DEEP LEARNING WITH GO

A Thesis

Submitted to the Faculty

of

Purdue University

by

Derek L. Stinson

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science in Electric and Computer Engineering

May 2020

Purdue University

Indianapolis, Indiana

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF THESIS APPROVAL

Dr. Zina Ben Miled

Department of Electrical and Computer Engineering

Dr. Brian King

Department of Electrical and Computer Engineering

Dr. Maher Rizkalla

Department of Electrical and Computer Engineering

Approved by:

Dr. Brian King

Head of Graduate Program

ACKNOWLEDGMENTS

I would like to thank my thesis advisor, Dr. Zina Ben Miled, for her support and guidance throughout this thesis. I would like to thank Dr. Brian King for advising me throughout the M.S.E.C.E program at IUPUI. I would like to thank Dr. Maher Rizkalla for being a part of my thesis committee. I would like Sherrie Tucker for making sure I am aware of the things that need to complete before the deadlines. I would like to thank Dr. Steven Rovnyak for giving me the opportunity to teach the ECE 20700 lab. I would like to thank my Mother and Father. Without them, I would not have been able to pursue graduate school. I would like to thank my son for contributing more at home and allowing me to be able to focus on completing my thesis work. I would also like to thank my daughter with her ability to make me mindful of things around me.

TABLE OF CONTENTS

Page

LIST OF TABLES vii

LIST OF FIGURES viii

SYMBOLS ix

ABBREVIATIONS x

ABSTRACT xi

1 INTRODUCTION 1

2 RELATED WORK 3

2.1 Go and Cuda 3

2.2 Deep Learning Frameworks 3

3 METHODOLOGY 5

3.1 Neural Networks 5

3.1.1 Forward Propagation 6

3.1.2 Backward Propagation 7

3.2 Convolution Layer 8

3.2.1 Forward Propagation 11

3.2.2 Back Propagation 15

3.3 Activation Layer 18

3.4 Weight Optimization 18

3.4.1 Gradient Descent 18

3.4.2 Momentum Update 19

3.4.3 Adagrad 19

3.4.4 Adam 19

3.5 Implementation 20

4 RESULTS 22

Page

4.1 Code Samples . . . 25
5 CONCLUSION . . . 27
5.1 Challenges . . . 27
5.2 Future Work . . . 28
5.3 GoCuNets . . . 28
REFERENCES . . . 29
APPENDICES
A ConvNetGo . . . 32
A.1 Tensor . . . 32
A.2 Convolution . . . 32
A.2.1 Convolution Struct . . . 32
A.2.2 Convolution Forward Algorithm . . . 33
A.2.3 Convolution Forward Window . . . 34
A.3 Leaky . . . 35
A.3.1 Leaky Struct . . . 35
A.3.2 Leaky Forward . . . 35
A.3.3 Fully Connected . . . 36
B GoCuNets . . . 37
B.1 Tensor . . . 37
B.2 Builder . . . 38
B.2.1 Builder Data Structure . . . 38
B.2.2 Builder Method Convolution Layer . . . 39
B.2.3 Builder Method Convolution Weights . . . 40
B.2.4 Builder Method Activation Layer . . . 41
B.3 Module Interface . . . 42
B.3.1 VanillaModule . . . 43
B.3.2 ModuleNetwork . . . 44
C GoCudnn . . . 45

Page

C.1 Linking Cuda to Go 45
C.2 NewModuleData 45
C.3 Cuda Concat Kernel 46
C.4 MakeKernel 47
C.5 (k *Kernel) Launch() 48
C.6 Concat Kernel Launch 49
C.7 MallocManagedGlobalEx 50
C.8 Copy Memory 51
C.9 Interface io.Reader 52
C.10 Interface io.Writer 53

LIST OF TABLES

Table — Page

4.1 Samples taken from the MNIST database with the corresponding one-hot encoding of their labels. 23

4.2 Execution time (in seconds) per epoch for each batch size for the ConvNetGo (CPU implementation) and GoCuNets (GPU implementation). . . 24

4.3 Number of epochs, training time for each batch size when executing the GoCuNets (GPU) model. Convergence was decided when the average loss for the testing data was less than 0.01. 26

LIST OF FIGURES

Figure — Page

3.1 Fully connected network with one input layer, one hidden layer and one output layer. 6

3.2 Schematic representation of 4D tensors. Each column is a tensor. 10

3.3 A visualization of padding stride and dilation with an input of (4,4) and weights of (3,3). 11

4.1 The layers of the CNN used to test the proposed framework. 22

4.2 Execution time per epoch for CoCuNets (GPU implementation) and ConvNetGo (CPU implementation) with increasing batch size. 25

SYMBOLS

W	Weights or filter for a layer
Y	Output tensor on a forward pass.
X	Input tensor on a forward pass.
ΔW	Gradients for weighs or filter.
ΔY	Input tensor for gradients on a backward pass.
ΔX	Output tensor for gradients on a backward pass.
$.N$	Batch dimension or number of neurons.
$.C$	Tensor channel dimension
$.H$	Tensor height dimension
$.W$	Tensor width dimension

ABBREVIATIONS

CNN	Convolutional neural network
FCNN	Fully connected neural network
ANN	Artificial neural network
NHWC	Tensor format (N-Total Batch, H-Input Height, W-Input Width, C-Total Channels)
NCHW	Tensor format (N-Total Batch,C-Total Channels, H-Input Height, W-Input Width)
dims	Dimensions
dim	Dimension

ABSTRACT

Stinson, Derek L. M.S.E.C.E., Purdue University, May 2020. Deep Learning with Go. Major Professor: Zina Ben Miled.

Current research in deep learning is primarily focused on using Python as a support language. Go, an emerging language, that has many benefits including native support for concurrency has seen a rise in adoption over the past few years. However, this language is not widely used to develop learning models due to the lack of supporting libraries and frameworks for model development. In this thesis, the use of Go for the development of neural network models in general and convolution neural networks is explored. The proposed study is based on a Go-CUDA implementation of neural network models called GoCuNets. This implementation is then compared to a Go-CPU deep learning implementation that takes advantage of Go's built in concurrency called ConvNetGo. A comparison of these two implementations shows a significant performance gain when using GoCuNets compared to ConvNetGo.

1. INTRODUCTION

In late 2007 at Google, Robert Griesemer, Rob Pike and Ken Thompson began working on a new computer language. They were frustrated with the excessive complexity and lack of safe and efficient multiprocessor features in the languages they used to develop server software. When looking at all the available languages, they concluded that in picking a language you had to choose at most two out of three options. These are efficient compilation, efficient execution, or ease of programming [1].

Their solution was the creation of Go. Go attempts to address these issues by being a statically typed, compiled language. Go has built in concurrency, a garbage collector, rigid dependency specification (no codependent packages) [1], and tools used to compile, link, test, format, import and document Go code [2].

There are several frameworks used in deep learning. These include TensorFlow [3], PyTorch [4], Keras [5], MXNet [6] and Chainer [7]. TensorFlow contains APIs for Python, c, Java, and Go. MXNet also has multiple APIs that are in Python, C++, Clojure, Java, Julia, Perl, R, and Scala. PyTorch API uses Python, but it has bindings in C++. Keras and Chainer only use Python. Out of the above mentioned deep learning frameworks, only TensorFlow has a Go API. However, this API is mostly used for running models in a Go application that were developed with Python.

There is growing support for the use of Go in data science and computer vision with packages like Gonum [8] and GoCV [9]. However, there is still a demand for deep learning tools in Go. In response to this demand, ConvNetGo [10], GoCudnn [11] [12], HipGo [13] [14], MIOpenGo [15] [16], and GoCuNets [17] were developed.

GPU computation is used heavily in deep learning in order to accelerate execution time. There are 3rd party open source packages for Nvidia's CUDA such as cuda5 [18], gorgonia/cu [19], and cuda [20]. These packages have their strengths and weaknesses. GoCudnn was developed to overcome those weaknesses. GoCudnn started out as

bindings for cuDNN [11]. It includes bindings for some of the other libraries that are found in the CUDA [21] API. These libraries include nvJPEG, CUDA Runtime, CUDA Driver, NPP, and NVRTC. There are also kernels that were developed outside of cuDNN that are helpful for computer vision and deep learning.

This thesis proposes a Go-Cuda implementation to support the development of neural network models including convolutional neural networks called GoCuNets. To compare the performance of GoCuNets, a CPU implementation of neural network models called ConvNetGo was also developed. Chapter 2 includes a review of previous related work and in particular previous Go-Cuda implementations. Chapter 3 discusses the methodology used in the design of the convolutional neural networks under both GoCuNets and ConvNetGo. The performance of these implementations are compared in Chapter 4. Chapter 5 provides a summary of the benefits and limitations of the proposed GoCuNets frameworks and offers direction of future work.

2. RELATED WORK

Go is a new language. Go 1.0 was released in March 2012 [22]. The focus of this thesis is to integrate GPU computation with the Go language for the purpose of developing deep learning models. This chapter includes a review of some of the packages that were developed for GPU computation with Go, the applications that use them, and other deep learning frameworks.

2.1 Go and Cuda

Cuda5 [18] is the first binding package for CUDA. It is a highly flexible package that has one huge limitation. It handles errors by panicking. Gorgonia/cu [19] takes Cuda5 and gets rid of this issue by having the function return an error interface and by adding cuBLAS, NVRTC, and some of cuDNN.

Another binding that is available on Github is unixpickle/cuda [20]. It is a light weight package with a few functions that interface with the cuda driver. It contains sub-packages for cuBLAS and cuRAND. The best feature of this package is the use of Go's garbage collector to handle memory management in the GPU.

2.2 Deep Learning Frameworks

TensorFlow [3] is probably the most known deep learning framework. TensorFlow was originally developed by the Google Brain team. It is now an open source platform. TensorFlow has stable Python and C++ APIs. There are APIs in other languages, including Go, but they are not supported with the same level of maturity.

Caffe [23] is another widely known deep learning framework. It was developed at Berkeley by Yangqing Jia. It is an open source project. Caffe's official API is in

C++. In 2017, Caffe2 was announced by Facebook [24]. In 2018 it was integrated with another Facebook project called PyTorch [4]. Pytorch has APIs in Python and C++.

