Implementing a U-Net Architecture in MagmaDNN

T. Chow
Dept. of Mathematics
The Chinese University of Hong Kong
Hong Kong, China
nicolechow08@gmail.com

E. Karak
Dept. of Mathematics
Baruch College, City University of New York
New York City, USA
edward.karak@baruchmail.cuny.edu

S. Smith
Dept. of Computer Science
Univeristy of North Texas
Denton, USA
spencersmith4@my.unt.edu

***Abstract*—MagmaDNN is an open-source deep-learning library written in C++. It is built on top of *MAGMA*, a parallel, supercomputing-specific linear algebra package used by Oak Ridge National Laboratory (ORNL). MagmaDNN is unique in that it is tailored for parallel computing and, consequently, to supercomputing applications.**

A U-Net is a convolutional neural network (CNN) developed originally for biomedical image segmentation to detect potential tumors in humans. It can be defined in terms of down-sampling and up-sampling layers, which are abstractions of the underlying complexity. Our implementation of the U-Net in MagmaDNN is called semantic segmentation—it aims to learn the classification of every pixel in an image.

This paper will give a brief overview of MagmaDNN. Then we will look at our implementation of the U-Net, followed by the results of our network. We will end with a discussion of future directions.

***Index Terms*—U-Net, supervised learning, semantic segmentation, image segmentation, CUDA, HDF5**

I. Introduction

U-Net is a type of neural network that is commonly used in computer vision. U-Net is employed chiefly for image segmentation—that is, detecting objects in an image. To test the performance of our U-Net implementation, we input images from the CIFIAR-10, CIFAR-100 and MNIST datasets. In implementing a U-Net, we have integrated the function of segmentation into MagmaDNN. MagmaDNN is designed completely in C++ for better performance.

For the upsampling, we have implemented transposed convolution with a normal convolution. Transposed convolution is a type of convolution that "undoes" what a regular convolution has done [1]. Theoretically, only transposed convolution will be able to decode after convolution is applied. Therefore, having transposed convolution in MagmaDNN will be important.

For the loss function, we have taken reference from Facebook AI Research [2] and have integrated it into MagmaDNN. We may be the very first few to have implemented this unique kind of focal loss [2]. This is a type of cross entropy loss that will focus on learning hard misclassified examples. Cross-entropy with a four-dimensional tensor is also integrated into MamgaDNN. Additionally, MagmaDNN is now able to produce to four-dimensional tensors as the output of any neural network. Cross-entropy for four dimensions is also implemented.

We have also enhanced the I/O capabilities of MagmaDNN. It now works with HDF5 and ImageNet. While TensorFlow in C++ has an incomplete library lacking major functionality, we are among the few U-Nets written purely in C++ with comparable results to PyTorch and TensorFlow. With the I/O features implemented in MagmaDNN, it would be easier to use the data on different platforms including TensorFlow and PyTorch. We can also implement different image segmentation networks including ResUnet and RetinaNet.

Since we have limited GPU memory, we have trained our model with a small dataset that contains only 31 images. To test and compare the result of U-Net in MagmaDNN with PyTorch, we have put it into PyTorch with the same training parameters with the same dataset we used for U-Net in MagmaDNN. U-Net in MagmaDNN performs better or similarly compared to the PyTorch version of U-Net, proving the accuracy and efficiency of our U-Net.

II. Background

A. MagmaDNN

Similar to PyTorch and Keras, MagmaDNN is a machine learning package designed and developed by the Institute for Computing Laboratories (ICL) at the Univeirsity of Tennessee at Knoxville [3]. It is driven by *Magma*, a dense linear algebra package. It is an open-source software that has been developed by a number of different students, professors, and professionals. It is still in the development phase, so it is very limited in its scope; however, through constant improvements and continued documentation, it can have comparable performance to PyTorch and Tensorflow. Throughout this section on MagmaDNN, we will be talking about the structure of the framework and the setbacks found in MagmaDNN.

1) Structure of MagmaDNN: Like other machine learning libraries, MagmaDNN is built around a compute graph, which stores the operations of a network as nodes. The graph can then be evaluated and produce a result.

