DS 102 Data, Inference, and Decisions **Spring 2020**

Lecture 25: Neural Networks

Lecturer: Ramesh Sridharan

In the current lecture, we will discuss a specific nonparametric approach to classification and regression problems that has rapidly gained popularity over the last few years: neural networks. In particular, we will focus on understanding the underpinnings of how to train neural networks and some of the main advances that have allowed neural networks to be successful.

25.1 Review: Regression and Classification

In past lectures, we have discussed methods for solving both regression and classification problems. In either setting, we have observed data x with labels y, and we will denote our method's predictions by $\hat{y}$.

One of the first models we discussed was linear regression, where $\hat{y} = Wx + b$. Here, x can be a matrix (as in multivariate linear regression) and W is some matrix or vector of weights. We also discussed logistic regression for doing classification. In logistic regression, $\hat{y} = \text{round}(\sigma(Wx + b))$ where $\sigma(\cdot)$ denotes the sigmoid function. Both linear regression and logistic regression are *parametric methods* because they specify the full underlying model of how y is obtained from the input x—for example, linear regression presumes that there is an underlying linear relationship used by "nature" to get from x to y that generated our observed data.

Decision trees are an example of a *nonparametric* approach to regression and classification problems. In this approach, a binary tree is constructed where at each node the decision to go left or right is based on some feature of the data. Eventually one of k leaf nodes is reached, and the final decision is based on the labels associated to the training data associated with that leaf—for example, we could take a majority vote or an average over the training data corresponding to the given leaf. Mathematically, we can express this as $\hat{y} = \sum_k W_k \mathbb{1}(x \in A_k)$ where A_k indicates the partition of our data associated with the k^{th} leaf node. This approach is nonparametric because the resulting function is defined arbitrarily based on the partition of the training data into leaf nodes.

In general, these methods—linear regression, logistic regression, and decision trees—are able to model an increasingly complex mapping between x and y.

25.2 Neural Networks

Linear regression is nice because linear relationships are simple, and fitting them is computationally efficient. Can we use linear regression as a building block to design more complex models?

An initial idea might be to do two phases of linear regression, so that

$$\hat{y} = W_2(W_1x + b_1) + b_2.$$

It turns out, however, that this is not helpful when everything is linear; we can simplify this model to

$$\hat{y} = W_2W_1x + W_2b_1 + b_2$$

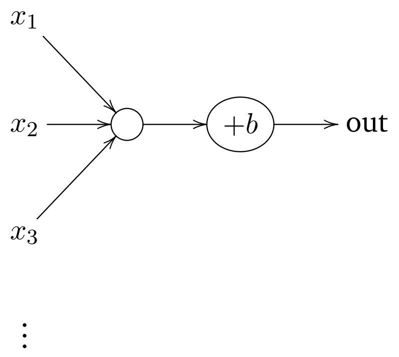
which is still a linear model. Thus, this model has the same complexity and utility as doing linear regression directly.

To improve upon this idea, we can introduce a nonlinearity (for example, a sigmoid, hyperbolic tangent, or rectified linear unit). For instance,

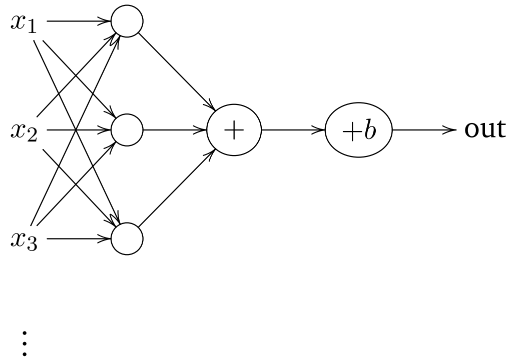
$$\hat{y} = W_2\sigma(W_1x + b_1) + b_2$$

is a two-layer neural network with a sigmoid activation. We could imagine doing this more than twice for a multi-layer network, but today will focus on the two-layer case.

It can be helpful to represent our models graphically to compare and contrast them. For example, here are the graphical representations for linear regression with a single output

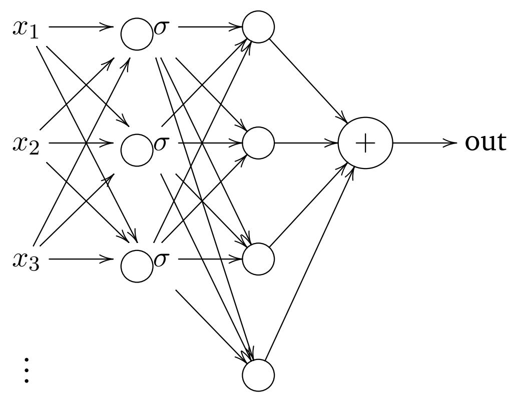


and with multiple (intermediate) outputs



where each weight is represented by a circle.

