Image Processing and Recognition

Dr. Călin-Adrian POPA

Lecture 3

October 18th, 2022

- in Chapter 1, we only talked about linear models
- while *neural networks* cover a much richer family of models, we can begin thinking of the linear model as a neural network by expressing it in the language of neural networks
- to begin, let us start by rewriting things in a “layer” notation
- diagrams are used in deep learning to visualize models
- in Figure 1, we depict the linear regression model as a neural network
- note that these diagrams highlight the connectivity pattern, such as how each input is connected to the output, but not the values taken by the weights or biases

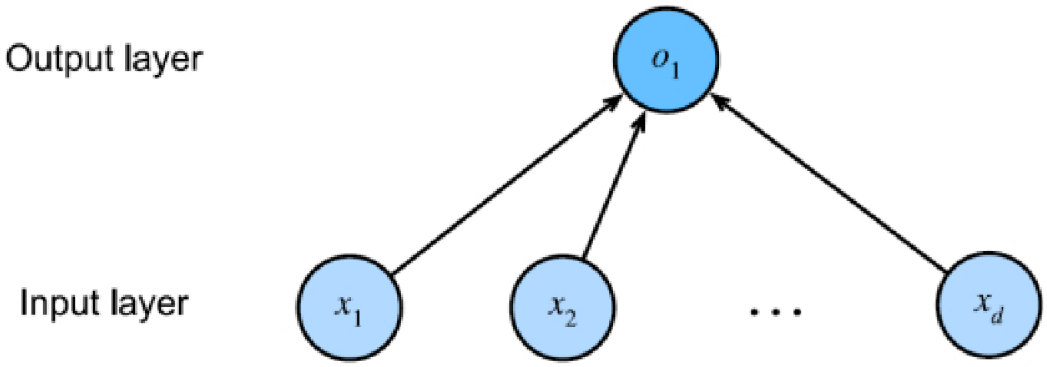

Figure 1: Linear regression is a single-layer neural network.

- for the neural network shown in Figure 1, the inputs are $x_1, \ldots, x_d$, so the *number of inputs* (or *feature dimensionality*) in the input layer is d
- the output of the network in Figure 1 is o_1, so the *number of outputs* in the output layer is 1
- note that the input values are all *given* and there is just a single *computed* neuron
- focusing on where computation takes place, conventionally, we do not consider the input layer when counting layers
- that is to say, the *number of layers* for the neural network in Figure 1 is 1
- we can think of linear regression models as neural networks consisting of just a single artificial neuron, or as *single-layer neural networks*

- since, for linear regression, every input is connected to every output (in this case, there is only one output), we can regard this transformation (the output layer in Figure 1) as a *fully-connected layer* or *dense layer*
- we will talk a lot more about networks composed of such layers in this and the following chapters
- since linear regression (invented in 1795) predates computational neuroscience, linear regression was not originally described as a neural network
- to see why linear models were a natural place to begin when the cyberneticists/neurophysiologists Warren McCulloch and Walter Pitts began to develop models of artificial neurons, consider the cartoonish picture of a biological neuron in Figure 2, consisting of *dendrites* (input terminals), the *nucleus* (CPU), the *axon* (output wire), and the *axon terminals* (output terminals), enabling connections to other neurons via *synapses*

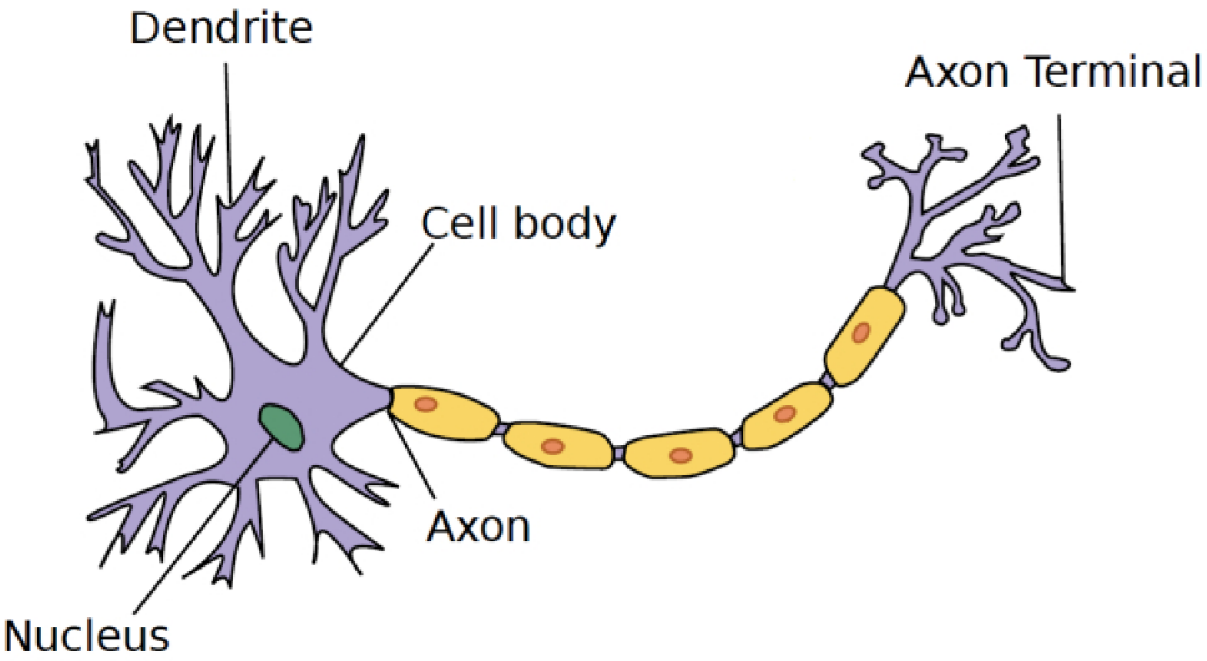

Figure 2: The real neuron.

