MOSES Data Inversion With Convolutional Neural Networks

Roy Smart

Montana State University
roytsmart@gmail.com

I. Introduction

The *Multi-Order Solar EUV Spectrograph* (MOSES) is a unique instrument developed by Charles Kankelborg's research group at Montana State University that captures solar images in three diffraction orders: $m = 0, +1$, and -1. Unlike most spectrographs, MOSES is not equipped with a slit to restrict the field of view on the dispersion axis. The advantage of this configuration is that spectral and spatial information is simultaneously viewed by the instrument, allowing interesting solar features to be more easily identified. However this lack of spatial restriction by a slit means that spectral and spatial information are convolved together to form the images captured by MOSES. These images are known as *overlappograms*.

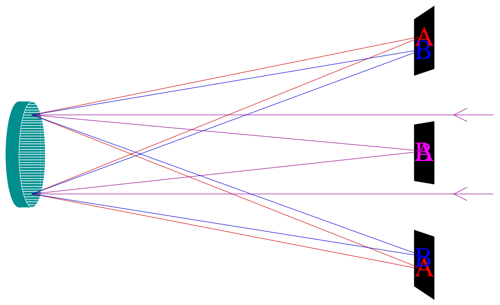


Figure 1: *Layout of the MOSES instrument demonstrating how different wavelengths are overlapped onto each detector [1].*

The main objective of the MOSES instrument is to determine Doppler shifts of the structures observed in the solar transition region. To find this quantity, we must perform what we call an *inversion*. This is best described as taking the 2-dimensional spectral and spatial information from each of the three detectors and constructing a spectral *cube* in three dimensions, with two spatial dimensions and one spectral dimension.

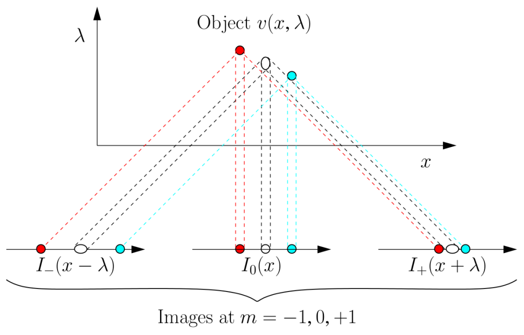


Figure 2: *Diagram demonstrating how compact objects in the data cube are convolved to form the images at each detector [1].*

The main challenge with inverting the MOSES data is that it is an obviously ill-posed problem, i.e. the spectral cube contains more information than is provided by the instrument. This is mathematically described using the expression

$$I_m(x', y') = \int_B v(x' - m\lambda, y', \lambda) d\lambda. \quad (1)$$

Where $I(x', y')$ is the intensity measured by the instrument, $v(x', y', \lambda)$ is an object located in the spectral cube, m is the spectral order, (x', y') are the detector coordinates, and the domain B is the passband of the instrument.

Since Equation (1) doesn't have a unique solution, there are many possible cubes that could produce the same images captured by MOSES. Consequently, we must use physical constraints to attempt to trim down the large number of potential solutions to the inversion problem.

II. Motivation

Many researchers throughout the history of the MOSES research project have developed computational tools to solve the inversion problem. These methods include Smoothed Multiplicative Algebraic Reconstruction Technique (SMART)[2], Fourier backprojection and pixon inversion [3] and are discussed in detail in the literature.

The above methods have proved successful in revealing the interior structure of the transition region [1]. However there is still room for progress to be made towards full inversions of the entire MOSES dataset. These algorithms have a tendency to produce what is known as *plaid*. This is a phenomenon where bright objects on the spectral cube tend to be smeared across the direction of the spectral projections, producing decidedly unphysical inversions.

