Training a Convolutional Neural Network on the GPU using Numba

João Castanheira (fc55052), Leon Ingelse (fc55969)

February 1, 2021

Abstract

Convolutional Neural Networks (CNNs) are become more relevant every day, with applications in image, video, text, and sound analysis. The high parameter complexity and data complexity, make it a suitable candidate for parallelization. Graphic Processing Units (GPUs) are on the rise for programming parallel processes. In this paper, using the available information on the parallelization of Neural Networks (NNs), we write a parallel programme for training a CNN on the GPU. We will analyze its performance in comparison to CPU versions with different types of complexity levels. We discuss the problem of complexity differences between the CNN-typical convolutional layer and the common fully-connected layer. We achieve considerable speed ups, with the prospect of more as there are still improvements to be made which were out of scope for this project.

1 Introduction

Convolutional neural networks (CNNs) are a particular type of neural networks (NNs), specifically, NNs optimized for certain multiple-dimensional pattern recognition problems. It is optimized to detect local patterns, which are then bundled to recognize global patterns. These local patterns can be seen as features. For example in image analysis of buildings: first search for lines, then bundle them to recognize features, such as windows and doors, and finally put these features together to conclude a house is depicted. In this example, lines are local patterns in an image. Other use-cases of CNNs can be found in text analysis, and sound analysis.

Learning a CNN consist of three parts. As an example, consider the MNIST data set [6], with images of hand-written numbers. Step 1 is to analyze a batch of images by a CNN with randomly-created parameters, resulting to a classification of the images. Step 2 is to put the CNN's classification next to the true classification, and calculate the prediction error of the CNN. In the last step, step 3, the parameters are updated so that the predictions have a lower error. This implementation is a very costly job for two reasons.

1. Usually, a CNN has many neurons, and thus many parameters that should be optimized. As for every optimization gradients of the error must be calculated, this is a very costly process.

2. To get accurate results, a lot of data must be analyzed, taking a lot of time.

For our project we will parallelize a CNN in order to speed up the learning process on the GPU. We use Python [10] and the LLVM compiler library, through Numba [5].

2 Background

For this project, a basic knowledge of CNNs is required. In this section we introduce Neural Networks (NNs). After that, we extend on the convolutional layer, the layer that give CNNs their name, and the max-pooling layer, which typically follows a convolutional layer.

2.1 Introduction to NNs

A Neural Network is similar to a classification function. Given a data point as input, instead of giving a single classification, it gives a score for the classification of each class. This could be adapted to represent a probability that a data point will be correctly classified if that class was predicted.

Similar to CNNs, training an NN consists of three parts. First, classifying the training data, known as the forward pass. Then, grading the prediction. And lastly, updating the parameters of the NN, called backpropagation, so that the prediction grade increases. To begin with, we will discuss these three processes, starting with the second, grading of the prediction.

2.1.1 Loss function

Given some input, an NN gives a score for all the possible classes, representing whether it would classify the input as that class. These scores can be graded in different ways, depending on our needs. A function that grades the classifiction scores is called a *loss function*. Here we will introduce the *softmax* loss function.

Definition 1 (Softmax classifier)**.** *Let $C = \{c_1, \dots, c_n\}$ be a set of classes, X any space, and*

$$X \times C = \{(x_1, y_1), \dots, (x_k, y_k) | y_i \text{ is the true class of } x_i\}$$

a classified data set. Let $f : X \to \mathbb{R}^n$ be a classifier, such that for $x \in X$, the mapping $f(x) = (s_1, \dots, s_n) \in \mathbb{R}^n$ represents the score for each class, with s_j being the score for class c_j.

Let $(x, c_i) \in X \times C$ and $f(x) = (s_1, \dots, s_n)$. To grade the classification of x by f, we will use the loss function $L : X \to \mathbb{R}$, defined as

$$L(x) = -\log \frac{e^{s_i}}{\sum_{j=1,\dots,n} e^{s_j}}.$$

The full loss of the data set is the mean loss of all $x \in X$.