- information x_i arriving from other neurons (or environmental sensors, such as the retina) is received in the dendrites
- in particular, that information is weighted by *synaptic weights* w_i, determining the effect of the inputs (e.g., activation or inhibition via the product $x_i w_i$)
- the weighted inputs arriving from multiple sources are aggregated in the nucleus as a weighted sum $y = \sum_{i=1}^{d} x_i w_i + b$, and this information is then sent for further processing in the axon y, typically after some nonlinear processing via $\sigma(y)$
- from there, it either reaches its destination (e.g., a muscle) or is fed into another neuron via its dendrites

- certainly, the high-level idea that many such units could be stitched together with the right connectivity and right learning algorithm, to produce far more interesting and complex behavior than any one neuron alone could express, owes to our study of real biological neural systems
- at the same time, most research in deep learning today draws little direct inspiration from neuroscience
- although airplanes might have been *inspired* by birds, ornithology has not been the primary driver of aeronautics innovation for many years
- likewise, inspiration in deep learning these days comes in equal or greater measure from mathematics, statistics, and computer science

2.1 From linear models to deep neural networks

- *softmax regression* can also be framed as a neural network
- let us start off with a simple image classification problem; here, each input consists of a 2×2 grayscale image
- we can represent each pixel value with a single scalar, giving us four features x_1, x_2, x_3, x_4
- further, let us assume that each image belongs to one among the categories "cat", "chicken", and "dog"
- next, we have to choose how to represent the labels; we have two obvious choices
- perhaps the most natural impulse would be to choose $y \in \{1, 2, 3\}$, where the integers represent $\{\text{dog}, \text{cat}, \text{chicken}\}$, respectively; this is a great way of *storing* such information on a computer
- if the categories had some natural ordering among them, say if we were trying to predict $\{\text{baby}, \text{toddler}, \text{adolescent}, \text{young adult}, \text{adult}, \text{geriatric}\}$, then it might even make sense to cast this problem as regression, and keep the labels in this format

- but general classification problems do not come with natural orderings among the classes
- fortunately, statisticians long ago invented a simple way to represent categorical data: the *one-hot encoding*
- a one-hot encoding is a vector with as many components as we have categories
- the component corresponding to a particular instance's category is set to 1, and all other components are set to 0
- in our case, a label y would be a three-dimensional vector, with $(1, 0, 0)$ corresponding to "cat", $(0, 1, 0)$ to "chicken", and $(0, 0, 1)$ to "dog":

$$y \in \{(1, 0, 0), (0, 1, 0), (0, 0, 1)\}.$$

- in order to estimate the conditional probabilities associated with all the possible classes, we need a model with multiple outputs, one per class
- to address classification with linear models, we will need as many affine functions as we have outputs
- each output will correspond to its own affine function
- in our case, since we have 4 features and 3 possible output categories, we will need 12 scalars to represent the weights (w with subscripts), and 3 scalars to represent the biases (b with subscripts)
- we compute these three *logits*, o_1, o_2, and o_3, for each input:

$$\begin{aligned} o_1 &= x_1 w_{11} + x_2 w_{12} + x_3 w_{13} + x_4 w_{14} + b_1, \\ o_2 &= x_1 w_{21} + x_2 w_{22} + x_3 w_{23} + x_4 w_{24} + b_2, \\ o_3 &= x_1 w_{31} + x_2 w_{32} + x_3 w_{33} + x_4 w_{34} + b_3. \end{aligned}$$

- we can depict this calculation with the neural network diagram shown in Figure 3
- just as in linear regression, softmax regression is also a *single-layer neural network*
- and, since the calculation of each output, o_1, o_2, and o_3, depends on all inputs, x_1, x_2, x_3, and x_4, the output layer of softmax regression can also be described as a fully-connected layer

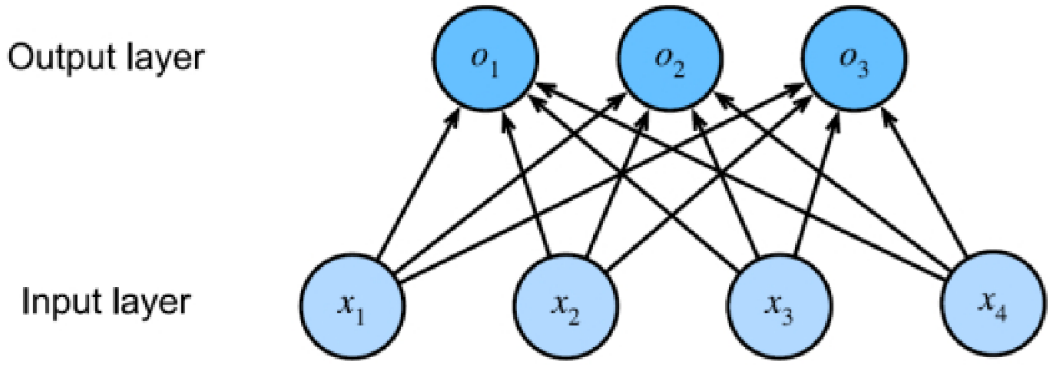

Figure 3: Softmax regression is a single-layer neural network.

- to express the model more compactly, we use the linear algebra notation
- in vector form, we arrive at $\boldsymbol{o} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$, a form better suited both for mathematics, and for writing code
- note that we have gathered all of our weights into a 3×4 matrix, and that, for features of a given data example $\boldsymbol{x}$, our outputs are given by a matrix-vector product of our weights by our input features, plus our biases $\boldsymbol{b}$
- as we will see in this and subsequent chapters, fully-connected layers are ubiquitous in deep learning
- however, as the name suggests, fully-connected layers are *fully* connected, with potentially many learnable parameters
- specifically, for any fully-connected layer with d inputs and q outputs, the parameterization cost is $\mathcal{O}(dq)$, which can be prohibitively high in practice

- as we discussed in Chapter 1, we want to interpret the outputs of our model as probabilities
- we do this by applying the *softmax function* to the logits vector $\boldsymbol{o}$:

$$\hat{\boldsymbol{y}} = \text{softmax}(\boldsymbol{o}) \qquad \text{where} \qquad \hat{y}_j = \frac{\exp(o_j)}{\sum_{j'=1}^{q} \exp(o_{j'})}.$$

- it is easy to see that $\sum_{j=1}^{q} \hat{y}_j = 1$, with $0 \leq \hat{y}_j \leq 1$, for all $1 \leq j \leq q$
- thus, $\hat{\boldsymbol{y}}$ is a proper *probability distribution*, whose element values can be interpreted accordingly
- note that the softmax operation does not change the ordering among the logits $\boldsymbol{o}$, which are simply the pre-softmax values that determine the probabilities assigned to each class
- therefore, during prediction, we can still pick out the most likely class by:

$$\underset{j}{\text{argmax}}\, \hat{y}_j = \underset{j}{\text{argmax}}\, o_j.$$

- although softmax is a nonlinear function, the outputs of softmax regression are still *determined* by an affine transformation of input features; thus, softmax regression is a linear model
- assume that we are given a design matrix $\boldsymbol{X}$ of n examples with feature dimensionality (number of inputs) d
- assume that we have q categories in the output
- then, the design matrix $\boldsymbol{X} \in \mathbb{R}^{n \times d}$, weights $\boldsymbol{W} \in \mathbb{R}^{d \times q}$, and the bias satisfies $\boldsymbol{b} \in \mathbb{R}^{1 \times q}$
- thus, we have:

$$\begin{aligned} \boldsymbol{O} &= \boldsymbol{X}\boldsymbol{W} + \boldsymbol{b}, \\ \hat{\boldsymbol{Y}} &= \text{softmax}(\boldsymbol{O}). \end{aligned}$$

