Homework 03 – Calculus

Arthur J. Redfern
arthur.redfern@utdallas.edu
Feb 04, 2019

0 Outline

1 Logistics
2 Reading
3 Theory
4 Practice

1 Logistics

Assigned:	Mon Feb 04, 2019
Due:	Mon Feb 11, 2019
Format:	PDF uploaded to eLearning

2 Reading

1. Read the calculus slides

    Calculus
    https://github.com/arthurredfern/UT-Dallas-CS-6301-CNNs/blob/master/Lectures/xNNs_03_Calculus.pdf

    Complete

2. We didn't discuss RNN training and back propagation through time (summary: you unroll the RNN in time, perform back propagation as you would for a typical feed forward network to compute gradients of the error with respect to the weights at each time step, then sum the gradients of errors with respect to the weights for common weights before their update in a gradient descent based algorithm). Read the following reference to get a better feel for this

    Recurrent neural networks tutorial, part 3 – backpropagation through time and vanishing gradients http://www.wildml.com/2015/10/recurrent-neural-networks-tutorial-part-3-backpropagation-through-time-and-vanishing-gradients/

Complete

3. [**Optional**] If you would like more information on automatic differentiation with reverse mode accumulation (and more), see the following survey paper. Note that it can be dense at times, is not necessarily fully up to date with evolving libraries and not all of it applies to this class (but that's ok).

Automatic differentiation in machine learning: a survey
https://arxiv.org/abs/1502.05767

Complete

3 Theory

4. Let **x** be the K x 1 vector output of the last layer of a xNN and e = crossEntropy(**p***, softMax(**x**)) be the error where **p*** is a K x 1 vector with a 1 in position k* representing the correct class and 0s elsewhere. Derive ∂e/∂**x**. Large portions of this are shown in the slides, however, the purpose of this question is for you to derive all of the parts yourself to gain more confidence with error gradients. Here's a cookbook of steps and hints:

4.1. Derive the gradient of the cross entropy for a 1 hot label at position k*. Use derivative rule for log (assume base e) and note that only 1 element of the gradient is non zero.

$e = H_{CE}(\mathbf{p}^*, \mathbf{p})$

$= -\Sigma_k\, p^*(k) \log(p(k))$

$= -\log(p(k^*))$

$\partial e/\partial \mathbf{p} = [0, \ldots, 0, -1/p(k^*), 0, \ldots, 0]^T$, with the nonzero element at position k*

4.2. Derive the Jacobian of the soft max. Use the derivative quotient rule and note 2 cases: i != j and i == j (where i and j refer to the Jacobian row and col). Apply a common trick for functions with exponentials and re write the derivatives in terms of original function.

$\mathbf{p} = \text{softmax}(\mathbf{x})$

$= (1/(\Sigma_k\, e^{x(k)}))\ [e^{x(0)}, \ldots, e^{x(K-1)}]^T$

$p(i) = e^{x(i)}/(\Sigma_k\, e^{x(k)})$

$\partial p(i\ != j)/\partial x(j) = (0 - e^{x(j)} e^{x(i)}) / (\Sigma_k\, e^{x(k)})^2$

$= -p(j)\,p(i)$

$\partial p(i == j)/\partial x(j) = (e^{x(j)} \Sigma_k e^{x(k)} - e^{x(j)}e^{x(i)}) / (\Sigma_k e^{x(k)})^2$
$= e^{x(j)} (\Sigma_k e^{x(k)} - e^{x(i)}) / (\Sigma_k e^{x(k)})^2$
$= p(j)(1 - p(i))$
$= p(i)(1 - p(i))$

$\partial \mathbf{p}/\partial \mathbf{x}$

```
= [ p(0)(1-p(0))    -p(0)p(1)        ...   -p(0)p(K-1)       ]
  [-p(1)p(0)         p(1)(1-p(1))          -p(1)p(K-1)       ]
  [...                               ...                     ]
  [-p(K-1)p(0)      -p(K-1)p(1)             p(K-1)(1-p(K-1))]
```

4.3. Apply the chain rule to derive the gradient of e = crossEntropy(**p***, softMax(**x**)) as the Jacobian matrix times the gradient vector. Take advantage of only 1 element of the gradient vector being non zero effectively selecting the corresponding col of the Jacobian matrix.

$\partial e/\partial \mathbf{x} = (\partial \mathbf{p}/\partial \mathbf{x})\,(\partial e/\partial \mathbf{p})$
$= [p(0), \ldots, p(k^* - 1), p(k^*) - 1, p(k^* + 1), \ldots, p(K - 1)]$

4.4. Note the beautiful and numerically stable result

Noted

4.5. Remind yourself in the future when implementing classification networks in software to use a single call to the high level library's built in combined soft max cross entropy function instead of making 2 calls to separate soft max and cross entropy functions.

