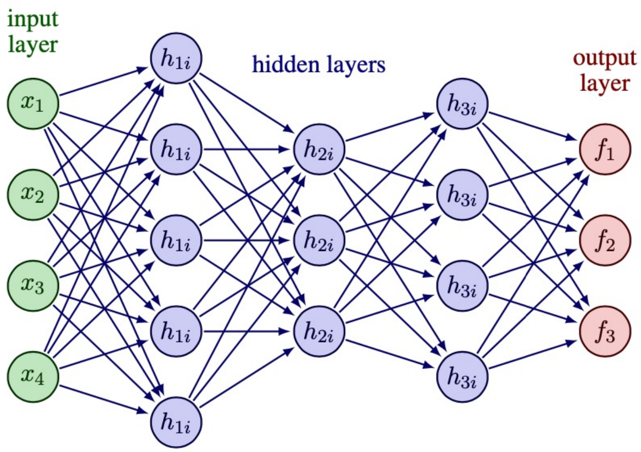
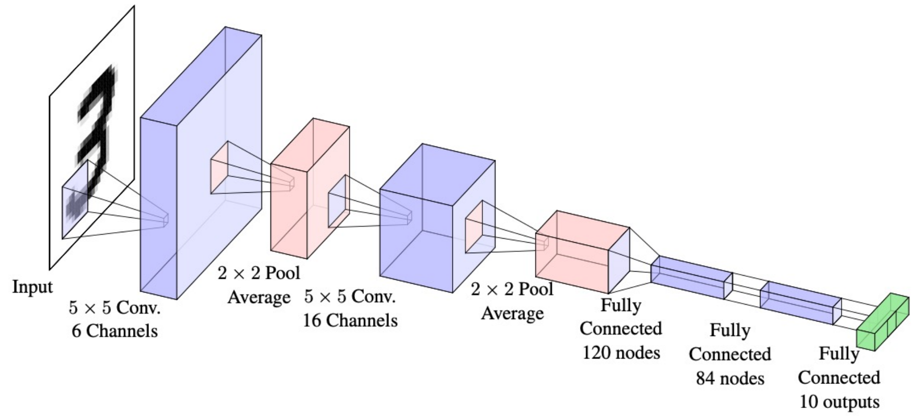
CS273A:
Neural Networks





Prof. Alexander Ihler

Fall 2024

Neural Networks

Multi-Layer Perceptrons

Backpropagation Learning

Architectures

Convolutional	Residual	Attention

Training Deep Networks

More Tricks: Dropout, BatchNorm

Neural Networks

Multi-Layer Perceptrons

Backpropagation Learning

Architectures

Convolutional	Residual	Attention

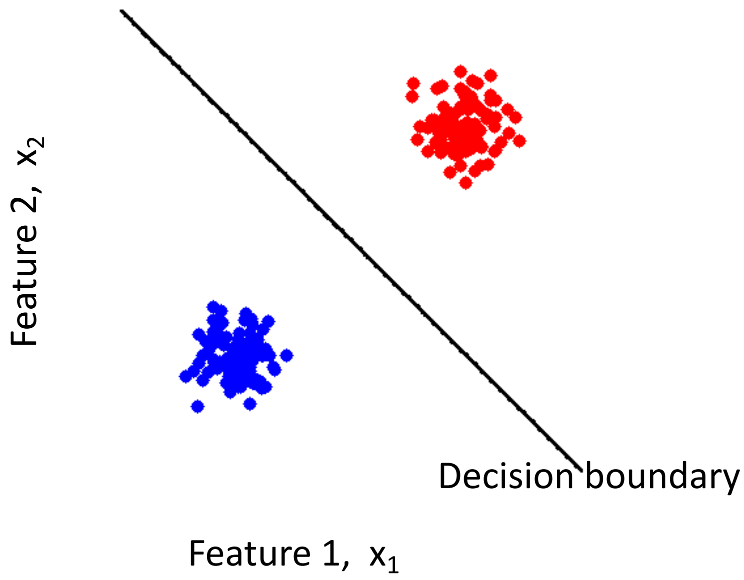
Training Deep Networks

More Tricks: Dropout, BatchNorm

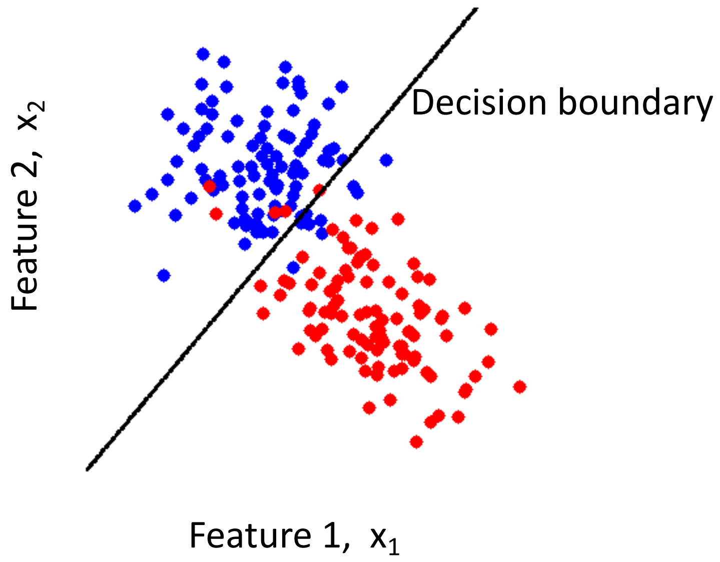
Linear classifiers (perceptrons)

- Linear Classifiers
  - a linear classifier is a mapping which partitions feature space using a linear function (a straight line, or a hyperplane)
  - separates the two classes using a straight line in feature space
  - in 2 dimensions the decision boundary is a straight line

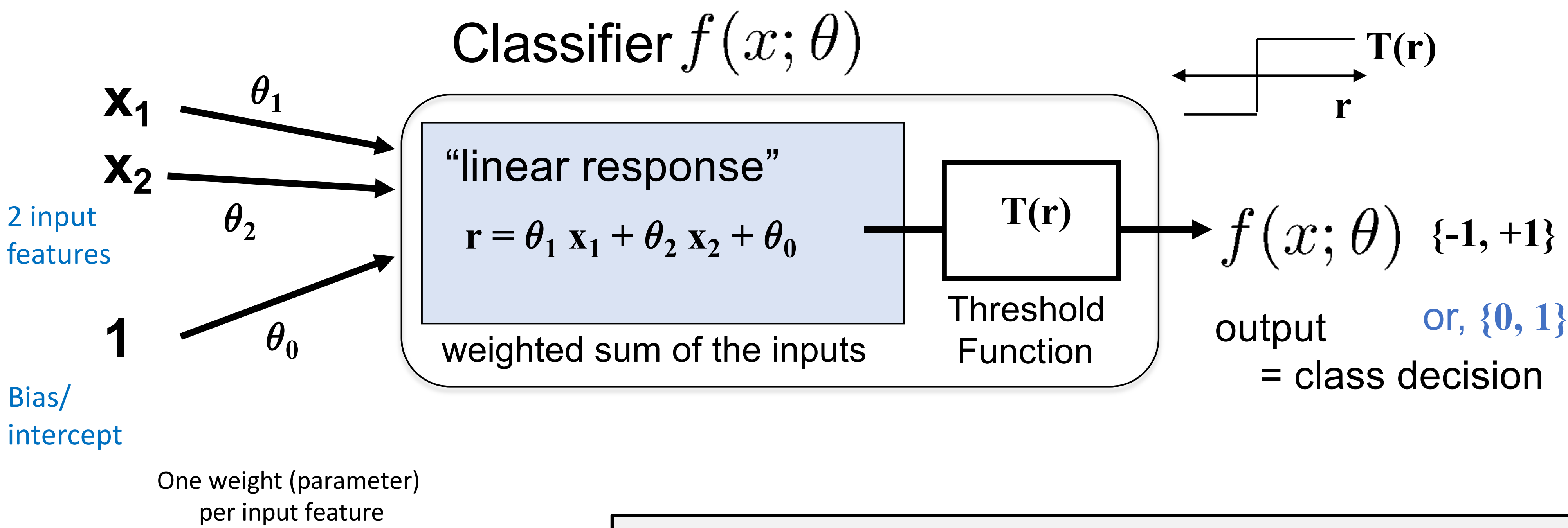
Linearly separable data



Linearly non-separable data



Linear Classifier (2 features)




```
r = X @ theta.T          # compute linear response
yhat = 2*(r > 0)-1       #  "sign": predict +1 / -1
```

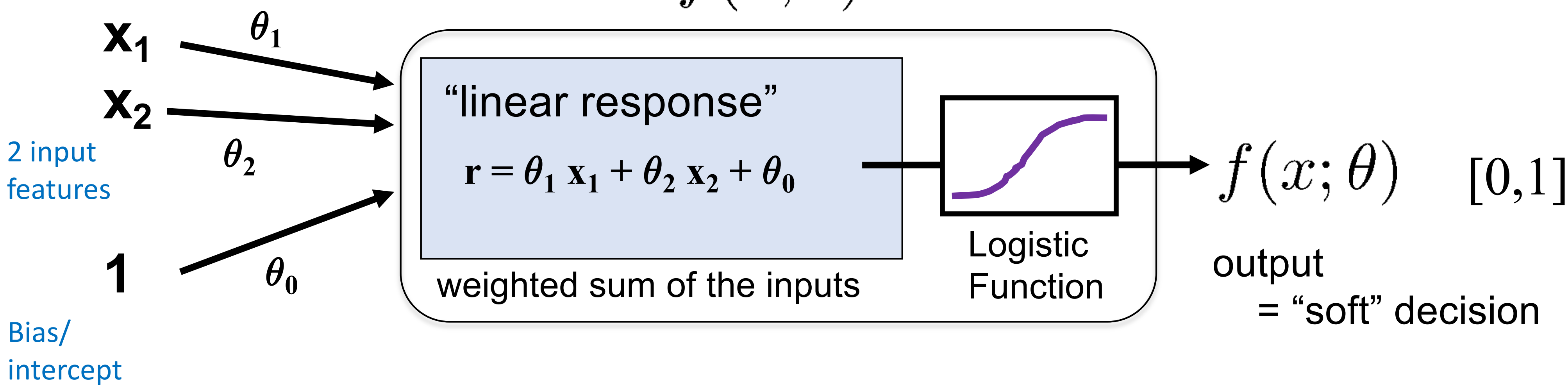
If r(x) > 0, predict "positive" (class +1)
If r(x) < 0, predict "negative" (class -1) (or 0)

Decision Boundary at r(x) = 0

Solve: $x_2 = -w_1/w_2\ x_1 - w_0/w_2$ (Line)

Training: Logistic Regression

Classifier $f(x;\theta)$

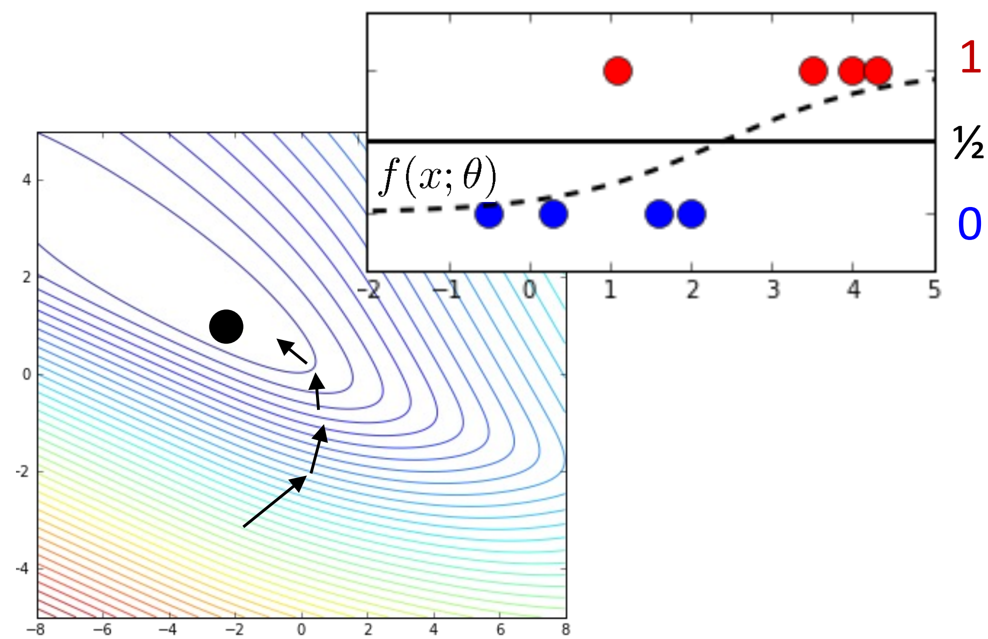


One weight (parameter) per input feature

Log-loss (Logistic NLL)

- Interpret output as probabilities
- Prediction loss is -log Pr(true class)
- Train via gradient descent

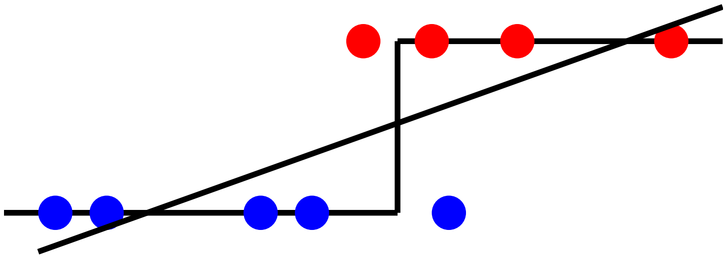
Generalizes to more classes by having a set of weights for each class.



Features and perceptrons

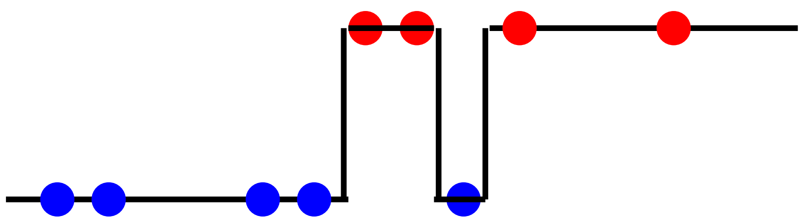
- Recall the role of features
  - Create extra features to allow more complex decision boundaries

  - Linear classifiers
  - Features [1,x]
    - Decision rule: $T(ax+b) = w x + b > 0$
    - Boundary $ax+b = 0$ => point



  - Features $[1,x,x^2]$
    - Decision rule $T(w_2 x^2 + w_1 x + b)$
    - Boundary $w_2 x^2+w_1 x + b = 0$ = ?

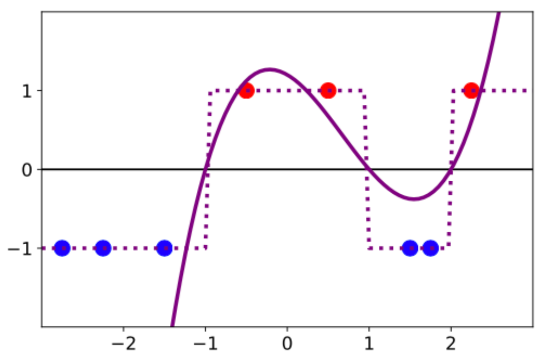
  - What features can produce this decision rule?

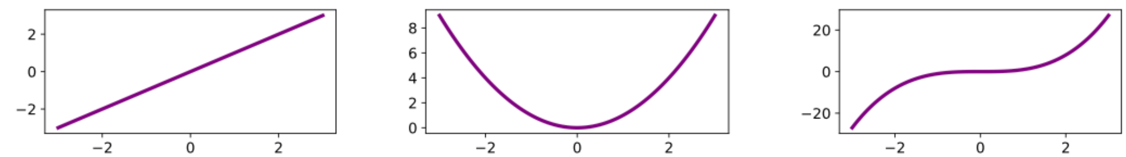


Features and perceptrons

- Recall the role of features
  - Extra features can allow more complex decision boundaries
  - For example, polynomial features

    $\Phi(x) = [1 \;\; x \;\; x^2 \;\; x^3 \;\ldots]$

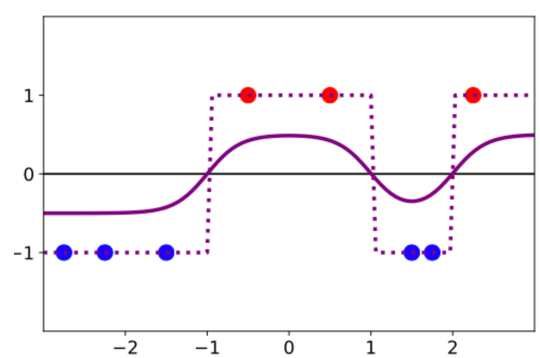


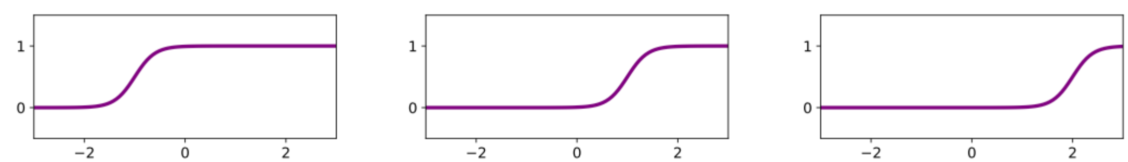


$\phi_1(x) = x$ $\qquad \phi_2(x) = x^2$ $\qquad \phi_3(x) = x^3$

$$r(x) = b + w_1\,\phi_1(x) + w_2\,\phi_2(x) + w_3\,\phi_3(x)$$

- What other kinds of features could we choose?



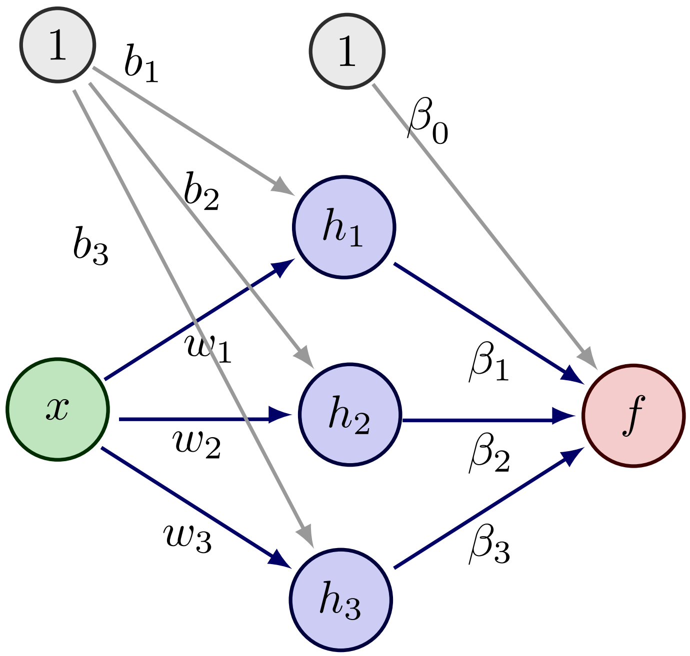


$\phi_1(x) = \sigma(5x + 5)$ $\qquad \phi_2(x) = \sigma(5x - 5)$ $\qquad \phi_3(x) = \sigma(5x - 10)$

$$r(x) = b + w_1\,\phi_1(x) + w_2\,\phi_2(x) + w_3\,\phi_3(x)$$

Multi-layer perceptron model

- These features are just perceptrons!
  - Feature transform is a collection of perceptrons
  - Combination of features output of another



Logistic sigmoid:

$$\sigma(r) = \frac{1}{1 + \exp(-r)}$$

$$h_1(x) = \sigma(\ b_1 + w_1 x\ )$$

$$h_2(x) = \sigma(\ b_2 + w_2 x\ )$$

$$h_3(x) = \sigma(\ b_3 + w_3 x\ )$$

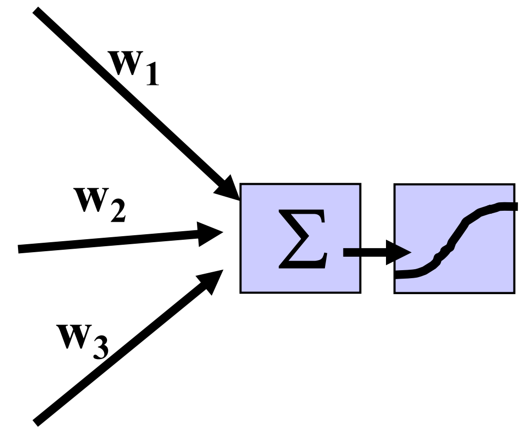
$$f(x) = \sigma(\ \beta_0 + \beta_1 h_1 + \beta_2 h_2 + \beta_3 h_3\ )$$

Regression version:

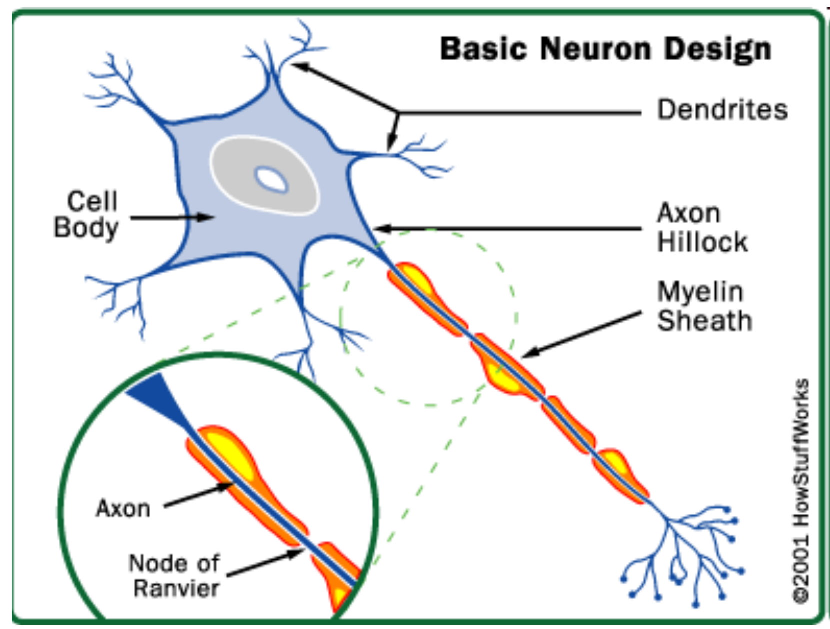
$$f(x) = \beta_0 + \beta_1 h_1 + \beta_2 h_2 + \beta_3 h_3$$

Neural networks

- Another term for Multi-Layer Perceptrons
- Biological motivation



- Neurons
  - “Simple” cells
  - Dendrites sense charge
  - Cell weighs inputs
  - “Fires” axon



[“How stuff works: the brain”]

A Little History on Neural Networks

- Phase 1, 1950s to 1970s
  - Logistic-like models, no hidden units
  - Initial enthusiasm died out

- Phase 2, 1980s to 2000s
  - Invention of backpropagation: could train models with hidden units
  - But training was slow, data was scarce… initial enthusiasm died out

- Phase 3, 2010s to present
  - Demonstrations of the power of deep learning models
  - (re)invention of a technique called stochastic gradient
  - Commercial successes, great enthusiasm…

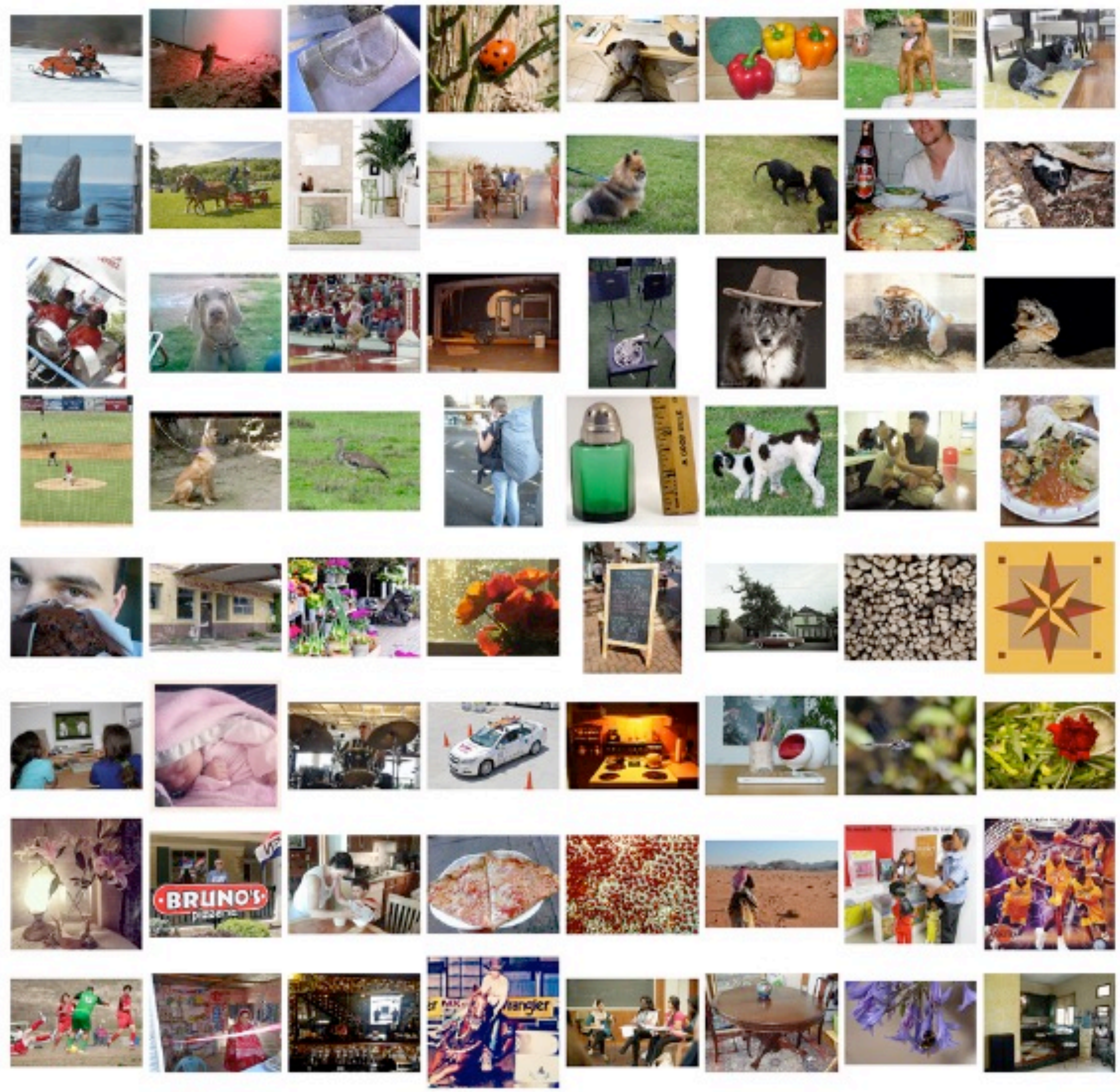
Applicaton: Image Recognition

ImageNet

A testbed for evaluating image classification algorithms

Over 10 million images

1000 class labels



From Russakovsky et al., ImageNet Large Scale Visual Recognition Challenge, 2015

Application: Image Recognition

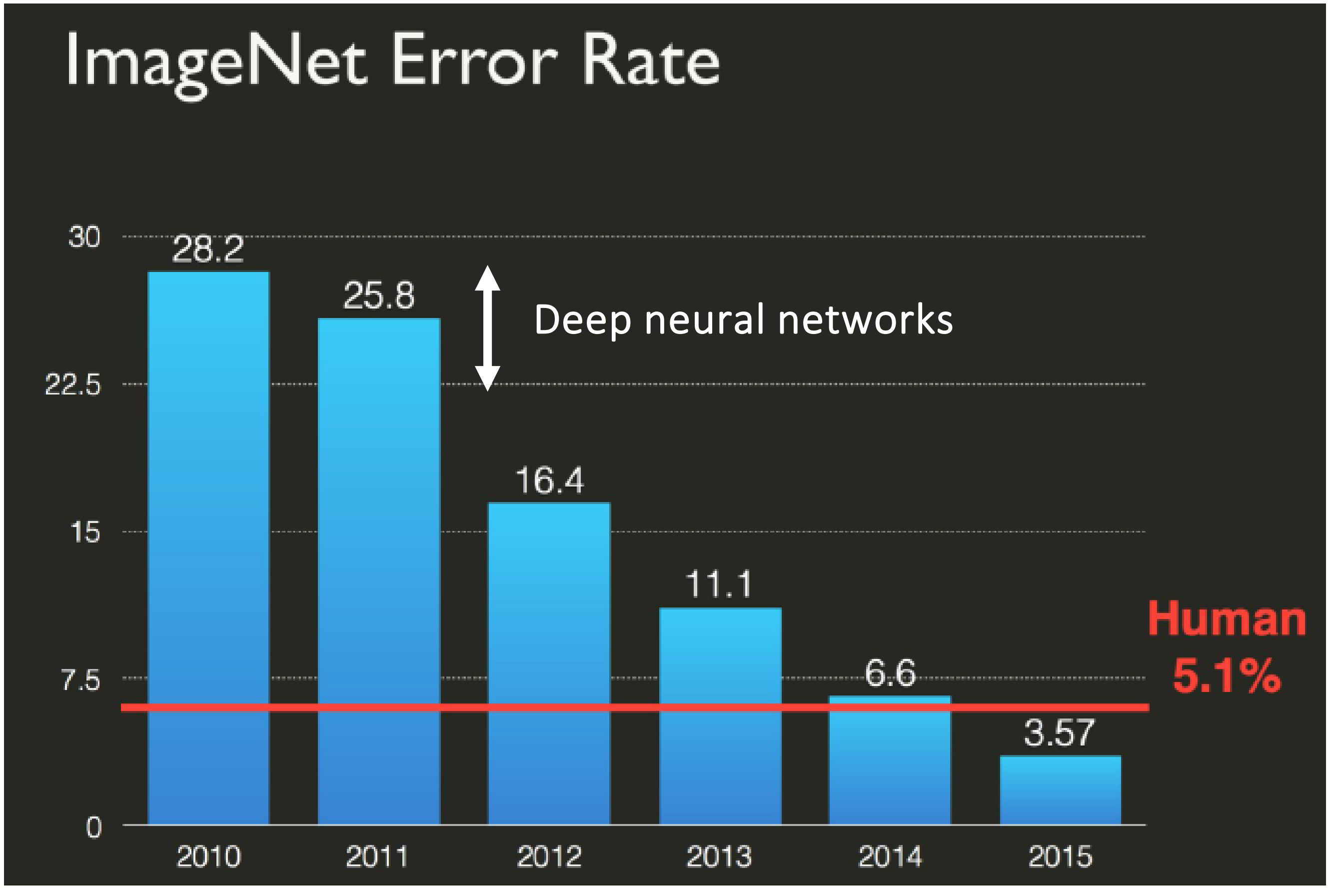
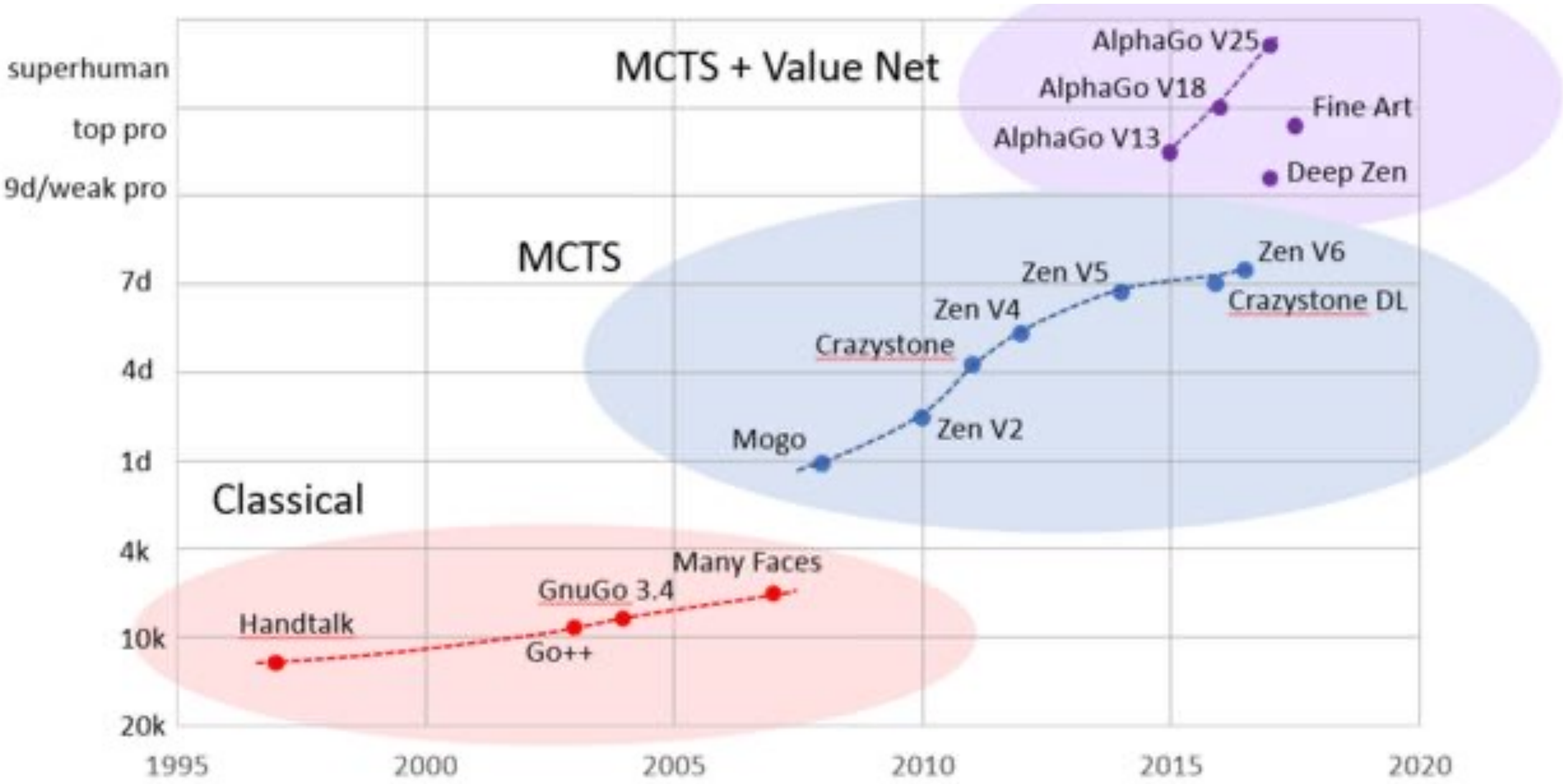


Figure from Kevin Murphy, Google

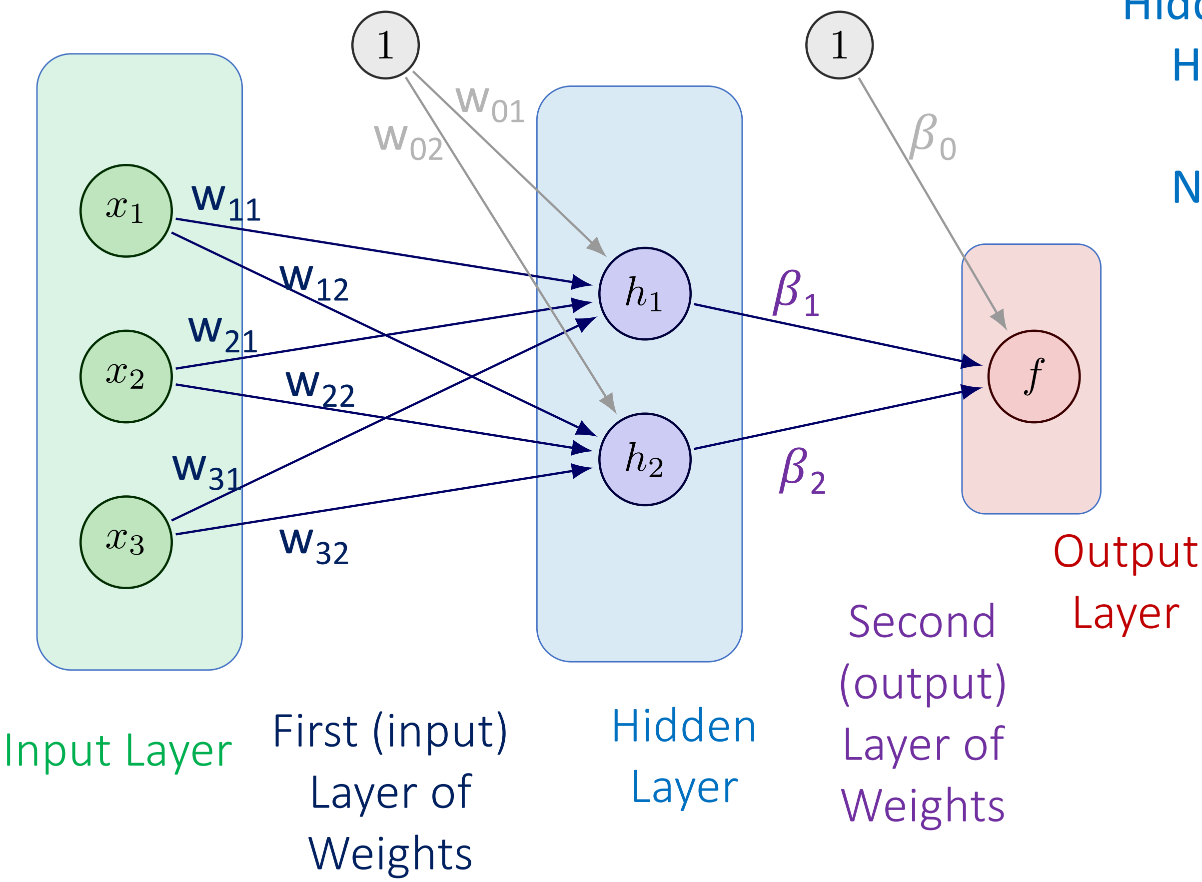
Application: Go



https://www.reddit.com/r/baduk/comments/6ttyyz/better_graph_of_go_ai_strength_over_time/

Two-layer Neural Network

- "Two layer" = 1 hidden layer, 1 output layer

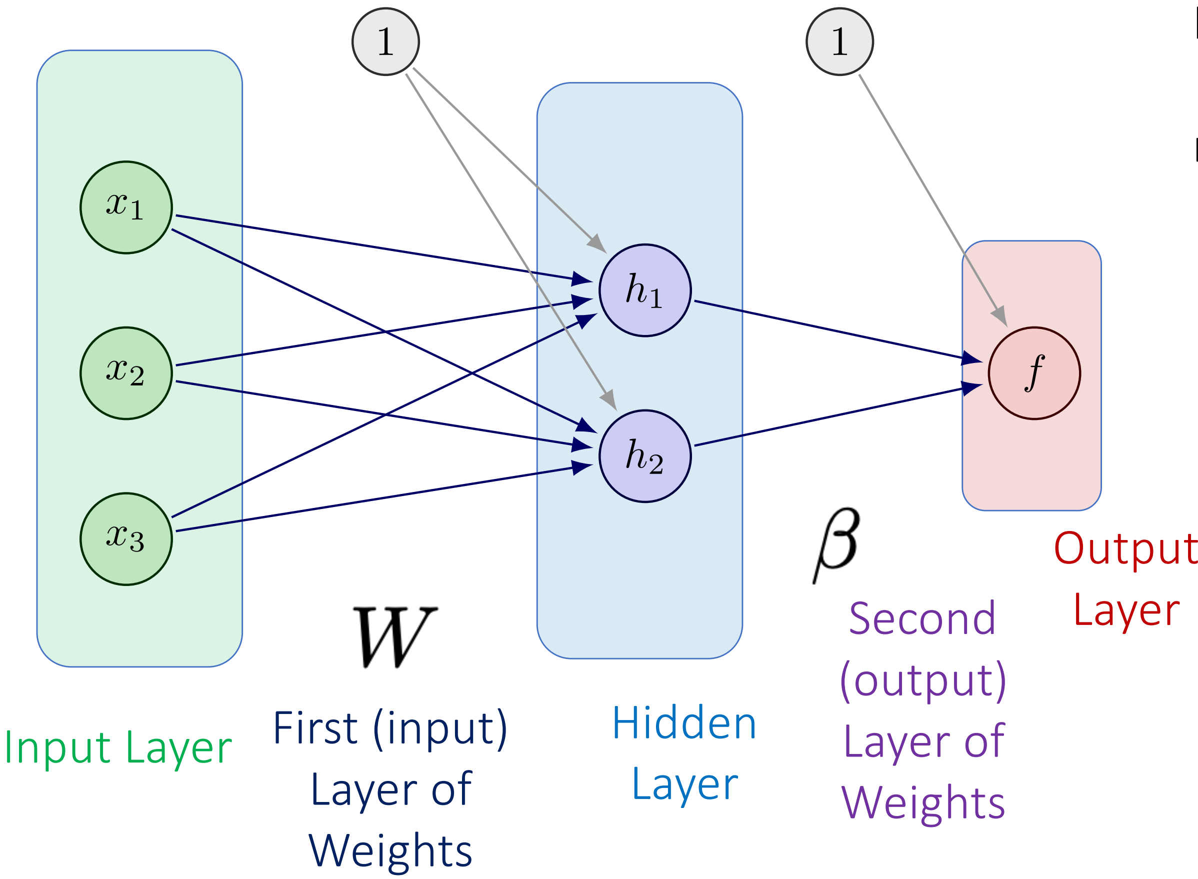


"Hidden units" = non-linear f'ns of inputs
Here, logistic sigmoid
(but other options often used)
Non-linear function is called
the "activation function"

We can think of neural networks in terms of "layers" (of nodes, and of weights)

Neural Network Parameters

- Parameters θ = {all weights & biases}



How many parameters total?