The graphical representation for our simple two-layer neural network is:



where we have dropped the $+b$ nodes for simplicity.

25.3 Fitting Network Parameters with Gradient Descent

In the case of linear regression, we can derive a simple closed-form expression for the model parameters by taking the derivative and setting it equal to zero. This may not always be possible in an arbitrary model. Gradient descent is one of the simplest and most powerful approaches to solving optimization problems for model parameters.

Recall, to do gradient descent we must define some loss function $\ell(\theta, X)$ that depends on the model parameters, θ, and the observed data, X. We begin with some initial guess for the parameters, $\theta^{(0)}$, and for each iteration t we update the parameters according to

$$\theta^{(t)} = \theta^{(t-1)} - \alpha \nabla_\theta \ell(\theta, X)$$

where α is the learning rate or step size.

Further recall that the gradient $\nabla_\theta \ell(\theta, X)$ is a vector of partial derivatives with respect to the parameters. For example, for our two-layer neural network $\theta = (W_1, W_2, b_1, b_2)$, and

$$\nabla_\theta \ell = \begin{bmatrix} \frac{\partial \ell}{\partial W_1} \\ \frac{\partial \ell}{\partial W_2} \\ \frac{\partial \ell}{\partial b_1} \\ \frac{\partial \ell}{\partial b_2} \end{bmatrix}$$

In reality, our loss ℓ is usually an average over our entire dataset X, and the dataset is too large to efficiently compute the entire gradient at once. Usually, people instead use stochastic gradient descent, where on each iteration the gradient is computed with respect to a single mini-batch from the training data.

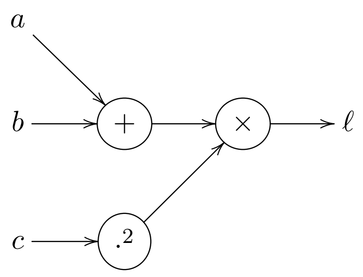
25.4 Backpropagation

In order to use (stochastic) gradient descent to fit the parameters of our neural network, we must be able to take all of the partial derivatives. It turns out that if we compute each partial derivative independently using the chain rule, we are doing a lot of repeated computation. To avoid doing such repeated work, we can use dynamic programming to share pieces of the computation.

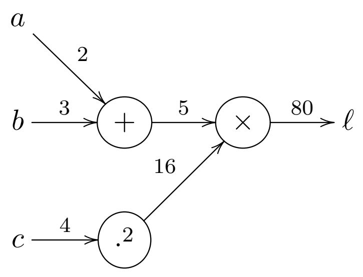
Suppose we have the simple loss function $\ell(a, b, c) = (a + b)c^2$. Let $q = (a + b)$ and $r = c^2$. Then, we can compute each partial derivative using the chain rule:

$$\nabla \ell = \begin{bmatrix} \frac{\partial \ell}{\partial a} = \frac{\partial \ell}{\partial q}\frac{\partial q}{\partial a} = c^2 \\ \frac{\partial \ell}{\partial b} = \frac{\partial \ell}{\partial q}\frac{\partial q}{\partial b} = c^2 \\ \frac{\partial \ell}{\partial c} = \frac{\partial \ell}{\partial r}\frac{\partial r}{\partial c} = 2(a+b)c \end{bmatrix}$$

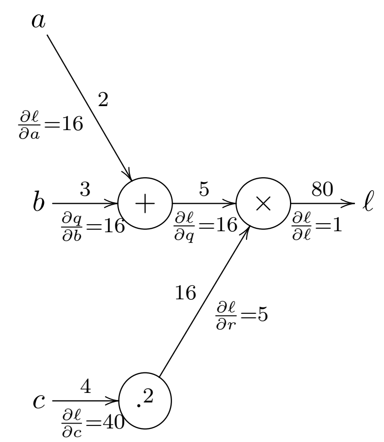
To see how we could have shared results between these computations, it helps to write out our loss as a computation graph:



Using this computation graph, we can define a process that lets us compute both the loss and the gradient efficiently, given numerical values for a, b, and c. Suppose $a = 2, b = 3, c = 4$. By making a forward pass through the computation graph, we can compute $\ell = 80$:



By making a backward pass through the computation graph, we can compute the partial derivatives efficiently:


a
2
$\frac{\partial \ell}{\partial a}=16$
b
3
$\frac{\partial q}{\partial b}=16$
$+$
5
$\frac{\partial \ell}{\partial q}=16$
$\times$
80
$\frac{\partial \ell}{\partial \ell}=1$
ℓ
16
$\frac{\partial \ell}{\partial r}=5$
c
4
$\frac{\partial \ell}{\partial c}=40$
$.^2$