In Fig. 1, you can see an example of the piece of a U-Net's compute graph in MagmaDNN. Everything in MagmaDNN is

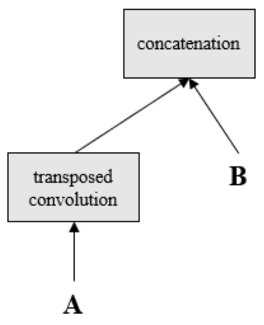

Figure 1. Compute Graph Example

built on top of tensors; however, a tensor can not be evaluated in a compute graph. So, in order to create and keep track of the compute graph in MagmaDNN, everything must be an *Operation*. Take the *A* variable in Fig. 1 for instance:

$$op :: var(\text{"}A\text{"}, tensor\_ptr); \quad (1)$$

The *tensor_ptr* corresponding to *A*, must be wrapped in an Operation before being added to the compute graph. *op::var( )* is the most basic operation there is in MagmaDNN and is used to create variables like *A* and *B* from Fig. 1. Each Operation must have a *grad* and *eval* function built into it. The grad function is what writes into the Operation's gradient tensor, and returns the gradient tensor after it is done. The eval is simply responsible for the evaluation of the Operation. It should return a tensor pointer with the same shape and memory type as defined in the Operations constructor. Now that we have covered some basics of Operations, lets build on top of that. Operations are an abstraction of the underlying math functions that come from Cuda, Cudnn, or Magma. Well *Layers* are an abstraction of the Operations. We know that a U-Net is a deep learning tool that has many *layers*. Furthermore, in MagmaDNN, to build a neural network we must keep track of the layers we define and add them to the model. In Fig. 2, there is a list of all the layers currently in MagmaDNN. There are most of the basic layers that you would find in a typical neural network. To make the framework more user-friendly, every Layer class has a function that is called if the programmer wants to create a layer. This allows for cleaner and more human-readable code. A layer function can be called like any other function in C++. For example, if we wanted to create a *conv2d* layer as we see in Fig. 2, then we would do it like this:

```
layer::conv2d(op, {3, 3}, 32, SAME);  (2)
```

This will return a layer, and what is typical in MagmaDNN development is to use `auto layer1` to store the result of the conv2d layer function. To sum up our discussion over

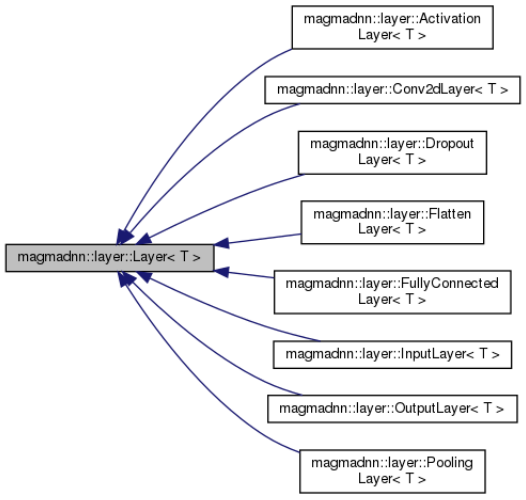
magmadnn::layer::Layer< T >
magmadnn::layer::Activation Layer< T >
magmadnn::layer::Conv2dLayer< T >
magmadnn::layer::Dropout Layer< T >
magmadnn::layer::Flatten Layer< T >
magmadnn::layer::FullyConnected Layer< T >
magmadnn::layer::InputLayer< T >
magmadnn::layer::OutputLayer< T >
magmadnn::layer::Pooling Layer< T >

Figure 2. Layers in MagmaDNN

the structure of MagmaDNN, we can see that a lot of the functionality stems from Layers and Operations. In order to successfully create a U-net model, we will have to develop Layers, Operations, and underlying math functions. We will talk more about the implementation of these new Layers and Operations in a later section.

2) Setbacks of MagmaDNN: Since MagmaDNN is still in the development phase, documentation and the overall guidelines for development are very limited. Therefore, developers who are new to MagmaDNN, will need to spend some time exploring the code to understand the structure. Also, there are a few known bugs in MagmaDNN; however, given the limited documentation, we have stumbled across them ourselves and typically just try to work around them if they are not an easy fix. The functionality in MagmaDNN is very limited. Take the loss function as an example, MagmaDNN only supports categorical cross-entropy loss and MSE. MagmaDNN can only do classification but not segmentation and hence the *Output Layer* of a neural network must be a *flattend* two-dimensional tensor or else errors will occur. We will discuss in a later section, how we have overcome the setbacks and limitations of MagmaDNN.

