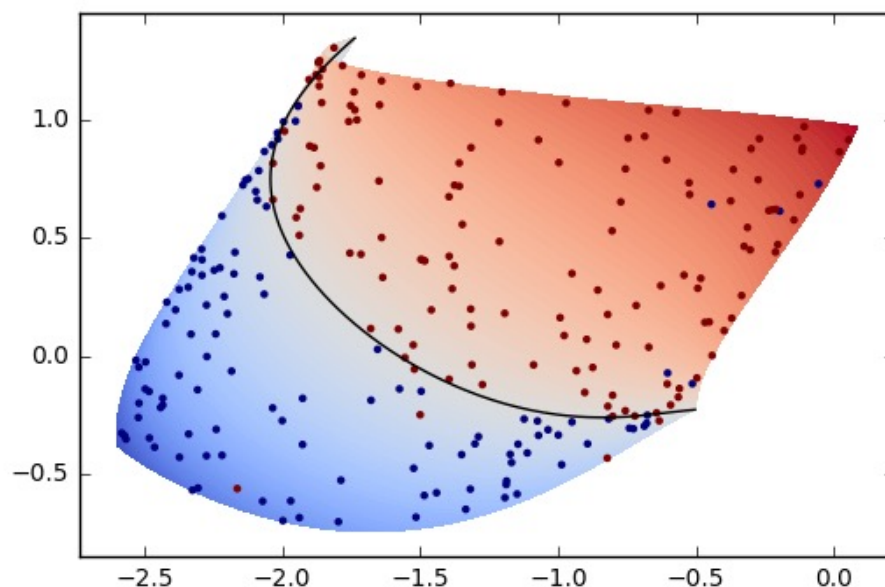


Lecture 13: Neural Networks

Part 2



Gavin Kerrigan
Spring 2023

Some materials courtesy Padhraic Smyth, Alex Ihler.

Midterm Exam

- What to study
 - Lectures, Homeworks + Solutions, Discussion sections
 - Exam will be based on understanding the material above
- Exam covers up to (and including) logistic regression
 - No neural networks
- Format of exam
 - In lecture hall, in person, 1pm-1:50pm Friday this week
 - Exam will last 50 minutes, will start once everyone is seated
 - All you need is a pen or pencil
 - Closed book: no notes, books, etc
 - No electronic devices (calculators, phones, etc)
- Assigned Seating per UCINetID
 - Check Ed for details (will be posted Wed afternoon or Thurs morning)

Additional Announcements

- Discussion section
 - Midterm review – come with questions
 - Sample midterm solutions
- Homework 2
 - Grading in progress
 - Solutions available on Canvas
- Homework 3 – due in 2 weeks (Friday 5/12)
 - Released today or tomorrow
 - Shorter HW due to exam
 - Focused on neural networks

Questions?