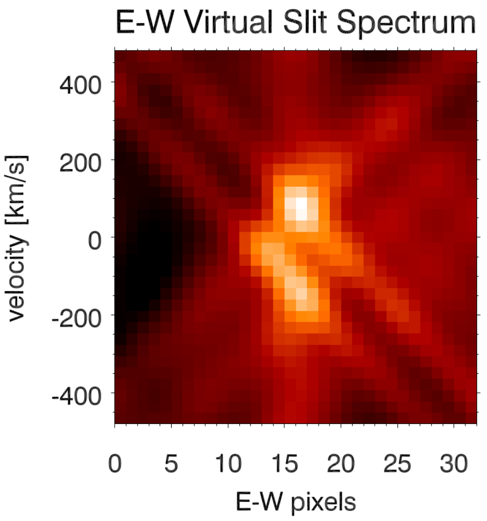


Figure 3: *An example of plaid. Notice how the bright pixels in the center are smeared diagonally (projection dimension) across the image. This is an example of an unphysical inversion. [4]*

Developing an inversion method that is resistant to plaid would be very beneficial, as such an inversion would allow researchers to extract more scientific information about the structure of the transition region.

III. Proposed Inversion Method

I. Machine Learning

Thus far, all of the attempts to invert the MOSES data have relied on traditional, human-designed algorithms to carry out the computation. However, in the past few years, great strides have been made in machine learning, where a computer program is trained to solve a particular problem by providing examples of the problem along with each corresponding solution. Feedback is then provided to the network so that it may utilize minimization techniques to converge on a solution.

We propose that a method based off of machine learning might allow some progress to be made towards the goal of plaid-resistant MOSES inversions. This is made possible by the fact that the learning algorithm could be penalized for producing plaid solutions to the inversion problem and rewarded for producing physical solutions.

While machine learning sounds like a fantastic approach to problem solving, there is still a great deal of challenges to contend with. First, the structure of the learning algorithm is very important in determining its problem-solving ability, and unlike traditional algorithms, the right structure can often only be found by trial and error. Furthermore, the learning process takes a large number of examples to converge on the solution. Solving these problems will be discussed in the following pages.

II. Artificial Neural Networks

Artificial neural networks (ANNs) are machine learning algorithms that are modeled off of the structure of the animal brain. ANNs can be described as a system of *neurons* that have the ability to communicate with each other. The behavior of each neuron is dictated by an activation function. This function maps neuron inputs to outputs, e.g. determines when the neuron is *active*. Active neurons propagate information through the network while dormant neurons restrict it.

Communication is facilitated through a series of connections between the neurons, where each connection has an associated weight representing the amount of influence one neuron may have on another. The weights are tuned based off of experience and taken with the activation function, allow the neural network to learn.

The neurons are usually organized into layers, where each layer can be interpreted as a different computational step used to solve the problem. The first layer is known as the input layer, and the number of input neurons corresponds to the amount of information supplied to the network. The output layer is the last layer, and the number of output neurons equals the amount of information to be expected in the solution of the problem being solved. In between the input and output layers, there are a number of hidden layers used to compute the solution. The amount of hidden layers and the number of neurons per hidden layer depends on the complexity of the problem.

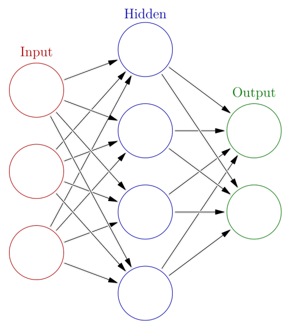


Figure 4: *An example of a simple neural network.*

Using this simple idea, many different species of neural networks may be constructed. The choices of network depth, number of connections, and activation function are examples of what are known as *hyperparameters* which determine the behavior of the network.

III. Convolutional Neural Networks

A. Convolutional Layer

Vanilla ANNs usually connect each neuron in one layer to every other neuron in the preceding layer. This configuration is sufficient when the number of input neurons is small. However, as the number of input neurons increases (for example, as large as an image), the huge number of free parameters quickly leads to overfitting. Furthermore, the Vanilla ANN treats pixels with large spatial separation equally, which is often unnecessary for image processing where close spatial relationships are more important.

