



# Deep Learning with C#, .Net and Kelp.Net

The Ultimate Kelp.Net Deep Learning Guide

MATT R. COLE





# Deep Learning with C#, .NET and Kelp.NET

*The Ultimate Kelp.Net  
Deep Learning Guide*

by  
**Matt R. Cole**



---

**FIRST EDITION 2019**

**Copyright © BPB Publications, India**

**ISBN: 978-93-88511-018**

All Rights Reserved. No part of this publication may be reproduced or distributed in any form or by any means or stored in a database or retrieval system, without the prior written permission of the publisher with the exception to the program listings which may be entered, stored and executed in a computer system, but they can not be reproduced by the means of publication.

### **LIMITS OF LIABILITY AND DISCLAIMER OF WARRANTY**

The information contained in this book is true to correct and the best of author's & publisher's knowledge. The author has made every effort to ensure the accuracy of these publications, but cannot be held responsible for any loss or damage arising from any information in this book.

All trademarks referred to in the book are acknowledged as properties of their respective owners.

### **Distributors:**

#### **BPB PUBLICATIONS**

20, Ansari Road, Darya Ganj  
New Delhi-110002  
Ph: 23254990/23254991

#### **DECCAN AGENCIES**

4-3-329, Bank Street,  
Hyderabad-500195  
Ph: 24756967/24756400

#### **MICRO MEDIA**

Shop No. 5, Mahendra Chambers,  
150 DN Rd. Next to Capital Cinema,  
V.T. (C.S.T.) Station, MUMBAI-400 001  
Ph: 22078296/22078297

#### **BPB BOOK CENTRE**

376 Old Lajpat Rai Market,  
Delhi-110006  
Ph: 23861747

Published by Manish Jain for BPB Publications, 20 Ansari Road, Darya Ganj,  
New Delhi-110002 and Printed by him at Repro India Ltd, Mumbai

## About the Author

**Matt R. Cole** is a seasoned developer and author with over 30 years' experience in Microsoft Windows, C, C++, C# and .Net. He is the owner of Evolved AI Solutions, a premier provider of advanced Machine Learning/Bio-AI technologies. He developed the first enterprise-grade microservice framework (written completely in C# and .Net) used by a major hedge fund in NYC and also developed the first Bio Artificial Intelligence Swarm framework which completely integrates mirror and canonical neurons. He continues to push the limits of Machine Learning, Biological or Swarm Artificial Intelligence, Deep Learning and MicroServices. In his spare time, he continues his education taking every available course in advanced Mathematics, AL/ML/DL, Quantum Mechanics/Physics, String Theory and Computational Neuroscience and contributes to open source efforts such as Kelp.Net.

---

## Reviewer

**Gaurav Arora** has done M.Phil in computer science. He is a Microsoft X-MVP, life time member of Computer Society of India (CSI), Advisory member of IndiaMentor, certified as a scrum trainer/coach, XEN for ITIL-F and APMG for PRINCE-F and PRINCE-P. He is an Open source developer, contributor to TechNet Wiki, Founder of Ovatic Systems Private Limited. In 22+ years of his career, he has mentored thousands of students and industry professionals. You can tweet Gaurav on his twitter handle @g\_arora.

---

## Preface

For those of you who are C# developers, you know how painfully hard it is to find good examples of how to implement deep learning in your applications without resorting to using languages such as Python and R. This book is designed to give you all you need to accomplish this goal. You will find that Kelp.Net is an invaluable tool used to create powerful and expressive deep learning models. But I know that along the way, someone might ask you what is happening behind the scenes. This is why I have also chosen to include and deeply integrate ReflectInsight into this book, so that all the output goes into this power logging application. One of the hardest questions to answer is what your deep learning models are doing behind the scenes. This book makes it easy to answer this question. Not only will you be able to easily create your own deep learning models, but you will also gain a better understanding about what is going on behind the scenes. This book is a must for every C# developer who wants to learn deep learning and is the penultimate reference guide for Kelp.Net.

---

## Acknowledgements

This book would not have been possible without the incredible support I received from my wife, Neda every day. I would also like to thank everyone at BPB Publications for giving me the opportunity to write this incredible book and for being an incredible supportive platform for authors. I would like to thank Mindy and Cocoa for their unique kind of support. I wouldn't have it any other way! And finally, a big thank you all the readers for purchasing this book and taking your first (or next) step down the road to deep learning. May your journey be a good one!

## Downloading the code bundle and colored images:

Please follow the link to download the *Code Bundle* and the *Colored Images* of the book:

**<https://rebrand.ly/50f4c>**

## Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors if any, occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

**errata@bpbonline.com**

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

---

# Table of Contents

<b>1. Take This ___ and ___ It .....</b>	<b>1</b>
Objectives of this book.....	4
Neural network overview .....	7
Machine learning overview .....	9
Deep learning overview .....	10
Complexity.....	12
<i>Machine and deep learning differences</i> .....	13
Summary.....	14
<b>2. Machine Learning/Deep Learning Terms and Concepts.....</b>	<b>15</b>
Overview .....	15
Neuron/Perceptron.....	16
<i>Multi-Layer Perceptron (MLP)</i> .....	17
Features.....	18
Weights.....	19
Bias .....	
19	
<i>Activation Function</i> .....	19
<i>Sigmoid</i> .....	21
<i>ReLU (Rectified Linear Units)</i> .....	21
<i>Softmax</i> .....	22
Neural network.....	22
<i>Input/Output/Hidden Layers</i> .....	25
<i>Forward propagation</i> .....	25
<i>Back propagation</i> .....	25
The No Free Lunch theorem .....	27
<i>The Curse of Dimensionality</i> .....	27
<i>The more neurons versus more layers</i> .....	27
<i>Cost function</i> .....	27
<i>Gradient descent</i> .....	29
<i>Learning rate</i> .....	33
<i>Batches/Batch size</i> .....	35

---

<i>Epochs</i> .....	35
<i>Iterations</i> .....	36
<i>Dropout</i> .....	36
<i>Batch Normalization</i> .....	36
CNN (Convolutional Neural Network) .....	37
<i>Pooling</i> .....	39
<i>Padding</i> .....	39
Recurrent neuron.....	40
RNN (Recurrent Neural Network) .....	41
Vanishing gradient problem .....	42
Exploding gradient problem.....	43
Logistic Neurons.....	43
<i>Hidden layers</i> .....	44
Types of neural networks .....	45
<i>Generalization</i> .....	47
<i>Regularization</i> .....	47
<i>Loss</i> .....	48
<i>Loss over time</i> .....	48
<i>Loss versus learning curve</i> .....	48
Supervised learning .....	49
<i>Bias-Variance Trade-off (overfitting and underfitting)</i> .....	50
<i>Bias</i> .....	50
<i>Variance</i> .....	50
<i>Overfitting</i> .....	50
<i>Is your model overfitting or underfitting?</i> .....	51
<i>Prevention of overfitting and underfitting</i> .....	52
<i>Amount of training data</i> .....	54
<i>Input space dimensionality</i> .....	54
<i>Incorrect output values</i> .....	54
<i>Data heterogeneity</i> .....	55
Unsupervised learning .....	55
Reinforcement learning .....	57
Manifold learning.....	57
<i>Types of manifolds in deep learning</i> .....	58
<i>Topological</i> .....	59

---

<i>Differentiable</i>	59
<i>Riemannian</i>	59
<i>Principal Component Analysis (PCA)</i>	59
<i>Hyperparameter training</i>	60
<i>Approaches to hyperparameter tuning</i>	61
<i>Grid search</i>	61
<i>Random search</i>	61
<i>Bayesian optimization</i>	61
<i>Gradient-based optimization</i>	61
<i>Evolutionary optimization</i>	61
Summary	62
References	62
<b>3. Deep Instrumentation Using ReflectInsight</b>	63
Next generation logging viewers	65
Message log	65
Message details	66
Message properties	67
Bookmarks	68
Call Stack	68
Message Navigation	69
Advanced Search	70
User-Defined Views and Filtering	71
<i>Auto Save/Purge rolling log files</i>	72
Watches	73
Time zone formatting	74
Router	74
Log viewer	75
Live viewer	75
SDK	77
Configuration editor	78
Overview	79
XML configuration	79
Dynamic configuration	79
Configuration editor	79

---

<i>Message type logging reference</i> .....	80
<i>Assertions</i> .....	80
<i>Assigned variables</i> .....	81
<i>Attachments</i> .....	81
<i>Audit failure and success</i> .....	81
<i>Checkmarks</i> .....	81
<i>Checkpoints</i> .....	82
<i>Collections</i> .....	82
<i>Comments</i> .....	83
<i>Currency</i> .....	83
<i>Data</i> .....	84
<i>DataSet</i> .....	85
<i>DataSetSchema</i> .....	85
<i>DataTable</i> .....	85
<i>DataTableSchema</i> .....	85
<i>DataView</i> .....	85
<i>Date/Time</i> .....	85
<i>Debug</i> .....	86
<i>Desktop Image</i> .....	86
<i>Errors</i> .....	86
<i>Exceptions</i> .....	87
<i>Fatal Errors</i> .....	87
<i>Generations</i> .....	88
<i>Images</i> .....	88
<i>Information</i> .....	88
<i>Levels</i> .....	88
<i>Linq queries and results</i> .....	89
<i>Loaded assemblies</i> .....	89
<i>Loaded processes</i> .....	90
<i>Memory status</i> .....	90
<i>Messages</i> .....	91
<i>Notes</i> .....	91
<i>Process Information</i> .....	92
<i>Reminders</i> .....	92
<i>Serialized Objects</i> .....	92

---

<i>SQL strings</i> .....	93
<i>Stack Traces</i> .....	93
<i>System Information</i> .....	95
<i>Text files</i> .....	95
<i>Thread Information</i> .....	96
<i>Typed collections</i> .....	97
<i>Warning</i> .....	97
<i>XML</i> .....	97
<i>XML files</i> .....	98
<i>Tracing method calls</i> .....	98
<i>Attaching message properties</i> .....	99
<i>To one request</i> .....	100
<i>To all requests</i> .....	100
<i>To a single message</i> .....	100
<i>Watches</i> .....	101
<i>Using custom data</i> .....	102
<i>Output</i> .....	104
<i>Summary</i> .....	104
<b>4. Kelp.Net Reference</b> .....	<b>105</b>
Let us be honest .....	105
Downloading Kelp.Net.....	107
<i>Building the source code</i> .....	108
What is Kelp.Net?.....	108
N-dimensional arrays .....	114
Optimizers .....	116
<i>AdaDelta</i> .....	119
<i>AdaGrad</i> .....	120
<i>Adam</i> .....	121
<i>GradientClipping</i> .....	122
<i>MomentumSGD</i> .....	124
<i>RMSprop</i> .....	125
<i>SGD</i> .....	126
Poolings.....	127
<i>MaxPooling</i> .....	128

