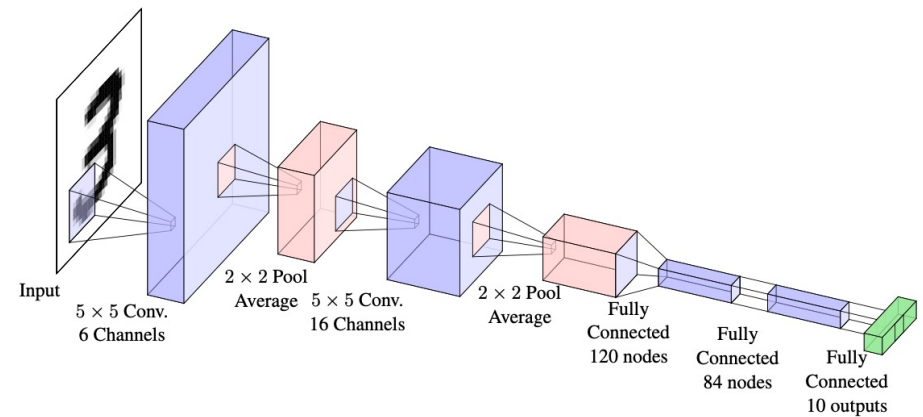
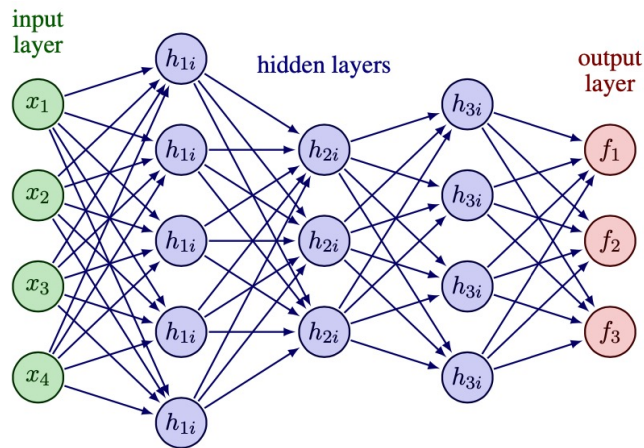


CS273A: Neural Networks



Prof. Alexander Ihler
Fall 2023

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

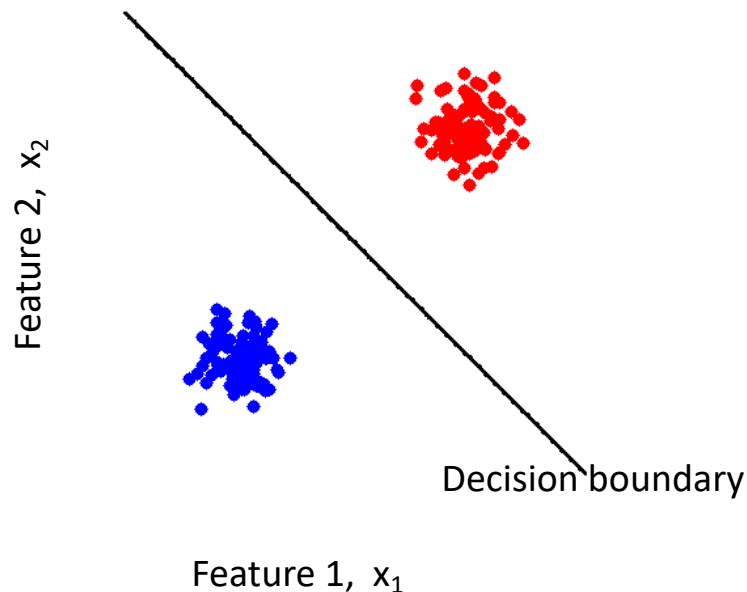
Training Deep Networks

More Tricks: Dropout, BatchNorm

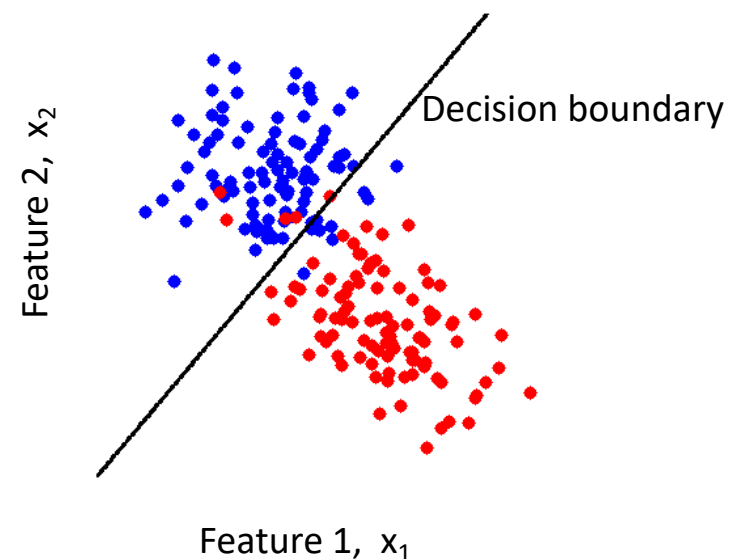
Linear classifiers (perceptrons)

- 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

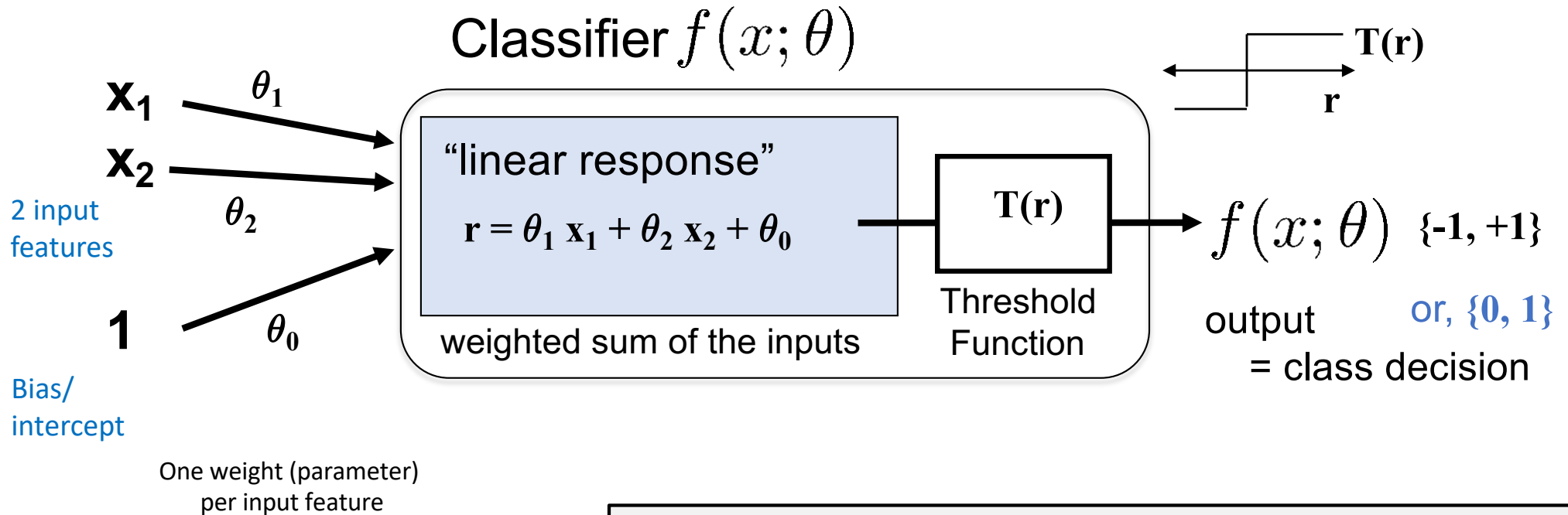
Linearly separable data



Linearly non-separable data



Linear Classifier (2 features)



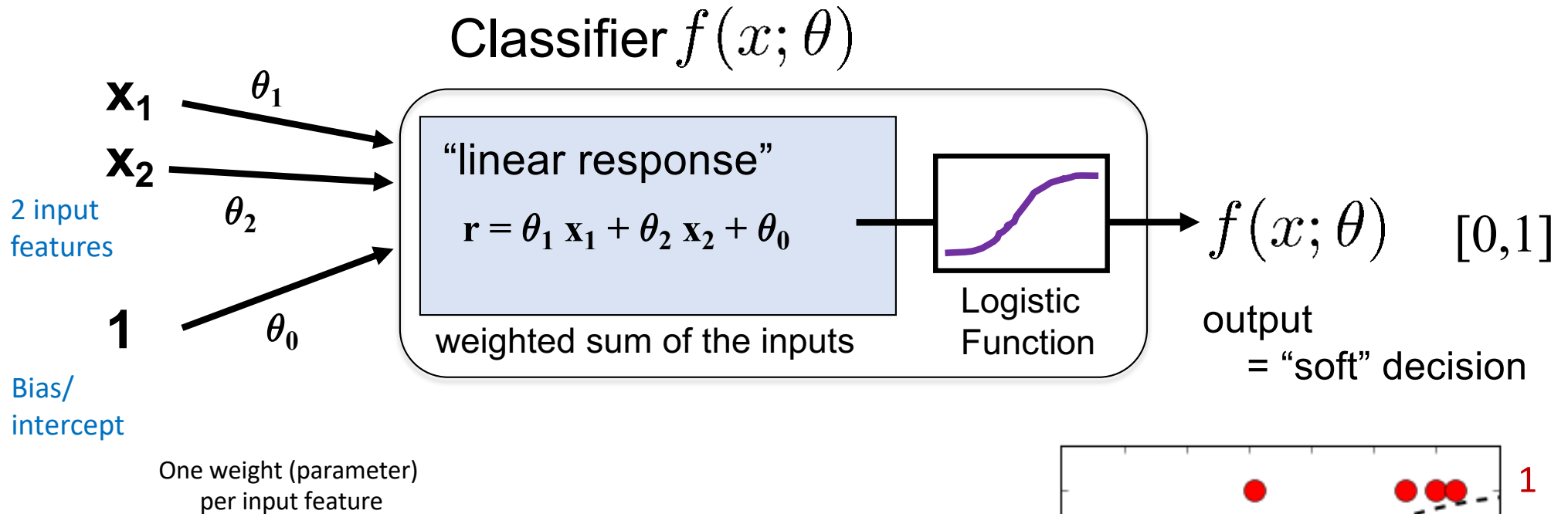
```
r = X @ theta.T      # compute linear response
yhat = 2*(r > 0)-1    # "sign": predict +1 / -1
```

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)

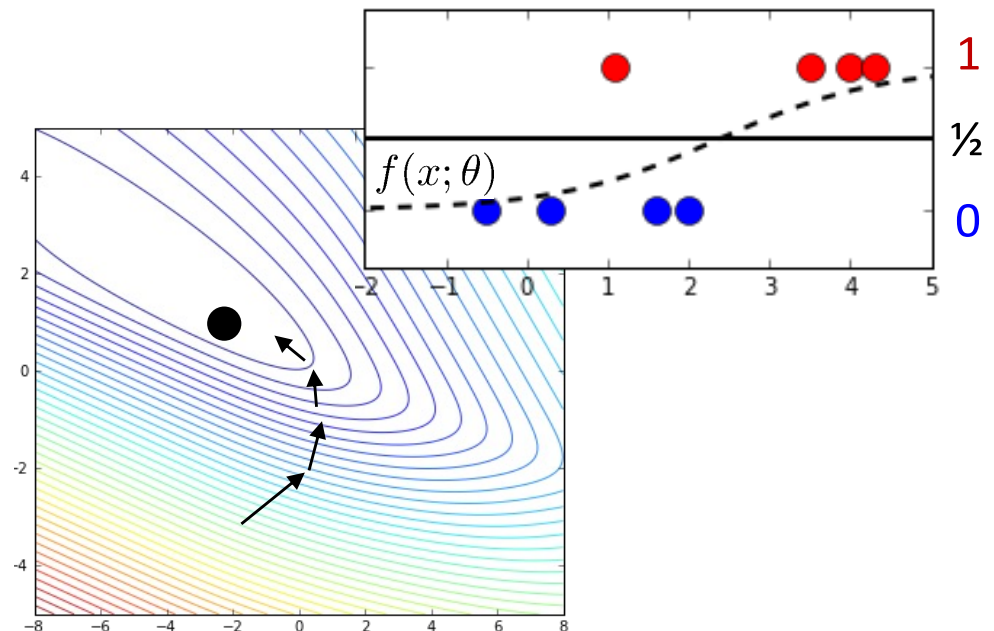
Training: Logistic Regression



Log-loss (Logistic NLL)

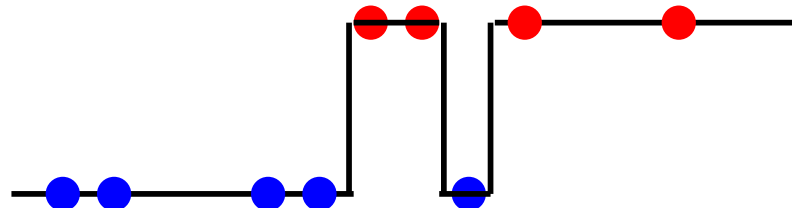
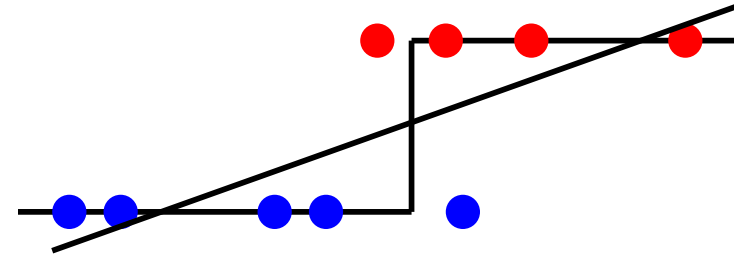
- Interpret output as probabilities
- Prediction loss is $-\log \text{Pr}(\text{true class})$
- Train via gradient descent

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



Features and perceptrons

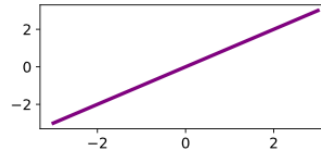
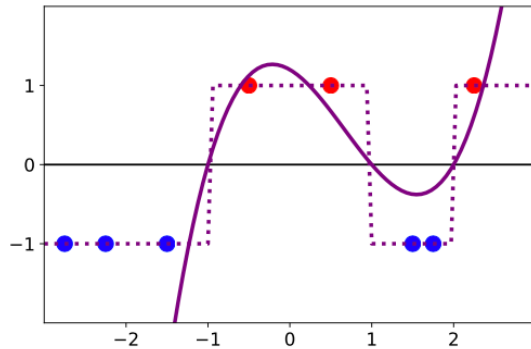
- Recall the role of features
 - Create extra features to allow more complex decision boundaries
 - Linear classifiers
 - Features $[1, x]$
 - Decision rule: $T(ax+b) = w x + b > 0$
 - Boundary $ax+b = 0 \Rightarrow$ point
 - Features $[1, x, x^2]$
 - Decision rule $T(w_2 x^2 + w_1 x + b)$
 - Boundary $w_2 x^2 + w_1 x + b = 0 = ?$
- What features can produce this decision rule?



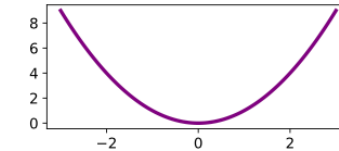
Features and perceptrons

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

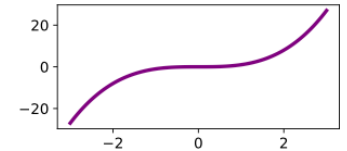
$$\Phi(x) = [1 \ x \ x^2 \ x^3 \dots]$$



$$\phi_1(x) = x$$



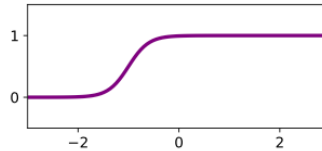
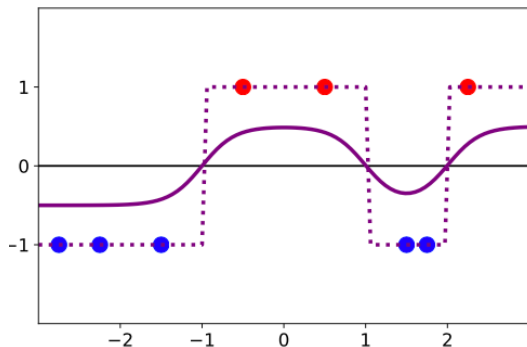
$$\phi_2(x) = x^2$$



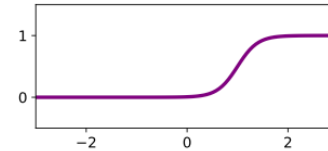
$$\phi_3(x) = x^3$$

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

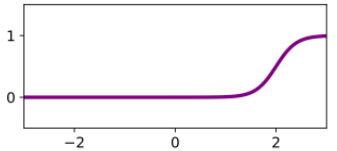
- What other kinds of features could we choose?



$$\phi_1(x) = \sigma(5x + 5)$$



$$\phi_2(x) = \sigma(5x - 5)$$



$$\phi_3(x) = \sigma(5x - 10)$$

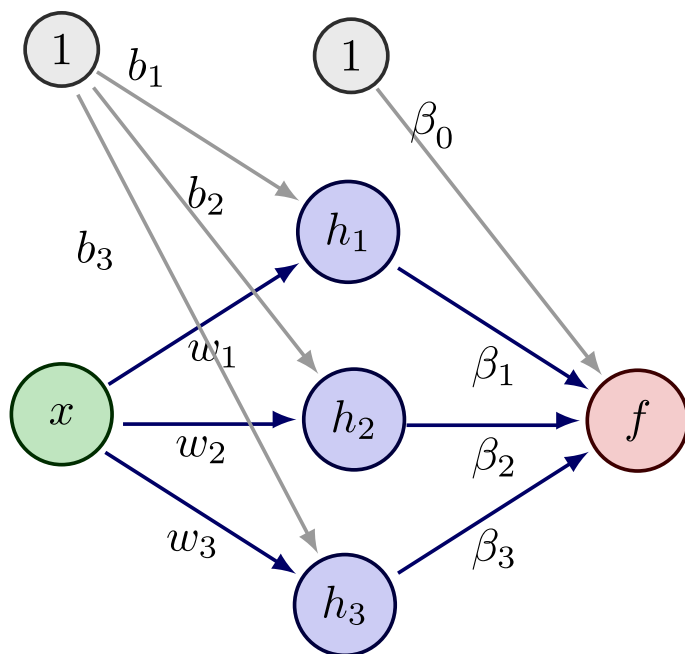
$$r(x) = b + w_1 \phi_1(x) + w_2 \phi_2(x) + w_3 \phi_3(x)$$

