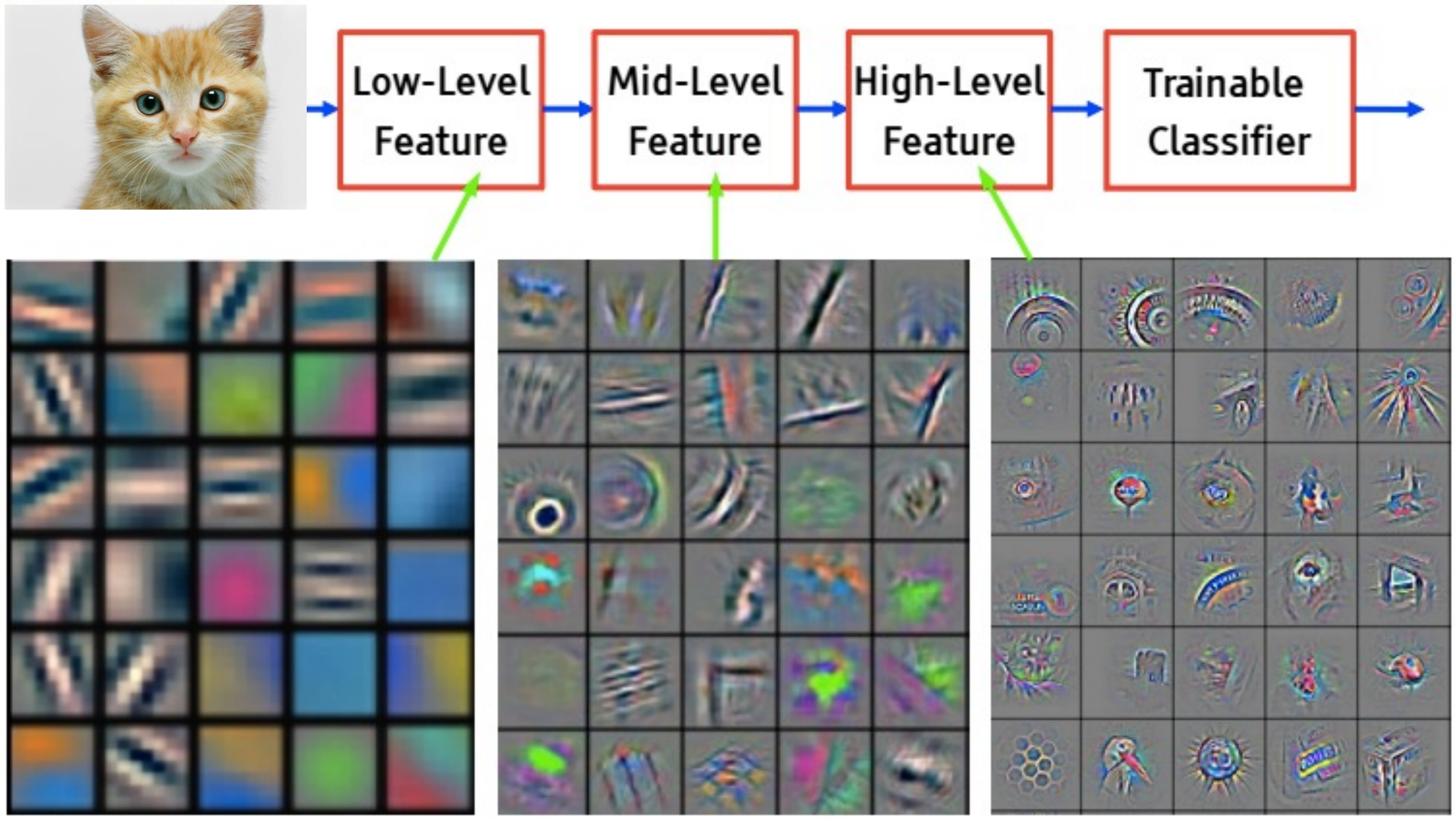
=> Number of parameters p would be about $300*10^4 + 9*(300)^2 + 300*1000$, which is approximately 4 million

Representation Learning

Hidden units at each layer in a neural network can be thought of as learning useful features (or "representations")



Hidden layers as "features"



Slide image from Yann LeCun

Feature representations learned by a deep neural network for face recognition

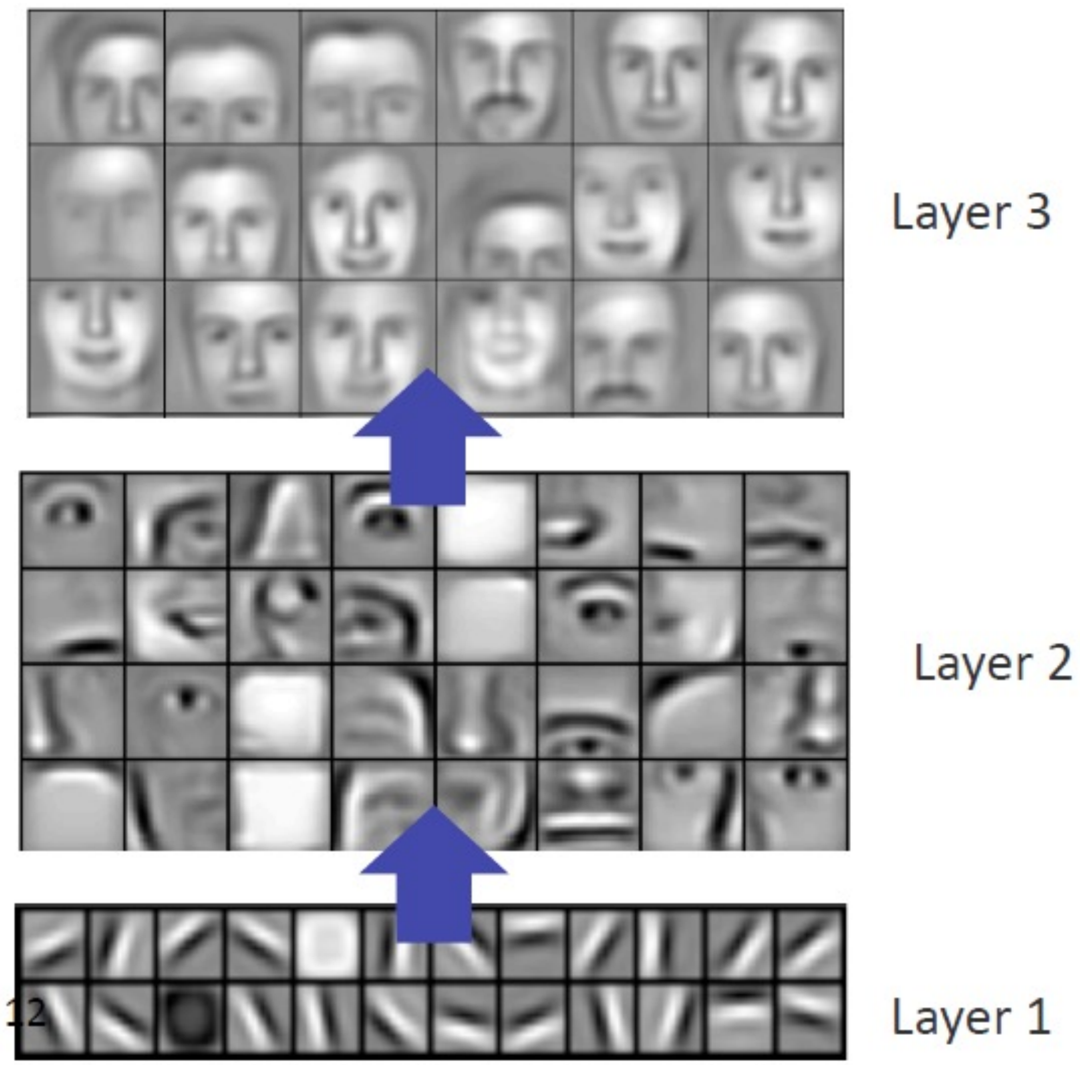


Figure from Lee et al., ICML 2009

Machine Learning before Deep Networks

Manual feature extraction step: time consuming and inefficient!