Note to self: use a single call to a high level library's built in soft max cross entropy function instead of making 2 calls to separate soft max and cross entropy functions

5. Consider a simple residual block of the form **y** = **x** + f(**H x** + **v**) where **x** is a K x 1 input feature vector, **H** is K x K linear transformation matrix, **v** is a K x 1 bias vector, f is a ReLU pointwise nonlinearity and **y** is a K x 1 output feature vector. Assume that ∂e/∂**y** is given. Write out a single expression using the chain rule for ∂e/∂**x** in terms of ∂e/∂**y** and the Jacobians of the other operations. For the ReLU, define the Jacobian as a K x K diagonal matrix **I**{0, 1}. Note the clean flow of the gradient from the output to the input, this is a key for training deep networks.

Define $\mathbf{x}_0 = \mathbf{x}$ after the split for the main path and $\mathbf{x}_1 = \mathbf{x}$ after the split for the residual path. Furthermore, define intermediate feature maps as:

$\mathbf{x}_2 = \mathbf{H}\,\mathbf{x}_1$
$\mathbf{x}_3 = \mathbf{x}_2 + \mathbf{v}$

$\mathbf{x}_4 \quad = f(\mathbf{x}_3)$

which allows the output to be written as

$\mathbf{y} \quad = \mathbf{x}_0 + \mathbf{x}_4$

Now compute partial derivatives using the chain rule given $\partial e/\partial \mathbf{y}$ for the main path

$\partial e/\partial \mathbf{x}_0 = (\partial \mathbf{y}/\partial \mathbf{x}_0)\ (\partial e/\partial \mathbf{y}) \qquad = (\partial e/\partial \mathbf{y})$

and the residual path

$$\begin{aligned}
\partial e/\partial \mathbf{x}_4 &= (\partial \mathbf{y}/\partial \mathbf{x}_4)\ (\partial e/\partial \mathbf{y}) & & &&= (\partial e/\partial \mathbf{y}) \\
\partial e/\partial \mathbf{x}_3 &= (\partial \mathbf{x}_4/\partial \mathbf{x}_3)\ (\partial e/\partial \mathbf{x}_4) &&= I_{x3}\{0, 1\}\ (\partial e/\partial \mathbf{x}_4) &&= I_{x3}\{0, 1\}\ (\partial e/\partial \mathbf{y}) \\
\partial e/\partial \mathbf{x}_2 &= (\partial \mathbf{x}_3/\partial \mathbf{x}_2)\ (\partial e/\partial \mathbf{x}_3) &&= (\partial e/\partial \mathbf{x}_3) &&= I_{x3}\{0, 1\}\ (\partial e/\partial \mathbf{y}) \\
\partial e/\partial \mathbf{x}_1 &= (\partial \mathbf{x}_2/\partial \mathbf{x}_1)\ (\partial e/\partial \mathbf{x}_2) &&= \mathbf{H}^T\ (\partial e/\partial \mathbf{x}_2) &&= \mathbf{H}^T\ I_{x3}\{0, 1\}\ (\partial e/\partial \mathbf{y})
\end{aligned}$$

In the reverse graph gradients sum at the splits of the forward graph

$$\begin{aligned}
\partial e/\partial \mathbf{x} &= (\partial e/\partial \mathbf{x}_0) + (\partial e/\partial \mathbf{x}_1) \\
&= (\partial e/\partial \mathbf{y}) + \mathbf{H}^T\ I_{x3}\{0, 1\}\ (\partial e/\partial \mathbf{y})
\end{aligned}$$

So $\partial e/\partial \mathbf{x}$ is $\partial e/\partial \mathbf{y}$ + a perturbation from the residual path

6. Write out the gradient descent update for **H** and **v** in the above example. Define intermediate feature maps as necessary. Note the need to save feature maps from the forward pass which has memory implications for xNN training.

Using the same definitions as problem 5, the gradients with respect to the weights can be found as

$$\begin{aligned}
\partial e/\partial \mathbf{v} &= (\partial e/\partial \mathbf{x}_3) \odot (\partial \mathbf{x}_3/\partial \mathbf{v}) &&= (\partial e/\partial \mathbf{x}_3) &&= I_{x3}\{0, 1\}\ (\partial e/\partial \mathbf{y}) \\
\partial e/\partial \mathbf{H} &= (\partial e/\partial \mathbf{x}_2)\ (\partial \mathbf{x}_2/\partial \mathbf{H}) &&= (\partial e/\partial \mathbf{x}_2)\ \mathbf{x}_1^T &&= I_{x3}\{0, 1\}\ (\partial e/\partial \mathbf{y})\ \mathbf{x}_1^T
\end{aligned}$$

