Source Localization
by Inverse Diffusion
and Convex Optimization

GABRIEL BENGTSSON
JESPER LARSSON
Källokalisering med inversdiffusion och konvex optimering

GABRIEL BENGTSSON
JESPER LARSSON
Sammanfattning

Målet med rapporten har varit att lokalisera källor till en emitterad substans med hjälp av en algoritm för inversdiffusion. Denna algoritm har gett lyckade resultat när den applicerats för biologisk detektering av celler och det har vidare föreslagits att körtiden skulle kunna reduceras avsevärt om algoritmen implementeras som en beräkningsgraf. Detta skulle automatisera beräkningar av gradienter och tillåta snabbare exekvering på en grafikprocessor.

För algoritmens implementation användes TensorFlow, som primärt är ett programmeringsbibliotek inriktat mot maskininlärning. Datagenererade testbilder av biologiska prover användes sedan för att utvärdera körrsultatet med hjälp av mjukvara för bildanalys samt prestandamått.

Jämförelser visar att implementationen i TensorFlow ger resultat som överensstämmer med den traditionella implementationen av samma algoritm. Sett ur ett bredare perspektiv visar detta på möjligheten att använda beräkningsgrafer för att lösa storskaliga optimeringsproblem och mer specifikt inversproblem.
Source Localization by Inverse Diffusion and Convex Optimization

Gabriel Bengtsson and Jesper Larsson

Abstract—The purpose of this project was to implement an inverse diffusion algorithm to locate the sources of an emitted substance. This algorithm has yielded successful results when applied to biological cell detection, and it has been suggested that the run time could be greatly reduced if adaptions for a computation graph framework are made. This would automate calculations of gradients and allow for faster execution on a graphics processing unit.

The algorithm implementation was realized in TensorFlow, which is primarily a machine learning oriented programming library. Computer-generated biological test images were then used to evaluate the performance using regular image analysis software and accuracy metrics.

Comparisons reveal that the TensorFlow implementation of the algorithm can match the accuracy metrics of traditional implementations of the same algorithm. Viewed in a broader scope this serves as an example to highlight the possibility of using computation graph frameworks to solve large scale optimization problems, and more specifically inverse problems.

I. INTRODUCTION

TESTING and monitoring immune systems is not a trivial task since it needs to be performed in a very complex system of biological processes in a living body. Generally, scientists want to see the response of a specific part of the immune system, but this may be difficult due to the overall complexity of the system. Therefore, a way of testing specific responses in an isolated environment is preferred, where there is no need for invasive procedures or dangerous side effects on the examined host system. Enzyme-Linked ImmunoSpot (ELISPOT) and its modification FluoroSpot are ways of doing this with high sensitivity and accuracy [1] [2]. ELISPOT and FluoroSpot assays provide methods for analyzing the response of the immune system in humans and animals. In the beginning, the main use was detection of enterotoxins produced from bacteria, only later moving on to specific antibody-secreting lymphocytes and cytokines emitted from cells [3].

This project deals with the problem of localizing active cells emitting cytokines. The data in the form of photographs acquired from the medical assays are hard to accurately analyze, and one is usually compelled to do the analysis with poor algorithms producing unsatisfactory results. This process can be improved and automated using the method described in [4]. The problem can be posed as an inverse problem where we have the final state of a process but want the initial state. This case we have the positions of captured particles, but we want the positions of the cells that have emitted the particles. Since the particles move by Brownian motion before capture, this problem can be modeled using a diffusion partial differential equation and, furthermore, the effect of the diffusion can be approximated using Gaussian kernels of varying sizes, where a detailed explanation can be found in [4].

Since we can approximate the physical phenomena that the particles undergo, this problem can be solved with optimization. The accelerated proximal gradient algorithm (APG) can solve the problem and be implemented in a TensorFlow interface [5]. The algorithm involves calculating a gradient, which is normally cumbersome to implement in code. Fortunately, TensorFlow is equipped with a built in gradient tool that allows automatic calculation of gradients given an existing well-defined function. Many algorithms, including previous solutions to the source localization problem, have adopted manual differentiation, meaning a broad group of similar algorithms could benefit from a new implementation in a TensorFlow environment. Our goal has been to explore the possibility of implementing an optimization algorithm for large scale problems in TensorFlow.