n features, H hidden nodes, 1 output

Layer 1 weights: n H
Layer 1 biases: H
Layer 2 weights: H
Layer 2 biases: 1

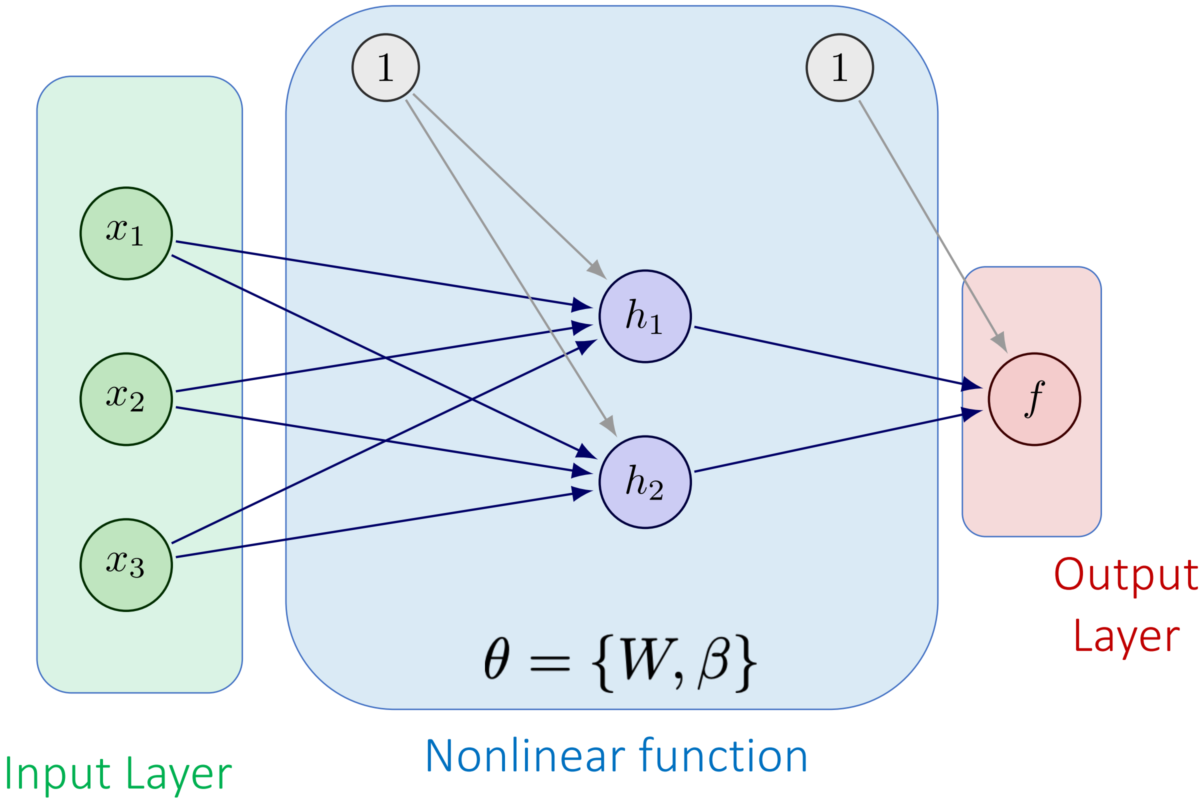
Total: H (n+1) + (H+1)

(approximately quadratic in layer sizes)

w_{ij} = weight of feature x_i in response of hidden node j

Neural Network Classifier

- Defines nonlinear mapping f(x ; θ)



Output values in [0,1]

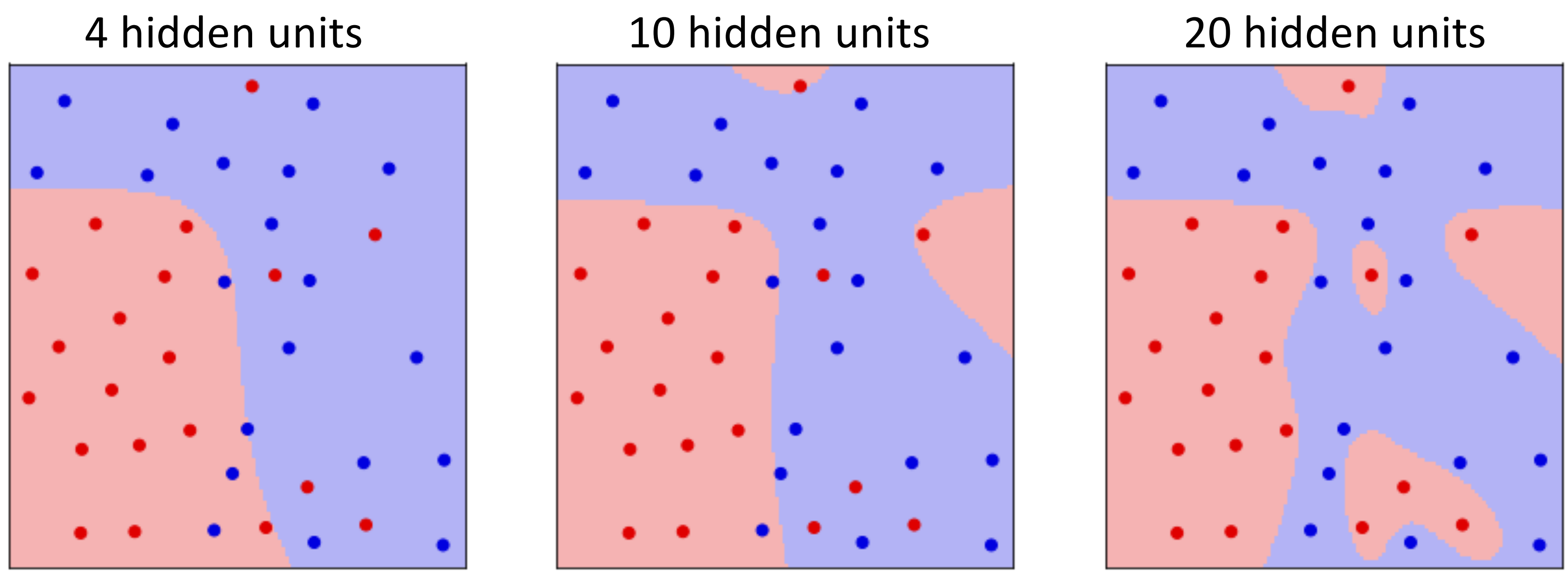
We can interpret these as e.g., class probabilities, p(y = 1 | x) & select log-loss (negative log-likelihood)

Or, we can train on MSE, $(y - f)^2$

Once we select a loss function, we can train the model!

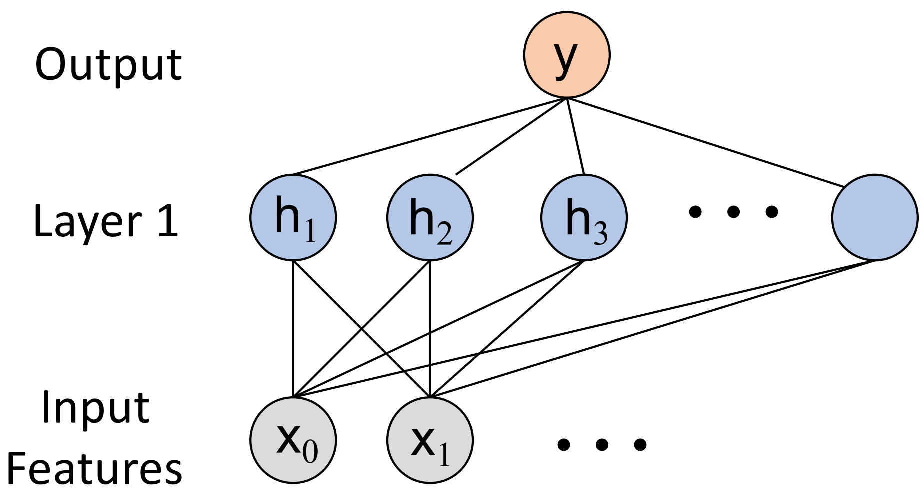
Neural Network Classifiers

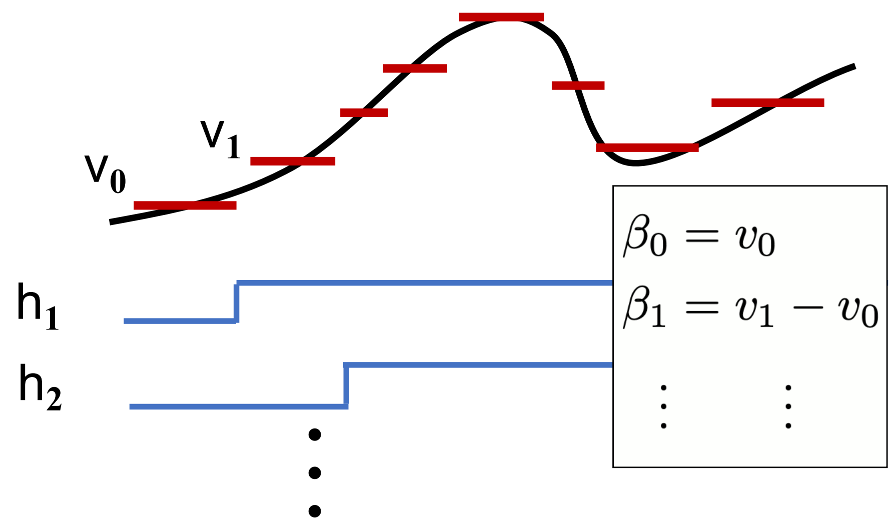
- Decision boundaries are non-linear
- Complexity depends on the number of layers and hidden nodes



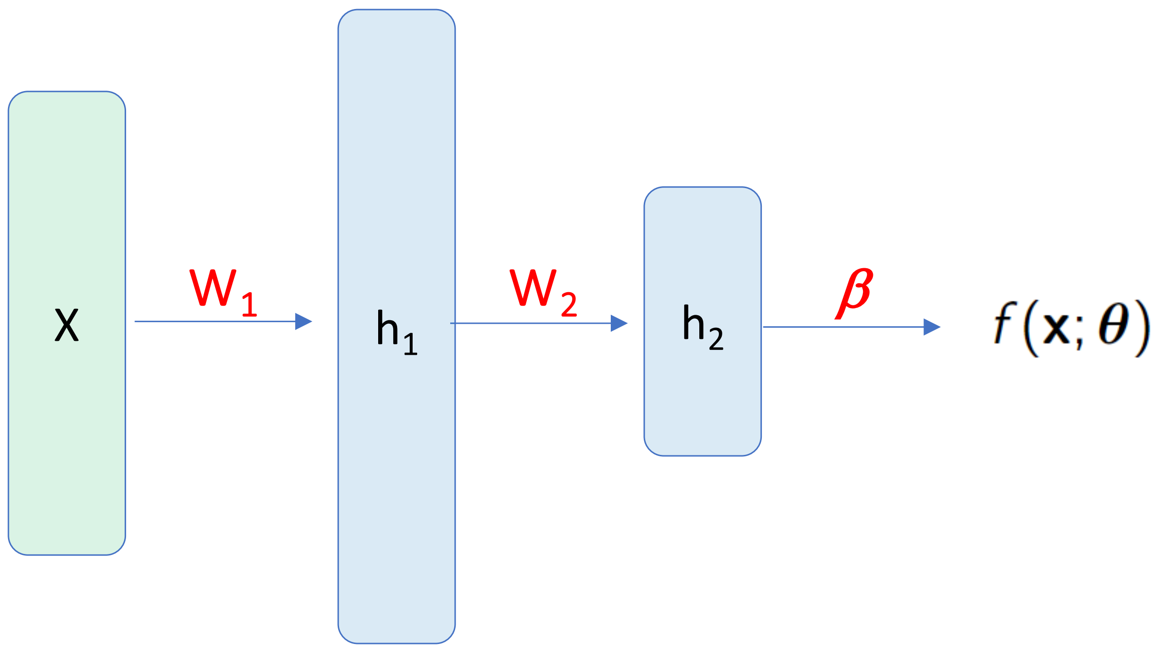
Features of MLPs

- Simple building blocks
  - Each element is just a perceptron function
- Can build upwards
- Flexible function approximation
  - Approximate arbitrary functions with enough hidden nodes



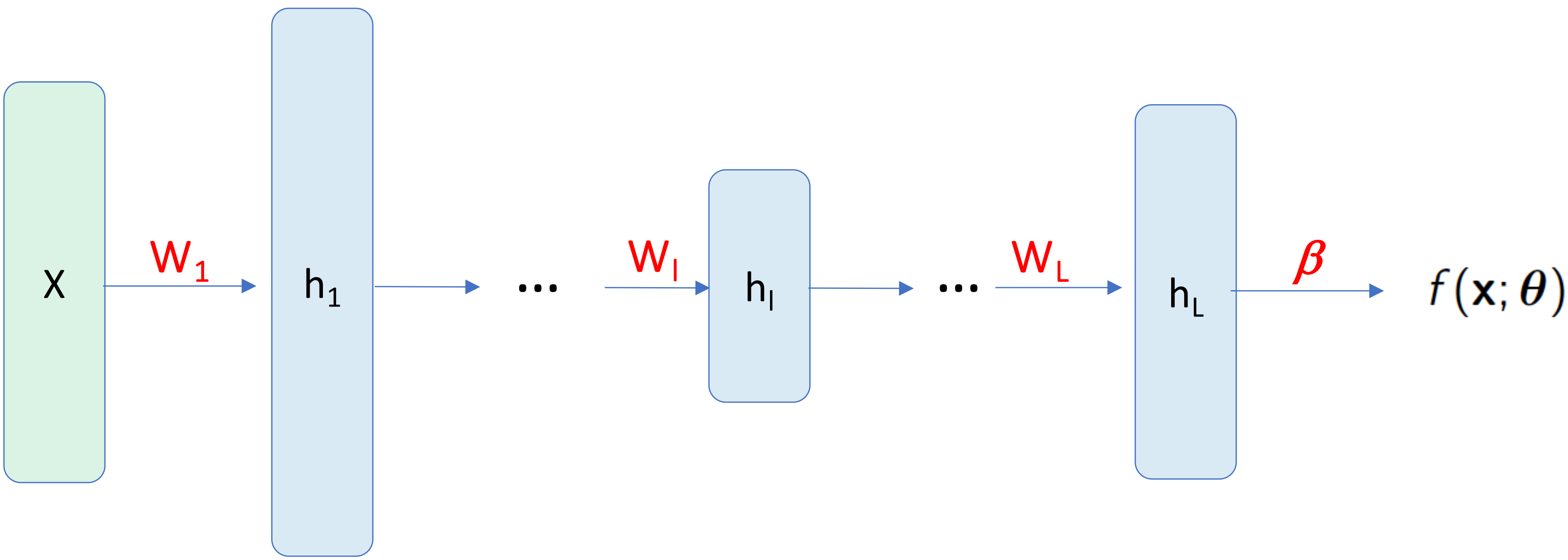


Networks with Two Hidden Layers



Each hidden unit layer can have different numbers of hidden units

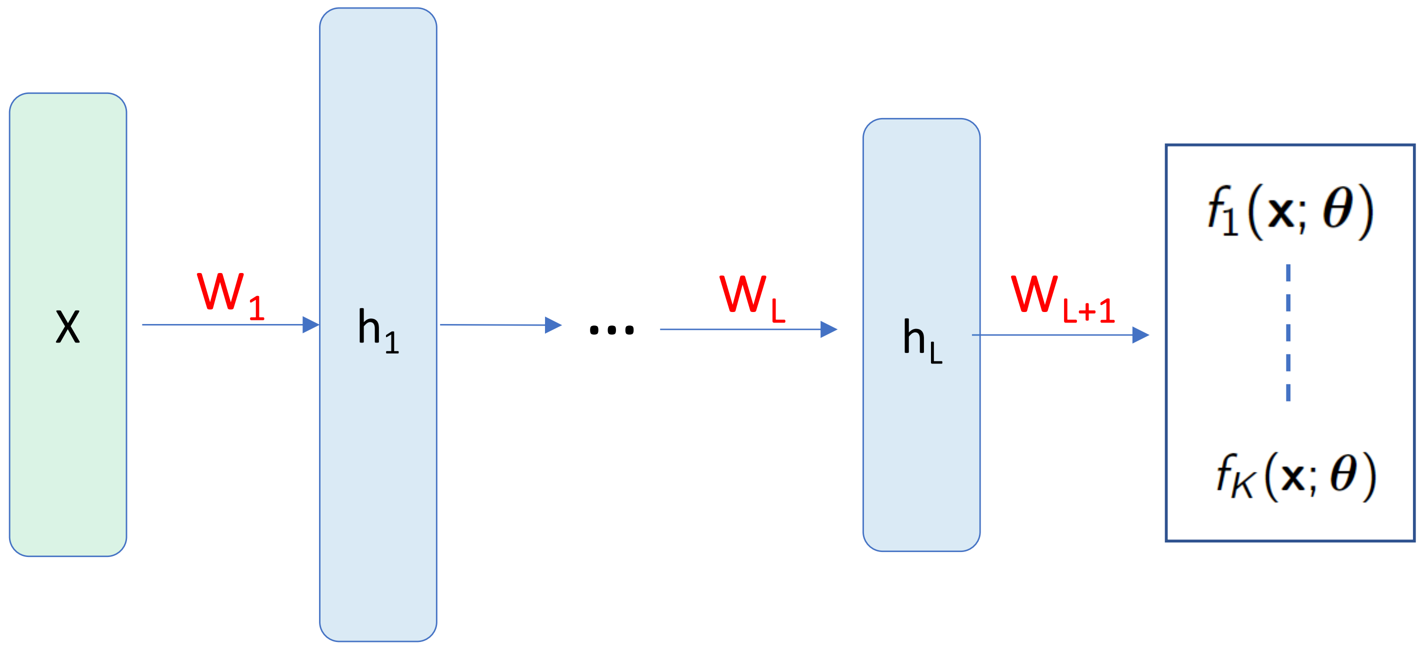
Networks with L Hidden Layers



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-class logistic classifier)

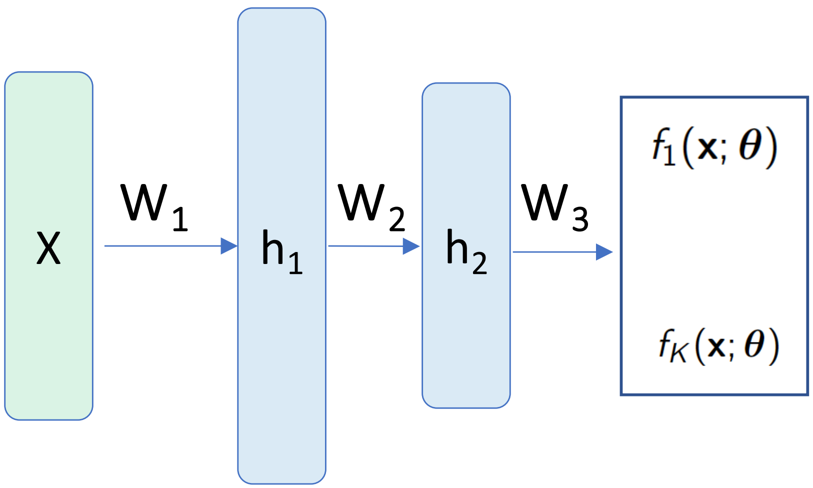
$$f_k(\underline{r}) = \frac{\exp(r_k)}{\sum_{k'} \exp(r_{k'})} \qquad \left(\text{so, } \sum_k f_k = 1 \right)$$

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

(assumes only one class is correct; compare to, say, image tagging)

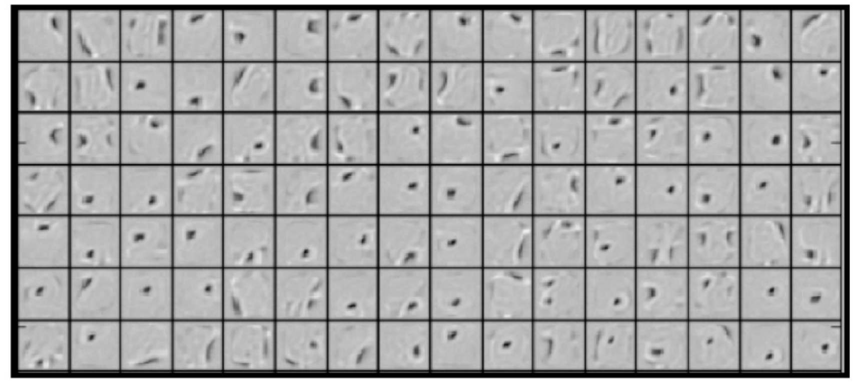
Example: MNIST Data

3-Layer Neural Network



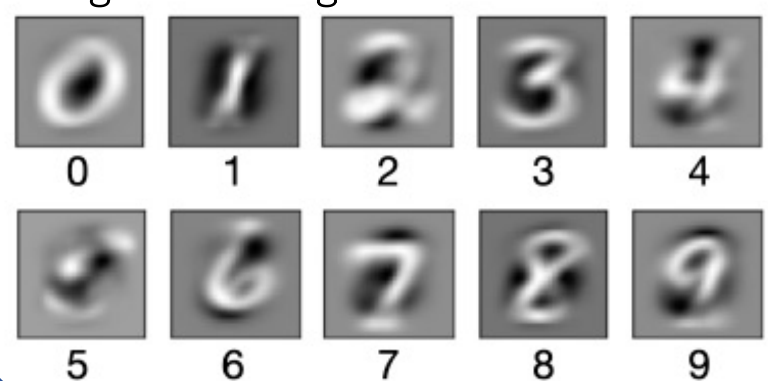
K = 10 classes
784-dimensional input (pixels)
200 hidden units at each hidden layer

What do the hidden nodes learn?



