Image Processing and Recognition

Dr. Călin-Adrian POPA

Lecture 4

October 25th, 2022

2.4 Weight decay

- now that we have characterized the problem of overfitting, we can introduce some standard techniques for regularizing models
- recall that we can always mitigate overfitting by going out and collecting more training data
- that can be costly, time consuming, or entirely out of our control, making it impossible in the short run
- for now, we can assume that we already have as much high-quality data as our resources permit and focus on regularization techniques

2.4 Weight decay

- *weight decay* (generally called ℓ_2 regularization), might be the most widely-used technique for regularizing parametric machine learning models
- the technique is motivated by the basic intuition that the *simplest* models are the ones for which some norm of their weight vector is as small as possible
- the most common method for ensuring a small weight vector is to add its norm as a penalty term to the problem of minimizing the loss
- thus, we replace our original objective, *minimizing the prediction loss on the training labels*, with the new objective, *minimizing the sum of the prediction loss and the penalty term*
- now, if our weight vector grows too large, our learning algorithm might focus on minimizing the weight norm vs. minimizing the training error, which is exactly what we want

2.4 Weight decay

- for example, consider the mean squared error loss given by:

$$\mathcal{L}(\boldsymbol{w}) = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{2}(\hat{y}^{(i)} - y^{(i)})^2.$$

- to penalize the size of the weight vector, we must somehow add the weight norm to the loss function, but how should the model trade off the standard loss for this new additive penalty?
- in practice, we characterize this trade-off via the *regularization constant* λ, a nonnegative hyperparameter that we fit using validation data
- for the one-hidden-layer MLP discussed above, the new objective becomes:

$$\mathcal{L}(\boldsymbol{w}) + \frac{\lambda}{2}\left(||\boldsymbol{W}^{(1)}||_F^2 + ||\boldsymbol{W}^{(2)}||_F^2\right),$$

where the Frobenius norm is a generalization for matrices of the Euclidean ℓ_2 norm, and is defined as: $||\boldsymbol{W}||_F = \sqrt{\text{Tr}(\boldsymbol{W}^\top \boldsymbol{W})}$

2.4 Weight decay

- for $\lambda = 0$, we recover our original loss function
- for $\lambda > 0$, we restrict the size of $||\boldsymbol{W}^{(1)}||_F^2 + ||\boldsymbol{W}^{(2)}||_F^2$, the norm of the weights of our network
- we divide by 2 by convention: when we take the derivative of a quadratic function, the 2 and $1/2$ cancel out, ensuring that the expression for the update looks nice and simple
- we might wonder why we work with the squared norm and not the standard norm (i.e., the Euclidean ℓ_2 norm); we do this for computational convenience
- by squaring the ℓ_2 norm, we remove the square root, leaving the sum of squares of each component of the weight vector
- this makes the derivative of the penalty easy to compute: the sum of derivatives equals the derivative of the sum

2.4 Weight decay

- now, when applying the mini-batch stochastic gradient descent algorithm, in addition to updating the weights based on the amount by which the prediction differs from the actual label, we also shrink the size of the weights $\boldsymbol{W}^{(1)}$ and $\boldsymbol{W}^{(2)}$ towards zero
- that is why the method is sometimes called "weight decay": given the penalty term alone, our optimization algorithm *decays* the weight at each step of training
- smaller values of λ correspond to less constrained weights, whereas larger values of λ constrain the weights more considerably

- we introduced the classical approach to regularizing statistical models by penalizing the ℓ_2 norm of the weights
- in probabilistic terms, we justified this technique by arguing that we have assumed a prior belief that weights take values from a Gaussian distribution with mean zero
- more intuitively, we might argue that we encouraged the model to spread out its weights among many features, rather than depending too much on a small number of potentially false associations

2.5 Dropout

- faced with more features than examples, linear models tend to overfit
- but given more examples than features, we can generally count on linear models not to overfit
- unfortunately, the reliability with which linear models generalize comes at a cost
- naively applied, linear models do not take into account interactions among features
- for every feature, a linear model must assign either a positive or a negative weight, ignoring context

- this fundamental tension between generalizability and flexibility was described as the *bias-variance trade-off*
- linear models have high bias: they can only represent a small class of functions
- however, these models have low variance: they give similar results across different random samples of the data
- deep neural networks inhabit the opposite end of the bias-variance spectrum
- unlike linear models, neural networks are not confined to looking at each feature individually
- they can learn interactions among groups of features

- even when we have far more examples than features, deep neural networks are capable of overfitting
- in 2017, a group of researchers demonstrated the extreme flexibility of neural networks by training deep nets on randomly-labeled images
- despite the absence of any true pattern linking the inputs to the outputs, they found that the neural network optimized by stochastic gradient descent could label every image in the training set perfectly
- consider what this means: if the labels are assigned uniformly at random and there are 10 classes, then no classifier can do better than 10% accuracy on holdout data
- the generalization gap here is a whopping 90%
- if our models are so expressive that they can overfit this badly, then when should we expect them not to overfit?