Multi-layer perceptron model

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

Logistic sigmoid:

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



$$h_1(x) = \sigma(b_1 + w_1x)$$

$$h_2(x) = \sigma(b_2 + w_2x)$$

$$h_3(x) = \sigma(b_3 + w_3x)$$

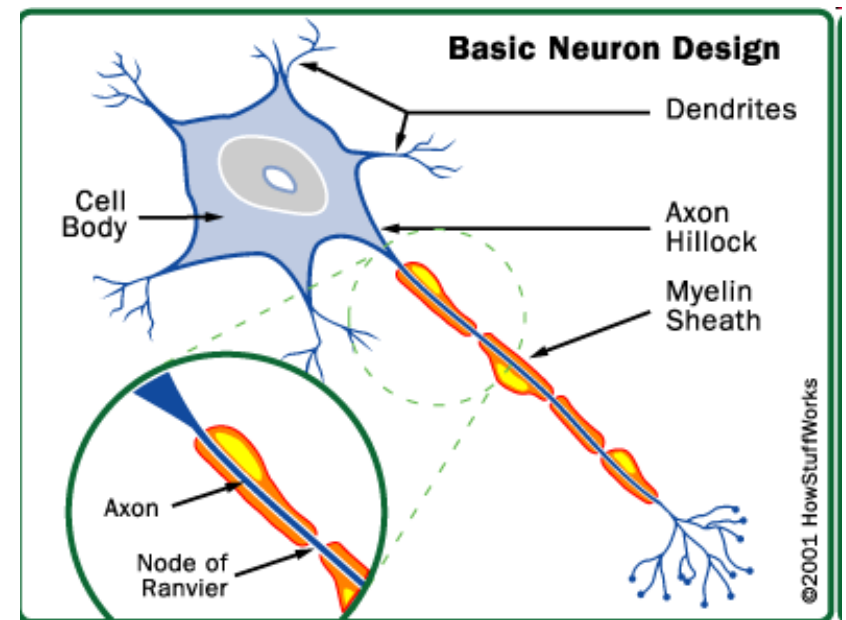
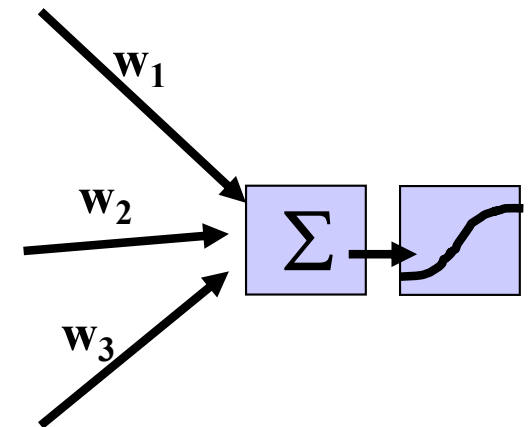
$$f(x) = \sigma(\beta_0 + \beta_1h_1 + \beta_2h_2 + \beta_3h_3)$$

Regression version:

$$f(x) = \beta_0 + \beta_1h_1 + \beta_2h_2 + \beta_3h_3$$

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....

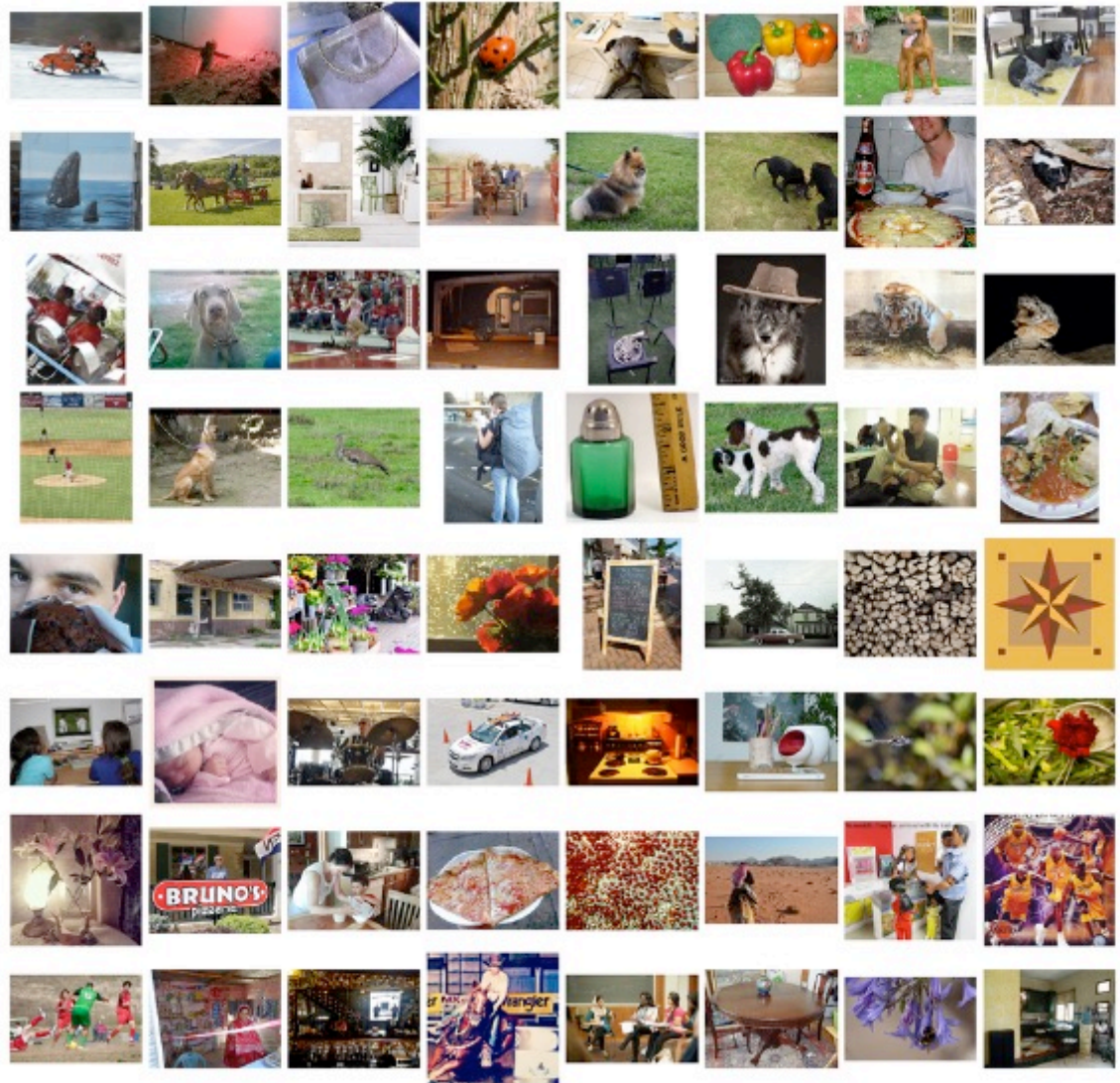
Applicaton: Image Recognition

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

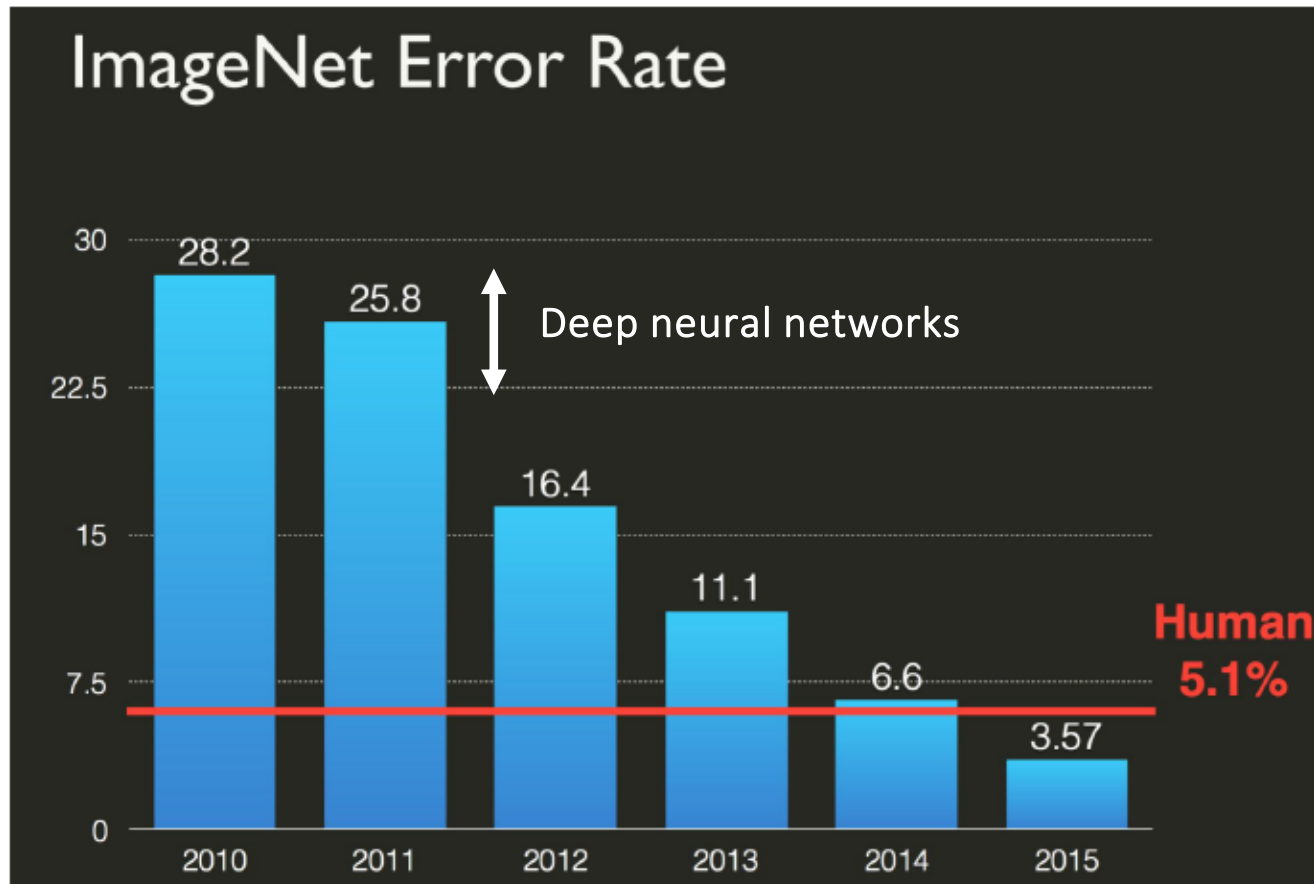
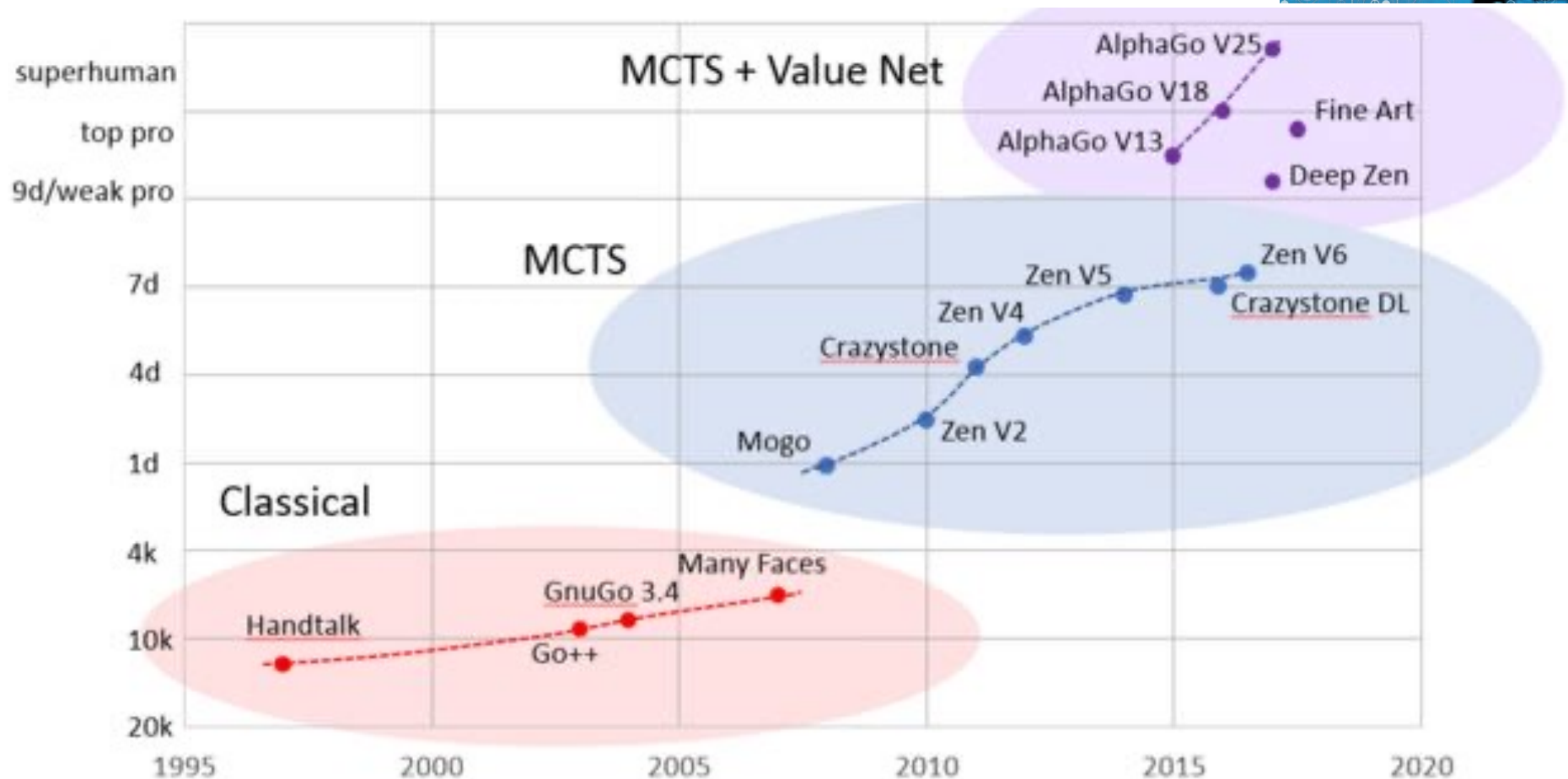