---

<i>AveragePooling</i>	136
<i>FunctionStack</i>	141
<i>FunctionDictionary</i>	147
<i>SplitFunction</i>	154
<i>SortedList</i>	157
<i>SortedFunctionStack</i>	165
Activation Functions	171
Activation plots	172
<i>ArcSinH</i>	172
<i>ArcTan</i>	174
<i>ELU</i>	176
<i>Gaussian</i>	178
<i>LeakyReLU</i>	179
<i>LeakyReLUShifted</i>	181
<i>LogisticFunction</i>	183
<i>MaxMinusOne</i>	185
<i>PolynomialApproximantSteep</i>	187
<i>QuadraticSigmoid</i>	189
<i>RbfGaussian</i>	191
<i>ReLU</i>	193
<i>ReLuTanh</i>	194
<i>ScaledELU</i>	196
<i>Sigmoid</i>	198
<i>Sine</i>	200
<i>Softmax</i>	202
<i>Softplus</i>	204
<i>SReLU</i>	206
<i>SReLUShifted</i>	208
<i>Swish</i>	210
<i>Tanh</i>	212
Connections	213
<i>Convolution2D</i>	214
<i>Deconvolution2D</i>	228
<i>EmbedID</i>	242
<i>Linear</i>	244

---

<i>LSTM</i> .....	252
Normalization .....	261
<i>BatchNormalization</i> .....	261
<i>Local Response Normalization</i> .....	268
Noise.....	272
<i>Dropout</i> .....	273
<i>StochasticDepth</i> .....	279
Loss.....	282
<i>MeanSquaredError</i> .....	283
<i>SoftmaxCrossEntropy</i> .....	284
Datasets.....	286
<i>CIFAR-10</i> .....	286
<i>CIFAR-100</i> .....	287
<i>MNIST</i> .....	288
<i>Street View House Numbers (SVHN)</i> .....	288
Summary.....	289
References.....	290
<b>5. Model Testing and Training.....</b>	<b>291</b>
Accuracy .....	293
Timing .....	295
Common stacks.....	296
Summary.....	299
<b>6. Loading and Saving Models.....</b>	<b>301</b>
Loading models .....	304
Saving models .....	305
Model size.....	306
Summary.....	307
<b>7. Sample Deep Learning Tests .....</b>	<b>309</b>
A simple XOR problem.....	309
Complete source code.....	310
<i>Output</i> .....	312
<i>A penny for your thoughts</i> .....	313

---

<i>A simple XOR problem (part 2) .....</i>	314
<i>Complete source code.....</i>	314
<i>Output .....</i>	316
<i>Recurrent Neural Network Language Models (RNNLM) .....</i>	316
<i>Complete source code .....</i>	316
<i>Vocabulary .....</i>	321
<i>Output .....</i>	322
<i>Word prediction test.....</i>	322
<i>Complete source code .....</i>	323
<i>Output .....</i>	329
<i>Decoupled Neural Interfaces using Synthetic Gradients .....</i>	329
<i>Output .....</i>	335
<i>MNIST accuracy tester.....</i>	335
<i>Complete source code .....</i>	336
<i>Output .....</i>	339
<i>Massively Deep Network Test.....</i>	339
<i>Complete source code .....</i>	340
<i>Output .....</i>	342
<i>Image prediction test.....</i>	342
<i>Complete source code .....</i>	343
<i>Output .....</i>	344
<i>Function benchmarking.....</i>	345
<i>Output .....</i>	358
<i>MNIST (handwritten characters) learning test .....</i>	358
<i>Complete source code .....</i>	358
<i>Output .....</i>	360
<i>LeakyReLu and PolynomialApproximantSteep Combination Network.....</i>	360
<i>Complete source code .....</i>	361
<i>Output .....</i>	365
<i>FunctionStack navigation tests.....</i>	365
<i>Complete source code .....</i>	365
<i>Output .....</i>	370
<i>Learning Rate Hyperparameter tester.....</i>	370
<i>Complete source code .....</i>	370
<i>Output .....</i>	373

---

Model scoring.....	373
<i>Complete source code</i> .....	373
<i>Output</i> .....	376
Summary.....	376
<b>8. Creating Your Own Deep Learning Tests.....</b>	<b>377</b>
<i>Example</i> .....	377
Implementing the Run function.....	378
<i>Create a FunctionStack with your functions.</i> .....	378
<i>Set the optimizer</i> .....	379
<i>Make your predictions</i> .....	379
<i>Save the model</i> .....	380
<i>Loading models</i> .....	380
Summary.....	380
Thank You.....	381
<b>Appendix A .....</b>	<b>383</b>
Evaluation metrics.....	383
Metrics terminology .....	383
Confusion matrix.....	385
<b>Appendix B.....</b>	<b>387</b>
OpenCL.....	387
OpenCL hierarchy .....	388

# CHAPTER 1

## Take This \_\_\_\_\_ and \_\_\_\_\_ It

Even though we are meeting for the first time, I can guess the two words that you have in mind to fill in the blanks for this sentence. How can that be? And not only that, I bet that it is not just you and I that have picked the same two words. The chances are high that many of us have. How could it be that the chances of all of us picking the same two words without thinking about it are so very high? I am sure we can all recall hearing this phrase somewhere in the past (and maybe one or two of us said it...), but is the pattern recognition in our brains so good that these two words immediately were the ones that came to mind when we saw a partial implementation of that sentence? And how could so many of us, seemingly unconnected from each other, arrive at the same two words and complete the sentence in the same way in probably close to the same amount of time? Did we just witness a marvelous example of unsupervised deep learning at its finest or was it something else? How can such advanced pattern recognition ability happen so fast and be so predictable among a huge population of people? Or maybe it was reinforcement learning that brought this to the mind (Lord knows I have received negative reinforcement for saying it!). Regardless of how, something marvelous surely has just happened.

Let us think about this for a second, shall we? We have an area in our brains called the hippocampus and this area stores episodic memories (stuff that happened to you and me). Within the hippocampus, many neurons just fired when we saw that sentence, and the exciting neurons which they were connected to and so on, which made them fire until the pattern was complete. And all this happened in the blink

of an eye. But did any of us have to give any thought into having to complete that sentence, or did it seem to pop up from our subconscious mind with the greatest of ease? The truth is that we may never know, at least not fully, but theories are abounded.

So how many of you work in the ‘real world’ as developers? For those who do, you know more often than not we walk through the doors, put our fireman’s hat on, and off we go. How we long for some pure R & D time where we can work on our code, create better algorithms, enhance the UI we have been putting off for what seems like forever. Ah! To be like the academic world for even the smallest amount of time! But alas, we are in the real world! We have products to deliver and timelines to meet. Some of us work around an agile approach to development, some do not. Everyone seems to be doing the same things, albeit a little bit differently.

With this in mind, let us start with a short story. I know there are some of you out there for whom this will hit home for sure. As we are embarking on a path down the deep learning lane, for those of you who are developers coming from the academic world, it is just a little bit different out here. But since we are talking about developing deep learning applications, it is time both the sides of the spectrum (developers and the academic world) meet! After all, we are about to start developing deep learning applications together and deliver them to our customers, so sit back and enjoy the early morning meeting. We will no doubt have many more like them!

Early one morning, I walked into my boss’s office for our daily morning meeting and could not help but notice that this time the room was full of people I had not seen before. No doubt! This was going to be a very important meeting I thought to myself, so I quickly took my seat, smiled and made the courtesy introductions while I awaited the start of the meeting. Today, I was happy because I was going to use Kelp.Net to start a new project. Our development team has little to no deep learning experience per se, neither on the academic nor the production side, so using Kelp. Net will allow us to make rapid progress in our project, something the boss would like to see. Until this point, he was a bit reluctant to embark on machine learning for our projects; he believed there was more hype than reality to a lot of it, and I could see his point.

Like most development shops, we do not write our production level code in R or Python; we are a Microsoft shop heavy on the C# .Net side, with some occasional Visual Basic and Java thrown in. Your chances of getting anything non-Microsoft related from our IT shop is somewhere between slim and none, which is no different from many places I am sure.

The boss began the meeting by saying, “We have been told from the upper management that we need to keep up with the times, so we will machine-learn our next project.” Nice, did not see that one coming. He continued with the introduction of the new *data science* team, which he told us has been put together to ensure that

the company is headed in the right direction. We were also told how the company was 100% behind this initiative of integrating AI, machine learning and deep learning technologies in our daily lives, starting with this project, and how all of us need to do our part to ensure its success. Then, the data science team began to speak.

The first words out of their mouths were **R** and **Python**. The boss slowly nodded his head in agreement. He thought he could not even get the latest version of Visual Studio from the support team without everyone on the planet needing to approve the request, and suddenly we are going to switch programming languages?

