COMPSCI 1810 Spring 2026 Section 4
Richer Features and Neural Networks

Authored by Ege Cakar

1 Richer Features

1.1 Motivation

To make our models more expressive, we typically transform our input data using basis functions, e.g, given an input x, we will define $\phi(\cdot)$ as some basis function, and then construct a model using $\phi(x)$. However, explicitly specifying ϕ is only one way of enriching our feature space.

For some motivation as to why we want richer features: By now you may have realized that our models will often work with the linear combination $\phi(x)^\top \mathbf{w}$. This is linear in the weights $\mathbf{w}$, which we may think will restrict how expressive the model can be, e.g., if we are trying to model an output which is not a linear function of the data. However, by constructing richer features, we are able to overcome this problem, so that the output is linear in the transformed data, in which case we could do, e.g., linear regression on the transformed data.

For simplicity, assume that there is no bias (intercept), as it can always be added later easily. Some examples we saw in the linear regression lecture were $\phi(x) = x^2$, $\phi(x) = \sin(x)$ etc., which are both $\mathbb{R} \to \mathbb{R}$. I can also define a ϕ such that:

$$\phi(x) = \begin{bmatrix} x \\ x^2 \\ x^3 \end{bmatrix}, \text{ in which case we have } \phi : \mathbb{R} \to \mathbb{R}^3.$$

Then, we would learn a corresponding weight for each dimension of this feature. Note that we don't necessarily need to use all dimensions here – if x is not useful for prediction given x^2, for example, w_1 can be set to 0.

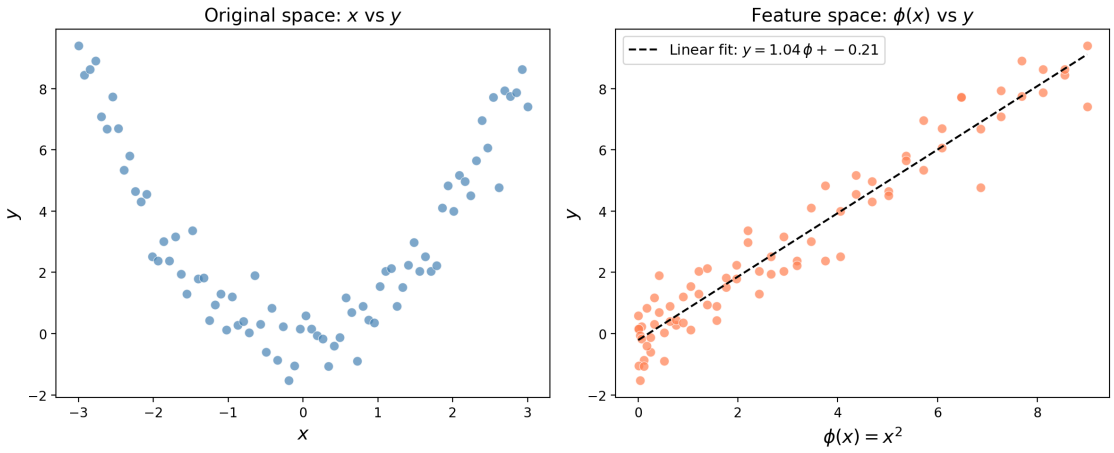


Figure 1: A demonstration of how basis functions transform space to turn our problem linear. Here, the regression problem is linear in x^2.

So using the notation of ϕ, we can represent many of the models we have seen in the form

$$y = \sigma(\phi(x)^\top \mathbf{w}),$$

where σ is task-dependent. For example, for linear regression we have previously set $\sigma(x) = x$, and for binary classification $\sigma(x) = \frac{1}{1+e^{-x}}$.

1.2 Beyond Basis Functions: From Ridge Regression in Feature Space to Kernel Methods

There are times when the features we are dealing with might be too high dimensional, or too expensive to compute and save. In those situations, we can reconfigure our problem such that we never need to instance $\phi(x)$. Below is an example of turning a ridge regression problem (parametric) into a (nonparametric) kernel problem.

Let $\phi : \mathcal{X} \to \mathbb{R}^m$ be a (possibly high-dimensional, expensive) feature map and define

$$\Phi \in \mathbb{R}^{n \times m}, \qquad \Phi_{i,:} := \phi(x_i)^\top, \qquad \mathbf{y} \in \mathbb{R}^n.$$

Ridge regression in feature space solves, as we are accustomed to

$$\min_{\mathbf{w} \in \mathbb{R}^m} \quad \|\mathbf{y} - \Phi\mathbf{w}\|_2^2 + \lambda \|\mathbf{w}\|_2^2, \qquad \lambda > 0. \tag{1}$$

First, we will show that the optimal w lies in the span of training features. Let

$$\mathcal{S} := \operatorname{span}\{\phi(x_1), \ldots, \phi(x_n)\} \subseteq \mathbb{R}^m$$

and decompose $w = w_\parallel + w_\perp$ with $w_\parallel \in \mathcal{S}$ and $w_\perp \perp \mathcal{S}$. Since each row of Φ is $\phi(x_i)^\top$, we have $\Phi w_\perp = 0$. Therefore

$$\|\mathbf{y} - \Phi\mathbf{w}\|_2^2 = \|y - \Phi\mathbf{w}_\parallel\|_2^2 \quad \text{and} \quad \|\mathbf{w}\|_2^2 = \|\mathbf{w}_\parallel\|_2^2 + \|\mathbf{w}_\perp\|_2^2.$$

