CS189Hw3

Trustin Nguyen

November 2, 2024

Neural Networks

Many of the most exciting recent breakthroughs in machine learning have come from "deep" (many-layered) neural networks, such as the deep reinforcement learning algorithm that learned to play Atari from pixels, or the ChatGPT model, which generates text that is nearly indistinguishable from human-generated text.

Neural network libraries such as Tensorflow and PyTorch have made training complicated neural network architectures very easy. You don't even really need to understand how they work! With just a few lines of code, you can take a pre-defined neural network architecture and train it on your dataset. These libraries are wonderful for experienced practitioners who understand neural networks inside and out and want to work with a lot of complex machinery at a high level. They're also wonderful for those who don't care to dive deep into the inner workings of neural networks and want to just use pre-defined functions. But for those who want to dive deep and are just learning the material, they tend to obscure the fundamental simplicity and elegance of the inner workings of neural networks. It is easy to get lost in the complexity of the very many classes and parameters defined in these libraries.

In this assignment, we want to emphasize that neural networks begin with a fundamentally simple model that is just a few steps removed from basic logistic regression. In this assignment, you will build two fundamental types of neural network models, all in plain `numpy`: a **feed-forward fully-connected network**, and a **convolutional neural network**. We will start with the essential elements and then build up in complexity.

A neural network model is defined by the following.

- An **architecture** defining the flow of information between computational layers. This defines the composition of functions that the network performs from input to output.
- A **cost function** (e.g. cross-entropy or mean squared error)
- An **optimization algorithm** (e.g. stochastic gradient descent with backpropagation)
- A set of **hyperparameters**. (here we use this as a catch-all term to also include algorithm parameters that technically are not "hyperparameters" in the traditional sense because they don't help you change the bias-variance trade-off, such as the learning rate and the mini-batch size for stochastic gradient descent with mini-batches)

Each *layer* is defined by the following components.

- A **parameterized function** that defines the layer's map from input to output (e.g. $f(x) = \sigma(Wx + b)$).
- An **activation function** σ (e.g. ReLU, sigmoid, etc.)
- A set of **parameters** (e.g. weights and biases).

Neural networks are commonly used for supervised learning problems, where we have a set of inputs and a set of labels, and we want to learn the function that maps inputs to labels. To learn this function, we need to update the parameters of the network (i.e., the weights, including the bias terms). We do this using **mini-batch gradient descent**. To compute the gradients for gradient descent, we use a dynamic programming algorithm called **backpropagation**.

In the backpropagation algorithm, we first compute what is called a **forward** pass of the network. In the forward pass, we send a mini-batch of input data (e.g. 50 training points) through the network. The output is a set of predicted labels, which we use as input to our loss function (along with the true labels from the training data). We then take the gradients of the loss with respect to the parameters of each layer, starting with the output of the network and using the chain rule to propagate backwards through the layers. This is called the backward pass. During the **backward pass** we compute the gradients of the loss function with respect to each of the model parameters, starting from the last layer and "propagating" the information from the loss function backwards through the network. This lets us calculate gradients with respect to all the parameters of our network while letting us avoid computing the same gradients multiple times. To summarize, training a neural network involves three steps.

Layer Implementations

In this question, you will implement the layers needed for basic classification neural networks. For each part, you will be asked to 1) derive the gradients and 2) write the matching code.

When doing the derivations, please derive the gradients element-wise. This means that rather than directly computing $\frac{\partial L}{\partial X}$ by manipulating entire tensors, we encourage you to compute $[\frac{\partial L}{\partial X}]_{ij}$ then stack those components back into a vector/matrix as appropriate. Also keep in mind that for all layers your code must operate on mini-batches of data and should not use loops to iterate over the training points individually. The code is marked with YOUR CODE HERE statements indicating what to implement and where. Please read the docstrings and the function signatures too.

Section 3.1: Activation Functions

First you will implement the ReLU activation function in `activations.py`. ReLU is a very common activation function that is typically used in the hidden layers of a neural network and is defined as

$$\sigma_{\text{ReLU}}(\gamma) = \begin{cases} 0 & \text{if } \gamma < 0 \\ \gamma & \text{otherwise} \end{cases}$$

Note that the activation function is applied element-wise to a vector input.

(1.) **Derive** $\frac{\partial L}{\partial Z}$, the gradient of the downstream loss with respect to the batched input of the ReLU activation function, $Z \in \mathbb{R}^{m \times n}$. First derive the gradient element-wise, i.e. find an expression for $[\frac{\partial L}{\partial Z}]_{ij}$, and then stack these elements appropriately to obtain a simple vector/matrix expression for $\frac{\partial L}{\partial Z}$. **Write your final solution in terms of** $\frac{\partial L}{\partial Y}$ (the gradient of the loss w.r.t. the output $Y = \sigma_{\text{ReLU}}(Z)$ where $Y \in \mathbb{R}^{m \times n}$) and Z. Include your derivation in your writeup.

(2.) **next, implement the forward and backward passes of the ReLU activation** in the script `activations.py`. **Do not iterate over training examples; use batched operations**. Include your code in the appendix and select the appropriate pages when submitting to Gradescope.