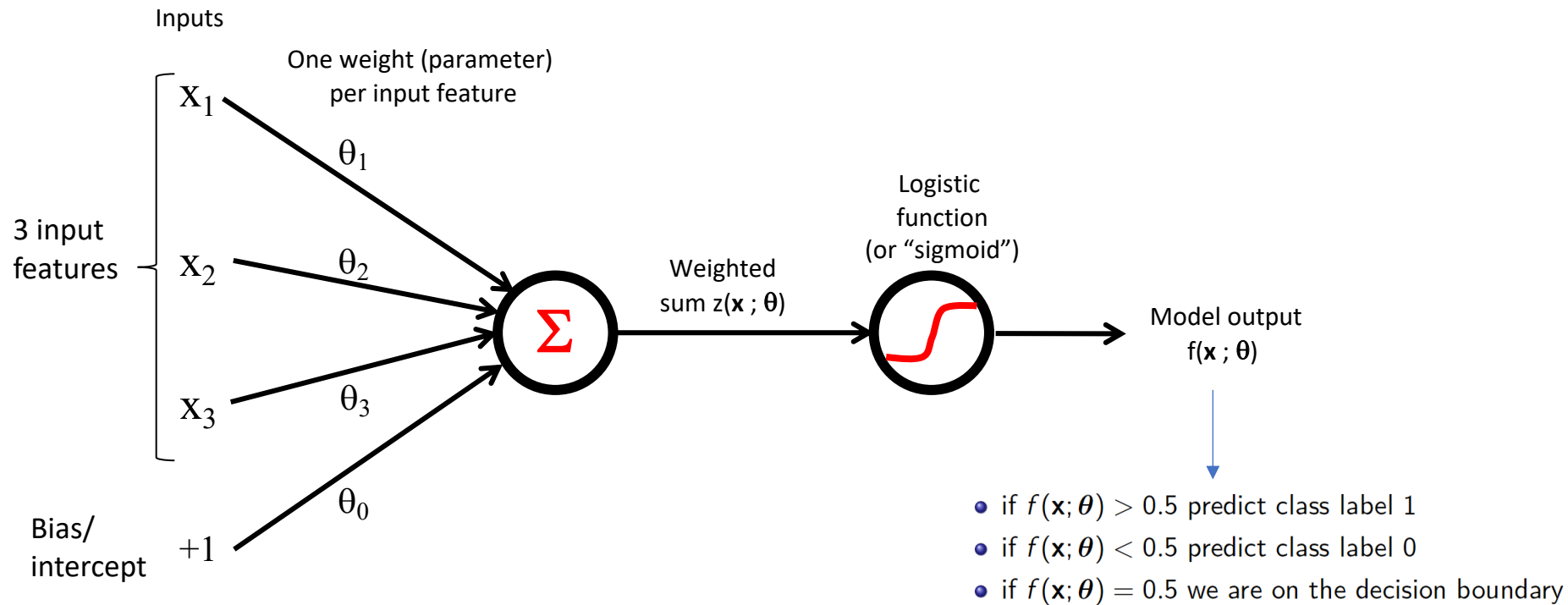
Recap of Neural Networks

More Complex Networks

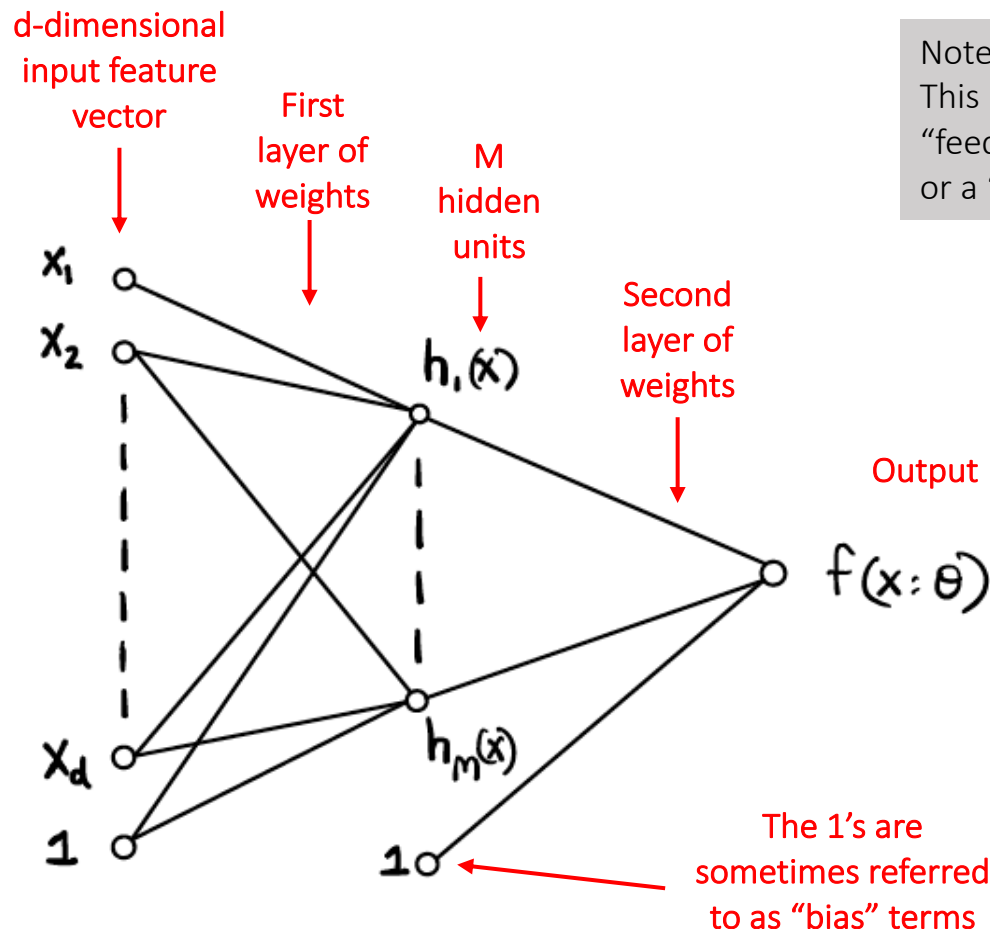
Network Architectures

Training Neural Networks

Block Diagram of a Logistic Classifier



A General Single Hidden Layer Neural Network



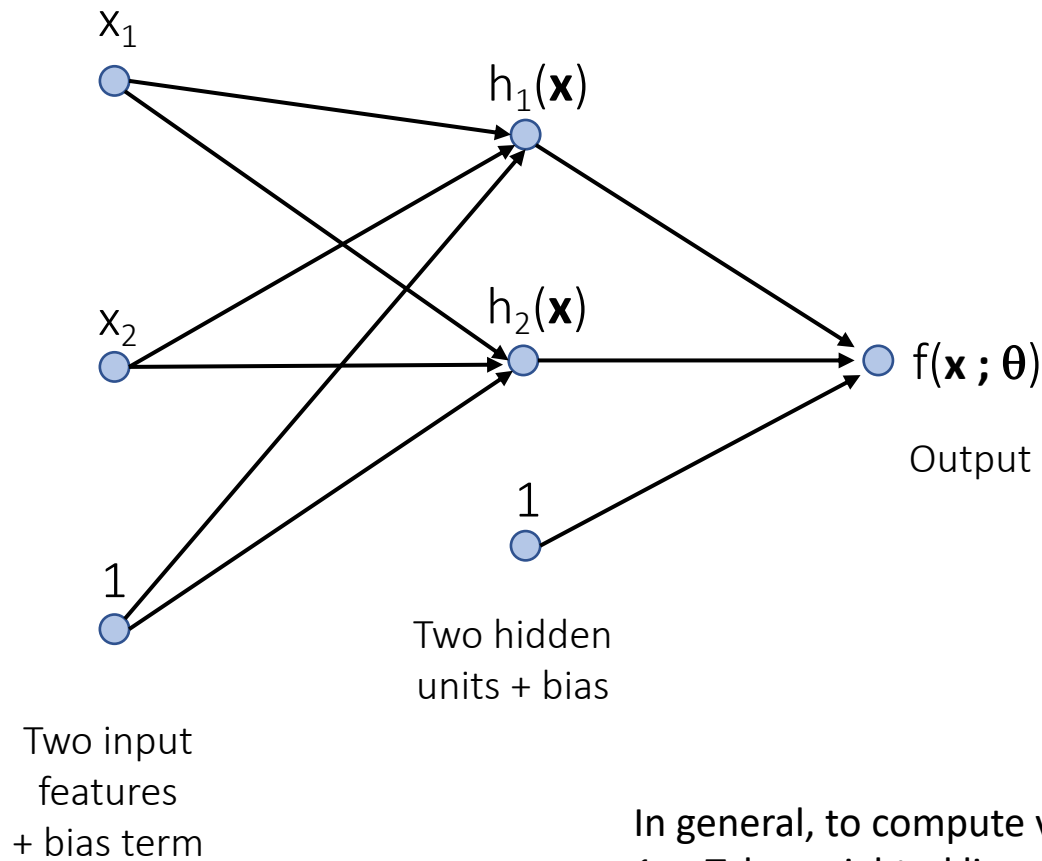
Note:

This is also sometimes referred to as "feedforward" neural network or a "multilayer perceptron"

Parameters θ = all parameters from all layers

How many for a neural network with M hidden units? $(d+1)*M + M+1 = O(dM)$

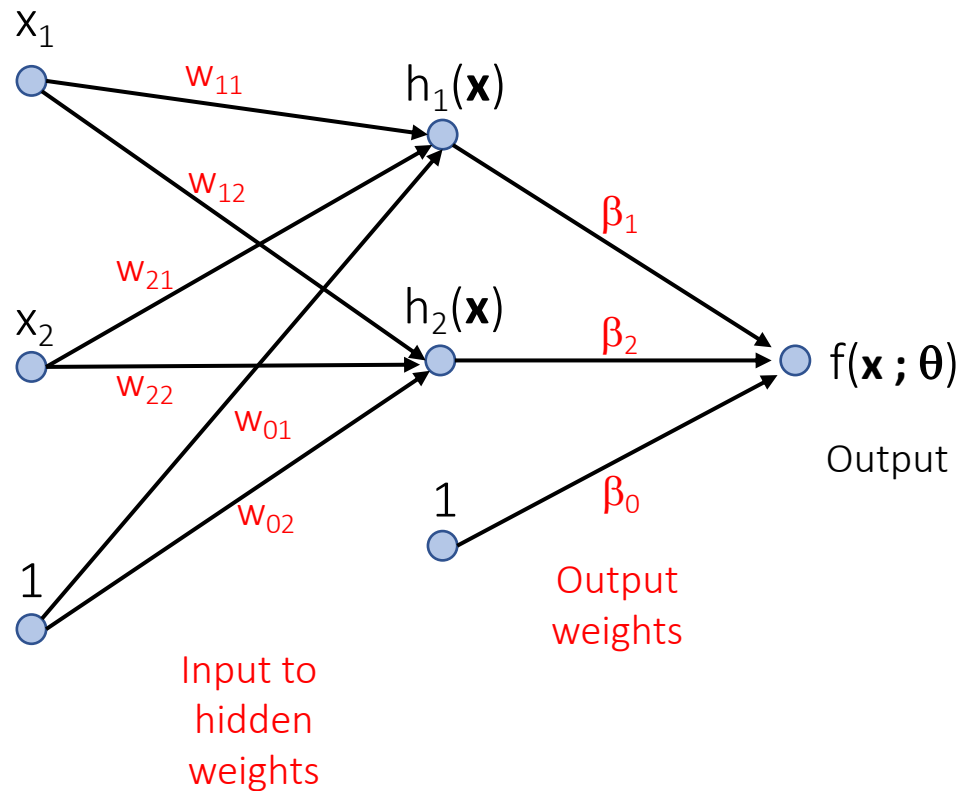
Example of a Neural Network with 1 Hidden Layer



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)

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)

Equations for the Example Neural Network

Feature vector $\mathbf{x} = (x_1, x_2)$

Non-linear activation function g for the hidden unit, e.g., a sigmoid

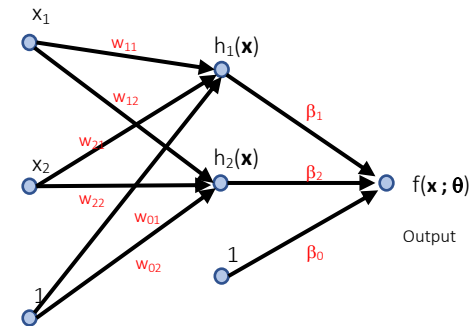
$$g(z) = \frac{1}{1 + e^{-z}}$$

$$h_1(\mathbf{x}) = g(w_{11}x_1 + w_{21}x_2 + w_{01})$$

$$h_2(\mathbf{x}) = g(w_{12}x_1 + w_{22}x_2 + w_{02})$$

$$f(\mathbf{x}; \boldsymbol{\theta}) = g(\beta_1 h_1(\mathbf{x}) + \beta_2 h_2(\mathbf{x}) + \beta_0)$$

where the parameter vector $\boldsymbol{\theta}$ includes all the w 's and β 's.



Math Notation for a Single Hidden Layer NN

Let $g(z)$ represent a general activation function, e.g., in our examples

$$g(z) = \frac{1}{1 + e^{-z}} = \text{sigmoid function}$$

Let $h_m(\mathbf{x})$ be the output of the m th hidden unit:

$$h_m(\mathbf{x}) = g\left(w_{m0} + \sum_{j=1}^d w_{mj} \cdot x_j\right)$$

where w_{mj} is the weight from input x_j to hidden unit h_m , with $j = 0, \dots, d$

Math Notation for a Single Hidden Layer NN

The output of the network is

$$\begin{aligned} f(\mathbf{x}; \boldsymbol{\theta}) &= g\left(\beta_0 + \sum_{m=1}^M \beta_m \cdot h_m(\mathbf{x})\right) \\ &= g\left(\beta_0 + \sum_{m=1}^M \beta_m \cdot g\left(w_{m0} + \sum_{j=1}^d w_{mj} \cdot x_j\right)\right) \end{aligned}$$

The parameters $\boldsymbol{\theta}$ of the neural network are

$$\boldsymbol{\theta} = (\mathbf{w}_1, \dots, \mathbf{w}_M, \boldsymbol{\beta})$$

where \mathbf{w}_m are the set of weights going from the input to the m th hidden unit and $\boldsymbol{\beta}$ is the set of M weights going from hidden units to the output.