- this accelerates the dominant operation into a matrix-matrix product $\boldsymbol{X}\boldsymbol{W}$ vs. the matrix-vector products we would be executing if we processed one example at a time
- since each row in $\boldsymbol{X}$ represents a data example, the softmax operation itself can be computed *row-wise*: for each row of $\boldsymbol{O}$, exponentiate all entries and then normalize them by the sum
- triggering *broadcasting* during the summation $\boldsymbol{X}\boldsymbol{W} + \boldsymbol{b}$, both the logits $\boldsymbol{O} \in \mathbb{R}^{n \times q}$ and output probabilities $\hat{\boldsymbol{Y}} \in \mathbb{R}^{n \times q}$ are $n \times q$ matrices

2.2 Multilayer perceptrons

- now, we will introduce our first truly *deep* network
- the simplest deep networks are called *multilayer perceptrons*, and they consist of multiple layers of neurons, each fully connected to those in the layer below (from which they receive input) and those above (which they, in turn, influence)
- we have described the affine transformation, which is a linear transformation added by a bias, in Chapter 1
- consider the model architecture corresponding to our softmax regression example, illustrated in Figure 3

- this model mapped our inputs directly to our outputs via a single affine transformation, followed by a softmax operation
- if our labels truly were related to our input data by an affine transformation, then this approach would be sufficient
- but linearity in affine transformations is a *strong* assumption
- for example, linearity implies the weaker assumption of monotonicity: that any increase in our feature must either always cause an increase in our model's output (if the corresponding weight is positive), or always cause a decrease in our model's output (if the corresponding weight is negative); sometimes that makes sense

- but what about classifying images of cats and dogs?
- should increasing the intensity of the pixel at location $(13, 17)$ always increase (or always decrease) the likelihood that the image depicts a dog?
- reliance on a linear model corresponds to the implicit assumption that the only requirement for differentiating cats vs. dogs is to assess the brightness of individual pixels
- this approach is doomed to fail in a world where inverting an image preserves the category

- and yet, despite the apparent absurdity of linearity here, it is not obvious that we could address the problem with a simple preprocessing fix
- that is because the significance of any pixel depends in complex ways on its context (the values of the surrounding pixels)
- while there might exist a representation of our data that would take into account the relevant interactions among our features, on top of which a linear model would be suitable, we simply do not know how to calculate it by hand
- with deep neural networks, we use training data to jointly learn both a representation via *hidden layers* and a *linear predictor* that acts upon that representation

- we can overcome these limitations of linear models and handle a more general class of functions by incorporating one or more hidden layers
- the easiest way to do this is to stack many fully-connected layers on top of each other
- each layer feeds into the layer above it, until we generate outputs
- we can think of the first $L-1$ layers as our *representation* and the final layer as our *linear predictor*, which can be a linear regressor (for regression) or a logistic/softmax regressor (for classification)
- this architecture is commonly called a *multilayer perceptron*, often abbreviated as MLP
- below, we depict the diagram of an MLP (Figure 4)

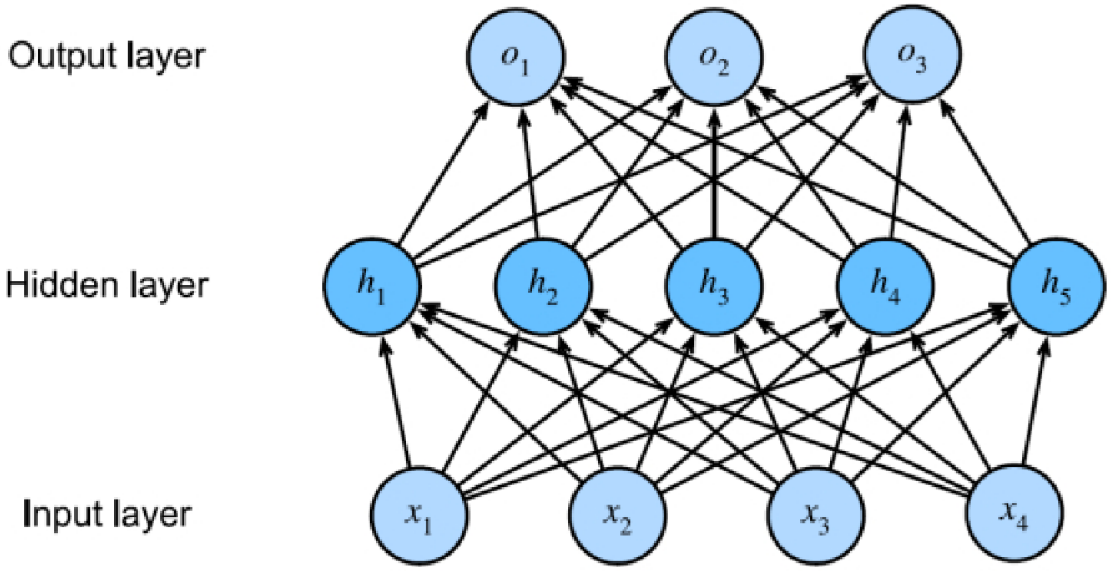

Figure 4: An MLP with a hidden layer of 5 hidden units.

- this MLP has 4 inputs, 3 outputs, and its hidden layer contains 5 hidden units
- since the input layer does not involve any calculations, producing outputs with this network requires implementing the computations for both the hidden and output layers; thus, the number of layers in this MLP is 2
- note that these layers are both *fully connected*
- every input influences every neuron in the hidden layer, and each of these in turn influences every neuron in the output layer

- as before, by the matrix $\boldsymbol{X} \in \mathbb{R}^{n \times d}$, we denote a design matrix of n examples, where each example has d inputs (features)
- for a one-hidden-layer MLP whose hidden layer has h hidden units, denote by $\boldsymbol{H} \in \mathbb{R}^{n \times h}$ the outputs of the hidden layer, which are *hidden representations*
- in mathematics or code, $\boldsymbol{H}$ is also known as a *hidden-layer variable* or a *hidden variable*
- since the hidden and output layers are both fully connected, we have hidden-layer weights $\boldsymbol{W}^{(1)} \in \mathbb{R}^{d \times h}$ and biases $\boldsymbol{b}^{(1)} \in \mathbb{R}^{1 \times h}$ and output-layer weights $\boldsymbol{W}^{(2)} \in \mathbb{R}^{h \times q}$ and biases $\boldsymbol{b}^{(2)} \in \mathbb{R}^{1 \times q}$
- formally, we calculate the outputs $\boldsymbol{O} \in \mathbb{R}^{n \times q}$ of the one-hidden-layer MLP as follows:

$$\begin{aligned} \boldsymbol{H} &= \boldsymbol{X}\boldsymbol{W}^{(1)} + \boldsymbol{b}^{(1)}, \\ \boldsymbol{O} &= \boldsymbol{H}\boldsymbol{W}^{(2)} + \boldsymbol{b}^{(2)}. \end{aligned}$$