ConvNetjs [25] is an open source deep learning framework that uses javascript and is ran in an internet browser. It was developed by Andrej Karpathy. It has visual demos of a few types of neural networks. The demo includes images of the tensors that are used in different layers of the network.

Gorgonia [26] is an open source deep learning API that uses Go. It uses the Gorgonia/cu Go bindings for CUDA. The goal of Gorgonia is to provide a machine learning/graph computation based library. Using Gorgonia should feel familiar to other Python learning APIs like TensorFlow or Keras. However, as a deep learning API on Go, Gorgonia might not be the right fit for developers that have never used Python.

3. METHODOLOGY

Neural networks consist of layers of neurons. These networks include an input layer, one or more hidden layers, and an output layer. The neurons accept a set of inputs which are multiplied by weights and processed through an activation function. The output of the neuron in one layer propagates to the input of a neuron in a subsequent layer until reaching the output layer.

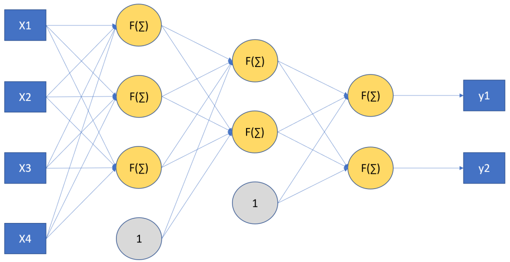
This architecture is at the foundation of most current networks. The key to developing a successful network is:

- Determining the value of the weights of the links between neurons, a process which is referred to as training, and
- Defining a suitable architecture for the network including the number of layers, the number of neurons in each layer, and the activation used at the output of each neuron.

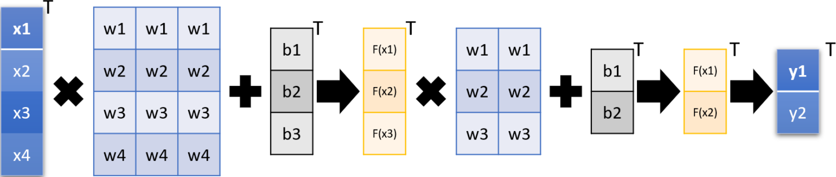
In this chapter, the implementation or the proposed neural network is described starting from a simple neural network to the target convolutional neural network.

3.1 Neural Networks

Figure 3.1(a) shows a fully connected neural network [27] with one single hidden layer. In this figure, the input and output of the network are represented by rectangular boxes. The neurons are represented by circles and the weights are depicted by the links between the neurons. Bias nodes are a constant input of one. The neurons will have a bias weight that is summed with the other links. These are indicated by shaded circles in Figure 3.1(a).



(a) Neural network architecture.



(b) Matrix representation of the network.

Fig. 3.1. Fully connected network with one input layer, one hidden layer and one output layer.

A fully connected layer can be viewed as a matrix multiplication between a 1xN input matrix and an NxM weights matrix. The result is a 1xM matrix. The bias 1xM matrix is then added to the previous result. This process is shown in Figure 3.1(b).

The main two operations associated with the network are forward propagation and backward propagation. These two operations are used during training to update the weights. Once the weights of the network are determined, the forward propagation is used on a new input to generate the estimated output.

3.1.1 Forward Propagation

In the forward propagation process, the output of each layer is generated by using Equation (3.1)

$$Y^{(t)}_{(1,M)} = X^{(t)}_{(1,N)} * W^{(t)}_{(N,M)} + B^{(t)}_{(1,M)} \tag{3.1}$$

Where, $X_{(1,N)}$ is the input vector with N elements, $W_{(N,M)}$ is the weight matrix, and $B_{(1,M)}$ is the bias vector for the layer. The bias values in Equation (3.1) are added to the weighted term $X_{(1xN)} * W_{(NxM)}$ resulting in an output vector with M elements, $Y_{(1,M)}$.

Equation (3.1) is implemented in Algorithm 1.

Algorithm 1 Forward Propagation in a Fully Connected Layer

Input:	X, W, B	Matrix
Output:	Y	Matrix

```
1: function ForwadPropagationFullyConnected(X, Y, W, B)
2:     for n = 0 to len(Y) do
3:         for m = 0 to len(X) do
4:             sum ← sum + W[n][m] * X[m]
5:         Y[n] ← sum + B[n]
```

3.1.2 Backward Propagation

Backward Propagation [28] is used to train the network. It consists of three functions:

- The first function takes the output error it receives from the next layer and uses it to calculate the errors associated with the input it received from the previous layer (Equation 3.2a),
- The second function accumulates the errors for the weights of each neuron (Equation 3.2b), and
- The third function evaluates the error for the bias vector (Equation 3.2c).

$$\Delta X^{(t)}_{(1xN)} = \Delta Y^{(t)}_{(1xM)} * (W^{(t)})^T_{(MxN)} \tag{3.2a}$$

$$\Delta W^{(t)}_{(NxM)} = (X^{(t)})^T_{(Nx1)} * \Delta Y^{(t)}_{(1xM)} \tag{3.2b}$$

$$\Delta B^{(t)}_{(1xN)} = \Delta Y^{(t)}_{(1xN)} \tag{3.2c}$$

Where, $\Delta X^{(t)}_{(1xN)}$ is the vector that holds the error due to the output $\Delta Y^{(t)}_{(1xM)}$ propagated back from the previous layer. $\Delta W^{(t)}_{(NxM)}$ represents the error matrix for the weights of the current layer. It will be used to adjust the current layer's weights during training. $\Delta B^{(t)}_{(1xM)}$ is the error vector for the bias vector. It is used to adjust the bias vector during training. $X^{(t)}_{(1xM)}$ is the input and $W^{(t)}_{(NxM)}$ is the weight matrix of the current layer from the previous iteration of the algorithm. Equation (3.2) is implemented in Algorithm 2.

Algorithm 2 Fully Connected Back Propagation

Input: dY, X, W Matrix

Output: dX, dW, dB Matrix

```
1: function FCBackProp(dY, dW, dB, X, W, dX)
2:     SetToZero(dX)
3:     for n = 0 to len(dY) do
4:         for m = 0 to len(dX) do
5:             dX[m] ← dX[m] + W[n][m] * dY[n]
6:             dW[m][n] ← X[m] * dY[n]
7:         dB[n] ← dY[n]
```

3.2 Convolution Layer

The convolution layer [29] is very similar to the fully connected layer. However, instead of each neuron having a weight for each input, and only one output. Each neuron will have a volume of weights that step through the input in multiple dimensions with each step returning an output value.

A typical architecture for a convolution layer receives a 4D input volume called a tensor with a format of NHWC or NCHW [30]. NHWC represents a tensor's ordering by batch, height, width, and channel, respectively. For example, a NHWC tensor of bytes with the dimensions [20,320,240,4] is used to process a batch of 20 images with a height of 320 and width of 240. The 32 bit color information is represented as 4 byte color vector. Under NCHW, the dimensions would look like [20,4,320,240] with the pixels separated out into 4 feature maps, with a height of 320 and a width of 240.

Performance can vary depending on the tensor format. For example, Intel recommends NCHW for their newer processors [31], and Nvidia recommends NHWC in order to take advantage of the tensor core in their new architectures [32]. In this thesis NCHW is adopted, because it is easier to visualize.

The input, weights, and output should be in the same format. A convolution layer will contain a 4D tensor of weights. Under the adopted NCHW tensor format, N represents a batch of "neurons" as opposed to a batch of inputs. The values stored in each CHW are the feature weights of N. These feature weights are also called kernels. The number of kernels is C, with a height of H and a width of W. For each neuron in $W.N$, there will be the same number of kernels ($W.C$) as there are feature maps from the input ($X.C$). The output tensor batch size is the same as the input's batch size (i.e., N). The result of a neuron's convolution of the input will be an output feature map with size HW. The number of neurons in the weights will determine the number of output feature maps. The size of the output channel dimension ($Y.C$) is determined by the number of neurons in the weights ($W.N$). The sizes of HW are determined by the properties of the convolution between the input and weights. An illustration of the 4D tensors in a convolutional neural network is shown in Figure 3.2.

There are a few convolutional processing steps that are used in accessing the data being held by the input tensor. These are performed in the H and W dimensions and consist of padding, stride, and dilation. The properties of these data processing

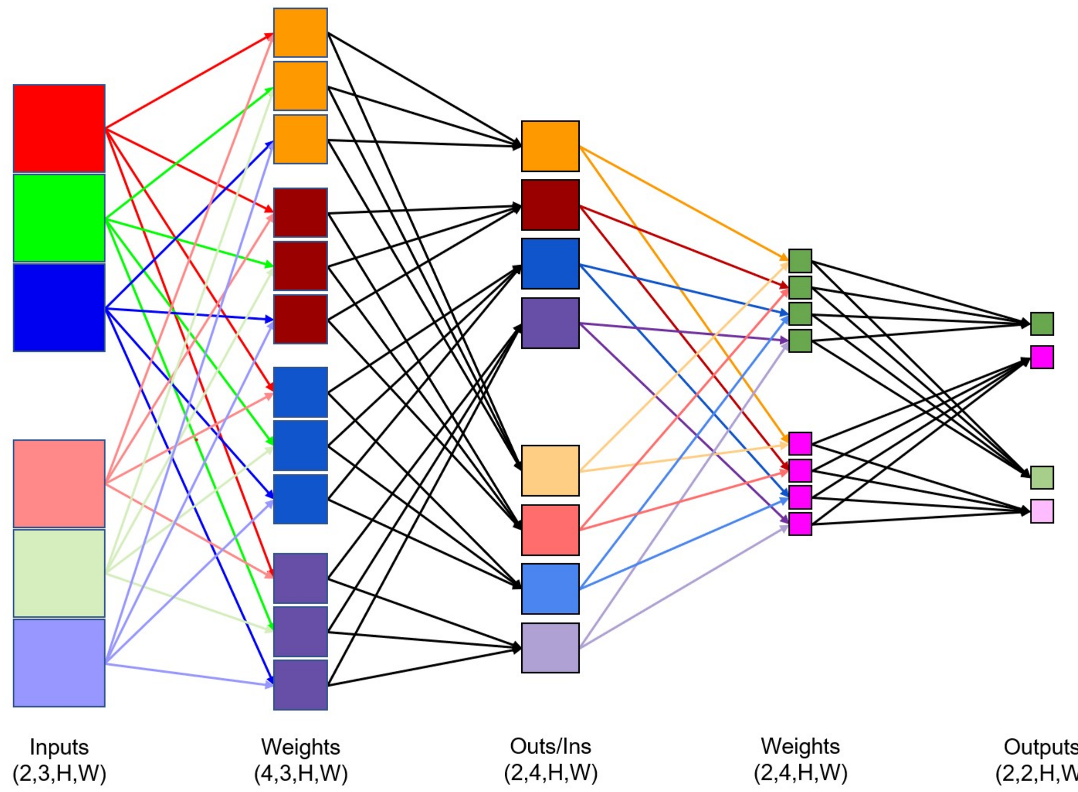


Fig. 3.2. Schematic representation of 4D tensors. Each column is a tensor.

steps will affect the size of the output tensor. A visualization of the convolutional processing steps can be seen in Figure (3.3).