Figure from Kevin Murphy, Google

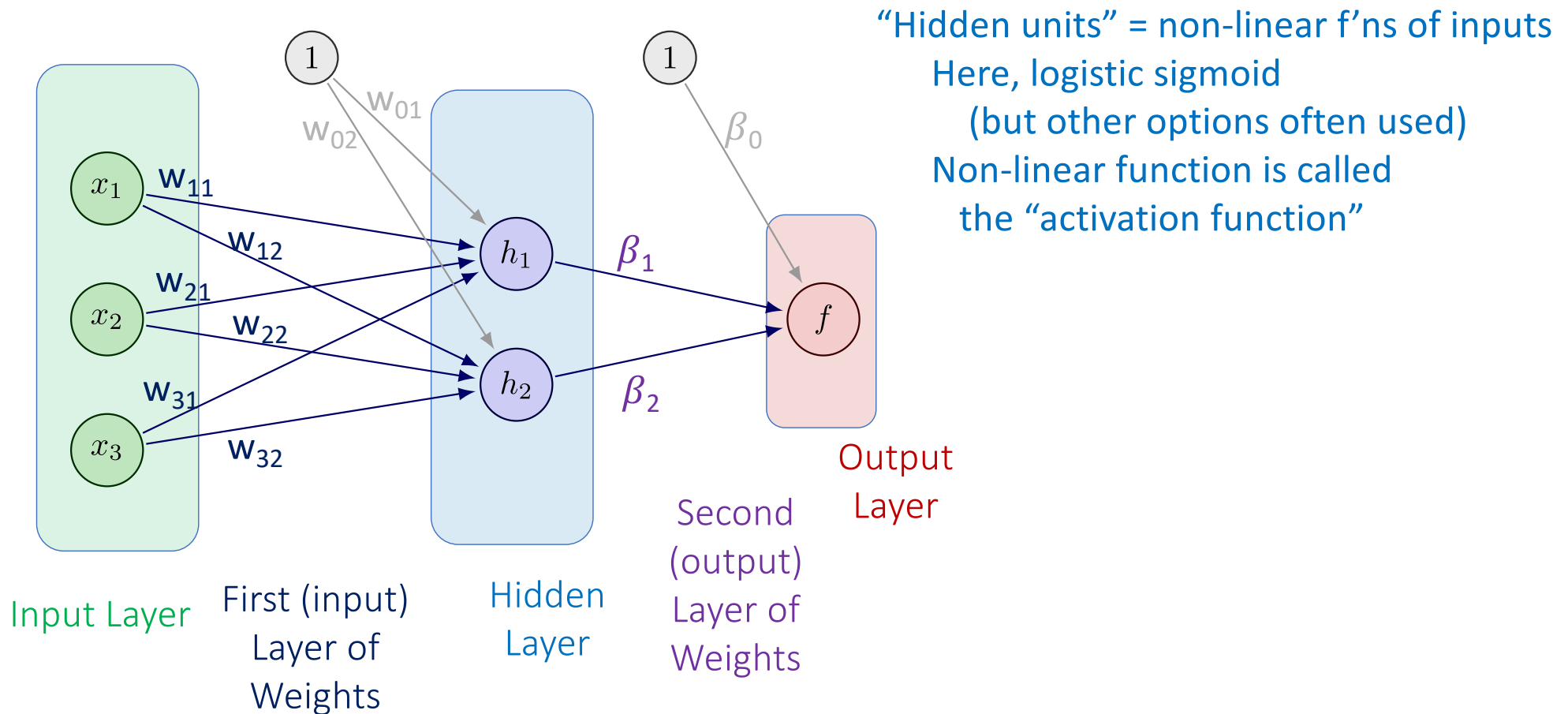
Application: Go



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

Two-layer Neural Network

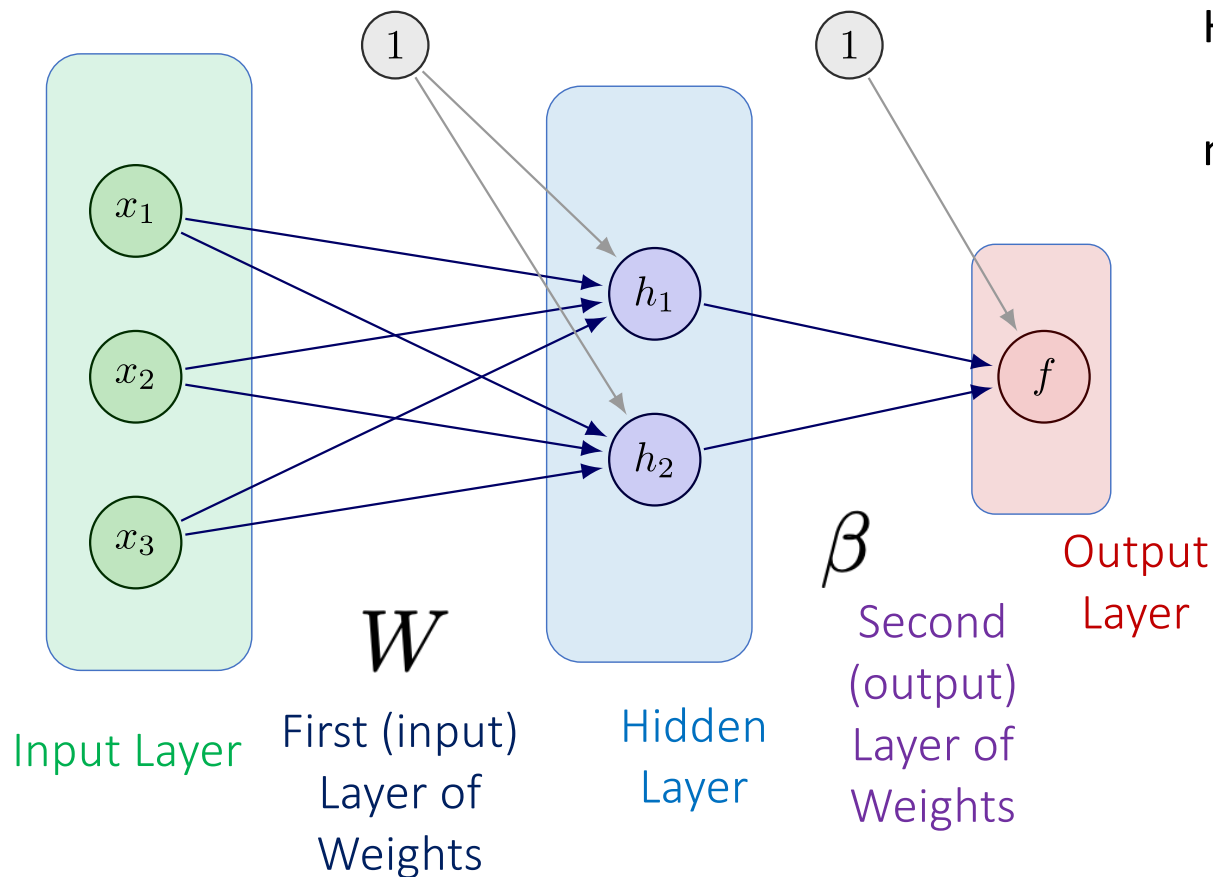
- “Two layer” = 1 hidden layer, 1 output layer



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

Neural Network Parameters

- Parameters $\theta = \{\text{all weights \& biases}\}$



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

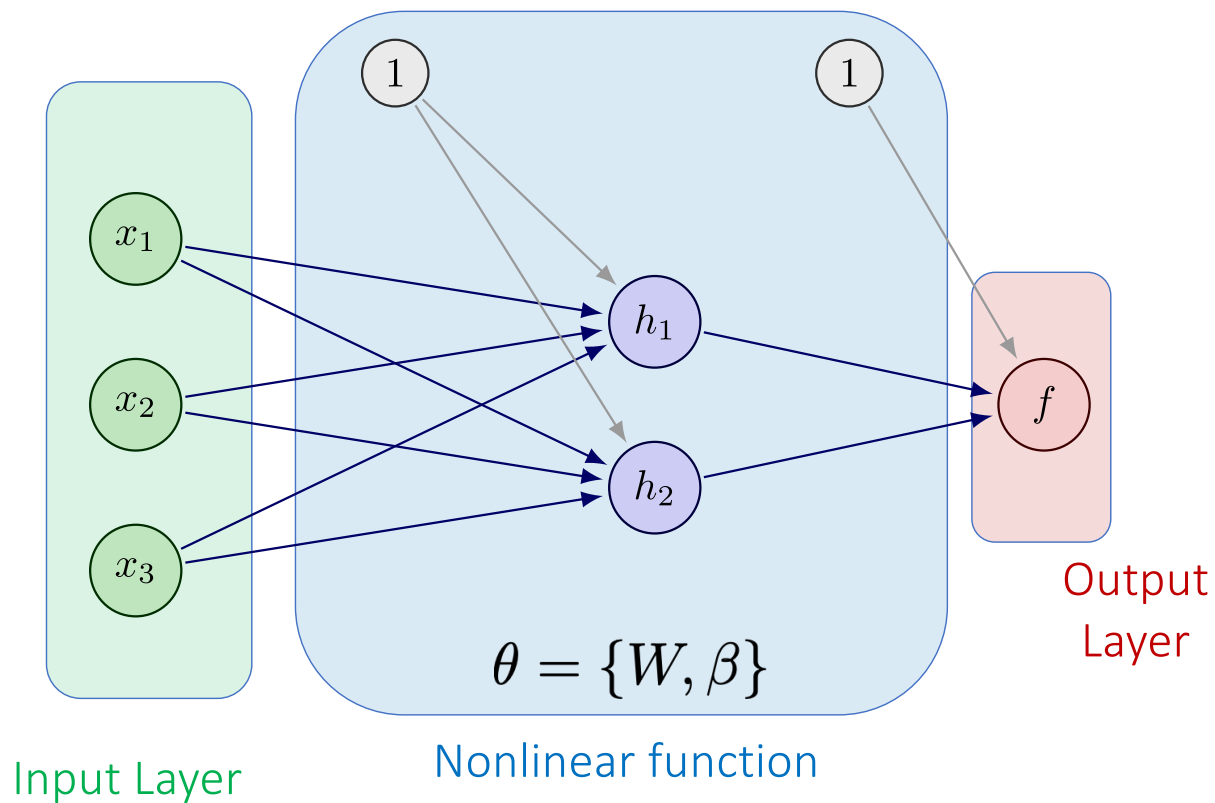
Total: $H (n+1) + (H+1)$

(approximately quadratic in layer sizes)

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

Neural Network Classifier

- Defines nonlinear mapping $f(x; \theta)$



Output values in $[0,1]$

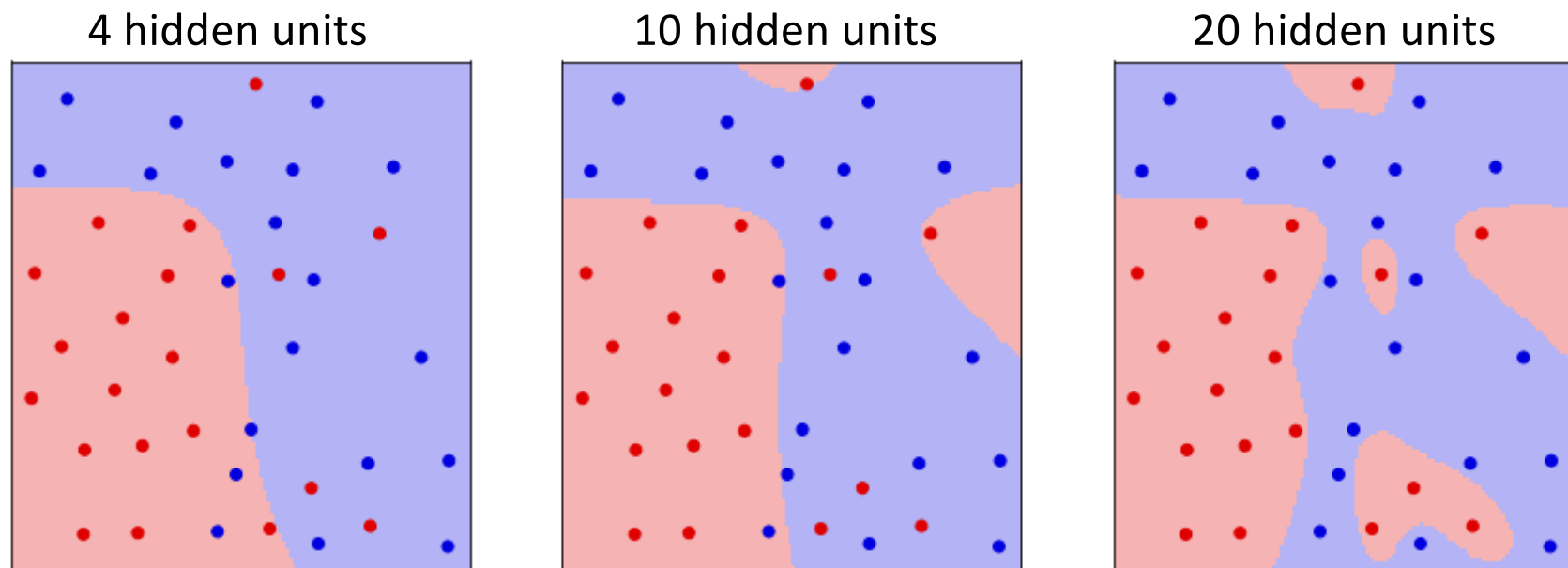
We can interpret these as e.g., class probabilities, $p(y = 1 | x)$ & select log-loss (negative log-likelihood)

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

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

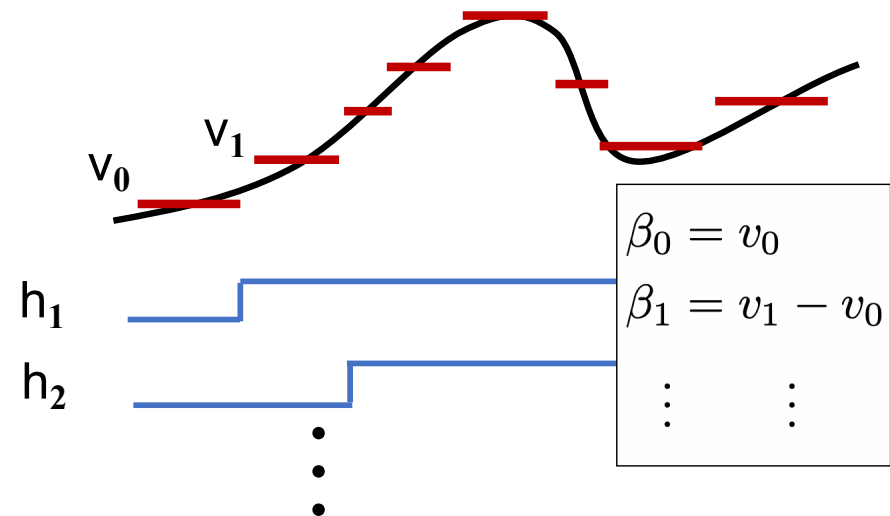
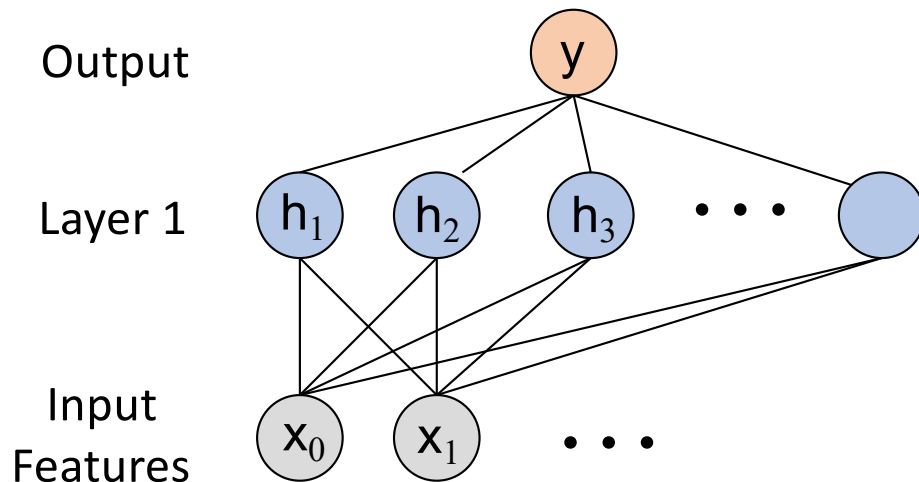
Neural Network Classifiers

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

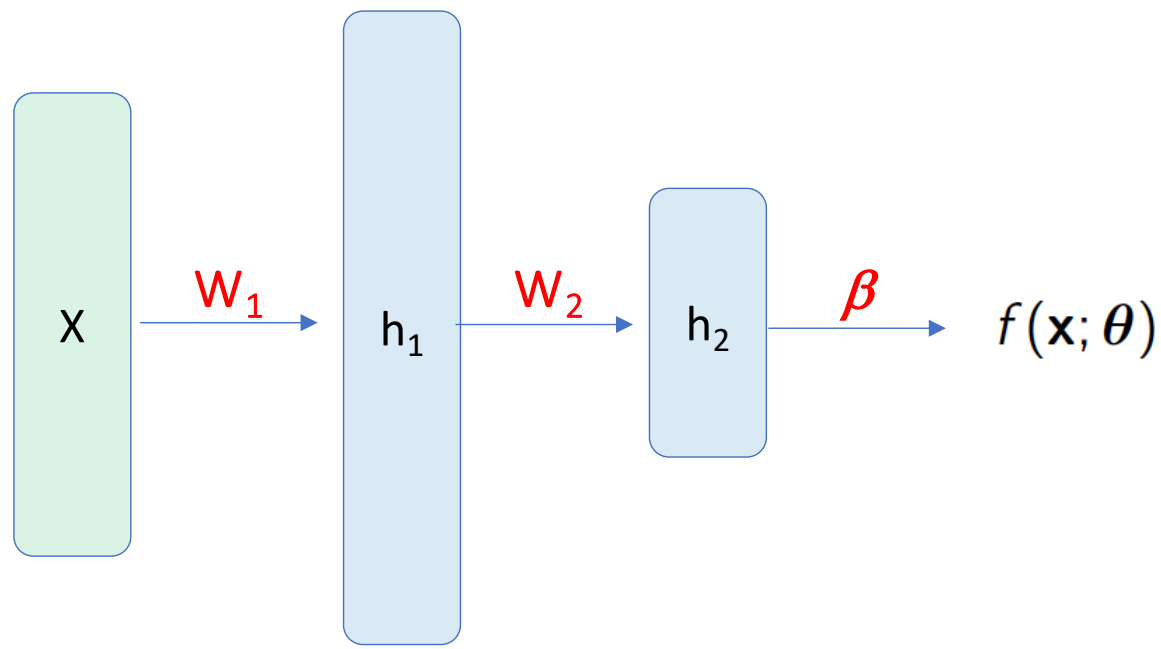


Features of MLPs

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

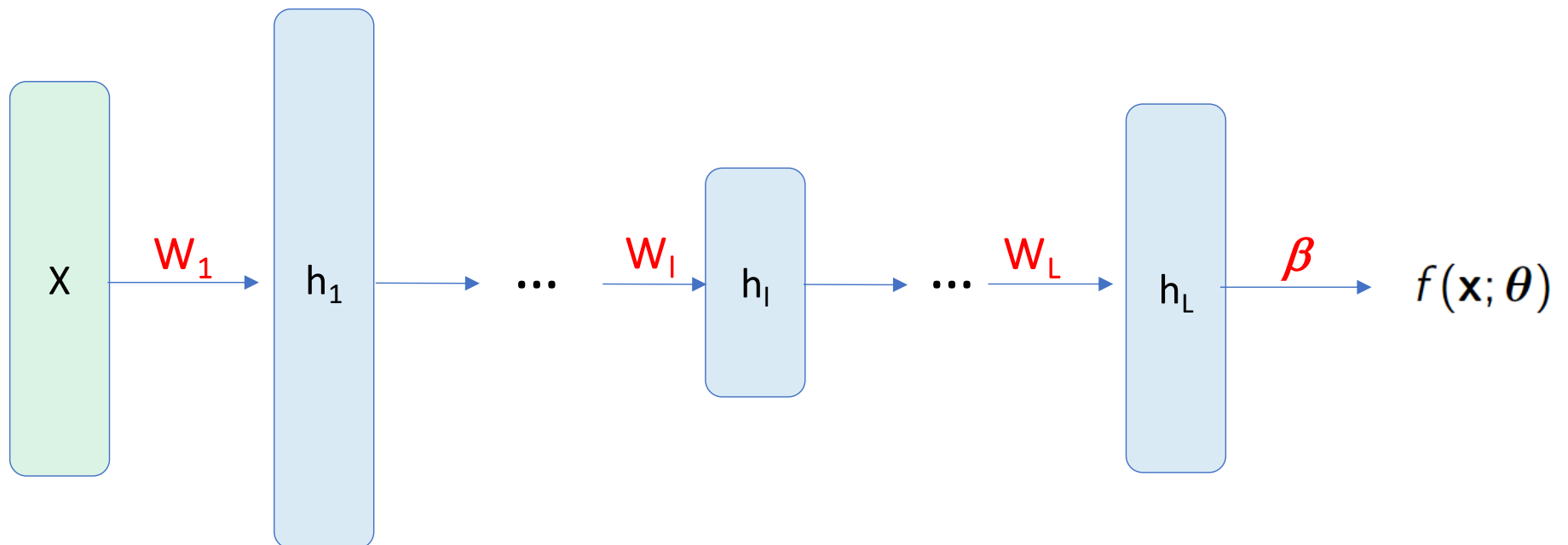


Networks with Two Hidden Layers



Each hidden unit layer can have different numbers of hidden units

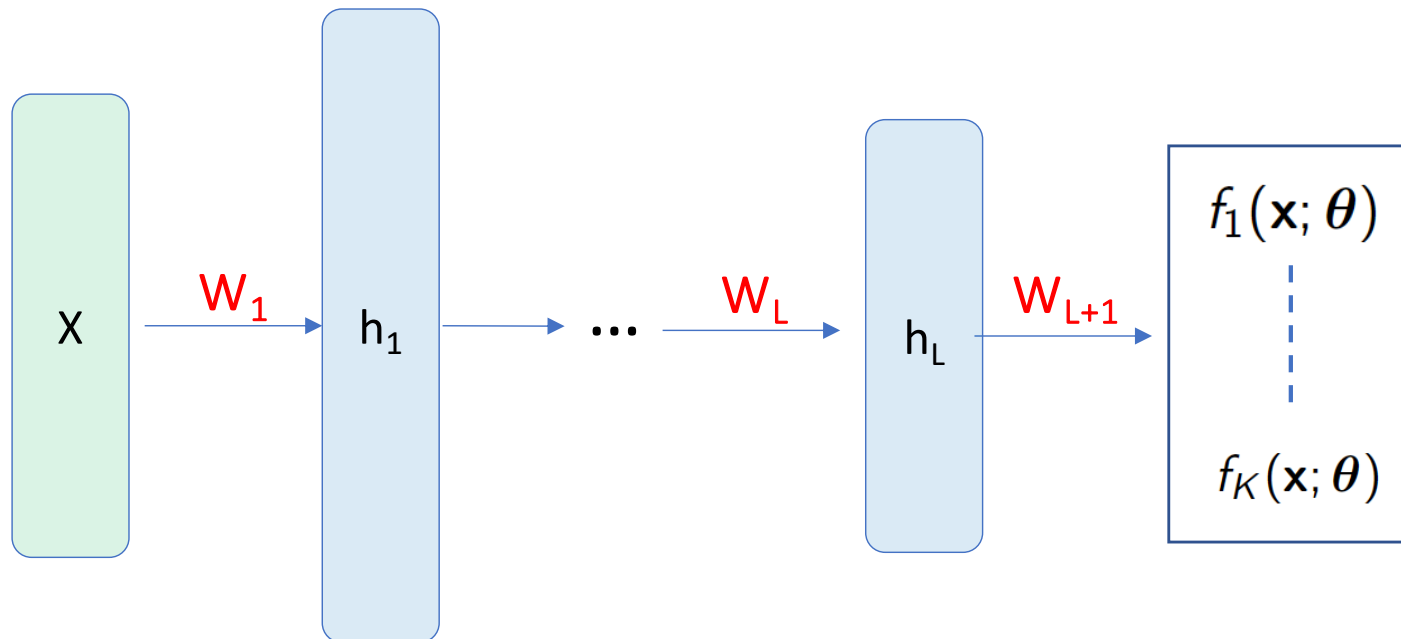
Networks with L Hidden Layers



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)