(each square is a different hidden unit)

For comparison: visualizing the weights of a logistic classifier:

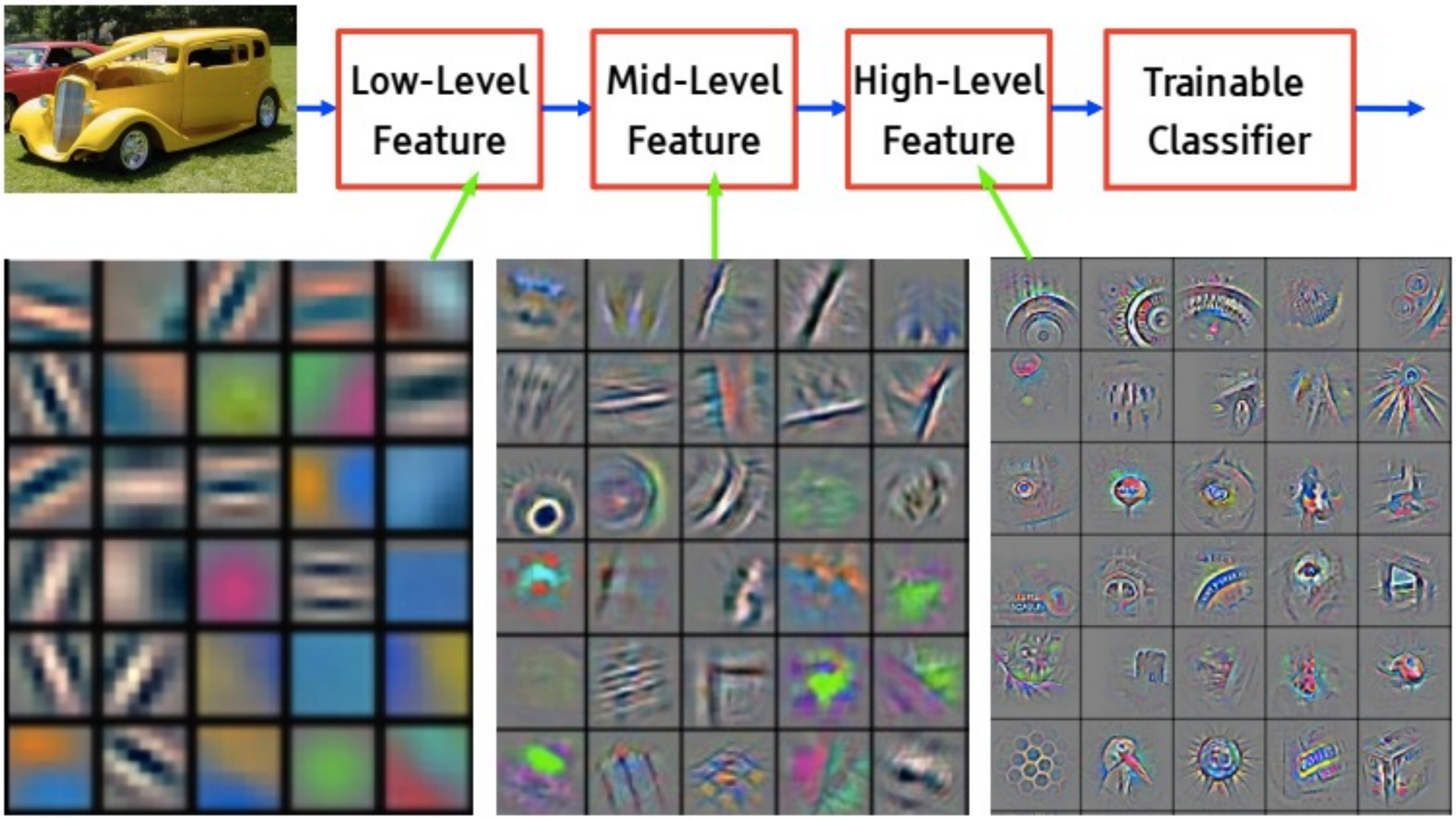


[Nalisnick et al., 2023]

Example: Visual Recognition

- Visualizing a convolutional network's filters

[Zeiler & Fergus 2013]



Slide image from Yann LeCun:
https://drive.google.com/open?id=0BxKBnD5y2M8NclFWSXNxa0JlZTg

Machine Learning before Deep Neural Networks

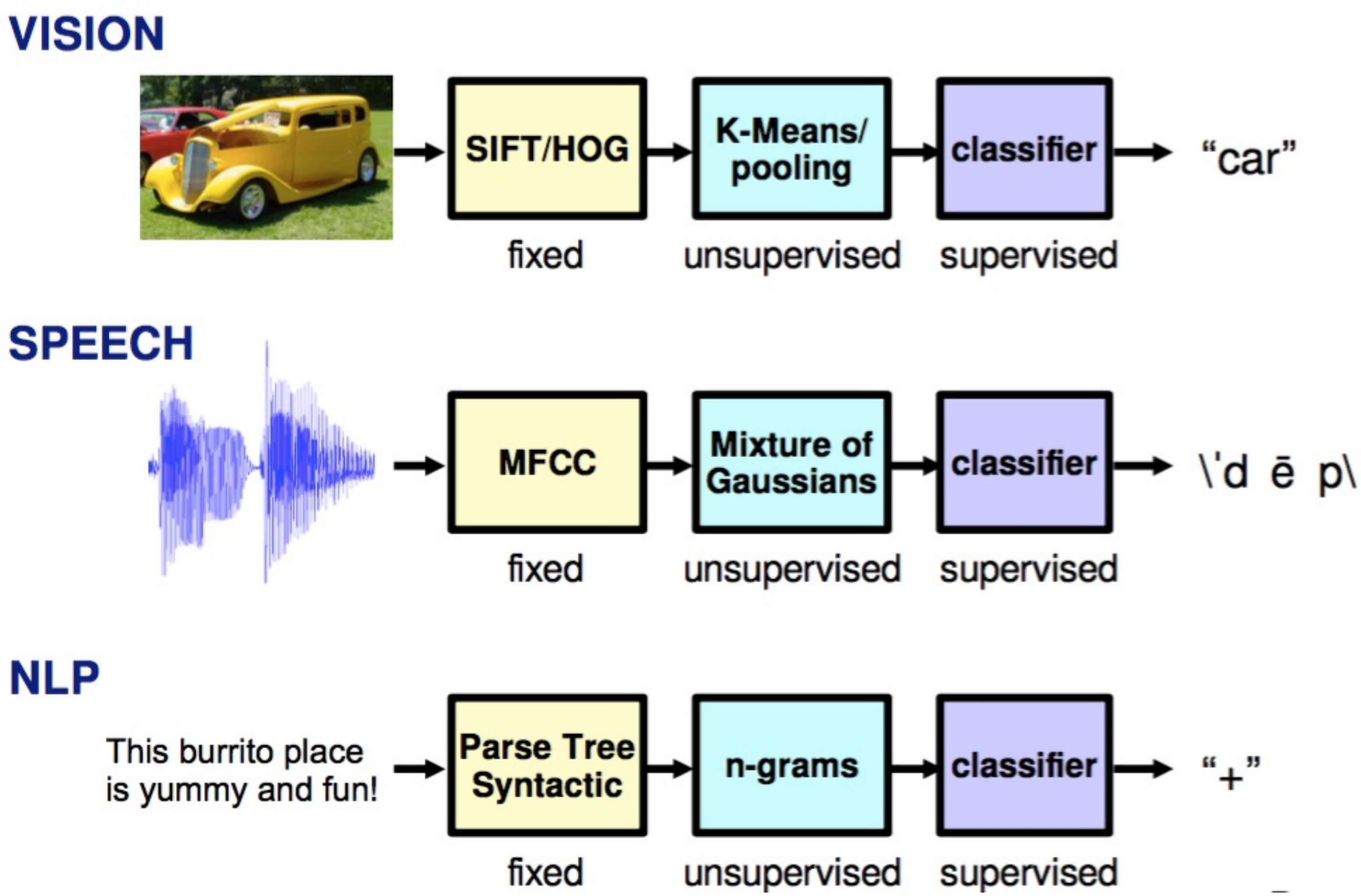
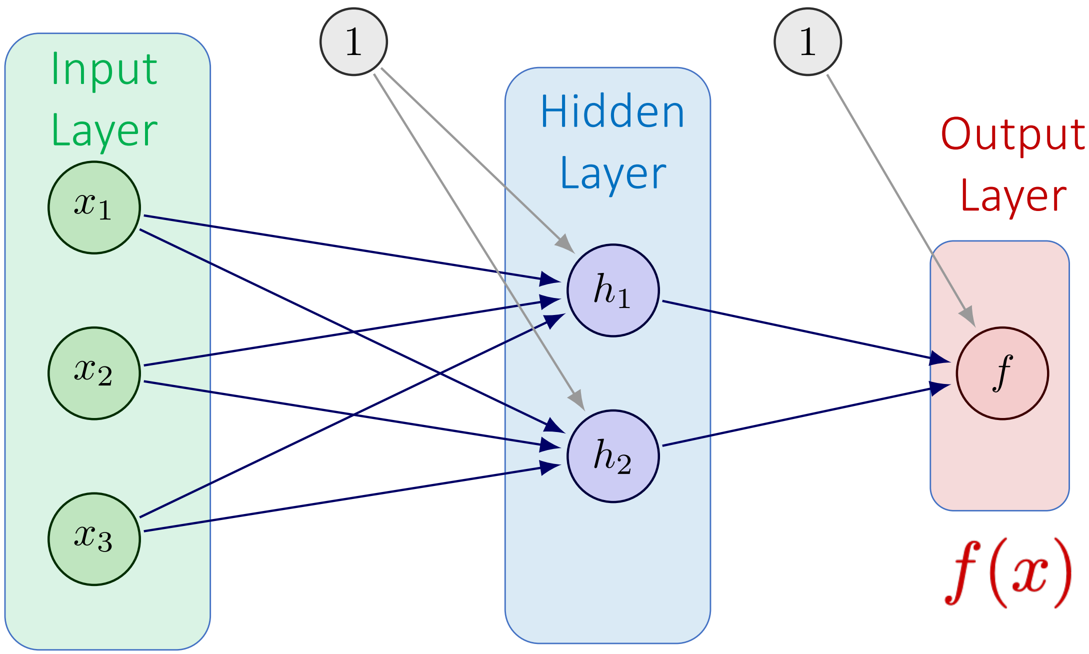


Figure from Marc'Aurelio-Ranzato

Activation functions

- Each hidden node applies a nonlinearity "a(r)"
  - May be the same or different per layer



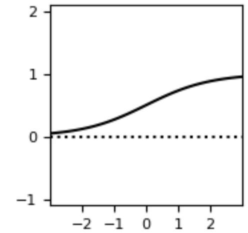
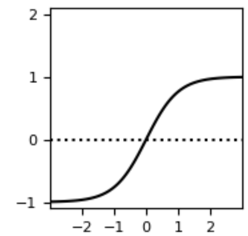
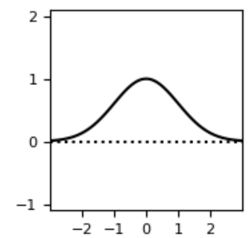
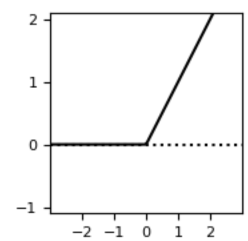
e.g., the logistic sigmoid:

$$a(r) = \sigma(r) = \frac{1}{1+\exp(-r)}$$

$$f(x) = a_2(\ \beta_0 + \beta_1 h_1 + \beta_2 h_2\ )$$

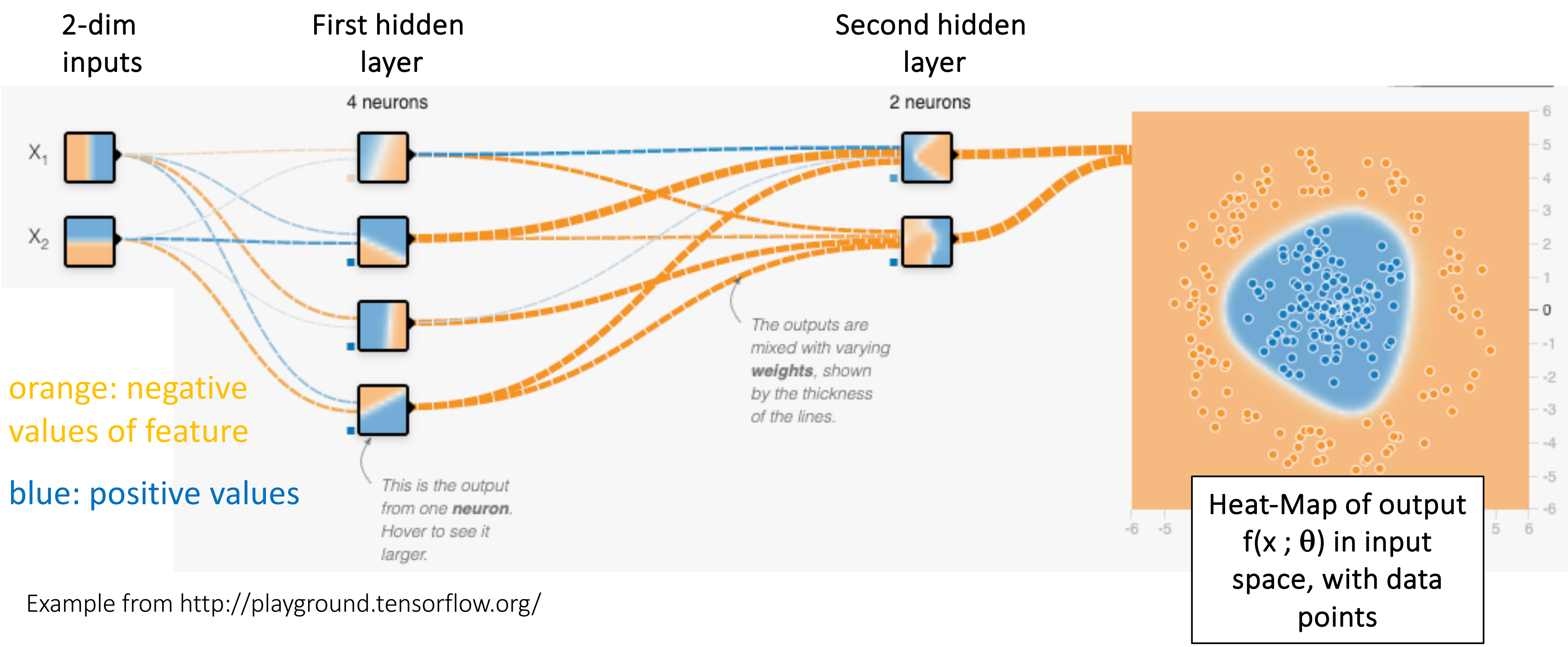
$$h_j(x) = a_1(\ w_{0j} + w_{1j}x_1 + w_{2j}x_2 + w_{3j}x_3\ )$$

Activation functions

Logistic	$a(r) = \frac{1}{1+\exp(-r)}$		$\frac{\partial a}{\partial r}(r) = a(r)(1-a(r))$
Hyperbolic Tangent	$a(r) = \frac{1-\exp(-2r)}{1+\exp(-2r)}$		$\frac{\partial a}{\partial r}(r) = 1-(a(r))^2$
Gaussian	$a(r) = \exp(-r^2/2)$		$\frac{\partial a}{\partial r}(r) = -r\,a(r)$
ReLU (rectified linear)	$a(r) = \max[0\,,\,r]$		$\frac{\partial a}{\partial r}(r) = \mathbb{1}[r>0]$
Linear	$a(r) = r$		and many others…

Example: Simple Network with 2 Hidden Layers

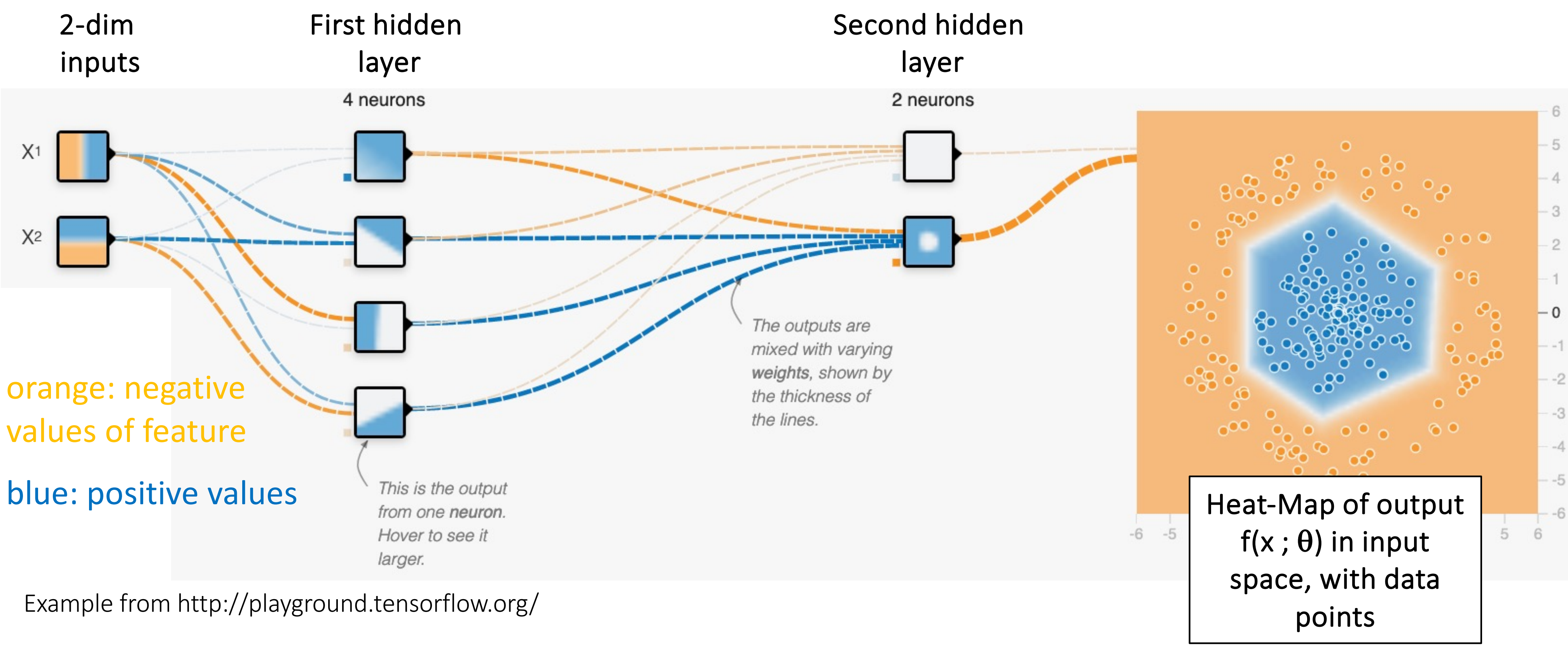
Two layers
- Layer 1: 4 hidden nodes
- Layer 2: 2 hidden nodes
- Activation: tanh



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

Example: Simple Network with 2 Hidden Layers

For comparison: same data, ReLU nonlinearity
- Now features are piecewise linear functions
- So, decision boundary is also piecewise linear
  - # pieces depends on # of layers and nodes…



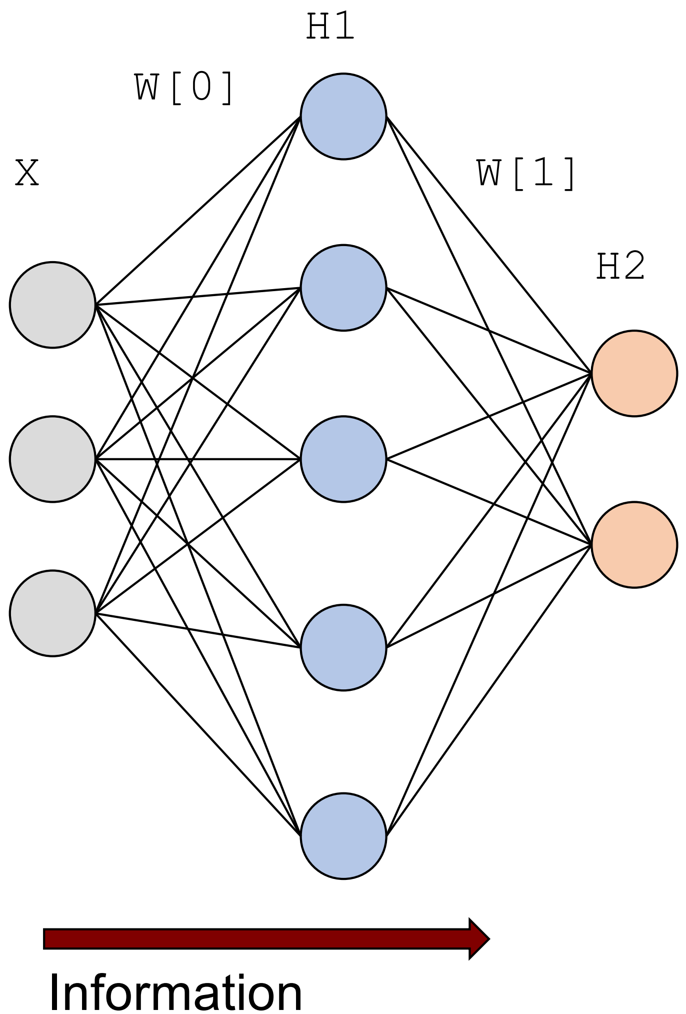
Example from http://playground.tensorflow.org/

Feed-forward networks

- Information flows left-to-right
  - Input observed features
  - Compute hidden nodes (parallel)
  - Compute next layer…

```
R = X @ W[0] + B[0]   # linear response
H1= Act( R )          # activation f'n

S = H1 @ W[1] + B[1] # linear response
H2 = Act( S )         # activation f'n
```



Neural Networks

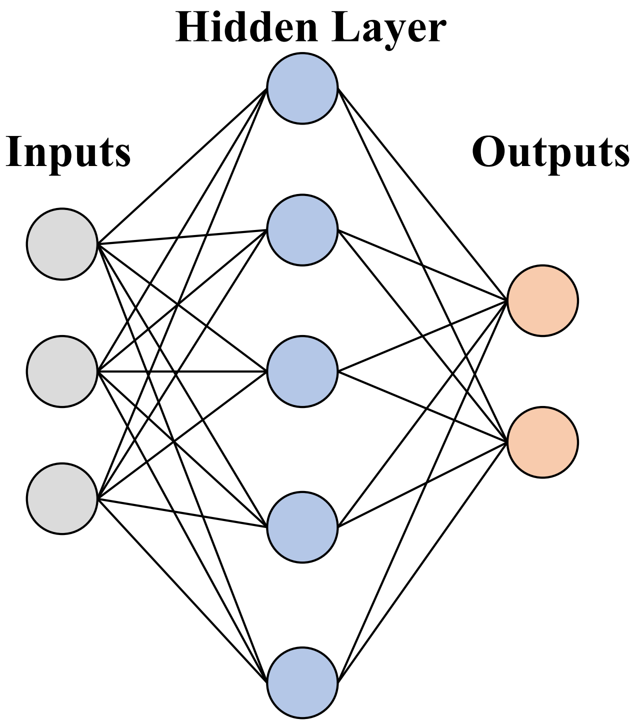
- Multi-Layer Perceptrons
- **Backpropagation Learning**
- Architectures
  - Convolutional
  - Residual
  - Attention
- Training Deep Networks
- More Tricks: Dropout, BatchNorm

Training MLPs

- Observe features “x” with target “y”
- Push “x” through NN = output is “f”
- Error: $(y – f)^2$ (Can use different loss functions if desired; e.g., log-loss/NLL)
- How should we update the weights to improve?