B. U-Net

U-net is a convolutional network developed originally for biomedical image segmentation for cancer and problematic cell detection [4]. After the paper is published, U-net is commonly used for image segmentation and object identification so that it can work with fewer training epochs but with a more accurate segmentation. The network is a modification of a fully convolutional network. The network is different from other convolutional networks since it does not require a fully connected layer to do the pixel-wise classification. Moreover, each class label is assigned to each pixel, while most of the convolutional networks do classification to an image.

U-net consists of contracting and expanding parts. For each part, they are consists of around three to four encoders or decoders in each part respectively. The number of encoders or decoders will be determined according to the size of the input image.

1) Encoder: For the encoder part of the U-Net, it is used to capture and classify objects to pass on to the decoder part. It consists of a combination of convolution, batch normalization and *ReLU* activation layers. Then it is down-sampled with a convolution of kernel size 2 with stride 2. The image size will get halved while the channels get doubled after each block of encoders.

2) Decoder: The decoder part of the U-Net, is symmetric to the encoder part. The decoder will locate the object precisely. It contains a large number of channels. Those channels can propagate features of the images to the upper layers. The decoder consists of a combination of upsampling, convolution and *ReLU* activation layer. The image size will get doubled while the channels get halved after each block of decoders. There are two choices of up-sampling methods that are commonly used with U-Net. The first one is convolution transpose, another one is *bilinear interpolation*.

3) Up-sampling Method: We have adopted convolution transpose instead of using bilinear interpolation for up-sampling in the U-Net. Convolution transpose have an advantage compared to bilinear interpolation because convolution transpose will learn when it is training. However, up-sampling using bilinear interpolation will consume less resources. However, when implementing the U-Net using PyTorch, we do see a small different in the resulted prediction after training for five epochs (Fig. 4, Fig. 5) . Bilinear interpolation out performed convolution tranpose with the *Carvana Image Masking Challenge* dataset. However, we still adopt the convolution tranpose because it overperform bilinear interpolation theoretically.

In fact, convolution transpose can be implemented with normal convolution theoretically. However, the implementation of convolution in MagmaDNN originally does not support down-sampling tensor. And then we have attempted to add padding to the tensor that is going to be convolutioned. However, this cannot be implemented to work with U-Net since adding padding to the tensor cannot pass on gradient from the previous layer to the next layer. Therefore, convolution transpose layer is required to be implemented for the U-Net to work properly.

The forward gradient for the convolution transpose is the same with the backward gradient of convolution since convolution transpose is the revserse progress of convolution. While the backward graidne for the convolution transpose is the same with the forward gradient of convolution.

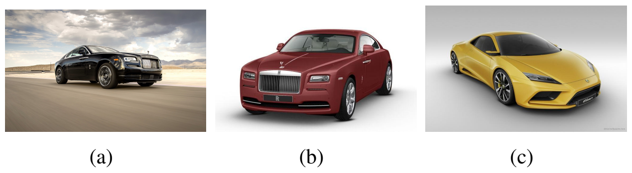

Figure 3. (a) Car 1 (b) Car 2 (c) Car 3

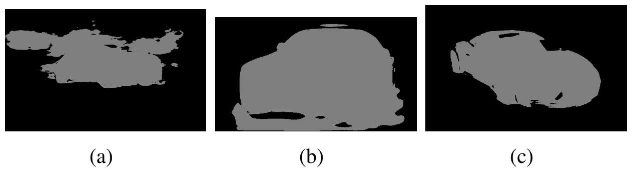

Figure 4. U-Net Prediction using Bilinear Interpolation (a) Car 1 (b) Car 2 (c) Car 3

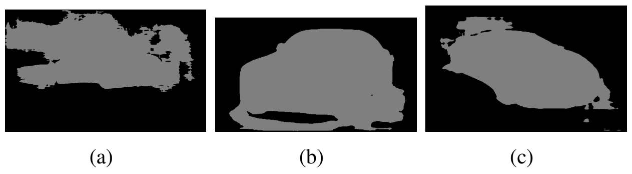

Figure 5. U-Net Prediction using Convoluntion Transpose (a) Car 1 (b) Car 2 (c) Car 3

4) Skip Connection: For some deep convolution neural networks, the gradient may vanish after a lot of layers. It affects the performance of the neural network. Therefore, a skip connection will be useful in this kind of neutral network. It provides an alternative for passing gradient in the backpropagation so that the gradient can move more freely in the model. In the U-Net, skip connection is for persevering and recovering original features in the image in the decoder part. There are a few skip connection options that are currently being used in the field in U-Net. First, it is *add*. The second is *concat*.

