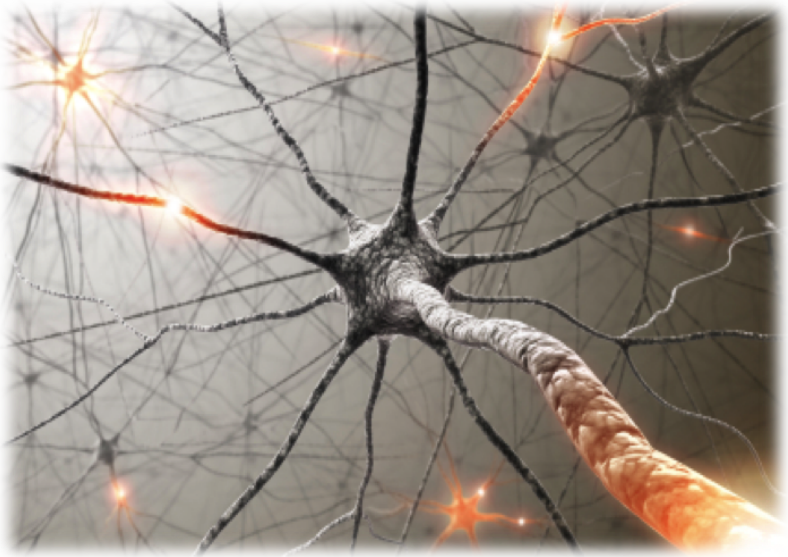
Machine learning: neural networks

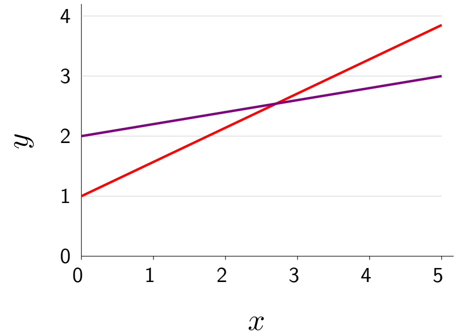


- In this module, I will present neural networks, a way to construct non-linear predictors via problem decomposition.

Non-linear predictors

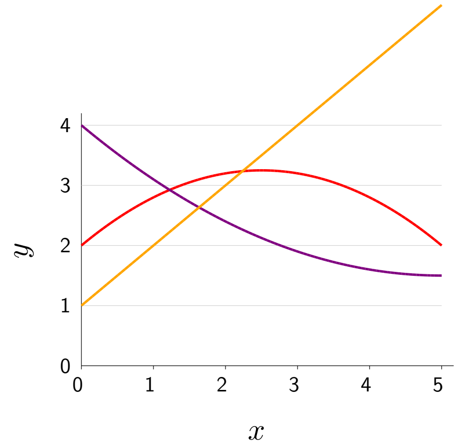
Linear predictors:

$$f_{\mathbf{w}}(x) = \mathbf{w} \cdot \phi(x), \quad \phi(x) = [1, x]$$



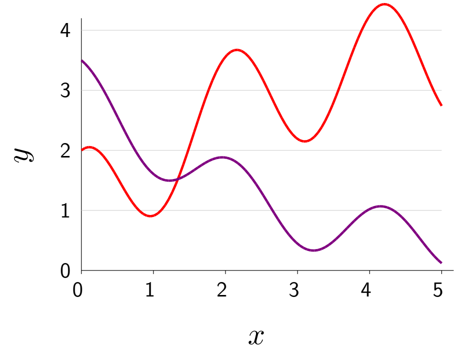
Non-linear (quadratic) predictors:

$$f_{\mathbf{w}}(x) = \mathbf{w} \cdot \phi(x), \quad \phi(x) = [1, x, x^2]$$



Non-linear neural networks:

$$f_{\mathbf{w}}(x) = \mathbf{w} \cdot \sigma(\mathbf{V}\phi(x)), \quad \phi(x) = [1, x]$$