Padding (p) adds zeros around the H and W dimensions of the input tensor x. The size of the padding should be less than the size of the weights. If the padding is greater than or equal to the weights (w) then the output edges will be zeros.

Stride (s) corresponds to the step of the window over the input tensor in the H and W dimensions. Larger strides will reduce the size of the output.

Dilation (d) spreads the weights apart in the H and W dimensions giving them extended coverage without additional parameters.

As implied by the above three transformations, the shape of the output tensor is dependent on the parameters used to process the data. Equation (3.3) shows the size

of the output tensor based on the padding, stride, dilation. Not all parameter values can be used since some choices of p, w, s, d may lead to a non-integer size y for the output tensor. Therefore, best practice starts by fixing the size y of the output tensor and then deriving the size x of the input tensor using Equation (3.4).

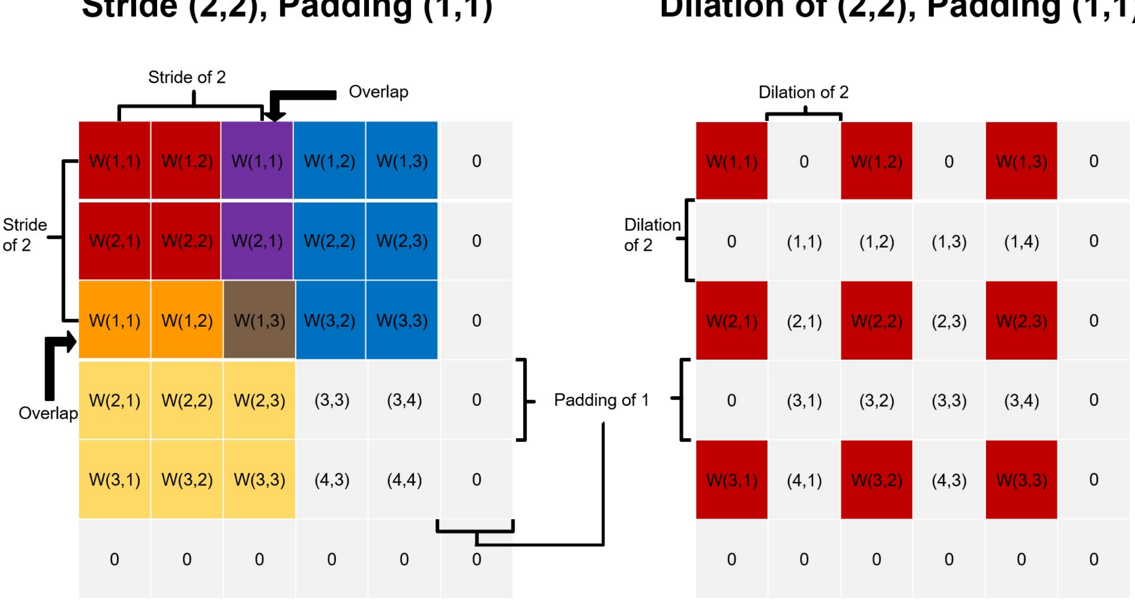


Fig. 3.3. A visualization of padding stride and dilation with an input of (4,4) and weights of (3,3).

$$y = \frac{x + 2p - ((w-1)*d+1)}{s} + 1 \tag{3.3}$$

$$x = (y-1)*s - 2p + ((w-1)*d+1) \tag{3.4}$$

3.2.1 Forward Propagation

The values of the output tensor are calculated using Equation (3.5).

$$Y^{(t)}_{n,k,yh,yw} = B_k + \sum_{c=0}^{W.C-1} \sum_{i=0}^{W.H-1} \sum_{j=0}^{W.W-1} W^{(t)}_{k,c,i,j} * X^{(t)}_{n,c,xh,xw} \tag{3.5}$$

Where X represents the input tensor. W are the filter weights with k, c, i, j representing the output channel size, input channel size, height position, and width position, respectively. B_k is a bias array. The size of the bias array is the same as the size of the output channel k.

Padding is realized with Equation (3.6), where the values n, c, xh, and xw, are the batch size, channel size, height position, and width position, respectively. The height and width positions are calculated by using xh and xw as shown in Equation (3.7). In turn, xh and xw are calculated with respect to the output tensor position yh, yw, slide (s), weight positions i, j, dilation (d), and padding offset (p).

$$X_{n,c,xh,xw} = \begin{cases} X_{n,c,xh,xw} & \text{if } 0 \leq xh < X.H \text{ and } 0 \leq xw < X.W \\ 0 & otherwise \end{cases} \tag{3.6}$$

$$xh = yh * s + i * d - p, \qquad xw = yw * s + j * d - p \tag{3.7}$$

The forward propagation function is executed into two steps. The first step is the sliding weight window over the input. This function returns the summation of the individual window as depicted in Algorithm 3. The second step stores the output of the previous layer as shown in Algorithm 4.

Algorithm 3 Convolution Forward Window - Equation(3.5).

Input:	X, W	Tensor
	x	input offset
	n,k	batch and neuron index
	d	dilation size
Returns:	sum	summation of W*X window

1: **function** ConvForwardWin(X,W,x,n,k,d)
2: $sum \leftarrow 0$
3: **for** $c = 0$ to W.C **do**
4: **for** $i = 0$ to W.H **do**
5: $xh \leftarrow x.h + i * d.h$ ▷ add dilation height offset for X
6: **if** $xh \geq 0$ and $xh < X.W$ **then**
7: **for** $j = 0$ to W.W **do**
8: $xw \leftarrow x.w + j * d.w$ ▷ add dilation width offset for X
9: **if** $xw \geq 0$ and $xw < X.H$ **then**
10: $sum \leftarrow sum + W[k][c][i][j] * X[n][c][xh][xw]$
return sum

Algorithm 4 Convolution Forward Propagation

Input:	X, W, B	Tensor
	p, s, d	Padding, stride and dilation sizes
Output:	Y	Tensor

1: **function** ConvForward( )
2:     **for** $n = 0$ to Y.N **do**
3:         **for** $k = 0$ to Y.C **do**
4:             $x.h \leftarrow -p.h$ ▷ set -padding height offset for X
5:             **for** $yh = 0$ to Y.H **do**
6:                 $x.w \leftarrow -p.w$ ▷ set -padding width offset for X
7:                 $x.h \leftarrow x.h + yh * s.h$ ▷ add stride height offset for X
8:                 **for** $yw = 0$ to Y.W **do**
9:                     $x.w \leftarrow x.w + yw * s.w$ ▷ add stride width offset for X
10:                     $sum \leftarrow ConvForwardWin(X, W, n, x, k, d)$
11:                     $Y[n][k][yh][yw] \leftarrow B[k] + sum$

3.2.2 Back Propagation

As in the case of a regular neural network, errors in a CNN are passed backward from the next layer. Each output error value is accumulated into two different tensors. The first is for the input tensor which is scaled according to the weights of each output as shown below.

$$\Delta X^{(t)}_{n,c,xh,xw} = \sum_{c=0}^{W.C-1} \sum_{i=0}^{W.H-1} \sum_{j=0}^{W.W-1} W^{(t)}_{k,c,i,j} * \Delta Y^{(t)}_{n,k,yh,yw} \tag{3.8}$$

Where, $\Delta X^{(t)}$ is the tensor that holds the errors for $X^{(t)}$, $W^{(t)}$ is the weight tensor for the layer, and $\Delta Y^{(t)}$ is the errors received for $Y^{(t)}$. Equation (3.8) is implemented in Algorithm 5.

Algorithm 5 Convolution Backward Data Window - Equation (3.8)

Input:	W	Weight tensor
	dy	gradient
	x	input offset value
	n,k	batch and neuron index
	d	dilation size
Output:	dX	Input error tensor

```
1: function ConvInputGrad(dX, W, dy, x, n, k, d)
2:     for c = 0 to W.C do
3:         for i = 0 to W.H do
4:             xh ← x.h + i * d.h
5:             if xh ≥ 0 and xh < X.W  then
6:                 for j = 0 to W.W do
7:                     xw ← x.w + j * d.w
8:                     if xw ≥ 0 and xw < X.H  then
9:                         dX[n][c][xh][xw] ← dX[n][c][xh][xw] + W[k][c][i][j] * dy
```

The second tensor is an accumulation of errors used to update the weights. It is obtained by multiplying the input values by the corresponding output errors as shown below:

$$\Delta W^{(t)}_{k,c,i,j} = \sum_{c=0}^{W.C-1} \sum_{i=0}^{W.H-1} \sum_{j=0}^{W.W-1} X^{(t)}_{n,c,xh,xw} * \Delta Y^{(t)}_{n,k,yh,yw} \tag{3.9}$$

Where $\Delta W^{(t)}$ is the tensor that holds the errors for the weights, $X^{(t)}$ is the input tensor for the layer, and $\Delta Y^{(t)}$ represents the corresponding errors from the output. Equation (3.9) is implemented in Algorithm 6.