For the add skip connection, it is possible adopts the residual similar to the *ResNet* (Fig. 6). However, it will change the structure of the U-Net completely. It requires adding the previous layer with the convolution of the layer passed through the skip connection. It will be non-sequential for all the layers. Therefore, we have adopted the concat skip connection since MamgmaDNN cannot accept non-sequential layers currently.

For the concat skip connection, it is concatenating the output of each layer to the upsampled layers. Since in the input, we have resized every single image to a square. Therefore, the tensor will maintain to be a square tensor and we did not need to crop any tensor in the process of concatenating.

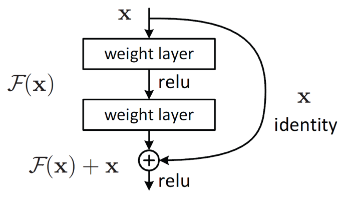

Figure 6. ResNet Skip Connection

C. CUDA

CUDA is a platform developed by NVIDIA Corp. in 2006 to facilitate GPGPU, or general-purpose computing on graphics processing units. Formerly, GPUs were used only to perform graphics operations. But because these graphics operations are expressible as matrix algebra, it is possible to translate arbitrary linear algebraic operations into graphical operations. CUDA is the framework that allows programmers to execute these computations on the GPU. By running operations on the GPU, one is able to exploit the GPU's faster floating-point arithmetic and its parallelization capabilities, leaving the CPU to perform more general operations for which it is better suited.

The CUDA language is a superset of the C++ language, which makes it naturally extensible from serial C++ code (Strictly, it is a subset of the features defined in Appendix D of [5]). Furthermore, the CUDA compiler, `nvcc`, integrates well with existing build systems by invoking the local compiler (i.e., `gcc`, `g++`) for host code (code which is to run on the CPU) and its own compiler for device code (code which is to run on the GPU). As such, MagmaDNN uses CUDA in its implementation of fundamental tensor operations on the GPU. These tensor operations are used extensively in convolutional neural networks, of which U-net is an example.

D. HDF5

HDF5 is an open-source file format that can contain a large amount of complex data. It was developed by the National Center for Supercomputing Applications and is currently supported by The HDF group. It is commonly use in the field of machine learning because of the fact that it is machine independent and the ability to store scientific data in files.

MagmaDNN was only able to input *MNIST*, *CIFAR10*, *CIFAR100* data and one-hot encoded ground-truth data. However, we have tried to use *MNIST* data as the dataset and the masking. Unfortunately, it does not have much success with the U-net since both of them are the same data and the neutral network cannot learn from the dataset. Moreover, MagmaDNN is only able do classification instead of image segmentation. Therefore, we need to use customize dataset for the training and testing for the U-Net. After we have intergrate OpenCV with MagmaDNN such that it can input data from ImageNet and Oxford-IIIT Pet Dataset.

We have put our target to intergrating the HDF5 API with MagmaDNN such that MagmaDNN can accept a wider range of dataset without needing to readjust the code in MagmaDNN to accept other input format.

E. Loss Functions

In machine learning, there must be a way to determine how far the model's prediction was from the actual value. This is where loss functions come in. The loss function defines how far the predicted value was from the ground truth and then the weights of the network are updated accordingly. Models with different purposes will have different loss functions. Currently in MagmaDNN, it has *cross-entropy* and *mean squared error* loss functions already implemented. U-net architectures use cross-entropy loss; however, the cross-entropy that is implemented in MagmaDNN has a few drawbacks. First, it only works for image classification not pixel-wise classification; secondly, it only accepts 2-dimensional tensors, which is a problem because the ground truth of most image segmentation datasets would a 3-dimensional or 4-dimensional tensor. In a later section, we will be looking at how we implemented a cross-entropy loss function specific for image segmentation problems.