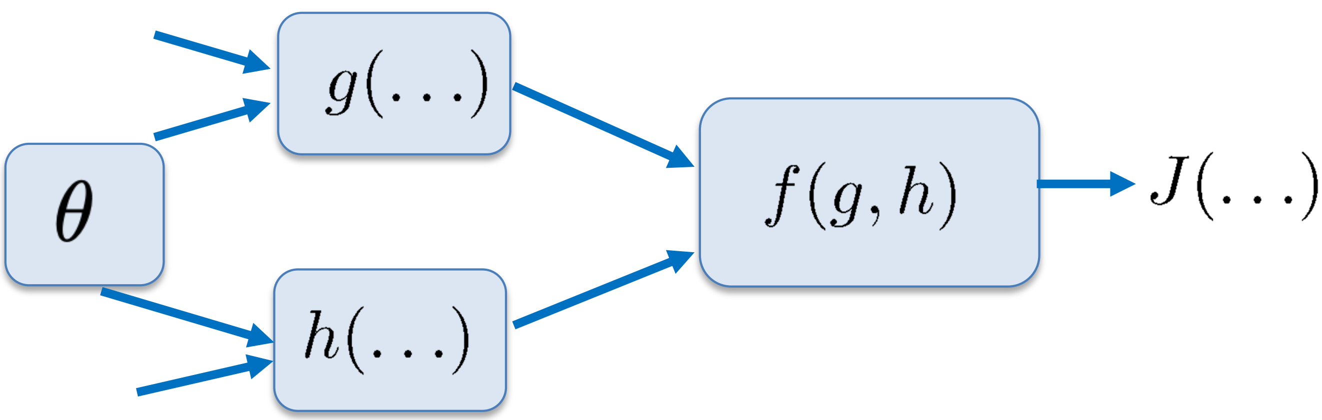
- Single layer
  - Logistic sigmoid function
  - Smooth, differentiable

- Optimize using:
  - Batch gradient descent
  - Stochastic gradient descent
  - What does the gradient look like?



Gradient: Forward Pass

- Think of NNs as "schematics" made of smaller functions
  - Building blocks: summations & nonlinear activations
  - For derivatives, just apply the chain rule!

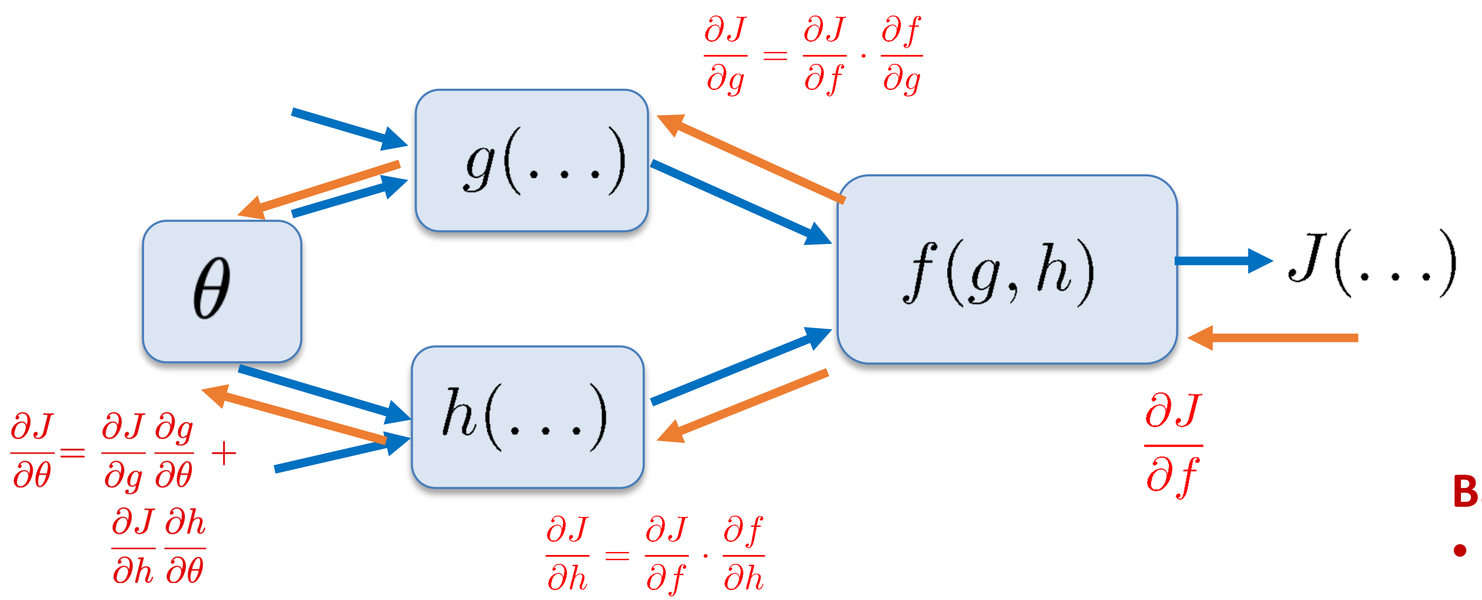


Forward pass:
Given initial value θ_0, x

$$g_0 = g(\theta_0, x)$$
$$h_0 = h(\theta_0, x)$$
$$f_0 = f(g_0, h_0)$$
$$J_0 = J(f_0, y)$$

Gradient: Backward Pass

- Think of NNs as "schematics" made of smaller functions
  - Building blocks: summations & nonlinear activations
  - For derivatives, just apply the chain rule!



Forward pass:
Given initial value θ_0, x

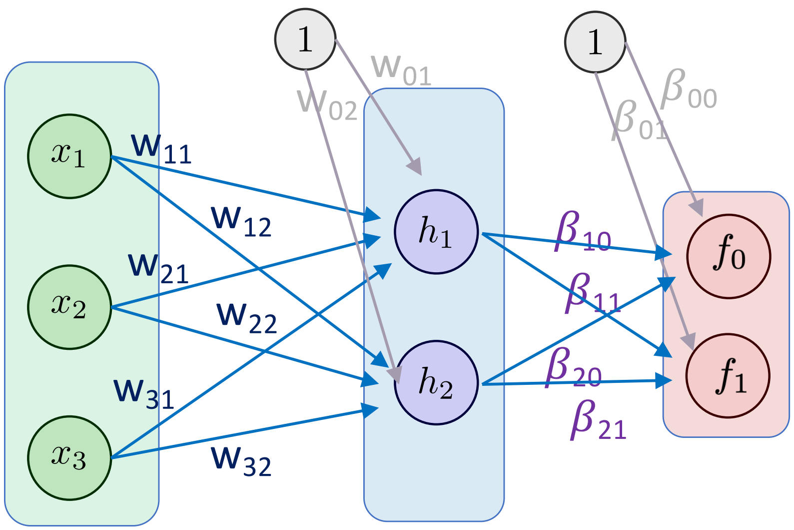
$$g_0 = g(\theta_0, x)$$
$$h_0 = h(\theta_0, x)$$
$$f_0 = f(g_0, h_0)$$
$$J_0 = J(f_0, y)$$

Backward pass:
- sum incoming derivative messages
- for each input arg:
  - multiply by slope
  - send back to arg

$$\left.\frac{\partial J}{\partial \theta}\right|_{\theta_0} = \left.\frac{\partial J}{\partial f}\right|_{f_0} \left(\left.\frac{\partial f}{\partial g}\right|_{g_0} \left.\frac{\partial g}{\partial \theta}\right|_{\theta_0} + \left.\frac{\partial f}{\partial h}\right|_{h_0} \left.\frac{\partial h}{\partial \theta}\right|_{\theta_0} \right)$$

Backpropagation

- Just gradient descent! Apply the chain rule to the MLP



Backward pass:

$$\frac{\partial J}{\partial \beta_{jk}} = \sum_{k'} \left(\frac{\partial J}{\partial f_{k'}} \right) \left(\frac{\partial f_{k'}}{\partial s_{k'}} \right) \left(\frac{\partial s_{k'}}{\partial \beta_{jk}} \right)$$

$$= \sum_{k'} \left(-2(y_{k'} - f_{k'}) \right) \left(\sigma'(s_k) \right) \left(\mathbb{1}[k' = k]\, h_j \right)$$

$$= \underbrace{-2\,(y_k - f_k)\sigma'(s_k)}_{\lambda_k} h_j$$

(Identical to logistic + mse loss classifier, with inputs "h_j")

Forward pass:

$$r_j = \sum_i w_{ij} x_i$$

$$h_j = a(r_j)$$

$$s_k = \sum_j \beta_{jk} h_j$$

$$f_k = \sigma(s_k)$$

$$J = \sum_k (y_k - f_k)^2$$

(or NLL loss, etc.)

$$\frac{\partial J}{\partial w_{ij}} = \sum_{k'} \left(\frac{\partial J}{\partial f_{k'}} \right) \left(\frac{\partial f_{k'}}{\partial s_{k'}} \right) \left(\frac{\partial s_{k'}}{\partial w_{ij}} \right)$$

$$\sum_{j'} \left(\frac{\partial s_{k'}}{\partial h_{j'}} \right) \left(\frac{\partial h_{j'}}{\partial r_{j'}} \right) \left(\frac{\partial r_{j'}}{\partial w_{ij}} \right)$$

$$\sum_{j'} \left(\beta_{j'k'} \right) \left(\sigma'(r_{j'}) \right) \left(\mathbb{1}[j' = j]\, x_i \right) \Big)$$

$$= \sum_{k'} \underbrace{\left(-2(y_{k'} - f_{k'}) \right) \left(\sigma'(s_{k'}) \right)}_{\lambda_k} \left(\beta_{jk'} \right) \left(\sigma'(r_j) \right) \left(x_i \right)$$

Backpropagation (AutoGrad Version)

```
# Define torch.tensor arrays
#  for any trainable parameters:
W = tensor(..., requires_grad=True)
B = tensor(..., requires_grad=True)
```

“requires grad” tells torch to track these parameters through subsequent computations.

```
# Define optimizer over params:
opt = torch.optim.SGD([W,B], lr=...)

for each mini-batch:
  opt.zero_grad() # Reset gradient

  # Apply forward computations:
  H = act( X @ W[:,1:].T + W[:,0] )
  F = act( H @ B[:,1:].T + B[:,0] )

  J = Loss(Y,F) # Compute your loss
  J.backward()  # Backprop gradient
  opt.step()    # Update W,B
```

SGD or other optimization algo: provide the parameters we plan to update with this process

”zero_grad” resets gradient storage before forward computation

Then, the parameters are used in a sequence of forward computations…

…which are then used in computing the (differentiable) loss.

J (and F,H) are grad-enabled tensors; “backward” backpropagates to accumulate into the gradient storage

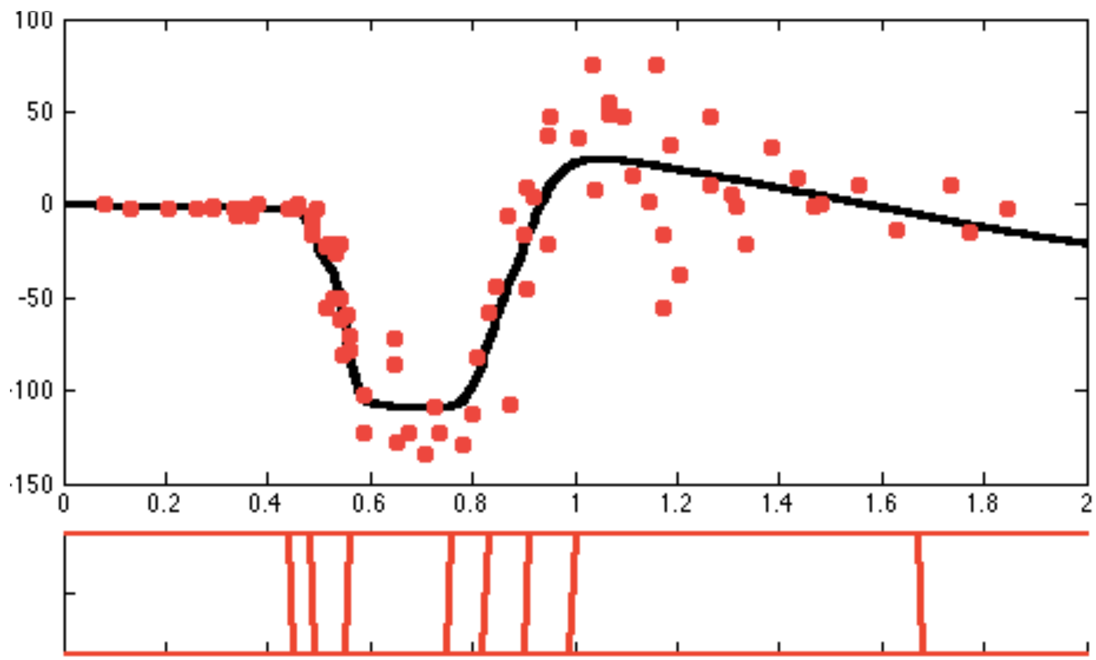
“step” updates the parameters associated with this optimizer

Example: Regression, MCycle data

- Train NN model, 2 layer
  - 1 input features => 1 input units
  - 10 hidden units
  - 1 target => 1 output units
  - Logistic sigmoid activation for hidden layer, linear for output layer

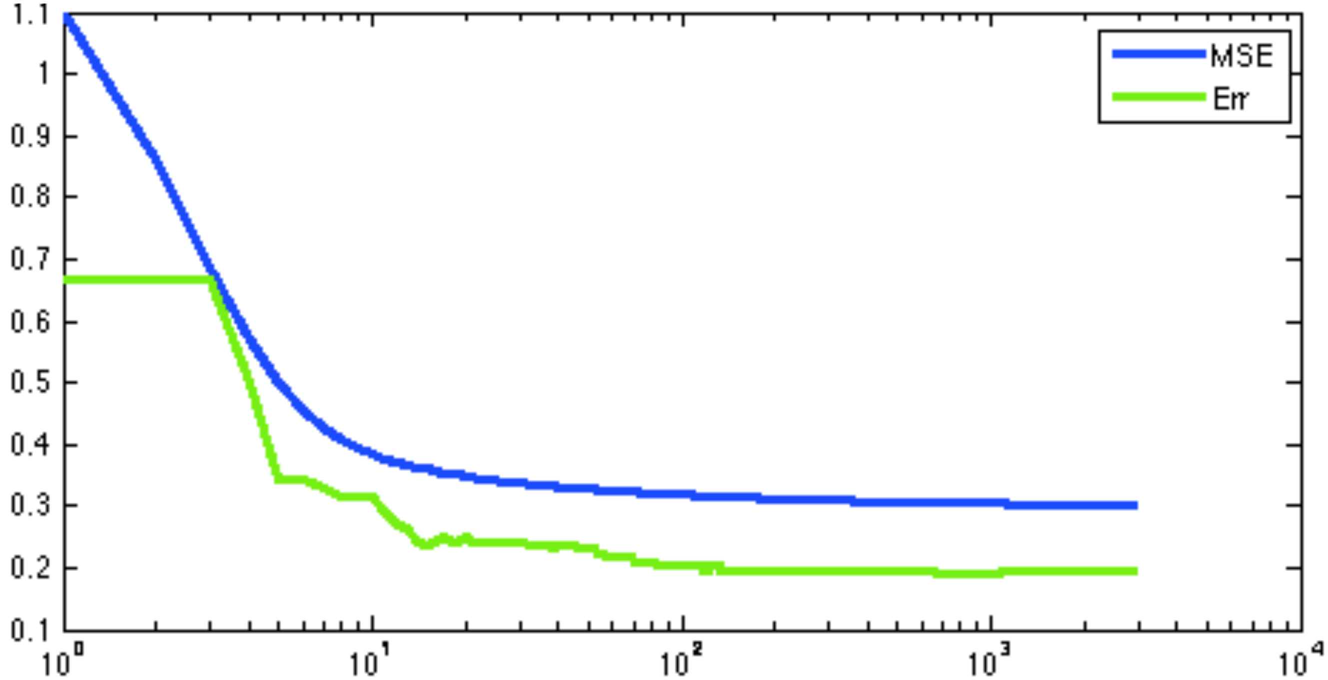
Data:
+
learned prediction f'n:

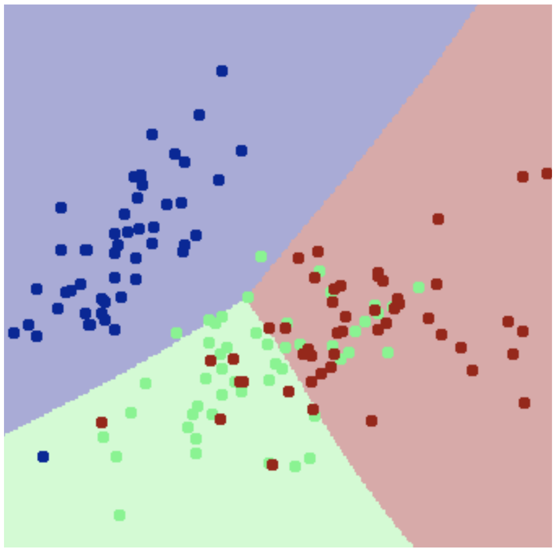
Responses of hidden nodes
(= features of linear regression):
select out useful regions of "x"



Example: Classification, Iris data

- Train NN model, 2 layer
  - 2 input features => 2 input units
  - 10 hidden units
  - 3 classes => 3 output units   (y = [0 0 1], etc.)
  - Logistic sigmoid activation functions
  - Optimize MSE of predictions using stochastic gradient





Demo Time!