2.5 Dropout

- let us think briefly about what we expect from a good predictive model
- we want it to peform well on unseen data
- classical generalization theory suggests that, to close the gap between train and test performance, we should aim for a *simple* model
- simplicity can come in the form of a small number of dimensions
- additionally, as we saw when discussing weight decay (ℓ_2 regularization), the norm of the weights also represents a useful measure of simplicity
- another useful notion of simplicity is *smoothness*, i.e., that the function should not be sensitive to noise
- for instance, when we classify images, we would expect that adding some random noise to the pixels should be mostly harmless

2.5 Dropout

- thus, we could inject noise into each layer of the network, before calculating the subsequent layer during training
- when training a deep network with many layers, injecting noise enforces smoothness on the input-output mapping
- this idea, called *dropout*, involves injecting noise while computing each internal layer during forward propagation, and it has become a standard technique for training neural networks
- the method is called *dropout* because we literally *drop out* some neurons during training
- throughout training, on each iteration, standard dropout consists of zeroing out some fraction of the nodes in each layer, before calculating the subsequent layer

- we may argue that neural network overfitting is characterized by a state in which each layer relies on a specific pattern of activations in the previous layer, this condition being called *co-adaptation*
- dropout breaks up co-adaptation, by not allowing neurons to be too reliant on each other
- in standard dropout regularization, with *dropout probability* p, each intermediate activation h is replaced by a random variable h' as follows:

$$h' = \begin{cases} 0, & \text{with probability } p \\ \frac{h}{1-p}, & \text{otherwise} \end{cases}.$$

- thus, each layer is normalized by the fraction of nodes that were retained (not dropped out), and so the expectation remains unchanged, i.e., $E[h'] = h$

- recall the MLP with a hidden layer and 5 hidden units in Figure 4
- when we apply dropout to a hidden layer, zeroing out each hidden unit with probability p, the result can be viewed as a network containing only a subset of the original neurons
- in Figure 12, h_2 and h_5 are removed
- consequently, the calculation of the outputs no longer depends on h_2 or h_5, and their respective gradient also vanishes when performing backpropagation
- in this way, the calculation of the output layer cannot be overly dependent on any one element of $h_1, \ldots, h_5$

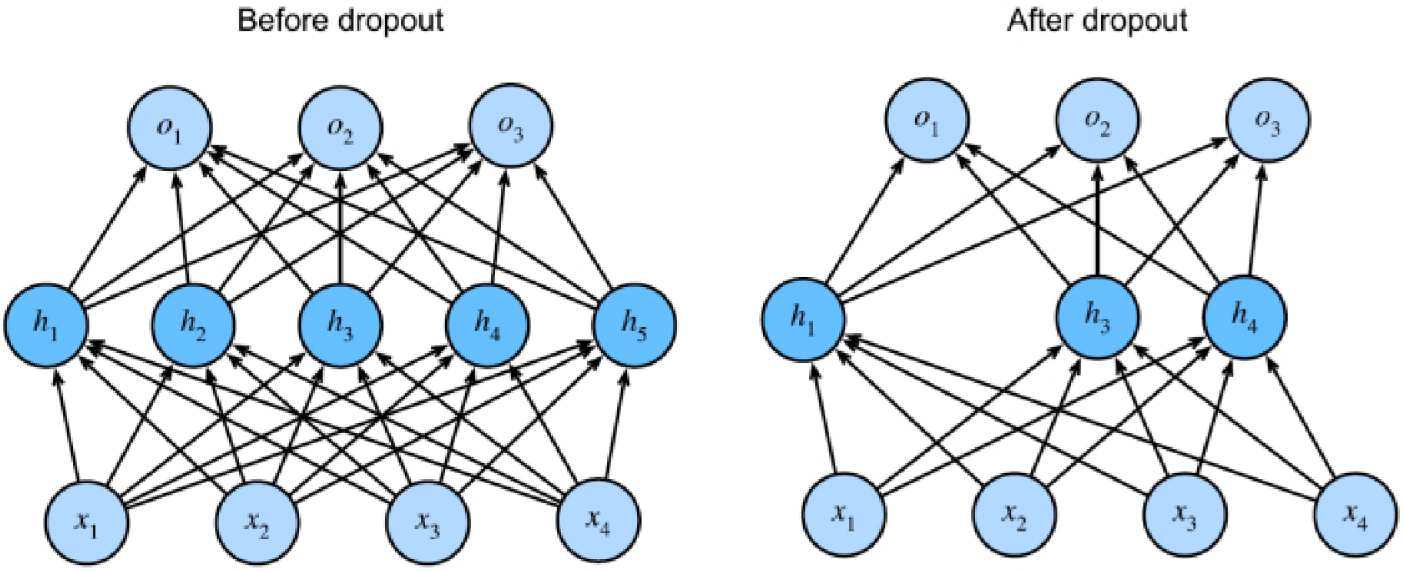

Figure 12: MLP before and after dropout.

2.5 Dropout

- dropout is disabled at test time: given a trained model and a new example, we do not drop out any nodes, and thus do not need to normalize
- to implement the dropout function for a single layer, we must draw as many samples from a Bernoulli (binary) random variable as our layer has dimensions, where the random variable takes value 1 (keep) with probability $1 - p$ and 0 (drop) with probability p
- one easy way to implement this is to first draw samples from the uniform distribution $\mathcal{U}(0, 1)$
- then we can keep those nodes for which the corresponding sample is greater than p, dropping the rest
- thus, we drop out the elements in the layer with probability p, rescaling the remainder as described above: dividing the survivors by $1 - p$

2.6 Forward propagation, backward propagation, and computational graphs

- when using modern deep learning frameworks, we only have to define the calculations involved in *forward propagation* through the model
- when it comes to calculating the gradients in order to perform gradient descent, we just invoke the backpropagation function provided by the deep learning framework
- the automatic calculation of gradients (*automatic differentiation*) profoundly simplifies the implementation of deep learning algorithms
- before automatic differentiation, even small changes to complicated models required recalculating complicated derivatives by hand
- surprisingly often, academic papers had to allocate numerous pages to deriving update rules
- while we must continue to rely on automatic differentiation so we can focus on the interesting parts, we should know how these gradients are calculated under the hood, if we want to go beyond a shallow understanding of deep learning