Algorithm 6 Convolution Backward Weight Window - Equation (3.9)

Input:	X	Input tensor
	dy	gradient
	x	input offset value
	n,k	batch and neuron index
	d	dilation size
Output:	dW	Weight update tensor

```
1: function ConvWeightGrad(dW, X, dy, x, n, k, d)
2:     for c = 0 to W.C do
3:         for i = 0 to W.H do
4:             xh ← x.h + i ∗ d.h
5:             if xh ≥ 0 and xh < X.W  then
6:                 for j = 0 to W.W do
7:                     xw ← x.w + j ∗ d.w
8:                     if xw ≥ 0 and xw < X.H then
9:                         dW[k][c][i][j] ← dW[k][c][i][j] + X[n][c][xh][xw] ∗ dy
```

The error for the bias neurons is the summation of the output errors for that neuron as shown below.

$$\Delta B_k^{(t)} = \sum_{n=0}^{Y.N-1} \sum_{yh=0}^{Y.H-1} \sum_{yw=0}^{Y.W-1} \Delta Y_{n,k,yh,yw}^{(t)} \tag{3.10}$$

Where $\Delta B^{(t)}$ is the tensor that holds the errors for the Bias, and $\Delta Y^{(t)}$ is the output error.

Algorithm 7 Convolution Back Propagation

Input:	X, W, dY	input, weight and output error tensors
	p, s, d	padding, stride and dilation sizes
Output:	dX, dW, dB	input, weight and bias update tensors

```
1:  function ConvBackward(X, dX, W, dW, dB, dY, p, s, d)
2:      ZeroAll(dX)
3:      for n = 0 to Y.N do
4:          for k = 0 to Y.C do
5:              x.h ← −p.h
6:              for yh = 0 to Y.H do
7:                  x.w ← −p.w
8:                  x.h ← x.h + yh * s.h
9:                  for yw = 0 to Y.W do
10:                     x.w ← x.w + yw * s.w
11:                     dy ← dY[n][k][yh][yw]
12:                     ConvInputGrad(dX, W, dy, n, x, k, d)      ▷ Algorithm 5
13:                     ConvWeightGrad(dW, X, dy, n, x, k, d)     ▷ Algorithm 6
14:                     dB[k] ← dB[k] + dy
```

3.3 Activation Layer

The activation layer introduces non-linearity to a neural network. This operation is performed element-wise. During forward propagation, the activation function is applied to the output of the previous layer. Some of the common activation functions include logistic ($\frac{1}{1+e^{-x}}$), rectified linear unit (Relu, if $x \leq 0$ $f(x) = 0$, otherwise, $f(x) = x$), and the leaky rectified linear unit (leaky, if $x \leq 0$ $f(x) = 0.01$, otherwise, $f(x) = x$).

3.4 Weight Optimization

The weights in the network are updated at every training iteration. Several, approaches can be used to perform this update. Moreover, some of these approaches are guided by hyper-parameters that are either defined before training or adjusted during training. The choice of the hyper-parameters may dictate the ability of the network to converge. A summary of the main weight optimization approaches is provided next.

3.4.1 Gradient Descent

The simplest way to minimize the loss function at the output of the network is to update the weight in the direction of the gradient descent [33] as shown below.

$$W^{(t)} = W^{(t-1)} - \epsilon * \Delta W^{(t)} \tag{3.11}$$

Where, $W^{(t)}$ is the updated weight value at iteration t, $W^{(t-1)}$ is the weight value at iteration $t-1$, and $\Delta W^{(t)}$ is the weight error tensor. The hyper-parameter, ϵ, is called the learning rate and indicates the rate at which the updates are being performed.

3.4.2 Momentum Update

Compared to gradient descent, momentum update [28] takes into consideration the weight adjustment in the previous iterations as shown below:

$$M^{(t)} = \alpha * M^{(t-1)} - \epsilon * \Delta W^{(t)} \tag{3.12}$$

$$W^{(t)} = W^{(t-1)} + M^{(t)} \tag{3.13}$$

Where, $M^{(t)}$ is the momentum at iteration t, $M^{(t-1)}$ is the momentum at the previous iteration, α is the momentum rate, ϵ is the learning rate, $W^{(t)}$ is the updated weight, $W^{(t-1)}$ is the weight at the previous iteration, and $\Delta W^{(t)}$ is the weight error tensor. The hyper-parameters for this approach are ϵ and α.

3.4.3 Adagrad

Adagrad [34] stores the sum of the squares of the gradient for each individual parameter as shown in Equation (3.14). This value is then used to scale the gradient as shown in Equation (3.15). The hyper-parameter β in the Adagrad approach is used to prevent a divide by zero.

$$S^{(t)} = S^{(t-1)} + (\Delta W^{(t)})^2 \tag{3.14}$$

$$W^{(t)} = W^{(t-1)} + \frac{-\epsilon * \Delta W^{(t)}}{\sqrt{S^{(t)}} + \beta} \tag{3.15}$$

Where, $S^{(t)}$ is the sum of the squares of the gradients at iteration t, $S^{(t-1)}$ is the squares of the gradients at iteration $t-1$, $\Delta W^{(t)}$ is the weight error tensor, $W^{(t)}$ is the updated weight, $W^{(t-1)}$ is the weight value at the previous iteration, ϵ is the learning rate, and β is a meta parameter.

3.4.4 Adam

Adam [35] is a weight update approach that uses a large number of hyper-parameters and therefore, may require extensive tuning.

$$\Omega_1^{(t)} = \beta_1 * \Omega_1^{(t-1)} + (1 - \beta_1) * \Delta W^{(t)} \tag{3.16}$$

$$\Omega_{1_temp}^{(t)} = \frac{\Omega_1^{(t)}}{1 - \beta_1^{C+1}} \tag{3.17}$$

$$\Omega_2^{(t)} = \beta_2 * \Omega_2^{(t-1)} + (1 - \beta_2) * (\Delta W^{(t)})^2 \tag{3.18}$$

$$\Omega_{2_temp}^{(t)} = \frac{\Omega_2^{(t)}}{1 - \beta_2^{C+1}} \tag{3.19}$$

$$W^{(t)} = W^{(t-1)} + \frac{-(\epsilon * \Omega_{1_temp}^{(t)})}{\sqrt{\Omega_{2_temp}^{(t)}} + \alpha} \tag{3.20}$$

$$C^{(t)} = C^{(t-1)} + 1 \tag{3.21}$$

Where β_1, β_2, α, and ϵ are meta parameters, $\Omega_1^{(t)}$ and $\Omega_2^{(t)}$ are accumulated values that are updated during every iteration, C is a counter, $\Delta W^{(t)}$ is the weight error tensor, $W^{(t)}$ is the weight tensor for the current iteration, and $W^{(t-1)}$ is the weight tensor from the previous iteration.

3.5 Implementation

The CNN was initially implemented in Go and executed on a regular CPU. The advantage of Go is that it has built-in concurrency and is supported by several APIs for image processing. Unfortunately, the performance of this implementation was not practical. In order to increase performance, the compute intensive section of the algorithm was migrated to a GPU implementation using CUDA.

The Go language has a pseudo-package "CGO" [36] which was used to allow the front end of the application to call C functions including libraries that are compatible with GCC. This was necessary as an intermediary step. CUDA is based on CPP with few extensions. CUDA code is not directly compiled with GCC. CUDA code is compiled using NVCC. Therefore, CUDA kernels cannot be directly accessed using the CGO driver.

This limitation can be addressed in three ways. The first approach consists of creating a static or shared library of the kernels made for the GPU. This approach was not ideal since it entails making changes to the CUDA kernels.

The second approach consists of pre-compiling kernels into a .ptx file using the ptx option in NVCC. PTX code is similar to assembly language code and is compatible with different NVIDIA GPU architectures.

The third approach would use directly the NVRTC library. This approach creates a run time library. This third approach is similar to the second approach but with added flexibility. CUDA code can be compiled into a ptx format during runtime. In fact, the compiler can directly use the NVRTC library.

After evaluation, the second approach was used, because of a feature involving CUDA Contexts made using NVRTC inconsistent.

In order to access the functions that are in ptx form, the Driver API needs to be used. Specifically, a module needs to be made. Modules are dynamically loaded packages. A module can be created using NewModuleData() found in sub-package of GoCudnn called cuda. MakeKernel() will return a *Kernel that uses the method (*Kernel)Launch() to execute the kernel with the name passed in MakeKernel().

In order to allocate memory to the GPU. The Malloc() function of the cudart sub-package is an option. This function allocates memory to the GPU and this memory is managed by Go's garbage collector. In addition, Memcpy() is used to copy memory to or from the GPU.

Code examples can be seen in Appendices A, B, and C. Appendix A exposes a few lower level deep learning functions using the CPU. Appendix C exposes the lower level bindings that are used to execute kernels on a GPU. Appendix B exposes code that is used to build deep learning models for execution on a GPU.

4. RESULTS

In order to test the results two different CNNs were built. One is using ConvNetGo, and the other is using GoCuNets. They are both based on the model seen in Figure (4.1). This model was chosen because it implements dilation, strides, and padding for each layer. The data being used is stored in a tensor with the dims of [N,1,28,28]. Each convolution has a filter with spacial dims [4,4], stride [2,2], and dilation [3,3]. The input Padding for the first convolution was set at [6,6] to make the output spacial dims [16,16]. The next two convolutions have a padding of [4,4]. The spatial dims of the output is half the size of the input. This convolution network is tested on an application that classifies hand-written digits. The final classification layer uses the input of the convolution layer and selects the digit that corresponds to the input image. This last layer consists of a fully connected layer with 10 neurons and 320 weights each. The digit image used to test the CNN implementation are extracted

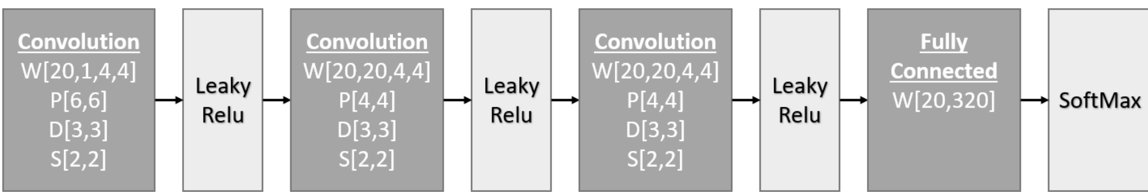


Fig. 4.1. The layers of the CNN used to test the proposed framework.

from the MNIST [37] database. MNIST is a good benchmarking database for testing CNN models because it converges quickly. It consists of 60,000 training samples and 10,000 testing samples of hand written arabic numerals. Each sample has a size of 28 by 28. The target classification labels are stored as a byte of 0-9. Since, the model uses a softmax classifier, the output needs to be represented using one-hot encoding. An example is shown in Table (4.1).