Convolutional Neural Networks (CNNs) solve this problem by connecting only a small number of the input neurons (known as the receptive field) to the layers below, known as convolutional layers.

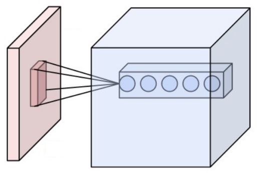


Figure 5: *The red box represents the input neurons, with the receptive field connected to the next five convolutional layers.*

The convolutional layer can be seen as a series of orthogonal filters, with each filter designed to identify a particular feature in an input image. An example of these orthogonal filters can be seen in Figure 6.

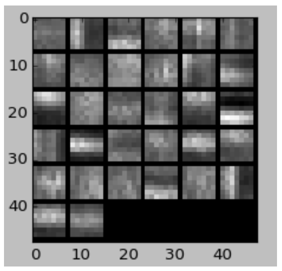


Figure 6: *Example of orthogonal filters in convolutional layer. NOTE: This is not from our network.*

Each filter size corresponds to the size of the receptive fields. The receptive fields overlap, so each filter is applied to every pixel in the input image. Weights are shared between each neuron in the filter so training time can be greatly reduced.

B. Max Pooling Layer

After each convolutional layer, the information is fed into a max pooling layer. This layer has the effect of increasing the non-linearity of the network, while also decreasing the amount of neurons to control overfitting. Max pooling layers divide up the output of the convolutional layer into zones, and then selects the maximum value within each zone. This can be visualized using Figure 7.

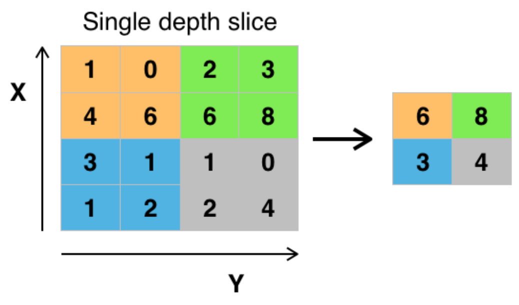


Figure 7: *Max pooling with a 2x2 kernel and a stride of 1.*

Max pooling layers may not be the right choice for the ideal network. Since the inversion problem is already ill-posed, layers that lose information such as the max pooling layer may not help the network converge on a solution. We have tested the current iteration of the network both with and without max pooling layers and it didn't appear to change the accuracy or training time of the network; however training the network on more complicated datasets will determine the usefulness of the max pooling layer.

C. Inner Product Layer

After several iterations of convolutional and max pooling layers, the data is fed into an inner product layer, also known as a fully-connected layer. The neurons in this layer are connected to every neuron in the layer above, as in Figure 4. This layer serves to provide high-level reasoning and logical capabilities to the network.

D. ReLU Layer

Between each inner product layer, we have a so-called Rectified Linear Unit (ReLU) layer. This layer adds further non-linearity to the decision making capacity of the network. The ReLU operation consists of applying the non-saturating activation function

$$f(x) = \max(0, x) \tag{2}$$

to each neuron in the layer. This layer has the added benefit of maintaining positivity, an important characteristic of solutions to the inversion problem.

E. Loss Layer

The loss layer drives the training of the network. The output of the last inner product layer (the proposed solution) is fed into the loss layer which is compared to the actual solution (known hereafter as *truth*) to determine how well the network performed. This quantified using what is known as Euclidean loss, defined by

$$r(\mathbf{p}, \mathbf{q}) = \frac{1}{2N}\sqrt{\sum_{i=0}^{N}(q_i - p_i)^2}. \tag{3}$$

This quantity is then used in the backpropogation phase of the training (explained below) to inform the network how it should adjust the weights.