http://playground.tensorflow.org/

Neural Networks

Multi-Layer Perceptrons

Backpropagation Learning

Architectures

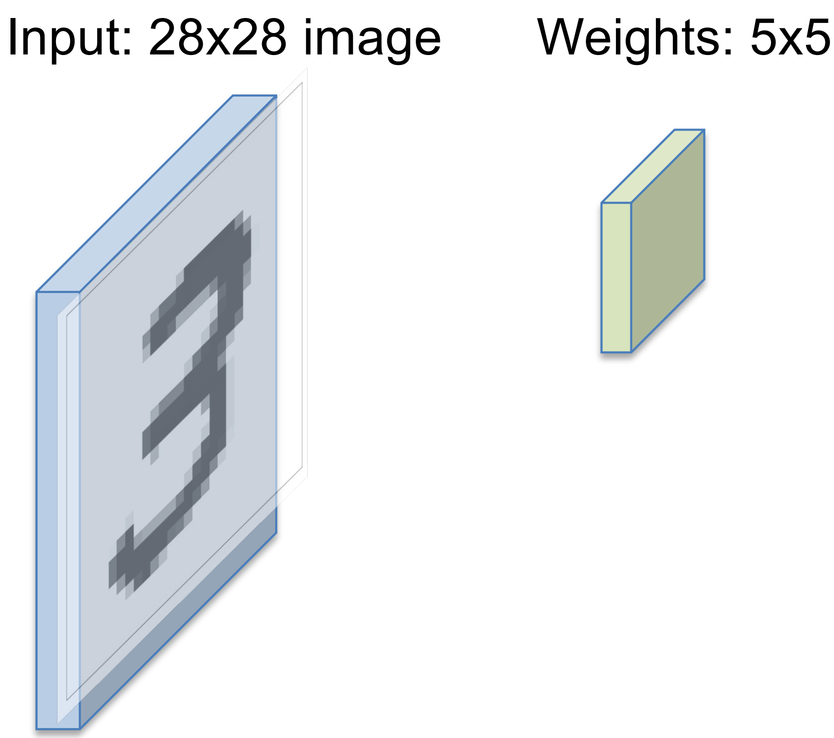
Convolutional | Residual | Attention

Training Deep Networks

More Tricks: Dropout, BatchNorm

Convolutional networks

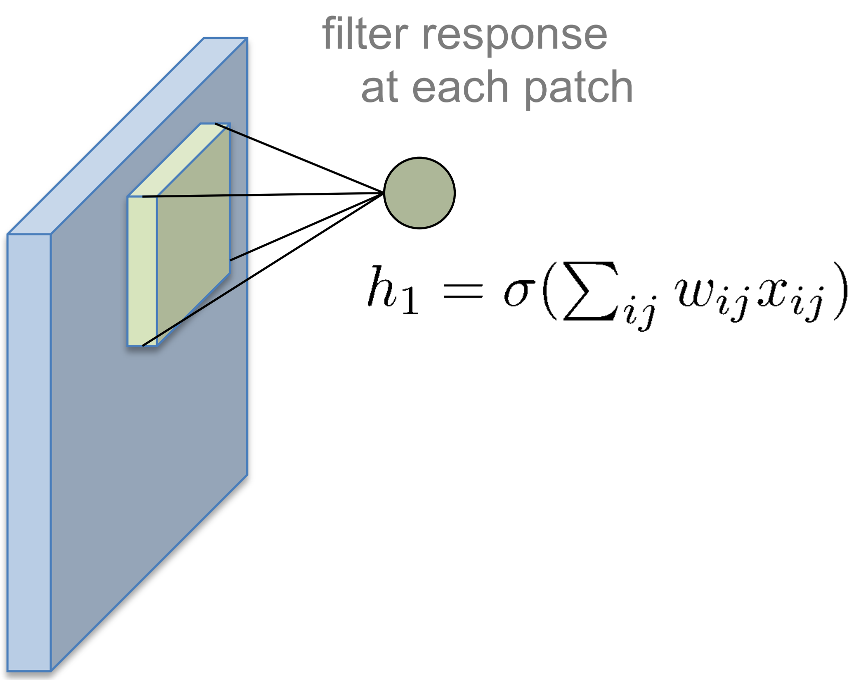
- Organize & share the NN's weights   (vs "dense")
- Group weights into "filters"

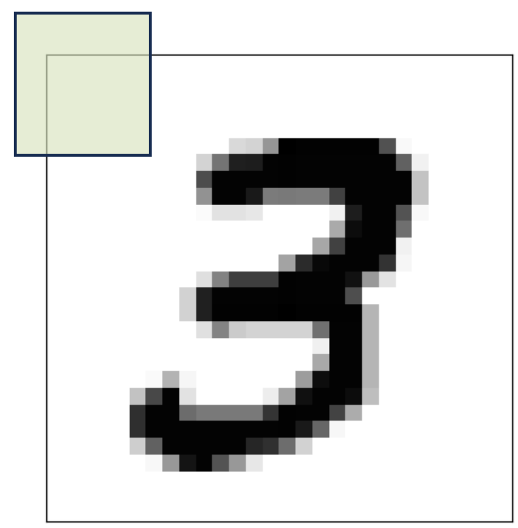


Convolutional networks

- Organize & share the NN's weights (vs "dense")
- Group weights into "filters" & convolve across input image

Input: 28x28 image   Weights: 5x5



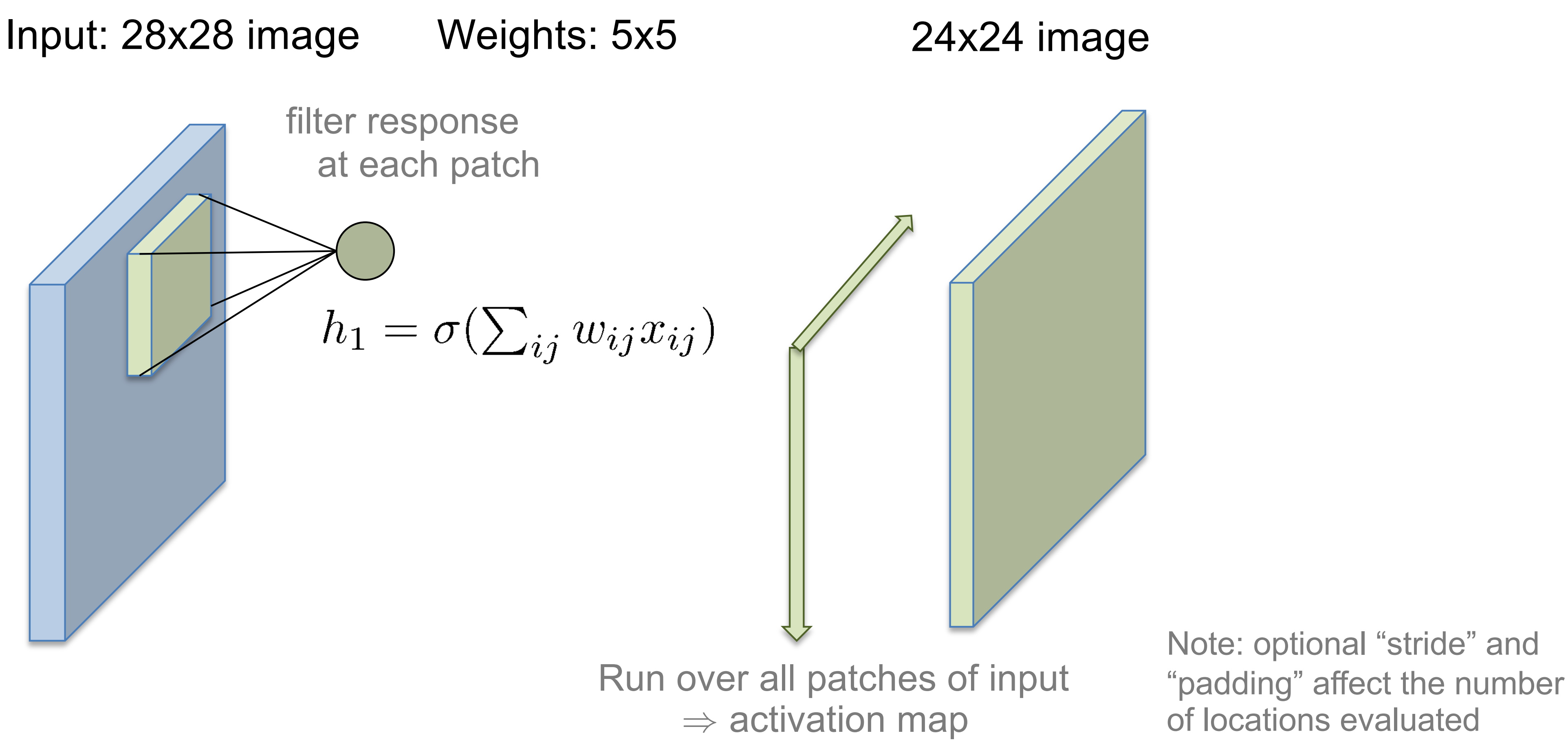


Run over all patches of input
$\Rightarrow$ activation map

Note: optional "stride" and "padding" affect the number of locations evaluated

Convolutional networks

- Organize & share the NN's weights (vs "dense")
- Group weights into "filters" & convolve across input image

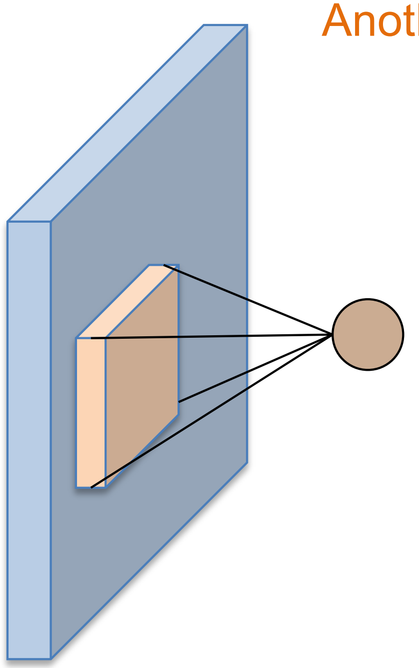


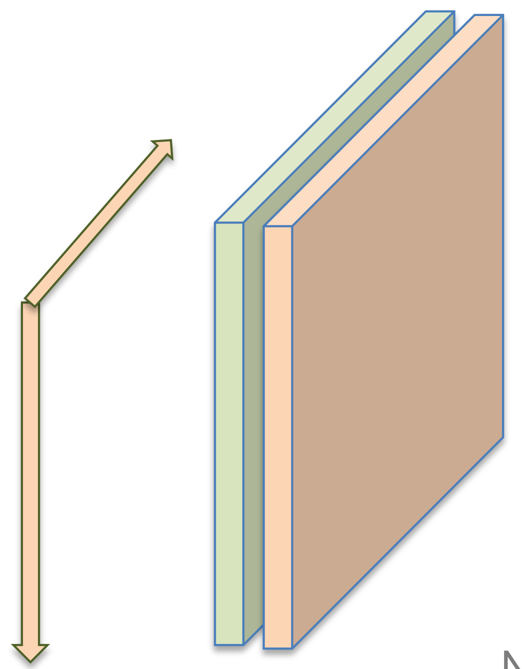
Convolutional networks

- Organize & share the NN's weights   (vs "dense")
- Group weights into "filters" & convolve across input image

Input: 28x28 image   Weights: 5x5

Another filter



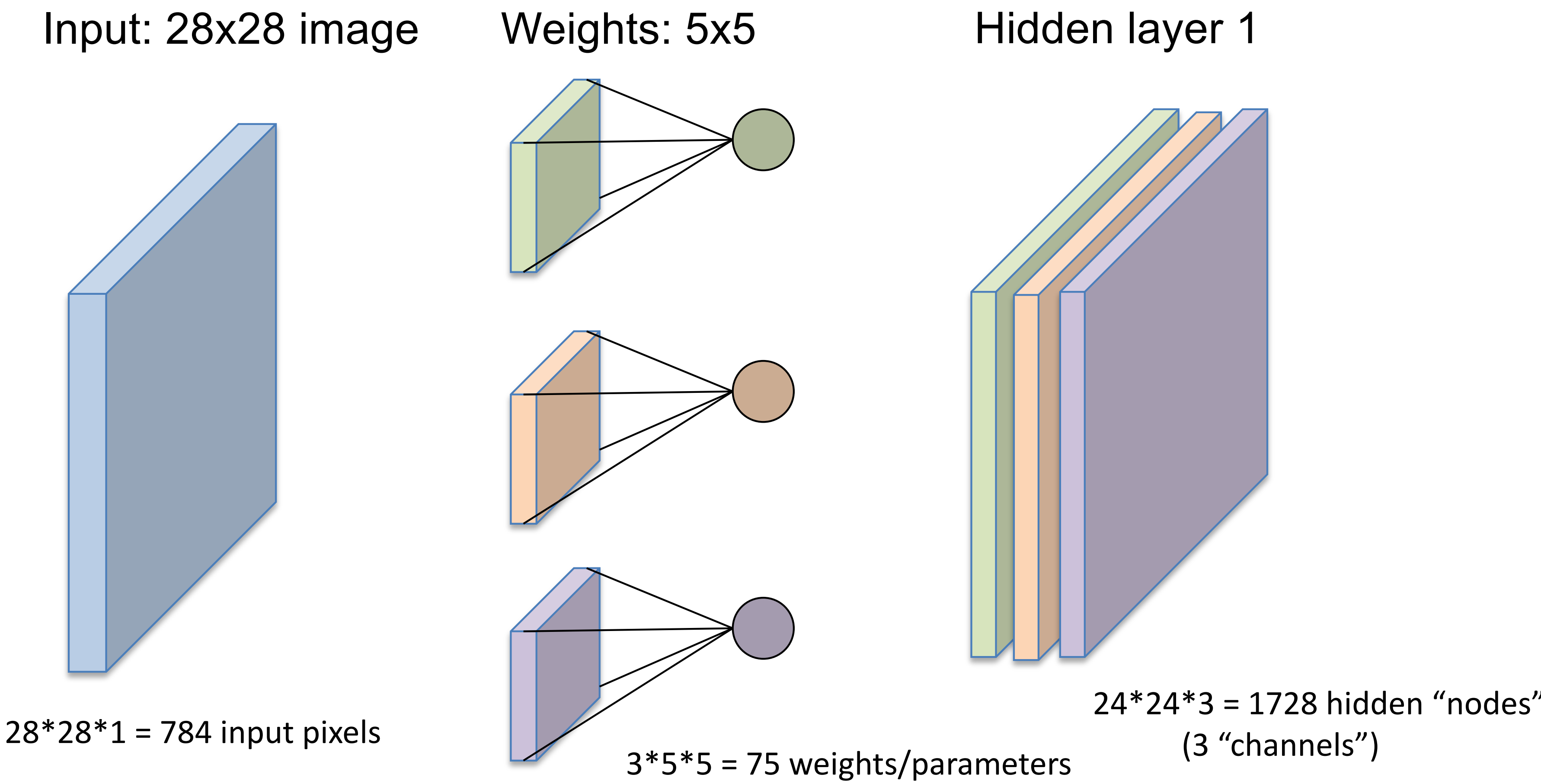


Run over all patches of input
⇒ activation map

Note: optional "stride" and "padding" affect the number of locations evaluated

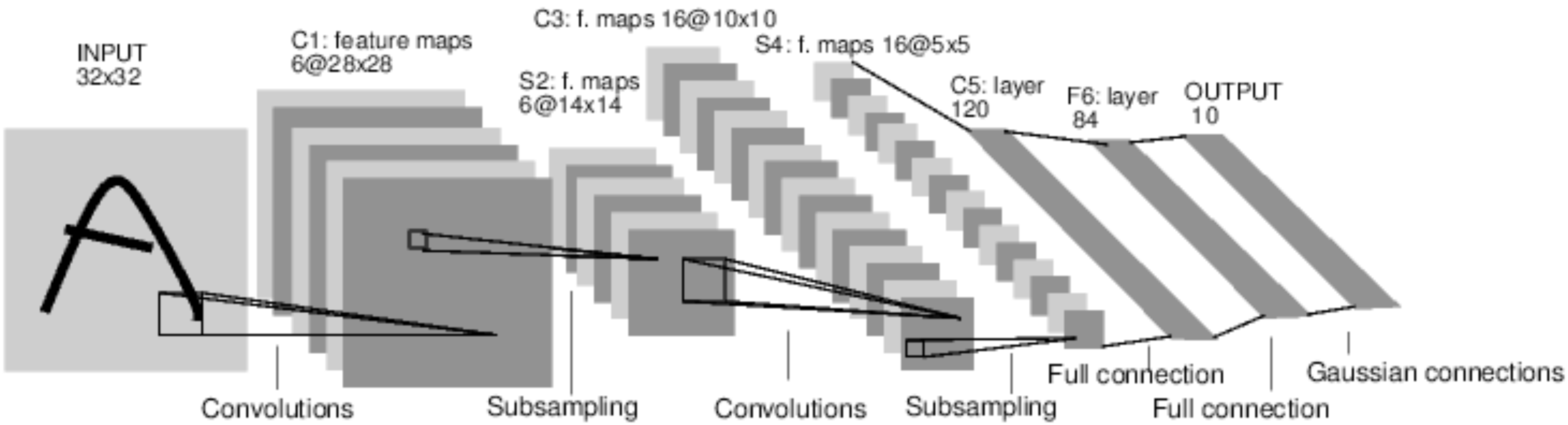
Convolutional networks

- Organize & share the NN's weights (vs "dense")
- Group weights into "filters" & convolve across input image
- Many hidden nodes, but few parameters!



Convolutional networks

- Again, can view components as building blocks
- Design overall, deep structure from parts
  - Convolutional layers
  - “Max-pooling” (sub-sampling) layers
  - Densely connected layers



LeNet-5 [LeCun 1980]

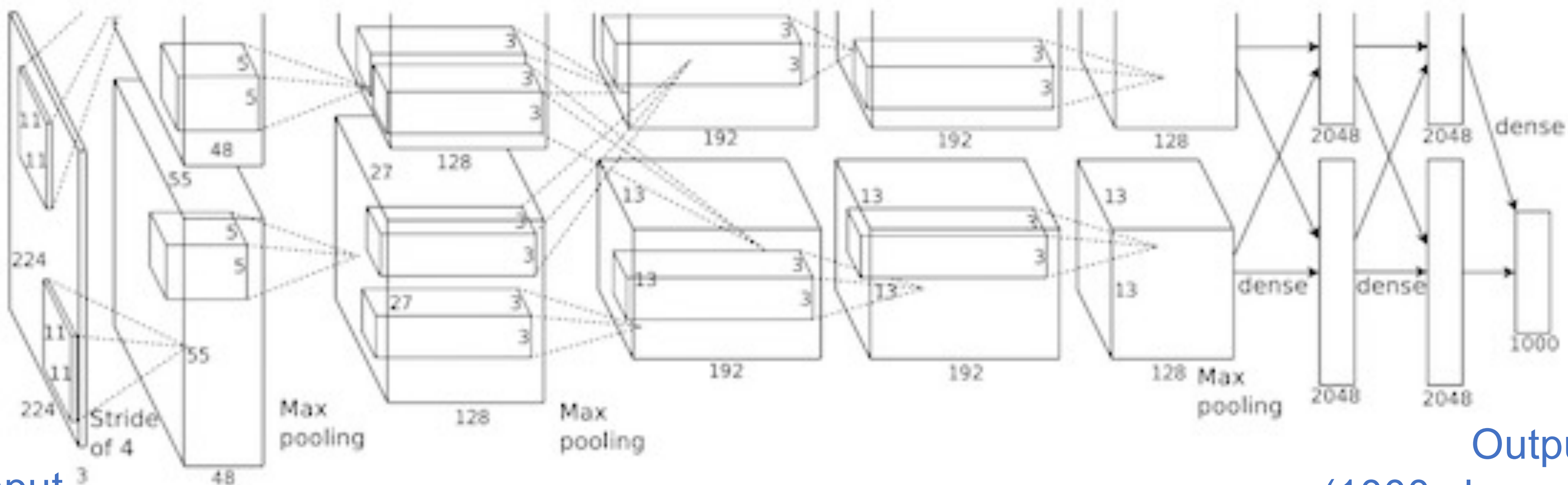
[Krizhevsky et al. 2012]

Ex: AlexNet

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

Convolutional Layers (5)

Dense Layers (3)

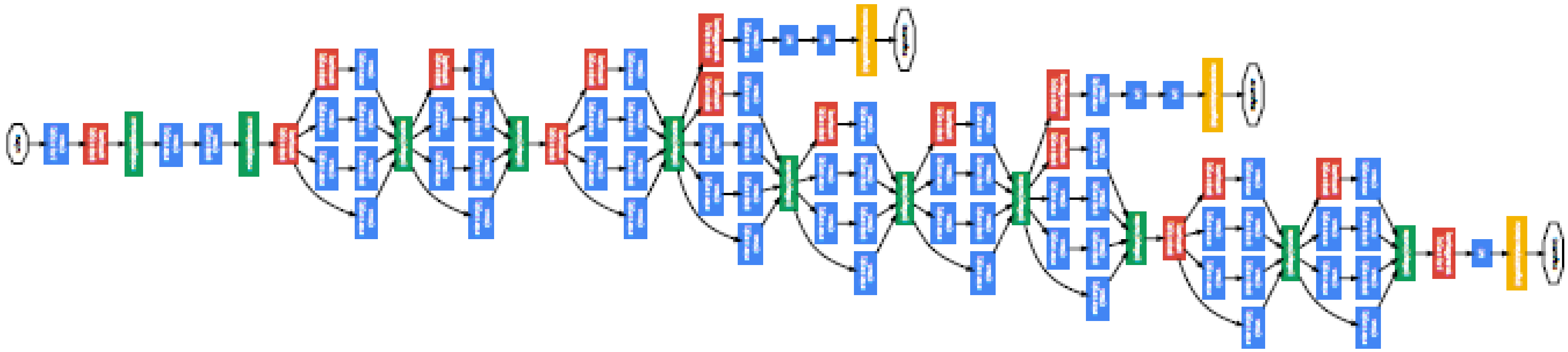


Input
224x224x3

Output
(1000 classes)