III. Implementation

A. Structure of our U-Net

Our U-Net consists of one double convolution, four encoders, four decoders and one out convolution. For the encoder part, one of the differences from the original U-net paper is that each convolution is followed by a batch normalization besides the last output convolution. The reason for adopting batch normalization is the advantage of higher learning rates and to be less careful about initialization. It also make using dropout optional.

For the down-sampling method, we use the same method as the original paper, which uses maxpooling instead of strided convolution. It will be to pass on gradient easier than strided convolution. Maxpooling will prevent problems with gradient propagation.

B. Transposed Convolution

Our implementation of the transposed convolution relies on the cuDNN C++ API functions. This allows for the transposed convolution to run on the GPU and in parallel, which is ideal for super computing applications. When using these cuDNN functions, we had to think about how normal convolution is performed, since all cuDNN convolution functions are meant for normal convolution. In essence, convolution either shrinks or keeps the same dimensions based on the parameters you pass it. Furthermore, the API does not have functions specifically for transposed convolution; so, we had to figure out how to manipulate the normal convolution functions to perform the transposed convolution that we desired. Our suspicion was that we could use the backward pass of the normal convolution to perform the forward pass of the transposed, and the forward pass of the normal convolution to perform the backward pass of the transposed.

```
template <typename T>
void conv2dtranspose_device
(Tensor<T> *x, Tensor<T> *w, Tensor<T> *out,
conv2dtranspose_cudnn_settings settings)
{
    T alpha = static_cast<T>(1);
    T beta = static_cast<T>(0);
    cudnnErrchk(cudnnConvolutionBackwardData(
    MAGMADNN_SETTINGS->cudnn_handle, &alpha,
    filter_desc, w->get_ptr(),
    x->get_cudnn_tensor_descriptor(),
    x->get_ptr(),
    conv_desc, bwd_data_algo.algo,
    grad_data_workspace,
    grad_data_workspace_size, &beta,
    out->get_cudnn_tensor_descriptor(),
    out->get_ptr()));
}
```

In the code above, we show how we used cuDNN to implement the forward pass of the transposed convolution. Given that the API function `cudnnConvolutionBackwardData()` computes the gradient of the input tensor for a normal convolution, it can be said that sometimes it must up-scale the output tensor to the size of the original input tensor. Given this fact, if we pass in the appropriate parameters for the API function, then we could theoretically perform a forward pass of the transposed convolution and in turn double the height and width of the tensor. We compared our transposed convolution with the transposed convolution in *Keras* and found that ours produced the same results. We looked into the source code for Keras, since they also use cuDNN, and found that they used the cuDNN functions in the same way we did. So this confirmed that our theory was correct and our implementation of the transposed convolution works. This code is for anyone looking to implement their own transposed convolution, as there is little documentation and cuDNN does not directly support this process through any API functions. To compute the gradient of the input tensor, for transposed convolutions, during the backward pass, one simply uses the forward pass of the standard convolution, `cudnnConvolutionForward()`. Furthermore, when computing the gradient of the filter tensor during the backward pass, one can use the same function that is used for normal convolution, `cudnnConvolutionBackwardFilter()`. Determining this method of implementation was crucial for the success of our implementation, as the back half of the U-Net relies on the up-sampling method.

C. Concatenation

We implement a skip connection in our U-Net by using concatenation. Our task is to concatenate two 4D tensors. MagmaDNN stores these tensors in NCHW (batch, channels, height, width) format. Here is MagmaDNN's implementation of concatenation on the CPU, written by Sedrick Keh. It is located in `src/math/concat.cpp`.

```
while (curr_pos >= 0) {
    curr_pos = target_shape.size() - 1;
    if (target_shape[axis] < A->get_shape(axis)) {
        C->set(target_shape, A->get(target_shape));
    } else {
        target_shape_copy = target_shape;
        target_shape_copy[axis]
        -= A->get_shape(axis);
        C->set(target_shape,
        B->get(target_shape_copy));
    }
    target_shape[curr_pos]++;
    while (target_shape[curr_pos]
    == C->get_shape(curr_pos)) {
        target_shape[curr_pos] = 0;
        curr_pos--;
        if (curr_pos < 0) break;
        target_shape[curr_pos]++;
    }
}
```

The GPU version, written in CUDA C++, is similar. MagmaDNN determines whether to call the CPU or GPU version by examining `output_tensor->get_memory_type()`. `output_tensor` is a protected variable defined in the `Operation` class. A MagmaDNN user specifies whether to use the CPU or the GPU by passing the value `magmadnn::memory_t::HOST` or `magmadnn::memory_t::DEVICE`, respectively, to the `magmadnn::Tensor` constructor.