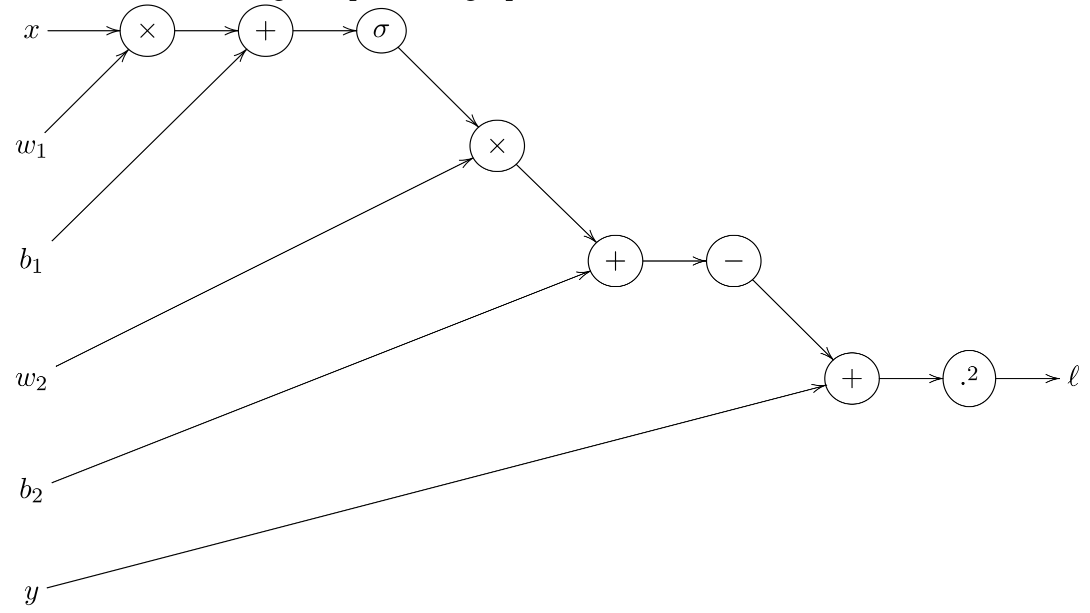
This is called **backpropagation**, and it is a clever way of taking several partial derivatives in such a way that we can reuse computation and avoid long computations.

Another important observation about this approach is that all of the gradient computations can be done automatically as long as each operation in the computation graph "knows" how to compute its forward operation *and* how to compute its gradient (given numerical inputs). This enables **automatic differentiation**, which lets us build programs and systems that can do differentiation completely automatically. This is the idea behind a lot of the powerful neural network libraries and toolkits: they automatically find the computation graph from code, and then can efficiently do a backward pass along this computation graph to compute gradients when fed data.

For our two-layer neural network under the squared loss,

$$\ell(\theta, y) = (y - W_2\sigma(W_1x + b_1) + b_2)^2,$$

we have the following computation graph:



As long as we know how to take derivatives of each of the building blocks (in this case, addition, negation, squaring, and sigmoid operations), we can do backpropagation. This saves us from having to write out any computational or analytical derivatives, and lets us do the computation using a program much more efficiently than we would if we just naively used the chain rule.

25.5 Strengths and Weaknesses of Neural Networks

What considerations should we make when choosing to use nonparametric methods like neural networks over parametric methods for a particular problem?

In any classification or regression problem, we assume that there are some unknown parameters θ that generate our data according to some true underlying model. What we really want is some way of getting from the observed data back to θ. When we take a nonparametric approach, we do not care about the underlying model itself—we presume that it is too hard to directly define some sufficiently complex model and fit parameters to it—so instead aim to learn a function that gets θ from the data. For example, since neural networks can be made almost arbitrarily complicated by adding many layers or by adding convolutional or recurrent structure, these networks can be thought of as very flexible function approximators, but do not have an obvious notion of knowing the true model of the world.

Generally speaking, nonparametric approaches tend to be able to do a very good job of fitting data and thus achieve high accuracy. However, one needs to be careful to make sure that one is actually fitting the data well rather than simply overfitting to the training data. Good practices regarding validation and test sets can help diagnose and mitigate these issues.

One potential downside of nonparametric approaches is that they tend not to be very explanable or interpretable. Even for our two-layer neural network, it is difficult to look at the weights W_2 and have a good sense of what they mean. In contrast, it is fairly easy to interpret the weights in linear regression, since each weight represents how much the output changes per one unit change in each input. Depending on our particular application, this interpretability issue can be a major weakness of neural networks. For instance, there is great debate as to whether neural networks should be used in the sciences—although the networks may be able to predict things with much higher accuracy, they may not give many insights regarding the underlying process which is often the main focus of scientific interest.