Weight Matrix Notation

$$W = \begin{bmatrix} \dots & w_1 & \dots \\ & w_2 & \\ & \vdots & \\ \dots & w_m & \dots \end{bmatrix} = \begin{bmatrix} w_{10} & w_{11} & \dots & w_{1d} \\ \vdots & & & \vdots \\ w_{m0} & w_{m1} & \dots & w_{md} \end{bmatrix}$$

The second matrix has a horizontal dimension of $d+1$ and a vertical dimension of m .

$$x = \begin{pmatrix} 1 \\ x_1 \\ \vdots \\ x_d \end{pmatrix}$$

Weight matrix
for the first layer

Weight Matrix Notation

$$W = \begin{bmatrix} \dots & w_1 & \dots \\ & w_2 & \\ & \vdots & \\ & w_m & \dots \end{bmatrix} = \begin{bmatrix} w_{10} & w_{11} & \dots & w_{1d} \\ \vdots & & & \vdots \\ w_{m0} & w_{m1} & \dots & w_{md} \end{bmatrix} \quad \begin{matrix} \xleftarrow{d+1} \\ \uparrow M \end{matrix}$$

$$x = \begin{pmatrix} 1 \\ x_1 \\ \vdots \\ x_d \end{pmatrix}$$

Weight matrix
for the first layer

$$W x = \begin{pmatrix} w_1 x \\ w_2 x \\ \vdots \\ w_m x \end{pmatrix} \quad \begin{matrix} \uparrow M \end{matrix}$$

M rows x (d+1) columns

(d+1) x 1 column vector

vector of M weighted sums (one per hidden unit)

$$g(\mathbf{W}\mathbf{x}) = g \begin{pmatrix} w_1 x \\ w_2 x \\ \vdots \\ w_m x \end{pmatrix} = \begin{pmatrix} g(w_1 x) \\ g(w_2 x) \\ \vdots \\ g(w_m x) \end{pmatrix} = \begin{pmatrix} h_1(x) \\ h_2(x) \\ \vdots \\ h_m(x) \end{pmatrix}$$

Compute the M hidden unit responses, applying the non-linearity $g(\cdot)$ element wise to the $\mathbf{W}\mathbf{x}$ vector

$$\mathbf{B} = (\beta_0, \beta_1, \beta_2, \dots, \beta_M)$$

Write the 2nd layer of weights as a row vector

$$\mathbf{h}(x) = \begin{pmatrix} 1 \\ h_1(x) \\ h_2(x) \\ \vdots \\ h_m(x) \end{pmatrix}$$

Augment the hidden unit responses with a "1" for the bias term

Element-wise application of non-linear $g(\cdot)$ activation function

$$f(\mathbf{x}; \boldsymbol{\theta}) = g(\boldsymbol{\beta} \mathbf{h}(\mathbf{x})) = g(\boldsymbol{\beta} g(\mathbf{W}\mathbf{x}))$$

Matrix-vector representation of a neural network with a single hidden layer

Questions?