- note that after adding the hidden layer, our model now requires us to track and update additional sets of parameters
- so what have we gained in exchange?
- we might be surprised to find out that – in the model defined above – *we gain nothing for our troubles*
- the reason is plain: the hidden units above are given by an affine function of the inputs, and the outputs (pre-softmax) are just an affine function of the hidden units
- an affine function of an affine function is itself an affine function
- moreover, our linear model was already capable of representing any affine function

- we can view the equivalence formally, by proving that, for any values of the weights, we can just collapse out the hidden layer, yielding an equivalent single-layer model with parameters $\boldsymbol{W} = \boldsymbol{W}^{(1)}\boldsymbol{W}^{(2)}$ and $\boldsymbol{b} = \boldsymbol{b}^{(1)}\boldsymbol{W}^{(2)} + \boldsymbol{b}^{(2)}$:

$$\boldsymbol{O} = (\boldsymbol{X}\boldsymbol{W}^{(1)} + \boldsymbol{b}^{(1)})\boldsymbol{W}^{(2)} + \boldsymbol{b}^{(2)} = \boldsymbol{X}\boldsymbol{W}^{(1)}\boldsymbol{W}^{(2)} + \boldsymbol{b}^{(1)}\boldsymbol{W}^{(2)} + \boldsymbol{b}^{(2)} = \boldsymbol{X}\boldsymbol{W} + \boldsymbol{b}.$$

- in order to realize the potential of multilayer architectures, we need one more key ingredient: a *nonlinear activation function* σ to be applied to each hidden unit, following the affine transformation
- the outputs of activation functions (e.g., $\sigma(\cdot)$) are called *activations*
- in general, with activation functions in place, it is no longer possible to collapse our MLP into a linear model:

$$\begin{aligned} \boldsymbol{H} &= \sigma(\boldsymbol{X}\boldsymbol{W}^{(1)} + \boldsymbol{b}^{(1)}), \\ \boldsymbol{O} &= \boldsymbol{H}\boldsymbol{W}^{(2)} + \boldsymbol{b}^{(2)}. \end{aligned}$$

- since each row in $\boldsymbol{X}$ corresponds to an example, with some abuse of notation, we define the *nonlinearity* σ to apply to its inputs in a row-wise fashion, i.e., one example at a time
- note that we used the notation for softmax in the same way to denote a row-wise operation before
- often, the activation functions that we apply to hidden layers are not merely row-wise, but element-wise
- that means that, after computing the linear portion of the layer, we can calculate each activation without looking at the values taken by the other hidden units; this is true for most activation functions
- to build more general MLPs, we can continue stacking such hidden layers, e.g., $\boldsymbol{H}^{(1)} = \sigma_1(\boldsymbol{X}\boldsymbol{W}^{(1)} + \boldsymbol{b}^{(1)})$ and $\boldsymbol{H}^{(2)} = \sigma_2(\boldsymbol{H}^{(1)}\boldsymbol{W}^{(2)} + \boldsymbol{b}^{(2)})$, one on top of the other, yielding ever more expressive models

2.2 Multilayer perceptrons

- MLPs can capture complex interactions among our inputs via their hidden neurons, which depend on the values of each of the inputs
- we can easily design hidden nodes to perform arbitrary computation, for instance, basic logic operations on a pair of inputs
- moreover, for certain choices of the activation function, it is widely known that MLPs are *universal approximators*
- even with a single-hidden-layer network, given enough nodes (possibly absurdly many), and the right set of weights, we can model any function, though actually learning that function is the hard part
- we might think of our neural network as being a bit like the C programming language
- the language, like any other modern language, is capable of expressing *any* computable program
- but actually coming up with a program that meets our specifications is the hard part

- moreover, just because a single-hidden-layer network *can* learn any function, does not mean that we should try to solve all of our problems with single-hidden-layer networks
- in fact, we can approximate many functions much more compactly by using deeper (vs. wider) networks
- activation functions decide whether a neuron should be activated or not, by calculating the weighted sum and further adding bias to it
- they are differentiable operators to transform input signals to outputs, while most of them add a nonlinearity
- because activation functions are fundamental to deep learning, let us briefly survey some common activation functions

- the most popular choice, due to both simplicity of implementation and its good performance on a variety of predictive tasks, is the *rectified linear unit* (ReLU)
- ReLU provides a very simple nonlinear transformation
- given an element x, the function is defined as the maximum of that element and 0:

$$\text{ReLU}(x) = \max(x, 0).$$

- informally, the ReLU function retains only positive elements and discards all negative elements by setting the corresponding activations to 0
- to gain some intuition, we plot the function in Figure 5; as we can see, the activation function is *piecewise linear*

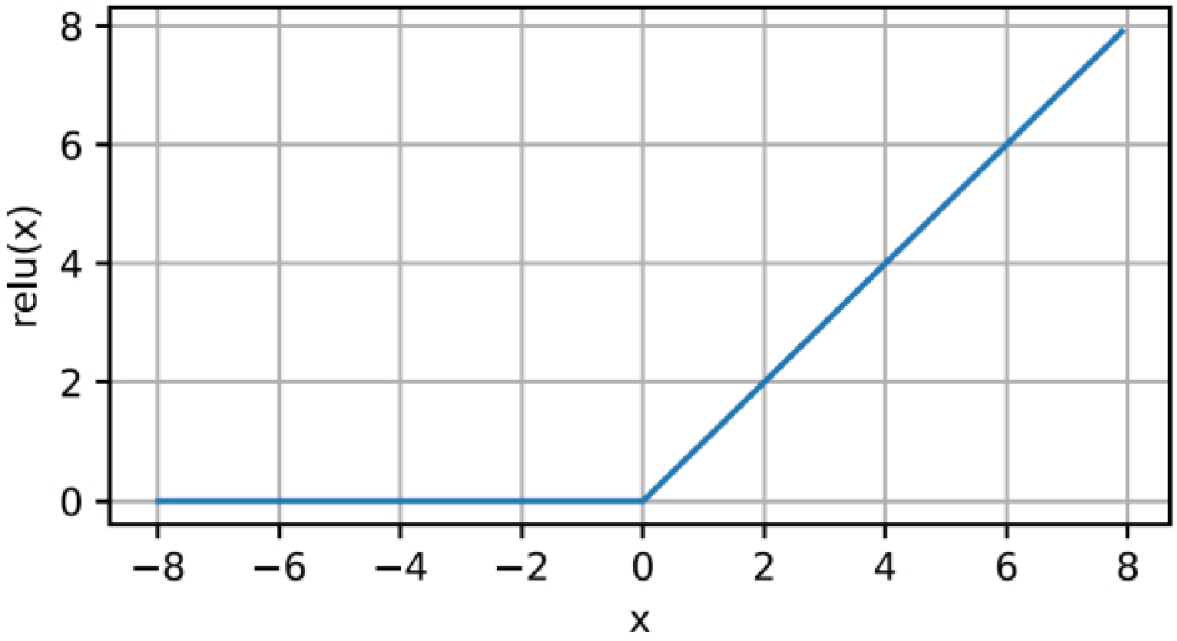

Figure 5: The ReLU function.