Table 4.1.
Samples taken from the MNIST database with the corresponding one-hot encoding of their labels.

Input	Target
7	[0,0,0,0,0,0,0,1,0,0]
2	[0,0,1,0,0,0,0,0,0,0]
1	[0,1,0,0,0,0,0,0,0,0]
0	[1,0,0,0,0,0,0,0,0,0]

Each network will run for 8 epochs. The average time it takes each epoch to complete is recorded along with the batch size. The ConvNetGo model is executed on a dual socket motherboard with two e5-2680v2 for a total of 20 cores and 40 threads. Each CPU has a 2.8 GHz clock with a boost clock of 3.6 GHz.

GoCuNets is executed on a GTX 1080ti GPU with a e5-2696v2 12 core / 24 thread CPU. The CPU has a 2.5 GHz base clock with a 3.3 GHz boost clock. The GPU has a base clock of 1.5 GHz and a boost of 1.6 GHz. Moreover, the GPU has 28 streaming multiprocessors with 128 CUDA cores each.

The execution times of the ConvNetGo and GoCuNets CNN classification models when applied to the MNIST database using different batches sizes are shown in Table 4.2. When the batch is set at 5, the model GoCuNes is twice as fast as ConvNetGo. When the batch increases to 5000, GoCuNets performs 57 times faster than ConvNetGo. The execution times for the two CNNs are plotted against the batch size in Figure(4.2).

Table 4.2.
Execution time (in seconds) per epoch for each batch size for the ConvNetGo (CPU implementation) and GoCuNets (GPU implementation).

Batch Size	**ConvNetGo**	**GoCuNets**	**CPU/GPU**
5	124.97	47.23	2.11
10	61.97	25.94	1.91
20	60.67	13.77	3.52
40	39.02	7.09	4.40
50	45.68	5.58	6.55
100	38.05	2.81	10.83
200	28.95	1.65	14.03
400	27.13	0.98	22.15
500	27.46	0.81	27.12
1000	27.51	0.56	39.30
2000	27.92	0.43	51.95
5000	27.08	0.38	57.02

Table 4.3 shows the training time for the GoCuNets CNN with varying batch sizes. The training time is measured based on the same convergence criteria across all experiments. Each epoch consists of training and testing runs. The number of runs were determined by the size of the data-set divided by the batch size. The weights were trained during the training runs using the training data. The classification was then performed on the testing runs using the testing data. Training was complete when the average loss across the MNIST testing data-set is less than 0.01.

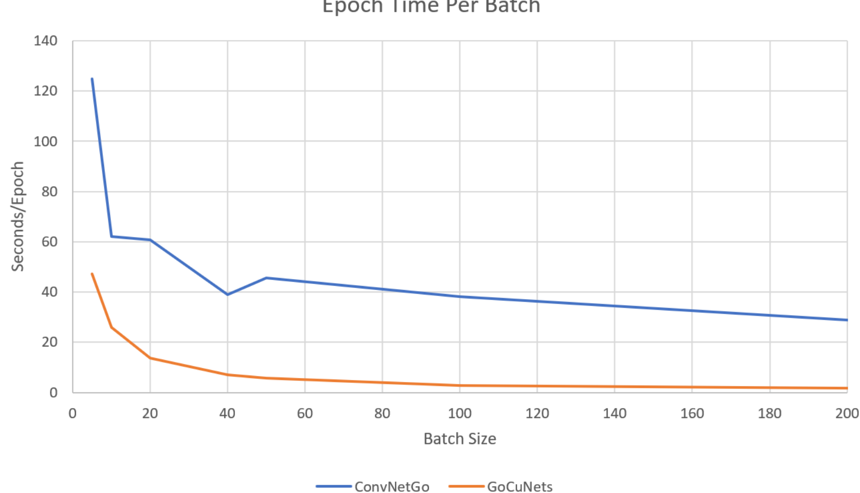


Fig. 4.2. Execution time per epoch for CoCuNets (GPU implementation) and ConvNetGo (CPU implementation) with increasing batch size.

4.1 Code Samples

Appendix A contains a few functions from ConvNetGo. These examples implement some of the algorithms shown in this chapter. Appendix A.1 is an example of the data structure of a Tensor. Some methods and functions include Convolution Forward (Appendices A.2.2 and A.2.3), Fully Connected Forward (Appendix A.3.3), and Leaky Relu (Appendix A.3.2).

Appendix B contains a few functions from GoCuNets. It mostly covers the methods that type Builder (Appendix B.2) uses. Builder is used to build deep learning models on a GPU. It contains methods such as ConvolutionLayer() (Appendix B.2.2) and Activation() (Appendix B.2.4) to create layers. These layers can be used to create structures that implement the Module Interface (Appendix B.3), like VanillaModule (Appendix B.3.1) and SimpleModuleNetwork (Appendix B.3.2).

Table 4.3.
Number of epochs, training time for each batch size when executing the GoCuNets (GPU) model. Convergence was decided when the average loss for the testing data was less than 0.01.

BatchSize	**N Epochs**	**Learn Time (s)**
5	8	377.84
10	7	181.58
20	6	82.62
40	9	63.81
50	9	50.22
100	11	30.91
200	15	24.75
400	22	21.56
500	25	20.25
1000	39	21.84
2000	62	26.66
5000	122	46.36

Appendix C contains a few samples taken from GoCudnn such as memory allocation on the GPU (Appendix C.7) and memory copy (Appendix C.8). The Appendix also includes sample GPU io.Reader and io.Writer interfaces (Appendix C.9 and C.10). In addition, Appendix C shows the use of ConcatEX (Appendix C.6) to execute a concat kernel (Appendix C.3) using (*Kernel)Launch() (Appendix C.5).

5. CONCLUSION

The proposed GoCuNets shows that deep learning can be implemented using Go and GPUs. In fact, GoCuNets was able to achieve fast learning times. For instance, for the number digit classification application, the network converges in 25 epochs to an average loss less than .01. The total execution time for all epochs is 20.25 seconds with a batch size equal to 500. Assuming, ConvNetGo takes the same number of epochs to converge, a smaller batch size of 40 would require 5 minutes and 51 seconds. By using GPUs, GoCuNets makes developing deep learning models in Go a practical option.

5.1 Challenges

Creating and using custom kernels is not very intuitive. Using NVCC to generate a ptx file for custom kernels, and copying the contents of the ptx file into a constant string literal in a Go package makes creating GPU kernels challenging.

Launching a GPU kernel also requires several setup steps. These steps include allocating CPU and GPU memory, assigning values to the CPU memory, and then copying the values from CPU to GPU. Except for single value constants, this process has to be done for every value passed to a kernel. Moreover, when launching a kernel, the user must specify the number of blocks per grid and the number of threads per block. Finally, in order to evaluate a returned value on the CPU, the corresponding memory needs to be copied from the GPU back to the CPU.

5.2 Future Work

There are several directions for future work. Even though a large speed-up was achieved using a GPU, additional speed-up can be achieved by using multiple GPUs in parallel.

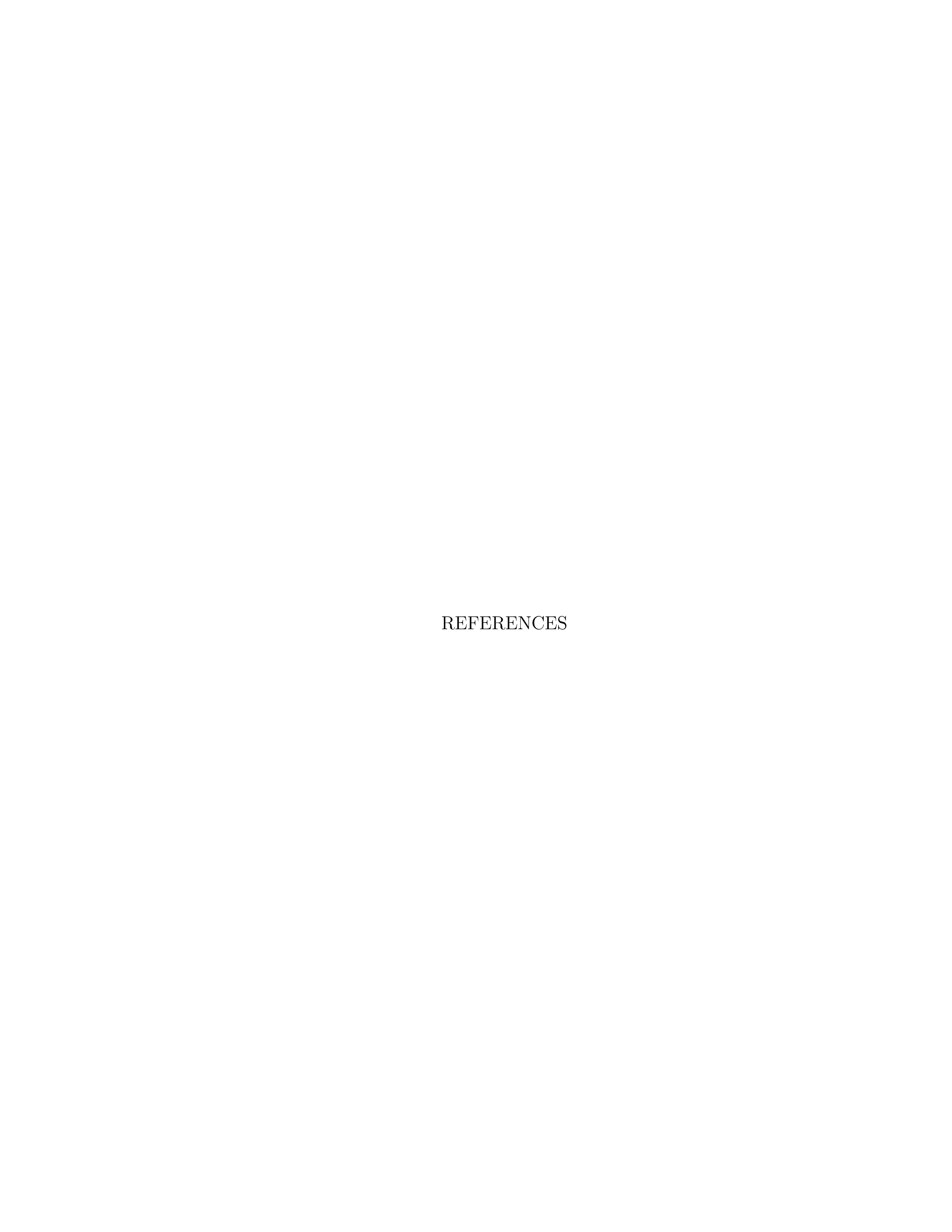
Moreover, the only GPUs currently supported under GoCuNets are Nvidia's. Either integrating AMD GPUs into GoCuNets or creating a new framework that supports both Nvidia and AMD GPUs would make deep learning using Go more widely available.