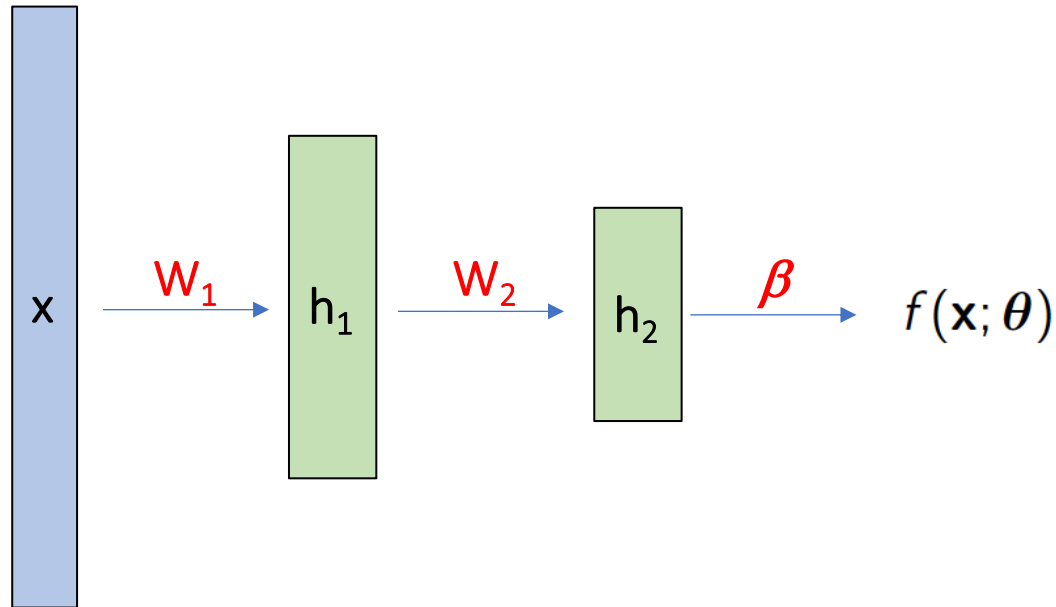
Recap of Neural Networks

More Complex Networks

Network Architectures

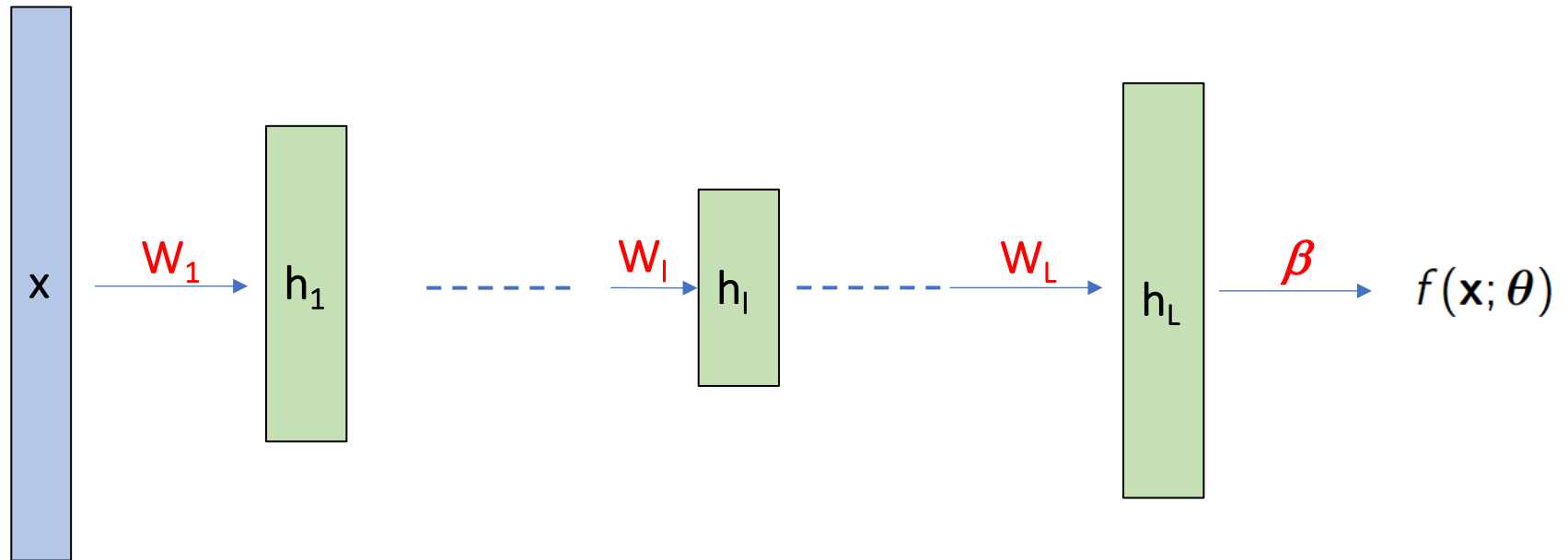
Training Neural Networks

Networks with Two Hidden Layers



Each hidden unit layer can have different numbers of hidden units

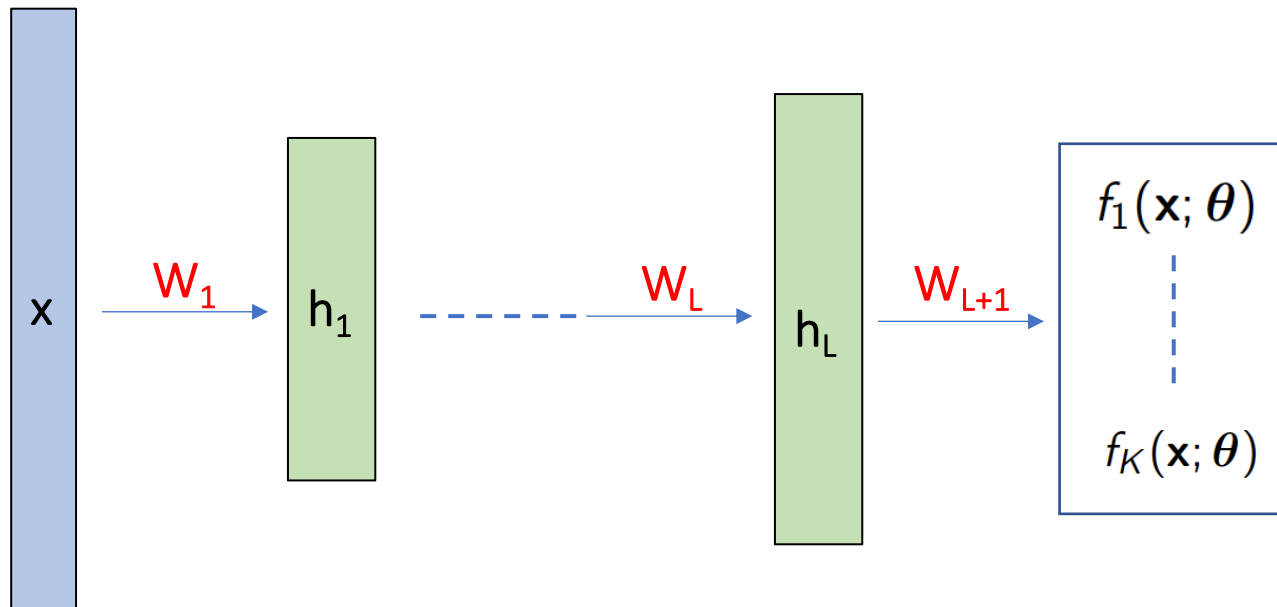
Networks with L Hidden Layers



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

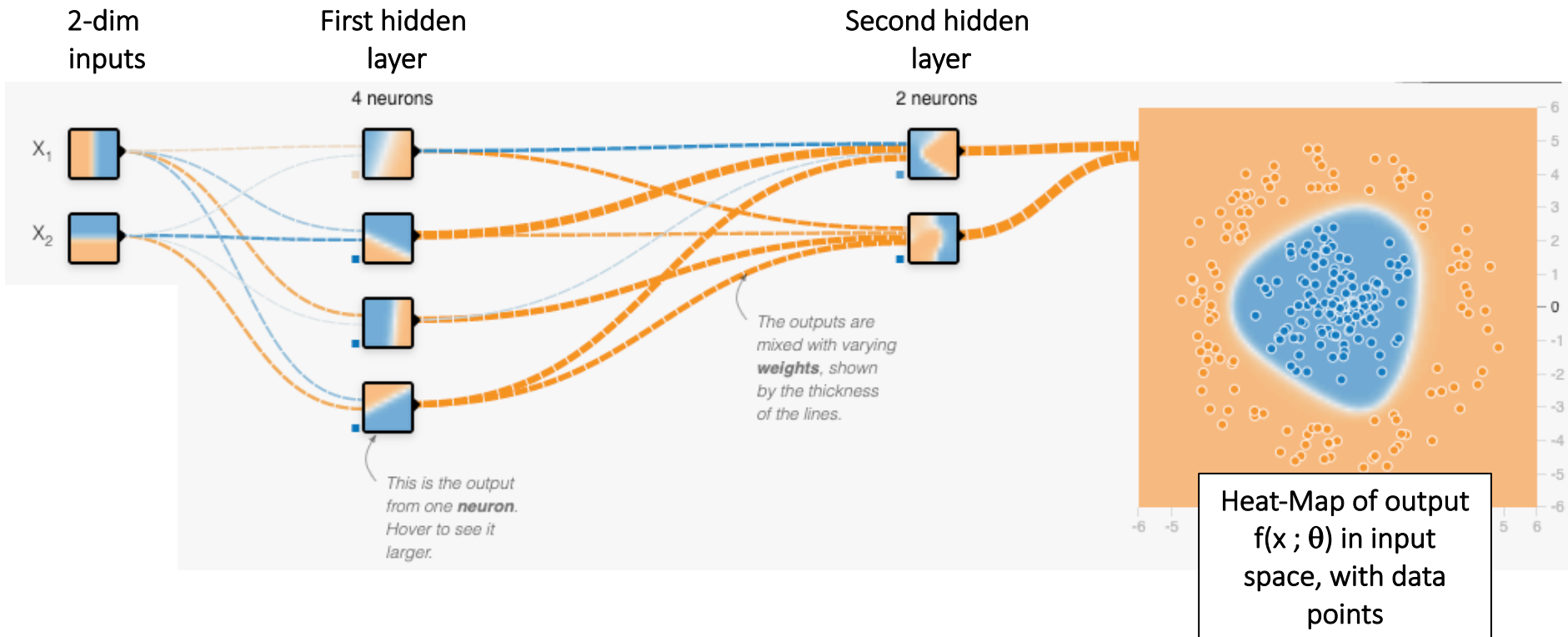


K different outputs, normalized by softmax function to sum to 1
(same softmax function as for K -ary logistic classifier)

Can interpret k th output as $P(y = k \mid x)$, i.e., probability of class k

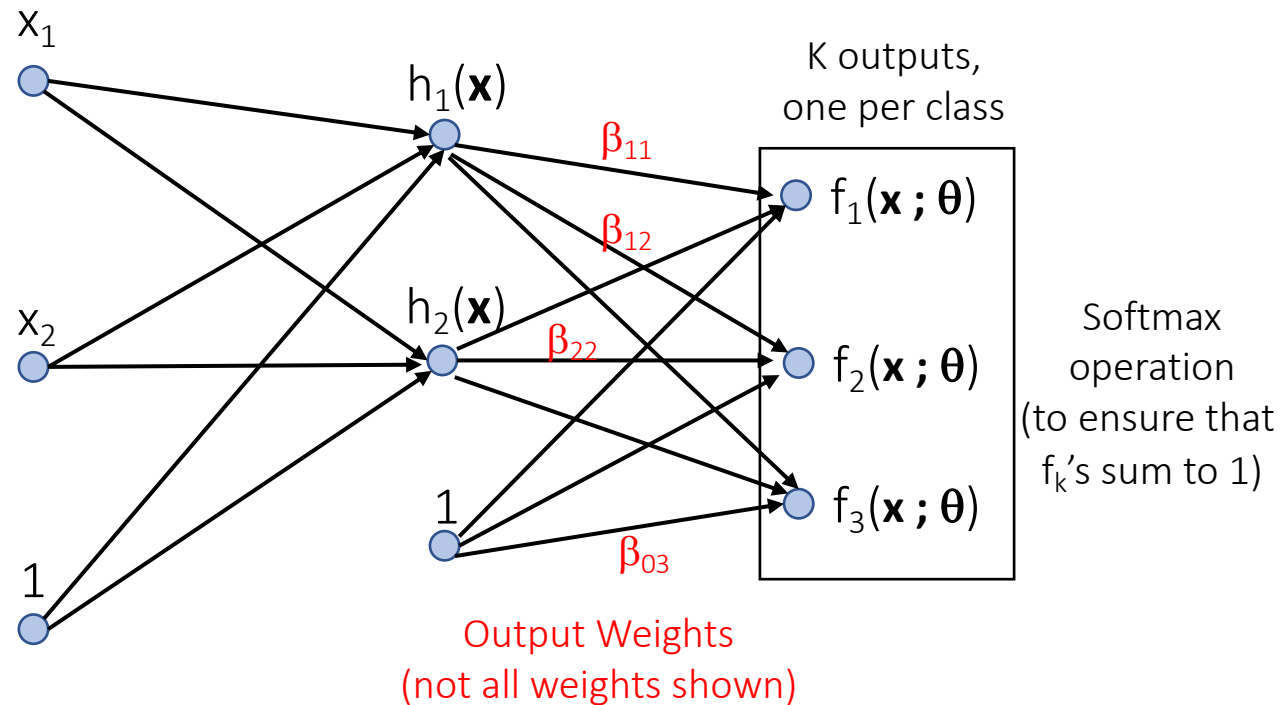
Notation warning:
We are using K here to denote the number of classes (instead of C)

Example of a Simple Network with 2 Hidden Layers



Example from <http://playground.tensorflow.org/>

Example Neural Network with $K = 3$ Outputs



Note: in theory since outputs sum to 1, we really only need $K-1$ outputs, but in machine learning, for $K > 2$, there are usually K outputs

Equations for Neural Network with K Outputs

Hidden units computed in the same way as before

Each output is now computed via softmax:

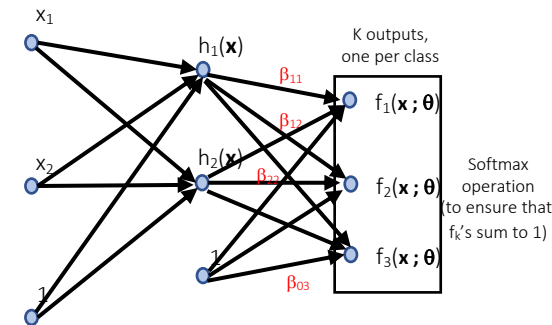
$$f_k(\mathbf{x}; \boldsymbol{\theta}) = \text{softmax}(\beta_{1k}h_1(\mathbf{x}) + \beta_{2k}h_2(\mathbf{x}) + \beta_{0k}), \quad k = 1, \dots, K$$

where the softmax function is the same as it was for the K -way logistic model:

$$\text{softmax}(z_k) = \frac{e^{z_k}}{\sum_{r=1}^K e^{z_r}}$$

i.e., it exponentiates the argument (to make the term positive), and then normalizes by dividing by the sum of exponentiated positive terms.

In this way as get K numbers that sum to 1 and that lie between 0 and 1 and we can interpret $f_k(\mathbf{x}; \boldsymbol{\theta}) = P(\text{class} = k|\mathbf{x})$, i.e., as the probability of the k th class given input \mathbf{x} .