- when the input is negative, the derivative of the ReLU function is 0, and when the input is positive, the derivative of the ReLU function is 1
- note that the ReLU function is not differentiable when the input takes value precisely equal to 0
- in these cases, we default to the left-hand-side derivative and say that the derivative is 0 when the input is 0
- we can get away with this, because the input may never actually be zero
- we plot the derivative of the ReLU function in Figure 6

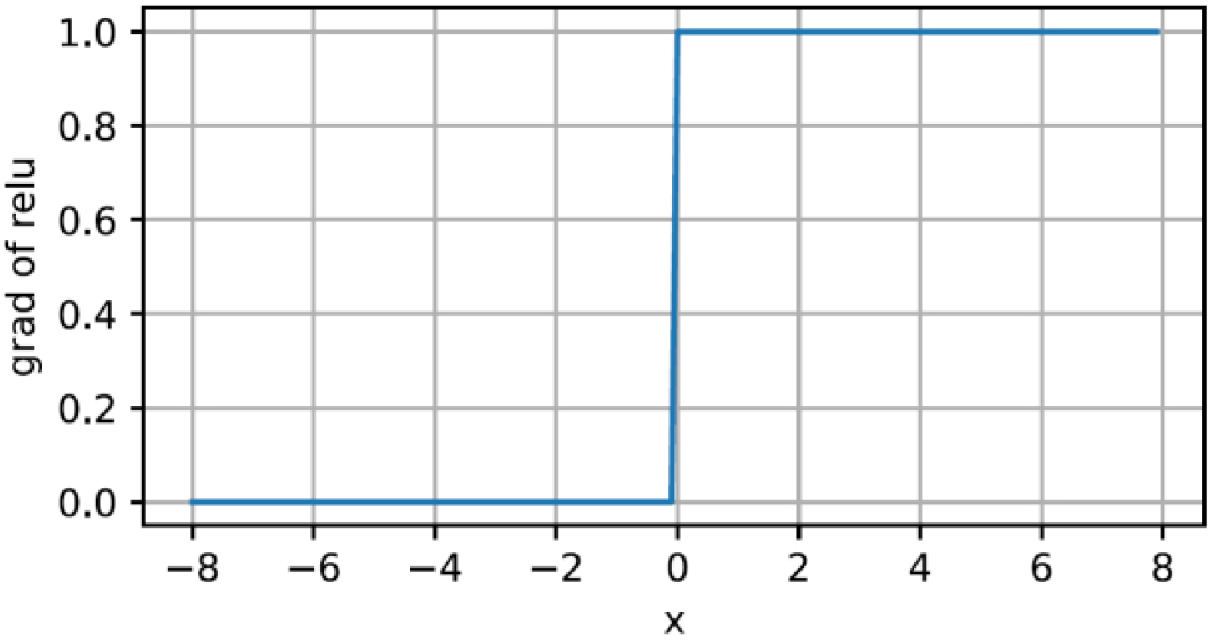

Figure 6: The derivative of the ReLU function.

- the reason for using ReLU is that its derivatives are particularly well behaved: either they vanish or they just let the argument through
- this makes optimization better behaved and it mitigates the well-documented problem of vanishing gradients that plagued previous versions of neural networks (more on this later)
- note that there are many variants of the ReLU function, including the *parameterized ReLU* (pReLU) function
- this variation adds a linear term to ReLU, so some information still gets through, even when the argument is negative:

$$\text{pReLU}(x) = \max(0, x) + \alpha \min(0, x).$$

- the *sigmoid function* transforms its inputs, for which values lie in the domain $\mathbb{R}$, to outputs that lie in the interval $(0, 1)$
- for that reason, the sigmoid is often called a *squashing function*: it squashes any input in the range $(-\infty, \infty)$ to some value in the range $(0, 1)$:

$$\text{sigmoid}(x) = \frac{1}{1 + \exp(-x)}.$$

- in the earliest neural networks, scientists were interested in modeling biological neurons which either *fire* or *do not fire*
- thus, the pioneers of this field, going all the way back to McCulloch and Pitts, the inventors of the artificial neuron, focused on *thresholding units*
- a thresholding activation takes value 0 when its input is below some threshold, and value 1 when the input exceeds the threshold

- when attention shifted to gradient based learning, the sigmoid function was a natural choice because it is a smooth, differentiable approximation to a thresholding unit
- sigmoids are still widely used as activation functions on the output units, when we want to interpret the outputs as probabilities for *binary classification* problems (we can think of the sigmoid as a special case of the softmax)
- however, the sigmoid has mostly been replaced by the simpler and more easily trainable ReLU for most use in hidden layers
- in the chapter on recurrent neural networks, we will describe architectures that leverage sigmoid units to control the flow of information across time
- we plot the sigmoid function in Figure 7; note that, when the input is close to 0, the sigmoid function approaches a linear transformation