II. THEORY

In this section we will clarify the concepts needed to implement a working inverse diffusion program in Python and TensorFlow. We will start by explaining the biomedical background, continue with the key mathematical concepts and the APG algorithm to finally finish with a way of measuring binary classification performance.
A. FluoroSpot
FluoroSpot works by suspending cells in a layer of agar above a layer of immobilized antibodies at the bottom of a microplate well. When the cells release particles, these undergo Brownian motion in the agar medium until they encounter the antibodies at the bottom and get captured [4]. The captured particles can then via a biological process be combined so that they react to light of a certain wavelength and become fluorescent. The samples are then photographed while being illuminated with a monochrome light and, because of the fluorescence, the captured particles can be observed, as in Figure 1. These images are then used for localizing positions of the sources, which, in this case, are the active cells.

B. Convolution
One of the main concepts when performing the inverse diffusion algorithm is the convolution. This is a mathematical transformation that can be interpreted as a way to mix two functions. It is visually represented by allowing one function to slide over the other one, where the convolution is the area under each product of the functions [6].

There are several types of convolutions such as the continuous integral convolution and the discrete convolution. This project solely uses the discrete version in the form of discrete Gaussian kernels which are isotropic (rotationally invariant).

C. Gaussian Filters
When applied through a convolution, a Gaussian filter $G_\sigma$ essentially blends the image together and removes high frequency signals.

$$G_\sigma(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2}{2\sigma^2}} \quad (1)$$

$$G_\sigma(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \quad (2)$$

Here, (1) is the Gaussian kernel for a 1-dimensional filter. It is normalized and thus an integral from $-\infty$ to $\infty$ over the kernel will have the value of 1. To get the Gaussian kernel for 2 dimensions we can simply multiply a vertical and horizontal version of the 1-dimensional kernel which yields the result in (2). We use the Gaussian kernel to approximate diffusion in the $xy$-plane as described in [7], where the $x$ and $y$ coordinates are defined relative to the center point of the filters.

When used as a discrete convolution, the Gaussian blur is usually applied as a 2D-array. However, since the 2D-kernel is the product of two 1D-kernels due to separability we can apply the convolution in two steps, one for the horizontal kernel and one for the vertical kernel. If the original matrix is $m \times n$ the computation will involve $mn$ calculations for every output pixel, but if we use the separated method this changes to $m+n$ calculations. With the maximum kernel dimensions used in the paper $m, n = 361$ as an example, a separation means that the number of calculations is reduced by a factor of $mn/(m+n) = 361^2/722 \approx 180$.

D. Computation Graphs
By defining mathematical operations as vertices connected by edges, these vertices can be linked together in order to form a graph. If the flow through this graph is set up in a way such that the movement across each edge is unidirectional and no edge loops back to a previous vertex the graph is said to be a computation graph [8]. The use of these and their broader generalization in computational networks have become fundamental throughout machine learning applications that rely on neural networks [9]. Apart from the mentioned areas of use it has also been suggested that computation graphs can be successfully applied to optimization problems, which enables computations to easily be carried out on different platforms such as GPUs and computer clusters. In addition to this the computation graph implementation has been indicated to outperform traditional methods when performing calculations on large scale problems [8].