F. Forward/Backward Propogation

Training the network involves two phases. The first phase, known as forward propagation, takes input and uses the neural network to compute the solution for that input. Forward propagation is used both for training the network and in the deployed network after it has been trained. This process is summarized in Figure 8.

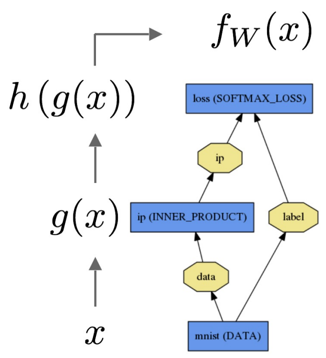


Figure 8: *A basic neural network to demonstrate forward propogation. [8]*

The data x is fed into an inner product layer $g(x)$ which is then given to a softmax layer (a common layer for classification problems) to form $h(g(x))$, which is ultimately provided to a softmax loss layer, which compares the proposed solution to the actual solution (often known as the *label* in the literature). The final result is $f_W(x)$ where W is a particular set of weights [8]. The second phase is termed backward propagation and is used only to train the network. It works by utilizing the gradient of the loss to construct an update to the weights. This process is demonstrated in Figure 9.

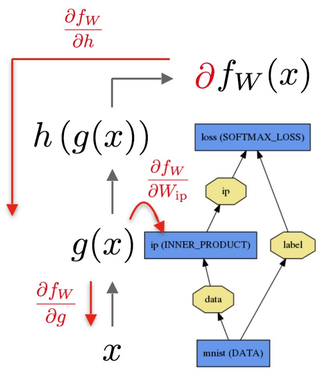

$\frac{\partial f_W}{\partial h}$
$\partial f_W(x)$
$h(g(x))$
$\frac{\partial f_W}{\partial W_{\text{ip}}}$
$g(x)$
$\frac{\partial f_W}{\partial g}$
x
loss (SOFTMAX_LOSS)
ip
ip (INNER_PRODUCT)
label
data
mnist (DATA)

Figure 9: *The same network as Figure 8 after applying backwards propogation.[8]*

The backward pass calculates the gradient of the loss with respect to the output $\frac{\partial f_W}{\partial h}$. The operation then continues to find the gradient with respect to each layer of the network using the chain rule, e.g. $\frac{\partial f_W}{\partial g}$. Furthermore, layers with input parameters, such as the inner product layer also compute the gradient with respect to their parameters, $\frac{\partial f_W}{\partial W_{ip}}$.

G. Updating Weights

To converge on a solution, the neural network must update thousands of weights in an attempt to decrease the average loss over the whole training dataset. Mathematically, we can describe the average loss $L(W)$ using the expression

$$L(W) = \frac{1}{|D|}\sum_i^{|D|} f_W(x_i) + \lambda r(W), \qquad (4)$$

where $f_W(x_i)$ is the loss on data instance x_i, $|D|$ is the set of all input data instances, and $r(W)$ is a regularization term incorporating the coefficient λ. This average loss is then used to update the weights. There are many different schemes for updating the weights, the one we selected is known as *Stochastic Gradient Descent* (SGD). It calculates the update value V_{t+1} and the updated weights W_{t+1} using from the previous iteration t the recursion relations

$$V_{t+1} = \mu V_t - \alpha_t \nabla L(W_t) \qquad (5)$$

and

$$W_{t+1} = W_t + V_{t+1}, \qquad (6)$$

where V_t is the previous weight update, W_t is the set previous weights, α_t is the learning rate, and μ is the momentum.

As the learning rate starts to plateau, it is often helpful to reduce the learning rate to converge on the solution. For our network we chose an *inverse decay* paradigm which is defined by

$$\alpha_t = \alpha_0(1+\gamma t)^{-\beta} \qquad (7)$$

where α_0 (the base learning rate), γ, and β are constant parameters provided to the network by the user.