and the gradient descent updates written as

$$\begin{aligned}
\mathbf{v}_{t+1} &= \mathbf{v}_t - \alpha_t\ (\partial e/\partial \mathbf{v}) &&= \mathbf{v}_t - \alpha_t\ I_{x3}\{0, 1\}\ (\partial e/\partial \mathbf{y}) \\
\mathbf{H}_{t+1} &= \mathbf{H}_t - \alpha_t\ (\partial e/\partial \mathbf{H}) &&= \mathbf{H}_t - \alpha_t\ I_{x3}\{0, 1\}\ (\partial e/\partial \mathbf{y})\ \mathbf{x}_1^T
\end{aligned}$$

using t to indicate time step t

7. [**Optional**] It was previously observed that a CNN style 2D convolution with N_o x N_i x F_r x F_c filters can be lowered to the sum of F_r*F_c matrix multiplications between matrices composed of

filter coefficients from a specific f_r and f_c and matrices composed of shifted input feature map elements. Specifically, starting from the following tensors

Input feature maps 3D tensor $\mathbf{X}$ of size N_i x L_r x L_c

Filter coefficients 4D tensor $\mathbf{H}$ of size N_o x N_i x F_r x F_c

Output feature maps 3D tensor $\mathbf{Y}$ of size N_o x $(L_r – F_r + 1)$ x $(L_c – F_c + 1)$

CNN style 2D convolution can be lowered to matrix multiplication via defining

Input feature map filtering matrix $\mathbf{X}_{\text{filter}}^{2D}$ of size $(N_i*F_r*F_c)$ x $((L_r – F_r + 1)*(L_c – F_c + 1))$

Filter coefficient matrix $\mathbf{H}^{2D}$ of size N_o x $(N_i*F_r*F_c)$

Output feature map matrix $\mathbf{Y}^{2D}$ of size N_o x $((L_r – F_r + 1)*(L_c – F_c + 1))$

and computing

$$\mathbf{Y}^{2D} = \mathbf{H}^{2D}\, \mathbf{X}_{\text{filter}}^{2D}$$

Each element of $\mathbf{X}$ is repeated ~ F_r*F_c times in $\mathbf{X}_{\text{filter}}^{2D}$ (where the approximation is due to edge effects) which complicates the computation of $\partial e/\partial\mathbf{X}$ in terms of $\partial e/\partial\mathbf{Y}$ due to increased memory requirements and the necessity to track the indices of repeated values of $\mathbf{X}$ in $\mathbf{X}_{\text{filter}}^{2D}$ indicating the gradients that need to be summed together.

To get around this, the above multiplication can be re written as the sum of F_r*F_c matrix multiplications by defining

Input feature map matrix $\mathbf{X}_{fr,fc}^{2D}$ of size N_i x $((L_r – F_r + 1)*(L_c – F_c + 1))$ as

$\mathbf{X}_{fr,fc}^{2D} = \mathbf{X}_{\text{filter}}^{2D}((f_r + f_c*F_r):(F_r*F_c):\text{end}, :)$

Filter coefficient matrix $\mathbf{H}_{fr,fc}^{2D}$ of size N_o x N_i as

$\mathbf{H}_{fr,fc}^{2D} = \mathbf{H}^{2D}(:, (f_r + f_c*F_r):(F_r*F_c):\text{end})$

Putting all of this together, CNN style 2D convolution can be lowered to the sum of F_r*F_c matrix multiplications

$$\begin{aligned}\mathbf{Y}^{2D} &= \mathbf{H}^{2D}\, \mathbf{X}_{\text{filter}}^{2D}\\ &= \Sigma_{fr,fc}\, \mathbf{H}_{fr,fc}^{2D}\, \mathbf{X}_{fr,fc}^{2D}\end{aligned}$$

Starting from this last equation and using the properties of join operations in the graph adjoint leading to the summing gradients (pay attention to shifting and edge effects) and the known formulas for propagating gradients backwards through matrix transforms, derive $\partial e/\partial\mathbf{X}$ in terms of $\partial e/\partial\mathbf{Y}$.

4 Practice

8. Run the Fashion MNIST with Keras and TPUs seed on Google's Colaboratory. Note the model structure in the "Define the model" section of 3x CNN style 2D convolution layers and 2x dense layers. Run the model as is, then play around with some of the parameters in this section as appropriate (filter size, number of input / output feature maps, number of layers, ...). How is accuracy affected by your changes? How is performance affected by your changes?

Complete

9. Run the Neural Translation with Attention seed on Google's Colaboratory. Read through the comments and Code in the iPython notebook, but don’t worry about understanding everything. This model uses a variant of RNNs and a variant of attention.

Complete