E. Accelerated Proximal Gradient Algorithm

Algorithm 1 Discretized Inverse Diffusion Algorithm

1: $\hat{b}^{(0)} \leftarrow \hat{a}^{(0)}$, $i \leftarrow 0$
2: repeat
3: $i \leftarrow i + 1$
4: $\hat{a}^{(i)} \leftarrow \hat{b}^{(i-1)} - \frac{\eta}{2} \nabla \hat{a} f(\hat{b})$
5: $\hat{a}^{(i)} \leftarrow \hat{a}^{(i)}$
6: $\hat{p} \leftarrow \left(1 - \frac{\eta}{2} \lambda \left[\sum_{k=1}^{K} \left(\hat{a}^{(i)} \right)^2 \right]^{-1}\right)$
7: for $k = 1 \text{ to } K$
8: $\hat{a}^{(i)} \leftarrow \hat{p} \odot \hat{a}^{(i)}$
9: end for
10: $\hat{b}^{(i)} \leftarrow \hat{a}^{(i)} + \alpha(i)(\hat{a}^{(i)} - \hat{a}^{(i-1)})$
11: until convergence
12: $\hat{a}_{opt} \leftarrow \hat{a}^{(i)}$

The accelerated proximal gradient algorithm derived in [4] and further discretized in [7] can be expressed as in Algorithm 1. The variable $\hat{a}$ is a 3-dimensional tensor representation of the image that will eventually be output as $\hat{a}_{opt}$ and further analyzed for cell detection purposes. The three dimensions of $\hat{a}$ comes from the stacking of multiple 2D-images, allowing Gaussian filters of different sizes to be separately applied to each 2D-image. The initial state $\hat{a}^{(0)}$ is comprised entirely of zeros. To increase the convergence speed the term $\hat{b}$ is calculated, including $\alpha(t) = \left[\frac{t(i-1) - t(i)}{i-1}\right]$ where $t(i) = \frac{1}{2} + \frac{i}{4} + t^2(i - 1)^{-1/2}$. This forms the acceleration part of the algorithm.

The main objective is to minimize the sum of a cost function and a regularizer, the former being

$$f(\hat{a}) = \left\| \hat{w} \odot \left(\hat{a}_{obs} - \sum_{k=1}^{K} \hat{g}_k \odot \hat{a}_k \right) \right\|^2_2 \quad (3)$$

with the K different layers of Gaussian kernels $\hat{g}_k$. The variable $\hat{w}$ is a weight parameter, which for the purposes of this project is in its simplest possible form as a tensor where all elements
have a value of 1, and $\tilde{d}_{\text{obs}}$ is the observed unaltered image. In the summation term of (3) a blurring operation is applied to $\tilde{a}$, and furthermore

$$\tilde{d}_{\text{obs}} = \sum_{k=1}^{K} \tilde{g}_k \ast \tilde{a}_k$$

(4)

is the condition for the global minimum of $f(\tilde{a})$. This equation is only true in a situation where the perfect $\tilde{a}$ is found with respect to the Gaussian kernel discretization.

$$\mathcal{R}(\tilde{a}) = \lambda \sum_{m=1}^{M} \sum_{n=1}^{N} ||\tilde{a}_{m,n}|| + \delta_{\text{act}} \cdot \text{max}_{K \times M \times N} (\tilde{a})$$

(5)

A regularizer used to steer the algorithm towards physically reasonable solutions is built into the $\tilde{p}$ variable. The viable solutions for this application are point-like sources representing biological cells. The proper mathematical expression for the regularizer used in this project can be seen in (5), with $M$ and $N$ as the image dimensions, while the $\delta$-function limits each element in $\tilde{a}$ to be a positive real number. Included in both the regularizer and its equivalent in Algorithm 1 is the adjustable user parameter $\lambda$. Additionally, line 6 of Algorithm 1 holds another parameter $\eta$ precisely selected to ensure convergence. The adjustable parameters have been chosen in [7] to be $\lambda = 0.5$ and $\eta = 1/\sigma_{\text{max}} \| \tilde{a} \|^2$ with the latter being simplified to $\eta = 1/\sigma_{\text{max}}$ as the weights are all 1. Here, $\sigma$ are the standard deviations for the Gaussian kernels as discretized in [7], which for $K = 8$ yields $\sigma_{\text{max}} \approx 64.7$. Furthermore, the K sigmas $\sigma_k$ actually applied in the algorithm are averaged from these $\sigma$ such that $\sigma_k = \{3.65, 7, 11, 18, 28, 38, 48, 60\}$, where $k = 1, 2, 3, \ldots, 8$.