For fixed $w_\parallel$, adding any $w_\perp \neq 0$ strictly increases the objective above. Hence the minimizer satisfies $w_\perp = 0$, i.e.

$$\mathbf{w}^\star \in \mathcal{S} \quad \Rightarrow \quad \exists\, \alpha \in \mathbb{R}^n \text{ such that } w^\star = \Phi^\top \alpha = \sum_{i=1}^n \alpha_i \phi(x_i).$$

That means we can rewrite our prediction using inner products. For any $x \in \mathcal{X}$,

$$f(x) := \phi(x)^\top \mathbf{w}^\star = \phi(x)^\top \Phi^\top \alpha = \sum_{i=1}^n \alpha_i \langle \phi(x), \phi(x_i) \rangle.$$

Define a kernel k by $k(x, x') := \langle \phi(x), \phi(x') \rangle$, and the Gram matrix

$$K \in \mathbb{R}^{n \times n}, \qquad K_{ij} := k(x_i, x_j) = \langle \phi(x_i), \phi(x_j) \rangle.$$

Then the predictor takes the *kernel expansion* form

$$f(x) = \sum_{i=1}^n \alpha_i \, k(x_i, x). \tag{2}$$

Now solve for α. The primal ridge solution satisfies the normal equations

$$(\Phi^\top \Phi + \lambda I)\mathbf{w}^\star = \Phi^\top \mathbf{y}.$$

Using the identity (see discussion of this in the remark below)

$$(\Phi^\top \Phi + \lambda I)^{-1}\Phi^\top = \Phi^\top (\Phi\Phi^\top + \lambda I)^{-1},$$

we obtain

$$\mathbf{w}^\star = (\Phi^\top \Phi + \lambda I)^{-1}\Phi^\top \mathbf{y} = \Phi^\top (\Phi\Phi^\top + \lambda I)^{-1}\mathbf{y}.$$

Comparing with $w^\star = \Phi^\top \alpha$ shows

$$\alpha = (\Phi\Phi^\top + \lambda I)^{-1}\mathbf{y} = (K + \lambda I)^{-1}\mathbf{y}. \tag{3}$$

Kernel trick. Equations 2 and 3 depend on the features only through inner products $\langle \phi(x), \phi(x') \rangle$, i.e. kernel evaluations $k(x, x')$. Thus we can train and make predictions without explicitly constructing $\phi(x)$, even when ϕ is very high-dimensional (or infinite-dimensional), provided we can compute the $k(x, x')$ directly. *The prediction at a test point is a weighted average of training labels, where the weights are similarities to training points.*

Remark. We used the identity

$$(\Phi^\top \Phi + \lambda I_m)^{-1}\Phi^\top \;=\; \Phi^\top (\Phi\Phi^\top + \lambda I_n)^{-1}.$$

One quick way to see this is to start from the associativity of matrix multiplication:

$$(\Phi^\top \Phi + \lambda I_m)\Phi^\top = \Phi^\top (\Phi\Phi^\top) + \lambda\Phi^\top = \Phi^\top (\Phi\Phi^\top + \lambda I_n).$$

Now left-multiply by $(\Phi^\top \Phi + \lambda I_m)^{-1}$ and right-multiply by $(\Phi\Phi^\top + \lambda I_n)^{-1}$ to obtain the claimed identity.

1.3 Some Common Kernels

Summary table. The below table summarizes a few important kernel functions, and the feature maps to which they correspond.

Feature Map $\boldsymbol{\phi(x)}$	**Kernel $\boldsymbol{k(x, x') = \langle \phi(x), \phi(x') \rangle}$**
$\phi(x) = x$	$k(x, x') = x^\top x'$
All monomials $x_i x_j$ up to degree 2, with appropriate scaling	$k(x, x') = (x^\top x' + 1)^2$
All monomials up to degree d	$k(x, x') = \left(\frac{x^\top x'}{\tau} + c_0\right)^d$
$\phi(x) = ???$	$k_{\text{RBF}}(x, x') = \exp\left(-\frac{\|x - x'\|_2^2}{\tau}\right)$

Table 1: Basis functions and their corresponding kernels.

You may also consider viewing this online demonstration here; it is a Python notebook (marimo) that goes through different kernel functions and visualizes their behavior in real time.

The RBF (Gaussian) kernel corresponds to an *infinite-dimensional* feature map. To see why, note that the exponential can be expanded as a Taylor series $e^z = \sum_{k=0}^{\infty} \frac{z^k}{k!}$, which, when applied to the RBF kernel, yields an inner product over monomials of *every* degree. Since the expansion has infinitely many terms, there is no finite vector $\phi(x)$ such that $\langle \phi(x), \phi(x') \rangle = k_{\text{RBF}}(x, x')$ exactly. This is precisely the setting where the kernel trick is indispensable: we can evaluate $k(x, x')$ directly as a simple scalar computation without ever needing to instantiate the (impossible) explicit feature map.

Deeper discussion about each kernel.