2.6 Forward propagation, backward propagation, and computational graphs

- in this section, we take a deep dive into the details of *backward propagation* (more commonly called *backpropagation*)
- to convey some insight for both the techniques and their implementations, we rely on some basic mathematics and computational graphs
- to start, we focus our exposition on a one-hidden-layer MLP with weight decay (ℓ_2 regularization)
- *forward propagation* (or *forward pass*) refers to the calculation and storage of *intermediate variables* (including outputs) for a neural network in order, from the input layer to the output layer

2.6 Forward propagation, backward propagation, and computational graphs

- for the sake of simplicity, let us assume that the input example is $\boldsymbol{x} \in \mathbb{R}^d$ and that our hidden layer does not include a bias term
- here, the intermediate variable is:

$$\boldsymbol{z} = \boldsymbol{W}^{(1)}\boldsymbol{x},$$

  where $\boldsymbol{W}^{(1)} \in \mathbb{R}^{h \times d}$ is the weight parameter of the hidden layer
- after running the intermediate variable $\boldsymbol{z} \in \mathbb{R}^h$ through the activation function ϕ, we obtain our hidden activation vector of length h:

$$\boldsymbol{h} = \phi(\boldsymbol{z}).$$

- the hidden variable $\boldsymbol{h} \in \mathbb{R}^h$ is also an intermediate variable
- assuming that the parameters of the output layer only possess a weight of $\boldsymbol{W}^{(2)} \in \mathbb{R}^{q \times h}$, we can obtain an output vector of length q:

$$\boldsymbol{o} = \boldsymbol{W}^{(2)}\boldsymbol{h}.$$

2.6 Forward propagation, backward propagation, and computational graphs

- assuming that the loss function is l, the squared error loss, and the example label is $\boldsymbol{y} \in \mathbb{R}^q$, we can then calculate the loss term for a single data example as:

$$L = l(\boldsymbol{o}, \boldsymbol{y}) = \frac{1}{2}(\boldsymbol{o} - \boldsymbol{y})^\top (\boldsymbol{o} - \boldsymbol{y}).$$

- according to the definition of ℓ_2 regularization, given the hyperparameter λ, the regularization term is:

$$s = \frac{\lambda}{2}\left(||\boldsymbol{W}^{(1)}||_F^2 + ||\boldsymbol{W}^{(2)}||_F^2\right), \tag{1}$$

  where the Frobenius norm is defined as: $||\boldsymbol{W}||_F = \sqrt{\text{Tr}(\boldsymbol{W}^\top \boldsymbol{W})}$
- finally, the model's regularized loss on a given data example is:

$$J = L + s.$$

- we refer to J as the *objective function* in the following discussion

2.6 Forward propagation, backward propagation, and computational graphs

- plotting *computational graphs* helps us visualize the dependencies of operators and variables within the calculation
- Figure 13 contains the graph associated with the simple network described above, where squares denote variables and circles denote operators
- the lower-left corner signifies the input and the upper-right corner is the output
- notice that the directions of the arrows (which illustrate data flow) are primarily rightward and upward

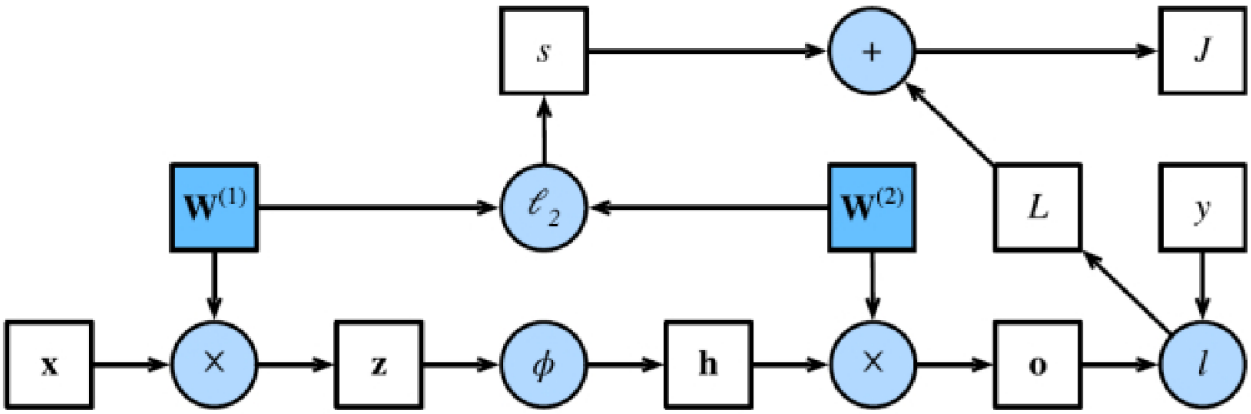

Figure 13: Computational graph of forward propagation.

2.6 Forward propagation, backward propagation, and computational graphs

- *backpropagation* refers to the method of calculating the gradient of neural network parameters
- in short, the method traverses the network in reverse order, from the output to the input layer, according to the *chain rule* from calculus
- the algorithm stores any intermediate variables (partial derivatives) required while calculating the gradient with respect to some parameters
- assume that we have functions $\mathsf{Y} = f(\mathsf{X})$ and $\mathsf{Z} = g(\mathsf{Y})$, in which the input and the output $\mathsf{X}, \mathsf{Y}, \mathsf{Z}$ are scalars, vectors, or matrices of arbitrary shapes

2.6 Forward propagation, backward propagation, and computational graphs

- by using the chain rule, we can compute the derivative of Z with respect to X via the chain rule as:

$$\frac{\partial Z}{\partial X} = \text{prod}\left(\frac{\partial Z}{\partial Y}, \frac{\partial Y}{\partial X}\right).$$

- here, we use the prod operator to multiply its arguments after the necessary operations, such as transposition and swapping input positions, have been carried out
- for vectors, this is straightforward: it is simply matrix-matrix multiplication
- for matrices, we use the appropriate counterpart
- the operator prod hides all the notation overhead