As for the mathematical operations in the algorithm, the subscript (+)-operator is a positive clipping changing all negative elements to zero, the $\ast$ is a convolution and $\odot$ is element-wise multiplication.

Viewing the algorithm as a whole it can be thought of as inverse diffusion taking place in multiple layers simultaneously. These layers are then merged to form the final image. Each layer corresponds to a time frame, where larger $\sigma$-values in the Gaussian filters mean that effects over longer periods of time are simulated. This repeated merging of layers over thousands of iterations forms the basis for the source localization algorithm and is required to enable computer software to accurately process the images.

F. $F_1$-score

The $F_1$-score is used in statistics to determine a test’s accuracy by measuring the number of true positives (TP) which are correct identifications, false positives (FP) which are incorrect identifications and false negatives (FN) which are unidentified true data points. There is one additional complementary type: true negatives (TN), which are all false points that have been correctly unidentified. The TP, FP and FN values from a test are then used to calculate the $F_1$-score as follows.

$$\text{precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}, \quad \text{recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

(6)

A perfect $F_1$-score of 1 can only be achieved with solely true positives being detected and the worst case $F_1$-score of 0 is only possible with no true positives. We use $F_1$-scores to compare our results to the cell counts of other implementations.

III. IMPLEMENTATION AND EVALUATION

A. The Algorithm as a Computation Graph

Figure 2 shows the general outline of our computation graph implementation, and the order in which the graph is to be traversed when executed. Each of the nodes consist of multiple mathematical operations combined to build the more complex expressions seen in Algorithm 1. The nodes in the middle column represent the convolution, gradient and regularizer with their associated auxiliary operations, including modifications of tensor shapes. The Speedup node is a direct representation of the acceleration part of the APG, which is found on line 10 of Algorithm 1. Moreover, the image on which the algorithm is to be run is converted to a tensor format by the Image node, while the Variables node handles instantiation of variables, and their transfer across iterations.

In the Theory section it is stated that computation graphs are acyclic, while our graph implementation clearly is not. This contradiction is due to the graph in Figure 2 representing the computations as they are performed over a number of iterations. The connection from the Variables node leading to the Convolution and Gradient nodes is there exclusively to enable stored values to be used in subsequent iteration cycles. If only a single iteration of the algorithm was to be performed, the forward connection originating in the Variables node would not be there.

Even though the edges between nodes have an explicit orientation as displayed through the arrows, the order in which sections of the graph are executed is not well-defined. This execution order seems to be internally decided by TensorFlow and may give rise to errors if not controlled correctly. An
example highlighting the problem is encountered when a variable is reassigned. Other operations depending on the reassigned variable seem able to access this variable either before or after the reassignment occurs, which affects the calculation results.

B. TensorFlow Implementation

All of the code we have written and used to generate the results discussed later can be found in [10].

The general procedure for the code is as following. In the first step the observed image is loaded using ImageMagick in the form of Pillow, a fork of the Python Imaging Library. The next step is to generate the Gaussian filters that we use for the convolutions, which is done in a separate file, where filters of sizes up to $6 \cdot 60 + 1$ in length are created. Our filters are then weighted with $(\Delta_k)^{1/4}$ where $\Delta_k = \{2, 7, 4, 10, 10, 10, 10, 14\}$ and $k = 1, 2, 3, \ldots, 8$. This weighting is due to the diffusion density, with values based on the original $\tilde{\sigma}$ discretization presented in [7]. After we have done all necessary data pre-processing we can start constructing the TensorFlow graph.

As an initial step the required variables are declared, before moving on to the mathematical operations of Algorithm 1. The two 1-dimensional convolutions are performed with TensorFlow’s tf.nn.conv2d in two steps as described in the Theory section. During the convolution the data structure is changed to [Batch, Channels, Height, Width] from the usual [Batch, Height, Width, Channels] due to the GPU being more efficient in performing calculations on the former [11].