1. Radial Basis Function (RBF / Gaussian)

$$k_{\text{RBF}}(x, x') = \exp\left(-\frac{\|x - x'\|_2^2}{\tau}\right), \qquad \tau > 0$$

   - Corresponds to an infinite-dimensional feature map, as we just said.
   - The function f is C^∞ (infinitely differentiable).
   - Small $\tau \to$ narrow bumps $\to$ under-smoothing; large $\tau \to$ broad bumps $\to$ over-smoothing.
   - This is the universal default. It can approximate any continuous function on a compact set given enough data (universal kernel).

2. Laplacian (Exponential)

$$k_{\text{Lap}}(x, x') = \exp\left(-\frac{\|x - x'\|_1}{\tau}\right), \qquad \tau > 0$$

- Uses the ℓ_1 norm instead of ℓ_2^2, producing a non-differentiable kernel at the origin.
- The resulting set of solutions contains rougher functions, better for targets with kinks or sharp transitions.

3. Polynomial

$$k_{\text{poly}}(x, x') = \left(\frac{x^\top x'}{\tau} + c_0\right)^d$$

- Feature map is finite-dimensional: all monomials up to degree d.
- τ inversely scales the inner product; c_0 controls the trade-off between lower and higher-degree terms.
- Captures global polynomial trends but can diverge wildly outside the training domain.

1.4 Practice Question

When would you kernelize?

Recall that the explicit (primal) ridge regression approach inverts an $m \times m$ system (where m is the feature dimension), while the kernelized (dual) approach inverts an $n \times n$ system (where n is the number of training points).

1. For the polynomial kernel $k(x, x') = (x^\top x' + 1)^2$ with inputs $x \in \mathbb{R}^3$, write down an explicit feature map $\phi(x)$ such that $k(x, x') = \langle \phi(x), \phi(x') \rangle$. What is the dimension m of this feature space?

   Hint: expand $(x^\top x' + 1)^2$ and identify each term with a component of ϕ.

2. Suppose you have $n = 50{,}000$ data points in $d = 3$ dimensions and want to fit a degree-2 polynomial model with ridge regression. Would you prefer the primal (explicit ϕ) or the dual (kernel) formulation? Justify in one sentence.

3. Now suppose $d = 10{,}000$ and $n = 200$, still with a degree-2 polynomial kernel. This kind of high-dimensional, sparse setting arises naturally in text classification: for example, in a *bag-of-words* representation, each document is encoded as a vector in $\mathbb{R}^d$ where d is the vocabulary size and the i-th entry counts how many times word i appears. With a vocabulary of 10,000 words, each document becomes a point in $\mathbb{R}^{10{,}000}$.

   Would your answer from part 2 change? Why?

4. For the RBF kernel $k(x, x') = \exp(-\|x - x'\|^2/\tau)$, explain in 1–2 sentences why the primal approach is impossible, and why the kernel formulation still works.

5. **Bonus:** Random Fourier Features (RFF) approximate the RBF kernel with a D-dimensional explicit feature map. In what sense is RFF a "middle ground" between the primal and dual formulations? When might you prefer RFF over exact kernel regression?

Solution:

1. Expanding $(x^\top x' + 1)^2 = (x_1 x_1' + x_2 x_2' + x_3 x_3' + 1)^2$, we collect all cross terms. One valid feature map is:

$$\phi(x) = \begin{bmatrix} x_1^2 & x_2^2 & x_3^2 & \sqrt{2}\,x_1 x_2 & \sqrt{2}\,x_1 x_3 & \sqrt{2}\,x_2 x_3 & \sqrt{2}\,x_1 & \sqrt{2}\,x_2 & \sqrt{2}\,x_3 & 1 \end{bmatrix}^\top$$

   so $m = 10$. In general, $m = \binom{d+p}{p}$, which for $d = 3, p = 2$ gives $\binom{5}{2} = 10$.

2. Primal is far better. The primal inverts a 10×10 system; the dual inverts a $50{,}000 \times 50{,}000$ system. Since $m \ll n$, the explicit feature map is computationally cheap.

3. Yes — now $m = \binom{10{,}002}{2} = 50{,}015{,}001 \approx 50\text{M}$, while $n = 200$. The dual inverts a 200×200 system, which is trivial, while the primal would require inverting a $50\text{M} \times 50\text{M}$ matrix. The kernel formulation is overwhelmingly preferable. The key insight: kernelize when $n \ll m$, use primal when $m \ll n$.

   This also illustrates why kernel methods have historically been popular for text classification: bag-of-words features are very high-dimensional (large m) but datasets are often modest in size (small n), which is exactly the regime where the kernel trick pays off.

4. The RBF kernel corresponds to an *infinite*-dimensional feature map (its Taylor expansion has infinitely many terms), so we can never write down or store $\phi(x)$ explicitly. The kernel formulation only requires computing $k(x_i, x_j)$ (a scalar for each pair) which is always finite and cheap, regardless of the implicit feature dimension.

5. RFF approximates the infinite-dimensional RBF feature map with a finite D-dimensional random projection $z(x)$, turning kernel regression back into ordinary ridge regression in $\mathbb{R}^D$. You control D: small D is fast but approximate, large D is accurate but slower. You'd prefer RFF over exact kernel regression when n is large (say $n > 10{,}000$), since exact kernel methods require $O(n^3)$ time and $O(n^2)$ storage for the Gram matrix, while RFF requires only $O(nD)$ storage and $O(D^2 n)$ time for ridge regression. The fact that you can choose D means you can pick how accurate you want to be with respect to tradeoffs you are willing to make: an extra hyperparameter that ends up being very useful.

