TTIC 31230 Fundamentals of Deep Learning, winter 2019

Backpropagation Problems

Problem 0: The KL-divergence between two discrete distributions is defined by

$$KL(P,Q) = E_{x\sim P} \ln \frac{P(x)}{Q(x)}$$

We will show later in the class that $KL(P,Q) \geq 0$ for any P and Q.
The Cross Entropy $H(P,Q)$ is defined by

$$H(P,Q) = E_{x\sim P} - \ln Q(x)$$

When P is a population distribution and Q is a model distribution this is the cross entropy loss.
The entropy of a distribution is defined by

$$H(P) = Ex \sim P\ \ - \ln P(x)$$

Show the following.

$$H(P,Q) \geq H(P)$$

$$H(P,P) = H(P)$$

Explain why these facts are important when training a model to minimize cross entropy loss.

Problem 1: Backpropagation through softmax. Consider the following softmax.

$$\begin{aligned} Z[b] &= \sum_j \exp(s[b,j]) \\ p[b,j] &= \exp(s[b,j])/Z[b] \end{aligned}$$

An alternative way to compute this is to initialize the tensors Z and p to zero and then execute the following loops.

for b, j $\quad Z[b]$ += $\exp(s[b,j])$

for b, j $\quad p[b,j]$ += $\exp(s[b,j])/Z[b]$

Each individual += operation inside the loops can be treated independently in backpropagation.

(a) Give a back-propagation loop over += updates based on the second loop for adding to s.grad using p.grad (and using the forward-computed tensors Z and s).

(b) Give a back-propagation loop over `+=` updates based on the second equation for adding to Z.grad using p.grad (and using the forward-computed tensors s and Z).

(c) Give a back-propagation loop over `+=` updates based on the first equation for adding to s.grad using Z.grad (and using the forward-computed tensor s).

Problem 2: Optimizing Backpropagation through softmax. Show that the addition to s.grad shown in problem 1 can be computed using the following more efficient updates.

for b, j $\quad e[b]$ `-=` $p[b,j]p.\text{grad}[b,j]$

for b, j $\quad s.\text{grad}[b,j]$ `+=` $p[b,j](p.\text{grad}[b,j] + e[b])$

This formula shows how hand-written back-propagation methods for "layers" such as softmax can be more efficient than compiler-generated back-propagation code. While optimizing compilers can of course be written, one must keep in mind the trade-off between the abstraction level of the programming language and the efficiency of the generated code.

Problem 3. Backpropogation through batch normalization. Consider the following set of `+=` statements defining batch normalization where all computed tensors are initialized to zero.

For b, j $\;\mu[j]$ `+=` $\frac{1}{B}\; x[b,j]$

For b, j $\;s[j]$ `+=` $\frac{1}{B-1}\; (x[b,j] - \mu[j])^2$

For b, j $\;x'[b,j]$ `+=` $\frac{x[b,j]-\mu[j]}{\sqrt{s[j]}}$

Give backpropagation `+=` (or `-=`) loops for computing x.grad$[b,j]$, μ.grad$[j]$, and s.grad$[j]$ from x'.grad$[b,j]$. The loops should be given in the order they are to be executed.

Problem 4. Backpropagation through a UGRNN. Equations defining a UGRNN are given below.

$$\begin{aligned}
\tilde{R}_t[b,j] &= \left(\sum_i W^{h,R}[j,i]h_{t-1}[b,i]\right) + \left(\sum_k W^{x,R}[j,k]x_t[b,k]\right) - B^R[j] \\
R_t[b,j] &= \tanh(\tilde{R}_t[b,j]) \\
\tilde{G}_t[b,j] &= \left(\sum_i W^{h,G}[j,i]h_{t-1}[b,i]\right) + \left(\sum_k W^{x,G}[j,k]x_t[b,k]\right) - B^G[j] \\
G_t[b,j] &= \sigma(\tilde{G}_t[b,j]) \\
h_t[b,j] &= G_t[b,j]h_{t-1}[b,j] + (1 - G_t[b,j])R_t[b,j]
\end{aligned}$$

(a) Rewrite the first equation defining $\tilde{R}_t$ using `+=` loops instead of summations assuming that all computed tensors are initialized to zero.

(b) Give `+=` loops for the backward computation for your solution to part (a) using the convention that parameter gradients are averaged over the batch and where the batch size is B.

Problem 5. Writing framework code. Consider a function $c : R^d \times R^s \rightarrow R^s$, in other words a function that takes a vector of dimension d and a vector of dimension s and yields a vector of dimension s. Given a sequence of vectors x_0, x_2, …, x_T with $x_t \in R^d$ we can define a sequence of vectors h_0, h_1, …, h_T by the equations

$$\begin{aligned} h_0 &= c(x_0, 0) \\ h_t &= c(x_t, h_{t-1}) \text{ for } 1 \leq t \leq T \end{aligned}$$

When the function c is defined by a neural network the resulting network mapping x_1, …, x_T to h_0, …, h_T is called a recurrent neural network (RNN).
a. In the educational framework EDF we work with objects where each object has a value attribute and a gradient attribute each of which have tensor values where the value tensor and the gradient tensor are the same shape. Each object is assigned a value in a forward pass and assigned a gradient in a backward pass. Suppose that we are given an EDF procedure `CELL` which takes as arguments a parameter object `Phi` and two EDF objects `X` and `H` where the value attribute of the object `X` is a d-dimensional vector and the value attribute of the object `H` is an s-dimensional vector. A call to the procedure `CELL(Phi,X,H)` returns an EDF object whose value attribute is computed in a forward pass in some possibly complex way from the value attributes of `Phi`, `X` and `H`. Given a sequence `X[]` of EDF objects whose value attributes are d-dimensional vectors, and an EDF object `ZERO` representing the constant s-dimensional zero vector, write a procedure for constructing the sequence of EDF objects representing h_1, h_2, …, h_T as defined by the above RNN equations. Your solution can be in Python or informal high level pseudo code.
b. Deep learning systems generally make extensive use of parallel computation for training. How does the parallel running time of an RNN computation graph scale with the length T?