The cost function is defined as the $\ell^2$-norm of the difference between the current state of $\tilde{a}$ passed through the convolution and the original image. This is done with the convolution we just constructed and the function tf.nn.l2_loss which takes care of calculating the squared norm. The output of tf.nn.l2_loss is half of the squared $\ell^2$-norm value so we multiply it by a factor 2 to compensate. The gradient in TensorFlow uses automatic differentiation on the previous operations defined in the computation graph and is thus very easy to use. We use TensorFlow’s tf.gradients to minimize the cost function we defined in (3) with respect to $B$. This gradient calculation needs to be performed with respect to $B$, as otherwise TensorFlow will not be able to perform the automatic differentiation. The code snippet directly below shows the final step involved in the cost function calculation followed by the gradient calculation, corresponding to the last term of line 4 in Algorithm 1. It also demonstrates how name scopes are adopted to organize operations in a more structured way, and how it is possible to modify tensor dimensions through indexing as in tf.gradients(cost, B)[0].

```python
    with tf.name_scope('gradient') as scope2_3_1:
      gradient_step = tf.gradients(cost, B)[0]
```

This mixing provides a speedup in the convergence of the algorithm. After the mixing is done we save the results in variables for the next iteration to use. The following section of code shows how the variables $t$ and $\alpha$ from line 10 of Algorithm 1 are determined, but also illustrates the critical concept of controlling the execution order of operations in the computation graph. In this example, the tf.control_dependencies() scope is used both to guarantee that $t$ is assigned before $\alpha$ and that $\alpha$ is assigned before $t_{old}$, where $t_{old}$ stores the current value of $t$.

```python
    t = tf.assign(t, 1 / 2
                  + tf.sqrt(1 / 4 + tf.square(t_old)))
    with tf.control_dependencies([t]):
      alpha = tf.assign(alpha,
                        tf.subtract(t_old, 1) / t)
      with tf.control_dependencies([alpha]):
        t_old = tf.assign(t_old, t)
```

After defining the computation graph, it is time to actually run the graph we have constructed. Before we can run the graph we need to create a session with tf.Session() and then initialize all variables and constants, which is done with tf.global_variables_initializer(). After the initialization we run the graph by passing Python lists of all operations we have defined to session.run(). This is done at several places throughout the code, with one such location shown in the code snippet below. Here, difference_run handles the cost function and gradient calculations, alongside multiply_run which contains the multiplication of $\tilde{p}$ and $\tilde{a}_k$ from line 8 of Algorithm 1.

```python
    session.run([convolution_run, difference_run,
                 regularizer_run, multiply_run,
                 speedup_run])
```

The data stays inside the TensorFlow back-end until we run commands to extract it, which is done at regular intervals to plot graphs shown later in the paper. After we have run $10^4$ iterations the final image is extracted by squaring all elements of $\tilde{a}$, then summing the 3D-array down to a 2D-array before taking the square root of all elements. The final step in the image processing is to normalize the pixel values to $[0, 255]$ by using linear scaling.

Our code contains some additional features like estimates of remaining run time and graph plotting but those are not the focus of this report and as such will not be explained in any greater detail. One key feature to point out however is summaries, which are functions that gather data used for plots.

C. Running the Program

We performed most of our runs on one of our laptops with the following specifications: Intel Core i5-8250U, 8 gigabytes of RAM and an Nvidia GeForce MX150. The laptop was enough for testing the program, but for the final runs we had access to a powerful server with an Nvidia Tesla P100. Furthermore, the laptop runs version 1.6.0 of TensorFlow, while the server only runs version 1.0.1. Each run of $10^4$ iterations took about 45–50 minutes on average (0.27–0.30 seconds per
F2A. SOURCE LOCALIZATION BY INVERSE DIFFUSION