2 Neural Networks

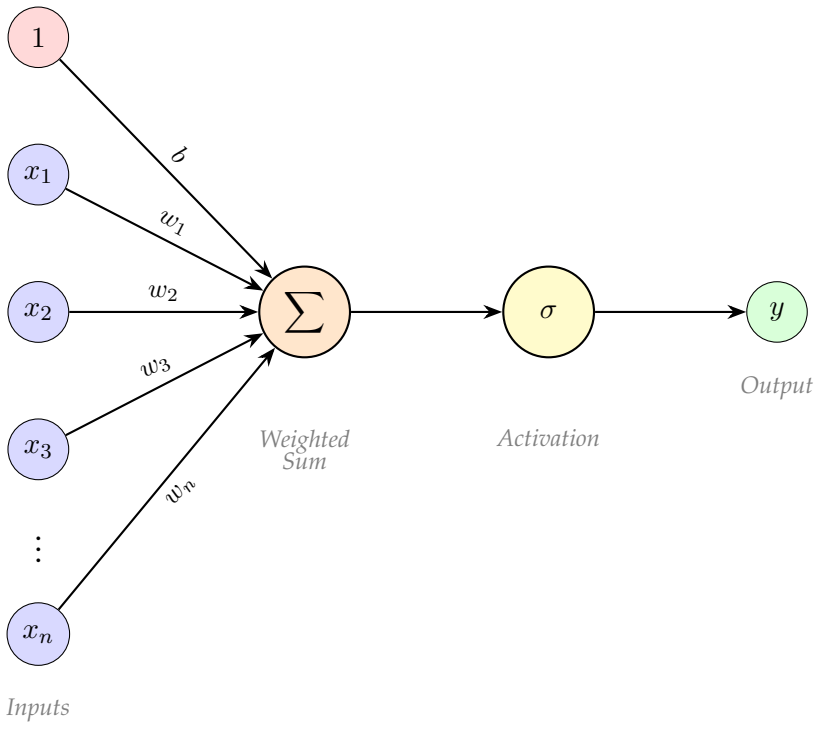
The big picture. We have now seen two paradigms for working with features, and are moving one step further:

1. **Hand-crafted basis functions**: This is where we started. You specify $\phi(x)$ explicitly (identity, polynomials, sinusoids, etc.) and learn w.
2. **Kernel methods**: This is what we just did. You specify a kernel $k(x, x')$ that implicitly defines ϕ, and never need to compute $\phi(x)$ directly.
3. **Neural networks**: This is what we are now going to discuss. You parameterize $\phi_\theta(x)$ and learn both θ and w from data.

Each approach trades off between human effort, computational cost, and flexibility. Kernel methods are elegant and well-understood, but the kernel is fixed and chosen by a person. Neural networks are more flexible and can scale to massive datasets, but they are harder to analyze theoretically and require more careful engineering (choice of architecture, activation functions, optimization hyperparameters, etc.).

2.1 The Perceptron

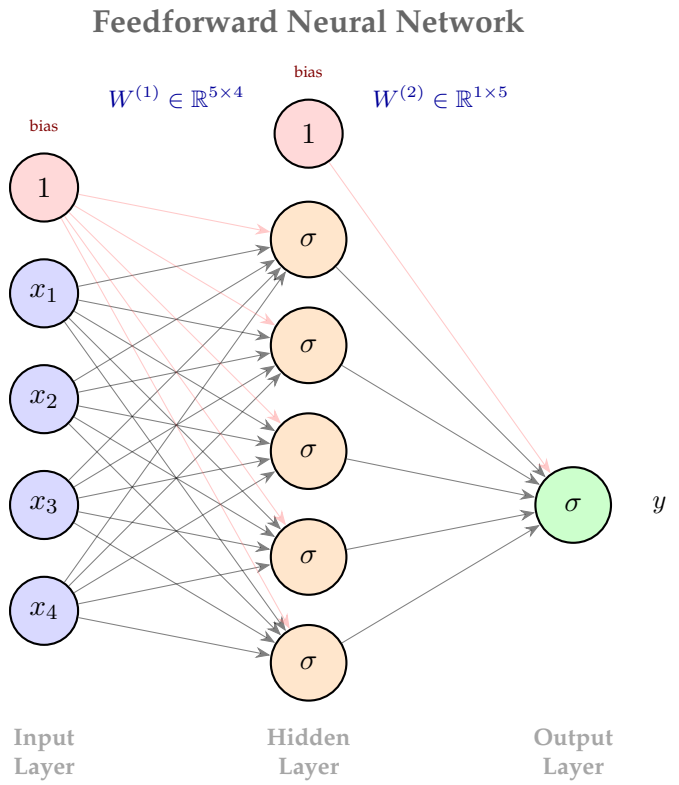
The perceptron was follows a very simple structure:



Mathematically the above model simply corresponds to

$$y = \sigma(x^\top \mathbf{w}),$$

so the perceptron is something we we have already seen before. Now imagine stacking perceptrons on top of each other. To keep the output dimension the same, we then run everything through an extra perceptron layer.



$$\mathbf{h} = \sigma\big(W^{(1)}\mathbf{x} + \mathbf{b}^{(1)}\big) \quad \rightarrow \quad y = \sigma\big(\mathbf{w}^{(2)\top}\mathbf{h} + b^{(2)}\big)$$

This is a neural network. It is also called a **Multilayer Perceptron (MLP)** when it has more than 1 layer. Notice how the outputs of the perceptrons in *layer 1* (the first "wall" of perceptrons) have essentially become the inputs in the first perceptron image. This is called *composition* and is the foundation of deep neural networks, or *deep learning*. We can view these inputs as transformed inputs. Thus, another way to write down the above neural network is as below:

$$y = \sigma(\phi(x)^\top \mathbf{w}),$$

where ϕ corresponds to the transformation on the inputs right before the output layer. Seem familiar?