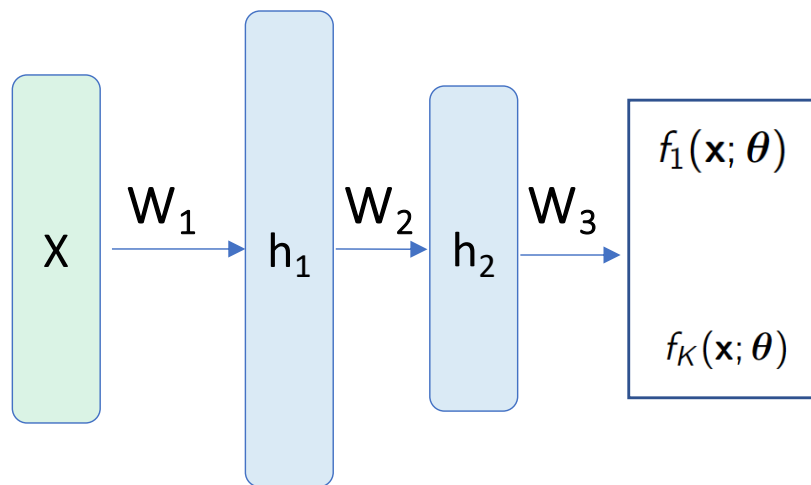
$$f_k(\underline{r}) = \frac{\exp(r_k)}{\sum_{k'} \exp(r_{k'})} \quad \left(\text{so, } \sum_k f_k = 1 \right)$$

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

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

Example: MNIST Data

3-Layer Neural Network

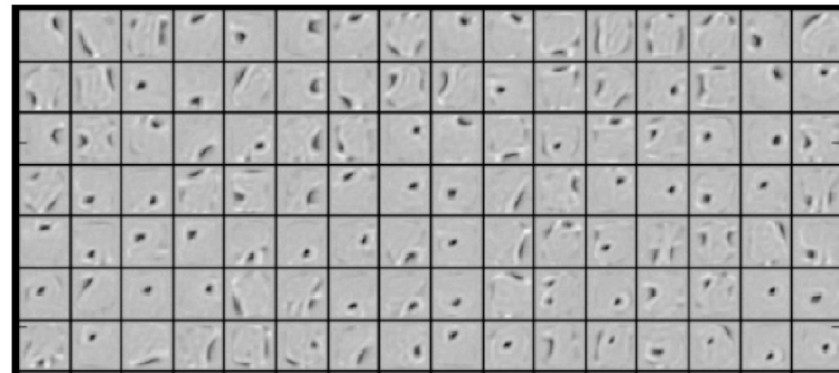


$K = 10$ classes

784-dimensional input (pixels)

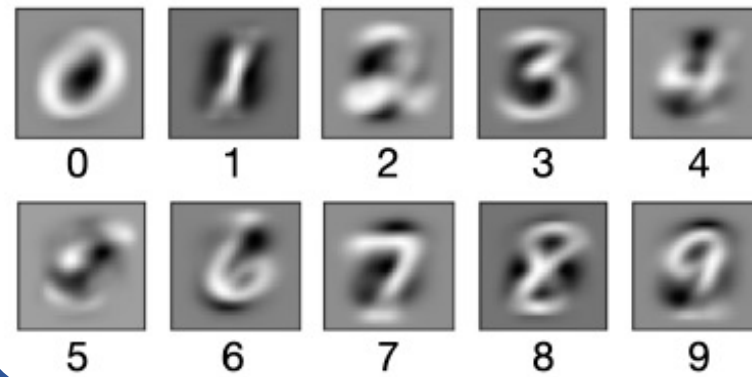
200 hidden units at each hidden layer

What do the hidden nodes learn?



(each square is a different hidden unit)

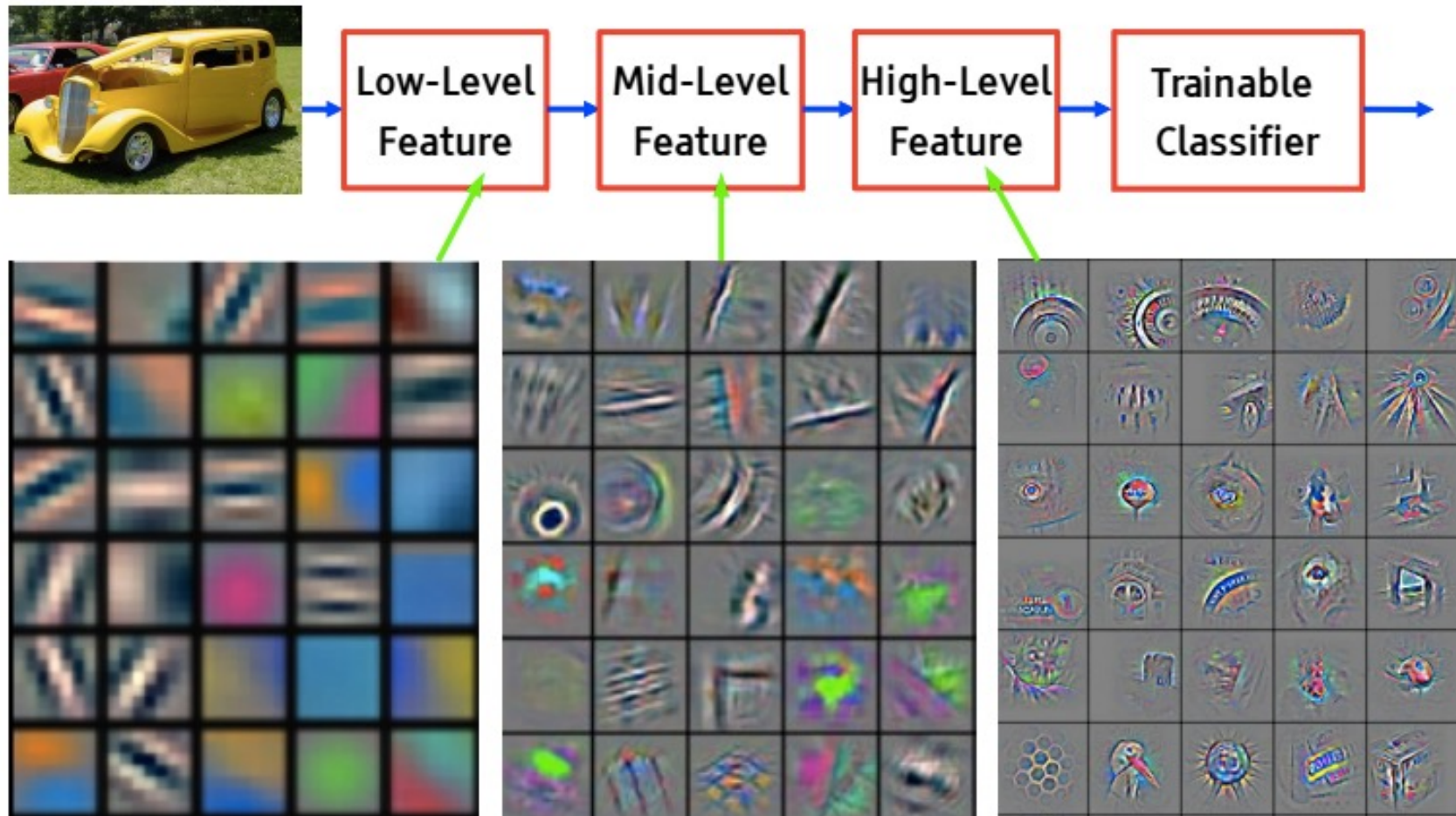
For comparison: visualizing the weights of a logistic classifier:



[Nalisnick et al., 2023]

Example: Visual Recognition

- Visualizing a convolutional network's filters [Zeiler & Fergus 2013]

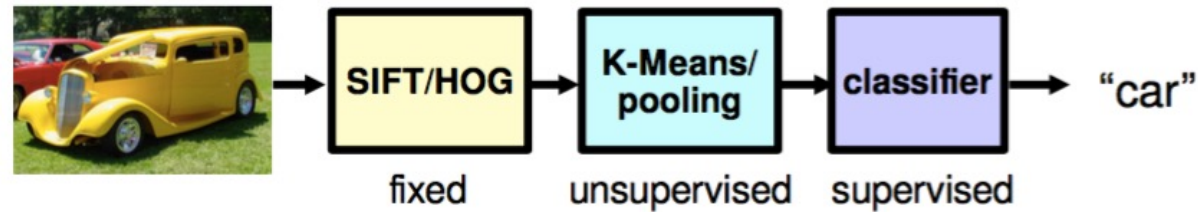


Slide image from Yann LeCun:

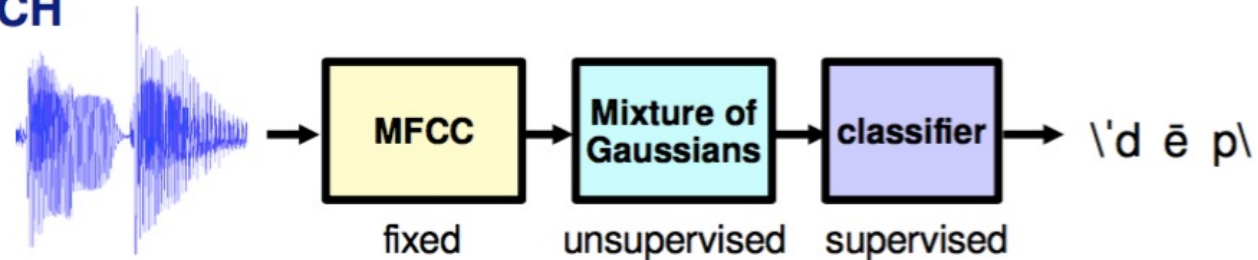
<https://drive.google.com/open?id=0BxKBnD5y2M8NcIFWSXNxa0JIZTg>

Machine Learning before Deep Neural Networks

VISION



SPEECH



NLP

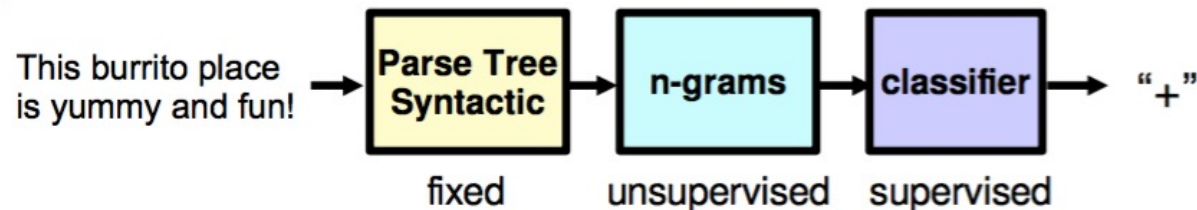
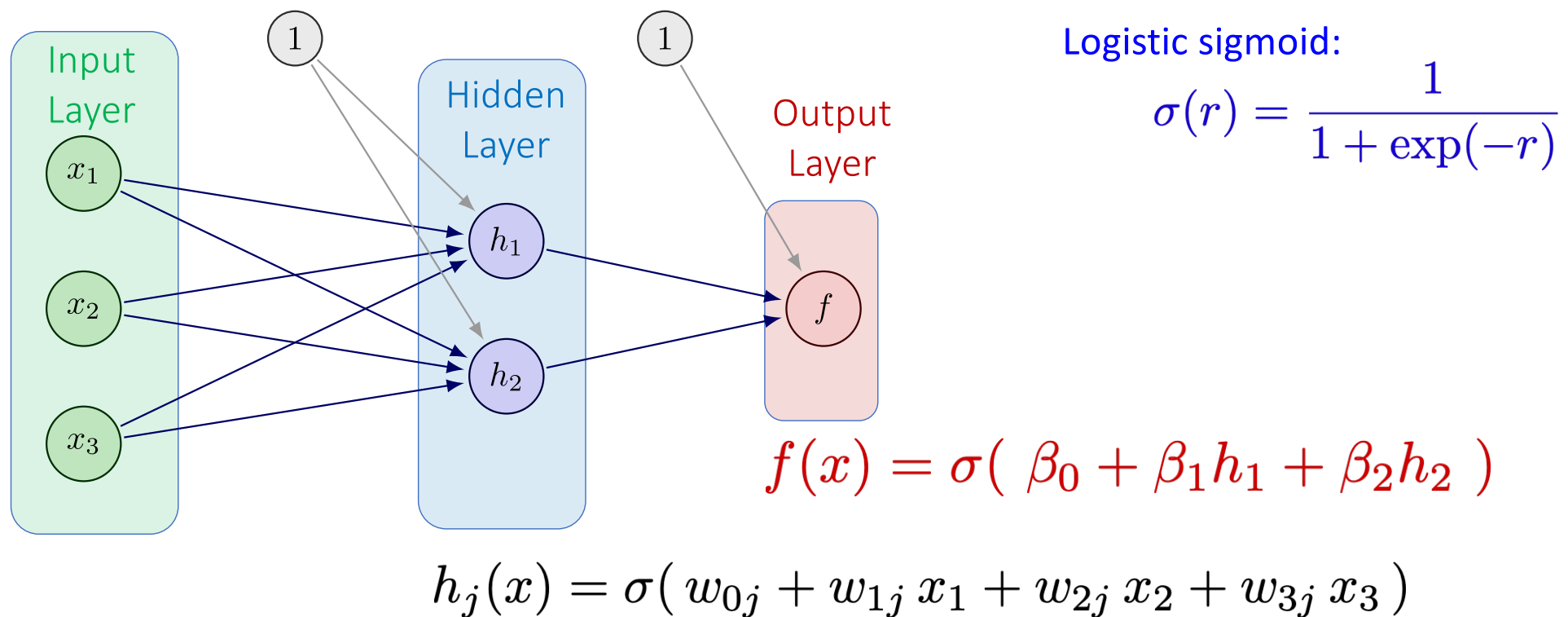


Figure from Marc'Aurelio-Ranzato

Activation functions

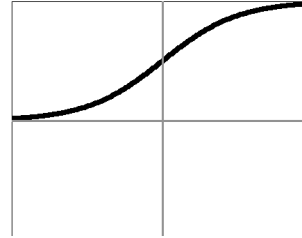
- Each hidden node applies a nonlinearity



Activation functions

Logistic

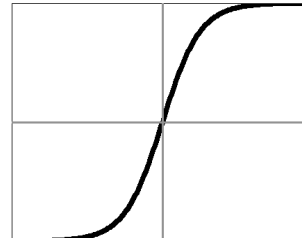
$$\sigma(z) = \frac{1}{1 + \exp(-z)}$$



$$\frac{\partial \sigma}{\partial z}(z) = \sigma(z)(1 - \sigma(z))$$

Hyperbolic
Tangent

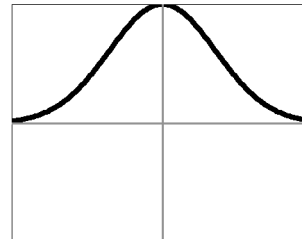
$$\sigma(z) = \frac{1 - \exp(-2z)}{1 + \exp(-2z)}$$



$$\frac{\partial \sigma}{\partial z}(z) = 1 - (\sigma(z))^2$$

Gaussian

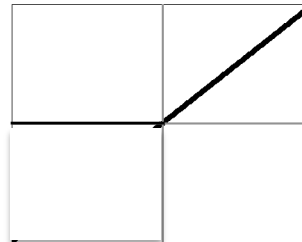
$$\sigma(z) = \exp(-z^2/2)$$



$$\frac{\partial \sigma}{\partial z}(z) = -z\sigma(z)$$

ReLU
(rectified linear)

$$\sigma(z) = \max(0, z)$$



$$\frac{\partial \sigma}{\partial z}(z) = \mathbb{1}[z > 0]$$

Linear

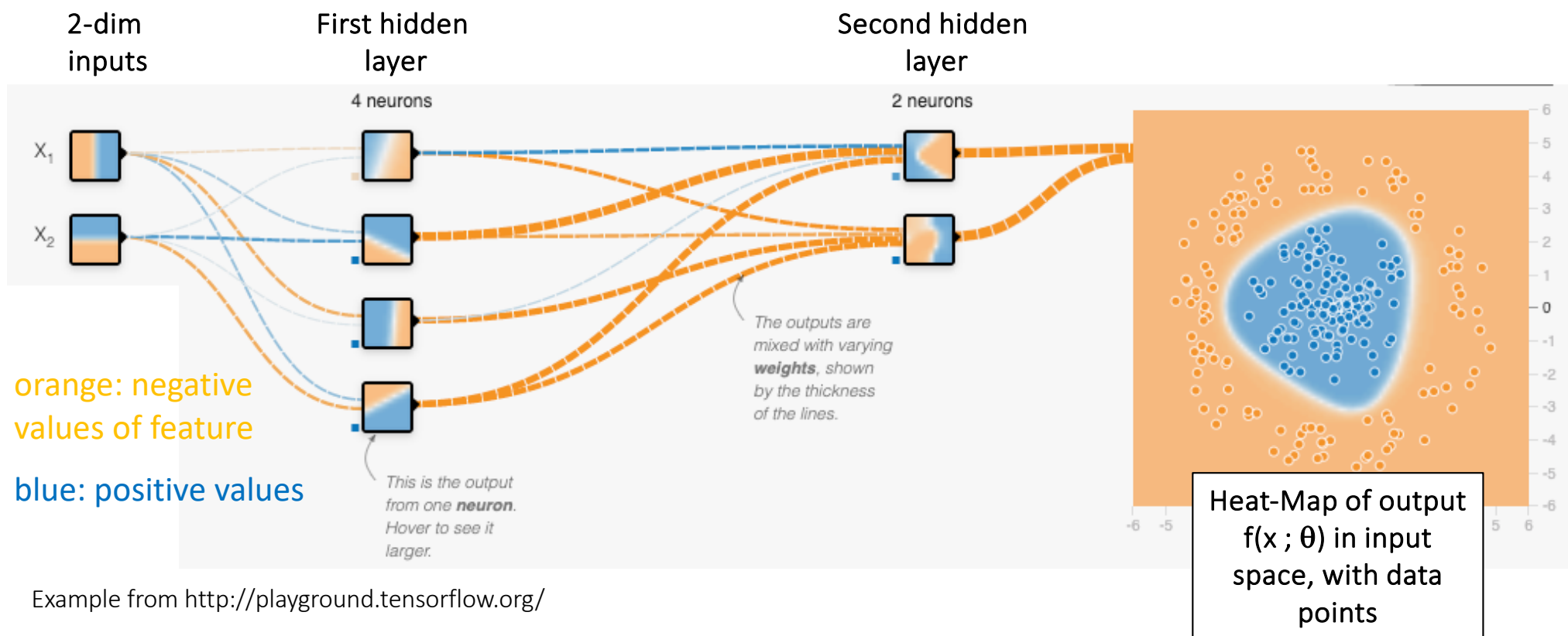
$$\sigma(z) = z$$

and many others...