D. Loss Function for Pixel-wise Classification

The loss function is the most important part of any deep learning network. Without it, the network will not be able to learn from its training and will ultimately not function as intended. The loss function used for image segmentation is typically cross-entropy. Whether the loss function is categorical or binary depends on the goal of the model. For the initial implementation of the U-Net, we decided to use binary classification, where 1 represents the foreground of the image and 0 represents the background. MagmaDNN already has a cross-entropy loss implemented; however, it is only able to handle flattened inputs and ground-truths. Given that the goal of a U-Net is image segmentation, we thought it would be appropriate to implement a cross entropy that can handle 4D Tensors in addition to 2D. Furthermore, since we were already making a new cross-entropy function, we decided to modify it in a way that tailors it to image segmentation. We followed the approach used in [2], where they added a variable to calculate the distance from the foreground. We called this approach *distance aware cross-entropy*. This allowed the network to be more lenient on background pixels that were classified incorrectly, but were relatively close to the foreground. However, background pixels that were far away from the foreground were punished severely for incorrect classifications. This allowed for the network to focus training onto the foreground and it would in turn converge in fewer epochs. Since our time in this program is limited, we stopped this implementation of the loss function, and we had to settle for a normal cross-entropy approach. Everything for the distance-aware cross entropy to work has been implemented; however, we ran out of time and could not reimplement it in CUDA. Our results reflect the fact that standard cross-entropy is not optimized for image segmentation.

E. HDF5 I/O

For efficient access to trained models on disk, we use HDF5 (Hierarchical Data Format 5). An HDF5 file stores tensors in a hierarchical structure consisting of *groups* and *datasets*. A group consists of zero or more groups (groups nest), or zero or more datasets (datasets do not nest). A dataset is an arbitrary-dimensional tensor and its accompanying metadata. The dataset is the deepest level of structure in an HDF5 file.

A group or dataset is uniquely identified by its *path*. An HDF5 path is analogous to a filesystem path and can be relative (to some particular group) or absolute. The root node is denoted `/`; the path separator is the same character.

The HDF Group, which maintains the HDF5 standard, provides a low-level C API for accessing HDF5 files. We wrote a set of C++ templates and classes that adds a thin layer of abstraction over this API. These templates are interchangeable with the C API structures and functions, granting MagmaDNN users both abstraction and access to a large and stable API. These templates allow the user to write and read arbitrary-dimensional host `magmadnn::Tensors`, `std::vectors` and plain C arrays.

In the future, we intend to add support for writing portions of tensors and reading portions of datasets. This functionality is already supported by the underlying API. This will improve performance by reducing the number of I/O operations needed. We additionally plan to use the HDF5 C API's file access property lists in order to have control over file locking and parallel file access. However, this functionality is not currently needed in MagmaDNN.

IV. Results

The data set we are using is the Oxford-III Pet data set. We are using an 80/20 training-testing split with 31 randomly chosen images for training, and 7 images used for testing our model. We chose image dimensions of 256 by 256 pixels and the input is in RGB format. Furthermore, we used a batch size of 1 because of the limited GPU memory we had access to. In the future, we would like to experiment with a larger batch size and training set. All results were gathered on an Nvidia-RTX 3060 with 12GB of memory. We decided to test these exact specification on two different optimizers: one is Stochastic Gradient Descent with Momentum, and the other is the Adam optimizer. It is important to note that we trained the PyTorch U-Net and our U-Net on the same dataset for consistency.