Equations with M Hidden Units, K Outputs

More generally, with M hidden units, and writing in matrix-vector form:

$$\mathbf{h} = g(\mathbf{W}\mathbf{x} + \mathbf{w}_0)$$

Diagram illustrating the hidden layer equation $\mathbf{h} = g(\mathbf{W}\mathbf{x} + \mathbf{w}_0)$ with annotations:

- \mathbf{h} : Vector of dimension $M \times 1$
- $g(\cdot)$: Activation function
- \mathbf{W} : Weight matrix of dimension $M \times d$
- \mathbf{x} : Feature vector of dimension $d \times 1$
- \mathbf{w}_0 : Vector of bias terms of dimension $M \times 1$

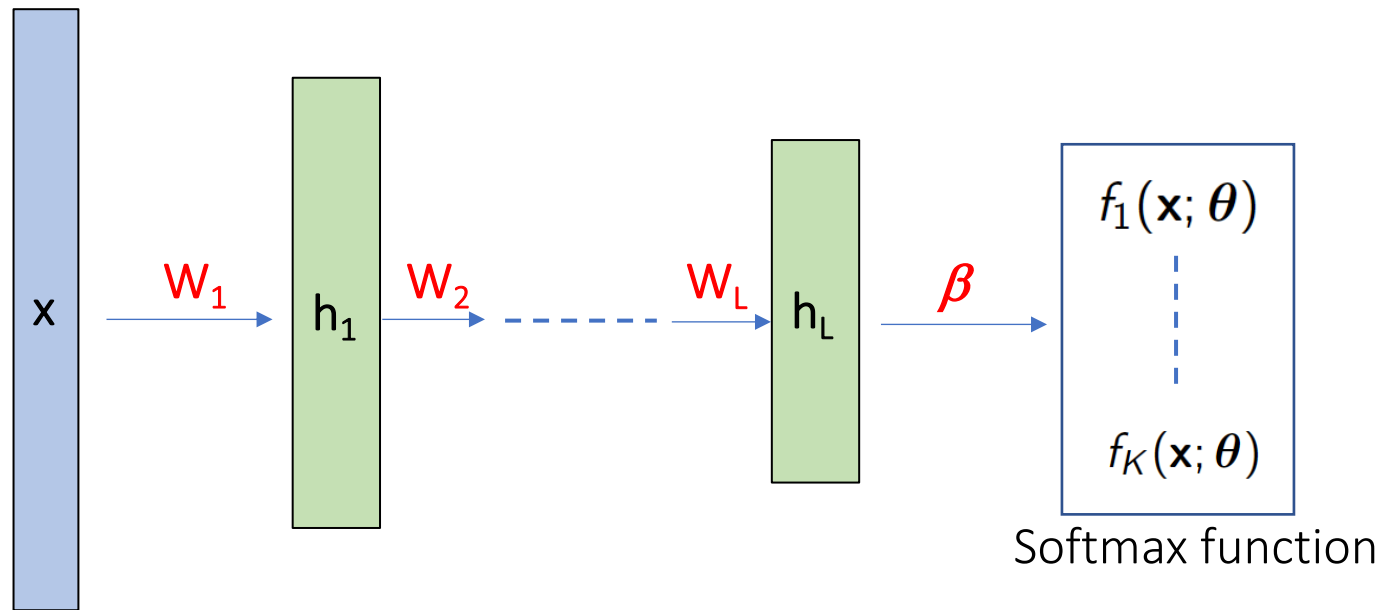
$$f_k(\mathbf{x}; \theta) = \text{softmax}(\beta_k \mathbf{h} + \beta_{0k})$$

Diagram illustrating the output layer equation $f_k(\mathbf{x}; \theta) = \text{softmax}(\beta_k \mathbf{h} + \beta_{0k})$ with annotations:

- $f_k(\mathbf{x}; \theta)$: Interpreted as $P(\text{class} = k \mid \mathbf{x})$
- $\text{softmax}(\cdot)$: Softmax function
- β_k : Weight vector for k th output of dimension $1 \times M$
- β_{0k} : Bias term for output k

Sidenote: the $g(\cdot)$ and $\text{softmax}(\cdot)$ non-linearities are applied element-wise to vectors

General Network with L Hidden Layers, K Outputs





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

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



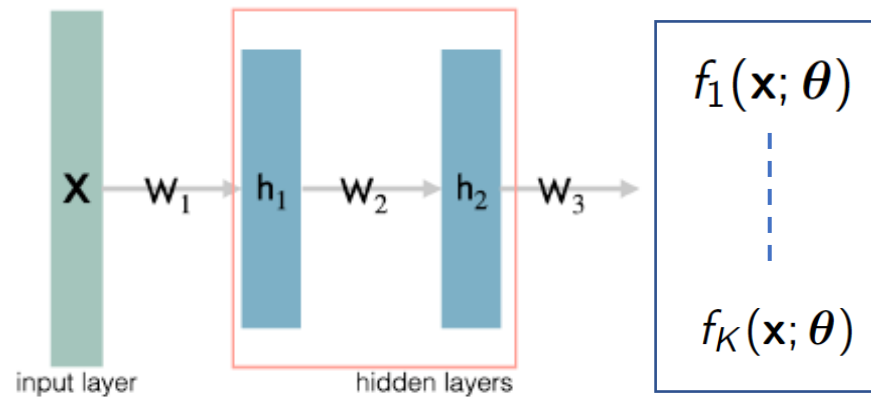
Recursive computation of vector of hidden unit values in each layer from hidden units in previous layer, $l = 2, \dots, L$

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



Softmax over outputs to produce k th output, corresponding to $P(\text{class} = k \mid \mathbf{x})$

MNIST Example



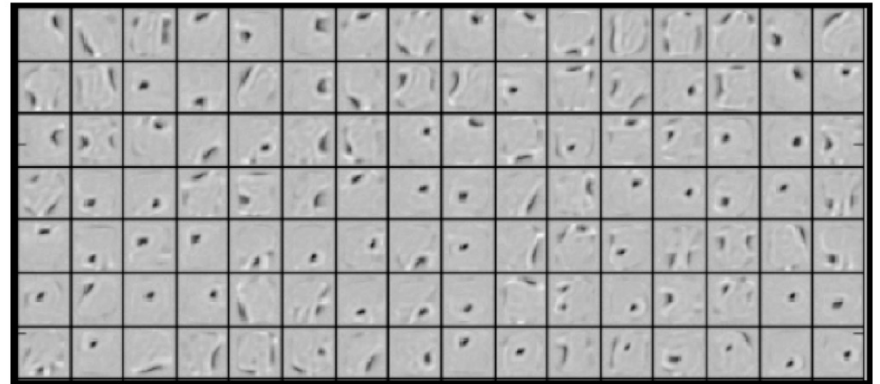
$K = 10$ classes

784-dimensional input (pixels)

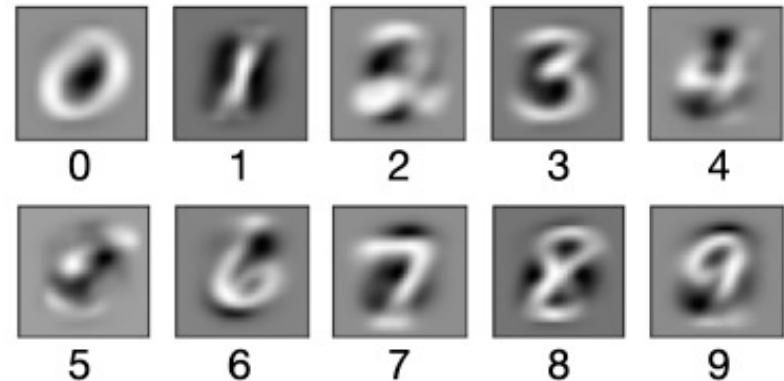
200 hidden units at each hidden layer

MNIST: What do the Hidden Units Learn?

Visualization of hidden unit weights
from a two hidden layer NN
(each square is a different hidden unit)