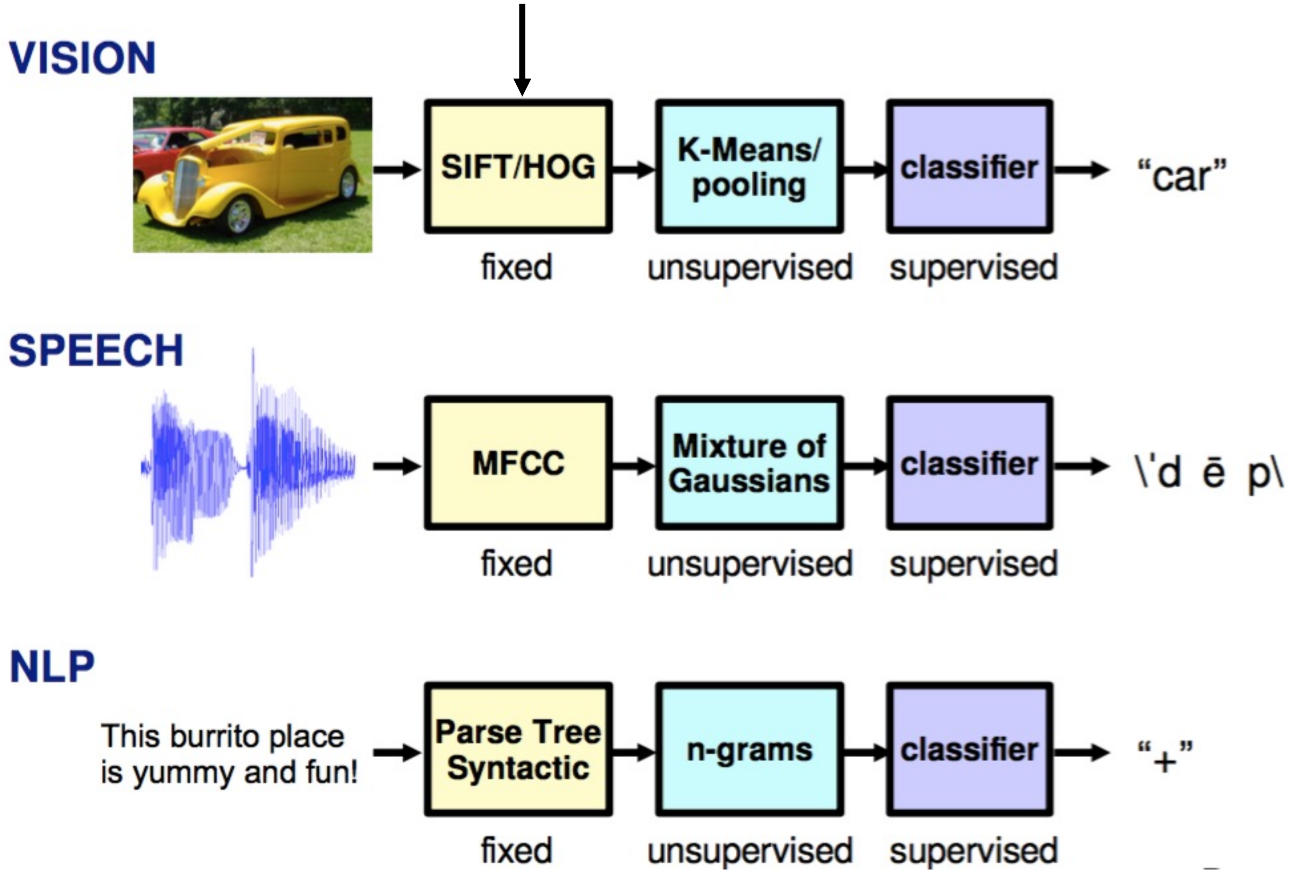
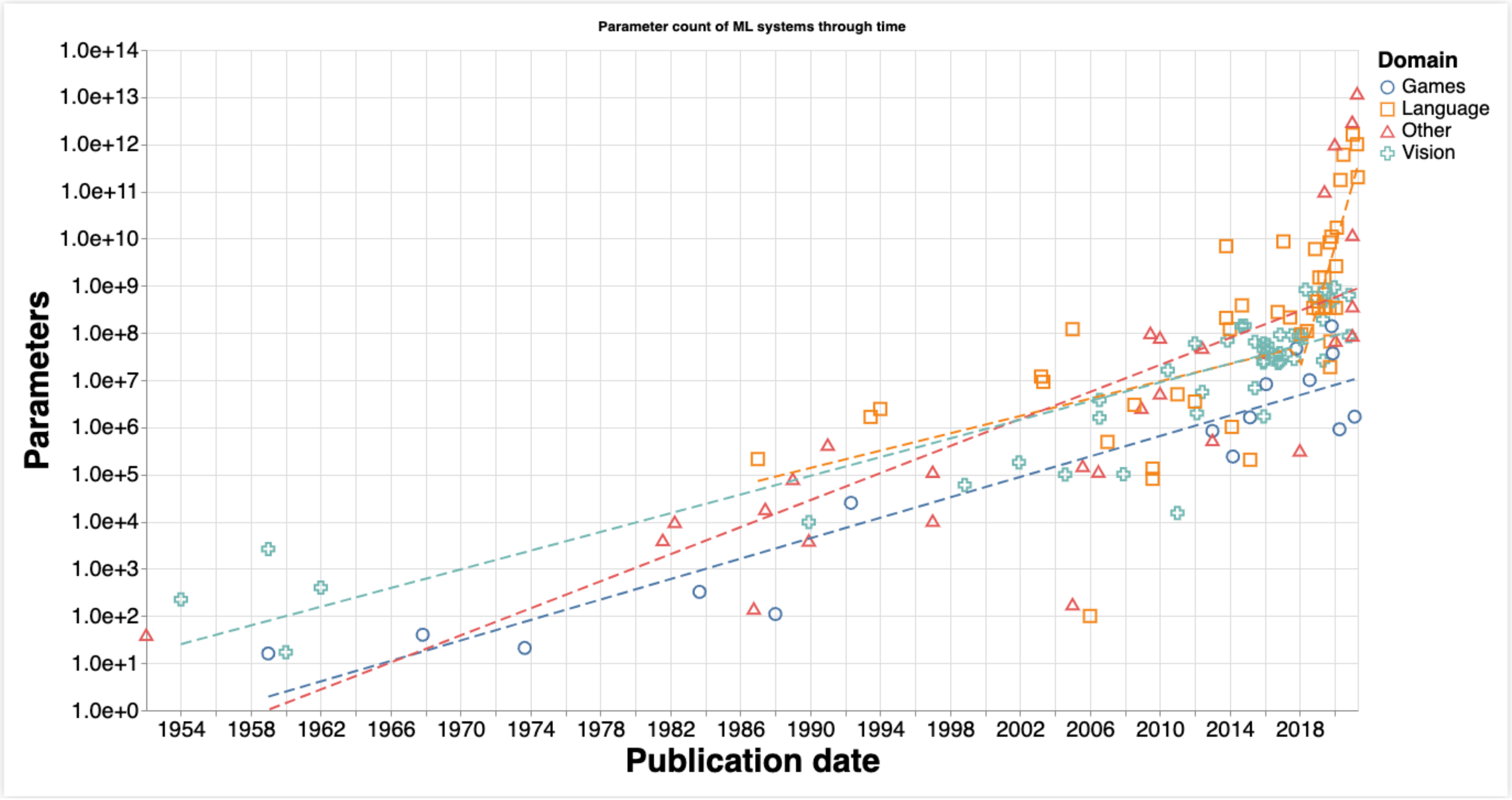


Figure from Marc'Aurelio-Ranzato

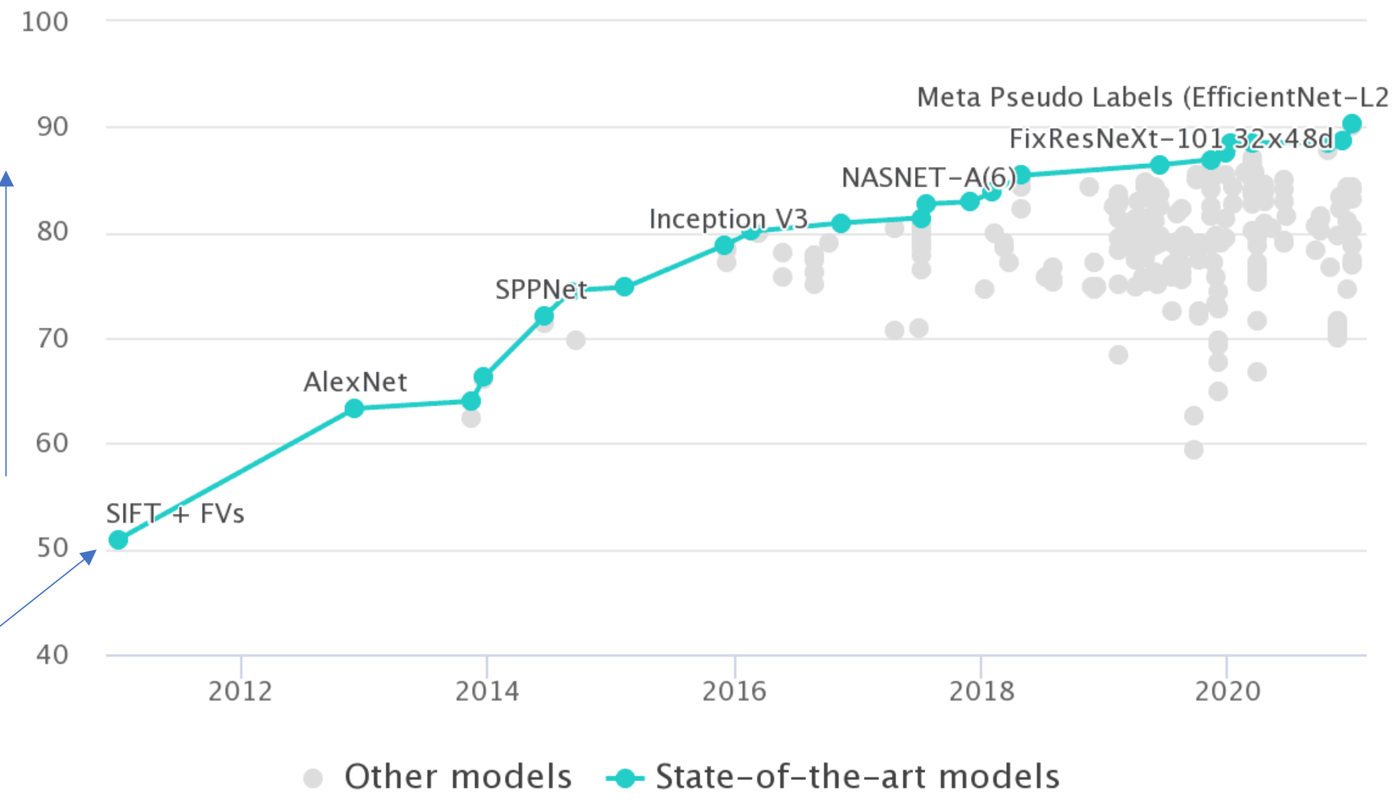
Networks are Bigger and Better



https://towardsdatascience.com/parameter-counts-in-machine-learning-a312dc4753d0

Progress in Image Classification

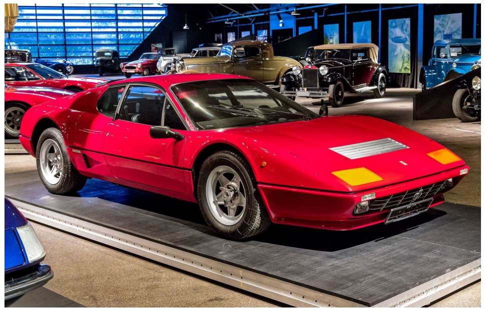
Classification accuracy on ImageNet testbed (1000 classes, millions of images)



Best model with manual features

Logistic regression

“Shallow” neural networks



Large deep neural networks

Why do we need activation functions?