Ex: GoogLeNet

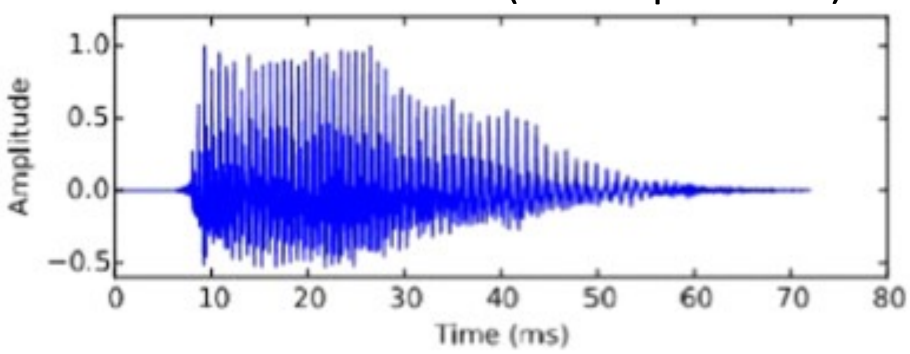
- Image recognition model
  - 27 layers, millions of parameters



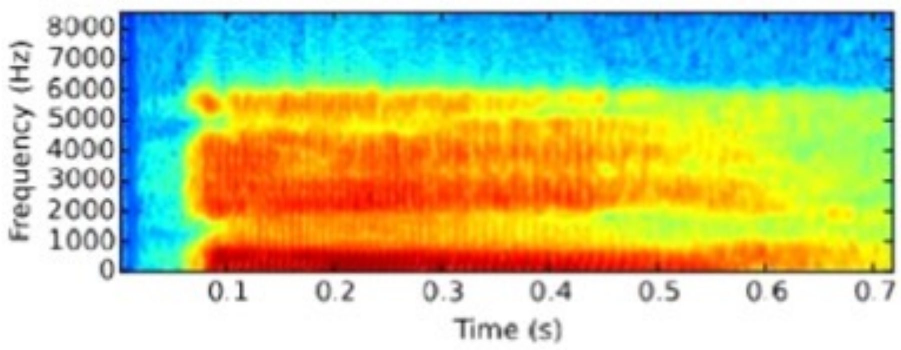
Not just for images…

- Ex: time series (speech, etc)
  - May pre-transform the data
    - Fourier: vector of frequency intensities at each time point
  - Then, convolve over time dimension

“Time domain” (scalar per time)



“Frequency domain” (vector per time)



Convolutional Layers (Torch)

```
# Define Layers
from torch.nn import *

# in & out channels, filter size, etc.
conv = Conv2d(1, 16, (5,5), stride=2)
# pool size, etc.
pool = MaxPool2d(3, stride=2)
# "normal" (fully connected) layers
linear = Linear(400, num_classes)
```

Torch layers contain trainable parameters (grad-enabled tensors)

Be careful declaring sizes, as these need to match correctly or you'll get mismatching shape errors

```
# Forward pass: apply each layer
r1  = conv(X)
h1a = relu(r1)
h1b = pool(h1a)
h1c = Flatten()(h1b)
r2  = linear(h1c)
f   = softmax(r2,axis=1)
```

"Applying" the layers computes their forward pass

Then, use "f" when calculating a differentiable loss and call "backward()" to compute the gradients

Neural Networks

Multi-Layer Perceptrons

Backpropagation Learning

Architectures

Convolutional | Residual | Attention

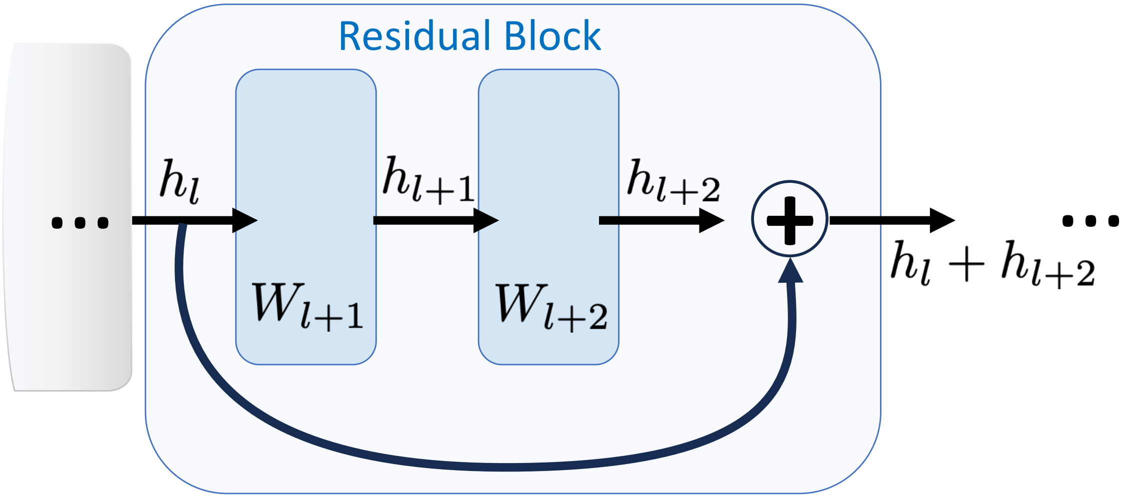
Training Deep Networks

More Tricks: Dropout, BatchNorm

Residual Networks

- In practice, deep networks can work worse than shallow ones!
  - Simple / shallow transforms are easier to find and may already work well
  - Transforming more not helpful – have to “find” identity transform?

- Residual Block
  - Use layers to estimate ”update” to features, rather than transform
  - If the block’s weights W are near zero, just keeps input representation
  - Helps with “vanishing gradient” problem: h_l has direct effect on output



Residual blocks:
- allow “deep” networks
- early layers train better
- unhelpful layers can learn to “skip” (zero value)
- Can use with any layer type (convolutional, dense, etc.)

Neural Networks

Multi-Layer Perceptrons

Backpropagation Learning

Architectures

Convolutional | Residual | Attention

Training Deep Networks

More Tricks: Dropout, BatchNorm

Attention & Transformers

- Alternative structure, made popular by natural language
  - ChatGPT, LLaMa, etc.

- Ex: Translation

English I will find you a red pen

French Je vais te trouver un stylo rouge

Given inputs & “current context”, predict the next output token

I will find you a red pen

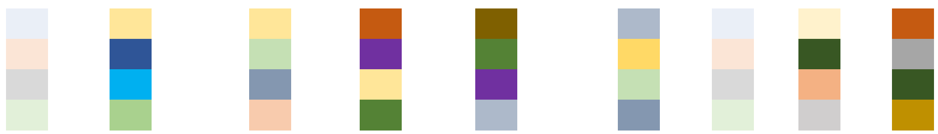
“Query”

Je vais te trouver un stylo …

Attention & Transformers

- Given a sequence (or other collection) of tokens (each a vector), process them based on their relevance to a “query” vector

The quick brown fox jumped over the lazy dog.



x_1 x_2 $\ldots$

This is our initial representation; for text data, an “embedding”

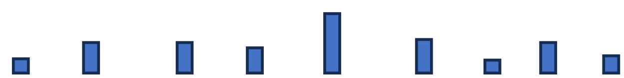
Query:

Sentence action?

$q =$

Which words are relevant? “Keys” $k_j = K \odot x_j$

The quick brown fox jumped over the lazy dog.



$$\alpha = \mathrm{softmax}(\ [k_1 \odot q, \ldots, k_n \odot q]\ )$$

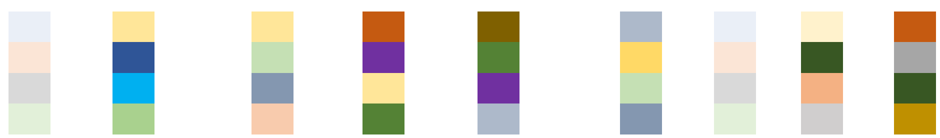
What should I take from those words? “Value” $v_j = V \odot x_j$

Query-Key-Value result: $h = \sum_j \alpha_j \, v_j$

Attention & Transformers

- Representation is permutation-invariant

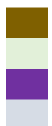
The quick brown fox jumped over the lazy dog.



x_1 x_2 $\ldots$

The fox, being quick, jumped over the dog.

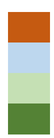
x_1 x_2 $\ldots$

Query: Sentence action? $q =$ 

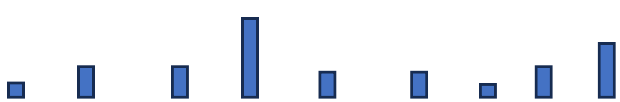
The quick brown fox jumped over the lazy dog.

The fox, being quick, jumped over the dog.

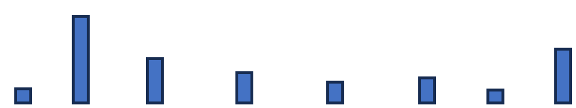
- Applying a different query extracts different information

Query: Sentence subject? $q =$ 

The quick brown fox jumped over the lazy dog.

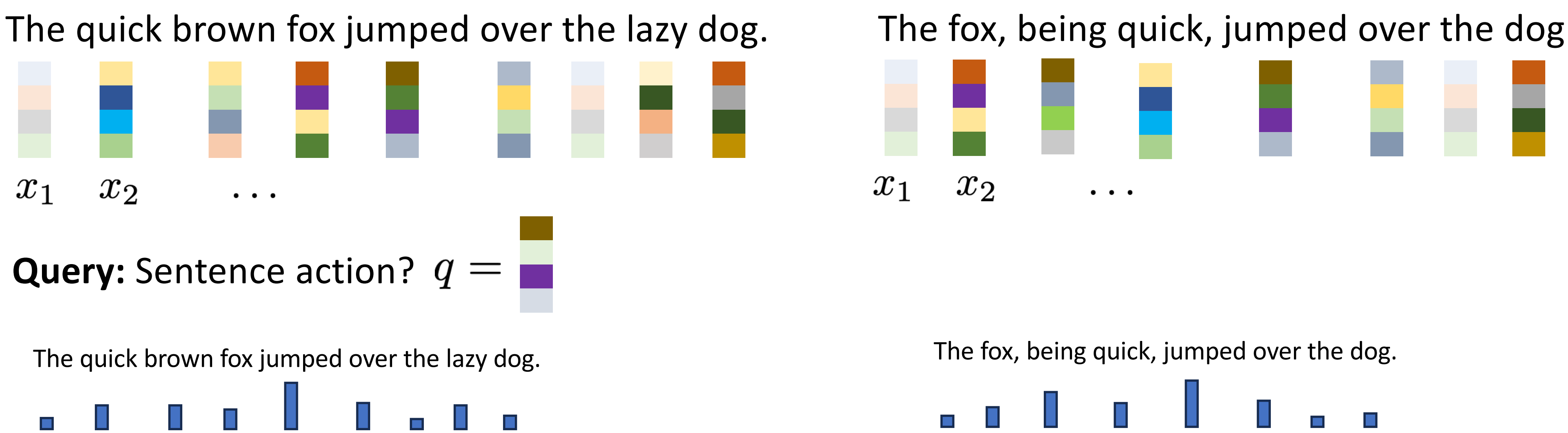


The fox, being quick, jumped over the dog.



Attention & Transformers

- Representation is permutation-invariant



- Applying a different query extracts different information
- What if we don't want invariance? (Position is meaningful!)
  - Can use "positional encoding"
  - Make position part of the input xi, or the query & key transforms

Attention Block

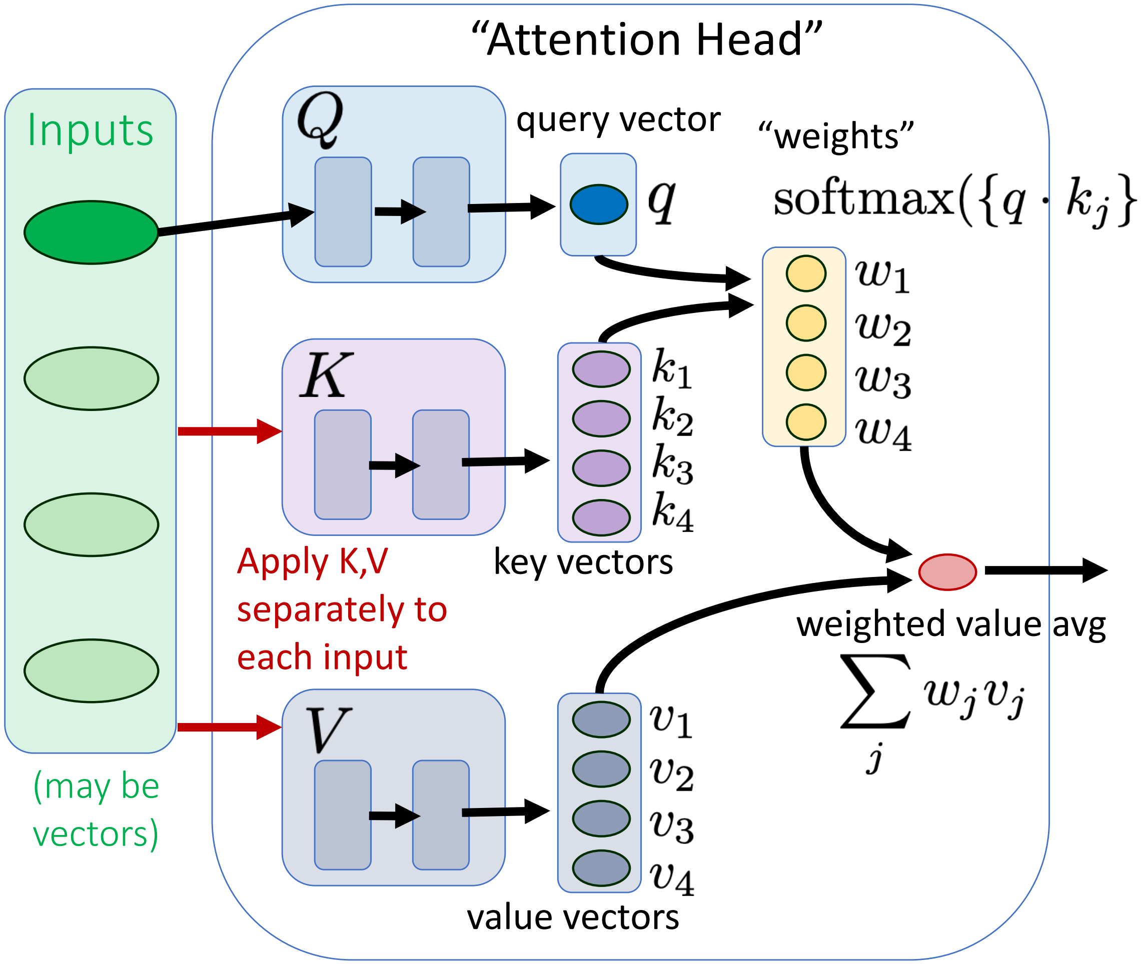
- Extract representation from many (mostly irrelevant?) inputs
  - Ex: predict next word from past sequence of observed words

Inputs: a collection of (possibly vector-valued) measurements

Query: context for current prediction

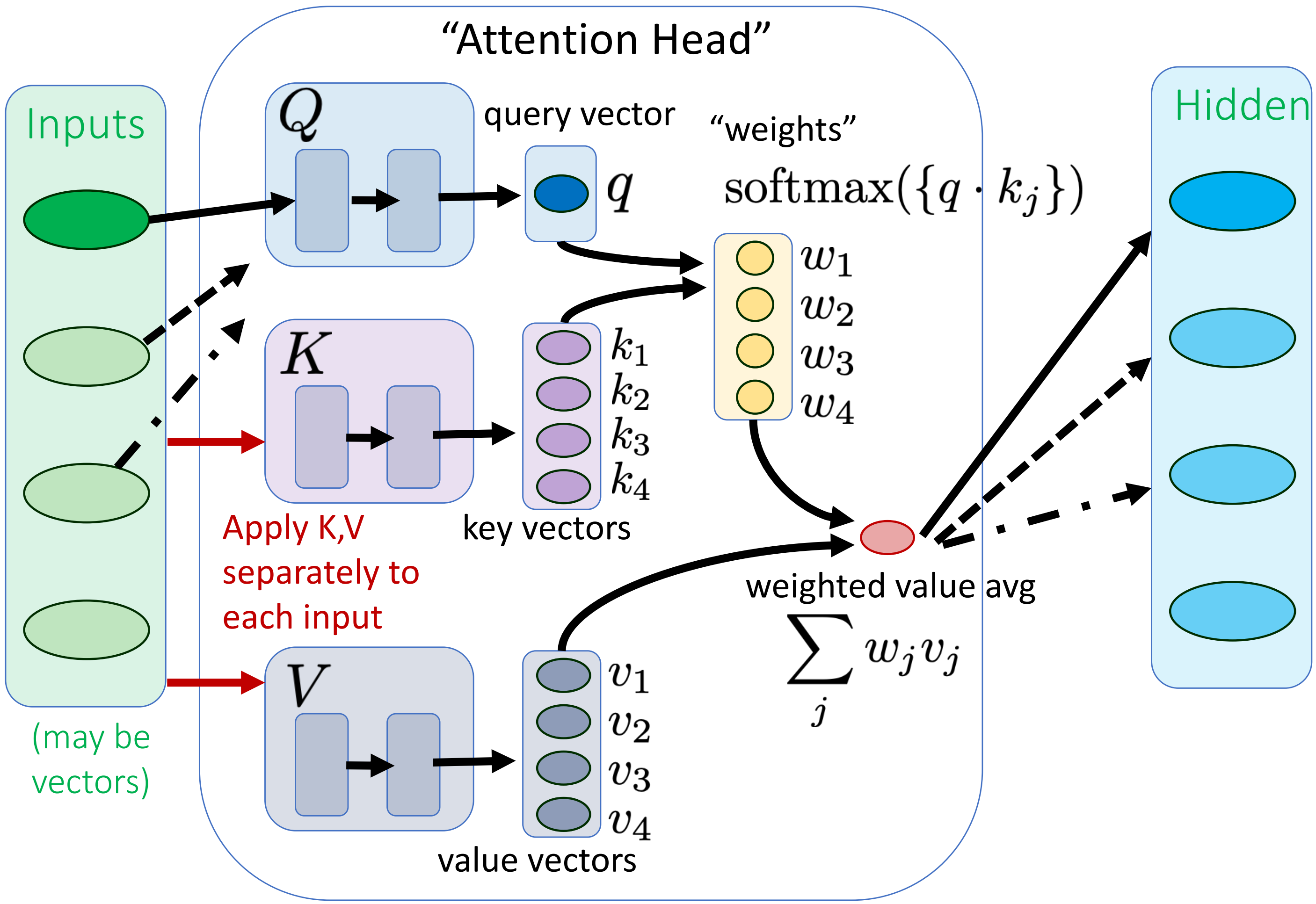
Keys: per-input vectors computed from inputs. Similarity to query determines per-input weights.

Values: per-input vectors from inputs used to compute output.



Self-Attention

- Apply Query-Key-Value computation to each input
  - One output per input: dimension-preserving transformation of the input



Neural Networks

Multi-Layer Perceptrons

Backpropagation Learning

Architectures

Convolutional

Residual

Attention

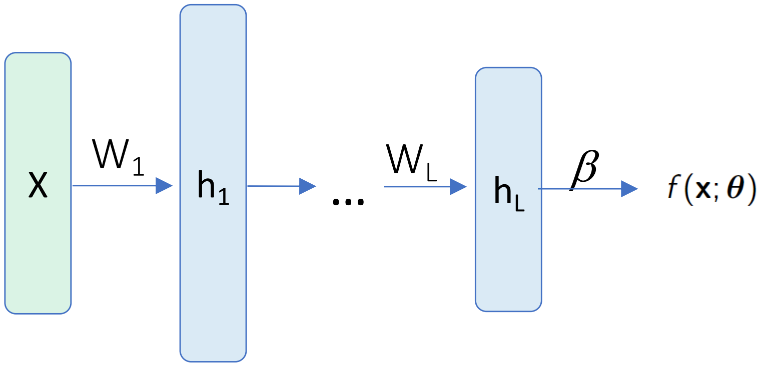
Training Deep Networks

More Tricks: Dropout, BatchNorm

Size of a deep network

- Model size grows quickly for deep networks:

Say the network has:
n-dimensional feature inputs
L layers of hidden units
K classes



Assume for simplicity that each hidden layer has H hidden units & ignore bias terms

Number of parameters p is roughly: $n\,H + (L\text{-}1)H^2 + H\,K$

e.g., d = 100 x 100 = 10^4 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

This means that a single epoch can be extremely slow!
So, stochastic gradient is the preferred optimization technique.