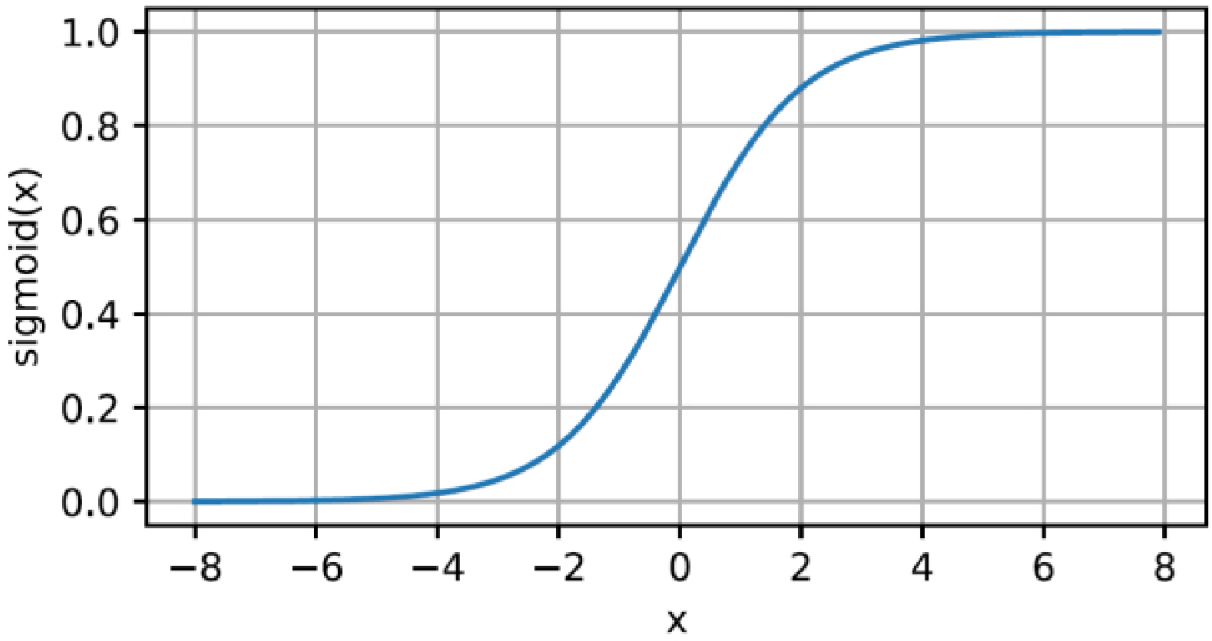

Figure 7: The sigmoid function.

- the derivative of the sigmoid function is given by the following equation:

$$\frac{d}{dx}\text{sigmoid}(x) = \frac{\exp(-x)}{(1+\exp(-x))^2} = \text{sigmoid}(x)(1-\text{sigmoid}(x)).$$

- the derivative of the sigmoid function is plotted in Figure 8
- note that when the input is 0, the derivative of the sigmoid function reaches a maximum of 0.25
- as the input diverges from 0 in either direction, the derivative approaches 0

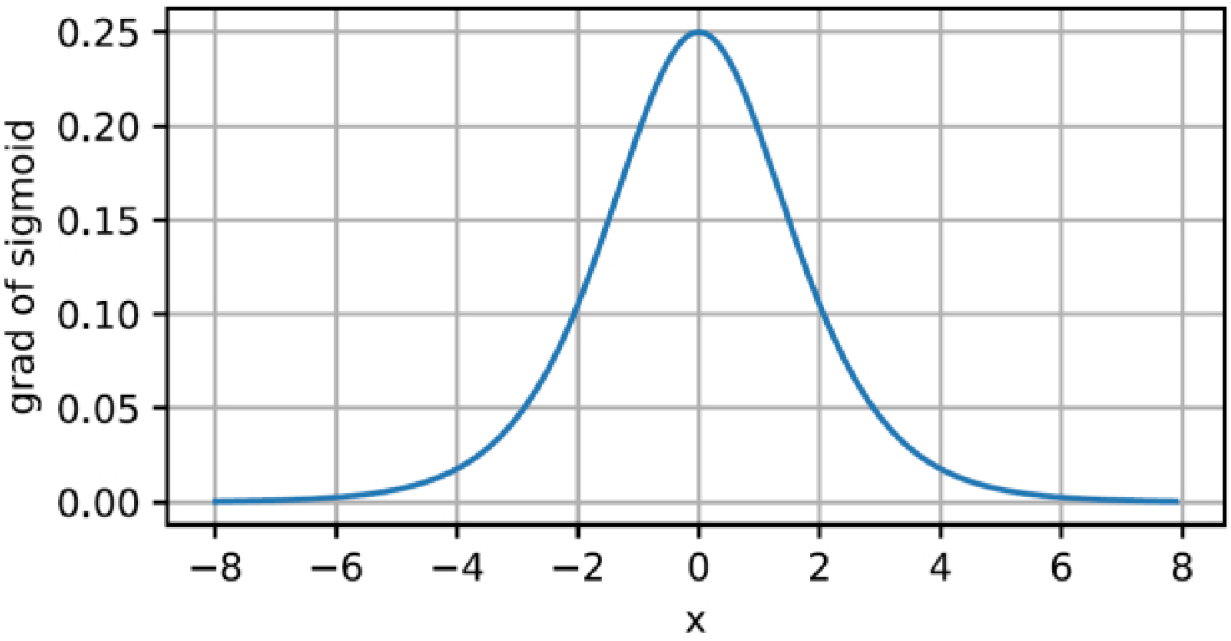

Figure 8: The derivative of the sigmoid function.

- like the sigmoid function, the *tanh* (*hyperbolic tangent*) function also squashes its inputs, transforming them into elements in the interval between -1 and 1:

$$\tanh(x) = \frac{1 - \exp(-2x)}{1 + \exp(-2x)}.$$

- we plot the tanh function in Figure 9
- note that, as the input nears 0, the tanh function approaches a linear transformation
- although the shape of the function is similar to that of the sigmoid function, the tanh function exhibits point symmetry about the origin of the coordinate system

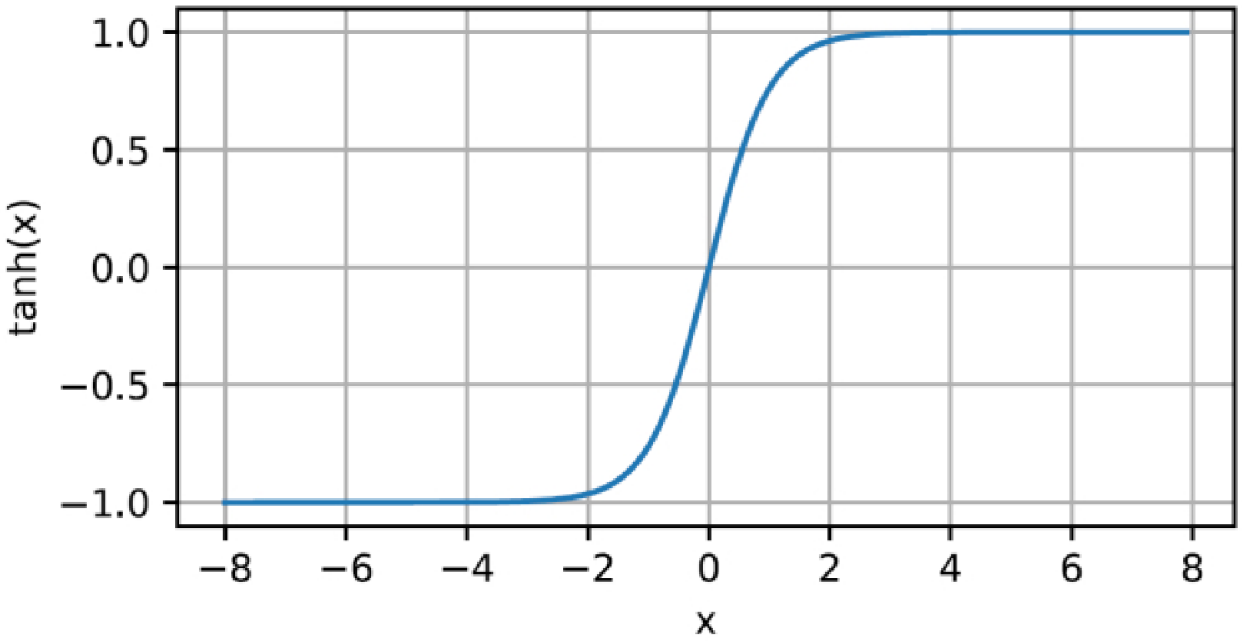

Figure 9: The tanh function.