With activation functions:

$$\mathbf{h}_1 = g(\mathbf{W}_1\mathbf{x} + \mathbf{w}_{10})$$

$$\begin{aligned}\mathbf{h}_2 &= g(\mathbf{W}_2\mathbf{h}_1 + \mathbf{w}_{20}) \\ &= g(\mathbf{W}_2 g(\mathbf{W}_1\mathbf{x} + \mathbf{w}_{10}) + \mathbf{w}_{20})\end{aligned}$$

$$\ldots$$

Hidden units can learn complex, non-linear functions (i.e. representations) of the data

Without activation functions:

$$\mathbf{h}_1 = \mathbf{W}_1 x + \mathbf{w}_{10}$$

$$\begin{aligned}\mathbf{h}_2 &= \mathbf{W}_2\mathbf{h}_1 + \mathbf{w}_{20} \\ &= \mathbf{W}_2\mathbf{W}_1\mathbf{x} + \mathbf{w}_{20} + \mathbf{w}_{10}\end{aligned}$$

$$\ldots$$

Hidden units are always *linear* functions of the data

- No better than having a single linear layer!

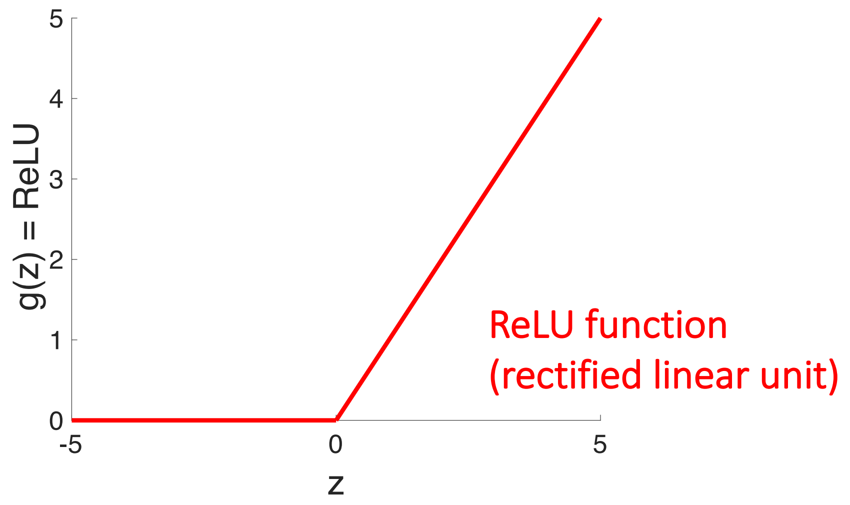
Non-Linearities: the ReLU Function

Its typical for neural network classifiers to use softmax at the output layer

In modern neural networks, it is now common to use the ReLU activation function as the g( ) function, instead of the sigmoid, for all hidden units

The ReLU function
 is very simple:

z >0 : g(z) = z
z<=0: g(z) = 0



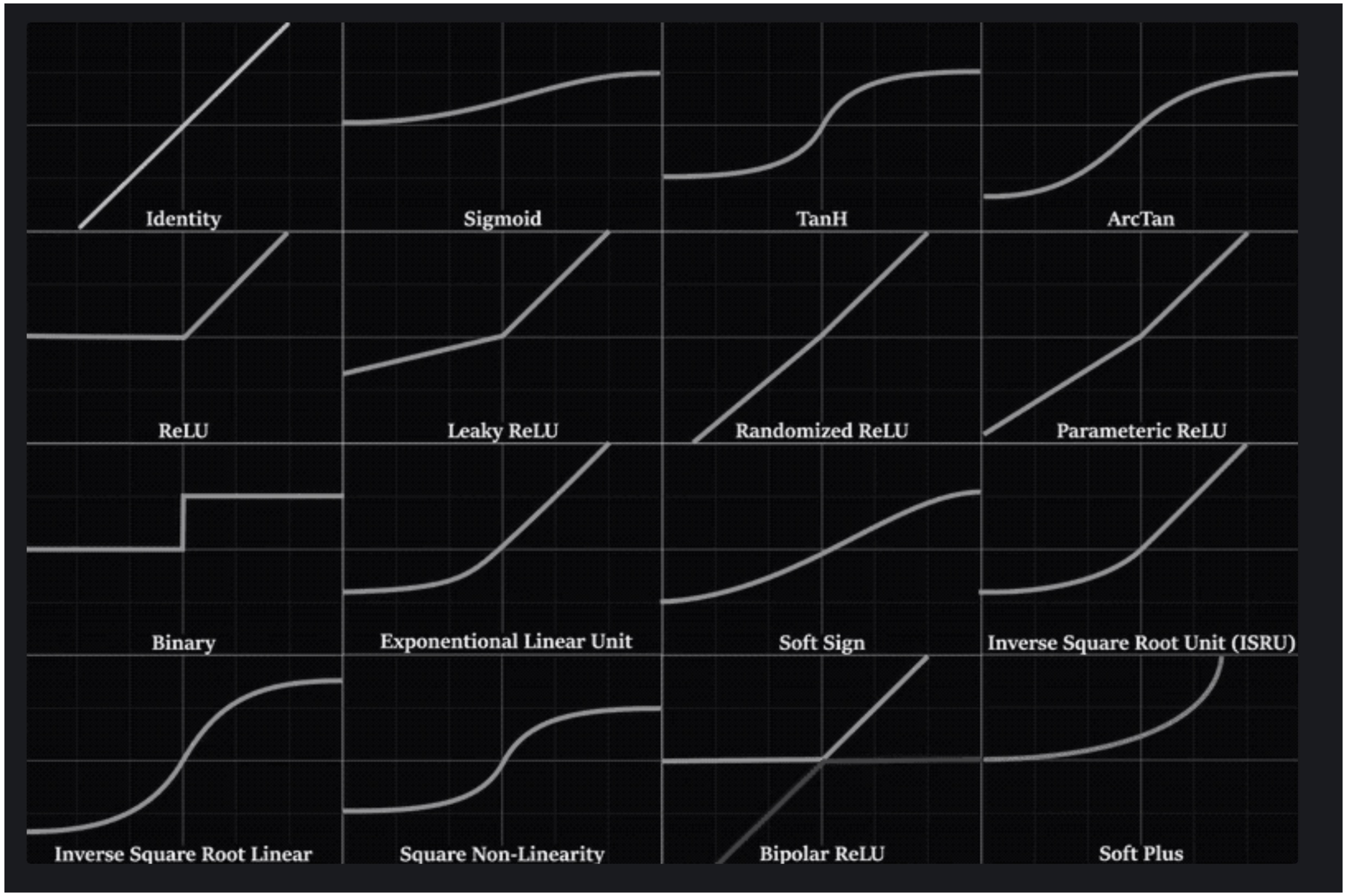
Why is it popular?