2.6 Forward propagation, backward propagation, and computational graphs

- recall that the parameters of the simple network with one hidden layer, whose computational graph is in Figure 13, are $\boldsymbol{W}^{(1)}$ and $\boldsymbol{W}^{(2)}$
- the objective of backpropagation is to calculate the gradients $\frac{\partial J}{\partial \boldsymbol{W}^{(1)}}$ and $\frac{\partial J}{\partial \boldsymbol{W}^{(2)}}$
- to accomplish this, we apply the chain rule and calculate, in turn, the gradient of each intermediate variable and parameter
- the order of calculations are reversed relative to those performed in forward propagation, since we need to start with the outcome of the computational graph and work our way towards the parameters

2.6 Forward propagation, backward propagation, and computational graphs

- we start by calculating the gradient $\frac{\partial J}{\partial \boldsymbol{W}^{(2)}} \in \mathbb{R}^{q \times h}$ of the model parameters closest to the output layer
- first, we can write that:

$$\begin{aligned} \frac{\partial J}{\partial \boldsymbol{W}^{(2)}} &= \frac{\partial (L + s)}{\partial \boldsymbol{W}^{(2)}} \\ &= \frac{\partial L}{\partial \boldsymbol{W}^{(2)}} + \frac{\partial s}{\partial \boldsymbol{W}^{(2)}}. \end{aligned} \tag{2}$$

- using the chain rule for the first term in (2) yields:

$$\begin{aligned} \frac{\partial L}{\partial \boldsymbol{W}^{(2)}} &= \text{prod}\left(\frac{\partial L}{\partial \boldsymbol{o}}, \frac{\partial \boldsymbol{o}}{\partial \boldsymbol{W}^{(2)}}\right) \\ &= \text{prod}\left(\frac{\partial L}{\partial \boldsymbol{o}}, \frac{\partial (\boldsymbol{W}^{(2)} \boldsymbol{h})}{\partial \boldsymbol{W}^{(2)}}\right) \\ &= \frac{\partial L}{\partial \boldsymbol{o}} \boldsymbol{h}^{\top}. \end{aligned} \tag{3}$$

2.6 Forward propagation, backward propagation, and computational graphs

- now, we just have to compute the gradient of the loss function L with respect to the variable of the output layer $\boldsymbol{o}$:

$$\begin{aligned}\frac{\partial L}{\partial \boldsymbol{o}} &= \frac{\partial}{\partial \boldsymbol{o}}\left(\frac{1}{2}(\boldsymbol{o}-\boldsymbol{y})^{\top}(\boldsymbol{o}-\boldsymbol{y})\right)\\ &= \boldsymbol{o}-\boldsymbol{y}.\end{aligned}$$

- thus, (3) becomes:

$$\frac{\partial L}{\partial \boldsymbol{W}^{(2)}} = (\boldsymbol{o}-\boldsymbol{y})\boldsymbol{h}^{\top}.$$

- we next calculate the gradient of the regularization term s with respect to $\boldsymbol{W}^{(2)}$:

$$\frac{\partial s}{\partial \boldsymbol{W}^{(2)}} = \lambda \boldsymbol{W}^{(2)}.$$

- putting it all together, (2) becomes:

$$\frac{\partial J}{\partial \boldsymbol{W}^{(2)}} = (\boldsymbol{o}-\boldsymbol{y})\boldsymbol{h}^{\top} + \lambda \boldsymbol{W}^{(2)}.$$

2.6 Forward propagation, backward propagation, and computational graphs

- in order to obtain the gradient $\frac{\partial J}{\partial \boldsymbol{W}^{(1)}} \in \mathbb{R}^{h \times d}$ of the model parameters closest to the input layer, we first write that:

$$\begin{aligned} \frac{\partial J}{\partial \boldsymbol{W}^{(1)}} &= \frac{\partial (L + s)}{\partial \boldsymbol{W}^{(1)}} \\ &= \frac{\partial L}{\partial \boldsymbol{W}^{(1)}} + \frac{\partial s}{\partial \boldsymbol{W}^{(1)}}. \end{aligned} \tag{4}$$

- for the first term in (4), according to the chain rule, we get:

$$\begin{aligned} \frac{\partial L}{\partial \boldsymbol{W}^{(1)}} &= \text{prod}\left(\frac{\partial L}{\partial \boldsymbol{z}}, \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{W}^{(1)}}\right) \\ &= \text{prod}\left(\frac{\partial L}{\partial \boldsymbol{z}}, \frac{\partial (\boldsymbol{W}^{(1)} \boldsymbol{x})}{\partial \boldsymbol{W}^{(1)}}\right) \\ &= \frac{\partial L}{\partial \boldsymbol{z}} \boldsymbol{x}^{\top}. \end{aligned}$$

- since the activation function ϕ applies element-wise, calculating the gradient $\frac{\partial L}{\partial \boldsymbol{z}} \in \mathbb{R}^h$ of the intermediate variable $\boldsymbol{z}$ requires that we use the element-wise matrix multiplication operator, which we denote by $\odot$:

$$\begin{aligned} \frac{\partial L}{\partial \boldsymbol{z}} &= \text{prod}\left(\frac{\partial L}{\partial \boldsymbol{h}}, \frac{\partial \boldsymbol{h}}{\partial \boldsymbol{z}}\right) \\ &= \text{prod}\left(\frac{\partial L}{\partial \boldsymbol{h}}, \frac{\partial \phi(\boldsymbol{z})}{\partial \boldsymbol{z}}\right) \\ &= \frac{\partial L}{\partial \boldsymbol{h}} \odot \phi'(\boldsymbol{z}). \end{aligned} \tag{5}$$