- the derivative of the tanh function is:

$$\frac{d}{dx}\tanh(x) = 1 - \tanh^2(x).$$

- the derivative of tanh function is plotted in Figure 10
- as the input nears 0, the derivative of the tanh function approaches a maximum of 1
- and, as we saw with the sigmoid function, as the input moves away from 0 in either direction, the derivative of the tanh function approaches 0

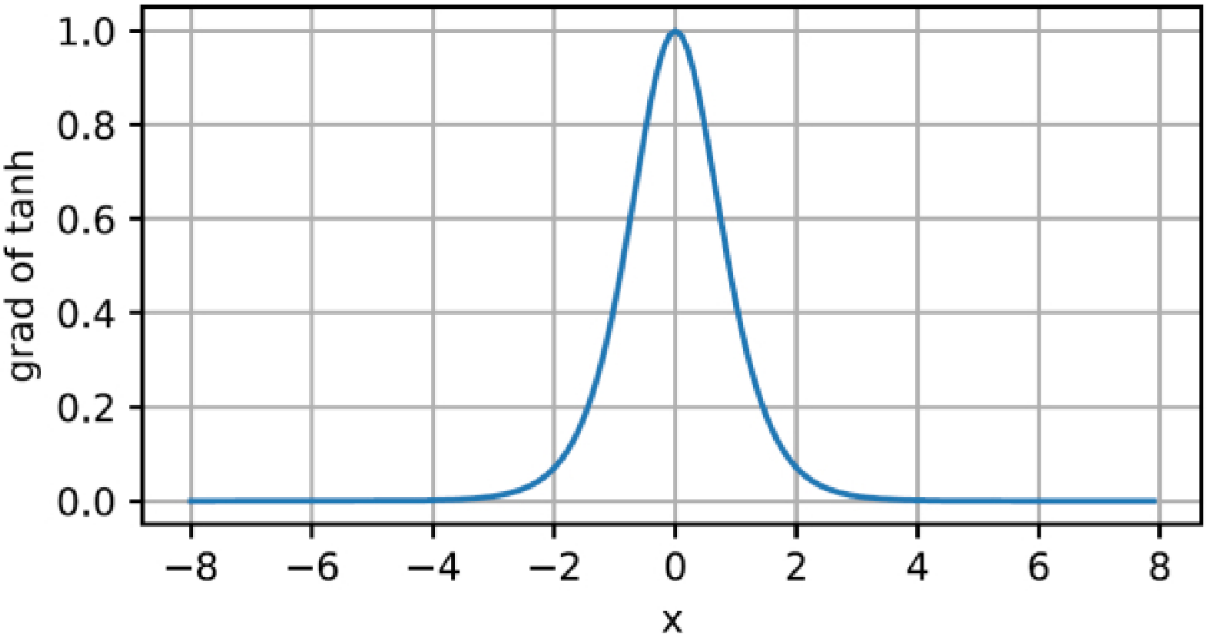

Figure 10: The derivative of the tanh function.

- in summary, we now know how to incorporate nonlinearities to build expressive multilayer neural network architectures
- as a side note, our knowledge already puts us in command of a similar toolkit to a practitioner circa 1990
- in some ways, we have an advantage over anyone working in the 1990s, because we can leverage powerful open-source deep learning frameworks to build models rapidly, using only a few lines of code
- previously, training these networks required researchers to code up thousands of lines of C and Fortran

2.3 Model selection, underfitting, and overfitting

- in machine learning, the goal is to discover *patterns*
- but how can we be sure that we have truly discovered a *general* pattern and not simply memorized our data?
- our goal is to discover patterns that capture regularities in the underlying population from which our training set was drawn
- if we are successful in this endeavor, then we could successfully assess the performance even for samples that we have never encountered before
- this problem – how to discover patterns that *generalize* – is the fundamental problem of machine learning

2.3 Model selection, underfitting, and overfitting

- the danger is that, when we train models, we access just a small sample of data
- the largest public image datasets contain roughly one million images
- more often, we must learn from only thousands or tens of thousands of data examples
- when working with finite samples, we run the risk that we might discover apparent associations that turn out not to hold up when we collect more data
- the phenomenon of fitting our training data more closely than we fit the underlying distribution is *overfitting*, and the techniques used to combat overfitting are *regularization* techniques

- in order to discuss this phenomenon more formally, we need to differentiate between training error and generalization error
- the *training error* is the error of our model as calculated on the training dataset, while *generalization error* is the expectation of our model's error were we to apply it to an infinite stream of additional data examples drawn from the same underlying data distribution as our original sample
- problematically, we can never calculate the generalization error exactly
- that is because the stream of infinite data is an imaginary object
- in practice, we must *estimate* the generalization error by applying our model to an independent test set constituted of a random selection of data examples that were withheld from our training set

2.3 Model selection, underfitting, and overfitting

- in the standard supervised learning setting, we assume that both the training data and the test data are drawn *independently* from *identical* distributions
- this is commonly called the *i.i.d. assumption*, which means that the process that samples our data has no memory
- in other words, the second example drawn and the third drawn are no more correlated than the second and the two-millionth sample drawn
- there are common cases where this assumption fails; however, we will assume that the i.i.d. assumption holds

2.3 Model selection, underfitting, and overfitting

- when we train our models, we attempt to search for a function that fits the training data as well as possible
- if the function is so flexible that it can catch on to spurious patterns just as easily as to true associations, then it might perform *too well*, without producing a model that generalizes well to unseen data
- this is precisely what we want to avoid or at least control
- many of the techniques in deep learning are heuristics and tricks aimed at guarding against overfitting

2.3 Model selection, underfitting, and overfitting

- when we have simple models and abundant data, we expect the generalization error to resemble the training error
- when we work with more complex models and fewer examples, we expect the training error to go down, but the *generalization gap* to grow
- what precisely constitutes model complexity is a complex matter
- many factors govern whether a model will generalize well
- for example, a model with *more parameters* might be considered more complex
- a model whose parameters can take a *wider range of values* might be more complex
- often with neural networks, we think of a model that takes *more training iterations* as more complex, and one subject to *early stopping* (fewer training iterations) as less complex

2.3 Model selection, underfitting, and overfitting

- it can be difficult to compare the complexity among members of substantially different model classes (say, decision trees vs. neural networks)
- for now, a simple rule of thumb is quite useful: a model that can readily explain arbitrary facts is what can be viewed as complex, whereas one that has only a limited expressive power, but still manages to explain the data well, is probably closer to the truth
- this conforms to the principle of Occam's razor: given two explanations for something, the explanation most likely to be correct is the simplest one – the one that makes fewer assumptions

- next, we will focus on a few factors that tend to influence the generalizability of a model class:
  1. The number of tunable parameters. When the number of tunable parameters, sometimes called the *degrees of freedom*, is large, models tend to be more susceptible to overfitting.
  2. The values taken by the parameters. When weights can take a wider range of values, models can be more susceptible to overfitting.
  3. The number of training examples. It is trivially easy to overfit a dataset containing only one or two examples, even if our model is simple. But overfitting a dataset with millions of examples requires an extremely flexible model.