Notice that $\frac{e^{s_i}}{\sum_{j=1,\dots,n} e^{s_j}}$ is an increasing function in s_i, so that $L(x)$ is a decreasing function in s_i. Also note the limits of the function:

$$\lim_{s_i \to -\infty} L(x) = \lim_{s_i \to -\infty} -\log \frac{e^{s_i}}{\sum_{j=1,\dots,n} e^{s_j}} = -\log(0) = \infty$$

$$\lim_{s_i \to \infty} L(x) = \lim_{s_i \to \infty} -\log \frac{e^{s_i}}{\sum_{j=1,\dots,n} e^{s_j}} = \lim_{s_i \to \infty} -\log(\frac{e^{s_i}}{e^{s_i} + c}) = -log(1) = 0,$$

where $c = \sum_{j=1,\dots,n, j \neq i} e^{s_j}$. Furthermore, notice that $\forall j \neq i$, $\frac{e^{s_i}}{\sum_{j=1,\dots,n} e^{s_j}}$ is a decreasing function in s_j, and thus $L(x)$ is an increasing function in s_j. These characteristics of the Softmax classifier make it perfect for our needs; a higher value presents a bigger loss, a worse model.

2.1.2 Forward pass

The most common layer of an NN is a *fully-connected* layer. This a set of weights which are grouped in so-called neurons. It's called a fully-connected layer, as all input is processed by all neurons to some output (or straight into a next layer). Let's formalize this type of layer.

Suppose we have some n-dimensional vector space input X, and an output space Y. Every neuron is a mapping from X to Y. This mapping can be represented by a matrix multiplication with a matrix of size $\dim(X) \times \dim(Y)$. The elements of this matrix are often called the weights. The weights, together with a *bias* as an independent (of data) parameter, will be denoted by W. The outputs in Y are then summed and run through a non-linear function. This function is called the *activation function*. If the fully-connected layer is the last one in the network, no activation function will be applied, as the outputs will be analysed by the loss function anyways.

There are many other types of layer, one being the *convolutional layer*. We will discuss this later.

2.1.3 Backpropagation

After running the network and obtaining the loss of the network, we will go back through the model and update the parameters of the forward pass, according to the gradient of the loss with respect to that parameter. This process is called backpropagation. Calculating the gradient of L over the parameters might seem strange as the loss function L we defined in Definition 1, is depending only on x. Let's revisit this function.

The Softmax classifier calculates the scoring of the NN. It depends on input x, its paired class y and the scores $f(x) \in \mathbb{R}^n$. Notice that f depends on the weights W of each layer. Indirectly, L is therefore depending on W. A more appropriate notation would thus be $L(x, W)$. Because of the aforementioned, we can calculate the gradient of L in $w \in W$ easily, $\frac{\delta L(x,W)}{\delta w}$. This gradient can then be used to update the weight w to minimize the loss in a future run of the model. This is done using a learning rate η, which allows you to update at a speed to your liking, or, more probable, in the way the network learns best.

When there are multiple layers the gradient will be calculated using the chain rule.

2.2 Introduction to CNNs[1]

The name of the Convolutional Neural Networks (CNNs), comes from one of its layers: the convolutional layer. This layer is created to run well input that has some kind of local structure. The main use is for image analyzation problems, as images often have local patterns. For example, in a face, the eyes, the nose, the mouth are local patterns. The convolutional layer does exactly this: It tries to find features within the picture, locally analyzing.

Another layer that is often seen in CNNs is the max-pooling layer. This layer comes after the convolutional layer, and summarizes the convolutional layer's outputs by solely taking the local maximum for every specified area. We will now formalize these layers.

[1]This part is largely based on [2].

2.2.1 Convolutional layer

In this section we explain the convolutional layer for images. Images can be seen as a set of locations with values representing pixels. These values could be multi-dimensional (e.g. representing the RGB colour model). These dimensions are also known as *channels*. We will keep to a single channel (e.g. black-white levels) image.