2.2 Neural Networks Learn Their Features

Recall that the overarching theme so far has been: instead of $x^\top w$, use $\phi(x)^\top w$. Everything we covered in the first section was about choosing or approximating a good ϕ. The fundamental

limitation of all these approaches is that ϕ **is fixed before training and is chosen by a human, rather than letting the data speak**. You pick your kernel or your basis functions, then you learn weights. If your choice of ϕ was poor for the task, no amount of weight-tuning will save you.

Neural networks remove this limitation (though introduce some other engineering problems). Consider a one-hidden-layer network:

$$\hat{y} = \mathbf{w}^{(2)^\top} \sigma\big(W^{(1)} x + b^{(1)}\big) + b^{(2)}$$

Define $\phi_\theta(x) := \sigma(W^{(1)} x + b^{(1)})$, where $\theta = \{W^{(1)}, b^{(1)}\}$. Then we are back to our familiar form:

$$\hat{y} = \mathbf{w}^{(2)^\top} \phi_\theta(x) + b^{(2)},$$

(you can absorb the bias into w if you so desire) except now **the feature map itself has learnable parameters**. During training, gradient descent simultaneously:

- Updates $w^{(2)}, b^{(2)}$: the "regression weights" on top of the features, just like before, and
- Updates $W^{(1)}, b^{(1)}$: the parameters *of the feature map itself*.

This is the key conceptual leap. With kernels, you engineer similarity; with neural networks, you learn it, letting data dictate your features. The learning of these features happens through *backpropagation* and something you should be familiar with by now: gradient descent.

A concrete example: XOR. Consider the XOR function: the points $(0,0), (1,1)$ have label 0 and $(0,1), (1,0)$ have label 1. No linear classifier in $\mathbb{R}^2$ can separate these classes. But a single hidden layer can learn a ϕ that maps these four points into a new space where they *are* linearly separable. For instance, if the network learns

$$h_1 = \text{ReLU}(x_1 + x_2 - 1.5), \qquad h_2 = \text{ReLU}(-x_1 - x_2 + 0.5),$$

then in (h_1, h_2) space, the two classes are separable by a line. The network discovered this representation automatically via backpropagation — you never had to hand-specify it. This is what we mean by *learning* features. You may consider viewing this interactive demonstration of a neural network learning the XOR task.

Depth and compositionality. Deeper networks compose feature maps: $\phi_L \circ \phi_{L-1} \circ \cdots \circ \phi_1(x)$. Each layer refines the representation built by the previous one.

2.2.1 The Universal Approximation Theorem

Universal Approximation Theorem (Cybenko 1989; Hornik et al. 1989)

Let σ be any continuous, nonconstant, nonlinear activation function (e.g. sigmoid, tanh, ReLU). For any continuous function $f^* : [0,1]^d \to \mathbb{R}$ and any $\epsilon > 0$, there exists a single-hidden-layer network

$$f(x) = \sum_{j=1}^{m} v_j \, \sigma\big(w_j^\top x + b_j\big)$$

with some finite width m such that

$$\sup_{x \in [0,1]^d} \big|f(x) - f^*(x)\big| < \epsilon.$$

In this course, we will not dive deep into the technicalities of this result. But we do want to understand the intuition in words: a single hidden layer with enough neurons can approximate any continuous function on a compact domain to arbitrary precision.

What the UAT says.

- Neural networks are *universal function approximators*: there is no continuous target function that is inherently out of reach.
- This holds for a wide range of activation functions — the result is not specific to one particular choice.
- The result is an *existence* theorem: for any target accuracy ϵ, a wide enough single-layer network exists that achieves it. **Crucially, this is not bounded.**

What the UAT does not say.

- It says nothing about how large m needs to be. For some functions, the required width may be astronomically large — exponential in d.
- It says nothing about whether gradient descent (or any other algorithm) can *find* the right weights. Existence $\neq$ learnability.
- It says nothing about *generalization*: a network that perfectly fits f^* on the training set may not generalize to unseen inputs.
- It says nothing about *sample efficiency*: how much data you need to learn a good approximation.

The UAT is reassuring. It tells us we are working with a sufficiently rich model class. However, it does not tell us that any particular network will work well in practice. The gap between "a solution exists" and "gradient descent on finite data will find a solution that generalizes" is where most of the difficulty in deep learning actually lives and why this field is still continuing.

The power of depth. The UAT shows that width alone is sufficient for universality, but depth offers efficiency. There are functions that a depth-L network can represent with a polynomial number of neurons, but that require an *exponential* number of neurons to represent with a single hidden layer. The canonical intuition is compositionality:

- Consider computing the function $f(x) = g_1 \circ g_2 \circ \cdots \circ g_L(x)$, where each g_ℓ is a simple transformation. A deep network mirrors this structure directly, with one layer per g_ℓ.
- A shallow network must represent the entire composed function "in one shot," which can require far more neurons.