- the gradient with respect to the hidden layer's outputs $\frac{\partial L}{\partial \boldsymbol{h}} \in \mathbb{R}^h$ is given by the chain rule:

$$\begin{aligned}
\frac{\partial L}{\partial \boldsymbol{h}} &= \text{prod}\left(\frac{\partial L}{\partial \boldsymbol{o}}, \frac{\partial \boldsymbol{o}}{\partial \boldsymbol{h}}\right) \\
&= \text{prod}\left(\frac{\partial L}{\partial \boldsymbol{o}}, \frac{\partial (\boldsymbol{W}^{(2)} \boldsymbol{h})}{\partial \boldsymbol{h}}\right) \\
&= \boldsymbol{W}^{(2)\top} \frac{\partial L}{\partial \boldsymbol{o}} \\
&= \boldsymbol{W}^{(2)\top} (\boldsymbol{o} - \boldsymbol{y}).
\end{aligned}$$

- now, (5) becomes:

$$\frac{\partial L}{\partial \boldsymbol{z}} = \left(\boldsymbol{W}^{(2)\top}(\boldsymbol{o} - \boldsymbol{y})\right) \odot \phi'(\boldsymbol{z}).$$

- we similarly as before calculate the gradient of the regularization term s with respect to $\boldsymbol{W}^{(1)}$:

$$\frac{\partial s}{\partial \boldsymbol{W}^{(1)}} = \lambda \boldsymbol{W}^{(1)}.$$

- putting it all together, (4) is finally given as:

$$\frac{\partial J}{\partial \boldsymbol{W}^{(1)}} = \left(\left(\boldsymbol{W}^{(2)\top}(\boldsymbol{o} - \boldsymbol{y})\right) \odot \phi'(\boldsymbol{z})\right) \boldsymbol{x}^{\top} + \lambda \boldsymbol{W}^{(1)}.$$

2.6 Forward propagation, backward propagation, and computational graphs

- thus, we take the following steps to compute $\frac{\partial J}{\partial \boldsymbol{W}^{(2)}}$:

1. Compute $\frac{\partial L}{\partial \boldsymbol{o}} = \boldsymbol{o} - \boldsymbol{y}$
2. Compute $\frac{\partial L}{\partial \boldsymbol{W}^{(2)}} = \frac{\partial L}{\partial \boldsymbol{o}} \boldsymbol{h}^\top$
3. Compute $\frac{\partial s}{\partial \boldsymbol{W}^{(2)}} = \lambda \boldsymbol{W}^{(2)}$
4. Compute $\frac{\partial J}{\partial \boldsymbol{W}^{(2)}} = \frac{\partial L}{\partial \boldsymbol{W}^{(2)}} + \frac{\partial s}{\partial \boldsymbol{W}^{(2)}}$

- and the following steps to compute $\frac{\partial J}{\partial \boldsymbol{W}^{(1)}}$:

1. Compute $\frac{\partial L}{\partial \boldsymbol{h}} = \boldsymbol{W}^{(2)\top} \frac{\partial L}{\partial \boldsymbol{o}}$ ($\frac{\partial L}{\partial \boldsymbol{o}}$ is already computed in step 1 above)
2. Compute $\frac{\partial L}{\partial \boldsymbol{z}} = \frac{\partial L}{\partial \boldsymbol{h}} \odot \phi'(\boldsymbol{z})$
3. Compute $\frac{\partial L}{\partial \boldsymbol{W}^{(1)}} = \frac{\partial L}{\partial \boldsymbol{z}} \boldsymbol{x}^\top$
4. Compute $\frac{\partial s}{\partial \boldsymbol{W}^{(1)}} = \lambda \boldsymbol{W}^{(1)}$
5. Compute $\frac{\partial J}{\partial \boldsymbol{W}^{(1)}} = \frac{\partial L}{\partial \boldsymbol{W}^{(1)}} + \frac{\partial s}{\partial \boldsymbol{W}^{(1)}}$

2.6 Forward propagation, backward propagation, and computational graphs

- when training neural networks, forward and backward propagation depend on each other
- in particular, for forward propagation, we traverse the computational graph in the direction of dependencies, and compute all the variables on its path
- these are then used for backpropagation, where the compute order on the graph is reversed
- take the aforementioned simple network as an example to illustrate
- on one hand, computing the regularization term (1) during forward propagation depends on the current values of model parameters $\boldsymbol{W}^{(1)}$ and $\boldsymbol{W}^{(2)}$
- they are given by the optimization algorithm according to backpropagation, in the latest iteration
- on the other hand, the gradient calculation for the parameter (3) during backpropagation depends on the current value of the hidden variable $\boldsymbol{h}$, which is given by forward propagation

2.6 Forward propagation, backward propagation, and computational graphs

- therefore, when training neural networks, after model parameters are *initialized*, we alternate *forward propagation* with *backpropagation*, *updating* model parameters using *gradients* given by backpropagation
- note that backpropagation reuses the stored intermediate values from forward propagation to avoid duplicate calculations
- one of the consequences is that we need to retain the intermediate values until backpropagation is complete
- this is also one of the reasons why training requires significantly more memory than plain prediction
- besides, the size of such intermediate values is roughly proportional to the number of network layers and the batch size
- thus, training deeper networks using larger batch sizes more easily leads to out of memory errors

2.7 Numerical stability and initialization

- thus far, every model that we discussed requires that we initialize its parameters according to some pre-specified distribution
- until now, we took the initialization scheme for granted, and didn't discuss the details of how these choices are made
- we might have even gotten the impression that these choices are not especially important
- to the contrary, the choice of initialization scheme plays a significant role in neural network learning, and it can be crucial for maintaining numerical stability