	IoU	**Dice**	**Pixel**
30 epochs	0.2703989	0.23404101	0.189038
60 epochs	0.7130330	0.26042986	0.4059788
90 epochs	0.7820648	0.25603748	0.4331681

Table I
PyTorch U-Net using Adam

	IoU	**Dice**	**Pixel**
30 epochs	0.4309651	0.50146092	0.2908932
60 epochs	0.7270899	0.74655103	0.4190377
90 epochs	0.6567489	0.70364418	0.3932231

Table II
Our U-Net using Adam

	IoU	**Dice**	**Pixel**
30 epochs	0.394911641	0.45857022	0.272548213
60 epochs	0.697989235	0.732576941	0.408179616
90 epochs	0.677266841	0.717351087	0.400935175

Table III
Our U-Net using SGD w/ Momentum

We compared our model to a PyTorch model so we could get an accurate measurement of how our implementation compared to a reputable deep learning library. We have used IoU (intersection of unions), dice loss and pixel accuracy to measure the accuracy of our segmentation. IoU is a measurement of degree of overlap between the ground-truth and the predicted mask (Equation 3). A number larger than 0.5 is consider a good prediction. We can see from Table IV and Table IV that for 30 epochs, the prediction is not very accurate. However, after 60 epochs, the IoU score is better. It has comparable results compared to PyTorch. And after 90 epochs, it becomes overfit for our model.

$$IoU = \frac{TP}{TP + FP + FN} \tag{3}$$

TP: true positive, FP: false positive, FN: false negative

Dice coefficient computes the similarity between the ground-truth and the predicted mask (Equation 4). We can see from Table IV that our model performs better than PyTorch does, meaning that our predicted mask is more similar to the ground-truth than the output of PyTorch.

$$DSC = \frac{2TP}{2TP + FP + FN} \tag{4}$$

TP: number of true positive pixels, FP: number of false positive pixels, FN: number of false negative pixels

For pixel accuracy, the algorithm visits each pixel and measures how many other pixels have its ground-truth value. It is a measurement of predicting the value of the pixel. We could see that the pixel accuracy is not that high for PyTorch and our U-NET. However, they have a comparable result.

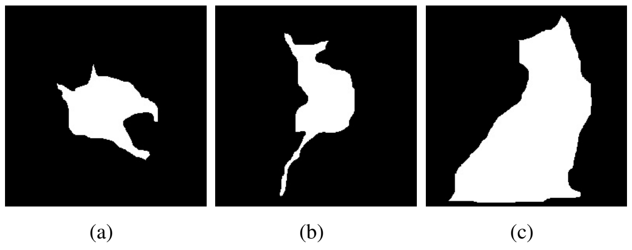

Figure 7. Groundtruth
(a) Cat 1 (b) Cat 2 (c) Cat 3

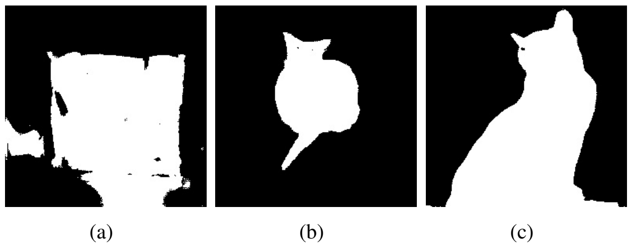

Figure 8. Our U-Net Prediction using Adam after 60 epochs
(a) Cat 1 (b) Cat 2 (c) Cat 3

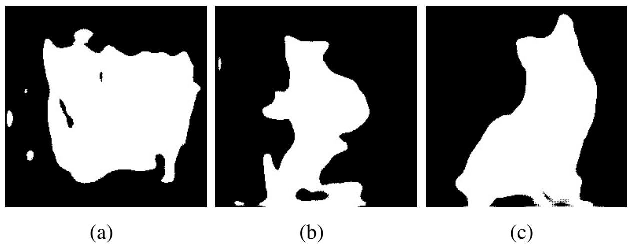

Figure 9. Our U-Net Prediction using SGD w/ Momentum after 60 epochs
(a) Cat 1 (b) Cat 2 (c) Cat 3

The model trained with the Adam optimizer proved to be more accurate, converged in fewer epochs and needed less