A concrete example: computing the parity of d binary inputs (i.e. XOR generalized to d bits) can be done with $O(d)$ neurons in a deep network (a tree of 2-input XOR gates) but requires $O(2^d)$ neurons in a single hidden layer. More practically, this is why deep networks tend to outperform shallow ones on tasks with natural compositional structure like vision and language.

2.3 Activation Functions

There are a wide range of possible nonlinear activation functions to choose from when designing neural networks. Among these, the most popular are ReLU, which you will encounter below, its derivatives like GeLU that provide a small negative signal, and the tanh activation function: $\tanh(z)$. There are many others, including ones usually utilized for the final output layers, that researchers decide between when constructing their models.

Why do we need nonlinear activation functions? Activation functions are vital to constructing neural networks because they introduce non-linear relationships between the inputs and outputs of various layers. With no nonlinear activation functions, all deep neural networks reduce to the perceptron.

$$\text{Let } x \in \mathbb{R}^{d_0}, \qquad h_0 := x, \qquad h_\ell := W_\ell h_{\ell-1} + b_\ell \ \ (\ell = 1, \dots, L), \qquad f(x) := h_L.$$

$$\begin{aligned} f(x) &= W_L h_{L-1} + b_L \\ &= W_L\big(W_{L-1}h_{L-2} + b_{L-1}\big) + b_L \\ &= W_L W_{L-1} h_{L-2} + W_L b_{L-1} + b_L \\ &\ \ \vdots \\ &= \Big(\prod_{\ell=L}^{1} W_\ell\Big)x \ + \ \sum_{k=1}^{L}\Big(\prod_{\ell=L}^{k+1} W_\ell\Big)b_k, \end{aligned}$$

where by convention $\prod_{\ell=L}^{L+1} W_\ell = I$ to make sure the multiplication works in the last index. Hence $f(x) = W_{\text{eq}}x + b_{\text{eq}}$ with

$$W_{\text{eq}} := \prod_{\ell=L}^{1} W_\ell, \qquad b_{\text{eq}} := \sum_{k=1}^{L}\Big(\prod_{\ell=L}^{k+1} W_\ell\Big)b_k,$$

so a depth-L network with no intermediate activations is exactly a single (one-layer) map.

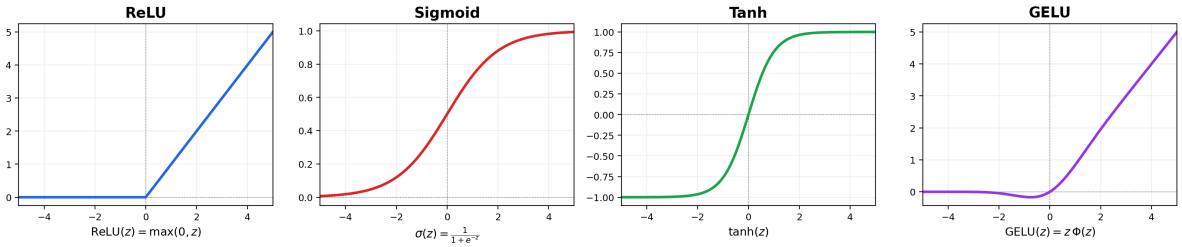


Figure 2: Common activation functions used in neural networks.

Some activation functions you will encounter.

1. **ReLU (Rectified Linear Unit):**

$$\mathrm{ReLU}(z) = \max(0, z)$$

   The most widely used activation function in modern deep learning. It is computationally cheap and avoids the vanishing gradient problem for positive inputs, since its derivative is exactly 1 for $z > 0$. The downside is that neurons can "die". If a unit's input is always negative, the gradient is always zero and the weights never update!

2. **Sigmoid:**

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

   Squashes inputs to $(0, 1)$. You have already seen this as the output activation for binary classification (logistic regression). As a hidden-layer activation, it suffers from vanishing gradients: when $|z|$ is large, $\sigma'(z) \approx 0$, which slows learning in deep networks.

3. **Tanh:**

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

   A rescaled sigmoid: $\tanh(z) = 2\sigma(2z) - 1$. Its output is centered at zero with range $(-1, 1)$, which tends to make optimization easier than sigmoid. It still saturates for large $|z|$, but generally performs better than sigmoid as a hidden-layer activation.

4. **GELU (Gaussian Error Linear Unit):**

$$\mathrm{GELU}(z) = z \cdot \Phi(z)$$

   where Φ is the CDF of the standard normal distribution. Unlike ReLU, which hard-zeros negative inputs, GELU provides a smooth, non-monotonic gate that allows a small negative signal through. This is the default activation in most modern transformer architectures (BERT, GPT, etc.), often only replaced by options like SiLU.

As a rule of thumb: use ReLU or GELU for hidden layers and choose your **output activation** based on the task (sigmoid for binary classification, softmax for multiclass, identity for regression).

2.4 Conceptual Question

What is the most basic nonlinear activation function you can have that will serve its purpose?

Solution: Answer: ReLU. ReLU is simply $y = x$ with negative values removed, which gives you the capability of expression nonlinearity. Technically the step function (either 0 or 1) can be said to be simpler, but it does not give backprop anything to work with.