Suppose we have a picture of $n \times m$ pixels, all with a real value. This can be represented by a matrix $M \in \mathbb{R}^{n \times m}$ matrix. A convolutional layer consists of multiple *filters*, each representing a feature the network can learn. These filters will analyze the images locally. A filter has a size, the *receptive field*, together with a bias, like in the fully-connected layer, and a *stride*, the distance the filter moves. Suppose we have a receptive field $F \in \mathbb{R}^{a \times c}$ where $a < n, c < m$, a bias b, and stride s. The output P of the receptive field is now a matrix with elements p_{ij} such that

$$p_{ij} = b + \sum_{k=1}^{c} \sum_{l=1}^{a} F_{(k+(i-1)s),(l+(j-1)s)} * M_{(k+(i-1)s),(l+(j-1)s)}.$$

Notice that we must choose $a, c, s \in \mathbb{N}$ such that $n \mod s \equiv a$, and $m \mod s \equiv c$ for everything to fit. This will then also define the size of the output. One issue with the above method is that not all pixels are included in the same number of filter areas, so that the border pixels are weighted less in our analysis. To fix this problem, we can add *zero-padding* to the border of the image, artificially enlarging the image with zeroes. This way the padding will not be analysed as well as the other pixels. As the padding does not contain any information, this does not matter. The padding will increase the size of the image, so the restrictions on the variable a, c, s should be updated. Altogether, for padding p, the size of the output will be a matrix in $\mathbb{R}^{r \times t}$ where $r = (n + 2p - a)/s + 1$ and $t = (m + 2p - c)/s + 1$.

Similar to the fully-connected layer, the convolutional layer's outputs are run through an activation function.

A convolutional layer can have many filters. The number of filters is called the *depth* of the layer. For depth d the output will be off the form $\mathbb{R}^{r \times t \times d}$.

The structure of this convolutional layer brings a couple of properties, which motivates the use of the layer:

1. **Sparse interaction** - the input is not completely linked with all the output. Therefore, less operations are needed, improving run time.

2. **Parameter sharing** - as the filter weights are the same for all the different locations, parameters are shared, so that drastically less memory is used.

3. **Equivariant representation** - translating the input before or after the layer gives the same result. Practically, this means that when the same feature is recorded in different images, but on different locations within the image, this will give the same representations in the output.

For a more detailed explanation of these properties we point the reader to Section 9.2 from *Deep Learning* [2].

Backpropagation works in the same way as for the fully-connected layer.

2.2.2 Max-pooling layer

The max-pooling layer is a layer that reduces the size of the data. Again, this layer has a size, the receptive field, and a stride. This receptive field solely finds the maximum value of the area. This results in every area being represented by the maximum of

that area. For a 2 by 2 receptive field with stride 2, which is most common, the max-pooling layer is the mapping

$$L : \mathbb{R}^{2\times 2} \to \mathbb{R}, \text{ such that } L\begin{pmatrix} a & b \\ c & d \end{pmatrix} = \max\{a, b, c, d\}.$$

Notice the size of the data is halved in 2 dimensions, leaving only a quarter of the size.

When back propagating, it's important to only update the weights through the maximum of the inputs. This is justified by the gradient of $\max(x, y)$ in x (without loss of generality), being

$$\frac{\delta \max(x, y)}{\delta x} = \begin{cases} 1 \text{ if } x \geq y \\ 0 \text{ if } x < y \end{cases}.$$

3 Related work

Cabral and Fonseca [1] prototyped an NN, which we used as an inspiration for our fully-connected layer. Furthermore, Krizhevsky [4] discussed a main issue that comes up with the parallelization of a CNNs. Our code didn't implement the solutions Krizhevsky proposes, which could be a next steps.