2.3 Model selection, underfitting, and overfitting

- in machine learning, we usually select our final model after evaluating several candidate models; this process is called *model selection*
- sometimes, the models subject to comparison are fundamentally different in nature (say, decision trees vs. linear models)
- at other times, we are comparing members of the same class of models that have been trained with *different hyperparameter settings*
- with MLPs, for example, we may wish to compare models with different numbers of hidden layers, different numbers of hidden units, and various choices of the activation functions applied to each hidden layer
- in order to determine the best among our candidate models, we will typically employ a validation dataset

- in principle, we should not touch our test set until after we have chosen all our hyperparameters
- were we to use the test data in the model selection process, there is a risk that we might *overfit the test data*, which is very problematic
- if we overfit our training data, there is always the evaluation on test data to keep us honest
- but if we overfit the test data, how would we ever know?
- thus, we should never rely on the test data for model selection
- and yet we cannot rely solely on the training data for model selection either, because we cannot estimate the generalization error on the very data that we use to train the model

- the common practice to address this problem is to split our data *three ways*, incorporating a *validation set*, in addition to the training and test sets
- the performance on this validation set will be used for model selection
- when training data is scarce, we might not even be able to afford to hold out enough data to constitute a proper validation set
- one popular solution to this problem is to use *K-fold cross-validation*
- here, the original training data is split into K non-overlapping subsets
- then, model training and validation are executed K times, each time training on $K - 1$ subsets and validating on a different subset (the one not used for training in that round)
- finally, the training and validation errors are estimated by averaging over the results from the K experiments

2.3 Model selection, underfitting, and overfitting

- when we compare the training and validation errors, we should be aware of two common situations
- first, we should watch out for cases when our training error and validation error are both substantial, but there is a *little gap* between them
- if the model is unable to reduce the training error, that could mean that our model is too simple (i.e., insufficiently expressive) to capture the pattern that we are trying to model
- moreover, since the *generalization gap* between our training and validation errors is small, we have reason to believe that we could use a more complex model
- this phenomenon is known as *underfitting*

- on the other hand, as we discussed above, we should watch out for the cases when our training error is significantly lower than our validation error, indicating severe *overfitting*
- note that overfitting is not always a bad thing
- with deep learning especially, it is well known that the best predictive models often perform far better on training data than on holdout data
- ultimately, we usually care more about the validation error than about the gap between the training and validation errors

2.3 Model selection, underfitting, and overfitting

- whether we overfit or underfit can depend both on the complexity of our model and the size of the available training datasets, two topics that we discuss below
- to illustrate some classical intuition about overfitting and model complexity, we give an example using polynomials
- given training data consisting of a single feature x and a corresponding real-valued label y, we try to find the polynomial of degree d to estimate the labels y

- a higher-order polynomial function is more complex than a lower-order polynomial function, since the higher-order polynomial has more parameters and the model function's selection range is wider
- fixing the training dataset, higher-order polynomial functions should always achieve lower (at worst, equal) training error relative to lower degree polynomials
- in fact, whenever the data examples each have a distinct value of x, a polynomial function with degree equal to the number of data examples can fit the training set perfectly
- we visualize the relationship between polynomial degree and underfitting vs. overfitting in Figure 11

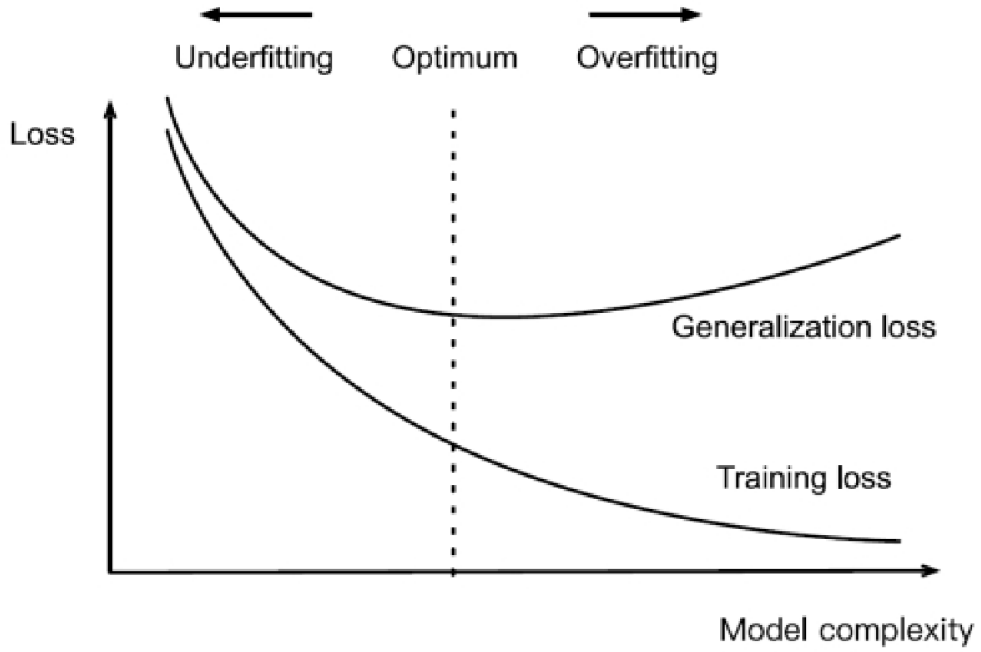

Figure 11: Influence of model complexity on underfitting and overfitting.

- the other big consideration to bear in mind is the *dataset size*
- fixing our model, the fewer samples we have in the training dataset, the more likely (and more severely) we are to encounter overfitting
- as we increase the amount of training data, the generalization error typically decreases
- moreover, in general, more data never hurts
- for a fixed task and data distribution, there is typically a relationship between model complexity and dataset size

- given more data, we might profitably attempt to fit a more complex model
- without sufficient data, simpler models may be more difficult to beat
- for many tasks, deep learning only outperforms linear models when many thousands of training examples are available
- in part, the current success of deep learning owes to the current abundance of massive datasets due to Internet companies, cheap storage, connected devices, and the broad digitization of the economy

Thank you!