IV. Software

I. Caffe

To expedite the development process, we have taken advantage of a program called *Caffe*, an open-source implementation of a convolutional neural network. Caffe allows the user to define a neural network using a script, and then trains the network using provided input and truth datasets.

The developers of Caffe have reduced the CNN algorithm to a sequence of matrix multiplication. This allows computations using the *Basic Linear Algebra Subprograms* (BLAS) routines, highly optimized programs that have been implemented on many different architectures. This generalization means that Caffe can be run on a graphics processing unit (GPU) to greatly accelerate training time through the GPU's impressive floating-point performance.

This program also includes an API which allows development of custom code that can access the network and use the output in other routines.

II. Custom Routines

All routines that were used to generate the datasets and validate the network were programmed in C++. This language choice allowed easy access to the Caffe API (which was also implemented in C++), and the benefits of decreased runtime for large dataset manipulation.

V. Testing CNNs Using Simulated Data

I. Introduction

To help us learn how to use CNNs, we solved two simple cases of the MOSES inversion problem: 4x4 pixel input/output image with one pixel greater than zero, all other pixels zero and a 8x8 pixel input/output image with five pixels greater than zero. Using simple cases allowed us to build and debug the network without having to deal with large, complicated real datasets.

II. Dataset Preparation

The MOSES instrument observes the sun in spatial directions x and y. This measurement is the projection of a spectral cube in x, y, and λ. In the MOSES coordinate system x is the dispersion direction. Therefore, (if we neglect the point-spread function,) we may ignore the y component of the images without loss of generality.

For this test we randomly construct a slice of the spectral cube with dimensions x and λ. This is the truth image, and the output of the neural network should strive to create this output.

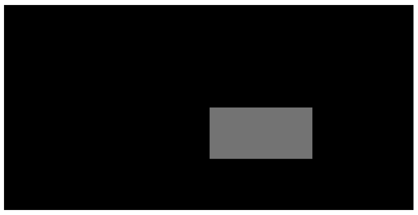


Figure 10: *An example of a 4x4 pixel truth image. For this image and all following images, the vertical axis is λ and the horizontal axis is x.*

From the truth image, we simulate the measurements captured by the instrument using the MOSES forward model. The forward model works by summing along each spectral direction (as described in Figure 2)for the $m = 0$, 1, and -1 spectral orders. Since the y dimension has been ignored, we are left with a graph, described in Figure 11

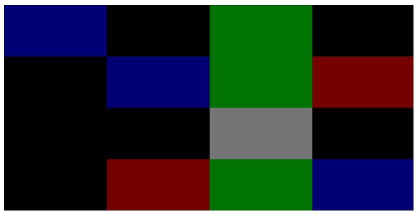
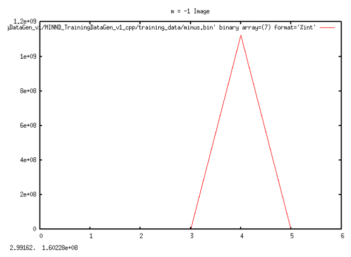
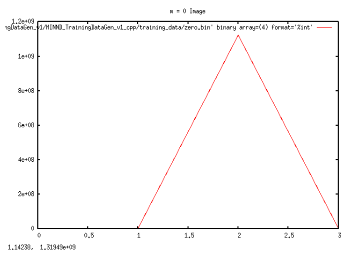


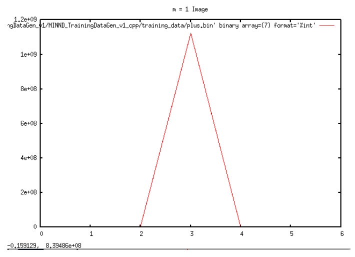
Figure 12: *An example of a 4 pixel by 4 pixel input image.*



(a) *$m = -1$ order*



(b) *$m = 0$ order*



(c) *$m = 1$ order*

Figure 11: *Simulated results of the MOSES instrument for the simple case described in Figure 10.*

We can speed up the training time of the neural network by providing it some information about how the MOSES forward model works. To do this, we transform the graphs from the three orders back into an image by backprojecting along each spectral direction. The result is essentially an image that contains the maximum possible pixel value at every location in the spectral slice. For our purposes we will store the projections from each order on separate color channels (RGB). We will call this image the "input" image, and it will be the image that is fed into the neural network. An example of this image is given in Figure 12.