5.3 GoCuNets

GoCuNets is a GPU centric deep learning framework written in Go. It is available at www.github.com/dereklstinson/gocunets.

GoCudnn are bindings for Cuda. These bindings are used in GoCuNets. The package is available at www.github.com/dereklstinson/gocudnn.

ConvNetGo is a CPU centric deep learning framework written in Go. It uses Go's built in concurrency to execute deep learning models. It is available at www.github.com/dereklstinson/convnetgo.



REFERENCES

REFERENCES

[1] "Go frequently asked questions (faq)," last accessed: 05/07/2020. [Online]. Available: https://golang.org/doc/faq

[2] "Go codetools," last accessed: 05/07/2020. [Online]. Available: https://github.com/golang/go/wiki/CodeTools

[3] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, last accessed: 05/07/2020. [Online]. Available: https://www.tensorflow.org/

[4] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," 2017, last accessed: 05/07/2020. [Online]. Available: https://openreview.net/pdf?id=BJJsrmfCZ

[5] F. Chollet *et al.*, "Keras," 2015, last accessed: 05/07/2020. [Online]. Available: https://keras.io

[6] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," 2015, last accessed: 05/07/2020. [Online]. Available: https://arxiv.org/abs/1512.01274

[7] S. Tokui, R. Okuta, T. Akiba, Y. Niitani, T. Ogawa, S. Saito, S. Suzuki, K. Uenishi, B. Vogel, and H. Yamazaki Vincent, "Chainer: A deep learning framework for accelerating the research cycle," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. ACM, 2019, pp. 2002–2011, last accessed: 05/07/2020. [Online]. Available: https://arxiv.org/abs/1908.00213

[8] "Gonum," last accessed: 05/07/2020. [Online]. Available: https://www.gonum.org/

[9] "Gocv," last accessed: 05/07/2020. [Online]. Available: https://gocv.io/

[10] D. Stinson, "Convnetgo," last accessed: 05/07/2020. [Online]. Available: https://github.com/dereklstinson/convnetgo

[11] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cudnn: Efficient primitives for deep learning," 2014.

[12] D. Stinson, “Gocudnn,” last accessed: 05/07/2020. [Online]. Available: https://github.com/dereklstinson/GoCudnn

[13] “Hip,” last accessed: 05/07/2020. [Online]. Available: https://github.com/ROCm-Developer-Tools/HIP

[14] D. Stinson, “Hipgo,” last accessed: 05/07/2020. [Online]. Available: https://github.com/dereklstinson/hip

[15] J. Khan, P. Fultz, A. Tamazov, D. Lowell, C. Liu, M. Melesse, M. Nandhimandalam, K. Nasyrov, I. Perminov, T. Shah, V. Filippov, J. Zhang, J. Zhou, B. Natarajan, and M. Daga, “Miopen: An open source library for deep learning primitives,” 2019, last accessed: 05/07/2020. [Online]. Available: https://arxiv.org/abs/1910.00078

[16] D. Stinson, “Miopengo,” last accessed: 05/07/2020. [Online]. Available: https://github.com/dereklstinson/miopen

[17] D. L. Stinson, “Gocunets,” last accessed: 05/07/2020. [Online]. Available: https://github.com/dereklstinson/GoCuNets

[18] Barnex, “Cuda5,” last accessed: 05/07/2020. [Online]. Available: https://github.com/barnex/cuda5

[19] “Gorgoniacu,” last accessed: 05/07/2020. [Online]. Available: https://github.com/gorgonia/cu

[20] “Unixpckle/cuda,” last accessed: 05/07/2020. [Online]. Available: https://github.com/unixpickle/cuda

[21] J. Nickolls, I. Buck, M. Garland, and K. Skadron, “Scalable parallel programming with cuda,” *Queue*, vol. 6, no. 2, p. 40–53, 2008. [Online]. Available: https://doi.org/10.1145/1365490.1365500

[22] “Go’s release history,” last accessed: 05/07/2020. [Online]. Available: https://golang.org/doc/devel/release.html

[23] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, “Caffe: Convolutional architecture for fast feature embedding,” *arXiv preprint arXiv:1408.5093*, 2014, last accessed: 05/07/2020. [Online]. Available: https://arxiv.org/abs/1408.5093

[24] “Caffe 2,” last accessed: 05/07/2020. [Online]. Available: https://caffe2.ai/blog/2017/04/18/caffe2-open-source-announcement.html

[25] A. Karpathy, “Convnetjs,” last accessed: 05/07/2020. [Online]. Available: https://github.com/karpathy/convnetjs

[26] “Gorgonia,” last accessed: 05/07/2020. [Online]. Available: https://gorgonia.org/

[27] “Fully connected layer,” last accessed: 05/07/2020. [Online]. Available: https://www.mathworks.com/help/deeplearning/ref/nnet.cnn.layer.fullyconnectedlayer.html

[28] G. O. Y. LeCun, L. Bottou and K. Muller, "Efficient backprop," in *Neural Networks Tricks of the trade*. Springer, 1998.

[29] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[30] *CUDNN DEVELOPERS GUIDE*. Nvidia, 2020, last accessed: 05/07/2020. [Online]. Available: https://docs.nvidia.com/deeplearning/sdk/cudnn-developer-guide/index.html

[31] E. O., *TensorFlow* Optimizations on Modern Intel® Architecture*. Intel, 2017, last accessed: 05/07/2020. [Online]. Available: https://software.intel.com/en-us/articles/tensorflow-optimizations-on-modern-intel-architecture

[32] *Deep Learning Performance Guide*. Nvidia, 2019, last accessed: 05/07/2020. [Online]. Available: https://docs.nvidia.com/deeplearning/sdk/dl-performance-guide/index.html

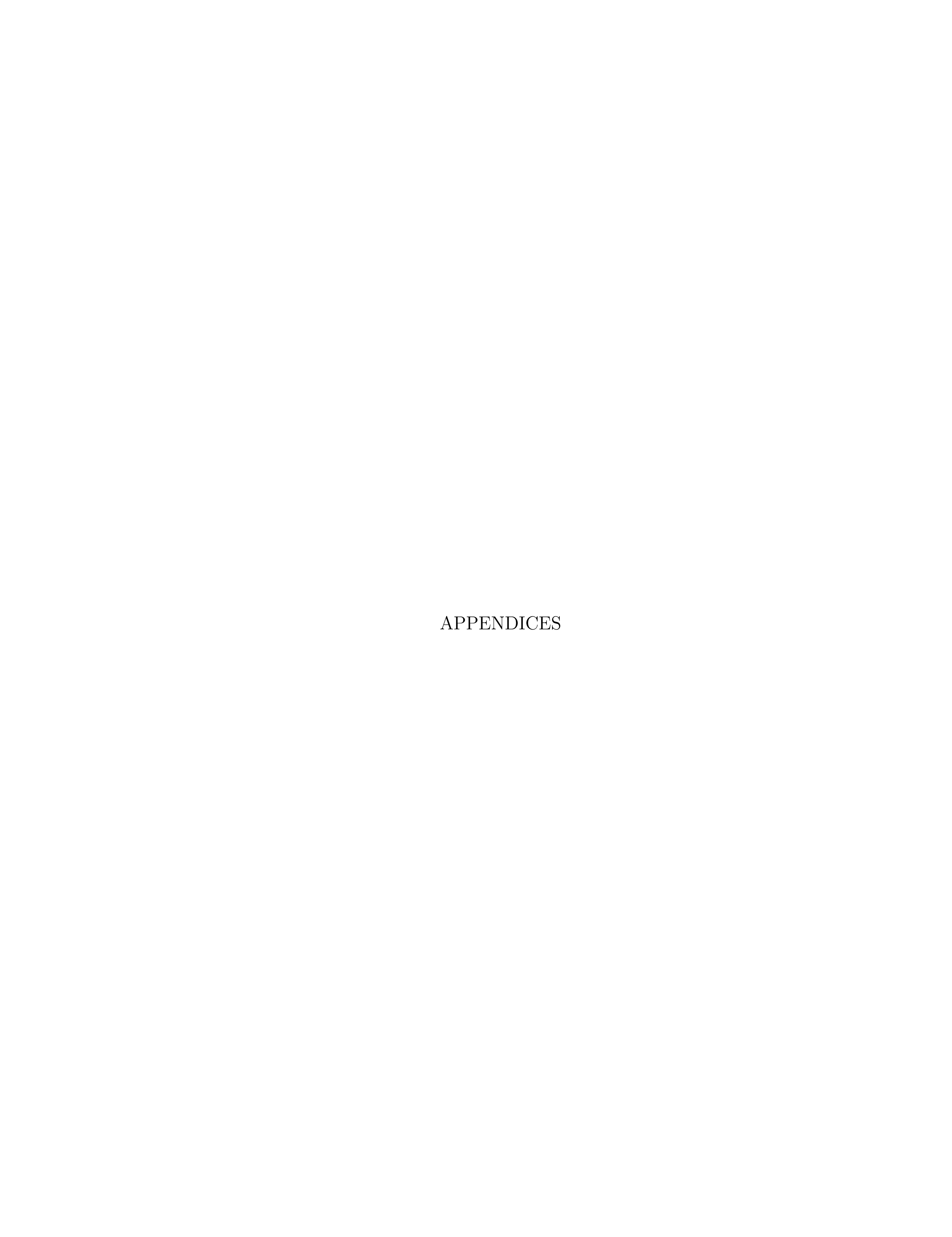
[33] Y. Bengio, "Practical recommendations for gradient-based training of deep architectures," *CoRR*, vol. abs/1206.5533, 2012. [Online]. Available: http://arxiv.org/abs/1206.5533

[34] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *J. Mach. Learn. Res.*, vol. 12, pp. 2121–2159, Jul. 2011. [Online]. Available: http://dl.acm.org/citation.cfm?id=1953048.2021068

[35] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015, 2014, last accessed: 05/07/2020. [Online]. Available: https://arxiv.org/abs/1412.6980

[36] "Command cgo," last accessed: 05/07/2020. [Online]. Available: https://golang.org/cmd/cgo/

[37] Y. LeCun, C. Cortes, and C. Burges, "Mnist handwritten digit database," *ATT Labs [Online]. Available: http://yann. lecun. com/exdb/mnist*, vol. 2, 2010.