- Recall that our first hypothesis class was linear (in x) predictors, which for regression means that the predictors are lines.
- However, we also showed that you could get non-linear (in x) predictors by simply changing the feature extractor ϕ. For example, by adding the feature x^2, one obtains quadratic predictors.
- One disadvantage of this approach is that if x were d-dimensional, one would need $O(d^2)$ features and corresponding weights, which presents considerable computational and statistical challenges.
- We will show that with neural networks, we can leave the feature extractor alone, but increase the complexity of predictor, which can also produce non-linear (though not necessarily quadratic) predictors.
- It is a common misconception that neural networks allow you to express more complex predictors. You can define ϕ to include essentially all predictors (as is done in kernel methods).
- Rather, neural networks yield non-linear predictors in a more **compact** way. For instance, you might not need $O(d^2)$ features to represent the desired non-linear predictor.

Motivating example

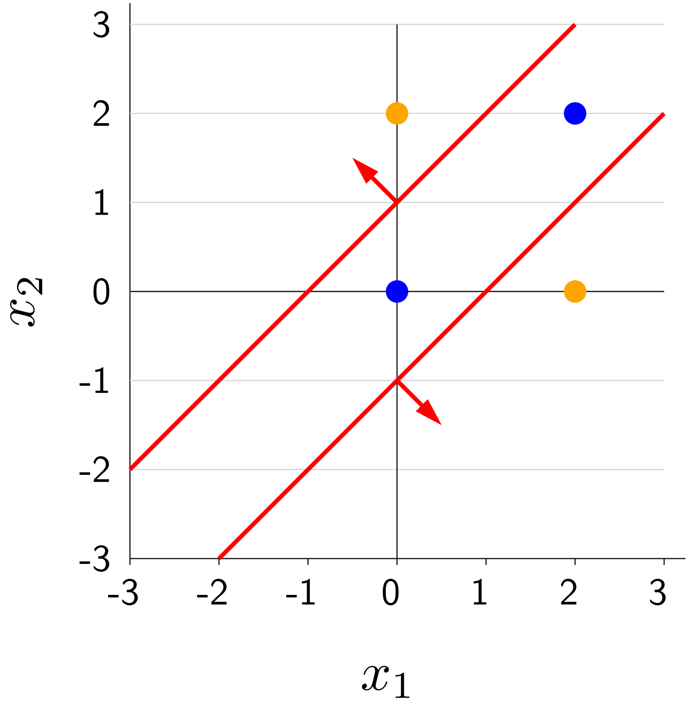
Example: predicting car collision

Input: positions of two oncoming cars $x = [x_1, x_2]$

Output: whether safe $(y = +1)$ or collide $(y = -1)$

Unknown: safe if cars sufficiently far: $y = \text{sign}(|x_1 - x_2| - 1)$

x_1	x_2	y
0	2	1
2	0	1
0	0	-1
2	2	-1



- As a motivating example, consider the problem of predicting whether two cars are going to collide given the their positions (as measured from distance from one side of the road). In particular, let x_1 be the position of one car and x_2 be the position of the other car.
- Suppose the true output is 1 (safe) whenever the cars are separated by a distance of at least 1. This relationship can be represented by the decision boundary which labels all points in the interior region between the two red lines as negative, and everything on the exterior (on either side) as positive. Of course, this true input-output relationship is unknown to the learning algorithm, which only sees training data. Consider a simple training dataset consisting of four points. (This is essentially the famous XOR problem that was impossible to fit using linear classifiers.)