Figure 3 shows some metrics during runs on the simulated images with 250, 750 and 1250 cells. The cost function represents the $\ell^2$-norm of the difference between the observed image and the blurring operation applied to $\tilde{a}$ as in the right-hand side of (4). Put in other words it shows how far off our approximation of the original state is. The normalized prediction’s square error (NSE) is a modification of the cost function which is essentially obtained by dividing the cost function by the norm of the observed image, expressed mathematically in (8).

$$\text{NSE} = \frac{\|\tilde{d}_{obs} - \sum_{k=1}^{K} \tilde{g}_k \otimes \tilde{a}_k\|^2}{\|\tilde{d}_{obs}\|^2}$$  \hspace{1cm} (8)

The group sparsity regularizer (GS) graph shows a square norm measurement of the tensor $\tilde{a}$

$$\text{GS} = \sum_{m=1}^{M} \sum_{n=1}^{N} \left( \sum_{k=1}^{K} \tilde{a}_k^2 \right)$$  \hspace{1cm} (9)

which is the core of the regularizer implicitly used in line 6 of Algorithm 1.

D. Cell Detection

When the run of the inverse diffusion algorithm is completed the cell locations are obtained. This is done using the function peak_local_max included in the Python package scikit-image, which can be set to find all pixels whose 8 surrounding pixels have a lower value than the pixel itself [12]. The majority of image locations returned by this function will be from faint pixels which for the most part are image artifacts created during the inverse diffusion. To filter out the false detections a threshold value is chosen by employing a separate optimization algorithm which maximizes the $F_1$-score with respect to the threshold. The threshold is then applied to the local maxima locations, removing them if their pixel value is below the threshold value.

Figure 4 shows a small section of the graphical output from the cell detection program. Furthermore, the $F_1$-scores calculated with the program have been tabulated in Table I.

### IV. RESULTS AND DISCUSSION

In the rightmost image of Figure 4 there are examples of the three binary classifications TP, FP and FN. The fourth type, TN, is expressed indirectly by all white areas correctly ignored by the algorithm. In the same image two examples of off-centered TP detections can be seen in the top section,
where the inverse diffusion algorithm converged to a point near the true cell location. The chosen segment is representative in that most detections are true positives in the complete output image as well, meaning the output seems reasonable. The reliability of the output is further strengthened by the similarity with results presented in [7]. This resemblance is present when comparing output images, but also when matching the values for 750 cells in Figure 3 and the obtained F1-scores with their equivalents in [7]. A significant difference between our results and the original is the discrepancy in NSE graph values. This has multiple possible explanations, such as the use of slightly different Gaussian matrices compared to the original implementation, or the possibility of an incorrectly implemented NSE calculation in our code.

In Figure 3b we see that the algorithm converges at around 8000 iterations with regards to the cost function. Because of this it is rather unnecessary to run the algorithm any further than $10^4$ iterations as there will probably not be any improvements in the results. The fact that the GS graph values converge at around 8000 iterations as well confirms that the algorithm does not benefit from additional computational time after that point.

The running time of the algorithm is far from perfect. Even when considering the fact that most algorithm runs are executed on a laptop, the run speed is slower than expected. The new implementation should be exceptionally fast compared to the old implementation in Matlab as TensorFlow’s computation graph enables a significant improvement in calculation efficiency [8]. Our initial guess trying to explain this discrepancy is that we could improve the process by using operation fusing and a more optimized compiler for special GPU-operations [11]. However, despite the extended processing time for images, the method still has a great potential compared to traditional approaches, e.g. since the new method can easily be transferred to a cloud computing network like Google’s tensor processing unit Cloud TPU. Even though the run time is an easy metric to compare one should not forget that the primary focus of this work was to implement the algorithm as a computation graph, not to improve the speed of the program. The main benefit of using the automatic differentiation in TensorFlow is the ease of use and not its efficiency while running. This advantage in terms of usability should allow for swift prototype testing and proofs of concept for new solution approaches. Should a method be confirmed to function properly in a TensorFlow environment it can then be refined on other platforms.