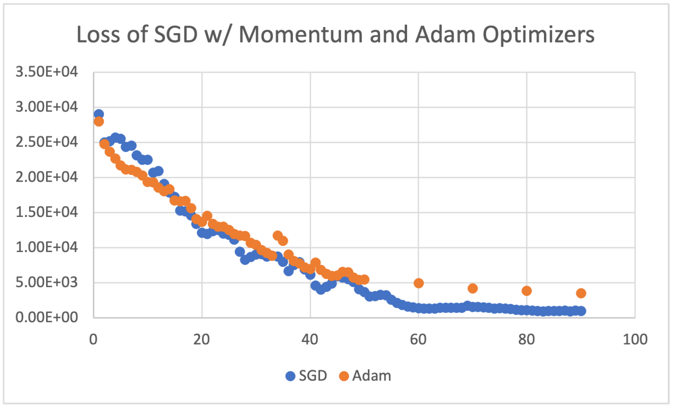

Figure 10. Loss of SGD with momentum vs with Adam optimizers

memory to train. You can see in Fig. 10 a comparison of the two models' losses. Adam has a smooth curve which is expected, and SGD seems to find its minima relatively quickly, but there could be room for tuning. When looking at the results of the model with Adam in Fig. 8 versus the results of the model with SGD in Fig. 9, we saw that they both had an easier time with Cat 3; however, the model with Adam did have a slightly higher accuracy score at 96%. We can also clearly see that with Cat 2, the model with Adam had a far easier time separating the foreground from the background, and the accuracy score reflects that. In Fig. 8 on Cat 2, it got an accuracy of 93%, where in Fig. 9 the accuracy rating was only 83%. In regards to Cat 1, we can see that both models had a hard time segmenting the foreground from the background. There were a few testing samples that turned out similarly that one, and we could pin that result on a number of different factors. Perhaps the small training set did not expose the model to enough variety, and therefore it encountered difficulty segmenting some cats as opposed to others. We could also deduce it to the loss function and the fact that we ended up using standard cross-entropy. Perhaps the model was not able to focus the training onto the foreground and therefore we ended up with sub-optimal results on a few images. We would like to test both of these theories and gain more results.

V. Future Directions

We believe that the distance-aware cross-entropy will help our network gain better results through focusing the training onto the foreground of the image. In the future, we would like to adjust our implementation of the distance-aware cross-entropy to make it more efficient and then proceed with writing it in CUDA code. This would allow our network to run completely on the GPU and in parallel, which is ideal for super computing applications.

Additionally, from the discussion above we see that PyTorch has good results from the up-sampling method, bi-linear interpolation. It would be interesting to implement that method in MagmaDNN and to compare the results of the two up-sampling methods.

With respect to the tuning of our model, we believe that there is a lot of improvement left on the table. We would like to test our model with a larger data set and batch size. Even though the accuracy of our model was good, compared to PyTorch, we believe that we could see better results if our model was exposed to a greater variety of samples. Also with a bigger batch size, the model would be able to converge in a smaller number of epochs.

Due to limited time, we have not implemented the GPU version of distance-aware cross entropy and using CUDA to calculate the gradient of the concatenation operation. Therefore, we would like to implement those in the future to do segmentation with speed comparable to PyTorch and TensorFlow.

VI. Conclusion

MagmaDNN is an up-and-coming deep learning framework that has gained traction over the years. Given that it is an in-house development by the ICL at UTK, we believe that it will gain popularity as development continues. MAGMA is also currently in the process of being imported as a library into Python (the PyMagma project), and is looking to extend its support to Intel GPUs with the help of *oneAPI*. Our U-Net implementation will further help MagmaDNN as a reputable deep learning package.

References

[1] V. Dumoulin and F. Visin, “A guide to convolution arithmetic for deep learning,” *ArXiv*, vol. abs/1603.07285, 2016.

[2] T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollr, “Focal loss for dense object detection,” in *2017 IEEE International Conference on Computer Vision (ICCV)*, 2017, pp. 2999–3007.

[3] D. Nichols, N.-S. Tomov, F. Betancourt, S. Tomov, K. Wong, and J. Dongarra, “Magmadnn: Towards high-performance data analytics and machine learning for data-driven scientific computing,” in *High Performance Computing*, M. Weiland, G. Juckeland, S. Alam, and H. Jagode, Eds. Cham: Springer International Publishing, 2019, pp. 490–503.

[4] O. Ronneberger, P. Fischer, and T. Brox, “U-Net: Convolutional Networks for Biomedical Image Segmentation,” *arXiv e-prints*, p. arXiv:1505.04597, May 2015.

[5] S. Ruder, “An overview of gradient descent optimization algorithms,” *arXiv e-prints*, p. arXiv:1609.04747, Sep. 2016.