Decomposing the problem

Test if car 1 is far right of car 2:

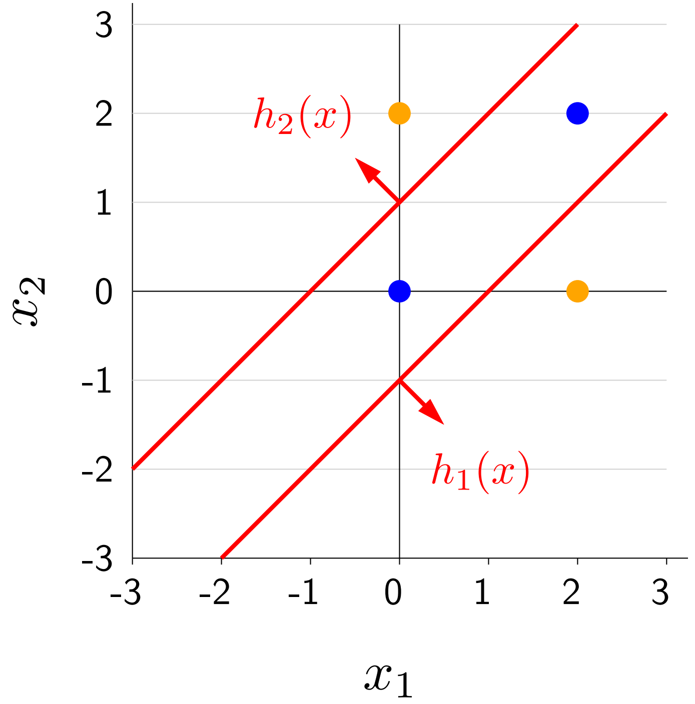
$$h_1(x) = \mathbf{1}[x_1 - x_2 \geq 1]$$

Test if car 2 is far right of car 1:

$$h_2(x) = \mathbf{1}[x_2 - x_1 \geq 1]$$

Safe if at least one is true:

$$f(x) = \textsf{sign}(h_1(x) + h_2(x))$$



x	$h_1(x)$	$h_2(x)$	$f(x)$
$[0,2]$	0	1	$+1$
$[2,0]$	1	0	$+1$
$[0,0]$	0	0	-1
$[2,2]$	0	0	-1

- One way to motivate neural networks (without appealing to the brain) is **problem decomposition**.
- The intuition is to break up the full problem into two subproblems: the first subproblem tests if car 1 is to the far right of car 2; the second subproblem tests if car 2 is to the far right of car 1. Then the final output is 1 iff at least one of the two subproblems returns 1.
- Concretely, we can define $h_1(x)$ to be the output of the first subproblem, which is a simple linear decision boundary (in fact, the right line in the figure).
- Analogously, we define $h_2(x)$ to be the output of the second subproblem.
- Note that $h_1(x)$ and $h_2(x)$ take on values 0 or 1 instead of -1 or $+1$.
- The points can then be classified by first computing $h_1(x)$ and $h_2(x)$, and then combining the results into $f(x)$.

Rewriting using vector notation

Intermediate subproblems:

$$h_1(x) = \mathbf{1}[x_1 - x_2 \geq 1] \;= \mathbf{1}[[-1, +1, -1] \cdot [1, x_1, x_2] \geq 0]$$

$$h_2(x) = \mathbf{1}[x_2 - x_1 \geq 1] \;= \mathbf{1}[[-1, -1, +1] \cdot [1, x_1, x_2] \geq 0]$$

$$\mathbf{h}(x) = \mathbf{1}\left[\begin{bmatrix} -1 & +1 & -1 \\ -1 & -1 & +1 \end{bmatrix} \begin{bmatrix} 1 \\ x_1 \\ x_2 \end{bmatrix} \geq 0\right]$$

Predictor:

$$f(x) = \mathsf{sign}(h_1(x) + h_2(x)) \;= \mathsf{sign}([1, 1] \cdot \mathbf{h}(x))$$