2.7 Numerical stability and initialization

- moreover, these choices can be tied up in interesting ways with the choice of the nonlinear activation function
- which function we choose and how we initialize parameters can determine how quickly our optimization algorithm converges
- poor choices here can cause us to encounter exploding or vanishing gradients while training
- in this section, we delve into these topics with greater detail and discuss some useful heuristics

- consider a deep network with L layers, input $\boldsymbol{x}$ and output $\boldsymbol{o}$
- with each layer l defined by a transformation f_l parameterized by weights $\boldsymbol{W}^{(l)}$, whose hidden variable is $\boldsymbol{h}^{(l)}$ (let $\boldsymbol{h}^{(0)} = \boldsymbol{x}$), our network can be expressed as:

$$\boldsymbol{h}^{(l)} = f_l(\boldsymbol{h}^{(l-1)}) \text{ and thus } \boldsymbol{o} = f_L \circ \ldots \circ f_1(\boldsymbol{x}),$$

  where $\circ$ denotes function composition: $f \circ g(\boldsymbol{x}) = f(g(\boldsymbol{x}))$
- if all the hidden variables and the input are vectors, we can write the gradient of $\boldsymbol{o}$ with respect to any set of parameters $\boldsymbol{W}^{(l)}$ as follows:

$$\frac{\partial \boldsymbol{o}}{\partial \boldsymbol{W}^{(l)}} = \underbrace{\frac{\partial \boldsymbol{h}^{(L)}}{\partial \boldsymbol{h}^{(L-1)}}}_{\boldsymbol{M}^{(L)} \stackrel{\text{def}}{=}} \cdot \ldots \cdot \underbrace{\frac{\partial \boldsymbol{h}^{(l+1)}}{\partial \boldsymbol{h}^{(l)}}}_{\boldsymbol{M}^{(l+1)} \stackrel{\text{def}}{=}} \cdot \underbrace{\frac{\partial \boldsymbol{h}^{(l)}}{\partial \boldsymbol{W}^{(l)}}}_{\boldsymbol{v}^{(l)} \stackrel{\text{def}}{=}}.$$

- in other words, this gradient is the product of $L - l$ matrices $\boldsymbol{M}^{(L)} \cdot \ldots \cdot \boldsymbol{M}^{(l+1)}$ and the gradient vector $\boldsymbol{v}^{(l)}$
- thus, we are susceptible to the same problems of numerical underflow that often appear when multiplying together too many probabilities
- when dealing with probabilities, a common trick is to switch into log-space, i.e., shifting pressure from the mantissa to the exponent of the numerical representation
- unfortunately, our problem above is more serious: initially, the matrices $\boldsymbol{M}^{(l)}$ may have a wide variety of eigenvalues
- they might be small or large, and their product might be *very large* or *very small*

2.7 Numerical stability and initialization

- the risks posed by unstable gradients go beyond numerical representation
- gradients of unpredictable magnitude also threaten the stability of our optimization algorithms
- we may be facing parameter updates that are either:
  - excessively large, destroying our model (the *exploding gradient* problem);
  - excessively small (the *vanishing gradient* problem), rendering learning impossible as parameters hardly move on each update.
- the vanishing gradient problem is often caused by the choice of the activation function that is appended following each layer's linear operations
- historically, the sigmoid function $1/(1 + \exp(-x))$ was popular, because it resembles a thresholding function
- since early artificial neural networks were inspired by biological neural networks, the idea of neurons that fire either *fully* or *not at all* (like biological neurons) seemed appealing
- let us take a closer look at the sigmoid to see why it can cause vanishing gradients

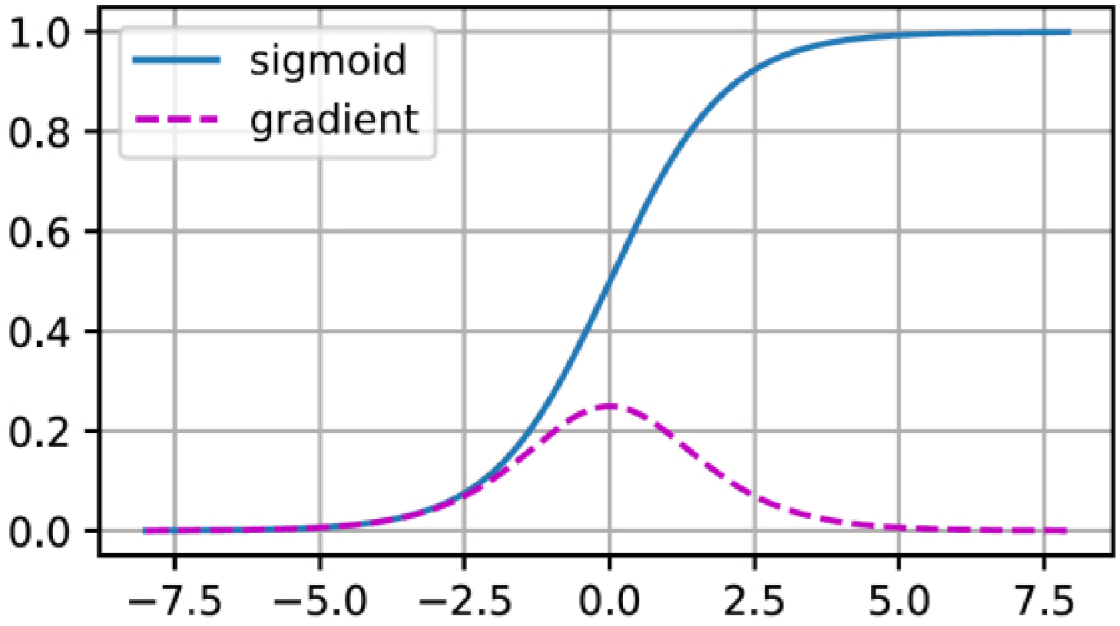

Figure 14: The sigmoid function and its derivative.