The data science team completed their presentation and the meeting disbanded for the day. Everyone looked at each other as they so often did, and we walked out of the office. My mind was now in an alternate universe with many questions swirling around in my head. I was thinking to myself:

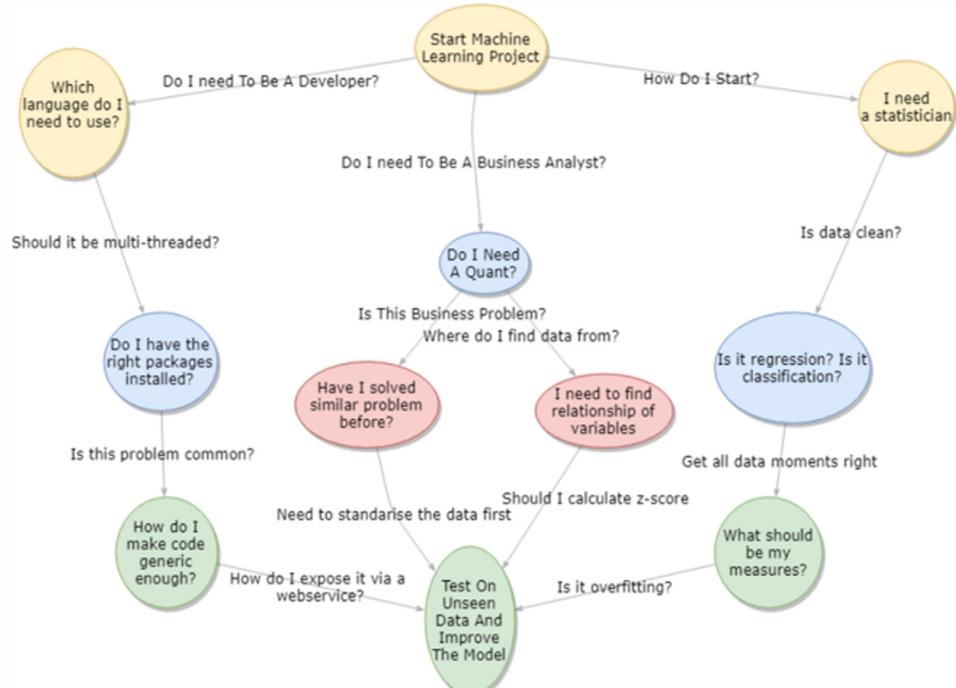
1. My boss wanted to use machine learning to solve problems, but of course, none of our projects have really good problem definitions.
2. My boss, as well as the rest of the senior management, has been told how simple yet important deep learning is. We cannot let our competitors go ahead of us. "Just a combination of massive amount of data, cloud technologies and coding," he said. How hard is it in my corporate world to request even the smallest amount of data from an external owner, yet now I need access to reams and reams of it across multiple owners? Hopefully, he and others will understand that to do this properly, we will need to spend a considerable amount of time locating, cleaning and preparing the data to be useful for this project, and that is before we even start coding! First, however, is getting the permissions from each data silo owner to access the data.
3. Once I do get access to the 75 petabytes of data the company has been storing for 15 years, how in the world am I going to fit it into this wonderful, third hand-me-down laptop that IT support has given me? 2 GB RAM was the standard 10 years ago, but not really sure if the machine learning objective is going to work out so well like this. I will surely have a lot of time to start drinking coffee!
4. I tried and let our data scientist team know why we might need to consider some alternative routes, but of course that explanation fell on deaf ears because they all have PhDs!
5. Our models do and will fail while we are perfecting them, and that is an iterative part of how we make things work. This was going to be fun when confronted with why is *it* failing and trying to explain *it* to the upper management! I was hoping that the new data science team would be there to help explain to my boss why model failure is a part of the process, but they were insisting that we rewrite everything in R or Python because that is what

real deep learning projects use.

If this sounds like a day in your development shop, in whole or in part, welcome to the crowd! This short story was meant to illustrate some of what happens when we take things from the academic world and become *buzzword-compliant* in the corporate world. Real-world software development and academia are many light years apart in things we encounter and how we must perform daily, especially when it comes to machine and deep learning. Most of us long for the pristine world of academia or even research and development. For most of us, we put the fireman's hat on the minute we walk through the doors and off we go.

I am happy to present you two very powerful and flexible tools: Kelp.Net and ReflectInsight. I will show you a great framework that provides power and flexibility for C# .Net developers, while doing a great job of bridging the corporate and academic worlds. I will show you how to use the software, as well as give you a high-level overview of machine and deep learning to help you to be able to have meaningful conversations with that new team of data scientists. For those concerned as to whether the software may meet your needs, as Kelp.Net is open source, let me tell you that you and your team will have complete control over customizing it to handle any situation you may encounter.

## Objectives of this book



In this book, we will discuss how you, the C# developer, can add amazing and powerful deep learning techniques to your applications. We will do so by presenting the powerful, flexible and extendable open source software package known as Kelp.Net. We will also present ReflectInsight, an incredibly powerful, flexible and robust logging framework which will prove to be invaluable when debugging your machine learning models and processes.

**INFO:** *Kelp.Net – Original Japanese version from Twitter: <https://twitter.com/harujoj>*

**English version by Matt R. Cole**

**ReflectInsight – ReflectSoftware**

While ReflectInsight is not open source, the value that it adds to redefine real-time logging makes it an easy choice for any project where we need deep instrumentation and real-time logging.

But perhaps even more important than all of that, you will have an open source software package in Kelp.Net which you can enhance and embellish to create new and exciting versions with incredibly powerful features. You will also have an incredibly powerful logging framework which is of immense importance when it comes to debugging your models and algorithms. For both, I have created what I hope will be your de facto standard reference manuals to use and to help you understand what each tool has to offer individually as well as in tandem.

We will start with a quick chat about neural networks, machine and deep learning. We will talk more about some terms and concepts you need to be familiar with and some of the similarities and the differences between the two. Since our focus is on deep learning, we will place great emphasis on it. Next, we will talk about logging, the importance of logging, and what ReflectInsight can do for you. With all this in place, the stage will be set for us to deep dive into Kelp.Net and talk about its terminology, what it means, and how it can be used. In our last few sections, we will discuss about Kelp.Net deep learning models, how you can train and test them, and how you can write model tests yourself.

Let us be honest. Deep learning is a rage nowadays, isn't it? It is the latest in a series of buzzwords being used across the corporate world and is on every billboard and advertisement across the land. We are flooded with targeted advertisements about machine learning and AI on our desktops every time we do a search for something. What used to be a pristine environment within academia has now escaped and been embraced by the corporate world. And what a difference that has made. Luckily for us, our increasing and unfillable desire to accumulate more and more data to make new innovations in deep learning seems more probable with every passing day. The more data we have access to, the more we can learn and improve our accuracy and quest for knowledge from deep learning models.

Even though we all know that the topic of deep learning is being used everywhere, did you ever stop to wonder what does it really mean? Can deep learning really make your life easy, or will it be more work? Will I be able to spend more time at home with my family while the ‘machines’ crunch away and figure things out for me? Will this great technology make my job obsolete in the coming years and put me out of a job? All are great questions, and the aim of this book is to help you answer these and many more (ok, not the out of a job one, let us just hope that never happens!).

There are many areas in which machine and deep learning can help improve our everyday lives. For those of you who may not have any experience or education in deep learning, or for those wanting a quick refresher, we are going to spend a few moments at the beginning of this book going through some basic knowledge which may prove helpful later. This knowledge will lay the groundwork for some future discussions. If you are already well versed in this, then please feel free to browse to the part(s) of the book that may interest you the most. In any event, make sure we are all on the same page when it comes to terminology as this will help eliminate confusion later.

Let me be clear about one thing right from the start about the approach of this book. While we will provide you some basic ideas and information on machine and deep learning, my main goal is to show you how to use the Kelp.Net and ReflectInsight frameworks to add the machine and deep learning functionality to your applications. This book is not meant to be a reference or a definitive guide on deep learning; there are far too many of those out there already. This is a much focused application of technology, which is meant to get you excited and start contributing to the world of deep learning. I hope I have put together the most detailed reference on this software that you can find anywhere, as well as provided you with an updated version of Kelp.Net I have made specifically for this book. It is my goal to expose you to an intuitive and powerful way of using this technology in your applications, while providing you with the ability to embellish this and make the world your table!

Let us begin our journey by stating a few of our definitions:

**Machine learning:** “Any algorithm which parses data, learns from that data, and then applies what it learned to make informed decisions regarding that data.”

**Deep learning:** “Any machine learning algorithm or task which utilizes more than one hidden layer to accomplish its task towards finding patterns in data with minimal supervision.”

**Neural network:** “A neural network, in its simplest form, is a system comprised of several simple but highly interconnected elements; of which, each processes information based on their response to external inputs.”

## Neural network overview

A neural network is a computer system which is modeled after the human brain and nervous system. This modeling occurs based on the limited knowledge we have of the brain, the nervous system, and how they interact. Let me stress on the word *limited* here. Over the years we have made amazing strides considering that we still have very limited knowledge of the brain. Therefore, a lot of what is discussed may, and sometimes does, change from time to time. No doubt! Tremendous knowledge will be gained between the time of writing of this book and the time you are reading it now. We learn, we improve (hopefully), we learn some more, and on it goes. Please keep an open mind on this topic as many ‘thought leaders’ do who continue to refine our path forward.

Did you ever stop and take time to wonder how a human can do some things so effortlessly and a computer needs massive hardware and data and still not get it right? To me that effortlessness is the key to our next generation of deep learning and neural network development. How can it be that we humans just seem to know things already, without requiring all that massive data and knowledge? Could it be that portions of our brains (our internal neural networks) are already pre-programmed with information that can be easily accessed in future references? Is this information stored in the conscious or subconscious parts of our memory, or if it is not in the memory at all, how are we able to do all the calculations required to achieve our results in what seems like no time at all? Or could it be that the mirror and canonical neurons hold the key that unlocks the next generation of deep learning and neural networks? We will continue to ask ourselves all these good questions.

Let us go off on another tangent for a moment (are you not glad that I am getting this ‘tangent’ thing out of my system now rather than further in the book?) Look at the following sequence of handwritten digits for a moment:

504192

Most people will recognize the fact that these are digits and will do so without any effort. Pretty straightforward you might say. As children, we were taught how to identify numbers, collectively and individually as well as how to identify them. We were taught how to read single numbers as well as groups of numbers put together to create larger ones. Somewhere in the past our brains have been trained with massive datasets which we will now use to quickly and accurately identify the digits given above. This information is stored as memories in our subconscious system for lightning fast retrieval.

Humans have a visual cortex in their brains, which contains millions of neurons. These neurons each inspire billions of connections. Also remember that we have several visual cortices in different brain hemispheres, so if you look at that as a super highway of connectivity, our raw brainpower is truly mindboggling. Somewhere in that interconnection of elements lies the knowledge that we see five digits on our screen. Our goal is to now be able to accomplish this artificially via the computer, just as fast and as accurate as we do it ourselves. We have only (relatively) just entered the age of deep learning and expanded it beyond using a single layer of neurons so how far do we have to go until our goal is a reality. In some ways we have made incredible progress, but in others, we still have a long way to go if our goal is to truly mirror the functionality of the human brain.

So, back to the neurons and stored memory for a second. Could it be that the neurons in our brains have adapted and have been performance-tuned over hundreds of years or more? Did we just now learn all this vast knowledge, or did we inherit it somehow based on the information that humans have learned over the centuries? Did we learn or inherit this information ourselves or did we get some or all from a hereditary mechanism? Do our brains come pre-filled with memories and signals which humans have obtained and processed over the years? A lot of questions for sure.

Now, let us talk about image processing and advanced algorithms. The human brain and visual system are monster performers at image recognition and much faster than any computer system. But why is this sort of task more difficult for an artificial neural network? Is it due to limitations with the data, the hardware, or is the limitation in our limits of knowledge from a software development perspective?