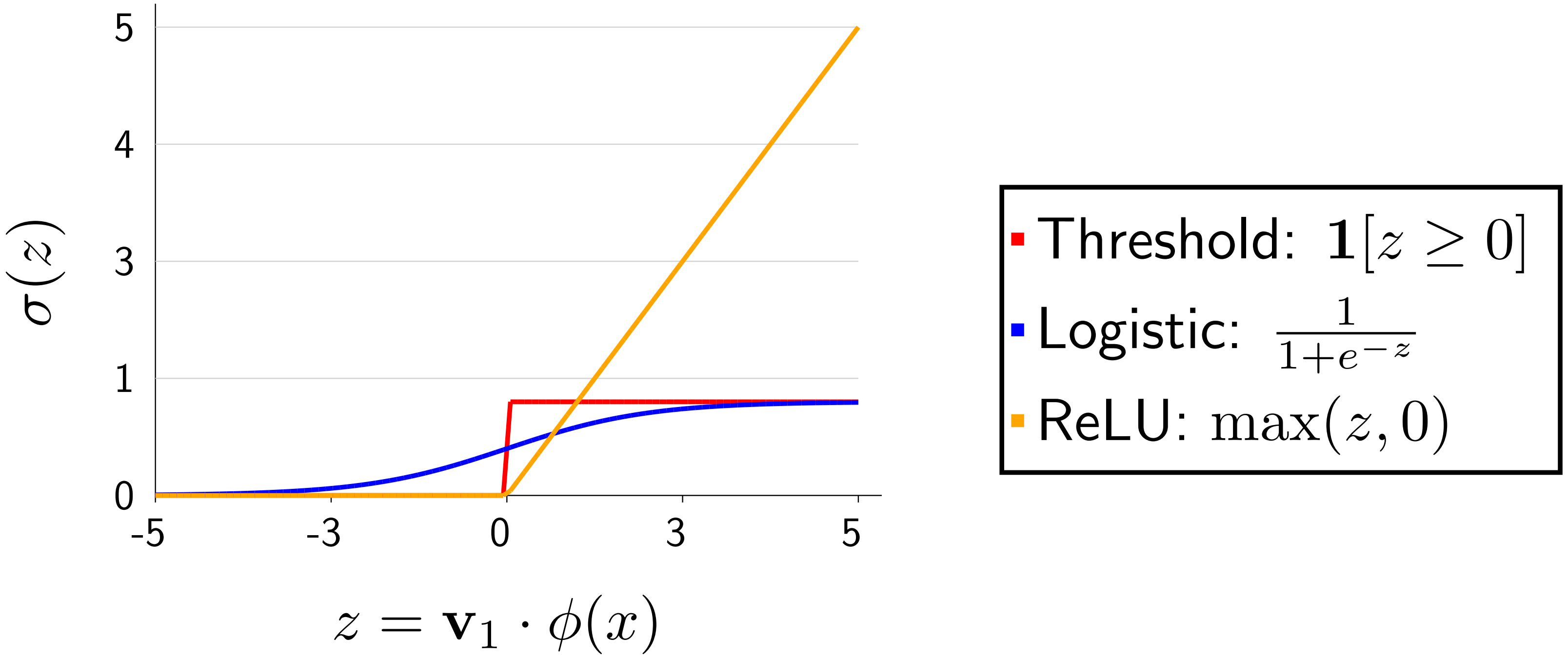
- Now let us rewrite this predictor $f(x)$ using vector notation.
- We can define a feature vector $[1, x_1, x_2]$ and a corresponding weight vector, where the dot product thresholded yields exactly $h_1(x)$.
- We do the same for $h_2(x)$.
- We put the two subproblems into one equation by stacking the weight vectors into one matrix. Recall that left-multiplication by a matrix is equivalent to taking the dot product with each row. By convention, the thresholding at 0 ($\mathbf{1}[\cdot \geq 0]$) applies component-wise.
- Finally, we can define the predictor in terms of a simple dot product.
- Now of course, we don't know the weight vectors, but we can learn them from the training data!