III. Results

A. One Active Pixel

The learning and input parameters of the one active pixel network are summarized in Table 1. Unfortunately the image depth is not very large, it is a result of creating a simple database of bitmap images for the CNN's input and truth dataset.

The topography of our first neural network was based off of Google's LeNET image classification neural network, described in Figure 13a. It consists of one convolutional layer, one max pooling layer, and three inner product layers with ReLU layers in between.

This network performed very well, according to the loss. We can see visually in Table 1 that the output image is very close to the truth image. Since the training dataset (10,000) is larger than the number of possible images $4 \times 4 \times 256 = 4096$, it likely that the neural network simply memorized the training dataset.

B. Five Active Pixels

The input parameters for the five active pixel case are cataloged in the second column of Table 1. They are very similar to the case above, with the exception of the base learning rate α_0 which was reduced to account for the longer training time.

The five active pixel case is a much harder problem, than the one pixel case, therefore the solutions posed by the neural network aren't perfect matches to truth. However in looking at the results, displayed in Table 3, we can see that the many of the brightest pixels in the output and truth images match.

This behavior is encouraging since this problem is potentially harder than the actual MOSES inversion problem. This is because the data is arranged randomly in space, and there are no solar features to be identified by the network. It is troubling that the images produced by the network aren't always solutions to the MOSES inversion problem. Perhaps the network could be constrained better to ensure all output is a valid solution to the inversion problem.

VI. Training CNNs Using IRIS Data

VII. Conclusion

Quantity	One Active Pixel	Five Active Pixels
α_0	0.1	0.01
γ	0.0001	0.0001
β	0.75	0.75
μ	0.9	0.9
x	4 pixels	8 pixels
λ	4 pixels	8 pixels
Image depth	8 bits	8 bits
Intensity range	$[0, 1]$	$[0, 1]$
Training dataset	10^4 images	10^6 images
Training time	5 minutes	60 minutes
Loss	5.385×10^{-5}	0.127

Table 1: *Parameters describing the CNN used to train the one active pixel case*

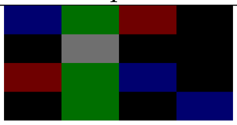
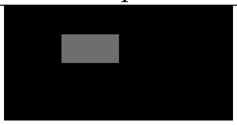
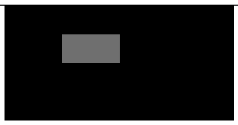
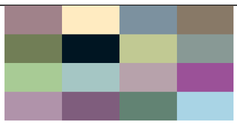
Iteration	Input	Output	Truth	Difference
1				

Table 2: *Characteristic output for the 4x4 case. MINND performs perfectly each time.*