Visualization of logistic weights
from a logistic classifier



Feature representations
learned by a deep neural
network for face recognition

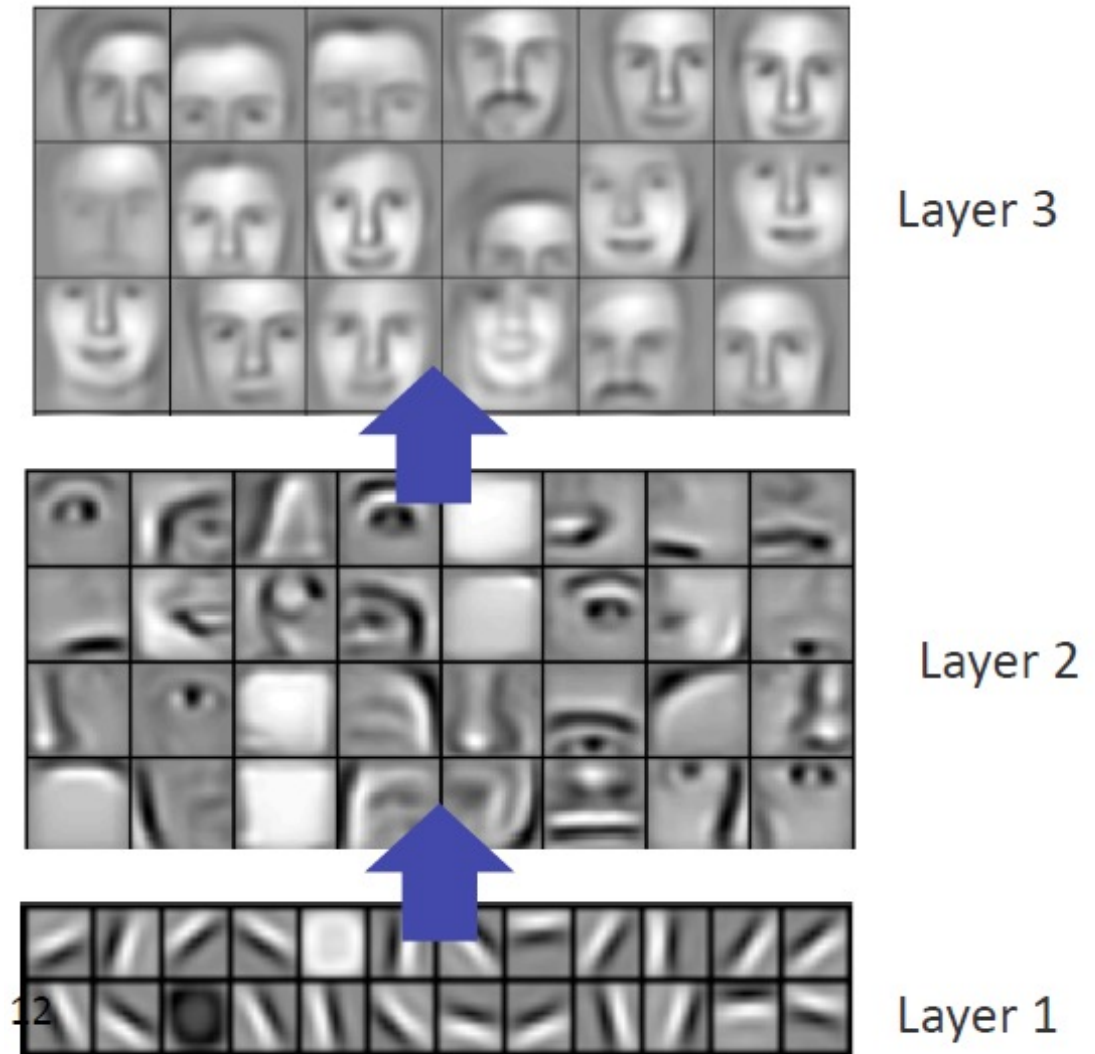
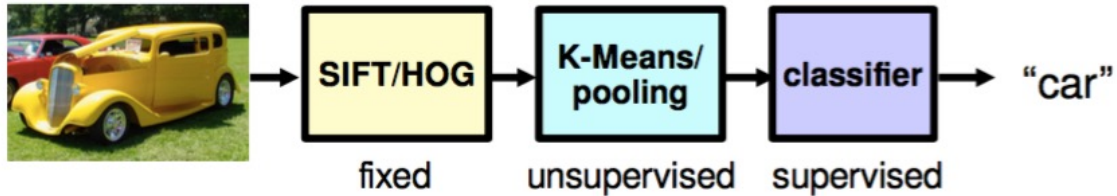


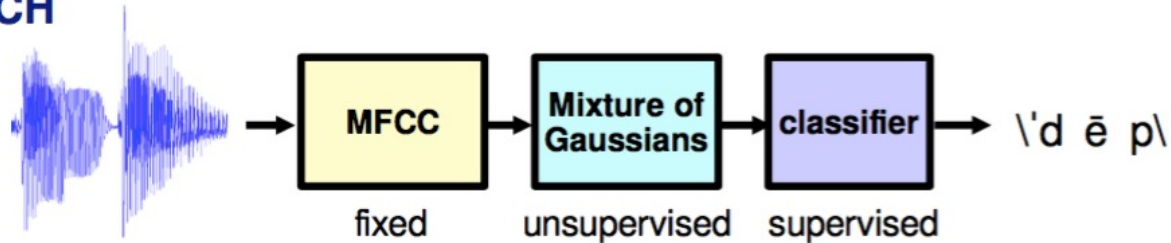
Figure from Lee et al., ICML 2009

Machine Learning before Deep Neural Networks

VISION



SPEECH



NLP

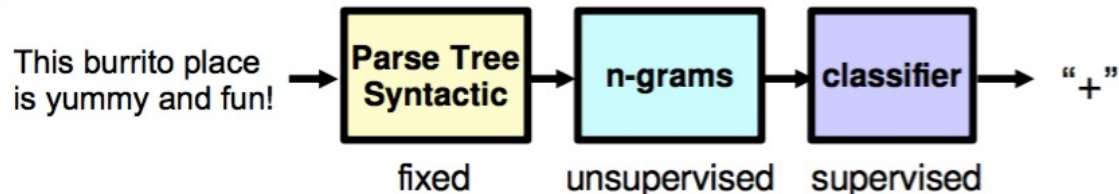


Figure from Marc'Aurelio-Ranzato

Questions?

Recap of Neural Networks

More Complex Networks

Network Architectures

Training Neural Networks

Neural Network Architectures and Computation

“Feedforward” neural networks consist of layers of matrix-vector operations with non-linearities at each hidden layer

Very suitable for processing on graphics processing units (GPU), which have been optimized for linear algebra

Real-world deep neural networks (e.g., for vision and speech) can be considerably more complex than our L-layer networks, e.g,

- residual connections
- convolutional operators
- attention mechanisms
-and so on.

Example: A Deep Neural Network for Digit Classification

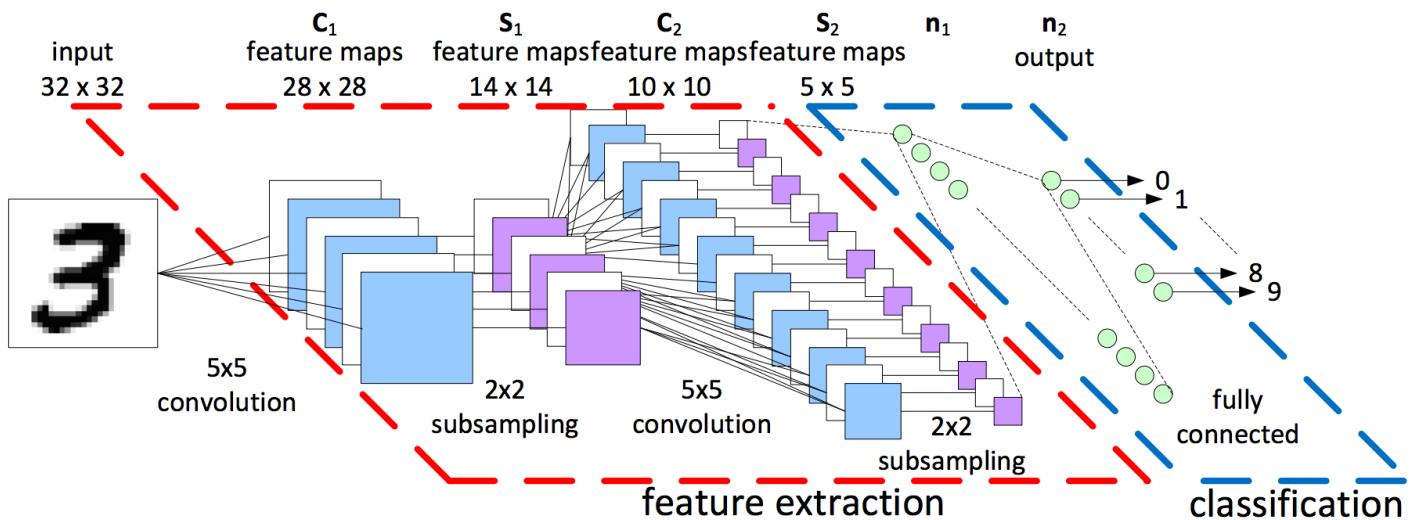
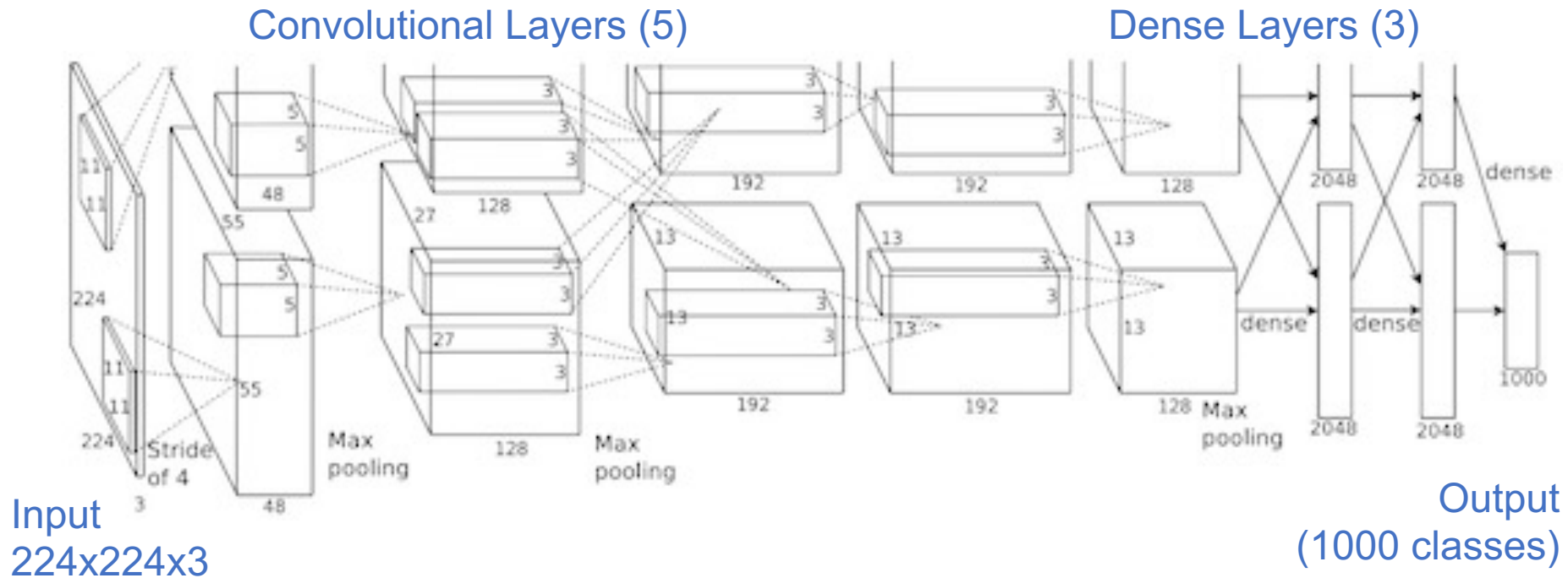