Avoid zero gradients

Problem: gradient of $h_1(x)$ with respect to $\mathbf{v}_1$ is 0

$$h_1(x) = \mathbf{1}[\mathbf{v}_1 \cdot \phi(x) \geq 0]$$

Solution: replace with an **activation function** σ with non-zero gradients

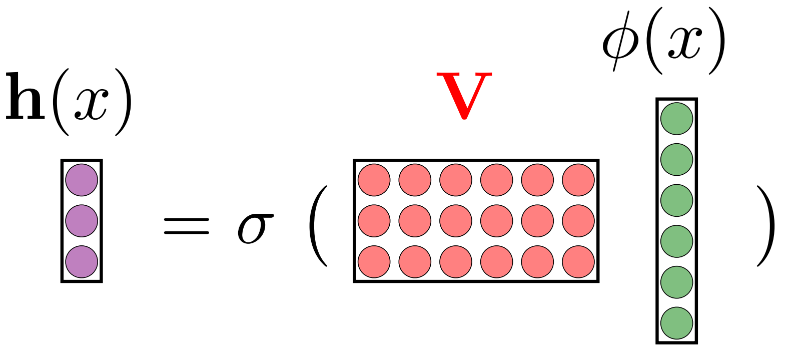


$$h_1(x) = \sigma(\mathbf{v}_1 \cdot \phi(x))$$

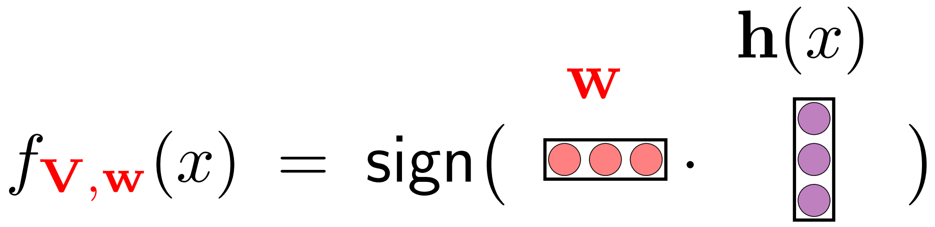
- Later we'll show how to perform learning using gradient descent, but we can anticipate one problem, which we encountered when we tried to optimize the zero-one loss.
- The gradient of $h_1(x)$ with respect to $\mathbf{v}_1$ is always zero because of the threshold function.
- To fix this, we replace the threshold function with an **activation function** with non-zero gradients
- Classically, neural networks used the **logistic function** $\sigma(z)$, which looks roughly like the threshold function but has non-zero gradients everywhere.
- Even though the gradients are non-zero, they can be quite small when $|z|$ is large (a phenomenon known as saturation). This makes optimizing with the logistic function still difficult.
- In 2012, Glorot et al. introduced the ReLU activation function, which is simply $\max(z, 0)$. This has the advantage that at least on the positive side, the gradient does not vanish (though on the negative side, the gradient is always zero). As a bonus, ReLU is easier to compute (only max, no exponentiation). In practice, ReLU works well and has become the activation function of choice.
- Note that if the activation function were linear (e.g., the identity function), then the gradients would always be nonzero, but you would lose the power of a neural network, because you would simply get the product of the final-layer weight vector and the weight matrix ($\mathbf{w}^\top \mathbf{V}$), which is equivalent to optimizing over a single weight vector.
- Therefore, that there is a tension between wanting an activation function that is non-linear but also has non-zero gradients.

Two-layer neural networks

Intermediate subproblems:



Predictor (classification):



Interpret $\mathbf{h}(x)$ as a learned feature representation!

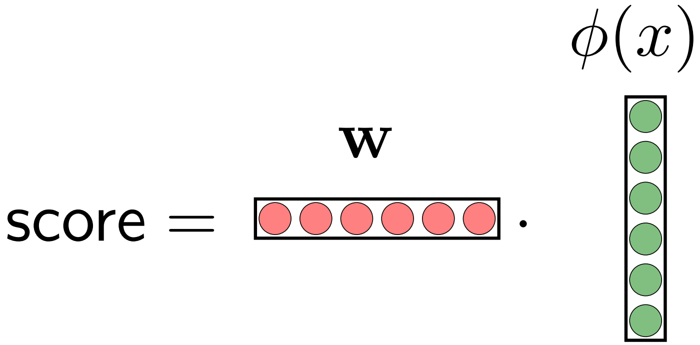
Hypothesis class:

$$\mathcal{F} = \{f_{\mathbf{V},\mathbf{w}} : \mathbf{V} \in \mathbb{R}^{k \times d}, \mathbf{w} \in \mathbb{R}^{k}\}$$

