TTIC 31230 Fundamentals of Deep Learning, winter 2019

Backpropagation Problems

Problem 1: Backprogation through a ReLU linear threshold unit. Consider the computation

$$\begin{aligned} y &= \sigma(w^\top x) \\ \ell &= \mathcal{L}(y) \end{aligned}$$

for $w, x \in R^d$ with $\sigma(z) = \max(z, 0)$ (the ReLU activation) and for $\mathcal{L}(y)$ an arbitrary function (a loss function). Let w_i denote the ith component of the weight vector w. Give an expression for $\frac{\partial \ell}{\partial w_i}$ as a function of $\frac{d\mathcal{L}(y)}{dy}$.

Solution: There are various correct ways of writing the answer. The following corresponds to a backpropagation computation.

$$\begin{aligned} \frac{d\ell}{dy} &= \frac{d\mathcal{L}(y)}{dy} \\ \frac{d\ell}{dw_i} &= \frac{d\ell}{dy}\frac{dy}{dw_i} = \frac{d\ell}{dy}\, x_i \mathbf{1}\left[w^\top x \geq 0\right] \end{aligned}$$

Problem 2: 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).

Solution: For $b, j \quad s.\mathrm{grad}[b,j]$ += $p.\mathrm{grad}[b,j] \exp(s[b,j])/Z[b]$

(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).

Solution: For b, j $\quad Z.\mathrm{grad}[b]$ -= $p.\mathrm{grad}[b,j]\exp(s[b,j])/Z[b]^2$

(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).

Solution: For b, j $\quad s.\mathrm{grad}[b,j]$ += $Z.\mathrm{grad}[b]\exp(s[b,j])$

Problem 3: Optimizing Backpropagation through softmax. Show that the addition to s.grad shown in problem 2 can be computed using the following more efficient updates.

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

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

Solution: The updates for problem 1 can be written as

$$\begin{aligned}\text{for } b \quad Z.\mathrm{grad}[b] &= \sum_j -p.\mathrm{grad}[b,j]\exp(s[b,j])/Z[b]^2 \\ &= \left(\sum_j -p[b,j]p.\mathrm{grad}[b,j]\right)/Z[b] \\ &= e[b]/Z[b]\end{aligned}$$

$$\begin{aligned}\text{for } b,j \quad s.\mathrm{grad}[b,j] &= p.\mathrm{grad}[b,j]\exp(s[b,j])/Z[b] + Z.\mathrm{grad}[b]\exp(s[b,j]) \\ &= p.\mathrm{grad}[b,j]\,(\exp(s[b,j])/Z[b]) + e[b]\,(\exp(s[b,j])/Z[b]) \\ &= p[b,j](p.\mathrm{grad}[b,j] + e[b])\end{aligned}$$

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 4. 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 $x.\mathrm{grad}[b,j]$, $\mu.\mathrm{grad}[j]$, and $s.\mathrm{grad}[j]$ from $x'.\mathrm{grad}[b,j]$. The loops should be given in the order they are to be executed.

Solution:

$$\begin{array}{lrcl}
\text{For } b,j & x.grad[b,j] & \texttt{+=} & \frac{x'.\mathrm{grad}[b,j]}{\sqrt{s[j]}} \\
\text{For } b,j & \mu.grad[j] & \texttt{-=} & \frac{x'.\mathrm{grad}[b,j]}{\sqrt{s[j]}} \\
\text{For } b,j & s.grad[j] & \texttt{-=} & \frac{1}{2}(x[b,j]-\mu[j])s[j]^{-3/2}\; x'.grad[b,j] \\
\text{For } b,j & x.grad[b,j] & \texttt{+=} & \frac{2}{B-1}\;(x[b,j]-\mu[j])s.\mathrm{grad}[j] \\
\text{For } b,j & \mu.\mathrm{grad}[j] & \texttt{-=} & \frac{2}{B-1}\;(x[b,j]-\mu[j])s.\mathrm{grad}[j] \\
\text{For } b,j & x.grad[b,j] & \texttt{+=} & \frac{1}{B}\;\mu.\mathrm{grad}[j]
\end{array}$$

Problem 5. Backpropagation through a UGRNN. Equations defining a UGRNN are given below.

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

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

Solution:

$$\begin{array}{rcl}
\text{for } b,j,i\;\tilde{R}_t[b,j] & \texttt{+=} & W^{h,R}[j,i]h_{t-1}[b,i] \\
\text{for } b,j,k\;\tilde{R}_t[b,j] & \texttt{+=} & W^{X,R}[k,i]x_t[b,k] \\
\text{for } b,j\;\tilde{R}_t[b,j] & \texttt{-=} & B^R[j]
\end{array}$$

(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.

Solution:

$$\begin{aligned}
\text{for } b,j,i\ W^{h,R}.\text{grad}[j,i] \quad & \texttt{+=} \quad \frac{1}{B}\, h_{t-1}[b,i]\tilde{R}_t.\text{grad}[b,j] \\
\text{for } b,j,i\ h_{t-1}.\text{grad}[b,j] \quad & \texttt{+=} \quad W^{h,R}[j,i]\tilde{R}_t.\text{grad}[b,j] \\
\text{for } b,j,k\ W^{x,R}.\text{grad}[j,k] \quad & \texttt{+=} \quad \frac{1}{B}\, x[b,k]\tilde{R}_t.\text{grad}[b,j] \\
\text{for } b,j\ B^{R}.\text{grad}[j] \quad & \texttt{-=} \quad \frac{1}{B}\, \tilde{R}_t.\text{grad}[b,j]
\end{aligned}$$

Problem 6. Writing framework code. Consider a function $c : R^d \times R^s \to R^s$, 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.

Solution: We can use the equations given as the definition of the computation graph if we replace c in the equations with the function `CELL`.

```
X = list()
H = list()
H[0] = CELL(Phi,X[0],ZERO)
for t in range(1,T)
    H[t] = CELL(Phi,X[t],H[t-1])
```

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?

Solution: The parallel running time is proportional to T. RNNS are fundamentally serial and this is a problem. RNNs have recently been largely replaced by the transformer architecture.