Figure from Peeman et al, 2012

Example: AlexNet (2012)

- Deep NN model for ImageNet classification
 - 650k units; 60m parameters
 - 1m data; 1 week training (GPUs)



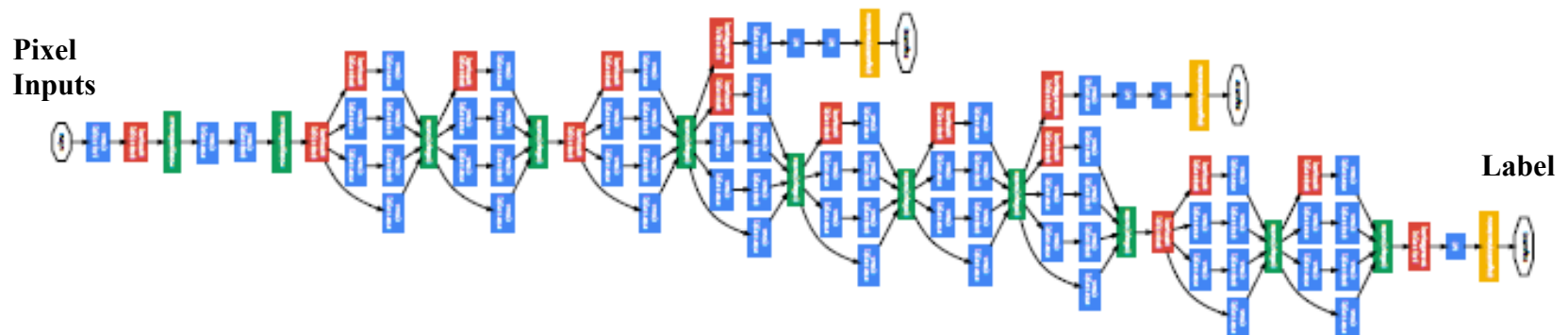
[Krizhevsky et al. 2012]

Example of a Complex Deep Neural Network GoogLeNet (2015) image recognition network

27 layers, ~4 million parameters

Much deeper than AlexNet

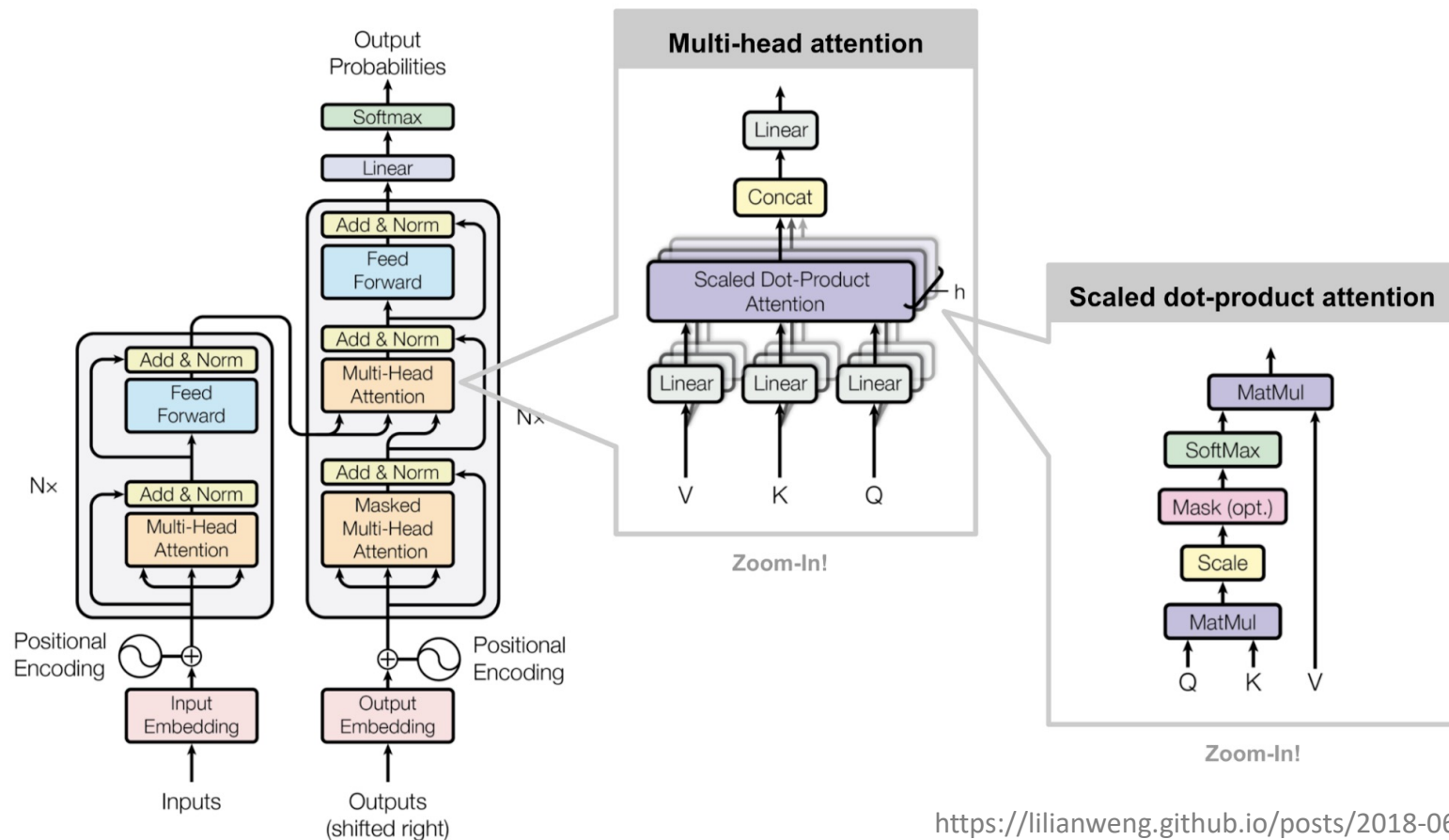
Uses “Inception” modules – fewer parameters



GPT-3

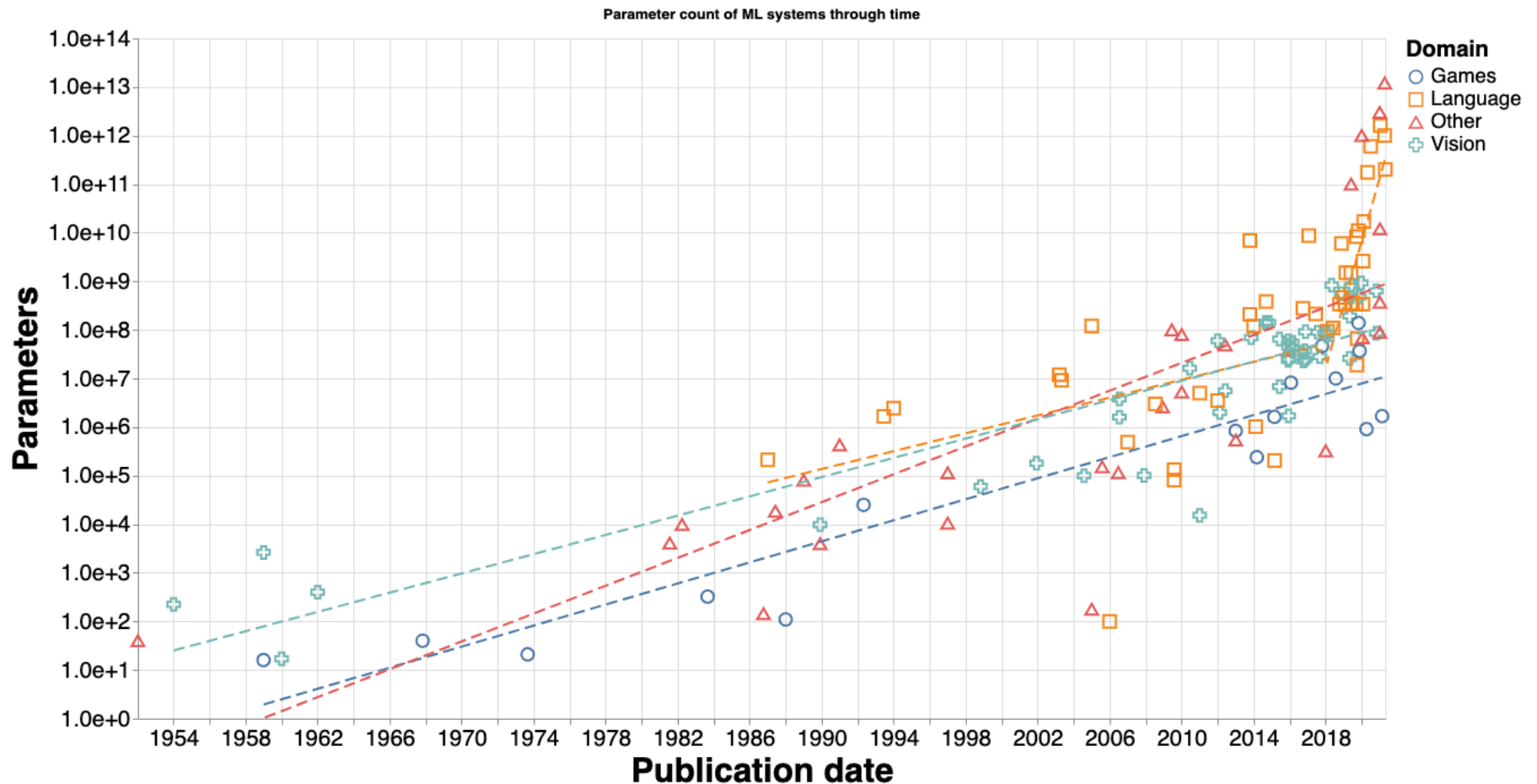
GPT-3 (e.g. ChatGPT) uses *transformers* (or *attention*)

- 175 **billions** parameters – 800GB of memory to store!