2.7 Numerical stability and initialization

- as we can see, the sigmoid's derivative vanishes both when its inputs are large and when they are small
- moreover, when backpropagating through many layers (unless we are in the zone where the input to the sigmoid is close to zero), the gradients of the overall product may vanish
- when our network has many layers, unless we are careful, the gradient will likely be cut off at some layer
- indeed, this problem used to plague deep network training
- consequently, ReLUs, which are more stable (but less neurally plausible), have emerged as the default choice today

- the opposite problem, when gradients explode, can be similarly troublesome
- to illustrate this a bit better, we can draw 100 Gaussian random matrices and multiply them with some initial matrix
- if we choose the variance $\sigma^2 = 1$, we will see that the matrix product explodes
- when this happens due to the initialization of a deep network, we have no chance of getting a gradient descent optimizer to converge

2.7 Numerical stability and initialization

- another problem in neural network design is the symmetry inherent in their parametrization
- assume that we have a simple MLP with one hidden layer and two units
- in this case, we could permute the weights $\boldsymbol{W}^{(1)}$ of the first layer and likewise permute the weights of the output layer to obtain the same function
- there is nothing special differentiating the first hidden unit vs. the second hidden unit
- in other words, we have permutation symmetry among the hidden units of each layer

2.7 Numerical stability and initialization

- this is more than just a theoretical problem
- consider a one-hidden-layer MLP with two hidden units
- for illustration, suppose that the output layer transforms the two hidden units into only one output unit
- imagine what would happen if we initialized all of the parameters of the hidden layer as $\boldsymbol{W}^{(1)} = c$, for some constant c
- in this case, during forward propagation, either hidden unit takes the same inputs and parameters, producing the same activation, which is fed to the output unit

- during backpropagation, differentiating the output unit with respect to parameters $\boldsymbol{W}^{(1)}$ gives a gradient whose elements all take the same value
- thus, after gradient-based iteration (e.g., mini-batch stochastic gradient descent), all the elements of $\boldsymbol{W}^{(1)}$ still take the same value
- such iterations would never break the symmetry on their own, and we might never be able to realize the network's expressive power
- the hidden layer would behave as if it had only a single unit
- note that, while mini-batch stochastic gradient descent would not break this symmetry, dropout regularization would

- one way of addressing – or at least mitigating – the issues raised above is through careful initialization
- additional care during optimization and suitable regularization can further enhance stability
- let us look at the scale distribution of an output (e.g., a hidden variable) o_i for some fully-connected layer *without nonlinearities*
- with n_{in} inputs x_j and their associated weights w_{ij} for this layer, an output is given by:

$$o_i = \sum_{j=1}^{n_{\text{in}}} w_{ij} x_j.$$

- the weights w_{ij} are all drawn independently from the same distribution
- furthermore, let us assume that this distribution has zero mean and variance σ^2
- note that this does not mean that the distribution has to be Gaussian, just that the mean and variance need to exist
- for now, let us assume that the inputs to the layer x_j also have zero mean and variance γ^2, and that they are independent of w_{ij} and independent of each other
- in this case, we can compute the mean and variance of o_i as follows:

$$
\begin{aligned}
E[o_i] &= \sum_{j=1}^{n_{\text{in}}} E[w_{ij} x_j] \\
&= \sum_{j=1}^{n_{\text{in}}} E[w_{ij}] E[x_j] \\
&= 0, \\
\text{Var}[o_i] &= E[o_i^2] - (E[o_i])^2 \\
&= \sum_{j=1}^{n_{\text{in}}} E[w_{ij}^2 x_j^2] - 0 \\
&= \sum_{j=1}^{n_{\text{in}}} E[w_{ij}^2] E[x_j^2] \\
&= n_{\text{in}} \sigma^2 \gamma^2.
\end{aligned}
$$

- one way to keep the variance fixed is to set $n_{\text{in}}\sigma^2 = 1$; now consider backpropagation
- there, we face a similar problem, although with gradients being propagated from the layers closer to the output
- using the same reasoning as for forward propagation, we see that the gradients' variance can explode, unless $n_{\text{out}}\sigma^2 = 1$, where n_{out} is the number of outputs of this layer
- this leaves us in a dilemma: we cannot possibly satisfy both conditions simultaneously
- instead, we simply try to satisfy:

$$\frac{1}{2}(n_{\text{in}} + n_{\text{out}})\sigma^2 = 1 \text{ or, equivalently, } \sigma = \sqrt{\frac{2}{n_{\text{in}} + n_{\text{out}}}}.$$

- this is the reasoning underlying the now-standard and practically beneficial *Xavier initialization*, named after the first author of its creators (Xavier Glorot)
- typically, the Xavier initialization samples weights from a Gaussian distribution with zero mean and variance $\sigma^2 = \frac{2}{n_{\text{in}}+n_{\text{out}}}$
- we can also adapt Xavier's intuition to choose the variance when sampling weights from a uniform distribution
- note that the uniform distribution $\mathcal{U}(-a, a)$ has variance $\frac{a^2}{3}$
- plugging $\frac{a^2}{3}$ into our condition on σ^2 yields the suggestion to initialize according to:

$$\mathcal{U}\left(-\sqrt{\frac{6}{n_{\text{in}}+n_{\text{out}}}}, \sqrt{\frac{6}{n_{\text{in}}+n_{\text{out}}}}\right).$$

- though the assumption for nonexistence of nonlinearities in the above mathematical reasoning can be easily violated in neural networks, the Xavier initialization method turns out to work well in practice