1. Can behave better than sigmoid for gradient descent
2. Computational reasons (fast and sparse)

Other Activation Functions

Many other activation functions (non-linearities) have been proposed

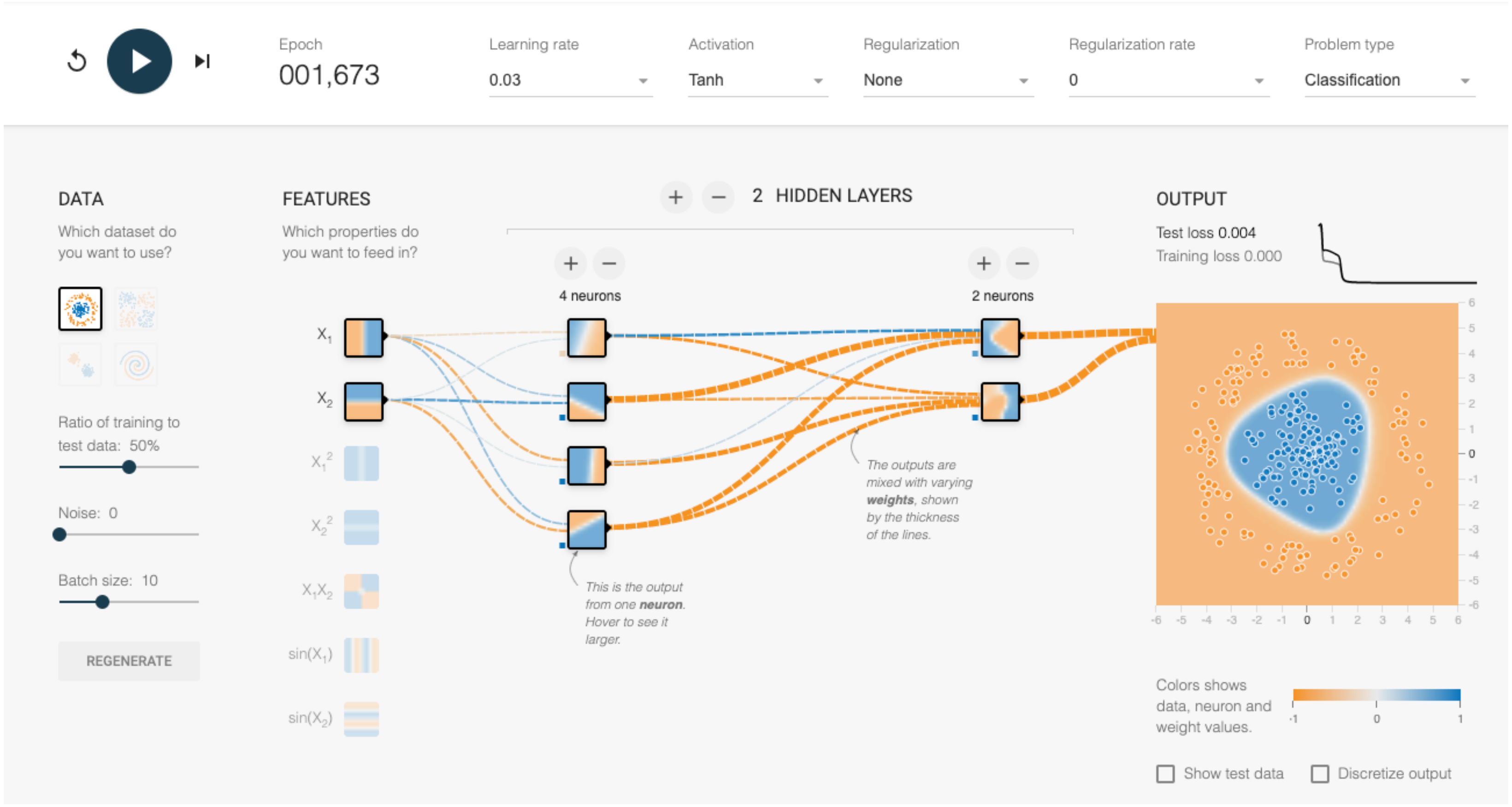
- Which should you use? Highly empirical – requires experiments



https://www.reddit.com/r/learnmachinelearning/comments/eoaq5c/activation_functions_cheat_sheet/

Questions?

Online Demo: http://playground.tensorflow.org/
(you are encouraged to explore this on your own outside of class)



Epoch 001,673

Learning rate 0.03

Activation Tanh

Regularization None

Regularization rate 0

Problem type Classification

DATA

Which dataset do you want to use?

Ratio of training to test data: 50%

Noise: 0

Batch size: 10

REGENERATE

FEATURES

Which properties do you want to feed in?

2 HIDDEN LAYERS

4 neurons

2 neurons

*This is the output from one **neuron**. Hover to see it larger.*

*The outputs are mixed with varying **weights**, shown by the thickness of the lines.*

OUTPUT

Test loss 0.004

Training loss 0.000

Colors shows data, neuron and weight values.

Show test data

Discretize output

More on Neural Networks

Training Neural Networks

Learning a Neural Network Classifier

Cross entropy is the standard loss function used for training neural network classifiers

Training data pairs $\mathbf{x}_i, y_i$, where y_i can take one of K values.

$$
\begin{aligned}
L(\boldsymbol{\theta}) &= \frac{1}{n}\sum_{i=1}^{n} -\log f_k(\mathbf{x}_i; \boldsymbol{\theta}) \\
&= \frac{1}{n}\sum_{i=1}^{n} -\log P(y_i = k|\mathbf{x})
\end{aligned}
$$

Cross-entropy loss function

This value of k is the true class label y_i in the training data.

Recall that minimizing –logP will "push" P towards 1.
So we are pushing the classifier towards having probability 1 for the correct class for each training example

Minimizing the Cross Entropy for a Neural Network

How can we minimize the cross-entropy? We can use gradient descent again!

Updates for parameters $\boldsymbol{\theta}$ in the gradient descent algorithm:

$$\boldsymbol{\theta}^{new} = \boldsymbol{\theta}^{current} - \lambda \cdot \nabla L(\boldsymbol{\theta}^{current})$$

where

$$\nabla L(\boldsymbol{\theta}) = \left(\frac{\partial L(\boldsymbol{\theta})}{\partial \theta_1}, \ldots, \frac{\partial L(\boldsymbol{\theta})}{\partial \theta_p}\right)$$

is a gradient vector of length p, where p is the total number of parameters.

$\boldsymbol{\theta}$ is the set of all parameters (weights, biases, at all layers) in the neural network.

Calculating each Component of the Gradient Vector

Consider a single parameter θ_j (e.g., a weight somewhere in the network):

$$
\begin{aligned}
\frac{\partial L(\boldsymbol{\theta})}{\partial \theta_j} &= \frac{\partial}{\partial \theta_j}\left(-\frac{1}{n}\sum_{i=1}^{n} \log f_k(\mathbf{x}_i; \boldsymbol{\theta})\right) \\
&= -\frac{1}{n}\sum_{i=1}^{n} \frac{\partial}{\partial \theta_j}\left(\log \underset{\uparrow}{f_k}(\mathbf{x}_i; \boldsymbol{\theta})\right)
\end{aligned}
$$

This function f () is a neural network

So the partial derivative for each parameter is much more complex than for a simpler model like the logistic model

Should we calculate this by hand? No!

- Difficult enough for very simple models (e.g. logistic)
- Would need to re-do this calculation for every new network architecture
- Very involved if you have a complex network

The Backpropagation Algorithm

Computation of the gradient for a neural network involves 2 phases:

1. **Forward propagation**: computing the output $f(x_i ; \theta)$ for every training data point using the current parameters θ

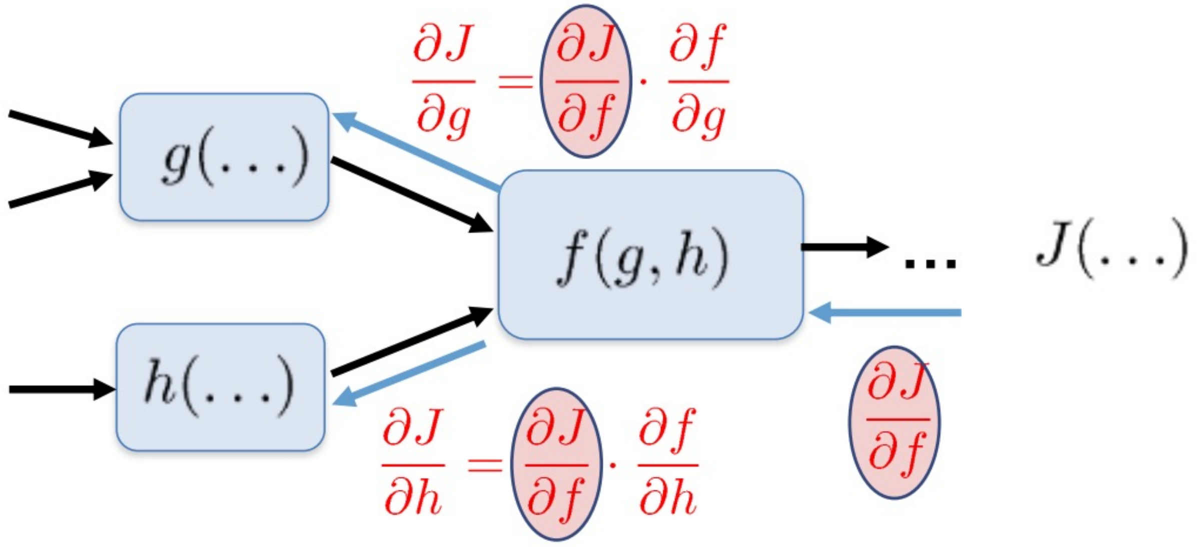
2. **Backward propagation**: using the outputs $f(x_i ; \theta)$ and the y_i values to "backpropagate", layer by layer, the gradient information to each parameter (a computation that uses the network structure)

The backpropagation algorithm mainly refers to step 2, but we can think of backpropagation as an efficient way to compute the partial derivatives (and the gradient) in a multi-layer neural network,

The Backpropagation Algorithm

We'll skip the details on Backprop

- Basic idea: *chain rule* from multivariable calculus



$\frac{\partial J}{\partial g} = \frac{\partial J}{\partial f} \cdot \frac{\partial f}{\partial g}$
$g(\ldots)$
$f(g,h)$
...
$J(\ldots)$
$h(\ldots)$
$\frac{\partial J}{\partial h} = \frac{\partial J}{\partial f} \cdot \frac{\partial f}{\partial h}$
$\frac{\partial J}{\partial f}$

Note: using J for the loss function (instead of L)

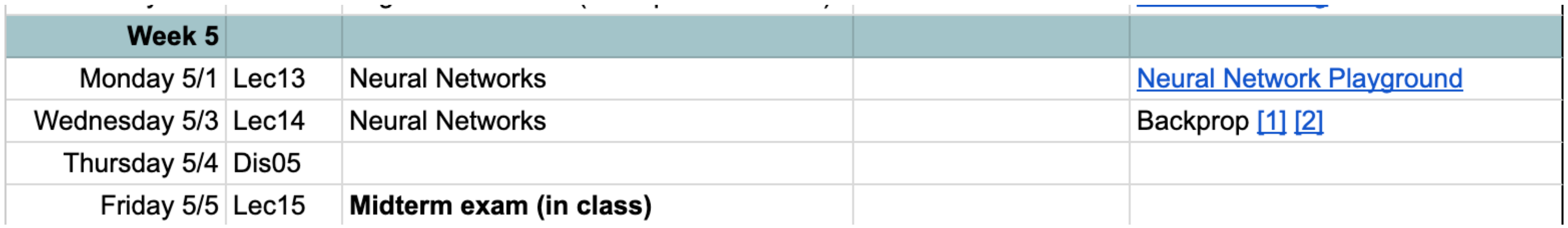
The Backpropagation Algorithm

We'll skip the details on Backprop

- Basic idea: *chain rule* from multivariable calculus

Deep learning libraries (e.g. Pytorch, Tensorflow, etc.) implement backprop via *automatic differentiation*

- Algorithm that efficiently computes the derivative of a composition of simple functions
- Exact computation -- no numerical approximations!