2.5 No Free Lunch

The preceding sections might suggest that neural networks are a silver bullet: they learn features, they are universal approximators, and deeper networks are more efficient. So what's the catch?

2.5.1 Optimization is Hard

The loss landscape of a neural network is non-convex. Unlike the convex objectives we saw in linear and ridge regression (which have a unique global minimum), neural network training involves navigating a surface with many local minima, saddle points, and flat regions. Gradient descent is not guaranteed to find the global optimum. In practice, it works surprisingly well for reasons that are still actively researched, but several concrete pitfalls arise:

- **Vanishing gradients**: In deep networks with sigmoid or tanh activations, gradients can shrink exponentially as they propagate backward through layers (since $|\sigma'(z)| \leq 1/4$ for sigmoid). Layers close to the input barely update. This was the primary reason deep networks were considered untrainable for decades, and is a major motivation for ReLU and residual connections.
- **Exploding gradients**: The opposite problem — gradients grow exponentially, causing unstable updates. Gradient clipping and careful initialization schemes are standard mitigations.
- **Sensitivity to hyperparameters**: Learning rate, batch size, initialization, and architecture choices can dramatically affect whether training converges at all, let alone to a good solution. Deep Reinforcement Learning is notorious for having this problem.

2.5.2 The Train–Test Gap: Overfitting and Generalization

A network that achieves zero training loss may perform terribly on unseen data. This is the fundamental challenge of generalization, and it is especially acute for neural networks because they are so expressive — a sufficiently large network can memorize any finite training set, including its noise.

Standard tools for controlling overfitting include:

- **Regularization**: Weight decay (L_2 penalty on $\|\theta\|^2$, analogous to ridge regression) discourages unnecessarily large weights.
- **Dropout**: During training, randomly set hidden units to zero with some probability p. This makes sure that the model can never rely on a single neuron / circuit too much.
- **Early stopping**: Monitor validation loss during training and stop when it begins to increase, even if training loss is still decreasing.

- **Data augmentation**: Artificially expand the training set by applying label-preserving transformations (rotations, crops, noise, etc.), effectively encoding prior knowledge about invariances.

2.6 Exercise: Forward Pass, Backward Pass, and Design Intuitions

Consider the following two-layer neural network for binary classification. The network takes input $x \in \mathbb{R}^2$, has a hidden layer of width 3, and produces a single scalar output:

$$\begin{aligned} z^{(1)} &= W^{(1)}x + b^{(1)} \in \mathbb{R}^3 \\ h &= \text{ReLU}(z^{(1)}) \in \mathbb{R}^3 \\ z^{(2)} &= {w^{(2)}}^\top h + b^{(2)} \in \mathbb{R} \\ \hat{y} &= \sigma(z^{(2)}) \end{aligned}$$

where ReLU is applied elementwise, $\sigma(z) = \frac{1}{1+e^{-z}}$ is the sigmoid function, and we train with binary cross-entropy loss:

$$L = -\big[y \log \hat{y} + (1-y)\log(1-\hat{y})\big]$$

Suppose the parameters are initialized as:

$$W^{(1)} = \begin{bmatrix} 0.5 & -0.3 \\ 0.2 & 0.8 \\ -0.4 & 0.6 \end{bmatrix}, \quad b^{(1)} = \begin{bmatrix} 0.1 \\ -0.1 \\ 0.0 \end{bmatrix}, \quad w^{(2)} = \begin{bmatrix} 0.7 \\ -0.5 \\ 0.3 \end{bmatrix}, \quad b^{(2)} = -0.2$$

1. **Forward pass.** For the input $x = [1,\ 2]^\top$, compute the hidden activations h_1, h_2, h_3 and the output $\hat{y}$. Leave $\hat{y}$ in terms of $\sigma(\cdot)$.

2. **Backward pass.** Using the chain rule, derive a general expression for $\frac{\partial L}{\partial w_{ij}^{(1)}}$ in terms of y, $\hat{y}$, $w_j^{(2)}$, h_j, and x_i.
   Hint: You should find that $\frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial (\cdot)} = (\hat{y} - y)$, where $(\cdot)$ denotes the pre-sigmoid activation, which simplifies the expression considerably. Recall that $\sigma'(z) = \sigma(z)(1-\sigma(z))$.

3. **Dead neurons.** Evaluate $\frac{\partial L}{\partial w_{11}^{(1)}}$ numerically for this input. What do you notice, and what does this imply about how $w_{11}^{(1)}$ and $w_{12}^{(1)}$ will change during this gradient step? Relate your answer to a known failure mode of ReLU, answer how this limits your learning from an information perspective.

4. **Design intuitions.** Answer each briefly (2–3 sentences).

   (a) Suppose you replaced every ReLU activation with the identity function $\sigma(z) = z$. Without computing anything, what would this network reduce to? Why does this mean nonlinear activations are essential?

   (b) You are training a network on a dataset with 500 samples and 10 features. A colleague proposes a network with 5 hidden layers of width 1024 each. What potential problems do you foresee? Would you suggest a different architecture, and why?

   (c) In the "automated feature learning" view, what is the role of the hidden layer versus the output layer? If you froze $W^{(1)}$ and $b^{(1)}$ at their initial random values and only trained $w^{(2)}$ and $b^{(2)}$, how would this compare to a kernel method?