4 Our method

The goal of our project is to parallelize a CNN on the GPU. For this we will create a CNN with one convolutional layer, one max-pooling layer, one fully-connected layer, and the Softmax classifier as our loss function (see the appendix for more details on implementation). See figure 1 for the architecture of our network. We run our CPU programmes on a Intel® Core™ i7-9750H Processor, 12M cache, up to 4.50 GHz, 6 cores, containing 12 threads in total. The GPU used was a NVIDIA GeForce GTX 1650 Max-Q with 7.5 CUDA capability, 4GB RAM, 1024 CUDA cores distributed across 16 streaming multiprocessors, and a warp size of 32.

4.1 Parameters

The goal of parallelizing code is to speed it up. After writing our code we ran some tests to compare our sequential code with the parallel code on the GPU. Furthermore, we added the results of parallel code on the CPU, spreading the network over two threads.

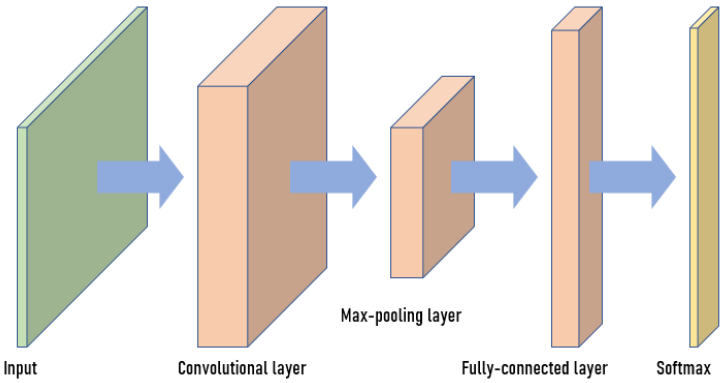

Figure 1: CNN architecture

As we discussed before, there are two reasons for the parallelization to have effect in our network: high data complexity, and high model complexity. For a higher data complexity, within the capacities of our GPU, the convolutional layer would not suffer as it could just create more threads for each image. Each thread in fully-connected layer would need more time to run. For a higher model-complexity it is the opposite, the convolutional layer would suffer, but the fully-connected layer would not. To understand the performance of the GPU version of our network we decided

to test it on varying data-complexity and varying model-complexity, independently. Varying data-complexity can be simulated by changing the batch size, varying model-complexity can be simulated by varying the number of filters (i.e. the depth), which is positively correlated with the number of parameters in both the convolutional layer, as well as in the fully-connected layer. We will show this below.

To train our network, we will be using the MNIST data set [6]. It contains 60000 black-and-white images of hand-written numbers between 0 and 9. Naturally, it has 10 classes.

The data we use is of size 28×28, with only one channel. For the convolutional layer, we don't use any padding, our receptive field is 3×3, the stride is 1, and the depth d varies. This gives an output of size $26 \times 26 \times d$. The activation function we use is the the ReLu activation function ($f(x) = \max\{0, x\}$). For the max-pooling layer, we use a receptive field of 2×2 and a stride of 2. Therefore, the output of the max-pooling layer has size $13 \times 13 \times d$. As the score will be on all 10 classes, the fully-connected layer has a size of $169d \times 10$, with 10 biases. We use a learning rate of 0.1.

4.2 Technical issues

There are two technicalities we would like to point out. First of all, the output of the max-pooling layer and the input of the fully-connected layer do not align. The output of the max-pooling layer has size $13 \times 13 \times 64$, whereas the fully-connected layer has a size of 10816×10, with 10 biases. The biases are no problem, as they are added later. To be able to multiple the max-pooling layer with the fully-connected layer matrix, we need to *flatten* the output data. Given max-pooling output $P \in \mathbb{R}^{13 \times 13 \times 64}$, *np.ravel* flattens for us. We obtain a vector $v \in \mathbb{R}^{10816}$ where $v_i = P_{a,b,c}$, with $a = \lceil \frac{i}{832} \rceil$, $b = (\lceil \frac{i}{64} \rceil \mod 13) + 1$, and $c = (i \mod 64) + 1$. The inverse of this flattening function is straight-forward.