Apart from recognizing and identifying numerical digits, harnessing that brainpower to do things like image processing is even more impressive, and as you might expect, more difficult for the computer. Yet for the human, it seems that processing images is done almost effortlessly. Think about how you recognize the digit **9**, and then think about what a neural network would have to account for and consider accomplishing the same thing. Is the digit a **9**, is it an **8**, a **0**, or is it some digit that is grossly malformed? The list of exceptions and caveats goes on and on. Yet, in the blink of an eye, we as humans have processed this image and have our answer.

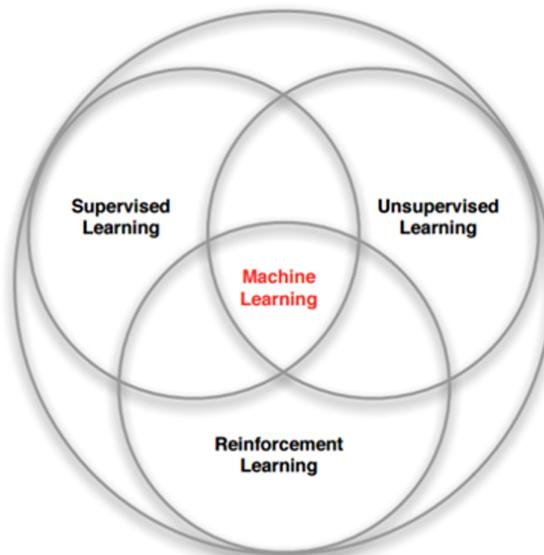
Think about how we can quickly and accurately recognize various images such as road signs, automobiles, and more. How are our brains compared to the self-driving cars coming of age? A lot of questions, some with and some without answers.

Let us continue with our digits and talk about how a neural network would approach such a problem. The idea would be to take a large sampling of handwritten digits with each digit having multiple samples, each done a bit differently. This large sample is known as our *training* examples and might resemble to something like this:

0	4	1	9	2	1	3	1	4	3
5	3	6	1	7	2	8	6	9	4
0	9	1	1	2	4	3	2	7	3
8	6	9	0	5	6	0	7	6	1
8	7	9	3	9	8	5	9	3	3
0	7	4	9	8	0	9	4	1	4
4	6	0	4	5	6	1	0	0	1
7	1	6	3	0	2	1	1	7	8
0	2	6	7	8	3	9	0	4	6
7	4	6	8	0	7	8	3	1	5

From this sample set, we would then need to develop an algorithm that is able to learn from the training examples. By increasing the number of training examples available to our algorithm, the algorithm can learn even more about handwriting and improve its conclusions and accuracy. This is a high-level example of a neural network and the implementation of this example is machine learning.

## Machine learning overview



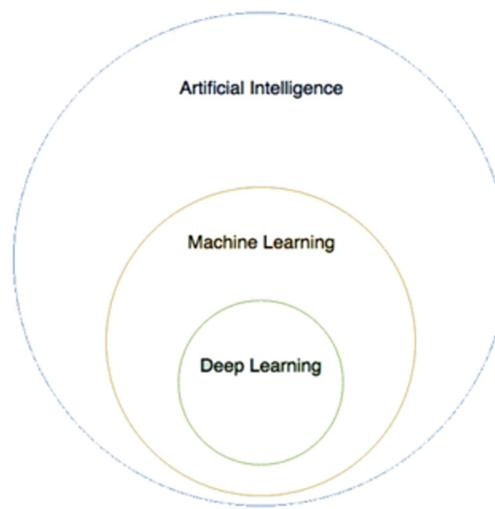
Today's society is *buzzword compliant*, no doubt about it. Buzzwords abound everywhere we go, and machine learning and deep learning are no exception. Let us try and put some definitions to these terms so that it is clear what we mean when discussing each of them.

Machine learning (ML) is the sub field of this big, broad sweeping category known as **Artificial Intelligence**. Machine learning gives machines the ability to improve their performance over time, without explicit intervention or help from a human being. Most of the current applications of the machine learning leverage what is known as **Supervised Learning**.

When we implement machine learning, we present thousands or millions of examples of *things* to our computer and show what each one means or is *labeled*. This, in turn, correctly teaches the computer how to solve our problem. For example, using the historical fraud data, we can train an algorithm to identify a fraudulent activity from a non-fraudulent activity. Once the machine learns how to correctly classify the cases, we can deploy the model for future usage.

Other usages of ML can be broadly classified as either **Unsupervised Learning** or **Reinforced Learning**. In unsupervised learning, there is no label or output which is used to train the machine as to what is correct. The machine is trained to correctly identify hidden patterns or segments. Reinforced learning, on the other hand, focuses on a constantly learning system which incentivizes an algorithm for meeting the final goals under the given constraints.

## Deep learning overview



In practical terms, deep learning is a subset of machine learning, which is a subset of artificial intelligence; hence, why the terms are used interchangeably. Deep learning is machine learning but with different capabilities. Deep learning typically has more learning *layers* than other types of algorithms and these layers are called **hidden layers**.

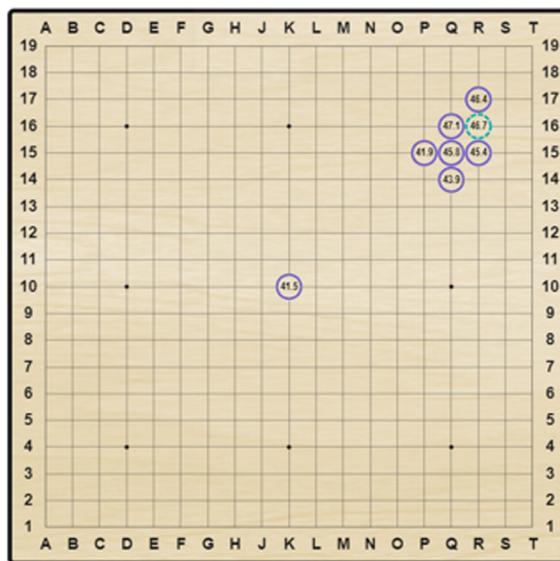
Machine learning usually requires some form of guidance, especially if it returns an inaccurate prediction. A person then needs to intervene and make the correct adjustments. But with deep learning, generally, the algorithms themselves can determine if a prediction is accurate or not.

Let us focus on deep learning for a minute. How does deep learning work you might ask yourself. To start with, we cannot talk about deep learning without talking about data. For a deep learning algorithm to be effective, a lot of data must be available for it to use. The more the better. In fact, if you do not have enough data, your deep learning efforts are more than likely going to be unsuccessful or inaccurate. The more data we have, the more we can analyze and learn from.

If you can imagine for a moment a rocket carrying a satellite into the orbit, then the rocket engine would be our deep learning model, and the massive amount of fuel stored on-board and in the boosters would be the huge amount of data we would feed our algorithm.

The deep learning process will rely on the availability of this data and continually analyze it to make an attempt to draw conclusions based on that analysis. The process 'learns' by using a 'layered' set of algorithms which are commonly referred to as an artificial neural network and the layers themselves are called hidden layers. The design of this network is inspired by the human brain, which is the most advanced neural network on the planet (that we know of!). This configuration and capability is what makes a deep learning system outperform, in terms of capacity, standard machine learning models.

A great example of deep learning that you may have heard of is the famous Google AlphaGo project. Google created this computer program to play the game Go, which is known for requiring very sharp intellect and intuition. By playing against the other Go players, AlphaGo's deep learning model learned how to play at the highest level ever seen from an artificial intelligence perspective, and it did this without being told when to make any specific moves. The following is a screenshot of AlphaGo showing an example of the percentages it calculates for what its next move should be:



The game of Go

## Complexity

No discussion of machine and deep learning would be complete without touching on the subject of complexity. This ties into ‘overfitting’ of data, which we will cover in more detail in Chapter 3. So let us talk about model complexity.

In your data, you have ‘features’ that you can envision as columns in your database or spreadsheet. In some deep learning models, you have many, many features. In some cases, you can have more features than your data that can have examples. If you specify or use all the features available, your algorithm will start to memorize everything in the data, including random noise, errors and characteristics that may not be truly important.

When this memorization happens, you are happy because you have the illusion that your algorithm is working perfectly because it appears, at an initial glance, to be fitting the data so well. Once you apply your algorithm to your test data, you will notice the problems.

We will talk more about how to address complexity in *Chapter 3, Deep Instrumentation using ReflectInsight*. For now, we need to simply know that just because your dataset has thousands of features more than likely, you will not be using them all, or you will spend a large portion of your time working on an overfitting problem.

## Machine and deep learning differences

Let us talk for a moment about some of the differences between machine learning and deep learning. While machine learning models are progressively better at whatever their functions are, the truth is that they still sometimes may need a little bit of hand holding. If a machine learning algorithm predicts something incorrectly, someone usually needs to step in and adjust the process. With deep learning, however, the algorithms themselves are designed such that they can determine on their own whether that prediction is accurate, without any outside help or intervention.

So, in fact, the hierarchy which we have here is that deep learning is a subset of machine learning, and machine learning is a subset of the broader category of artificial intelligence.

Here are a few differences between machine learning and deep learning that should be highlighted:

	Machine learning	Deep learning
How it works	Uses automated algorithms that learn to predict future decisions and model functions using the data which is provided.	Interprets features in data and the relationships using neural networks which pass the data through several layers of the algorithm.
Intervention	Algorithms usually require human interventions to examine different variables and dataset features.	Algorithms require no human intervention for data analysis.
Data points	A few hundred to a few thousand.	Can be into the millions.
Output	A numerical value such as a score or classification.	Could possibly be anything?
Hardware	Requires less hardware capacity than deep learning.	Requires high-end hardware capabilities such as GPUs to perform at its best.

	Machine learning	Deep learning
Feature extraction	Requires features to be identified.	Will look to determine features from patterns in data.
Training time	Training time is less.	Training time is more.

Let us do a recap:

- Machine learning uses algorithms to parse data, learn from that data, and then make informed decisions based on what it has learned.
- Deep learning is a subset of machine learning. It organizes algorithms in layers to create an artificial neural network that can learn and make intelligent decisions without any outside intervention and by finding patterns in the data provided.
- Deep learning uses massive amount of data to learn. With the phenomenal increase in the amount of data we collect, in the very near future we hope that deep learning will be able to provide new opportunities and innovations.