Week 5				
Monday 5/1	Lec13	Neural Networks		Neural Network Playground
Wednesday 5/3	Lec14	Neural Networks		Backprop [1] [2]
Thursday 5/4	Dis05			
Friday 5/5	Lec15	**Midterm exam (in class)**		

Convexity of Cross-Entropy and Neural Networks

Cross-entropy with the logistic model is a *convex* problem:

- > a single global minimum, no local minima in the loss function

Is the same true for neural networks? Unfortunately, no.

All neural networks (even with a single hidden unit) have non-convex losses

This is a price we pay for having a more complex model

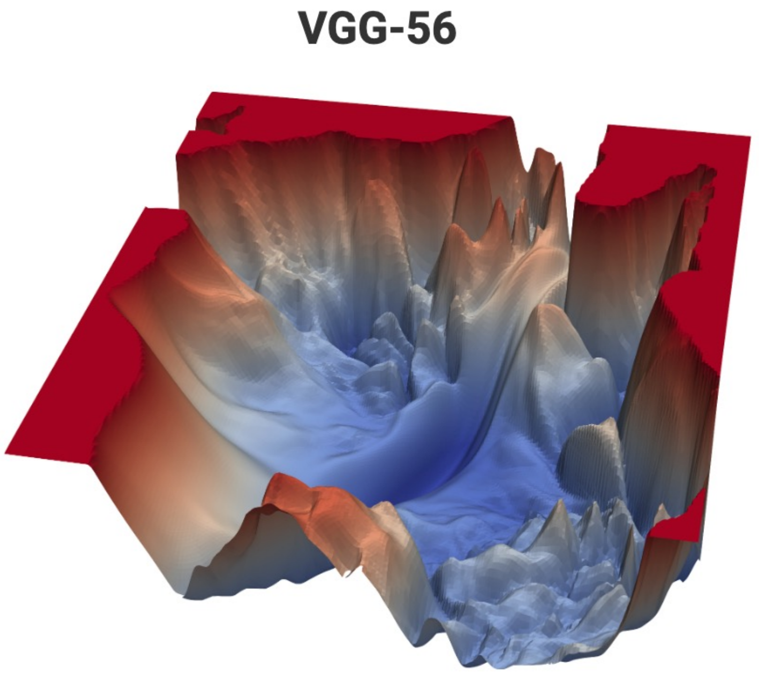
In practice, if we have large training datasets (e.g., $n = 10^5$ or bigger), gradient descent works remarkably well and typically produces good solutions

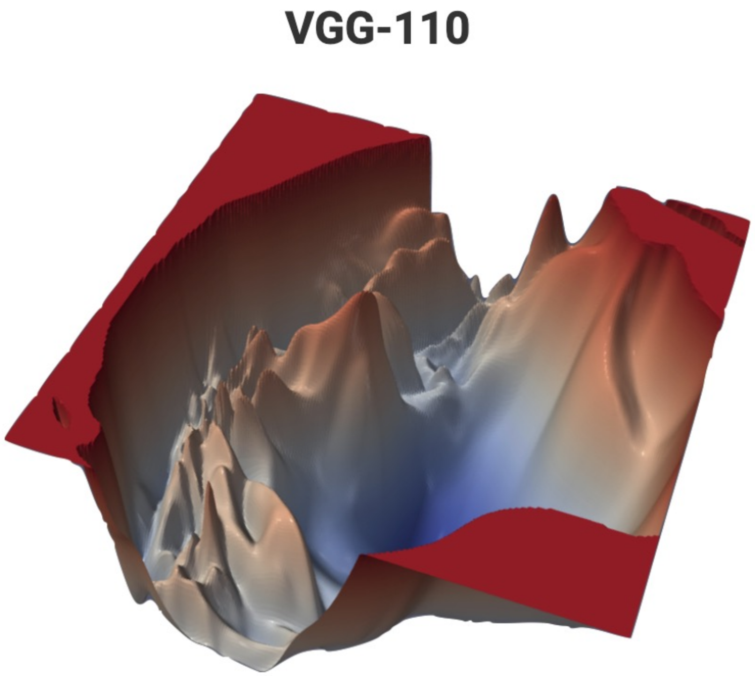
For small datasets (e..g, $n < 10^4$), local minima may be more of an issue

Initialization and Local Minima

Gradient descent might only converge to a **local** minimum

- neural networks can have very complex, high-dimensional loss surfaces





Hao Li et al. 2017, Visualizing the Loss Landscape of Neural Nets, losslandscape.com

Questions?

Complexity of Gradient Updates for a Neural Network

The forward computation step in gradient update, per data point,
is at least O(p) where p is the number of parameters

(Why? Each parameter is involved in at least 1 multiplication in forward pass)

So, the total complexity over all n points is at least O(n p),
for one gradient update in parameter space

Keep in mind that each of n and p can be very large, e.g.,
n = 1 million images, and p could be in the millions (or more) too

And we might require 100's or 1000's of updates..
…..so gradient descent with neural networks can be very slow!

Stochastic Gradient Descent

We can write the standard gradient update as:

$$\boldsymbol{\theta}^{new} = \boldsymbol{\theta} - \lambda \cdot \nabla L(\boldsymbol{\theta})$$

where

$$\nabla L(\boldsymbol{\theta}) = \frac{1}{n} \sum_{i=1}^{n} \nabla\left(-\log f_k(\mathbf{x}_i; \boldsymbol{\theta})\right)$$

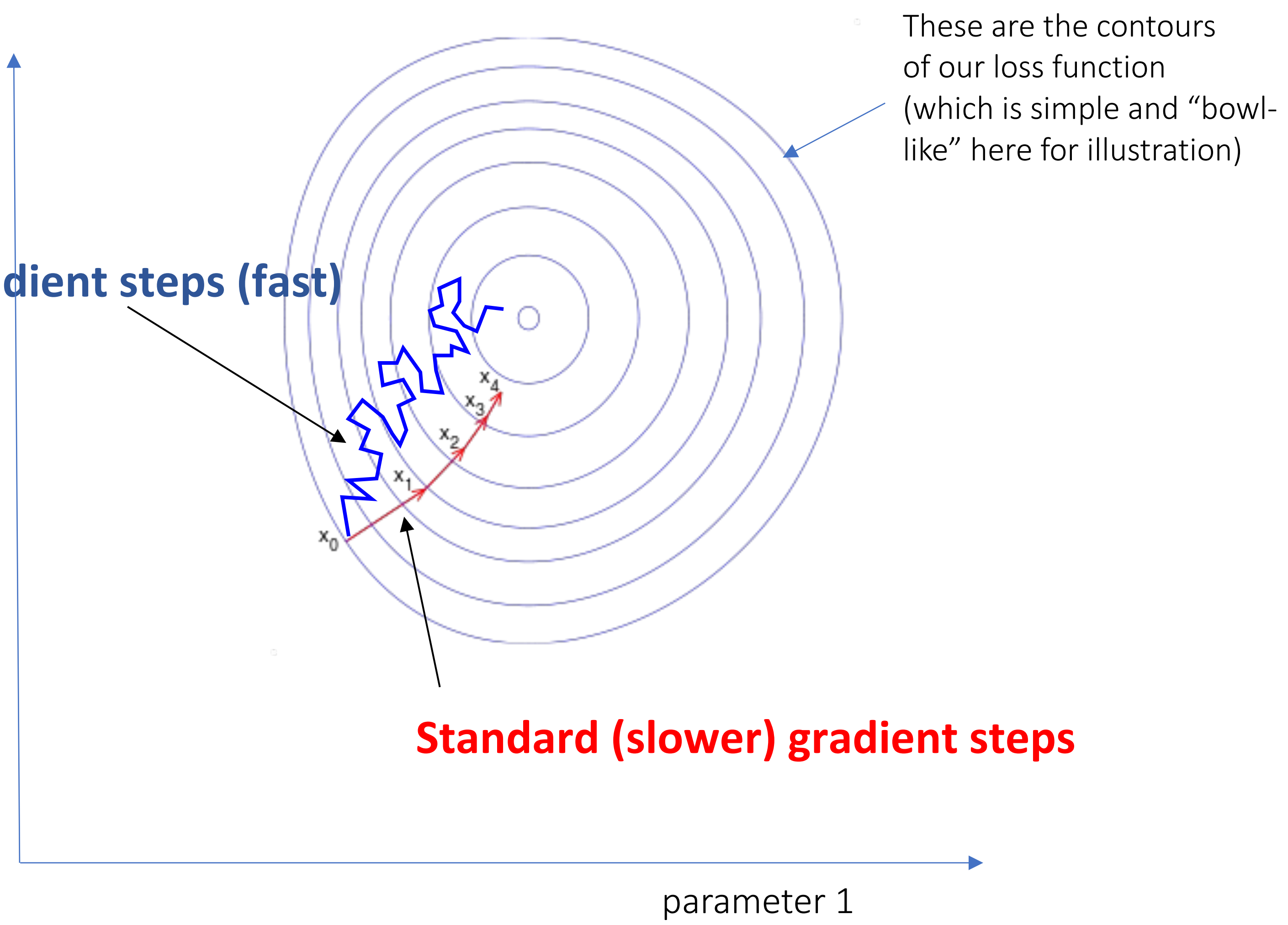
Idea: replace the full sum with a much smaller noisier sum:

$$\nabla L_b(\boldsymbol{\theta}) = \frac{1}{b} \sum_{j=1}^{b} \nabla\left(-\log f_k(\mathbf{x}_j; \boldsymbol{\theta})\right)$$

where the sum is over a randomly selected **batch** (or **minibatch**) of training examples, where b is the size of the batch, and $b << n$, e.g., $b = 1$ or $b = 10$ or $b = 100$.

Stochastic Gradient Descent (SGD)

(Example in 2-dimensional Parameter Space)


These are the contours of our loss function (which is simple and "bowl-like" here for illustration)
Stochastic gradient steps (fast)
Standard (slower) gradient steps
x_0
x_1
x_2
x_3
x_4
parameter 2
parameter 1

Stochastic Gradient Descent (continued)

Advantage:
Much faster than regular gradient: updates take time O(bp) versus O(np)

Disadvantage:
Is not computing the "correct" full gradient: is a noisy (stochastic) approximation

Empirical results in practice? Works very well

Theoretical guarantees? Some nice theory that shows it will converge to a local minimum just like standard gradient descent

SGD is the default "go to" method for training neural networks

Stochastic Gradient Descent (continued)

Implementation:
The SGD algorithm is the same as the standard gradient descent algorithm, except for how the gradient is computed

Random selection of training examples in the batches is important
In practice we randomly order the n datapoints before starting SGD and then just go through them b points at a time

How to select the batch size b? selected heuristically, or we can treat it as a hyperparameter and search over its values on a validation set

SGD can be used with any ML model that uses gradients: but its usually most effective with large n and large p (e.g., neural networks)
(for small n and/or small p, standard gradient may be better)

Stochastic Gradient Descent (continued)

Terminology:

Batch (or minibatch) size = b = number of examples per gradient update

Number of epochs = number of times algorithm sees all examples

e.g., n = 1000 datapoints, b = 100 => 10 gradient updates per epoch

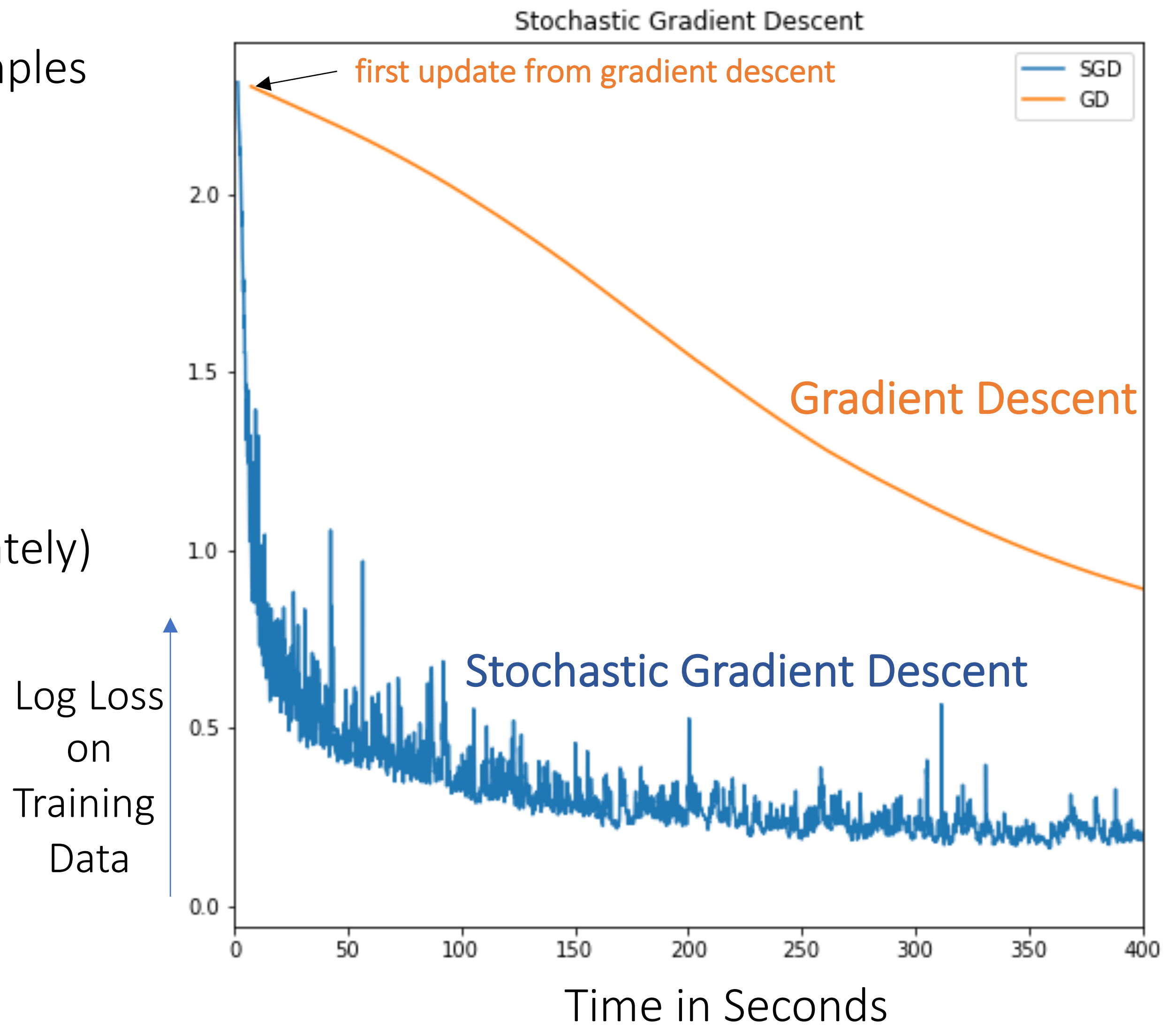
Common in SGD to specify length of training in terms of number of epochs, e.g., train for 3 epochs means passing through the entire dataset 3 times

Comparing SGD and GD on MNIST data, b=4

Minibatch size = 4 examples
n = 60k training images

Neural network with
d = 784 inputs
M = 256 hidden units
K = 10 classes

p = 203,000 (approximately)