APPENDICES

A. CONVNETGO

A.1 Tensor

```
1 type Tensor struct {
2     dims     []int
3     stride   []int
4     f32data  []float32
5     nhwc     bool
6 }
```

A.2 Convolution

A.2.1 Convolution Struct

```
1 //Convolution contains the parameters
2 //that are used to do a convolution
3 type Convolution struct {
4     padding, dilation, stride []int
5     set                       bool
6     nhwc                      bool
7 }
```

A.2.2 Convolution Forward Algorithm

```
1  func (c *Convolution) convnhwc4dwithwindow(x, w, wb, y *Tensor) {
2      var wg sync.WaitGroup
3      for yn := 0; yn < x.dims[0]; yn++ {
4          wg.Add(1) //Add to wait group
5          go func(yn int) { //concurrent execution
6              sh := -c.padding[0]
7              for yh := 0; yh < y.dims[1]; yh++ {
8                  sw := -c.padding[1]
9                  for yw := 0; yw < y.dims[2]; yw++ {
10                     for yc := 0; yc < y.dims[3]; yc++ {
11                         dh, dw := c.dilation[0], c.dilation[1]
12                         adder := c.fwdwinnhwc4d(x, w,
13                         sh, sw, dh, dw, yn, yc)
14                         adder += wb.f32data[wn] //add the bias
15                         y.Set(adder, []int{yn, yh, yw, yc})
16                     }
17                     sw += c.stride[1]
18                 }
19                 sh += c.stride[0]
20             }
21             wg.Done()//give done signal
22         }(yn)//end of concurrent function
23     }
24     wg.Wait()//Wait for threads to finish.
25 }
```

A.2.3 Convolution Forward Window

```
func (c *Convolution) fwdwinnhwc4d(
                        x, w *Tensor,
                        sh, sw, dh, dw int,
                        xn, wn int) (adder float32) {

    for wh := 0; wh < w.dims[1]; wh++ {
        xh := sh + (wh * dh)
        if xh >= 0 && xh < x.dims[1] {
            for ww := 0; ww < w.dims[2]; ww++ {
                xw := sw + (ww * dw)
                if xw >= 0 && xw < x.dims[2] {
                    for wc := 0; wc < w.dims[3]; wc++ {
                        adder += x.f32data[(x.stride[0]*xn)+
                            (x.stride[1]*xh)+
                            (x.stride[2]*xw)+
                            (x.stride[3]*wc)] *
                            w.f32data[(w.stride[0]*wn)+
                                (w.stride[1]*wh)+
                                (w.stride[2]*ww)+
                                (w.stride[3]*wc)]
                    }
                }
            }
        }
    }
    return adder
}
```

A.3 Leaky

A.3.1 Leaky Struct

```
1 //LeakyRelu is a struct that holds the neg and pos coef
2 type LeakyRelu struct {
3     negcoef, poscoef float32
4 }
```

A.3.2 Leaky Forward

```
1     func (l *LeakyRelu) Forward(x, y *Tensor) (err error) {
2     nbatches := x.dims[0]
3     nbatchelements := x.stride[0]
4     var wg sync.WaitGroup
5     for i := 0; i < nbatches; i++ {
6         wg.Add(1)
7         batchoffset := i * nbatchelements
8         go func(batchoffset, nbatchelements int) {
9             for j := 0; j < nbatchelements; j++ {
10                if x.f32data[batchoffset+j] < 0 {
11                    y.f32data[batchoffset+j] =
12                    x.f32data[batchoffset+j] * l.negcoef
13                } else {
14                    y.f32data[batchoffset+j] =
15                     x.f32data[batchoffset+j] * l.poscoef
16                }
17            }
18            wg.Done()
19        }(batchoffset, nbatchelements)
20    }
21    wg.Wait()
22    return nil
23 }
```

A.3.3 Fully Connected

```
1  func FullyConnectedForward(x, w, b, y *Tensor) error {
2      nvol := findvolume(w.dims[1:])
3      xbatchstride := x.stride[0]
4      ybatchstride := y.stride[0]
5      neurons := w.dims[0]
6      var wg sync.WaitGroup
7      for i := 0; i < x.dims[0]; i++ {
8          wg.Add(1)
9          go func(i int) {
10             yoffset := ybatchstride * i
11             xboffset := xbatchstride * i
12             for j := 0; j < neurons; j++ {
13                 neuronoffset := w.stride[0] * j
14                 var adder float32
15                 for k := 0; k < xbatchstride; k++ {
16                     adder += w.f32data[neuronoffset+k]
17                     * x.f32data[xboffset+k]
18                 }
19                 y.f32data[yoffset+j] = adder + b.f32data[j]
20             }
21             wg.Done()
22         }(i)
23     }
24     wg.Wait()
25     return nil
26 }
```

B. GOCUNETS

B.1 Tensor

```
type Tensor struct {
        *layers.Tensor
        id       int64
        to, from Module
    }
```

```
//Layer is a layer inside a network it holds inputs and outputs
type Layer struct {
        id             int64
        name           string
        h              *Handle
        op             Operation
        workspacefwd   *nvidia.Malloced
        workspacebwd   *nvidia.Malloced
        workspacebwf   *nvidia.Malloced
        x, dx, y, dy   *Tensor
    }
```

B.2 Builder

B.2.1 Builder Data Structure

```
1   //Builder will create layers with the flags set within the struct
2   type Builder struct {
3           h           *Handle
4           gpurng      *curand.Generator
5           Frmt        TensorFormat
6           Dtype       DataType
7           Cmode       ConvolutionMode
8           Mtype       MathType
9           Pmode       PoolingMode
10          AMode       ActivationMode
11          BNMode      BatchNormMode
12          Nan         NanProp
13          curngtype   curand.RngType
14      }
```

B.2.2 Builder Method Convolution Layer

```
1  //ConvolutionLayer creates a convolution layer
2  func (l *Builder) ConvolutionLayer(
3  id int64, groupcount int32,
4  w, dw, b, db *Tensor,
5  pad, stride, dilation []int32) (conv *Layer, err error) {
6          clayer, err := cnn.SetupBasic(
7          l.h.Handler,
8          l.Frmt.TensorFormat,
9          l.Dtype.DataType,
10         l.Mtype.MathType,
11         groupcount,
12         w.Tensor, dw.Tensor, b.Tensor, db.Tensor,
13         l.Cmode.ConvolutionMode,
14         pad,
15         stride,
16         dilation)
17         if err != nil {
18                 return nil, err
19             }
20         conv, err = createlayer(id, l.h, clayer)
21         if err != nil {
22                 return nil, err
23             }
24         return conv, nil
25     }
```

B.2.3 Builder Method Convolution Weights

```
1   //CreateConvolutionWeights creates the weights and
2   //delta weights of a convolution layer
3   func (l *Builder) CreateConvolutionWeights(dims []int32) (
4   w, dw, b, db *Tensor, err error) {
5           w, err = l.CreateTensor(dims)
6           if err != nil {
7                   return nil, nil, nil, nil, err
8               }
9           dw, err = l.CreateTensor(dims)
10          if err != nil {
11                  return nil, nil, nil, nil, err
12              }
13          b, err = l.CreateBiasTensor(dims)
14          if err != nil {
15                  return nil, nil, nil, nil, err
16              }
17          db, err = l.CreateBiasTensor(dims)
18          if err != nil {
19                  return nil, nil, nil, nil, err
20              }
21          return w, dw, b, db, nil
22      }
```

B.2.4 Builder Method Activation Layer

```
1  //Activation creates an activation layer
2  func (l *Builder) Activation(id int64) (a *Layer, err error) {
3  var act *activation.Layer
4  aflg := l.AMode
5  switch l.AMode {
6  case aflg.Leaky():
7  act, err = activation.Leaky(l.h.Handler, l.Dtype.DataType)
8  case aflg.ClippedRelu():
9  act, err = activation.ClippedRelu(l.h.Handler,
10 l.Dtype.DataType)
11 case aflg.Relu():
12 act, err = activation.Relu(l.h.Handler, l.Dtype.DataType)
13 case aflg.Elu():
14 act, err = activation.Elu(l.h.Handler, l.Dtype.DataType)
15 case aflg.Sigmoid():
16 act, err = activation.Sigmoid(l.h.Handler, l.Dtype.DataType)
17 case aflg.Tanh():
18 act, err = activation.Tanh(l.h.Handler, l.Dtype.DataType)
19 default:
20 return nil, errors.New("AppendActivation:"+
21 "Not supported Activation Layer")
22 }
23 if err != nil {
24         return nil, err
25     }
26 a, err = createlayer(id, l.h, act)
27
28 return a, err
29 }
```