## Summary

In this chapter, we talked about some basic concepts regarding neural networks, machine and deep learning. We discussed many terms that you will see been used later in this book. We also highlighted some of the differences between machine and deep learning.

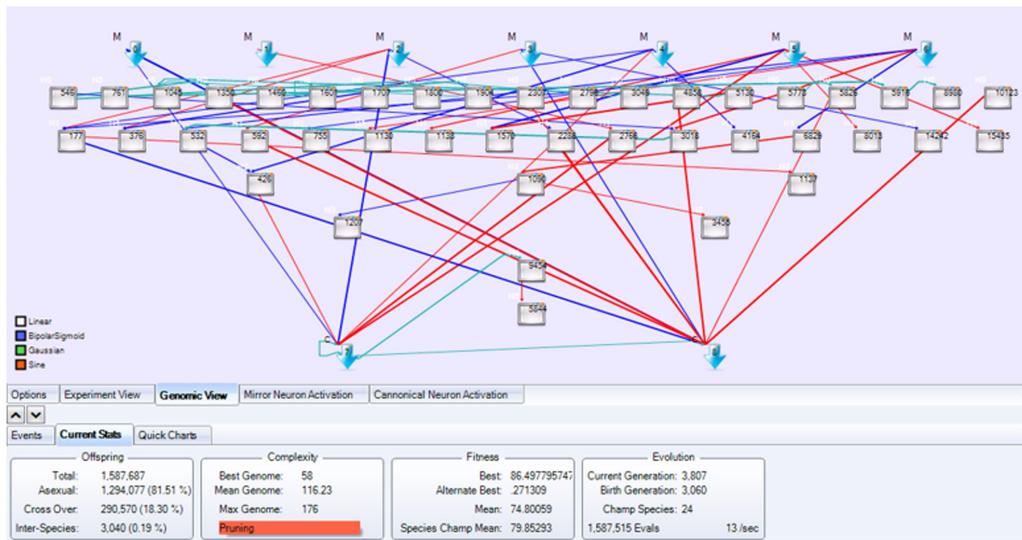
Now, let us move on and talk about the importance of logging within a deep learning framework. With our approach to logging (using ReflectInsight), you will be able to quickly find out what is happening within your applications. Hold on to your hats!!!

# CHAPTER 2

# Machine Learning/ Deep Learning Terms and Concepts

## Overview

In this chapter, we will briefly go through some of the terms and concepts that we encounter when talking about neural networks, machine and deep learning. Let me tell you up front that I will do my best to shield you from this once we get used to using Kelp.Net, but the underlying theory, terms and concepts is important for everyone to know. Anyone can learn to use a toolkit, framework or API. Differentiate yourself by learning what goes on under the hood as well! As you can see from the image given below, there is a lot of work in a neural network and although it looks somewhat cluttered, this image is a great abstrational view to explain the terms and definitions:



If you are already familiar with all the terms and concepts in this section, or simply want to get it right into the software, feel free to skip ahead to the upcoming chapters. The information in this chapter is not presented in any order of importance. Even if you are already an expert in this subject matter, it might be good for you to browse through the sections to make sure we are using the same definitions.

*Courtesy @EVOLVE Deep Learning Studio (Matt R. Cole)*

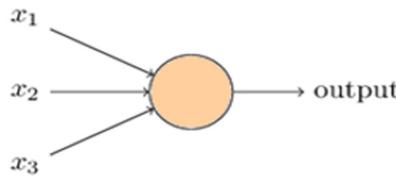
Let us kick things off by discussing all those square boxes in the image: Neurons and Perceptrons.

## Neuron/Perceptron

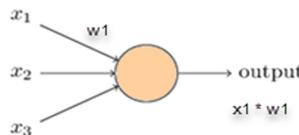
**Note:** For the purposes of this book, the terms *neuron* and *perceptrons* are used interchangeably.

Let me first start by saying that back in the days where neural networks began, the term *perceptron* was what was being used. As neural networks and computational neuroscience evolved, the term *neuron* replaced *perceptron* in neural network discussions as it more closely modeled the human brain.

Just like how a neuron forms the basic element of our brain, a neuron forms the basic structure of a neural network. A neuron receives an input, processes it and then generates an output which is then either sent to the other neurons for further processing or becomes the final output itself. The following image depicts a simple neuron which has three inputs and a single output:



As shown in the preceding image, our neuron has three inputs,  $x_1$ ,  $x_2$ , and  $x_3$ . The output is computed by introducing something called **weights**, which are real numbers expressing the importance of the respective inputs to the output.  $w_1$  is the additional weight shown in the example as follows:



The neuron's output is determined by whether the weighted sum is less than or greater than some threshold value. Just like the weights, the threshold is a real number which is a parameter of the neuron.

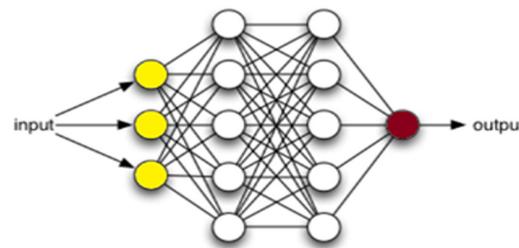
Think of a neuron like this: it's a device that makes decisions by weighing up data. The neurons in the first layer make very simple decisions by simply weighing the input data which is provided. Neurons in the second layer make decisions by weighing up the results from the first layer of decision-making. In this way, a neuron in the second layer can decide at a more complex and abstract level than the neurons in the first layer. And even more complex decisions can be made by the neurons in the third layer, and so on and so forth. In this way, a multi-layer network of neurons can engage in some very sophisticated decision making, hence the term **deep learning**.

We can also devise learning algorithms which can automatically tune the weights and biases of a network of artificial neurons. This tuning happens in response to external stimuli, without the direct intervention by a programmer. These learning algorithms enable us to use artificial neurons in a way that is radically different to conventional *logic gates*. Instead of explicitly laying out a circuit of NAND and other gates, our neural networks can simply learn to solve problems; sometimes problems where it would be extremely difficult to directly design a conventional circuit. We will see a test designed to replicate the basic XOR functionality a bit later on.

## Multi-Layer Perceptron (MLP)

A single neuron obviously would be limited in the complexity of tasks that it could perform. Therefore, we need to utilize *stacks* or *groups* of neurons in order to analyze more complex data and mirror more complex processes. In the simplest form of this

type of network, we can have an input layer, a hidden layer and an output layer. Each layer has multiple neurons, and all the neurons in each layer are connected to all the neurons in the next layer. These networks can also be called **fully-connected** networks. The following figure shows examples of a multi-layer perceptron:



## Features

One of the things you will need to deal a lot with is the features or different characteristics of the datasets you are using. It is very important for us to gather and identify these features. If the features are very few, our data might not be informative enough to get the results we need. Too many features will cause problems as well. It is important that we do not just blindly feed the network everything we know about the dataset. We must be careful and somewhat selective about the features we want to use and why. Feature processing and selection is a discipline in and of itself with its own best practices.

When it comes to feature selection, proper domain knowledge is a must. You must understand the problem domain in order to know what is and what is not important. Additionally, we do not want the entire feature set to be presented in our model. Fewer features are desirable because of the following points:

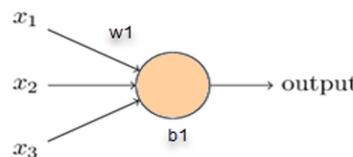
- It reduces the complexity of the model.
- It is faster and more cost effective.
- A simpler model is simpler to understand and explain.
- It makes the machine learning algorithm faster to train.
- It improves the accuracy of a model if the right subset is chosen.
- It reduces overfitting.

**A word of caution: Should you do feature selection on a different dataset than you train on (which is a big no-no; you can introduce bias in your models).**

## Weights

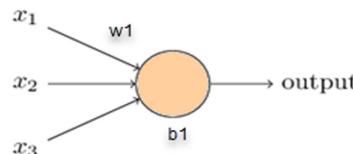
When an input enters the neuron, it is multiplied by what is known as a weight factor. This weight factor can range in values, depending on its purpose. The number of weights should be equal to the number of inputs to the neuron. In the following figure, there are three inputs so we will have three weights assigned to it. Weights are initialized randomly and then updated during the model training process. After the training, the neural network assigns a higher weight to the input that it considers the most important. A weight of zero denotes that the feature is insignificant and should not be considered in downstream layers.

Let us take a look at the following example. Assume that the input is  $x_1$  and the weight associated is  $w_1$  as shown in the following image. After passing through the node, the input now becomes  $x_1 * w_1$  as follows:



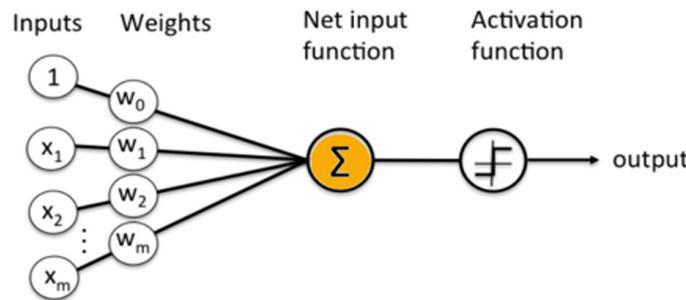
## Bias

In addition to the weights, another component that is applied to the input is called the bias ( $b_1$  as follows). Bias is added to the result of the multiplication of weight we just discussed and is provided to the input. The bias is basically added to change the range of the weight multiplied input. After adding the bias to the neuron depicted in the following figure, the result will look like  $x_1 * w_1 + b_1$ :



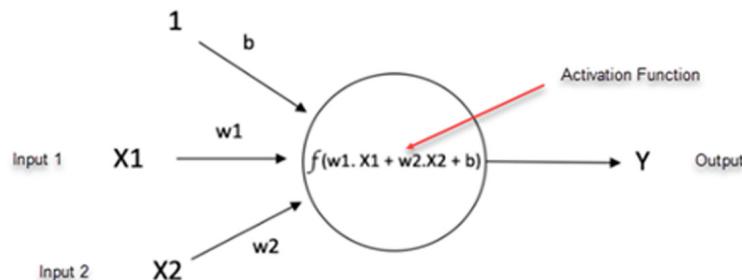
## Activation Function

Once weights and biases are applied to the neuron input, a nonlinear function is applied. This is done by applying what is known as the activation function to the combination of inputs, weights and bias. In the following diagram, you can see where the activation function lies visually in the process direction for the neuron. This diagram, in fact, shows what a complete process layer of a neuron looks like:



In a nutshell, the activation function translates the input signals into output signals. The output for layer 1, after application of the activation function shown in the neuron above will look something like  $f(x_1 \cdot w_1 + b_1)$ , where  $f()$  is the activation function itself.