2.8 Deep learning computation

- alongside giant datasets and powerful hardware, great software tools have played an indispensable role in the rapid progress of deep learning
- starting with the groundbreaking *Theano* library released in 2007, flexible open-source tools have enabled researchers to rapidly prototype models, avoiding repetitive work when recycling standard components, while still maintaining the ability to make low-level modifications
- over time, deep learning's libraries have evolved to offer increasingly coarse abstractions
- just as semiconductor designers went from specifying transistors to logical circuits to writing code, neural networks researchers have moved from thinking about the behavior of individual artificial neurons to conceiving of networks in terms of whole layers, and now often design architectures with far coarser *blocks* in mind

2.8 Deep learning computation

- when we introduced neural networks, we focused on linear models with a single output
- here, the entire model consists of just a single neuron
- note that a single neuron (i) takes some set of inputs; (ii) generates a corresponding scalar output; and (iii) has a set of associated parameters that can be updated to optimize some loss function of interest
- then, once we started thinking about networks with multiple outputs, we leveraged vectorized arithmetic to characterize an entire layer of neurons
- just like individual neurons, layers (i) take a set of inputs, (ii) generate corresponding outputs, and (iii) are described by a set of tunable parameters
- when we worked through softmax regression, a single layer was itself the model
- however, even when we subsequently introduced MLPs, we could still think of the model as retaining this same basic structure

- interestingly, for MLPs, both the entire model and its constituent layers share this structure
- the entire model takes in raw inputs (the features), generates outputs (the predictions), and possesses parameters (the combined parameters from all constituent layers)
- likewise, each individual layer ingests inputs (supplied by the previous layer), generates outputs (the inputs to the subsequent layer), and possesses a set of tunable parameters that are updated according to the signal that flows backwards from the subsequent layer

2.8 Deep learning computation

- while we might think that neurons, layers, and models give us enough abstractions, it turns out that we often find it convenient to speak about components that are larger than an individual layer, but smaller than the entire model
- for example, the ResNet-152 architecture, which is wildly popular in computer vision, possesses hundreds of layers
- these layers consist of repeating patterns of groups of layers; implementing such a network one layer at a time can grow tedious
- this concern is not just hypothetical – such design patterns are common in practice
- the ResNet architecture, mentioned above, won the 2015 ImageNet and COCO computer vision competitions for both recognition and detection and remains a go-to architecture for many vision tasks
- similar architectures, in which layers are arranged in various repeating patterns, are now ubiquitous in other domains, including natural language processing and speech

2.8 Deep learning computation

- to implement these complex networks, we introduce the concept of a neural network *block*
- a block could describe a single layer, a component consisting of multiple layers, or the entire model itself
- one benefit of working with the block abstraction is that they can be combined into larger artifacts, often recursively
- this is illustrated in Figure 15
- by defining code to generate blocks of arbitrary complexity on demand, we can write surprisingly compact code and still implement complex neural networks

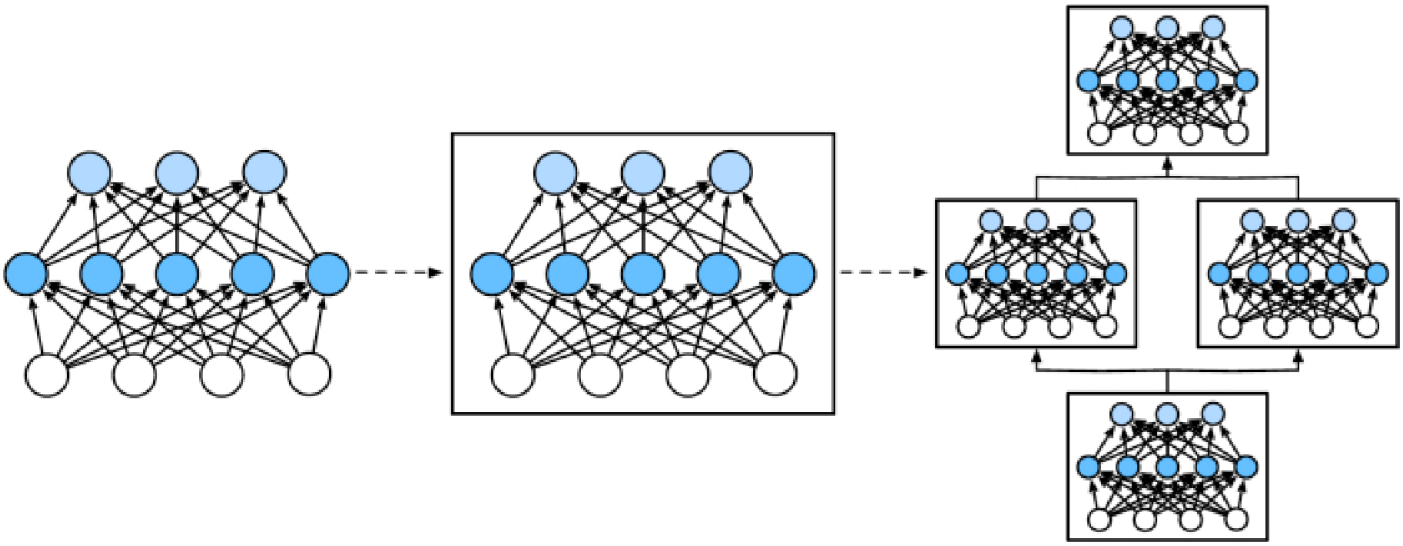

Figure 15: Multiple layers are combined into blocks, forming repeating patterns of larger models.

- from a programming point of view, a block is represented by a *class*
- any subclass of it must define a forward propagation function that transforms its input into output and must store any necessary parameters
- note that some blocks do not require any parameters at all
- finally, a block must possess a backpropagation function, for purposes of calculating gradients
- fortunately, due to the auto differentiation feature present in deep learning frameworks, when defining our own block, we only need to worry about parameters and the forward propagation function

Thank you!