Solution:

1. **Forward pass.** We compute the pre-activations $z_j^{(1)} = \sum_i w_{ij}^{(1)} x_i + b_j^{(1)}$:

$$z_1^{(1)} = 0.5(1) + (-0.3)(2) + 0.1 = 0.5 - 0.6 + 0.1 = 0.0$$
$$z_2^{(1)} = 0.2(1) + 0.8(2) + (-0.1) = 0.2 + 1.6 - 0.1 = 1.7$$
$$z_3^{(1)} = -0.4(1) + 0.6(2) + 0.0 = -0.4 + 1.2 = 0.8$$

   Applying ReLU: $h_1 = \text{ReLU}(0.0) = 0$, $h_2 = \text{ReLU}(1.7) = 1.7$, $h_3 = \text{ReLU}(0.8) = 0.8$. The output pre-activation is:

$$z^{(2)} = 0.7(0) + (-0.5)(1.7) + 0.3(0.8) + (-0.2) = 0 - 0.85 + 0.24 - 0.2 = -0.81$$

   So $\hat{y} = \sigma(-0.81)$.

2. **Backward pass.** We apply the chain rule:

$$\frac{\partial L}{\partial w_{ij}^{(1)}} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z^{(2)}} \cdot \frac{\partial z^{(2)}}{\partial h_j} \cdot \frac{\partial h_j}{\partial z_j^{(1)}} \cdot \frac{\partial z_j^{(1)}}{\partial w_{ij}^{(1)}}$$

   Computing each factor:

   - $\frac{\partial L}{\partial \hat{y}} = -\frac{y}{\hat{y}} + \frac{1-y}{1-\hat{y}}$
   - $\frac{\partial \hat{y}}{\partial z^{(2)}} = \hat{y}(1-\hat{y})$
   - Combined (using the hint): $\frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z^{(2)}} = \hat{y} - y$
   - $\frac{\partial z^{(2)}}{\partial h_j} = w_j^{(2)}$
   - $\frac{\partial h_j}{\partial z_j^{(1)}} = \mathbf{1}[z_j^{(1)} > 0]$ (the ReLU derivative)
   - $\frac{\partial z_j^{(1)}}{\partial w_{ij}^{(1)}} = x_i$

   Putting it together:

$$\boxed{\frac{\partial L}{\partial w_{ij}^{(1)}} = (\hat{y} - y) \cdot w_j^{(2)} \cdot \mathbf{1}[z_j^{(1)} > 0] \cdot x_i}$$

3. **Dead neurons.** For $w_{11}^{(1)}$, we have $j = 1$, $i = 1$. Since $z_1^{(1)} = 0.0$, the ReLU derivative $\mathbf{1}[z_1^{(1)} > 0] = 0$. Therefore:

$$\frac{\partial L}{\partial w_{11}^{(1)}} = (\hat{y} - y) \cdot 0.7 \cdot 0 \cdot 1 = 0$$

   The gradient is exactly zero. The same holds for $w_{12}^{(1)}$ (since both share the same indicator through $z_1^{(1)}$). This means **neither weight feeding into h_1 will update on this step**. This is the "dying ReLU" problem: once a neuron's pre-activation is non-positive for all training

inputs, it receives zero gradient and can never recover. This is one motivation for alternatives like Leaky ReLU or GELU, which allow a small gradient to flow even for negative pre-activations. Another motivation is activations that can provide negative signal allow us to preserve more useful information for prediction.

4. **Design intuitions.**

   (a) As shown in Section 2.3, with identity activations, all layers collapse into a single affine transformation: $f(x) = W_{\text{eq}}x + b_{\text{eq}}$. The network would be equivalent to logistic regression regardless of depth. Nonlinear activations are what give the hidden layers the ability to carve out nonlinear decision boundaries.

   (b) With 500 samples and $\sim$5 million parameters ($1024 \times 1024 \times 4$ layers of connections), the model will almost certainly overfit severely. Deep, wide networks on small datasets memorize the training set rather than learning generalizable features. A much smaller architecture (e.g., 1–2 hidden layers of width 32–64) would be more appropriate. Alternatively, a kernel method would be well-suited at this scale: deep neural networks thrive in high n regimes.

   (c) The first-layer parameters $W^{(1)}, b^{(1)}$ define the *learned feature map* $\phi_\theta(x) = \text{ReLU}(W^{(1)}x + b^{(1)})$; the output-layer parameters $w^{(2)}, b^{(2)}$ are the linear classifier on top of those features. If we froze $W^{(1)}, b^{(1)}$ at their random initialization, then $\phi(x)$ becomes a *fixed* feature map, one that was not designed or learned, but determined entirely by the random seed used at initialization. Training only $w^{(2)}$ is then exactly a linear model on these fixed features. This is closely related to the *random features* framework (and to Random Fourier Features from Section 1.4), where the key idea is that randomly chosen projections followed by a nonlinearity can approximate kernel evaluations, even though no learning was involved in choosing the projections. The core advantage of actually training $W^{(1)}$ is that the network can *adapt* its representation to the data, rather than relying on projections that happen to work well on average. Moreover, given good learned representations, in the real world we will often utilize this idea of training only the last layer for regressing on features: you can, for example, take a pretrained image encoder that learns good representations, and train a single classifier layer at the end to great accuracy! See DINO from Meta.