In the following diagram, we have two inputs,  $X_1$  and  $X_2$ , corresponding weights,  $W_1$  and  $W_2$ , and bias given as  $b$ . The weights are multiplied to their corresponding inputs and then added together with the bias. Once the activation function is applied, we receive the final output from the neuron:

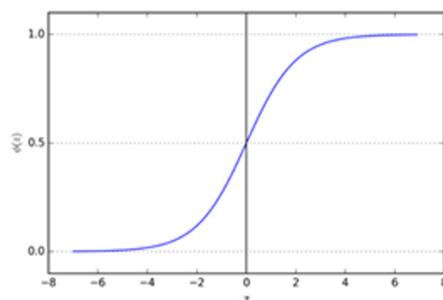


There are many activation functions which you may encounter. We will cover three of the most commonly applied activation functions in this section. They are in no particular order of importance:

- Sigmoid
- ReLU
- Softmax

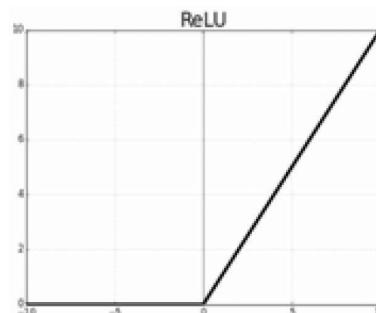
## Sigmoid

The sigmoid transformation generates a smooth range of values between 0 and 1 inclusive. Smooth curves allow us to observe changes in the output due to ever so slight changes in the input. A small change in the weights and bias will cause a small change in the output. Sigmoid functions are preferred over step functions for only this reason. The following graph shows what a sigmoid function will look like if plotted:



## ReLU (Rectified Linear Units)

**Rectified Linear Unit** activation functions (**ReLU**) are more preferred over sigmoids. You will see many of these used in Kelp.Net once we move ahead to the chapter on *Samples*. The major benefit of using a ReLU is that it has a constant derivative value for all inputs which are greater than 0. The constant derivative value helps the network to train faster, which can be of a huge value as the network becomes larger and more complicated. In our section on testing our models, you will see that quite a few of our deep learning networks are using ReLU units. The following diagram shows what a ReLU unit will look like when plotted:



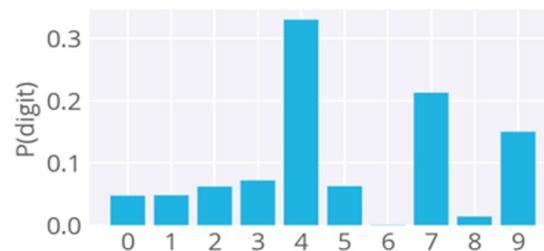
## Softmax

Softmax activation functions are normally used in the output layer and mostly for classification problems. It is similar to the sigmoid function; the only difference being that the outputs are normalized to sum up to 1. In the case of a multiclass classification problem, Softmax makes it very easy to assign values to each class which can then be easily interpreted as probabilities.

Let us look at an example to make this a little easy to understand. Suppose if we need to identify the number 4 from a series of trained digits. This number, depending on how it is written, could also look a bit like the number 7 or 9 as seen in the following image:



The function will assign probability values to each number bucket that it thinks matches the digit in its order of probability. Once complete, this is how the number bucket appears. As you can see, the majority consensus is that the number is a 4:



## Neural network

Neural Networks form the backbone of deep learning. Formally, the goal of a neural network is to find an approximation of an unknown function. A neural network is formed by a series of interconnected neurons. If you remember what we just discussed in the above sections, these neurons have weights and bias which are updated during the network training, depending on the error. The activation function puts a transformation to the weight-bias calculation, which then generates

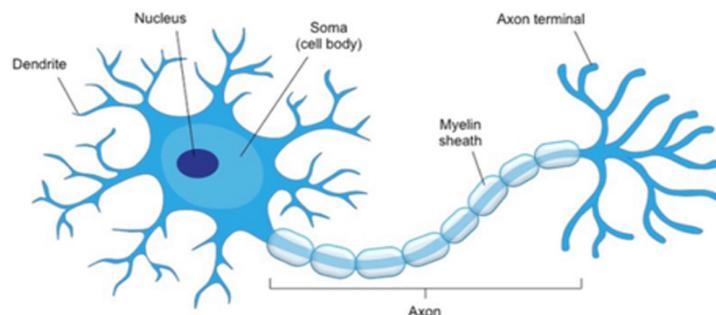
the final output. The combinations of the activated neurons give the output.

One of the best definitions of a neural network is credited to Liping Yang, and it is worth quoting here:

*"Neural networks are made up of numerous interconnected conceptualized artificial neurons, which pass data between themselves, and which have associated weights which are tuned based upon the network's "experience." Neurons have activation thresholds which, if met by a combination of their associated weights and data passed to them, are fired; combinations of fired neurons result in learning."*

I want to take a moment here now that we have discussed neurons and a neural network to compare it to our brain. You see the whole purpose of a neural (or artificial) network was to mimic what we know about the brain. While the biological component (science) in this area has progressed, so has the computational component (algorithms (**RNN's, GANs, GRUs**)) as well as different forms of neural networks available. For me, the biological component of neural networks is very important. For some others, the advancement of their algorithm may be more important than ensuring it remains in line with how the brain works. We must also be honest in saying that while our knowledge base has increased when it comes to the brain, there still is a lot we need to learn.

Let us consider a brain's neuron as shown here:



It consists of the following three main components:

- **Dendrites:** This is the input layer of our neural networks. Dendrites are tree-like structures which receive input through *synaptic connections*. This input can be a sensory input, or it can be a computational input. What we know is that a single neuron can have as many as 100,000 different inputs, and each input can be from a different neuron.

- Soma: This is the hidden layer of our neural networks, and this is where the calculations happen. Inputs from the dendrites come together and based on all the signals, a decision is made whether to fire an output (create a *spike*) or not. Truth be told; calculations also do happen in other areas.
- Axon: This is our output layer. Once a decision is made to fire an output signal, making the cell *active*, the axon is what carries that signal. Through a tree-like structure, it delivers this signal to the dendrites of the next layer of neurons via a synaptic connection.

So, you can see that the basic neurological foundation was there with the neural network design, but as time moved forward, more thought and emphasis had been given to the algorithmic component than the biological one. Also, we learn more through faster experimentation with our neural networks than we do through biological experiments and trials on the brain. Just ask any developer where their focus is and you will see my point. Let us talk about some of the differences between the biological component of our neural network and what we know about brain neurons.

First, the complexity and robustness of brain neurons is way more advanced and powerful than that of our computational neurons. Compared to our neural networks, the brain has many more neurons and dendritic connections.

Second, our implementation of neurons is much less complex than the brain. Even though we have a great computational foundation, parameters, weights and linear and activation functions that we do are very basic and crude compared to what the brain can do, and that is just the part we know about.

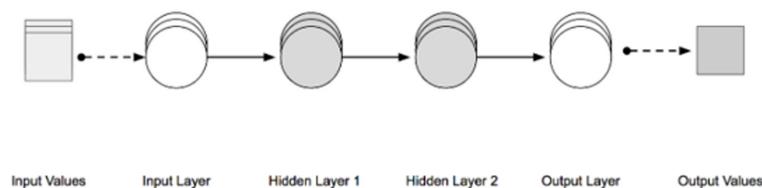
Finally, the overall network architecture of brain neurons is far more complex than our neural networks, especially the feed forward network, multi-layered recurrent neural networks and others. The network of neurons in the brain is incredibly complex with tens of thousands of dendrites crossing layers and regions in the brain in many directions.

Unsupervised and reinforcement learning is the secret sauce of AI in my opinion. We know that our brain learns through unsupervised and reinforcement learning but the current machine learning applications use supervised learning. Advancements in this area will be critical in the coming years I believe.

There is still so much we need to learn, and we will. If you look at a book on neural networks written 10 years ago and compare, you will see what I am talking about. 10 years from now, I suspect you will be able to do the same with this book.

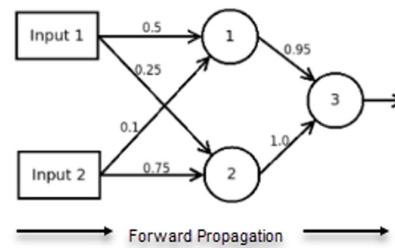
## Input/Output/Hidden Layers

Neural networks are built by connecting layers of functions. There are three types of layers present within a neural network. They are the input layer, which is the one that receives the input and is essentially the first layer of the network. The output layer is the one which generates the output and is the final layer of the network. The processing layers in between are the hidden layers within the network, which are the layers which perform specific tasks on the incoming data and pass on the output generated by them to the next layer. The input and output layers are the ones visible to us, while the **hidden layers**, obviously, are not:



## Forward propagation

Forward propagation refers to the path of movement of the input data through the hidden layers to the output layers. In forward propagation, the information travels in a single direction, which is FORWARD only. The input layer supplies the input to the hidden layers and then the output is generated from there. There is no backward movement of data. The following diagram illustrates this concept:



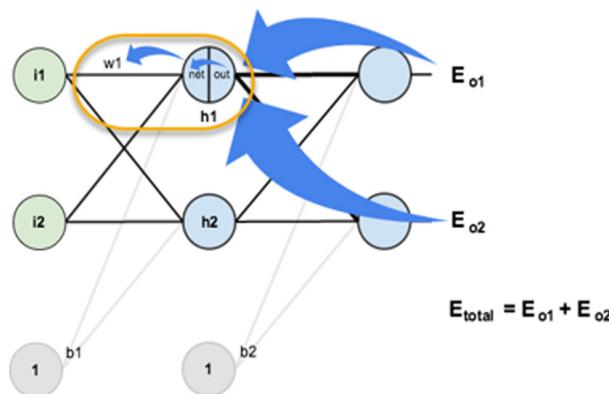
In neural networks, you do forward-propagation to get the output of your model and compare it with the real value to get the *error*.