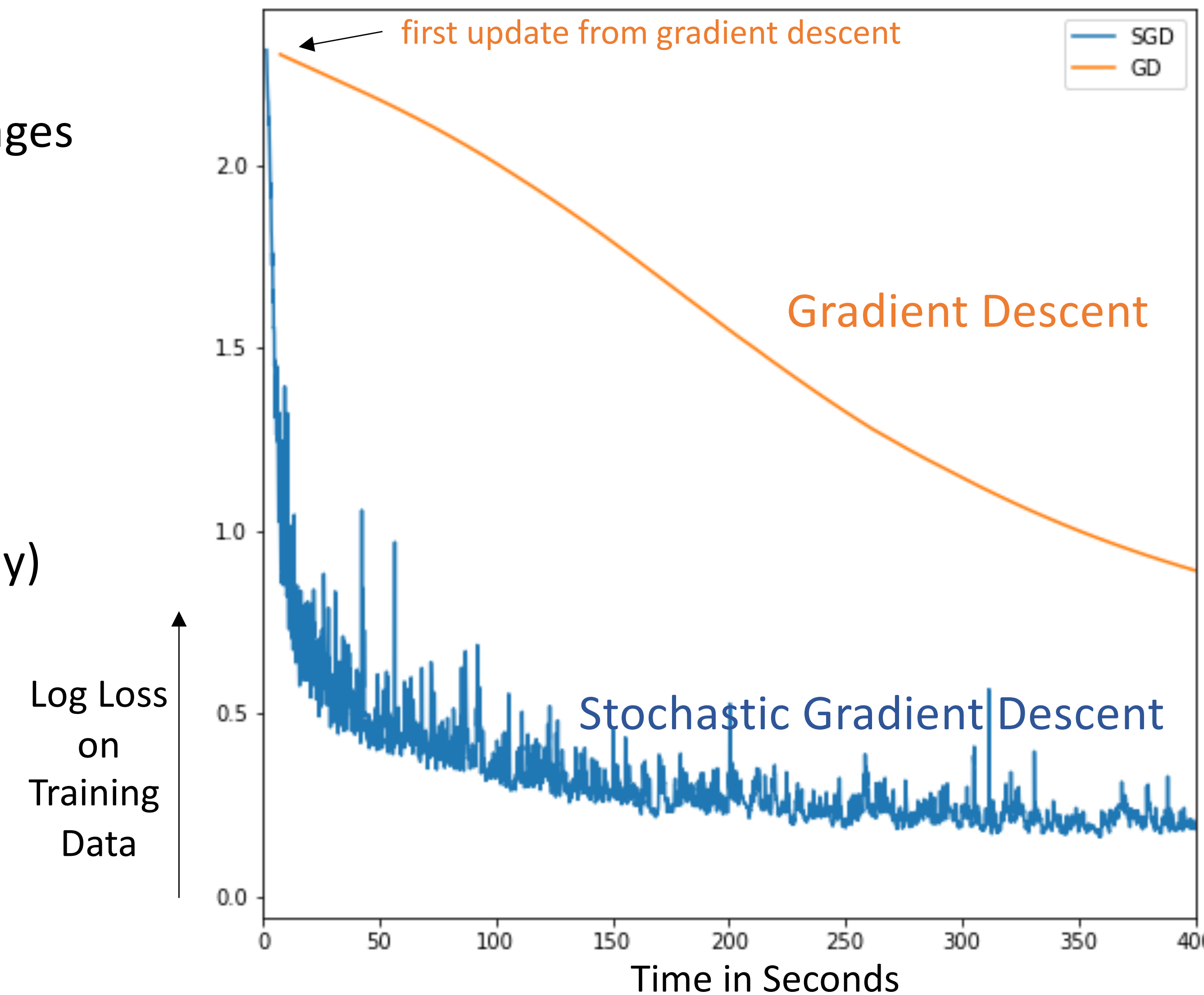
Ex: Stochastic Gradient

- Comparing SGD & GD on MNIST data

Minibatch size: b = 4
Data: m = 60k training images

Neural network with
n = 784 inputs
H = 256 hidden units
K = 10 classes

p = 203,000 (approximately)



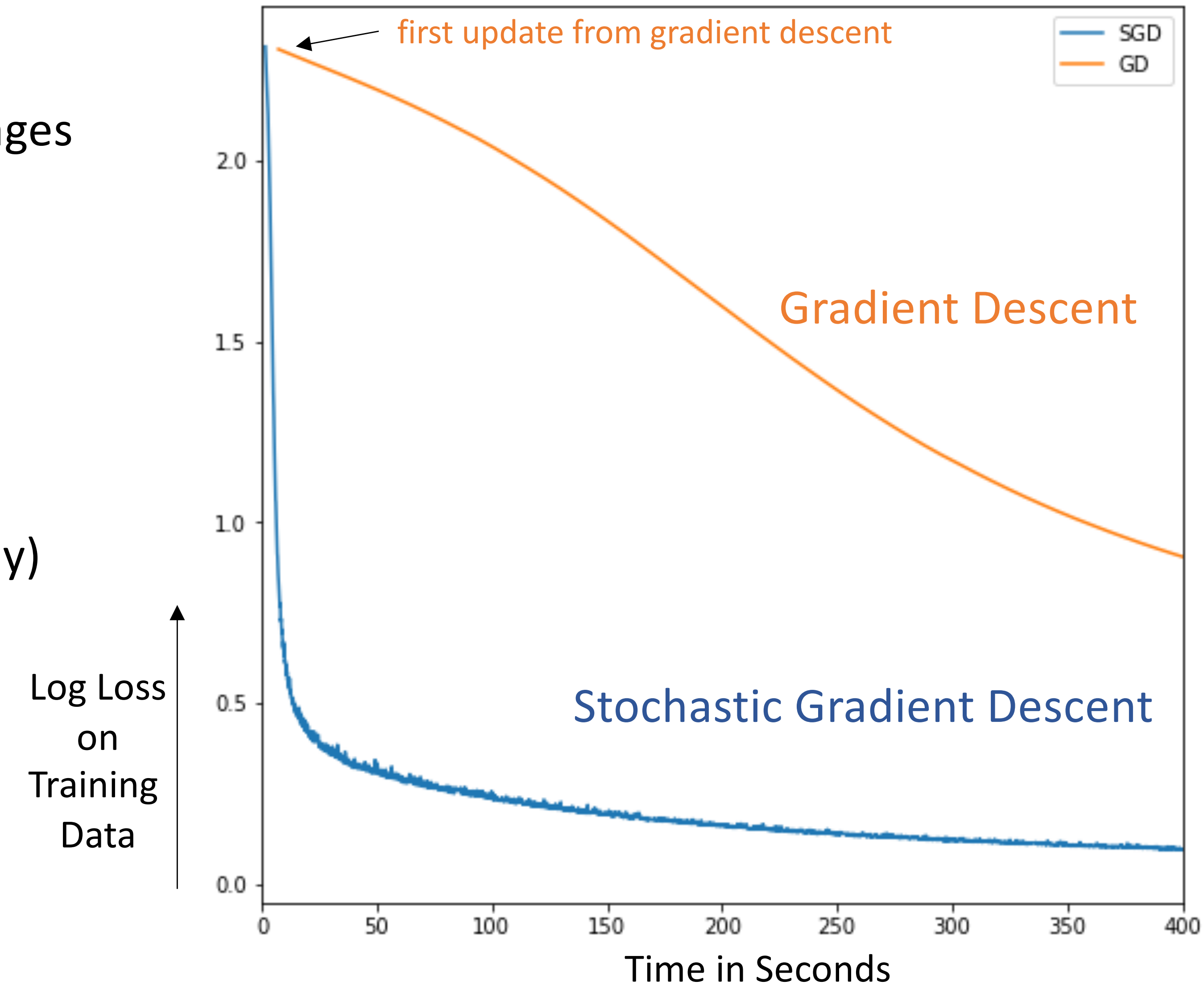
Ex: Stochastic Gradient

- Comparing SGD & GD on MNIST data

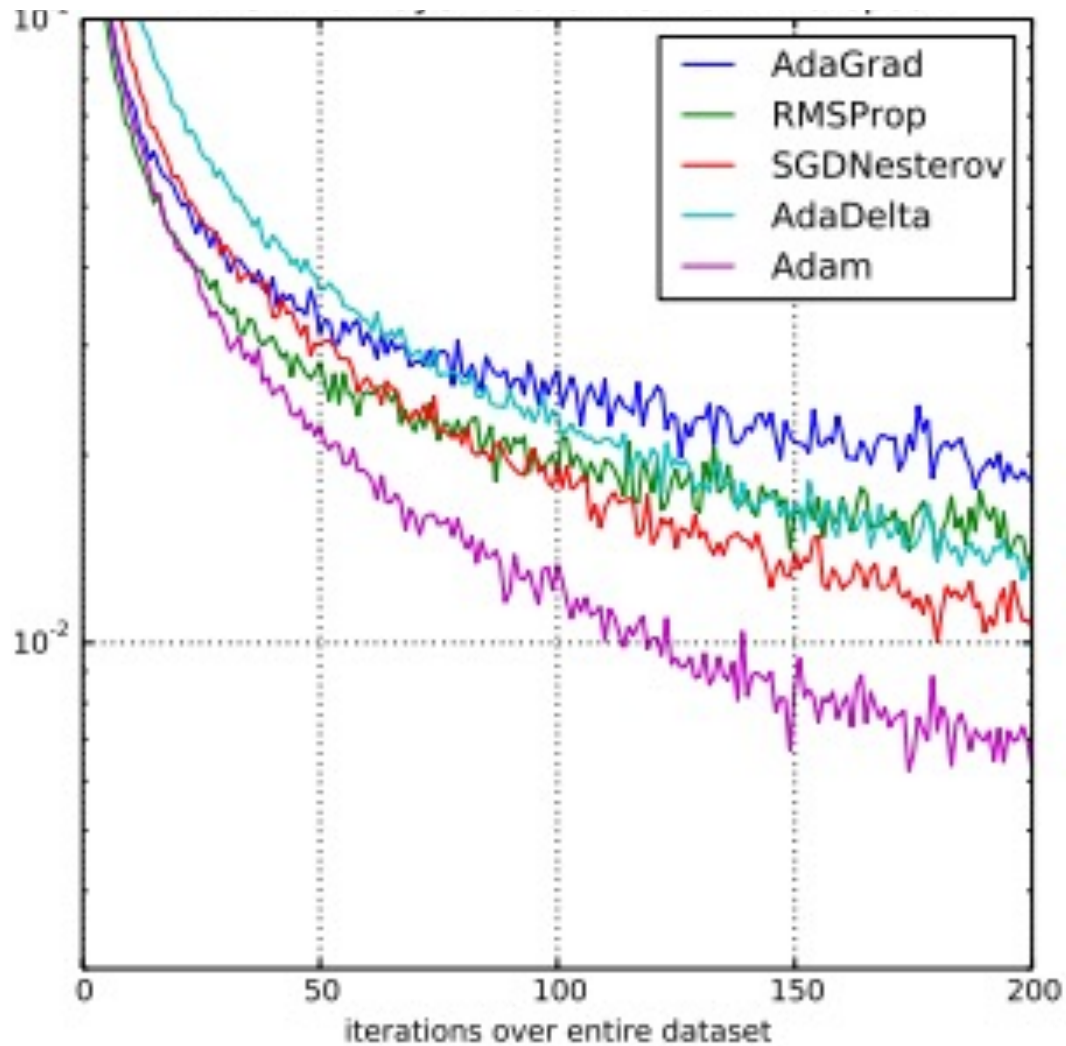
Minibatch size: b = 64
Data: m = 60k training images

Neural network with
n = 784 inputs
H = 256 hidden units
K = 10 classes

p = 203,000 (approximately)



Ex: SGD Variants



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/

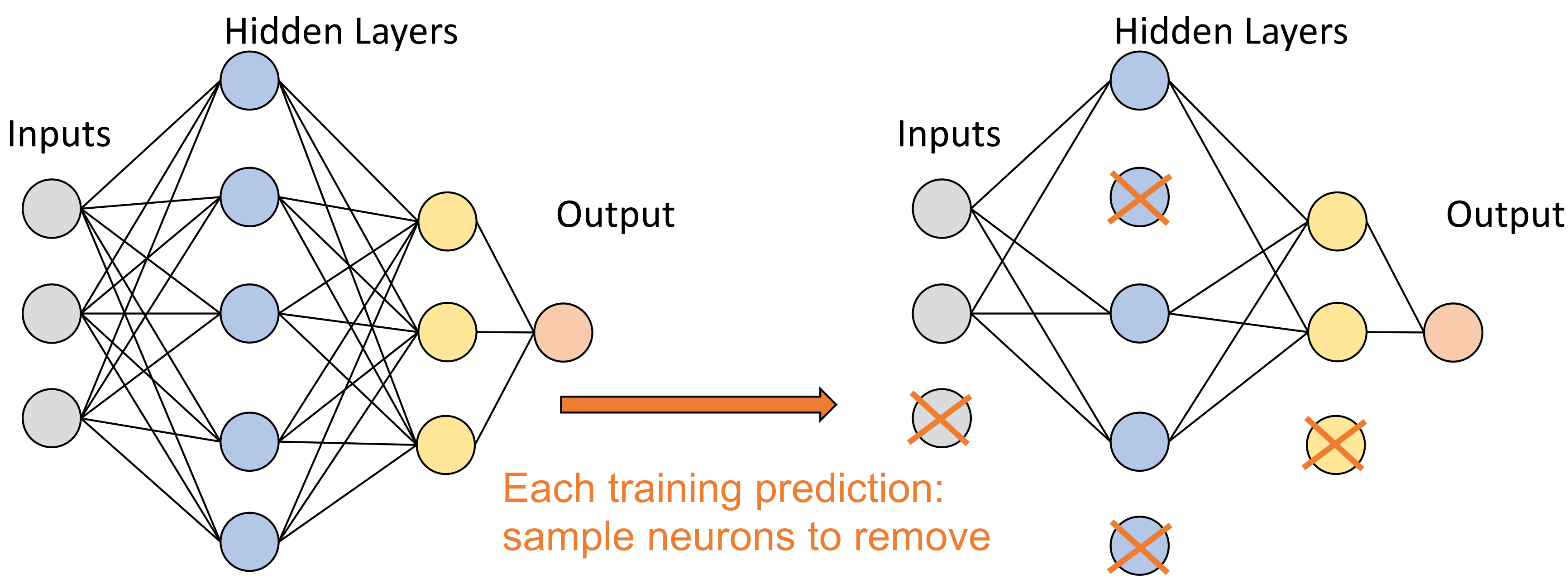
Neural Networks

- Multi-Layer Perceptrons
- Backpropagation Learning
- Architectures
  - Convolutional
  - Residual
  - Attention
- Training Deep Networks
- **More Tricks: Dropout, BatchNorm**

Dropout

[Srivastava et al 2014]

- Another recent technique
  - Randomly “block” some neurons at each step
  - Trains model to have redundancy (predictions must be robust to blocking)

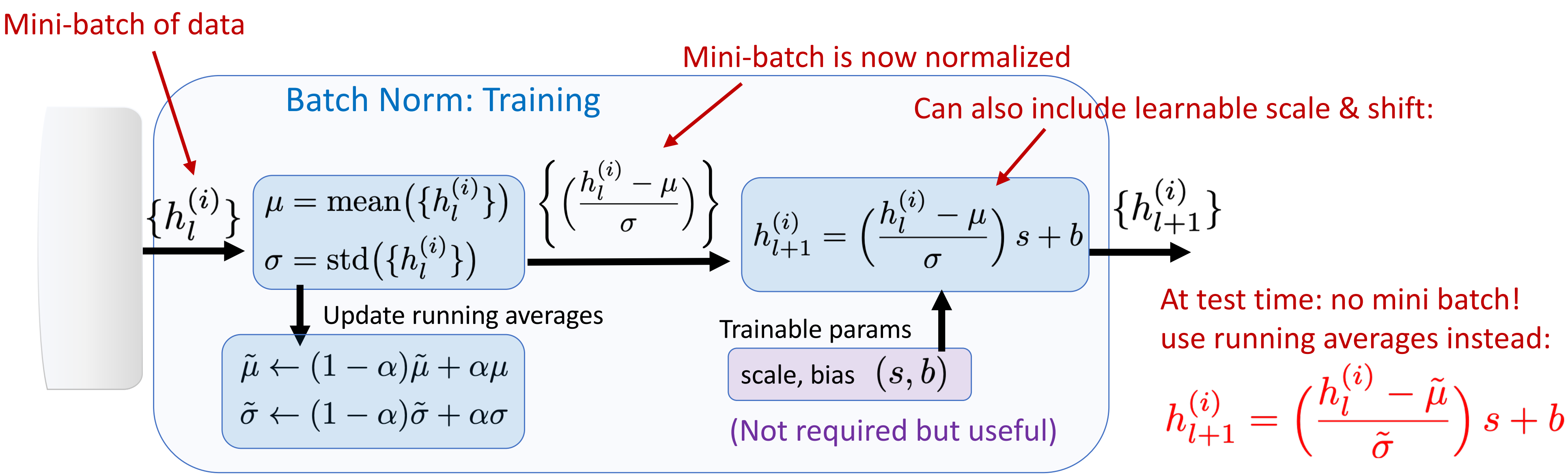



```
# ... during training ...
R  = X @ W[0] + B[0];       # linear response
H1 = Sig( R );              # activation f’n
H1 = H1 * np.random.rand(*H1.shape)<p; #drop out!
```

At test time: no deletions; sum all hidden nodes, but scale by “p” to match average response during train

Batch Normalization

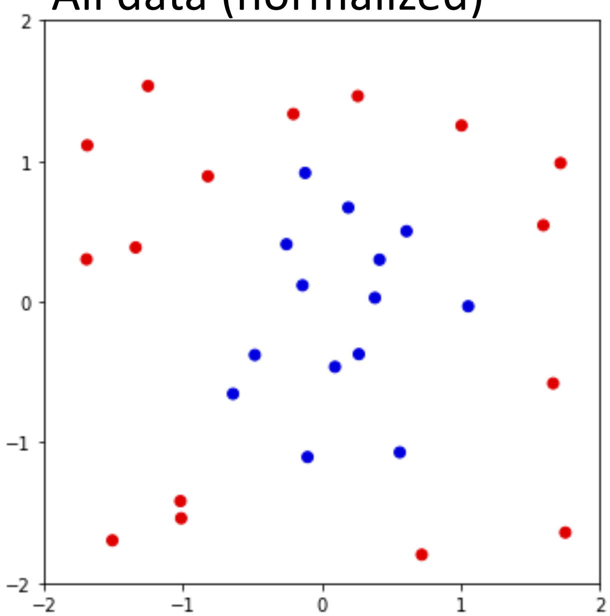
- Often, we normalize our data before input
  - What about later layers' inputs? Normalized?
- Can add a layer that "normalizes" the data
  - Parameters (m,v), same sizes as the input; out = (in – m)/v
  - Instead of "training" these parameters, just re-estimate them for each batch of data!
  - Then, estimate and "lock" their values before test time.



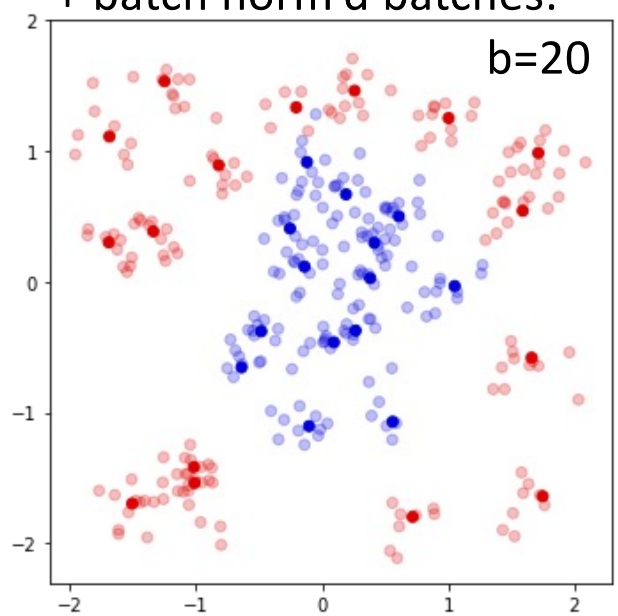
Batch Normalization

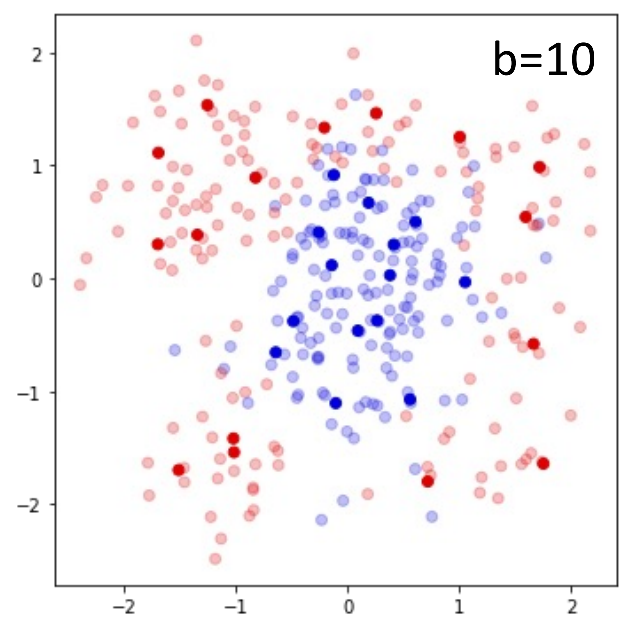
- Can view batch norm layers as a type of regularization
  - Norm’ing each subset of data adds “noise” to the samples
  - Each time we see a data point, it is slightly shifted & scaled from its “all data normalized” value
  - Amount of noise depends on the size of the batch used, “b”
  - Noisier data tends to produce smoother, simpler decision f’ns

All data (normalized)



+ batch norm’d batches:





Summary

- Neural networks, multi-layer perceptrons

- Cascade of simple perceptrons
  - Each just a linear classifier
  - Hidden units used to create new features

- Together, general function approximators
  - Enough hidden units (features) = any function
  - Can create nonlinear classifiers
  - Also used for function approximation, regression, …

- Training via backprop
  - (Stochastic) gradient descent; apply chain rule. Building block view.
  - In practice, use autograd to backpropagate

- Advanced:
  - Deep architectures: convolutional blocks, residual blocks, attention, …
  - Overfitting: dropout, batch norm, …