The speedups seen from disabling the summaries were expected because they acted as interruptions in the main iteration loop. In contrast, the limited utilization is unexpected and much more difficult to diagnose and fix because it probably is due to the general structure of our computational graph, meaning it could necessitate fundamental structural changes to fix.

One possible way of speeding up the program is to use fast Fourier transform (FFT) for the convolutions [13]. We have not had the time to explore this path and we are not completely sure that TensorFlow supports gradient computation of FFT operations as of this report’s writing. Implementing the convolution with FFT could speed up the calculation because then the image and the filters we created earlier would be transformed to the frequency domain. In the frequency domain a convolution is simply an element-wise multiplication which is less computationally heavy than the regular convolution. The result would then be transformed back using the inverse Fourier transform. These three computations have a lower total computational complexity than the normal convolution, giving an advantage when facing large data where the setup cost from the transformation step is negligible.

Through the implementation of the algorithm a few important details worth considering has surfaced. One such aspect is the need to use `tf.control_dependencies()` to control the order in which operation graph nodes are executed. For some
parts of the computation graph the order of operations seems to be crucial for the functionality of the program, as the desired outcome is not attained otherwise. In other code sections it has been possible to omit the tf.control_dependencies() scope without affecting functionality. Basic profiling has revealed that this manipulation of the operation order seems to have a negative impact on the performance of the code, increasing the run time. However, this effect on performance has not yet been fully explored and is left for future work.

Another general point to note is the strict dimensional requirements imposed on the tensors involved in some operations. Specifically, the multi-layered convolution operations expect a 4-dimensional tensor, which forces all previous tensors leading up to the convolution operation to be 4-dimensional as well. These requirements are met by choosing correct dimensions when declaring variables and changing shapes using tf.reshape() or indexing.

A more obvious yet critical aspect is related to the format in which images are saved after they have been processed. Some image formats like JPEG use compression to reduce data size, with the downside of simultaneously deteriorating the quality. This in turn means that artifacts are introduced, which would be detrimental to the output images as these are highly sensitive down to the single pixel level. To eliminate this source of error, formats implementing lossless compression should be used, like PNG or TIFF.

V. SUMMARY AND CONCLUSIONS

A. Summary

We managed to achieve the main goal of this project which was to implement the customized APG algorithm using TensorFlow. We used our implementation to simulate inverse diffusion on provided synthetic examples where the results were satisfactory, with the final images having a limited number of bright spots where the cells are located. In addition to this we implemented the F1-scoring and checked that our implementation matched the old implementation in terms of accuracy. While the results from running the program are fully satisfactory, the running time is not, meaning further optimization is possible.

B. Conclusions and Future Work

TensorFlow has proved to be an efficient framework for solving optimization problems. This usage is in contrast with machine learning, which is what we perceive to be the most common usage of TensorFlow. The success in employing the algorithm is a great showcase for the potential of the computation graph frameworks and their ability to solve large scale inverse optimization problems. The conceivable applications are numerous, as inverse problems include all situations with a known outcome and unknown initial conditions. Some of the more interesting applications are studies of ocean dynamics, air pollution, aeronautics, astrophysics and finance, all containing problems that could potentially be adapted to fit in a computation graph [14] [15]. However, for this to be an attractive option, actions will need to be taken to enhance performance. This could potentially be done through the already addressed merging of operations through operation fusing. Utilizing the FFT or different kinds of GPU-specialized compilers may also be valuable. Other ideas include experimentation with the intuitively chosen discretization of the Gaussian kernels to find better efficacy. Apart from the exploration of operation order restrictions, it is likewise imaginable that other, not yet contrived improvements of the computation graph composition, may provide further efficiency gains. If some, or all, of the above advancements are completed, the computation graph framework has the potential to surpass other implementations designed to solve optimization and inverse problems.

ACKNOWLEDGMENTS

The authors would like to especially thank Pol del Aguila Pla for always being available for questions and providing much needed feedback on the work process. We would also like to thank Joakim Jaldén for arranging such an interesting project, and Vidit Saxena for helping us understand TensorFlow.

REFERENCES