## Back propagation

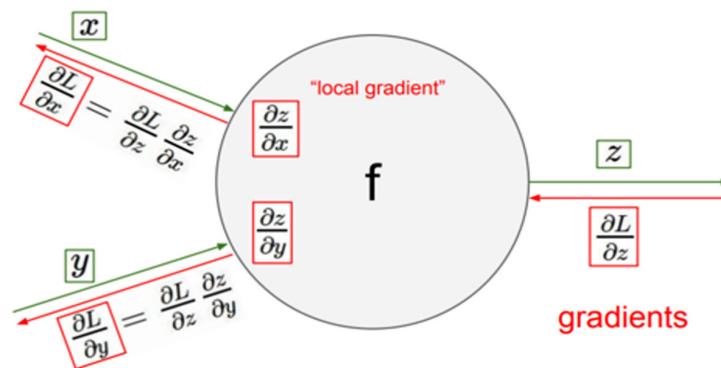
When we define a neural network, we assign random weights and bias values to our nodes. Once we receive the output for a single iteration, we can calculate the actual value against the expected value and derive the error. This error is then fed back to the network along with the gradient of the cost function to update the weights of

the network. These weights are then updated so that the errors in the subsequent iterations are reduced. This process of updating weights using the gradient of the cost function is known as back-propagation.

In back-propagation, the movement of the network is BACKWARDS, from the outputs to the inputs. The error along with the gradient flows back from the output layer through the hidden layers, and the weights are subsequently updated as shown in the following diagram:



When we do backward propagation, we go backwards through our neural network to find the partial derivatives of the error with respect to the weights. This allows us to subtract this value from the weights. The derivatives can then be used by processes such as **gradient descent** to iteratively minimize a given function. The result is the adjustment (up or down) of the weights, depending on which decreases the error:



## The No Free Lunch theorem

In machine learning, there is something called the No Free Lunch theorem. This theorem basically states that there is no one model that works best for every problem. The assumptions that you might make for one model may not necessarily apply to another one, and may also change based on your data. It is always best and highly recommended that you try multiple models to find the one that works best for your problem.

The No Free Lunch theorem holds true, especially in supervised learning. You need to assess the predictive accuracies of many models, all with varying complexity, in order to find the best model.

Finally, a model that works well for one problem could be trained by multiple algorithms and arrive at different results. For example, linear regression can be trained by normal equations or by gradient descent. This becomes very evident in our section on *Learning Rate* just a bit further on.

## The Curse of Dimensionality

The Curse of Dimensionality is a phenomenon whereby the possible configuration of a set of variables rises exponentially as the number of variables increases. When the number of dimensions becomes too high, we experience difficulties in solving our problems.

## The more neurons versus more layers

This is always an interesting discussion. What is the difference between adding another layer and increasing the neurons in an already available layer? Adding more layers basically allows your network to learn functions it wasn't able to learn before (increased complexity). Adding more neurons to an already existing layer means that we could approximate better.

## Cost function

When we build a network, the network tries to predict the output to be as close as possible to the actual value we expect. We measure this accuracy of the network using what is known as the *cost/loss* function. The cost (also called *loss*) function tries to penalize the network when it makes errors. The cost function tells us how well our algorithm performs optimization problems. What you need to remember is that the cost function is basically used for monitoring the error with each training example. There is also something known as the derivative of the cost function. With respect to a single weight, the derivative is where we need to shift that one weight so that we can minimize the error for that specific training example.

Our objective while running the network is to increase our prediction accuracy and to reduce the error, hence minimizing the cost function. The most optimized output is the one with the least value of the cost or loss function. The learning process revolves around minimizing the cost.

In machine learning, the focus is on learning things from data. The algorithm learns things to help minimize mistakes. For example, as children, we are always taught what is right or wrong, good or bad. This is normally accomplished by our parents telling us not to do things, getting punished (negative reward) for doing something we shouldn't, or in my case, both! It is the hope that by telling us what not to do, we will learn what we should do.

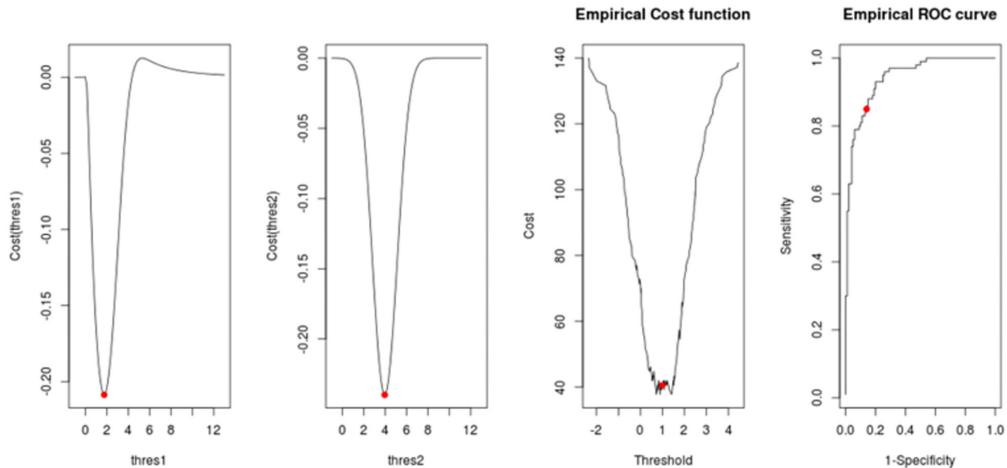
As a quick example, let us say we have a small child who is sitting on the floor playing with a toy. The child sees an electrical socket and reaches to put the toy into the socket. Fortunately, the child is quickly stopped by his/her parents and told that this can be dangerous and should not be repeated. If that parent wasn't around and the child placed something into that socket, he/she would have got a shock, which would be akin to 'negative reinforcement', letting them know that they should not do that again. We can think of the Bart Simpson cartoon which had him continually repeating that action with the socket; his ability or willingness to learn was a lot less than what we want our algorithms to achieve!

Through experience and feedback, the child hopefully learns good and bad behaviour. The electrical socket in this example serves its purpose as a cost function; it helps the child learn to correct or change his/her behaviour to minimize mistakes. In machine learning, cost functions are therefore used to estimate how poorly a model is relative to its ability to estimate the relationship between  $a$  and  $b$  (or  $X$  and  $Y$  if you prefer). This estimate is typically expressed as the difference (or distance sometimes) between the predicted and actual value. The cost is determined by iteratively running the model and comparing the estimated predictions against what is called the ground truth – the known good values of  $b$  (or  $Y$ ).

The cost function is also sometimes referred to as *loss* or *error*.

**We can state that the objective of a machine learning model is to find the parameters (or weights) that minimize the cost function.**

Here are some examples of the cost function plotted:



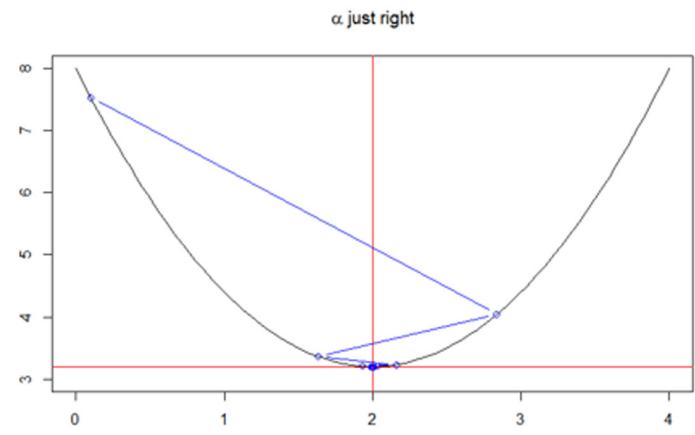
## Gradient descent

Now that we know what *cost* is, we can ask ourselves, what are some things that we can do to help minimize the cost in our models? Gradient descent is one of those things. Gradient descent is a very popular algorithm which is used for minimizing the cost. Given a function defined by a set of parameters, gradient descent starts with an initial set of parameter values and iteratively moves toward a set of parameter values that minimizes the function. Walking down the steps is gradient descent. Here is what I mean.

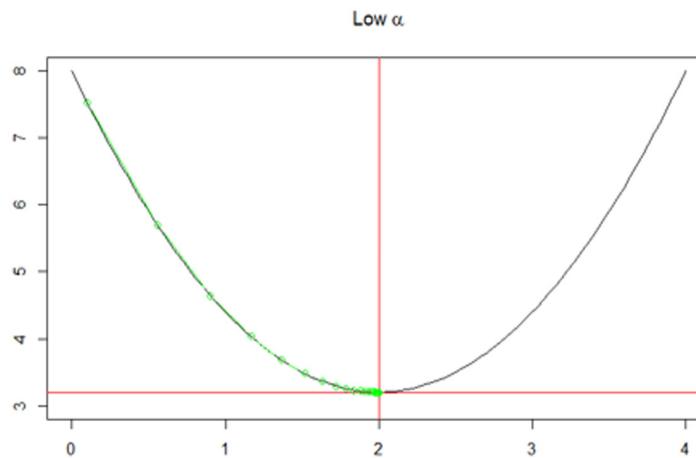
Let us think of us standing at the top of a steep hill or mountain and preparing to walk down. Our starting point was randomly assigned. Our goal is to arrive at a specific point at the bottom, which happens to be the overall lowest point. We do not want to stop short of the lowest point, and we also do not want to go too far (overshooting) and end up walking back up the hill. At the same time, we do not want to walk too fast and overshoot the bottom target point nor do we want to walk too slow and take forever to get there. We need to somehow optimize our steps to be in the right increments and the right direction in order to get us down the hill safely and to be able to stop where we want using the optimal number of steps. How large of a step we take is called our learning rate, which we will cover in depth next.

Therefore, if we start from a point  $x$ , we will move down a little (let us call this  $\delta h$ ) and update our position to  $x - \delta h$ . We will keep repeating this until we reach the bottom. Consider the bottom to be the minimum cost point, which is our target.