The second technical point concerns the max-pooling layer backpropagation step. As discussed above, the gradient of this function is only non-zero for the maximum input. To be able to obtain the correct gradient, we need look for the max another time, which is not the fastest solution. Other options are to store the index of the maximum for each receptive field.

5 Implementation on the GPU

In this section we discuss the GPU and its advantages and disadvantage. We will also show the reader some of the issue we had during building of the algorithm. After this, we shall present the performance results of the parallelization. Lastly, we shall discuss some future improvement possibilities.

5.1 The Graphics Processing Unit

The first Graphics Processing Units (GPUs) were created in the 80s, but were not accessible by programmers to have anything to say about the actual computations (section 1.3, [9]). These GPUs were primarily created for gaming, quickly computing the colour shadings of pixels in multi-dimensional planes. At the start of the 21$^{\text{st}}$ century, the first programmable GPUs arrived, giving the user the possibility to program the pixel shadings. Quickly after that, people found that these pixels shadings, being numbers, could represent whatever data (section 1.3, [9]). With the arrival of the

CUDA architecture, GPU programming became available to a wider public (section 1.4, [9]).

The main difference between the common Central Processing Unit (CPU) and the GPU is that a CPU is general-purpose, which comes with a higher complexity and more power consumption [7]. This results to a limited number of cores to be bundled within main stream CPUs [7]. On the other hand, GPUs have limited processing capabilities, making it possible to hold many cores within one unit making them suitable for processing highly parallelizable data [7].

Another advantage of the GPU is that we can create multi-dimensional thread blocks. These blocks have multiple advantages. Firstly, coding becomes more straightforward as the location of a computation can be represented by the location of the thread (section 5.3, [9]). This way, each block can run the same function, the so-called *kernel*, but on a different location within the data. Secondly, within a block, we can use shared memory which each thread has much faster access to in comparison to global memory (section 5.3, [9]). We did not explore the use of shared memory, which leaves possible room for improvement.

5.2 Parallelization on the GPU

This leaves the issue of where to parallelize our program. As thoroughly discussed by Krizhevsky [4], the parallelization of CNNs comes with an issue. Krizhevsky argues there is a different type of parallelization possibility in the different layers of a typical CNN. The convolutional layer and the max-pooling layer have a very low number of parameters, with a high number of computations, and the fully-connected layer has a high number of parameters, and a low number of computations. In our example, the convolutional layer has 64 filters, of size 3×3 with a bias, leaving only 640 parameters. The fully-connected layer, on the other hand, has a size of 10816×10, a total of 108160 parameters! This observations leads Krizhevsky to conclude convolutional and max-pooling layers profit most from data parallelization, i.e. letting different threads process different data points, images in our case. Concerning the fully-connected, model parallelization makes more sense, where each thread runs the processes of one or multiple parameters.

Notice that the kernels run on different block sizes. This led us to create multiple kernels, each running a different part of the process, running them in sequence. One kernel for the forward pass of the convolutional and max-pooling layer; one for the forward pass of the fully-connected layer, together with the Softmax classifier; one for the backpropagation of the fully-connected layer; one for updating the gradients of the fully-connected layer, and, lastly, one for the backpropagation and gradients update of the convolutional and max-pooling layers. This could be done in different ways, combining kernels intelligently as suggested by Krizhevsky [4], but this was out of scope for our project.

Lastly, we want to discuss an issue we had with the backpropagation of the convolutional layer. As the to-be-updated parameters should be updated from different threads, we had a concurrency issue. To work around this we calculated the changes for every image and send that back to the CPU, where we summed these changes. Obviously, this could be done in a much smarter way. We could sum them in parallel in multiple ways [3], [8]. This was out of scope for this project. Implementing this would make even better performance possible as there would be no need to return the weights to the CPU and back.