B.3 Module Interface

```
1     //Module is a wrapper around
2     //a neural network or set of operations
3     type Module interface {
4             ID() int64
5             Forward() error
6             Backward() error
7             FindOutputDims() ([]int32, error)
8             Inference() error
9             InitHiddenLayers() (err error)
10            InitWorkspace() (err error)
11            GetTensorX() (x *Tensor)
12            GetTensorDX() (dx *Tensor)
13            GetTensorY() (y *Tensor)
14            GetTensorDY() (dy *Tensor)
15            SetTensorX(x *Tensor)
16            SetTensorDX(dx *Tensor)
17            SetTensorY(y *Tensor)
18            SetTensorDY(dy *Tensor)
19        }
```

B.3.1 VanillaModule

```
1   //VanillaModule has a convolution and an activation
2   type VanillaModule struct {
3           id          int64
4           b           *Builder
5           conv        *Layer
6           act         *Layer
7       }
```

```
1   //Forward  satisfies module interface
2   func (m *VanillaModule) Forward() error {
3           err := m.conv.Forward()
4           if err != nil {
5                   return err
6               }
7           return m.act.Forward()
8       }
```

B.3.2 ModuleNetwork

```
//SimpleModuleNetwork is a simple module network
type SimpleModuleNetwork struct {
        id              int64
        Modules         []Module
        Output          *OutputModule
        Classifier      *ClassifierModule
        b               *Builder
    }
```

```
//Forward performs the forward operation
//of the simple module network
//Forward satasifies the Module Interface.
func (m *SimpleModuleNetwork) Forward() (err error) {
for _,mod := range m.Modules {
err = mod.Forward()
if err != nil {
        return err
    }
}
if m.Output != nil {
        err = m.Output.Forward()
        if err != nil {
                return err
            }
    }
if m.Classifier != nil {
        return m.Classifier.PerformError()
    }
return nil
}
```

C. GOCUDNN

C.1 Linking Cuda to Go

```
1  package cuda
2  /*
3  #cgo LDFLAGS:-L/usr/local/cuda/lib64 -lcuda
4  #cgo CFLAGS: -I/usr/local/cuda/include/
5  */
6  import "C"
```

C.2 NewModuleData

```
1  //NewModuleData takes a io.Reader and creates a Module with it.
2  func NewModuleData(r io.Reader) (*Module, error) {
3  ptxbytes, err := ioutil.ReadAll(r)
4  if err != nil {
5          return nil,err
6      }
7  var mod C.CUmodule
8  cptx := C.CString(string(ptxbytes))
9  defer C.free((unsafe.Pointer)(cptx))
10 err = status(C.cuModuleLoadData(&mod,
11                 (unsafe.Pointer)(cptx))).error("NewModuleData")
12 return &Module{
13 m:         mod,
14 loaded: true,
15 }, err
16 }
```

C.3 Cuda Concat Kernel

```
1  extern "C"
2  __global__ void ConcatNCHWEX(const int XThreads,
3  const int Batches,
4  const int DestBatchVol,
5  const int DestChannelOffset,
6  float *src,
7  const float alpha,
8  const int SrcBatchVol,
9  float *dest,
10 const float beta,
11 bool forward){
12 for (int i=0;i<Batches;i++){
13 GRID_AXIS_LOOP(idX,XThreads,x){
14 int deststride = (i*DestBatchVol)+(DestChannelOffset+idX);
15 int srcstride = (i*SrcBatchVol)+(idX);
16 if (forward){
17 dest[deststride]=src[srcstride]*alpha + dest[deststride]*beta;
18 }else{
19 src[srcstride]=dest[deststride]*alpha + src[srcstride]*beta;
20 }}}}
```

C.4 MakeKernel

```
1   //MakeKernel makes a kernel.
2   //kname is the kernels name that was written in cuda.
3   func MakeKernel(kname string, m *Module) (
4       k *Kernel, err error) {
5   var kern C.CUfunction
6   if m.loaded == false {
7           return nil, errors.New("MakeKernel: Module Not Loaded")
8       }
9   name := C.CString(kname)
10  defer C.free((unsafe.Pointer)(name))
11  err = status(
12      C.cuModuleGetFunction(&kern, m.m, name)).error("MakeKernel")
13  if err != nil {
14          return nil, err
15      }
16  k = &Kernel{
17  name: kname,
18  m:    m,
19  f:    kern,
20  }
21  //Give k to Go's Garbage Collector
22  runtime.SetFinalizer(k, destroycudakernel)
23  return k, nil
24  }
```

C.5 (k *Kernel) Launch()

```
1  //Launch will launch a kernal that's been assigned to it.
2  func (k *Kernel) Launch(gx, gy, gz, bx, by, bz,
3  shared uint32, stream gocu.Streamer,
4  args ...interface{}) error {
5  var shold unsafe.Pointer
6  if stream != nil {
7          shold = stream.Ptr()
8      }
9  cargs := makelaunchargs(len(args))
10 err := k.ifacetounsafecomplete(args, cargs)
11 if err != nil {
12         return err
13     }
14 return status(C.cuLaunchKernel(k.f,
15 C.uint(gx), C.uint(gy), C.uint(gz),
16 C.uint(bx), C.uint(by), C.uint(bz),
17 C.uint(shared),
18 (C.CUstream)(shold),
19 &cargs.args[0], C.voiddptrnull,
20 )).error("(k *Kernel) Launch()")
21
22 }
```

C.6 Concat Kernel Launch

```
1  func (c *ConcatEx) op(h *Handle,
2                    srcs []*gocudnn.TensorD, srcsmem []cutil.Mem,
3                    alpha float64,
4                    dest *gocudnn.TensorD, destmem cutil.Mem,
5                    beta float64, forward bool) error {
6                        dfrmt, ddtype, ddims, _, _ := dest.Get()
7                        batches := ddims[0]
8                        destbatchvol := findvol(ddims[1:])
9                        var srcchanoffset int32
10
11     for i:=range srcs{
12         srcdims := srcs[i].Dims()
13         srcbatchvol := findvol(srcdims[1:])
14         a := float32(alpha)
15         b := float32(beta)
16         config := h.LaunchConfig(srcbatchvol)
17         err = c.fp32.nchw.Launch(
18                    config.BlockCount, 1, 1, // grid parameters
19                    config.ThreadPerBlock, 1, 1, //block parameters
20                    0, c.streams[i], //shared memory, stream
21                    //rest are arguments
22                    config.Elements, batches,
23                    destbatchvol, srcchanoffset,
24                    srcsmem[i],
25                    a, srcbatchvol,
26                    destmem, b, forward)
27                    if err != nil {
28                        return err
29                    }
30                    srcchanoffset += srcbatchvol
31     }
32 }
```

C.7 MallocManagedGlobalEx

```
1      //MallocManagedGlobalEx allocates memory to
2      //device associated with w.
3      //If w is nil then it will
4      //behave like MallocManagedGlobal
5  func MallocManagedGlobalEx(w *gocu.Worker,
6                              mem cutil.Mem, size uint) error {
7      var err error
8      if w != nil {
9          err = w.Work(func() error {
10             err = status(C.cudaMallocManaged(
11             mem.DPtr(), C.size_t(size),
12             C.cudaMemAttachGlobal)).error("MallocManagedGlobalEx")
13             if err != nil {
14                 return err
15             }
16             return Memset(mem, 0, (size))
17         })
18     } else {
19         err = status(C.cudaMallocManaged(mem.DPtr(), C.size_t(size),
20         C.cudaMemAttachGlobal)).error("MallocManagedGlobalEx")
21         if err != nil {
22             return err
23         }
24         err = Memset(mem, 0, (size))
25     }
26     if err != nil {
27         return err
28     }
29     //Put Go Garbage Collector in charge of CUDA memory.
30     runtime.SetFinalizer(mem, devicefreemem)
31     return nil
32 }
```

C.8 Copy Memory

```
1   //Memcpy copies some memory from src to dest.
2   //If default is selected and if the system supports
3   //unified virtual addressing then the transfer is inferred.
4   func Memcpy(dest, src cutil.Pointer, sizet uint,
5                   kind MemcpyKind) error {
6       return status(C.cudaMemcpy(dest.Ptr(), src.Ptr(),
7        C.size_t(sizet), kind.c())).error("Memcpy")
8   }
```

C.9 Interface io.Reader

```
1  //Read satisfies the io.Reader interface
2  func (r *ReadWriter) Read(b []byte) (n int, err error) {
3  if r.i >= r.size {
4  r.Reset()
5  return 0, io.EOF
6  }
7  if len(b) == 0 {
8          return 0, nil
9      }
10 var size = r.size - r.i
11 if uint(len(b)) < size {
12         size = uint(len(b))
13     }
14 bwrap, _ := cutil.WrapGoMem(b)//don't need to check error,
15 //because []byte is supported by function.
16 if r.s != nil {
17         err = cudart.MemcpyAsync(bwrap, cutil.Offset(r, r.i),
18         size, r.cpflg, r.s)
19     } else {
20         err = cudart.Memcpy(bwrap, cutil.Offset(r, r.i),
21         size, r.cpflg)
22     }
23 if err != nil {
24         return 0, nil
25     }
26 r.i += size
27 n = int(size)
28 return n, nil
29 }
```

C.10 Interface io.Writer

```
1  //Write satisfies the io.Writer interface
2  func (r *ReadWriter) Write(b []byte) (n int, err error) {
3  if r.i >= r.size {
4  r.Reset()
5  return 0, errors.New("(r *ReadWriter) Write()" +
6  "Write Location Out of Memory")
7  }
8  if len(b) == 0 {
9          return 0, nil
10     }
11 var size = r.size - r.i
12 if uint(len(b)) < size {
13         size = uint(len(b))
14     }
15 bwrap, _ := cutil.WrapGoMem(b)//don't need to check error,
16 //because []byte is supported by function.
17 if r.s != nil {
18         err = cudart.MemcpyAsync(cutil.Offset(r, r.i), bwrap,
19         size, r.cpyflg, r.s)
20     } else {
21         err = cudart.Memcpy(cutil.Offset(r, r.i), bwrap,
22         size, r.cptflg)
23     }
24 r.i += size
25 n = int(size)
26 return n, err
27 }
```