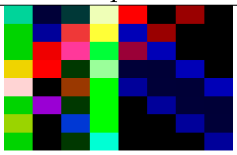
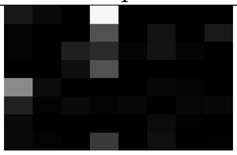
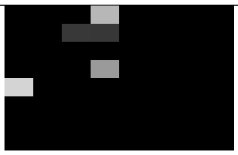
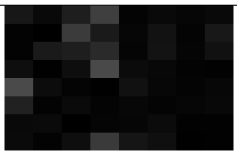
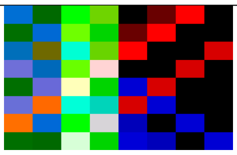
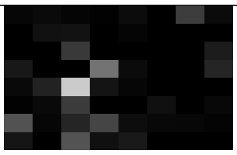
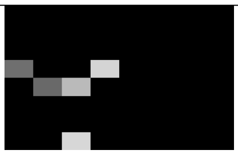
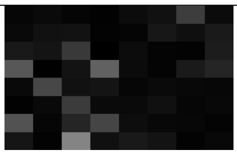
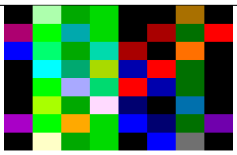
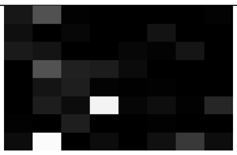
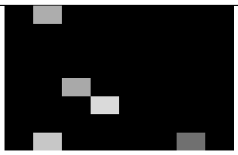
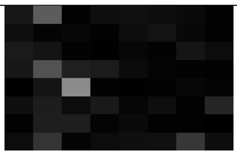
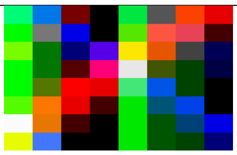
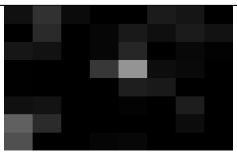
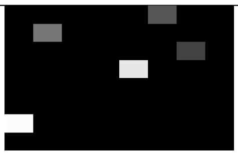
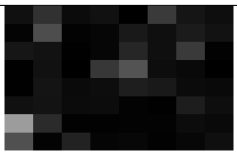
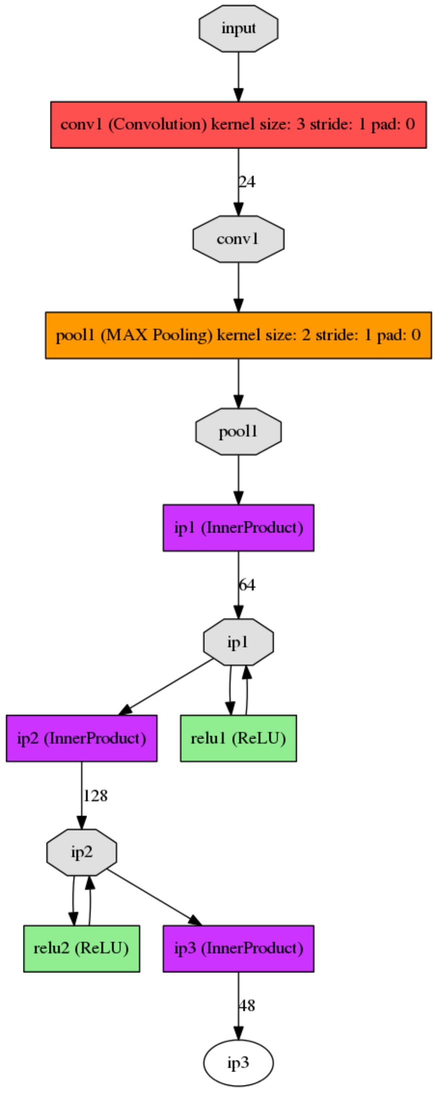
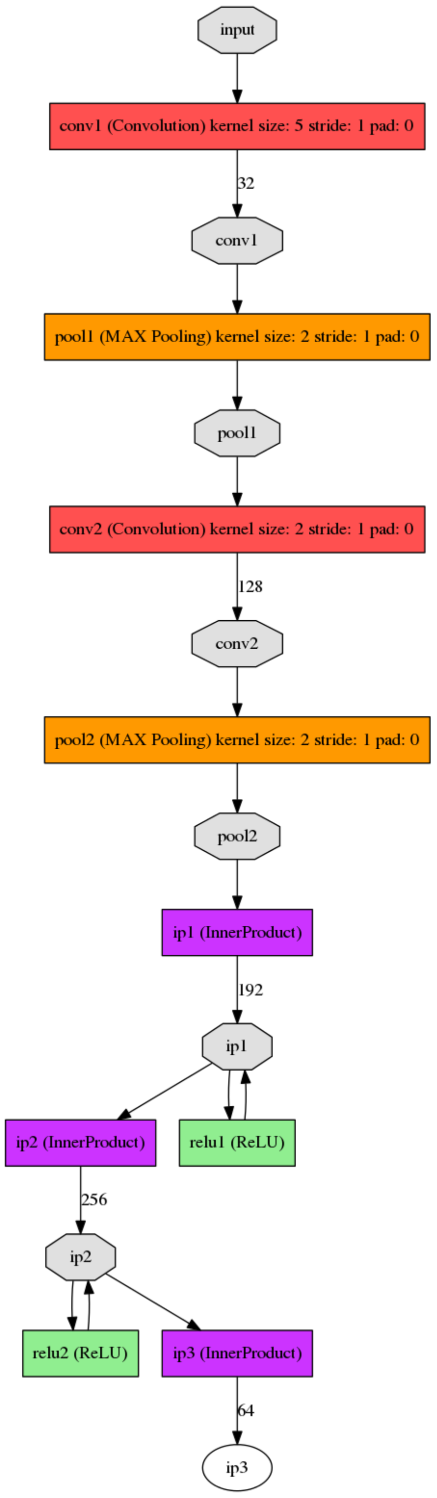
Iteration	Input	Output	Truth	Difference
1				
2				
3				
4				