Example: Simple Network with 2 Hidden Layers

Two layers

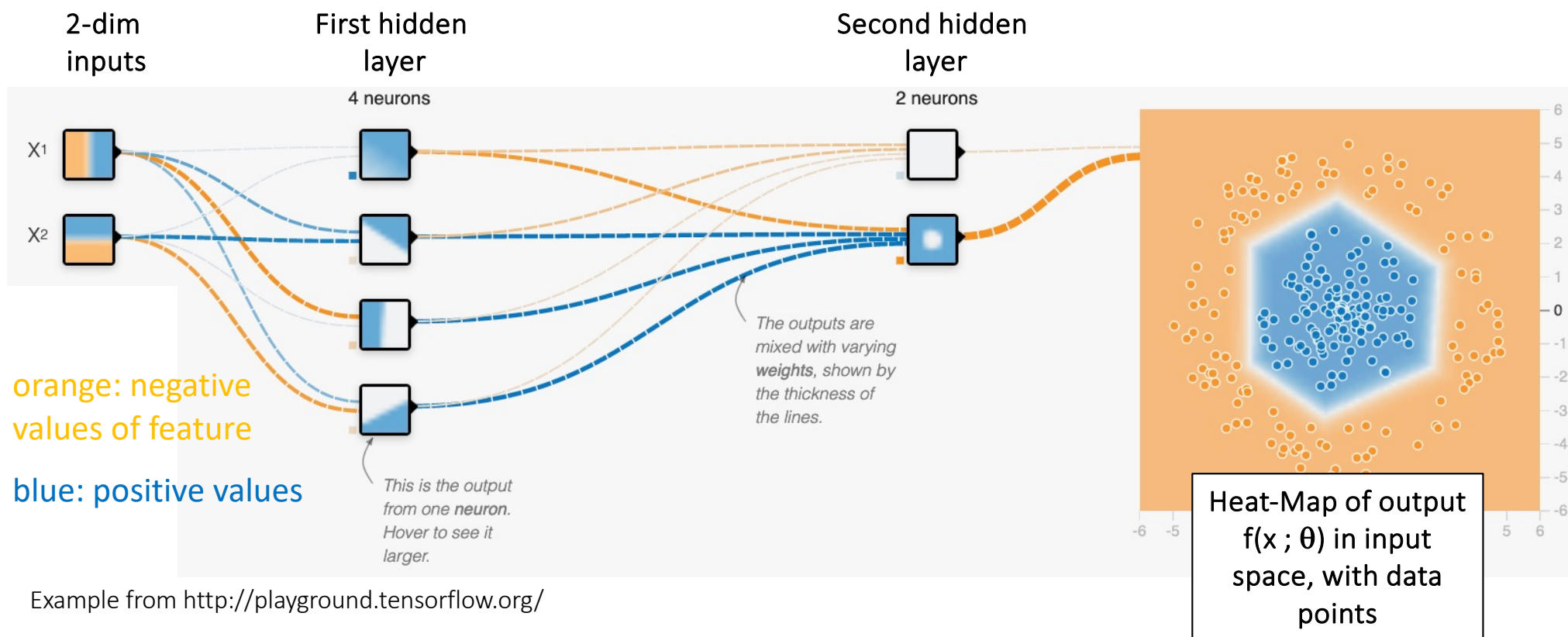
- Layer 1: 4 hidden nodes
- Layer 2: 2 hidden nodes
- Activation: tanh



Example: 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...

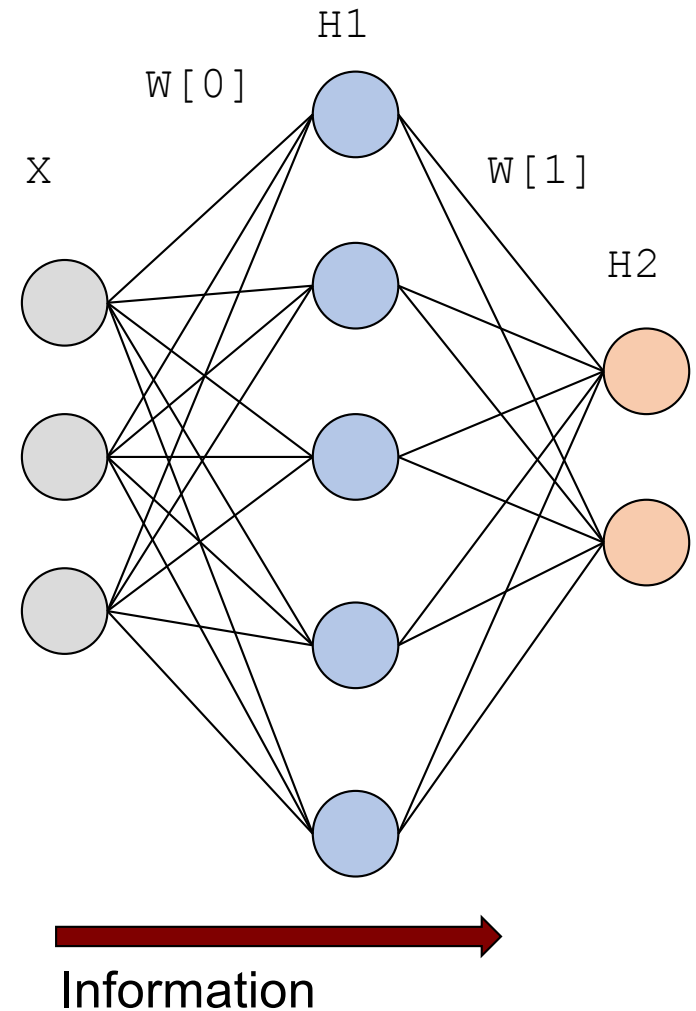


Feed-forward networks

- 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

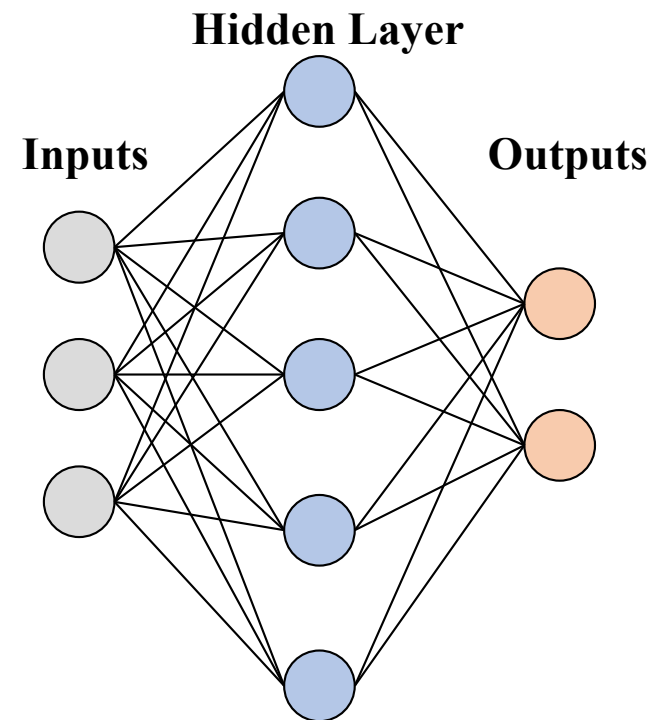
Attention

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



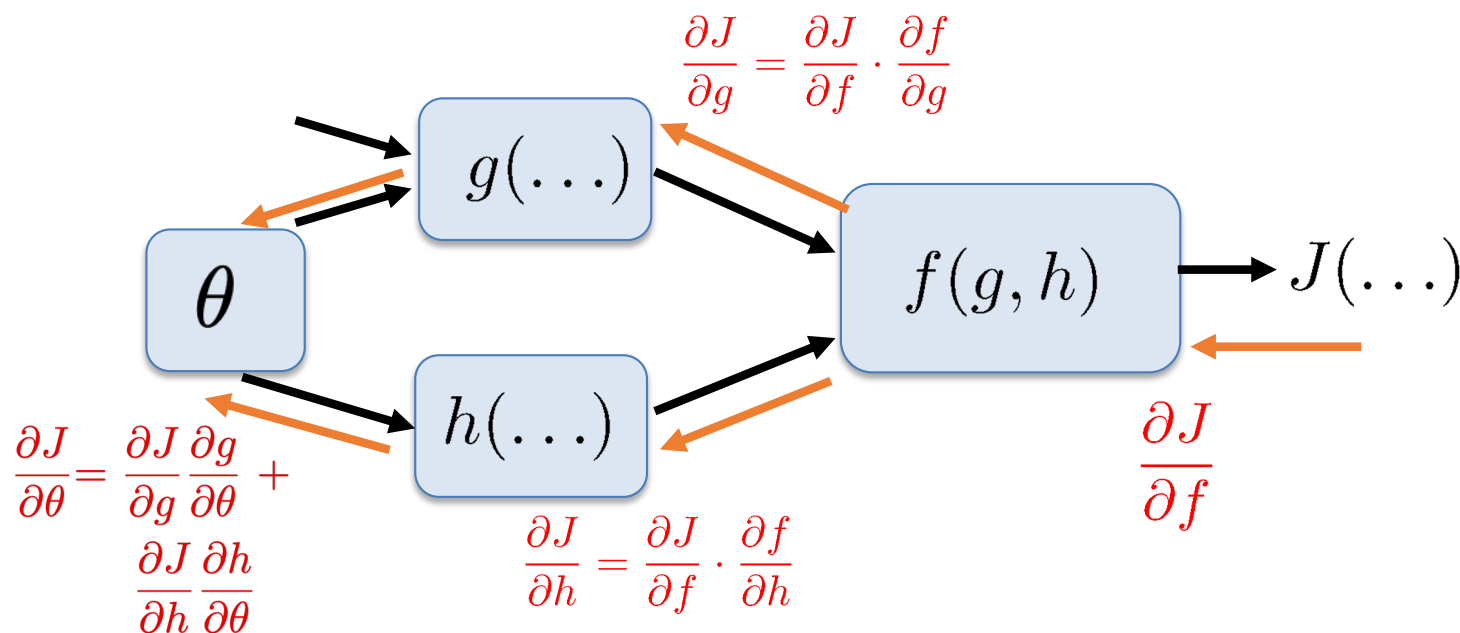
Gradient calculations

- Think of NNs as “schematics” made of smaller functions

- Building blocks: summations & nonlinearities
- For derivatives, just apply the chain rule, etc!

Backward pass:

- sum incoming derivative messages
- multiply by own slope wrt each argument
- Send back to each arg

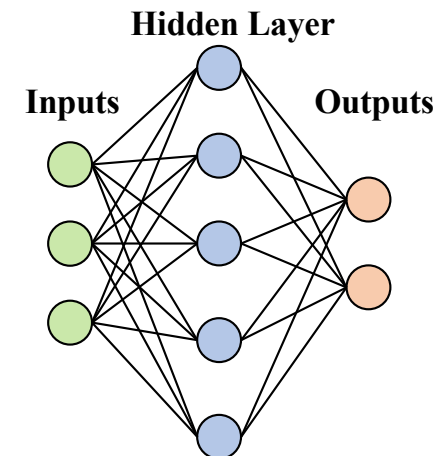


save & reuse info (g,h) from forward computation!

Ex: $f(g, h) = g^2 h$

$$\frac{\partial J}{\partial g} = \frac{\partial J}{\partial f} \cdot 2g(\cdot)h(\cdot)$$

$$\frac{\partial J}{\partial h} = \frac{\partial J}{\partial f} \cdot g^2(\cdot)$$



Backpropagation

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

$$\begin{aligned}\frac{\partial J}{\partial \beta_{jk}} &= \sum_{k'} \left(\frac{\partial J}{\partial f_{k'}} \right) \left(\frac{\partial f_{k'}}{\partial s_{k'}} \right) \left(\frac{\partial s_{k'}}{\partial \beta_{jk}} \right) \\ &= \sum_{k'} \left(-2(y_{k'} - f_{k'}) \right) \left(\sigma'(s_{k'}) \right) \left(\mathbb{1}[k' = k] h_j \right) \\ &= -2(y_k - f_k) \sigma'(s_k) h_j\end{aligned}$$

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

Forward pass

Loss function

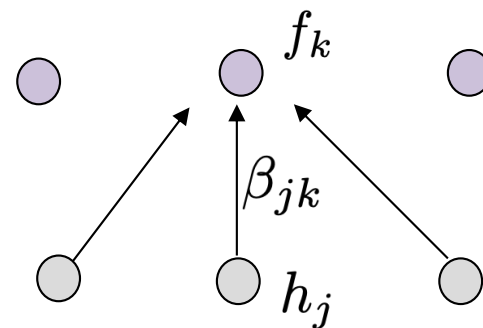
$$J(\theta) = \sum_k (y_k - f_k)^2$$

Output layer k

$$f_k = \sigma(s_k) = \sigma\left(\sum_j \beta_{jk} h_j\right)$$

Hidden layer

$$h_j = \sigma(r_j) = \sigma\left(\sum_i w_{ij} x_i\right)$$



Backpropagation

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

$$\begin{aligned}
 \frac{\partial J}{\partial \beta_{jk}} &= \sum_{k'} \left(\frac{\partial J}{\partial f_{k'}} \right) \left(\frac{\partial f_{k'}}{\partial s_{k'}} \right) \left(\frac{\partial s_{k'}}{\partial \beta_{jk}} \right) \\
 &= \sum_{k'} \left(-2(y_{k'} - f_{k'}) \right) \left(\sigma'(s_{k'}) \right) \left(\mathbb{1}[k' = k] h_j \right) \\
 &= \underbrace{-2(y_k - f_k) \sigma'(s_k)}_{\lambda_k} h_j
 \end{aligned}$$

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

$$\begin{aligned}
 \frac{\partial J}{\partial w_{ij}} &= \sum_{k'} \left(\frac{\partial J}{\partial f_{k'}} \right) \left(\frac{\partial f_{k'}}{\partial s_{k'}} \right) \left(\frac{\partial s_{k'}}{\partial w_{ij}} \right) \\
 &\quad \sum_{j'} \left(\frac{\partial s_{k'}}{\partial h_{j'}} \right) \left(\frac{\partial h_{j'}}{\partial r_{j'}} \right) \left(\frac{\partial r_{j'}}{\partial w_{ij}} \right) \\
 &\quad \sum_{j'} \left(\beta_{j'k'} \right) \left(\sigma'(r_{j'}) \right) \left(\mathbb{1}[j' = j] x_i \right)
 \end{aligned}$$

$$\frac{\partial J}{\partial w_{ij}} = \sum_{k'} \left(\underbrace{-2(y_{k'} - f_{k'})}_{\lambda_{k'}} \right) \left(\sigma'(s_{k'}) \right) \left(\beta_{jk'} \right) \left(\sigma'(r_j) \right) \left(x_i \right)$$

Forward pass

Loss function

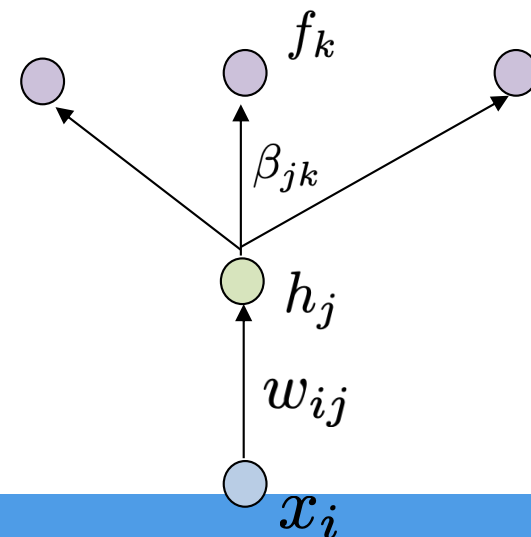
$$J(\theta) = \sum_k (y_k - f_k)^2$$

Output layer k

$$f_k = \sigma(s_k) = \sigma \left(\sum_j \beta_{jk} h_j \right)$$

Hidden layer

$$h_j = \sigma(r_j) = \sigma \left(\sum_i w_{ij} x_i \right)$$



Backpropagation

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

$$\frac{\partial J}{\partial \beta_{jk}} = \underbrace{-2(y_k - f_k)}_{\lambda_k} \sigma'(s_k) h_j$$

$$\frac{\partial J}{\partial w_{ij}} = \sum_{k'} \left(\underbrace{-2(y_{k'} - f_{k'})}_{\lambda_{k'}} \right) \left(\sigma'(s_{k'}) \right) \left(\beta_{jk'} \right) \left(\sigma'(r_j) \right) \left(x_i \right)$$