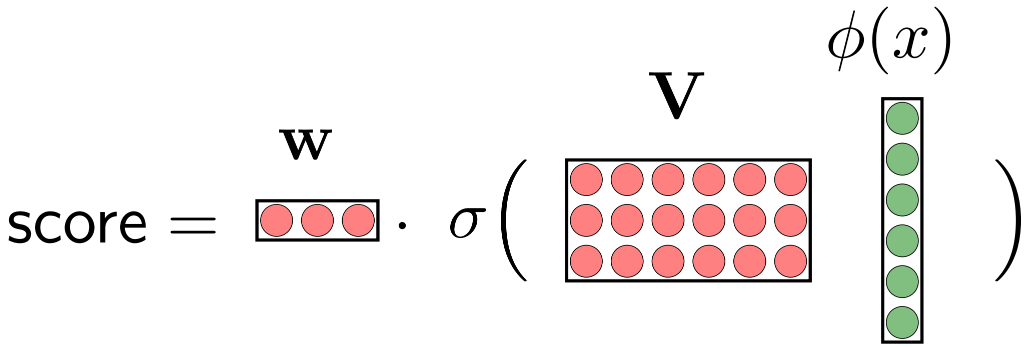
- Now we are finally ready to define the hypothesis class of two-layer neural networks.
- We start with a feature vector $\phi(x)$.
- We multiply it by a weight matrix $\mathbf{V}$ (whose rows can be interpreted as the weight vectors of the k intermediate subproblems.
- Then we apply the activation function σ to each of the k components to get the hidden representation $\mathbf{h}(x) \in \mathbb{R}^k$.
- We can actually interpret $\mathbf{h}(x)$ as a learned feature vector (representation), which is derived from the original non-linear feature vector $\phi(x)$.
- Given $\mathbf{h}(x)$, we take the dot product with a weight vector $\mathbf{w}$ to get the score used to drive either regression or classification.
- The hypothesis class is the set of all such predictors obtained by varying the first-layer weight matrix $\mathbf{V}$ and the second-layer weight vector $\mathbf{w}$.

Deep neural networks

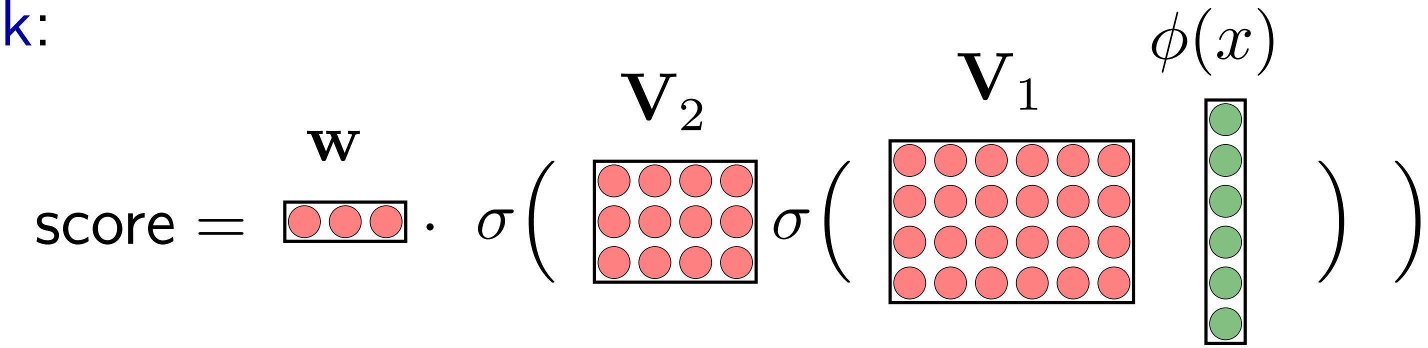
1-layer neural network:



2-layer neural network:



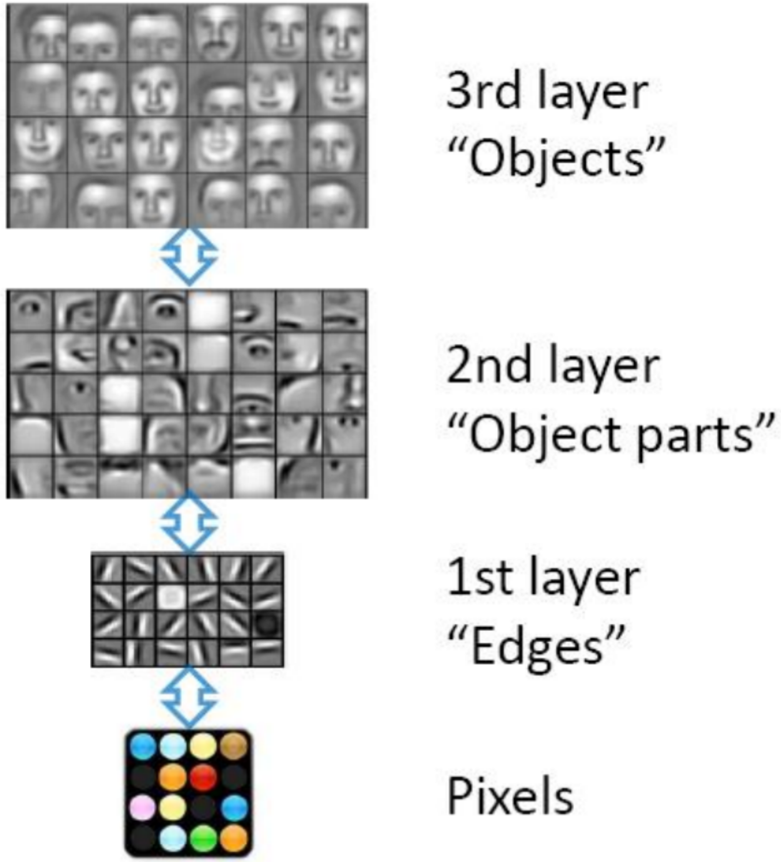
3-layer neural network:



- We can push these ideas to build deep neural networks, which are neural networks with many layers.
- Warm up: for a one-layer neural network (a.k.a. a linear predictor), the score that drives prediction is simply a dot product between a weight vector and a feature vector.
- We just saw for a two-layer neural network, we apply a linear layer $\mathbf{V}$ first, followed by a non-linearity σ, and then take the dot product.
- To obtain a three-layer neural network, we apply a linear layer and a non-linearity (this is the basic building block). This can be iterated any number of times. No matter now deep the neural network is, the top layer is always a linear function, and all the layers below that can be interpreted as defining a (possibly very complex) hidden feature vector.
- In practice, you would also have a bias term (e.g., $\mathbf{V}\phi(x) + b$). We have omitted all bias terms for notational simplicity.

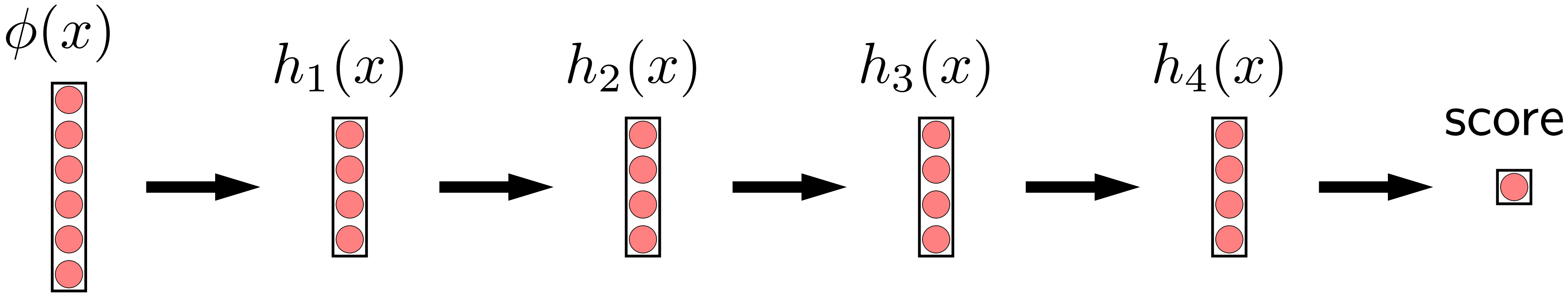
[figure from Honglak Lee]