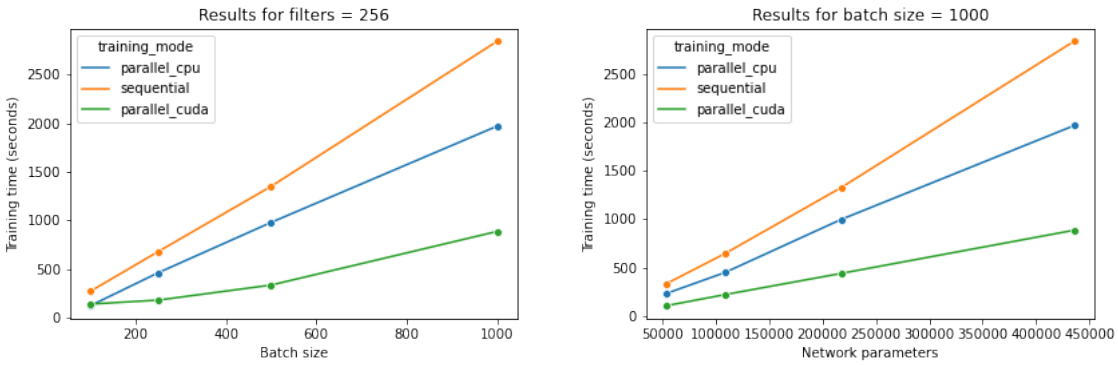

Figure 2: Left: Varying batch size. Right Varying number of filters.

5.3 Results

As discussed in 4.1, we tested the training of the CNN in two experiments. In one experiment, we varied the batch size (100,250,500,1000) for a constant number of filters (256). In the other we kept the batch size constant (1000), and varied the number of filters (32,64,128,256). See the results in figure 2. Notice that the number of filters is translated to number of parameters.

The results are promising. For higher numbers of batch size, as well as a higher number of parameters, parallelization on the GPU gives a better result. Even more so, the improvements also get bigger. CNN training performance was enhanced by parallelizing on the GPU. Still, further improvements are possible.

6 Conclusion

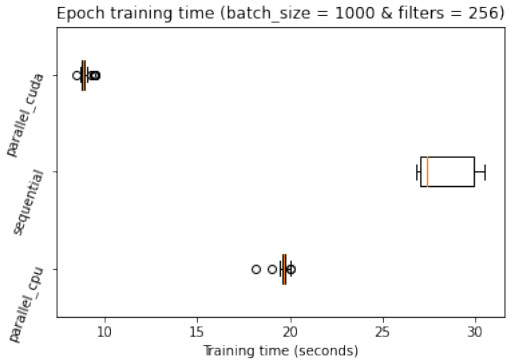

Figure 3: Epoch training time

The goal of the project, writing a parallel programme to train a CNN on the GPU, was accomplished. Furthermore, for certain parametric settings, a 3 times speed up was achieved. See the results in figure 3 where a hundred epochs were analyzed. Still, there are a couple of improvements possible for the Cuda code.

First of all, explore the possibility of using shared memory in the fully-connected layer. Secondly, caching the index of the input that gives the output in the max-pooling layer. Thirdly, sum the changes of the weights, right after the convolutional backpropagation, in parallel on the GPU. With the above implemented, we don't need to copy the weights to the CPU, and back, saving time. In total, the time taken to copy all the data (so not only the weights) from the GPU back to CPU take on average 3 seconds per epoch. This could be improved by writing smarter code. Furthermore, implement Kryzhevsky's proposed solutions for changing between parallelization types [4]. The last on performance enhancement, the arrangement of the Cuda cores should be optimized.

To make the algorithm more resilient to different data sizes and parameters, we should return an error when the data sizes and parameters don't give integer convolutional and max-pooling output sizes. This way the algorithm won't take in all available information from the data. We could fix this by automatically adding padding.