Forward pass

Loss function

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

Output layer k

$$f_k = \sigma(s_k) = \sigma\left(\sum_j \beta_{jk} h_j\right)$$

Hidden layer

$$h_j = \sigma(r_j) = \sigma\left(\sum_i w_{ij} x_i\right)$$

```
# X : (1xN1)
# W1 : (N2xN1)
H = Sig(X @ W[0])
# H : (1xN2)
# W2 : (N3xN2)
F = Sig(H @ W[1])
# Yh : (1xN3)
```

```
L2 = (Y-F) * dSig(S) # (1xN3)
```

```
G2 = L2.T @ H # (N3x1) * (1xN2) = (N3xN2)
```

```
L1 = (L2 @ W[1]) * dSig(R) # (1xN3) * (N3xN2) = (1xN2)
```

```
G1 = L1.T @ X # (N2xN1)
```

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

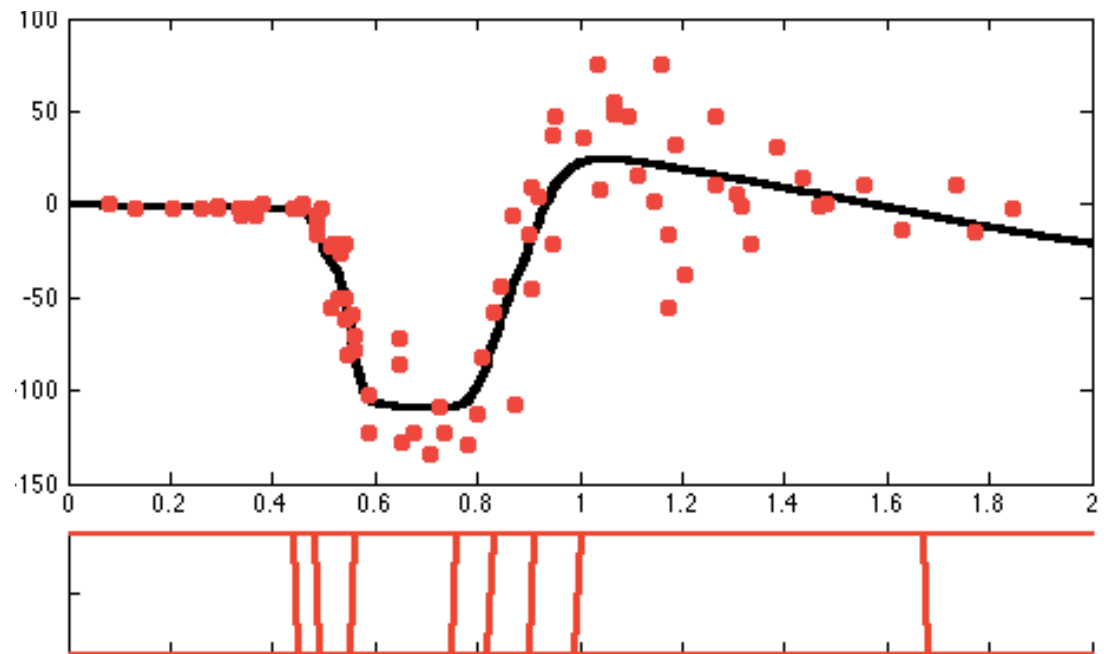
“step” updates the parameters associated with this optimizer

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

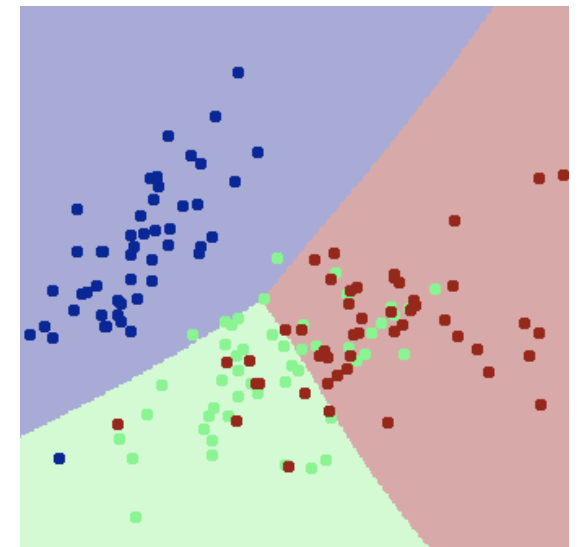
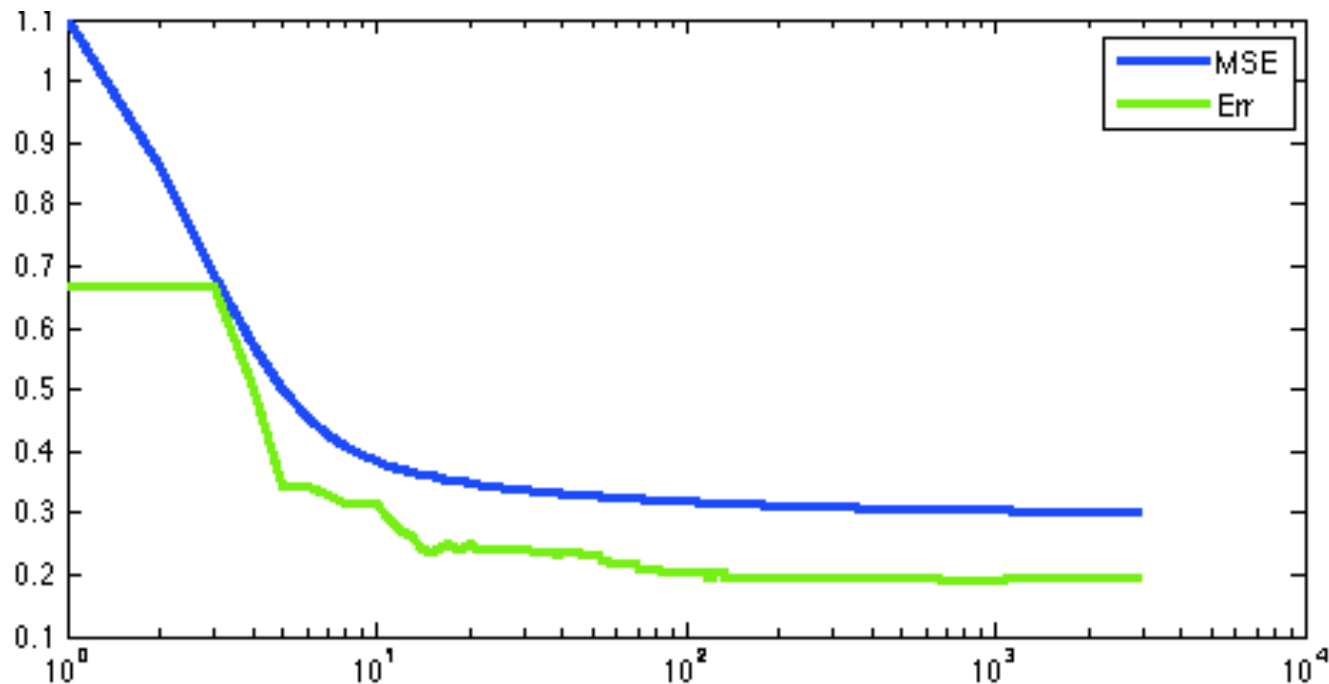
Data:
+
learned prediction f' n:

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

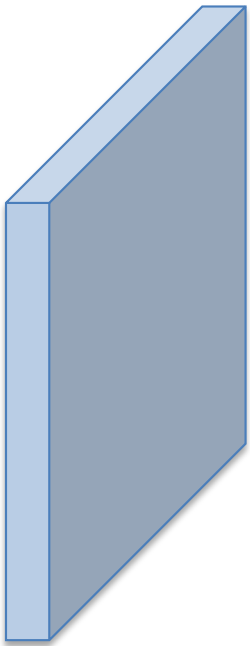
Training Deep Networks

More Tricks: Dropout, BatchNorm

Convolutional networks

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

Input: 28x28 image

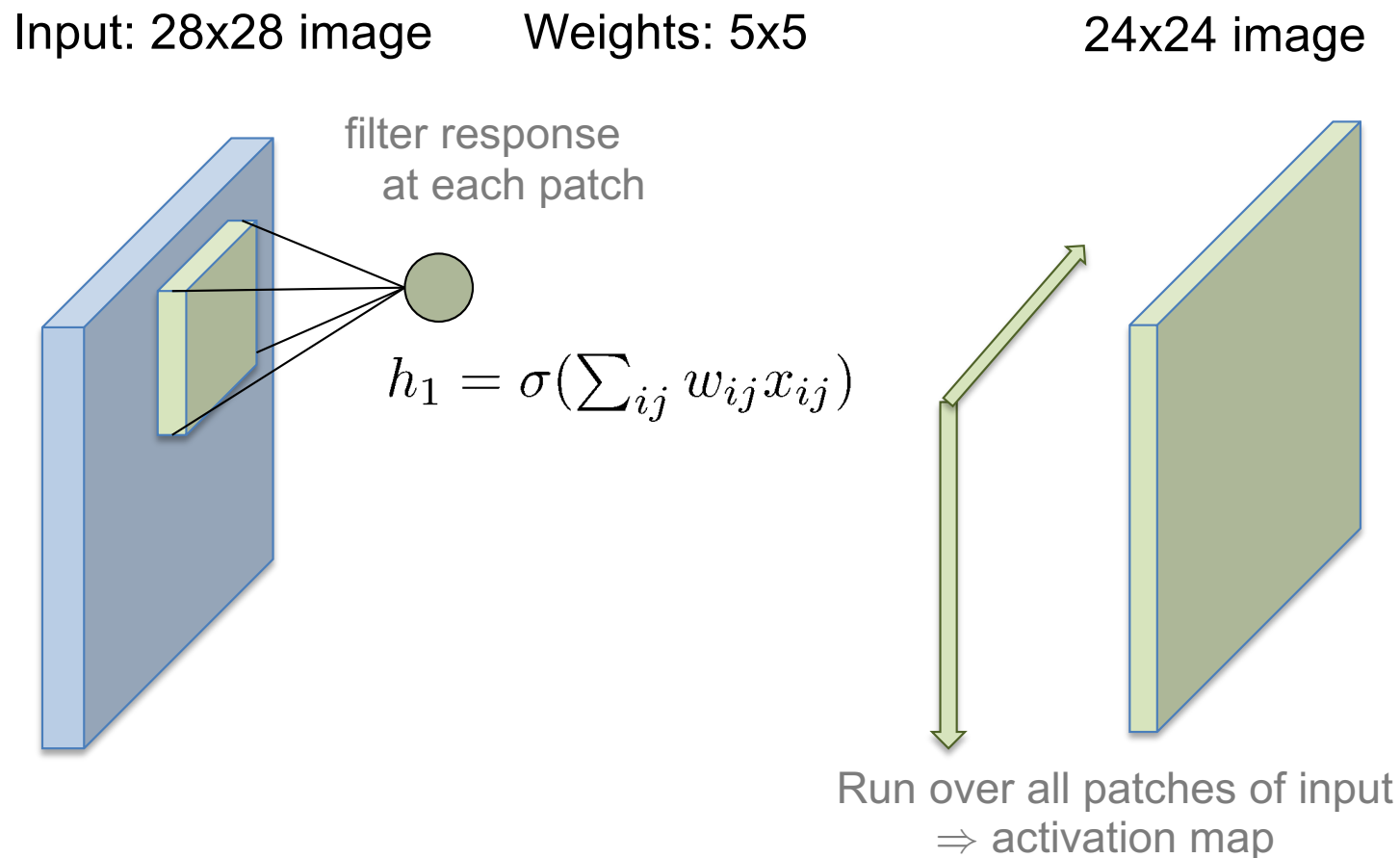


Weights: 5x5



Convolutional networks

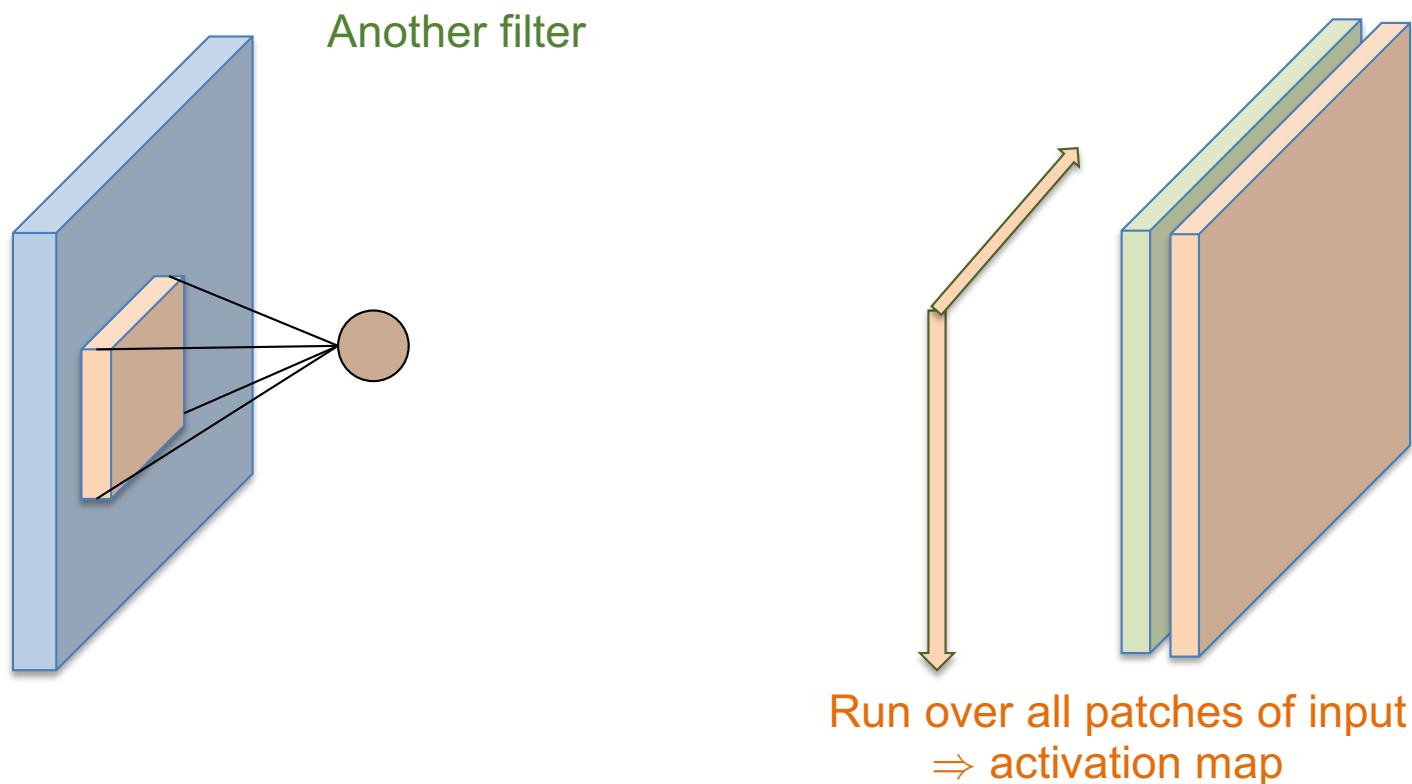
- Organize & share the NN's weights (vs “dense”)
- Group weights into “filters” & convolve across input image



Convolutional networks

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

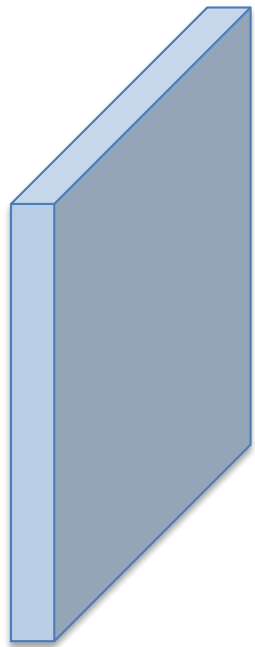
Input: 28x28 image Weights: 5x5



Convolutional networks

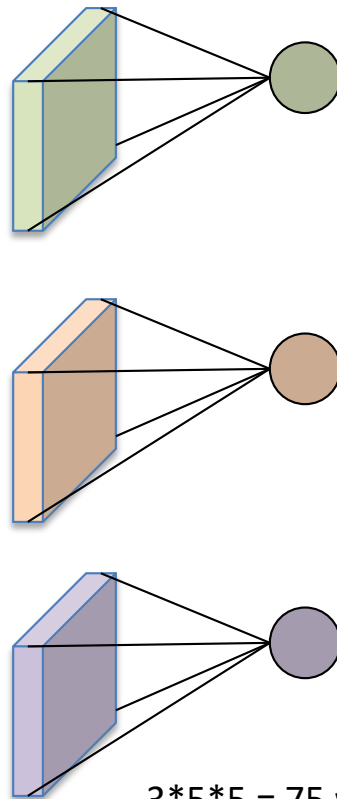
- Organize & share the NN's weights (vs “dense”)
- Group weights into “filters” & convolve across input image
- Many hidden nodes, but few parameters!