Layers represent multiple levels of abstractions



- It can be difficult to understand what a sequence of (matrix multiply, non-linearity) operations buys you.
- To provide intuition, suppose the input feature vector $\phi(x)$ is a vector of all the pixels in an image.
- Then each layer can be thought of producing an increasingly abstract representation of the input. The first layer detects edges, the second detects object parts, the third detects objects. What is shown in the figure is for each component j of the hidden representation $\mathbf{h}(x)$, the input image $\phi(x)$ that maximizes the value of $h_j(x)$.
- Though we haven't talked about learning neural networks, it turns out that the "levels of abstraction" story is actually borne out visually when we learn neural networks on real data (e.g., images).

Why depth?

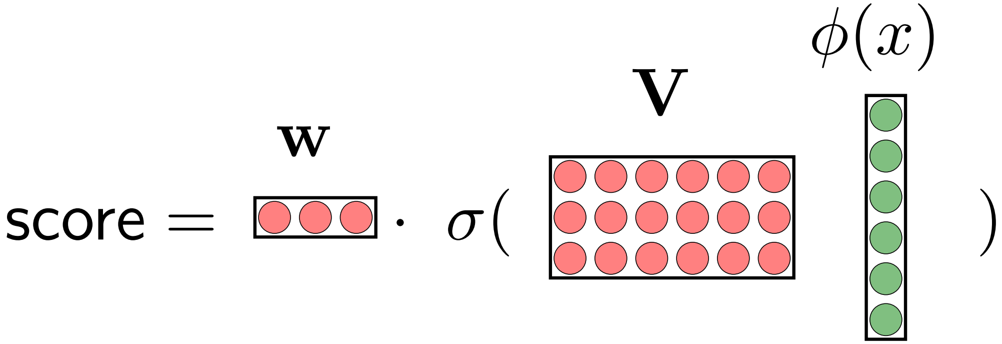


Intuitions:

- Multiple levels of abstraction
- Multiple steps of computation
- Empirically works well
- Theory is still incomplete

- Beyond learning hierarchical feature representations, deep neural networks can be interpreted in a few other ways.
- One perspective is that each layer can be thought of as performing some computation, and therefore deep neural networks can be thought of as performing multiple steps of computation.
- But ultimately, the real reason why deep neural networks are interesting is because they work well in practice.
- From a theoretical perspective, we have a quite an incomplete explanation for why depth is important. The original motivation from McCulloch/Pitts in 1943 showed that neural networks can be used to simulate a bounded computation logic circuit. Separately it has been shown that depth $k+1$ logic circuits can represent more functions than depth k. However, neural networks are real-valued and might have types of computations which don't fit neatly into logical paradigm. Obtaining a better theoretical understanding is an active area of research in statistical learning theory.

Summary



- Intuition: decompose problem into intermediate parallel subproblems

- Deep networks iterate this decomposition multiple times

- Hypothesis class contains predictors ranging over weights for all layers

- Next up: learning neural networks

- To summarize, we started with a toy problem (the XOR problem) and used it to motivate neural networks, which decompose a problem into intermediate subproblems, which are solved in parallel.
- Deep networks iterate this multiple times to build increasingly high-level representations of the input.
- Next, we will see how we can learn a neural network by choosing the weights for all the layers.