<https://lilianweng.github.io/posts/2018-06-24-attention/>

Networks are Bigger and Better



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

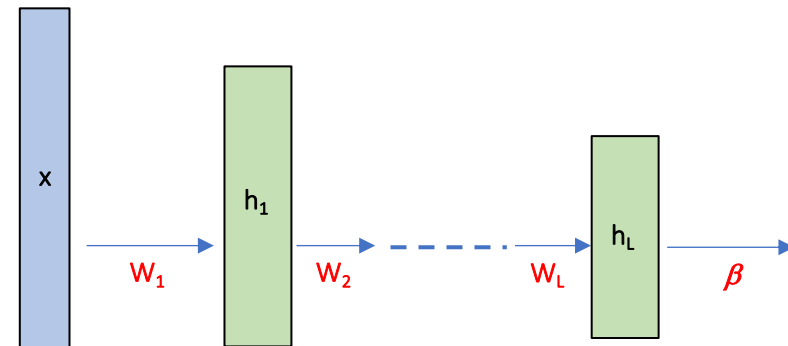
Number of Parameters in L-Layer Network

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 let's ignore bias terms

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

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

=> Number of parameters p would be

about $300 \times 10^4 + 9 \times (300)^2 + 300 \times 1000$, which is approximately 4 million

Non-Linearities: the ReLU Function

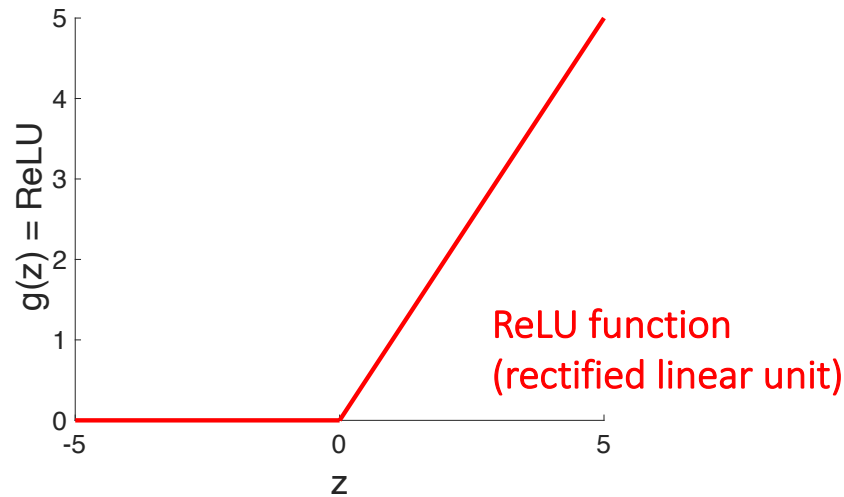
Its typical for neural network classifiers to use softmax at the output layer

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

The ReLU function is very simple:

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

$$z \leq 0 : g(z) = 0$$



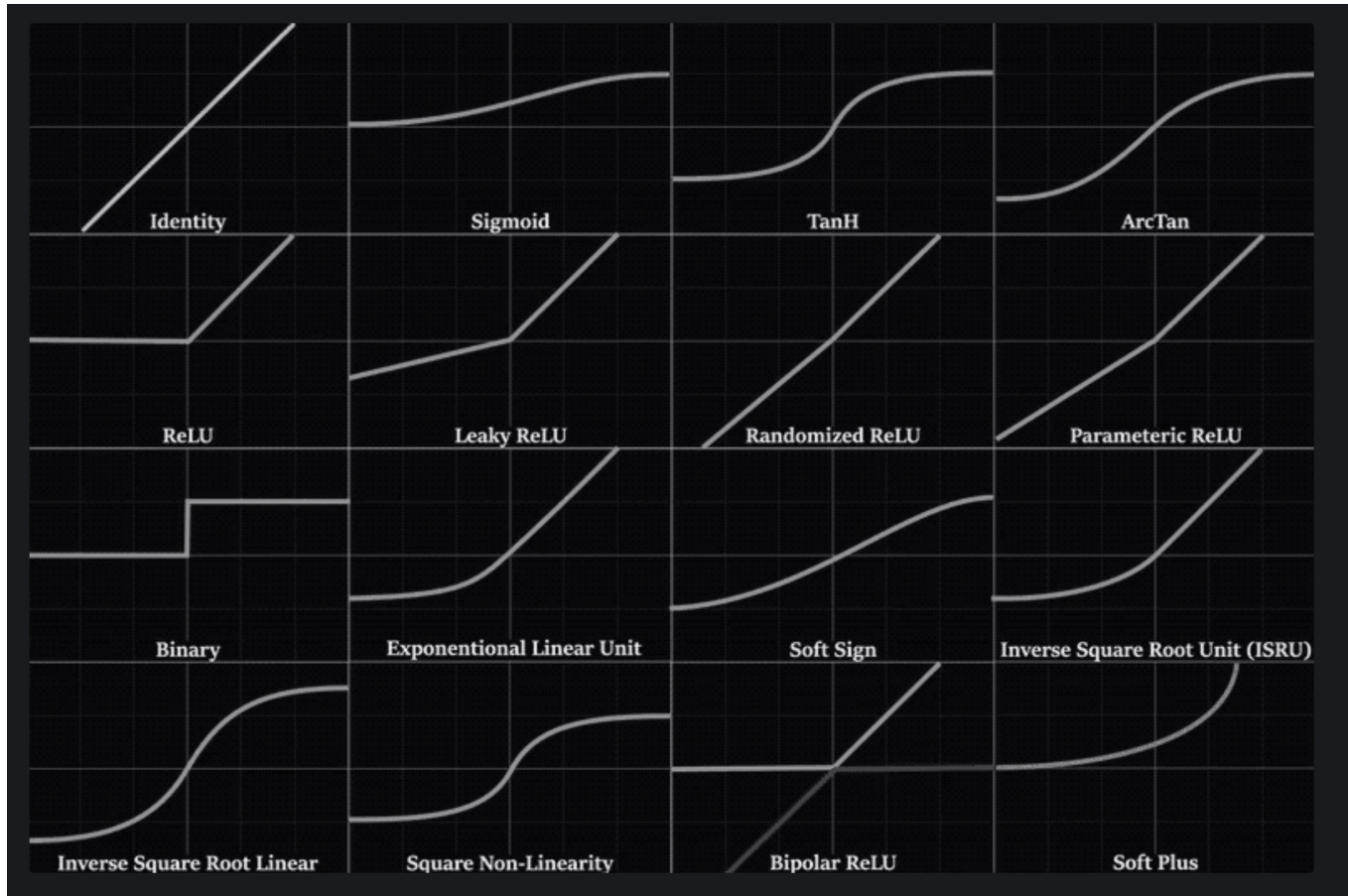
Why is it popular?

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/eaq5c/activation_functions_cheat_sheet/

Questions?

Recap of Neural Networks

More Complex Networks

Network Architectures

Training Neural Networks

Learning a Neural Network Classifier

Log-loss function 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.

Cross-entropy loss function $L(\theta)$

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

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

Recall that minimizing $-\log P$ 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 Log-Loss for a Neural Network

How can we minimize the log-loss? We can use gradient descent again!

Updates for parameters θ in the gradient descent algorithm:

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

where


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

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

θ 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):

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


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

...but with repeated application of the chain rule, the computation of gradients in a neural network can be done straightforwardly

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,

i.e., backpropagation algorithm = gradient descent for a neural network

(we will see more details next lecture)

Wrapup

- Neural networks are *feature extractors*
 - Internal representations learned from data
- NNs have very many parameters
 - Larger and larger over time
 - 10s of millions in 2012 (AlexNet)
 - Modern networks can have *~100 billion* parameters
- NNs have a large number of hyperparameters
 - Number of hidden units?
 - Number of layers?
 - Activation functions?
 - Even more that we haven't seen yet
- Next lecture:
 - Training neural networks:
 - Backprop and stochastic gradient descent