Lecture 11: Neural Networks and Matrix Multiply.

CS4787 — Principles of Large-Scale Machine Learning Systems

Review: Linear models and neural networks. From the homeworks and projects you should all be familiar with the notion of a linear model hypothesis class. For example, for multinomial logistic regression, we had the hypothesis class

$$h_W(x) = \mathrm{softmax}(Wx).$$

This is a specific example of a more general linear model of the form

$$h_W(x) = \sigma(Wx)$$

for some inputs $x \in \mathbb{R}^d$, matrix $W \in \mathbb{R}^{D \times d}$, and function $\sigma : \mathbb{R}^D \to \mathbb{R}^D$. Many important methods in machine learning use linear model hypothesis classes, including linear regression, logistic regression, and SVM.

One naive way that we can combine two hypothesis classes is by *stacking* or *layering* them. If I have one class of hypotheses $h^{(1)}_{W_1}$ that maps from $\mathbb{R}^{d_0}$ to $\mathbb{R}^{d_1}$ and a second class of hypotheses $h^{(2)}_{W_2}$ that maps from $\mathbb{R}^{d_1}$ to $\mathbb{R}^{d_2}$, then I can form the layered hypothesis class

$$h_{W_1,W_2}(x) = h^{(2)}_{W_2}(h^{(1)}_{W_1}(x))$$

that results from first applying $h^{(1)}$ and then applying $h^{(2)}$. Intutively, we're first having $h^{(1)}$ make a prediction and then using the result of that prediction as an input to $h^{(2)}$ to make our final prediction. If both our consituent hypothesis classes are linear models, we can write this out more explicitly as

$$h_{W_1,W_2}(x) = \sigma_2(W_2 \cdot \sigma_1(W_1 x)).$$

Of course, we don't need to limit ourselves to layering just two linear classifiers. We could layer as many as we want. For example, if we had $\mathcal{L}$ total layers, then our hypothesis would look like

$$h_{W_1,W_2,\ldots,W_l}(x) = \sigma_l(W_l \cdot \sigma_{l-1}(W_{l-1} \cdots \sigma_2(W_2 \cdot \sigma_1(W_1 x)) \cdots)).$$

We can write this out more generally and explicitly in terms of a recurrence relation.

Typical runtime cost:

$$o_0 = x$$

$$\forall l \in \{1, \ldots, \mathcal{L}\}, \quad a_l = W_l \cdot o_{l-1} + b_l$$

$$\forall l \in \{1, \ldots, \mathcal{L}\}, \quad o_l = \sigma_l(a_l)$$

$$h_{W_1,b_1,W_2,b_2,\ldots,W_l,b_l}(x) = o_{\mathcal{L}}.$$

where $a_l, o_l \in \mathbb{R}^{d_l}$, and here we've also added an explicit *bias* parameter $b_l \in \mathbb{R}^{d_l}$ to each layer. This type of model is called a *multilayer perceptron* (MLP), *artificial neural network* (ANN), or *deep neural network* (DNN). (Specifically, it's a type of deep neural network called a *feedforward neural network*.) Here, the functions σ_l are called the *activation functions* and are almost always chosen to **operate independently along each dimension**; that is (with abuse of notation)

$$(\sigma_l(x))_i = \sigma_l(x_i).$$

Note that this is not true for the softmax, but it's true about pretty much every other major activation function.

Not all neural networks are alike! There are many things we can change when deciding how to structure a neural network. The way we structure a neural network is called its *architecture*.

What things do we need to decide on when picking an architecture?

Which of these things affect the runtime cost of computing the hypothesis given some example x?

Variants of neural networks:

- *Recurrent neural networks* include feedback connections in which the outputs of the model are fed back into itself.
- *Convolutional neural networks* restrict some of the linear transformations W_l to be members of some subset of linear transformations, typically convolutions with some filter.

Computational cost of neural network forward pass. What we just observed is that the overall computational cost of computing the hypothesis is *dominated by the matrix-vector multiply $W_l o_{l-1}$*. That is, the bottleneck for deep neural networks is matrix multiply. As a result, *any* good deep learning system must involve **efficient matrix multiplication**.

- Downside: it's very hard to write an efficient matrix multiply function.
- Upside: you don't need to write an efficient matrix multiply function. Just use one that someone else has already written.
  - Standard library to use on CPU is called BLAS: *Basic Linear Algebra Subroutines*
  - But you can also just use a programming language that wraps BLAS, like Python (numpy), MATLAB, or Julia.
- ...but this means that if you don't use BLAS or something like it, you are leaving a huge amount of potential performance on the table.

Deep learning. We've now defined a hypothesis class for neural networks. But how do we choose a good hypothesis from the class given the data? The solution: use *stochastic gradient descent* and its variants.

Problem: in order to use SGD, we have to be able to compute the derivative of the empirical risk with respect to the parameters of the neural network. How do we do this? **We can just apply the chain rule to our recurrence relation.** One way to compute the gradient is to compute the directional derivative of the loss

$$\frac{d}{d\eta} L(h_{W_1+\eta\Delta W_1, b_1+\eta\Delta b_1, W_2+\eta\Delta W_2, b_2+\eta\Delta b_2, \ldots, W_l+\eta\Delta W_l, b_l+\eta\Delta b_l}(x), y)$$

we can do this algebraically using the chain rule on the recurrence relation that defines the neural network.

Backpropagation. So with a bunch of manual effort, we can compute an expression for the gradient. But this doesn't give us an *algorithm* for computing that gradient efficiently. Backpropagation is one such algorithm. The idea is to first do a forward pass through the network, where we compute the activations just as if we were doing a prediction, and then do a second backward pass to compute the gradients.

What operation takes the most time in the computation of backpropagation?