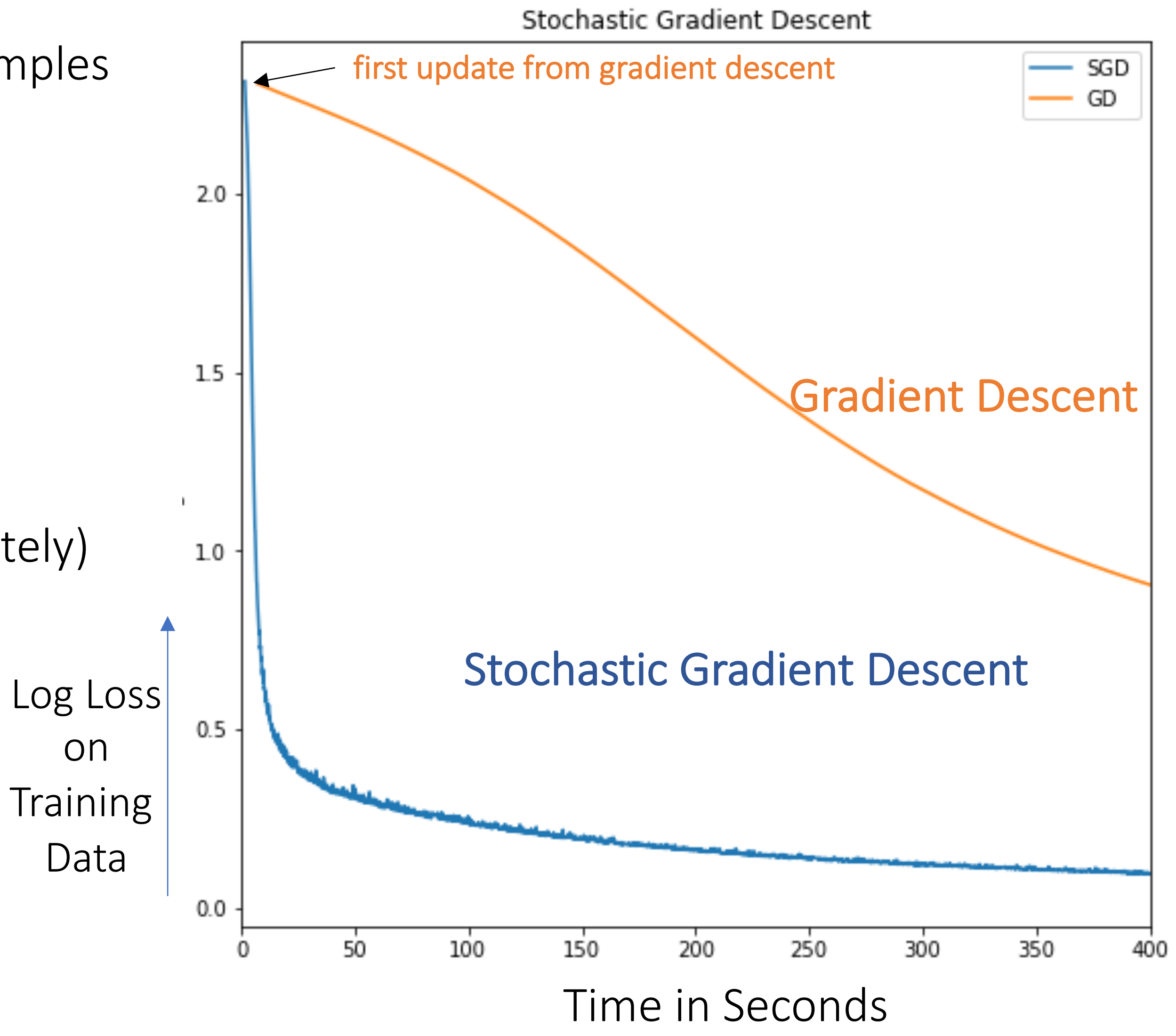


Comparing SGD and GD on MNIST data, b=64

Minibatch size = 64 examples
n = 60k training images

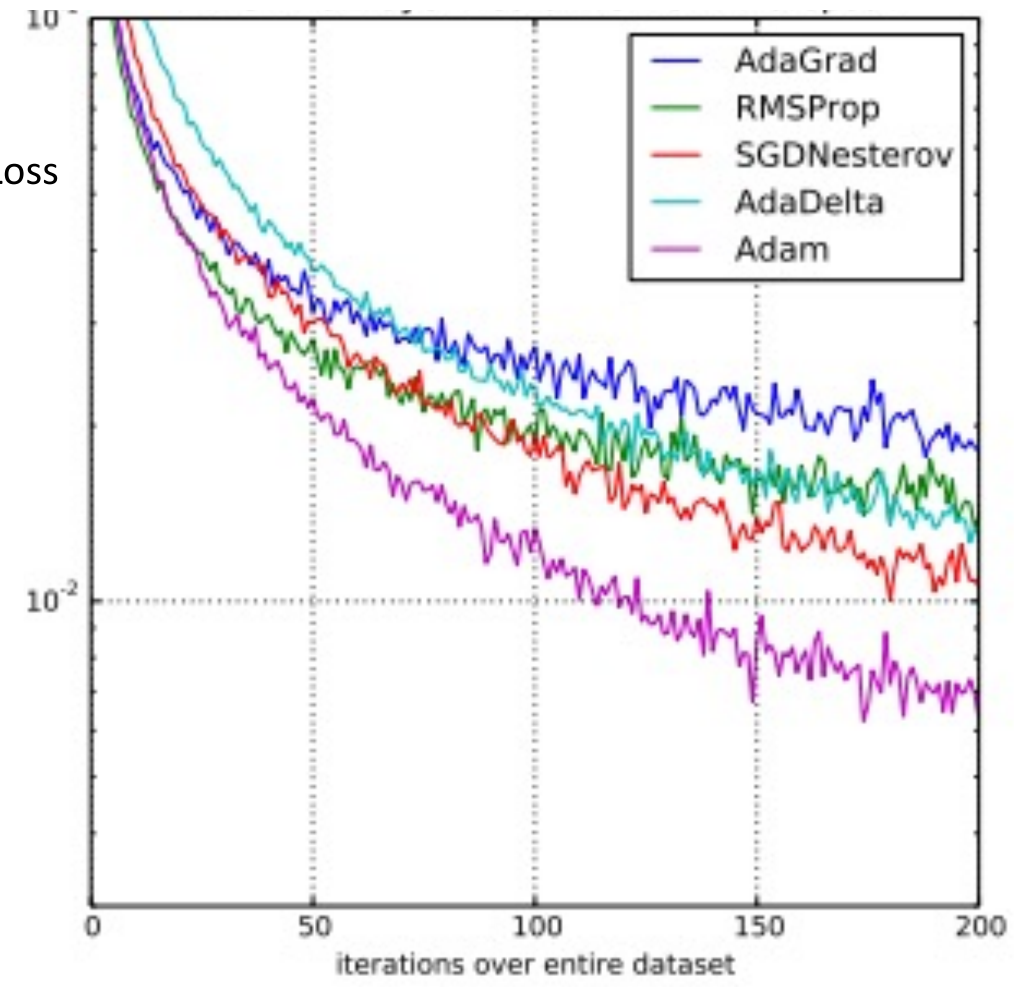
Neural network with
d = 784 inputs
M = 256 hidden units
K = 10 classes

p = 203,000 (approximately)



Another Example of SGD Training with Log-Loss for a Neural Network

LogLoss



Graph shows different algorithmic variations of stochastic gradient descent

Note the noisy nature of the plots as the log-loss decreases. With small batch sizes (values of b) the gradient information can be noisy

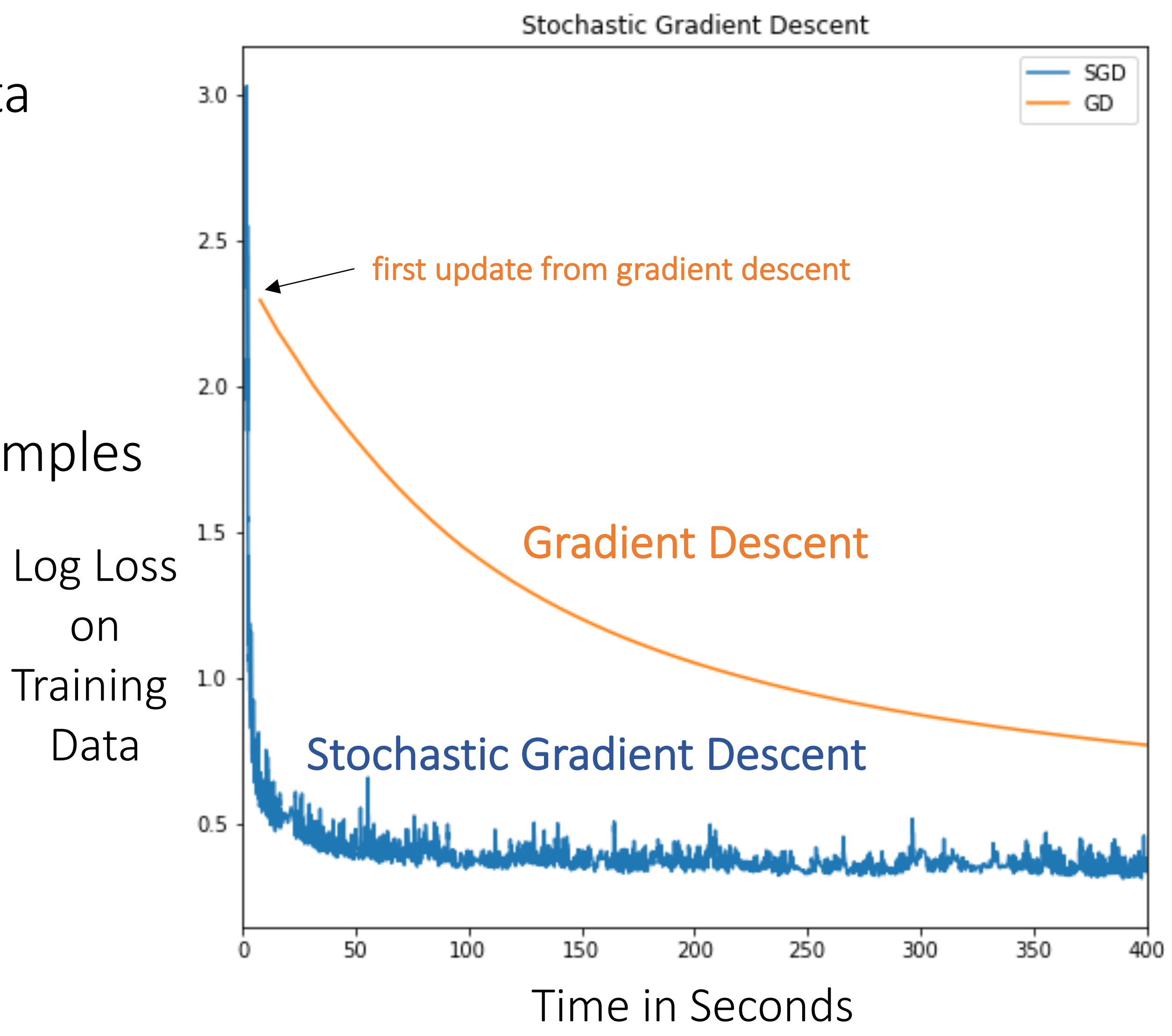
…. but the overall trajectory is still clearly "downhill" for the loss