Input: 28x28 image

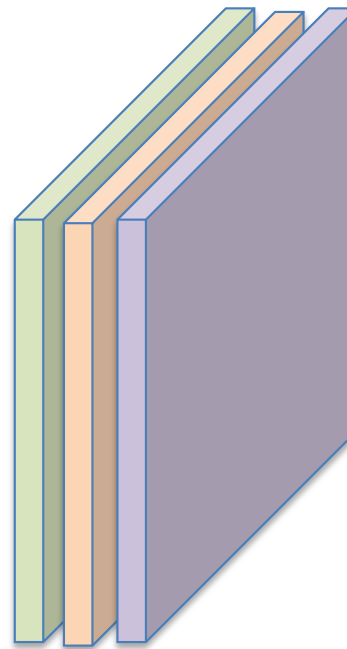


$28*28*1 = 784$ input pixels

Weights: 5x5



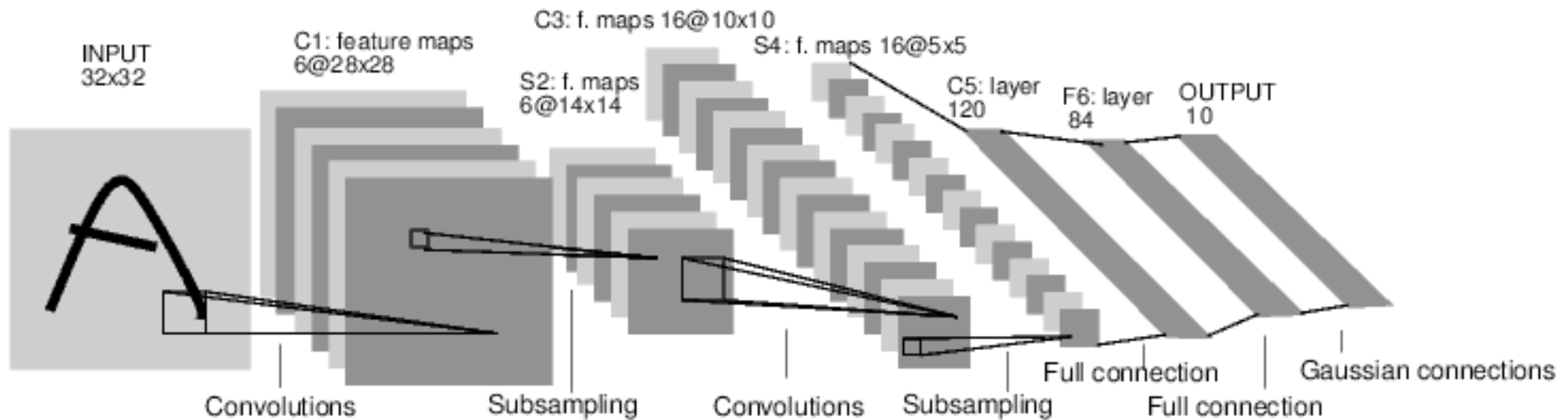
Hidden layer 1



$24*24*3 = 1728$ hidden “nodes”
(3 “channels”)

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



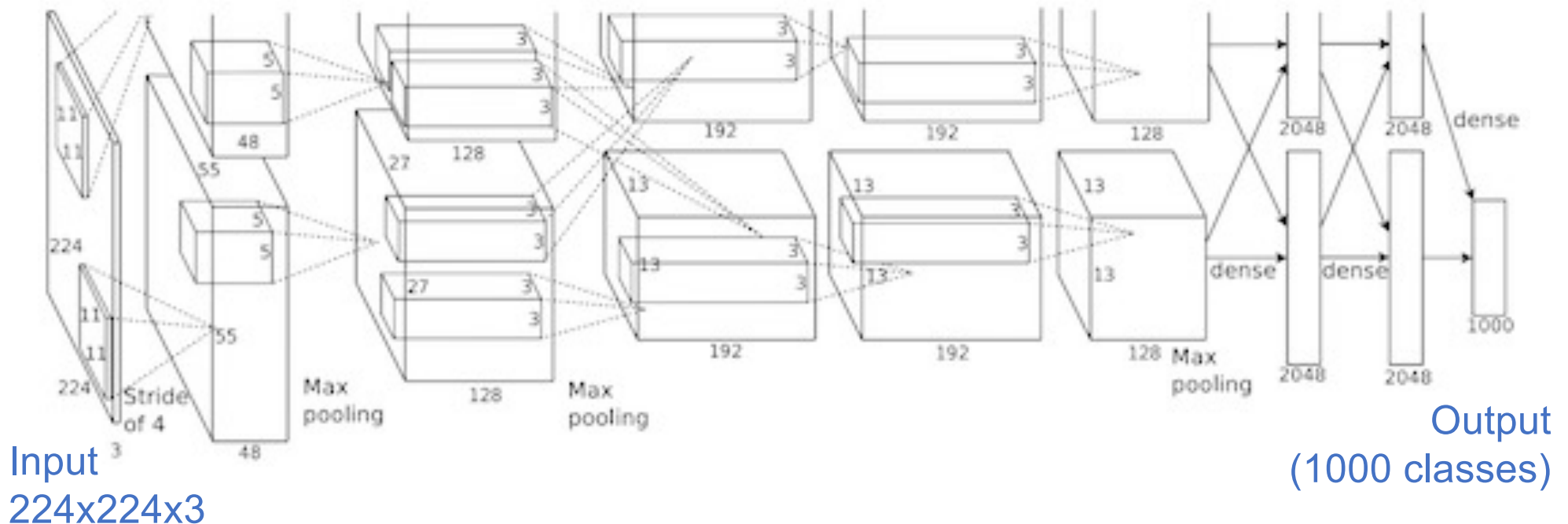
LeNet-5 [LeCun 1980]

Ex: AlexNet

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

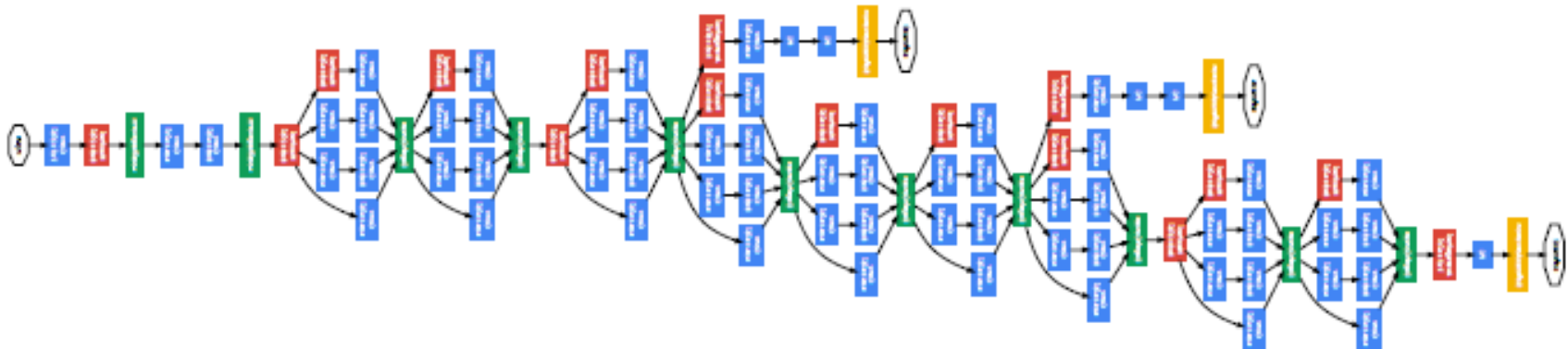
Convolutional Layers (5)

Dense Layers (3)



Ex: GoogLeNet

- Image recognition model
 - 27 layers, millions of parameters



Neural Networks

Multi-Layer Perceptrons

Backpropagation Learning

Architectures

Convolutional

Residual

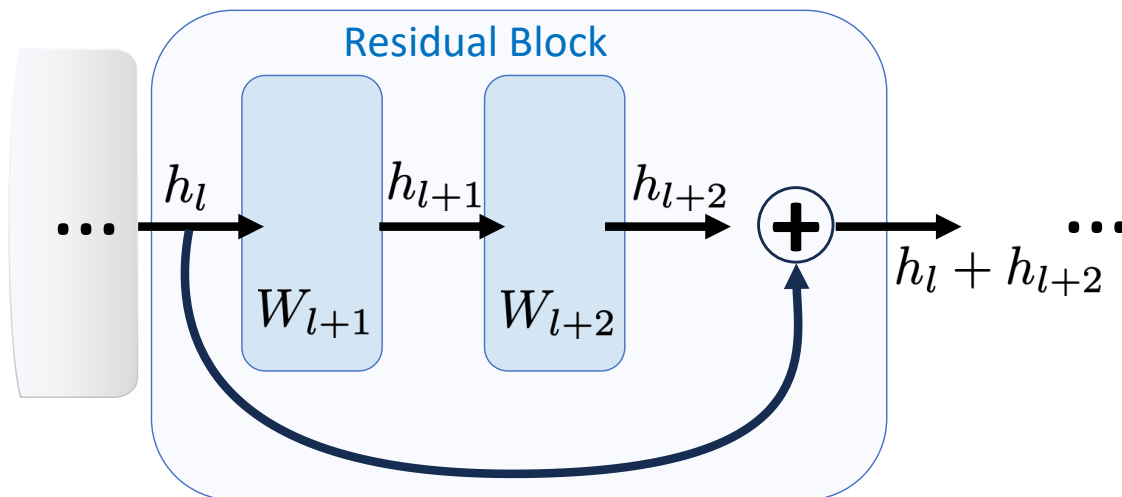
Attention

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.)

U-Networks

- Similar idea to residual blocks, but for multiple scales
 - Ex: Image-to-image (segmentation, etc.)

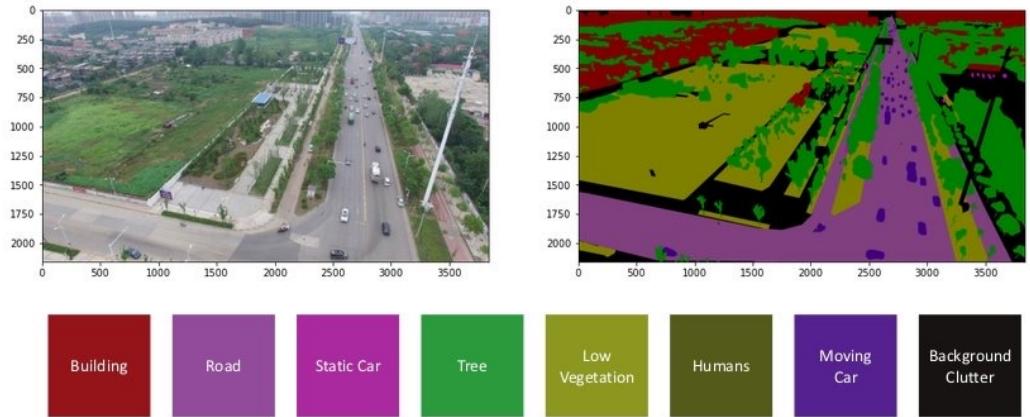


Image: <https://viso.ai/deep-learning/image-segmentation-using-deep-learning/>

- Adds skip connections at same scale:
 - Each layer uses local features at the same scale, along with aggregating features from higher scales

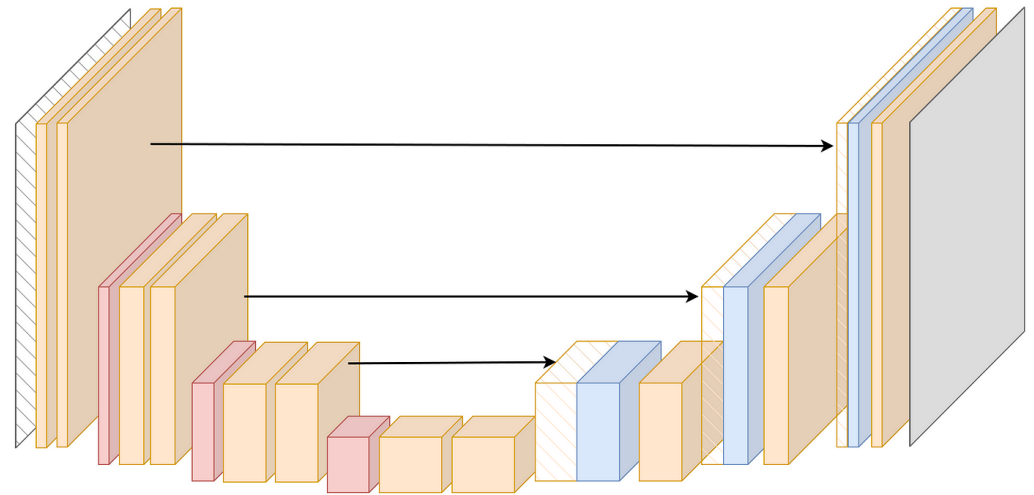


Image: <https://towardsdatascience.com/u-net-explained-understanding-its-image-segmentation-architecture-56e4842e313a>

Neural Networks

Multi-Layer Perceptrons

Backpropagation Learning

Architectures

Convolutional

Residual

Attention

Training Deep Networks

More Tricks: Dropout, BatchNorm

Attention Block

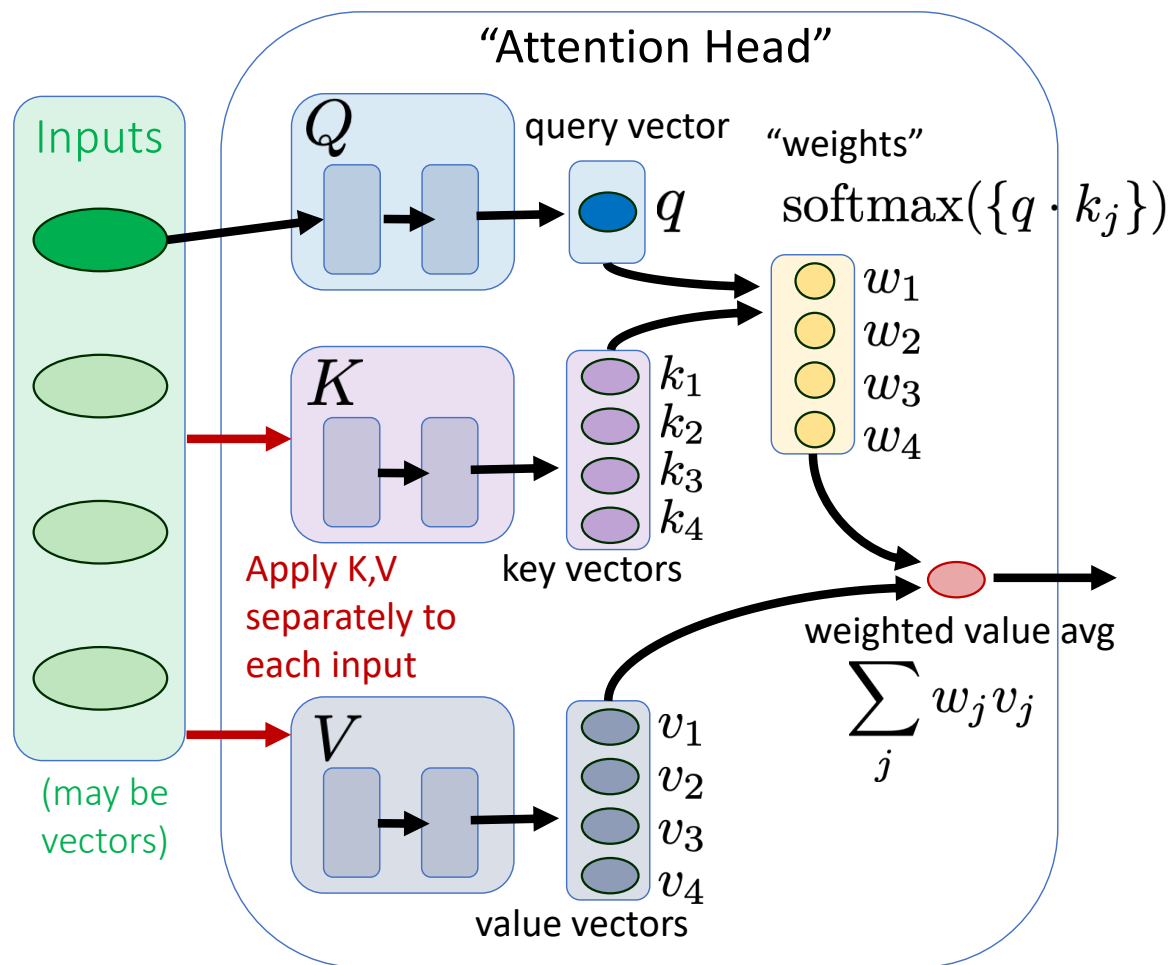
- Extract representation from many (mostly irrelevant?) inputs
 - Ex: predict next word from past sequence of observed words

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.



Neural Networks

Multi-Layer Perceptrons

Backpropagation Learning

Architectures

Convolutional

Residual

Attention

Training Deep Networks

More Tricks: Dropout, BatchNorm

Size of a deep network

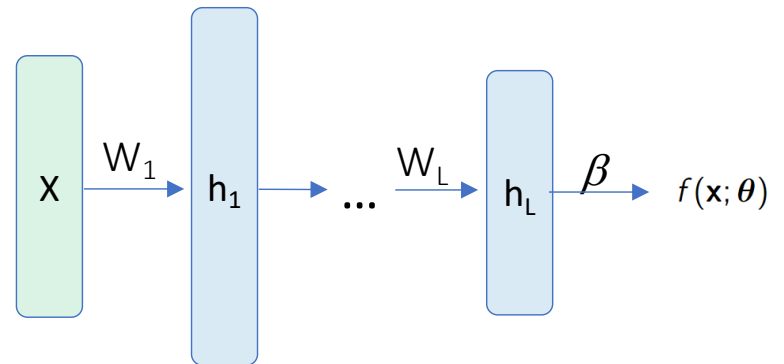
- Model size grows quickly for deep networks:

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: $n H + (L-1)H^2 + H K$

e.g., $d = 100 \times 100 = 10^4$ pixels, $H = 300$, $K = 1000$, $L = 10$ layers

=> Number of parameters p would be

about $300 \cdot 10^4 + 9 \cdot (300)^2 + 300 \cdot 1000$, 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