Table 3: *Output of the MINND neural network for four randomly selected input images. Notice how not all output images are solutions to the inversion problem.*



(a) *Topography of the our CNN for the one active pixel case*


input
conv1 (Convolution) kernel size: 5 stride: 1 pad: 0
32
conv1
pool1 (MAX Pooling) kernel size: 2 stride: 1 pad: 0
pool1
conv2 (Convolution) kernel size: 2 stride: 1 pad: 0
128
conv2
pool2 (MAX Pooling) kernel size: 2 stride: 1 pad: 0
pool2
ip1 (InnerProduct)
192
ip1
ip2 (InnerProduct)
relu1 (ReLU)
256
ip2
relu2 (ReLU)
ip3 (InnerProduct)
64
ip3

(b) *Topography of the our CNN for the five active pixels case*

Figure 13: *Topography of the two neural networks used in the test.*

References

[1] Charles C. Kankelborg J. Lewis Fox and Roger J. Thomas. A transition region explosive event observed in He II with the moses sounding rocket. *The Astrophysical Journal*, 719:1132–1143, August 2010. `http://solar.physics.montana.edu/MOSES/papers/2010/FoxKankelborgThomas2010.pdf`.

[2] T. Rust, L. Fox, C. Kankelborg, H. Courrier, and J. Plovanic. MOSES Inversions using Multiresolution SMART. 224:414.06, June 2014.

[3] J. L. Fox, C. C. Kankelborg, and T. R. Metcalf. Data inversion for the Multi-Order Solar Extreme-Ultraviolet Spectrograph. 5157:124–132, November 2003.

[4] T. Rust and C Kankelborg. Imaging spectroscopy with moses sounding rocket data. *SPD*, 2015.

[5] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012. `http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf`.

[6] iris rast cii1336 oi1356 siiv1403 mgiik2796sji 1400 raster position 0 20160129 0549 j, January 2016. `http://www.lmsal.com/solarsoft//irisa/data/level2/2016/01/29/20160129_054924_3664251603/www/raster/iris_rast_CII1336_OI1356_SiIV1403_MgIIk2796SJI_1400_Raster_Position_0_20160129_0549_j.html`.

[7] Cuda c programming guide. website, September 2015. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/`.

[8] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22Nd ACM International Conference on Multimedia*, MM '14, pages 675–678, New York, NY, USA, 2014. ACM. `http://doi.acm.org/10.1145/2647868.2654889`.