From: https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/

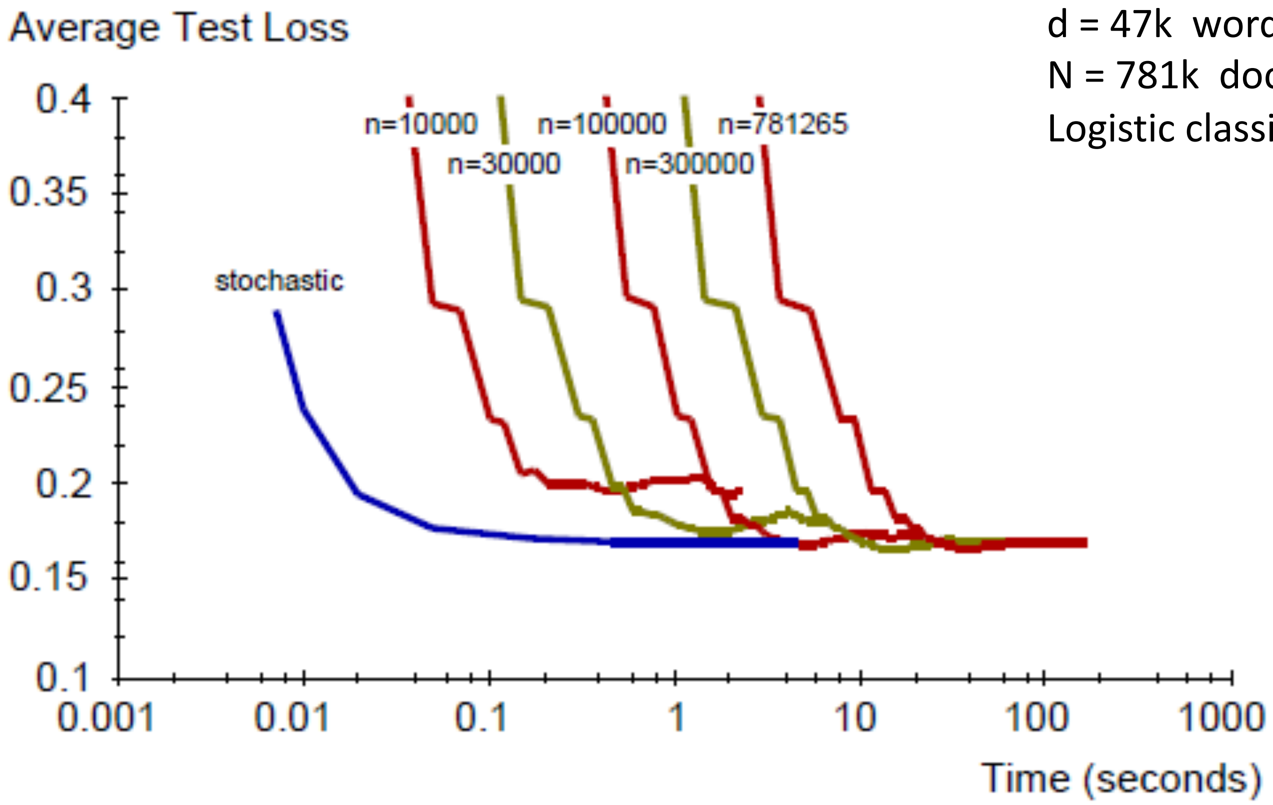
Training on MNIST data (60k images)

Logistic classifier

Minibatch size = 4 examples



Example of Stochastic Gradient Optimization



Binary Text Classification Problem
d = 47k word features
N = 781k documents
Logistic classifiers

From Leon Bottou, Stochastic Gradient Learning, MLSS Summer School 2011, Purdue University, http://learning.stat.purdue.edu/mlss/_media/mlss/bottou.pdf

Sidenote: Hyperparameters

- Hyperparameters versus Parameters?
  - Parameters θ: the weights, coefficients, etc in a model
  - Hyperparameters: general aspects of a model or learning algorithm we can control

- Neural networks have very many hyperparameters:
  - Regularization weight α
    - And type of regularization: L1, L2, etc.
  - Hyperparameters of optimizer
    - Learning rate
    - Batch size b for stochastic gradient descent
  - Network architecture for neural network models
    - Number of layers, size of layers, type of non-linearity, etc.

Sidenote: Hyperparameter Tuning

- We can select hyperparameters by hand (by intuition, by guessing, by experience) but with large datasets we can often get better performance by using the data to suggest good values for hyperparameters

- Recall that we can split our data into
  - Training (for estimating parameters q, for some setting of hyperparameters
  - Validation (for evaluating different hyperparameter settings)
  - Test (for evaluating the accuracy of our model with final settings)

- Grid search on validation data, to search for hyperparameter values that give the best validation accuracy, is widely used, but can be computationally very expensive

Sidenote: Hyperparameter Tuning

- We can select hyperparameters by hand (by intuition, by guessing, by experience) but with large datasets we can often get better performance by using the data to suggest good values for hyperparameters

- Recall that we can split our data into
  - Training (for estimating parameters q, for some setting of hyperparameters
  - Validation (for evaluating different hyperparameter settings)
  - Test (for evaluating the accuracy of our model with final settings)

- Common mistake in HW2:
  - You should select hyperparameters *only based on the validation set performance*

Complexity and Accuracy Tradeoffs

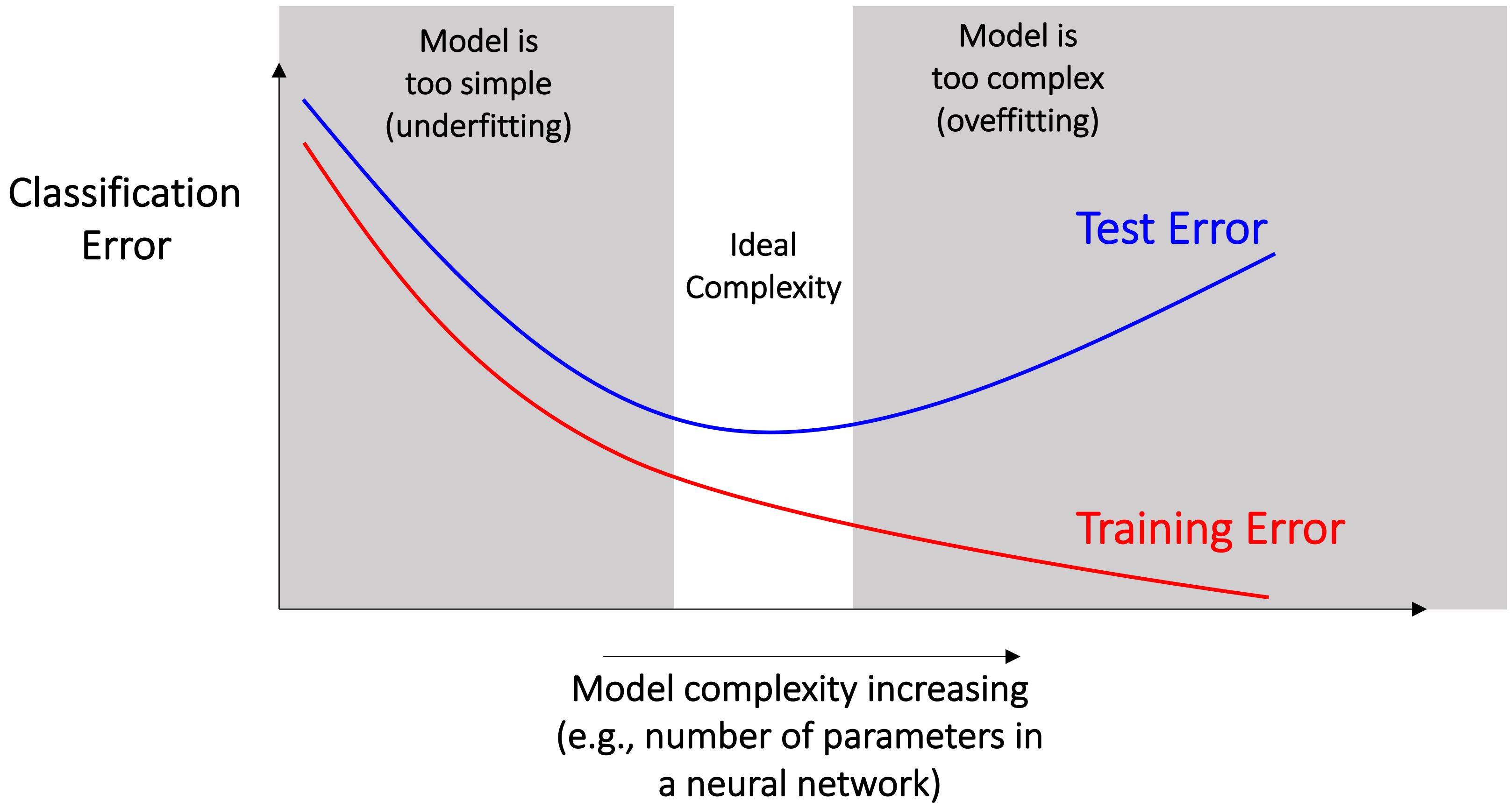
There is a fundamental trade-off between model complexity and model accuracy in machine learning models

As models get more complex, they can fit the training data perfectly
….but their performance on new test data may get worse

For neural networks, increasing the size of the network (the number of parameters) will generally increase the complexity of the model

(but the basic complexity/performance tradeoffs exists for any type of model where we can vary the complexity of the model)

Complexity/Accuracy Tradeoffs



Questions?

Wrapup

- Training neural networks is hard
  - Highly non-convex optimization problem
  - Stochastic gradient descent algorithm is the standard
  - Idea: take a mini-batch of data at every step
  - *Backprop* algorithm used to compute gradients

- Large number of hyperparameters in neural networks
  - Time consuming and difficult to tune

- Good luck on midterm!