These improvements were out of scope for this project, but could be implemented in future work. This could lead to a prototyping of the CNN, similar to Cabral and Fonseca's work on NNs [1].

Bibliography

[1] B. Cabral and A. Fonseca. Prototyping a gpgpu neural network for deep-learning big data. *Big Data Res. (2017)*, 2017.

[2] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[3] W. D. Hillis and G. L. Steele. Data parallel algorithms. December 1986. URL `doi:10.1145/7902.7903.`

[4] Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks. 04 2014.

[5] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: A llvm-based python jit compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM '15, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450340052. doi: 10.1145/2833157.2833162. URL `https://doi.org/10.1145/2833157.2833162`.

[6] Yann LeCun, Corinna Cortes, and CJ Burges. Mnist handwritten digit database. *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, 2, 2010.

[7] Victor Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu. *ACM SIGARCH Computer Architecture News*, 38:451–460, 01 2010. doi: 10.1145/1816038.1816021.

[8] Yu Ofman. On the algorithmic complexity of discrete functions. *Soviet Physics Doklady*, 7:589–591, 1963.

[9] Jason Sanders and Edward Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 1st edition, 2010. ISBN 0131387685.

[10] G. van Rossum. Python tutorial, technical report. May 1995.

A. Appendix

The gradient of the Softmax classifier in s_i, as defined in Definition 1:

$$\begin{aligned}
\frac{\delta L(x)}{\delta s_i} =& \frac{\delta(-\log \frac{e^{s_i}}{\sum_{j=1}^{n} e^{s_j}})}{\delta e^{s_i}} \\
=& -\frac{1}{\frac{e^{s_i}}{\sum_{j=1}^{n} e^{s_j}}} * \frac{\delta \frac{e^{s_i}}{\sum_{j=1}^{n} e^{s_j}}}{\delta s_i} && \text{chain rule} \\
=& -\frac{1}{\frac{e^{s_i}}{\sum_{j=1}^{n} e^{s_j}}} * (\frac{e^{s_i}}{\sum_{j=1}^{n} e^{s_j}} - \frac{e^{s_i}}{(\sum_{j=1}^{n} e^{s_j})^2} * e^{s_i}) && \text{product rule} \\
=& -(1 - \frac{e^{s_i}}{\sum_{j=1}^{n} e^{s_j}}) \\
=& \frac{e^{s_i}}{\sum_{j=1}^{n} e^{s_j}} - 1.
\end{aligned}$$

In a similar way we can calculate the gradient of $L(x)$ for an s_k where $k \neq s_i$:

$$\begin{aligned}
\frac{\delta L(x)}{\delta s_i} =& \frac{\delta(-\log \frac{e^{s_i}}{\sum_{j=1}^{n} e^{s_j}})}{\delta e^{s_k}} && k \neq i \\
=& -\frac{1}{\frac{e^{s_i}}{\sum_{j=1}^{n} e^{s_j}}} * \frac{\delta \frac{e^{s_i}}{\sum_{j=1}^{n} e^{s_j}}}{\delta s_k} && \text{chain rule} \\
=& -\frac{1}{\frac{e^{s_i}}{\sum_{j=1}^{n} e^{s_j}}} * -\frac{e^{s_i}}{(\sum_{j=1}^{n} e^{s_j})^2} * e^{s_k} && \text{product rule} \\
=& \frac{e^{s_k}}{\sum_{j=1}^{n} e^{s_j}}
\end{aligned}$$

As such, we can easily calculate the gradient G of the whole fully-connected layer, for $(x, c_i) \in X \times C$ (as defined in Definition 1) as:

$$\begin{aligned}
G_j =& \frac{e^{s_j}}{\sum_{k=1}^{n} e^{s_k}} && \text{for } j \neq i, \\
&\text{and} \\
G_j =& \frac{e^{s_j}}{\sum_{k=1}^{n} e^{s_k}} - 1 && \text{for } j = i.
\end{aligned}$$

In our code we add a translation.