Homework 04 – Calculus

Arthur J. Redfern
arthur.redfern@utdallas.edu

---

0 Outline

1 Reading
2 Theory
3 Practice

1 Reading

1. Calculus

   Motivation: a xNN related calculus refresher
   https://github.com/arthurredfern/UT-Dallas-CS-6301-CNNs/blob/master/Lectures/xNNs_040_Calculus.pdf

   Complete

2. Recurrent neural networks tutorial, part 3 – backpropagation through time and vanishing gradients

   Motivation: 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). This tutorial provides more information on the process.
   http://www.wildml.com/2015/10/recurrent-neural-networks-tutorial-part-3-backpropagation-through-time-and-vanishing-gradients/

   Complete

3. [Optional] Automatic differentiation in machine learning: a survey

   Motivation: if you would like more information on automatic differentiation with reverse mode accumulation (and more), read this 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).

https://arxiv.org/abs/1502.05767

Complete

2 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 the derivative rule for log (assume base e) and note that only 1 element of the gradient is non zero.

$$
\begin{aligned}
e &= H_{CE}(\mathbf{p}^*, \mathbf{p}) \\
&= -\Sigma_k\, p^*(k)\, \log(p(k)) \\
&= -\log(p(k^*))
\end{aligned}
$$

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

$$
\begin{aligned}
\mathbf{p} &= \text{softmax}(\mathbf{x}) \\
&= (1/(\Sigma_k\, e^{x(k)}))\, [e^{x(0)}, \ldots, e^{x(K-1)}]^T
\end{aligned}
$$

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

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

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

$$
\partial \mathbf{p}/\partial \mathbf{x} = \begin{bmatrix} p(0)(1-p(0)) & -p(0)p(1) & \cdots & -p(0)p(K-1) \\ -p(1)p(0) & p(1)(1-p(1)) & & -p(1)p(K-1) \end{bmatrix}
$$

```
[...                                    ...                                ]
[-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, use a single call to the high level library's built in combined soft max cross entropy function if it's available instead of making 2 calls to separate soft max and cross entropy functions. But realize that some libraries combine separate functions as an optimization step behind the scenes for you so if it's not available then it's probably still ok.

Reminded

5. Consider a simple residual block of the form $\mathbf{y} = \mathbf{x} + f(\mathbf{H}\,\mathbf{x} + \mathbf{v})$ 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 $\partial e/\partial \mathbf{y}$ is given. Write out a single expression using the chain rule for $\partial e/\partial \mathbf{x}$ in terms of $\partial e/\partial \mathbf{y}$ 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:

$\mathbf{x}_2 = \mathbf{H}\,\mathbf{x}_1$
$\mathbf{x}_3 = \mathbf{x}_2 + \mathbf{v}$
$\mathbf{x}_4 = f(\mathbf{x}_3)$

which allows the output to be written as

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

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

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

and the residual path

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

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

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

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

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

and the gradient descent updates written as

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

using t to indicate time step t

7. [**Optional**] It was previously observed that 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 **X** of size N_i x L_r x L_c
- Filter coefficients 4D tensor **H** of size N_o x N_i x F_r x F_c
- Output feature maps 3D tensor **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 $\mathbf{X}_{filter}^{2D}$ of size $(N_i*F_r*F_c) \times ((L_r - F_r + 1)*(L_c - F_c + 1))$
- Filter coefficient matrix $\mathbf{H}^{2D}$ of size $N_o \times (N_i*F_r*F_c)$
- Output feature map matrix $\mathbf{Y}^{2D}$ of size $N_o \times ((L_r - F_r + 1)*(L_c - F_c + 1))$

and computing

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

Each element of $\mathbf{X}$ is repeated ~ F_r*F_c times in $\mathbf{X}_{filter}^{2D}$ (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 $\mathbf{X}_{filter}^{2D}$ 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 $N_i \times ((L_r - F_r + 1)*(L_c - F_c + 1))$ as

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

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

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

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

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

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 $\partial e/\partial \mathbf{Y}$.

3 Practice

8. It would be nice to train a classifier from scratch on ~ 3 x 256 x 256 images in ImageNet but this is not practical in Colab due to the dataset size and network training time. Conveniently, TensorFlow Datasets also links to a down sampled version of ImageNet (https://www.tensorflow.org/datasets/datasets#downsampled_imagenet) with 3 x 64 x 64 image sizes (make sure to set the config name to properly to select this version and not the 3 x 32 x 32 version) that takes up a more manageable ~ 12 GB of memory and should train ~ 16x faster. Unfortunately, labels are not provided for the images.

So here's the strategy that I'd like you to take to try and train a down sampled ImageNet classifier from scratch:

- You need labels. We'll play a trick and use the output of a strong pre trained network as the ground truth. Yes, this isn't fully correct. No, for our purposes it's not the worst thing in the world.

- Do this for each training and testing "down sampled image batch" by creating an appropriately "resized image batch", passing the "resized image batch" to a strong pre trained network and generating a corresponding "label batch". Then use the "down sampled image batch" and corresponding "label batch" pairs for training and testing the new network.

- Multiple interpolation methods are possible for image resizing. Likely the interpolation method affects accuracy, so spend some time thinking about different options (e.g., plot out the results for different interpolation methods and pick the 1 which is the most visually appealing).

- TensorFlow Hub is a potential source for pre trained models. For information on available models and their use see:

    - https://www.tensorflow.org/tutorials/images/hub_with_keras
    - https://tfhub.dev/google/tf2-preview/inception_v3/classification/4
    - https://tfhub.dev/google/tf2-preview/mobilenet_v2/classification/4
    - ...

- Keras Applications are also a source for pre trained models. For information on available models and their use see: https://keras.io/applications/

- Pay attention to the pre trained network's required input normalization and input sizing as different networks have different requirements. It's not clear how well / not well the TensorFlow Hub TensorFlow 1 models will work with TensorFlow 2 so watch out for that too.

- For the new network you train, use the general sequential CNN structure used in the CIFAR-10 code with (a) an additional level (pooling followed by multiple convolutions) and (b) a larger decoder with 1000 / 1001 output classes.

- The new network takes you to a width of 256 channels at the output of level 3, a fair amount less than 1000 / 1001 classes, which will be 1 factor that limits accuracy. Optionally, consider adding an extra 1 or 2 levels without pooling and increasing the number of channels to 512 or 1024.

What is the best accuracy (really relative to the pre trained network) you achieve? What is the training time? Checkpointing and restarting training may be needed. Provide all key parameters and a link to your code.

Good luck!