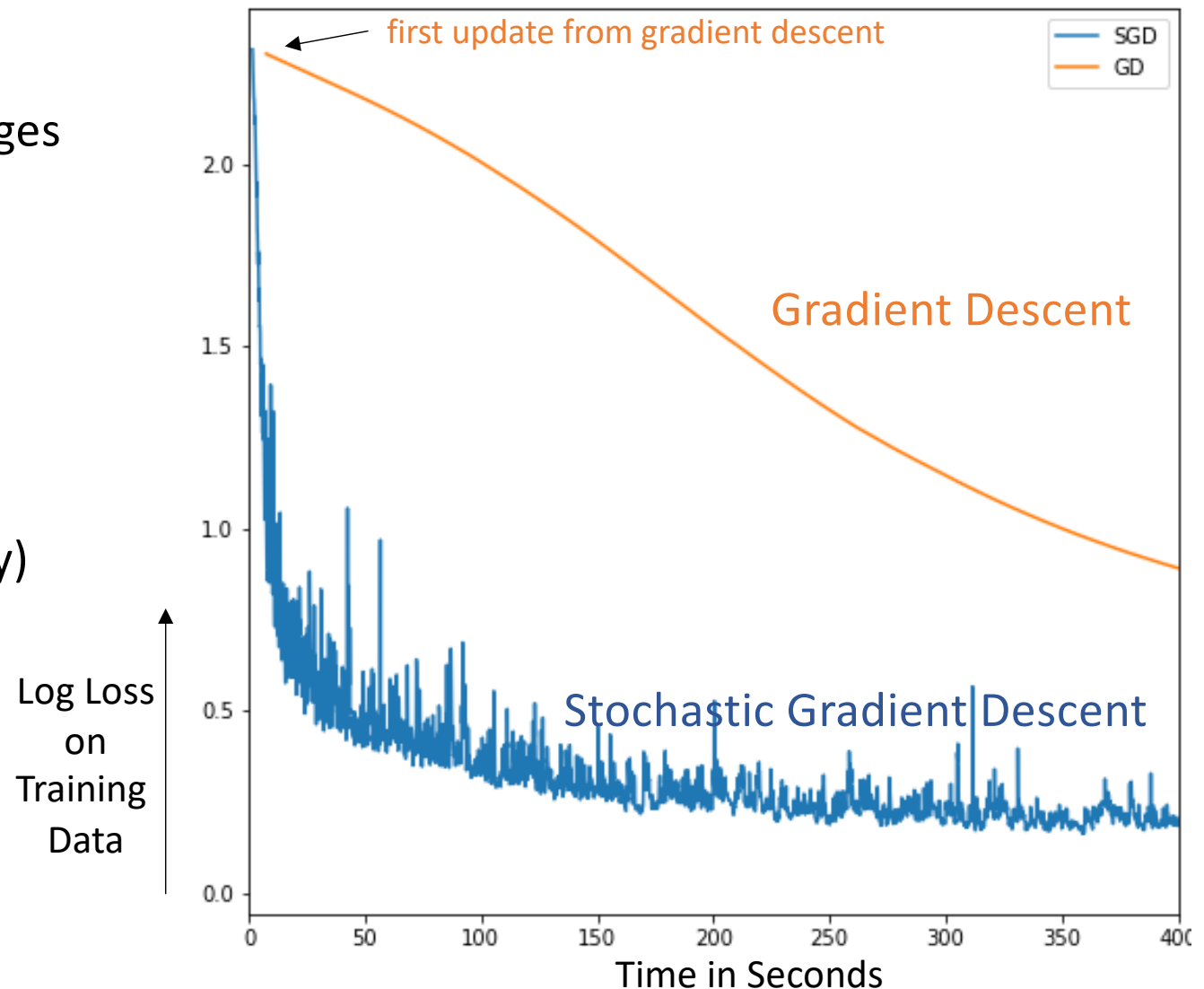
Neural network with

$n = 784$ 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

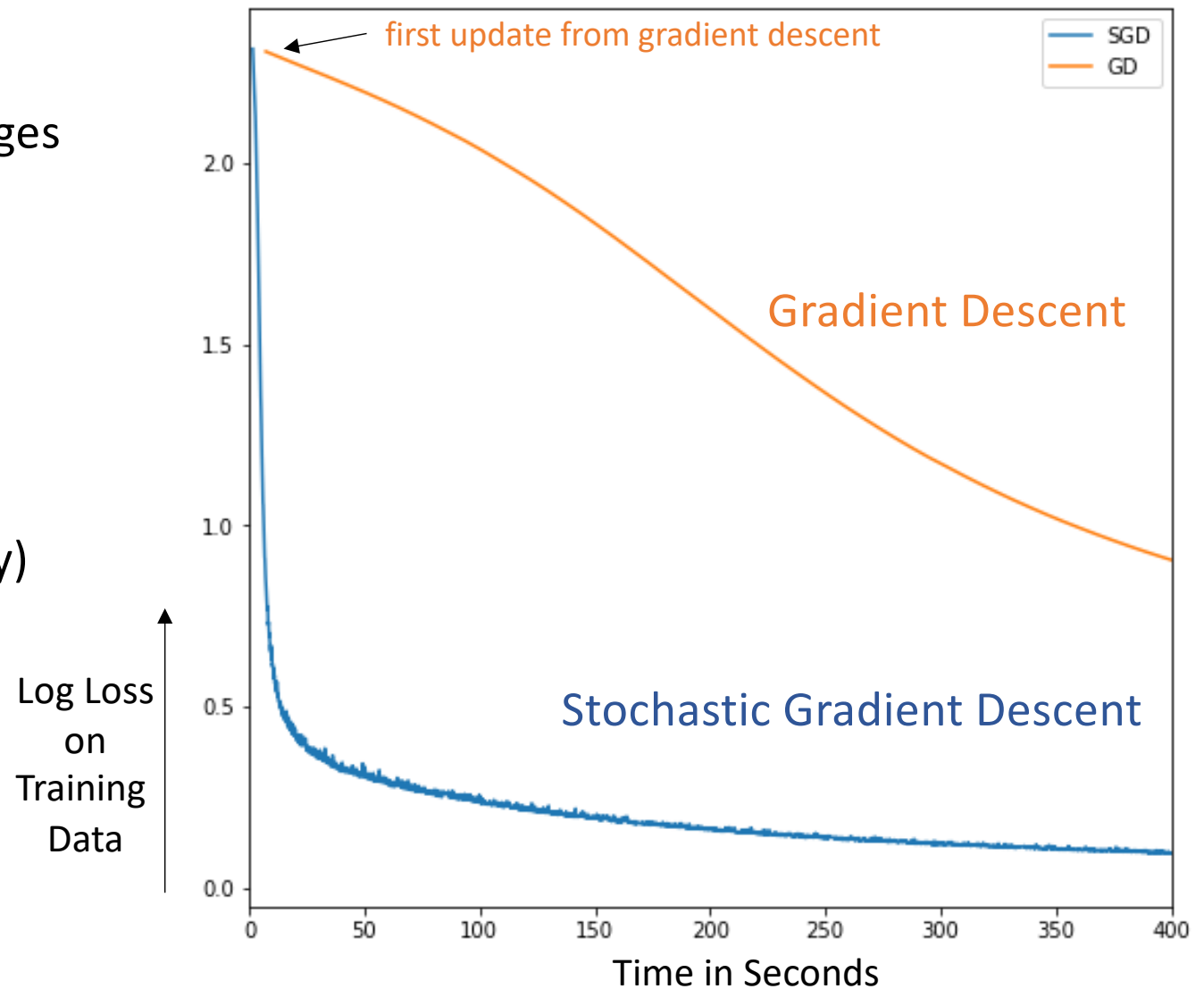
Neural network with

$n = 784$ inputs

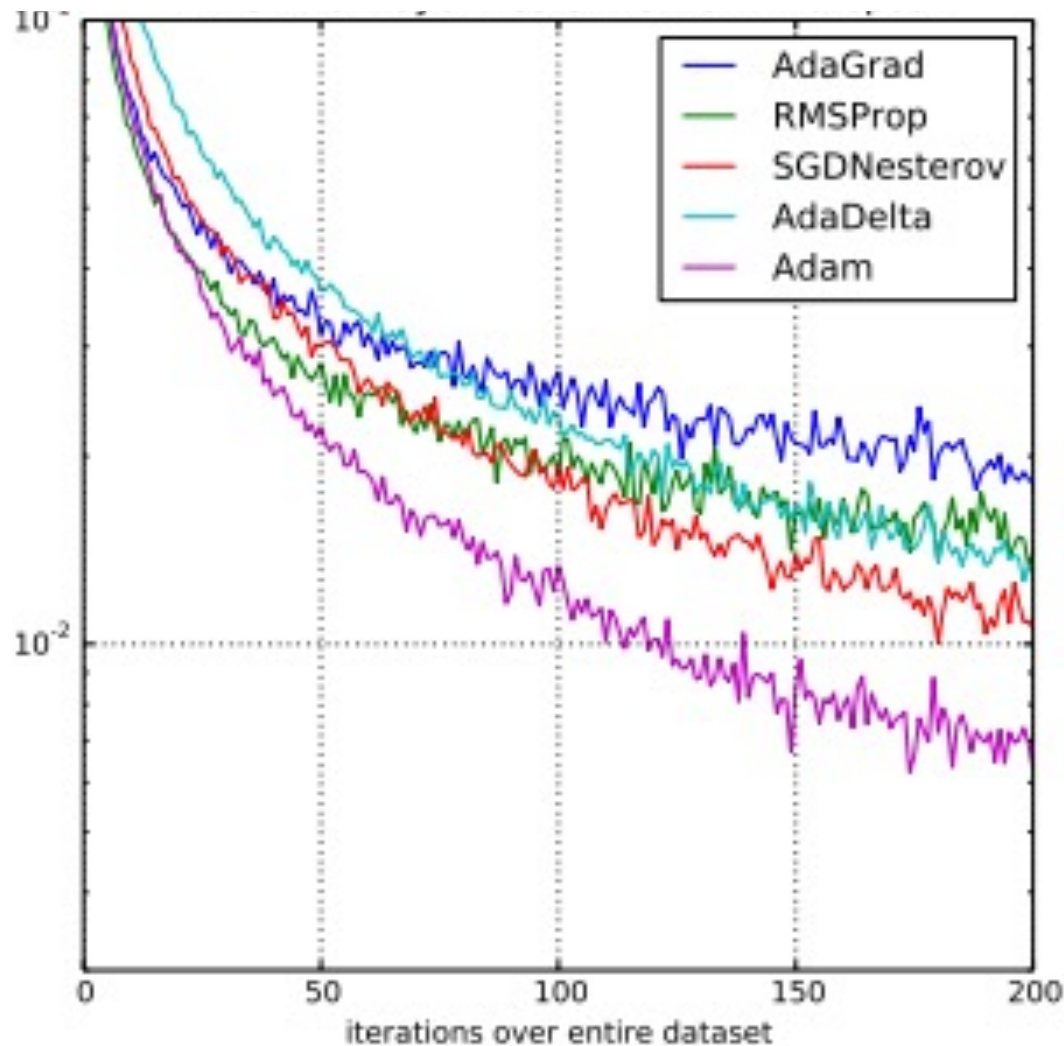
$H = 256$ hidden units

$K = 10$ 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

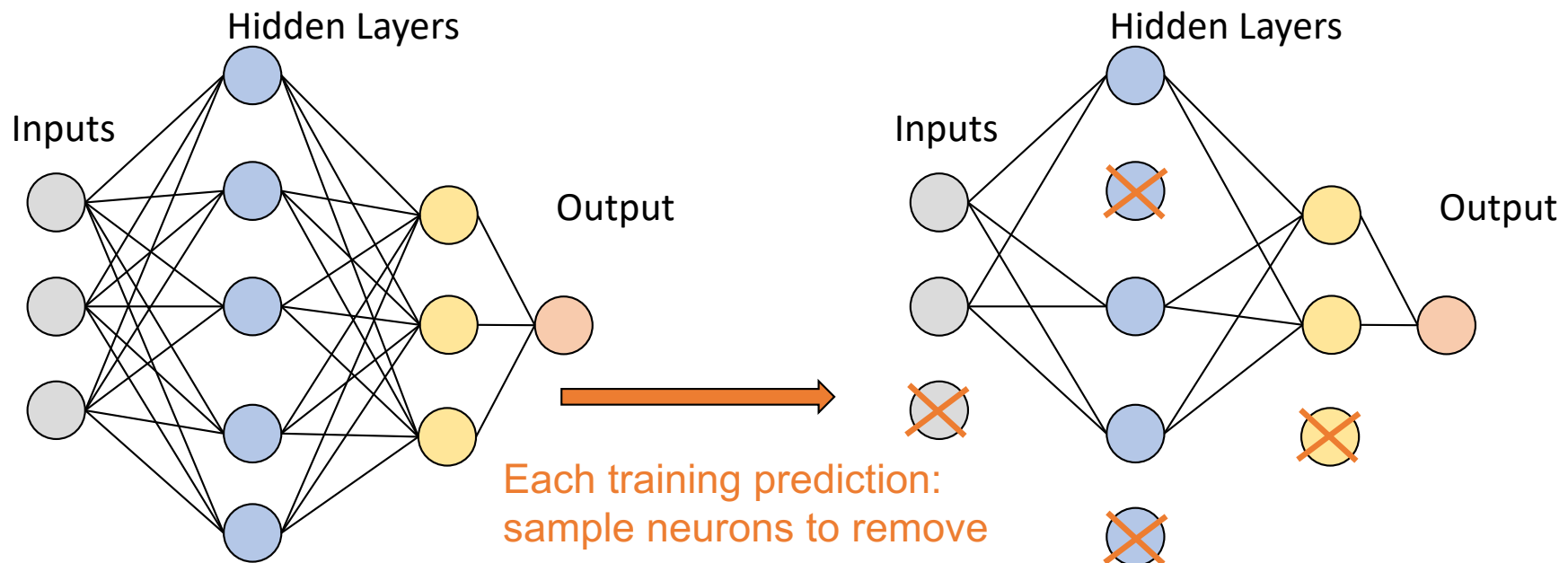
Attention

Training Deep Networks

More Tricks: Dropout, BatchNorm

Dropout

- 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

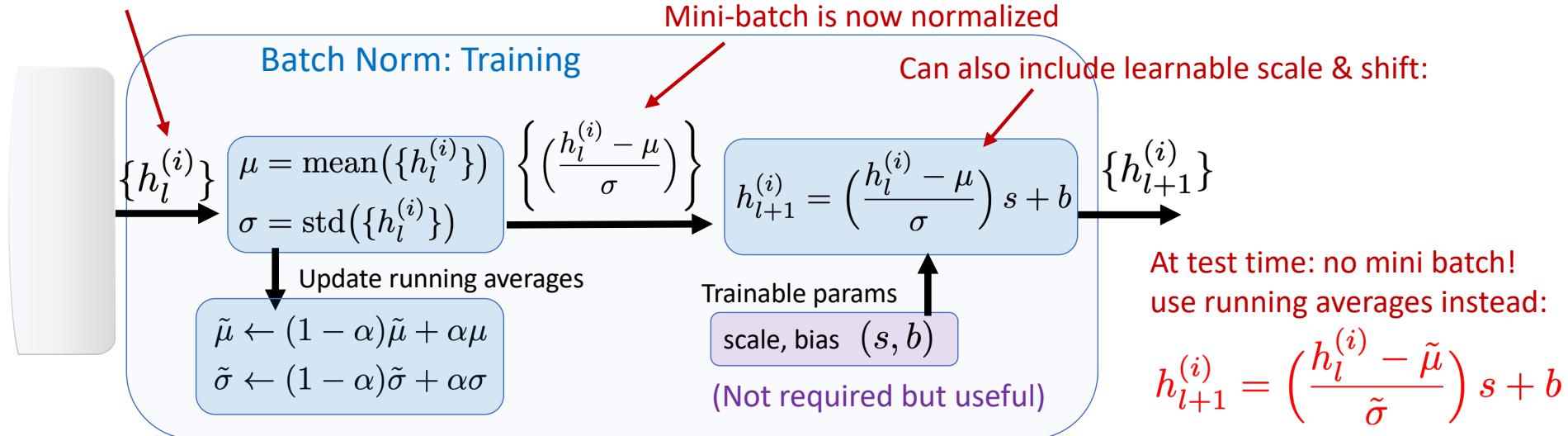
Batch Normalization

- 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; $\text{out} = (\text{in} - m)/v$
 - Instead of “training” these parameters, just re-estimate them for each batch of data!
 - Then, estimate and “lock” their values before test time.

Mini-batch of data

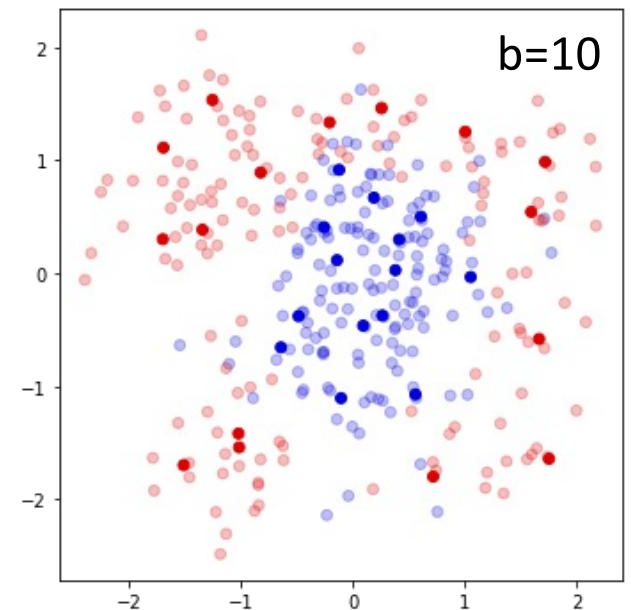
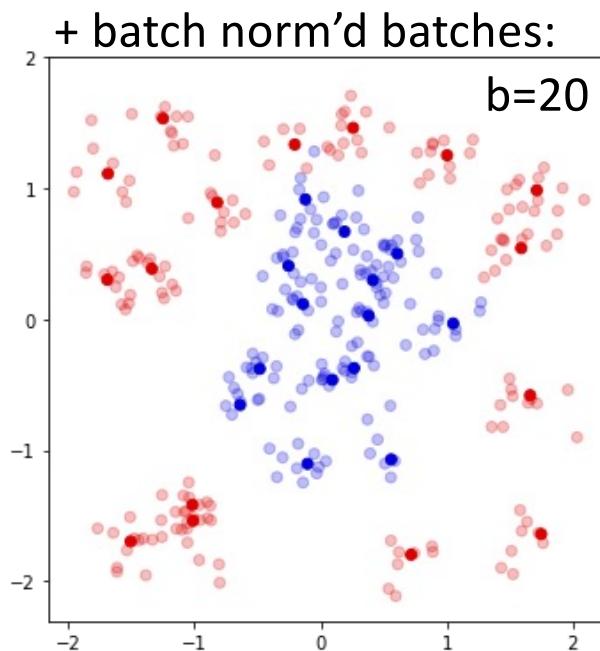
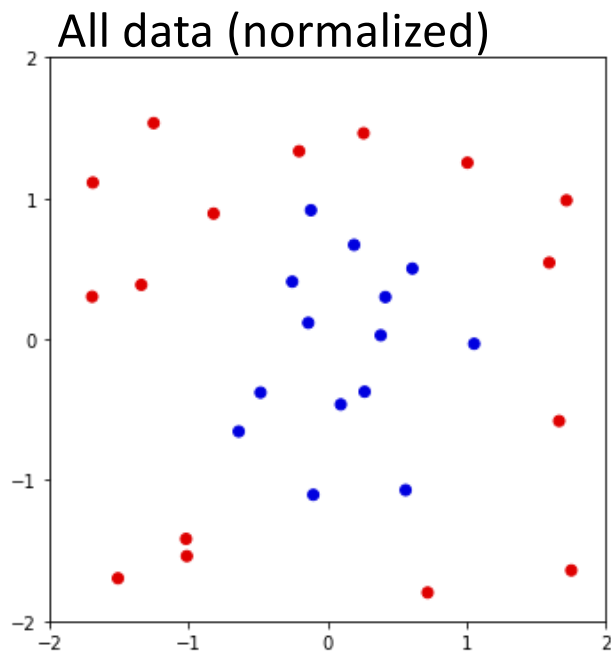
Mini-batch is now normalized

Can also include learnable scale & shift:



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



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, ...