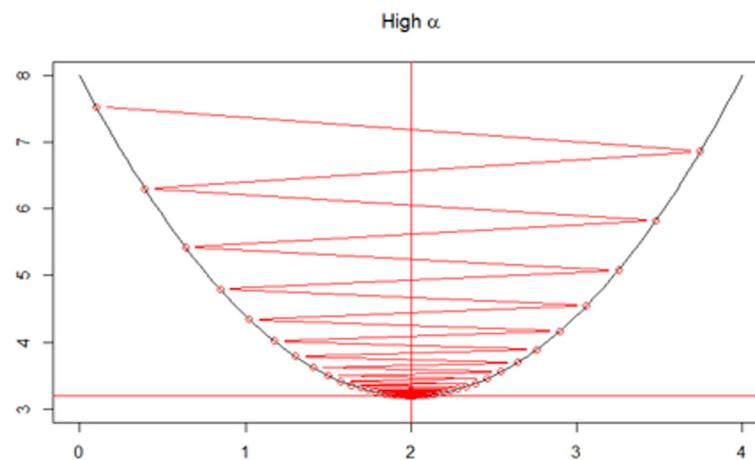
The following graph shows the process of gradient descent as well as different optima the algorithm could arrive at (blue circles). In the first graph, we arrive at our target point correctly without overshooting it or going too slow and never getting there:



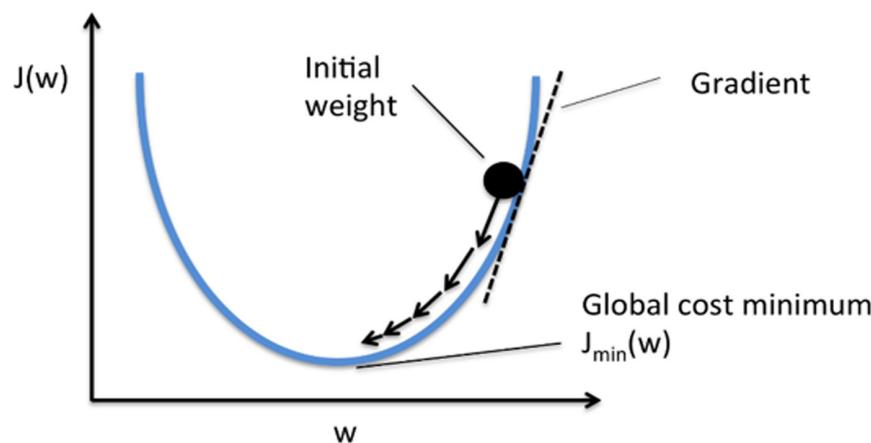
In the second graph, we arrive at our target point, but as you can see, it took many more steps than when our learning rate (discussed below) was optimal. This was a lot slower and took a lot more steps than what was optimal:



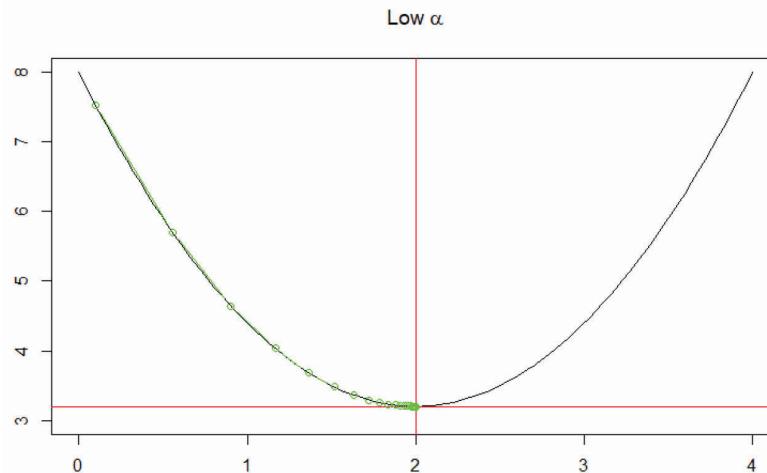
In the final graph, our journey towards our target is all over the place, not taking the optimal path at all. Way too many steps for us; we are going to exhaust ourselves and take forever to get to our destination:



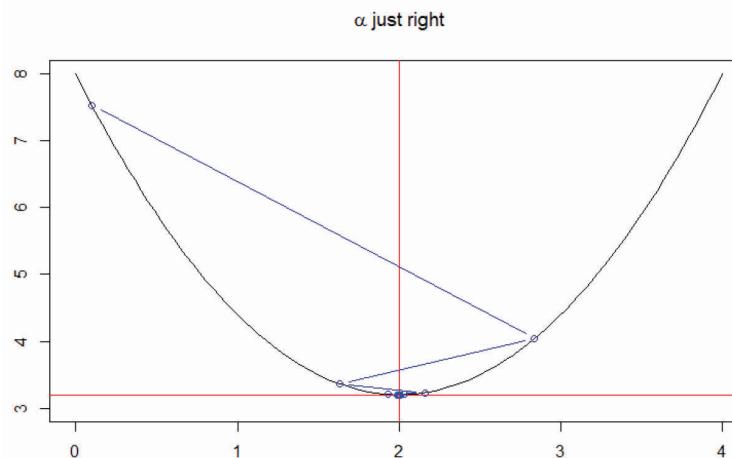
So to arrive at our optimal target, we used the initial weight to obtain the optimal gradient. The following graph puts the weights, gradient and cost into perspective for you:



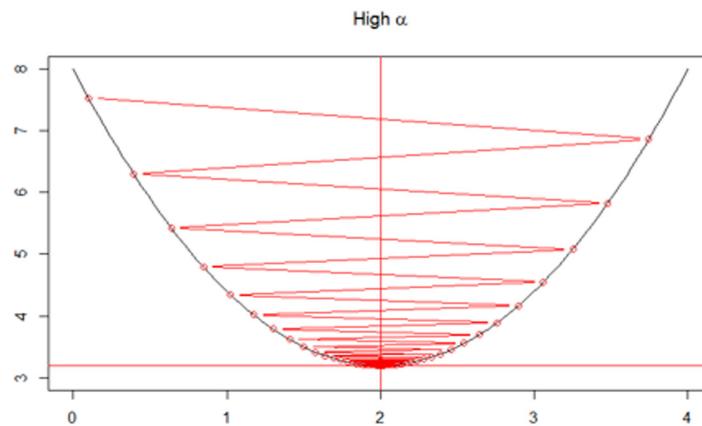
Finally, here are some graphs that will highlight the learning rates and how they affect the decent. The first graph uses a low learning rate. As shown in the following image, the descent time is very long and takes time to reach our minima.



Our next graph uses a learning rate that is just right. *Just Right* means that the descent to the local minima was not too fast or too slow and ended exactly where we needed it to be.



Our final image shows our descent made using a learning rate that is too high. As shown in the following image, we are taking wild steps to reach our minima, and it is taking much longer than desired.



## Learning rate

The learning rate is defined as the amount of minimization in the cost function in each iteration. In simple terms, the rate at which we descend towards the minima of the cost function is the learning rate. We must be very careful when selecting our learning rate. A learning rate that is too large may result in missing our optimal solution. Conversely, a learning rate that is too small may result in taking forever to reach optimality.

One problem that we face when working on deep learning projects is choosing a learning rate and optimizer (the hyperparameters). If you are like me, you will find yourself guessing an optimizer and learning rate, then checking to see if they work, only to find out that most of the times, they did not. There is no doubt that we spend a significant amount of time on our deep learning models in order to perfect this one aspect, if we ever achieve such. However, our testing has left us with some valuable insight that we can use to our advantage:

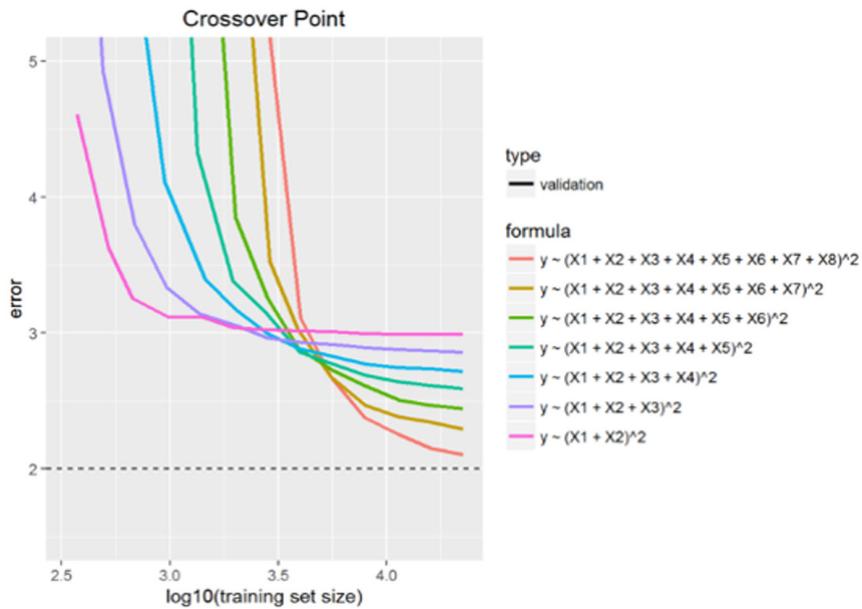
- For every optimizer, most learning rates fail to train the model effectively.
- There is a valley shape for each optimizer: too low a learning rate never progresses, too high a learning rate causes instability and never converges, and in between, there is a band of just right learning rates that can successfully train. These are what we need to seek out:
- There is no one single learning rate that works for all optimizers.
- Learning rates can affect the training time by an order of magnitude.

With this information in our back pockets, we can say that we know about the optimizers we have within Kelp.Net (they are there for a reason!):

- All the optimizers, apart from RMSProp, manage to converge in a reasonable time.
- Adam typically learns the fastest.
- Adam is more stable than the other optimizers, and it doesn't suffer any major decreases in accuracy.
- RMSProp might be a good use case for automated hyperparameter search to determine its most optimal parameters, apart from the default decay rate 0.9, epsilon 1e-10, and momentum 0.0.
- Usually, and I say usually here, Adam is our best choice but may perform poorly given the model and dataset. You will see in the experiments that we vary our optimizers quite a bit.

Sticking with Adam as our testbed optimizer, let us look at some insight we can garner about the learning rate. Again, I must stress on the fact that this can vary with the model and the dataset being used. We tried to eliminate the dataset issues with the experiments in Kelp.Net, the model, give its flexibility in definition and coding, can go either way! The learning rate performance depends on the following criteria:

- Learning rates of 0.0005, 0.001, and 0.00146 performed best. This seems to be a sweet spot band for this optimizer.
- Each learning rates' time to train grows linearly with the model size. You will see this as we move into massively deep neural networks a bit later.
- Learning rate performance did not depend on the model size; the same rates that performed best for 1 x size performed best for 10 x size.
- Above 0.001, increasing the learning rate increased the time to train, and increased the variance in the training time (as compared to a linear function of model size).
- Time to train can roughly be modeled as  $c + kn$  for a model with  $n$  weights, fixed cost  $c$  and learning constant  $k = f(\text{learning rate})$ .



The preceding graph shows that simple models that work best with small training sets are out-performed by more complex models on larger training sets. Of course, you need enough data to power these models, and the learning curve can be an indicator to let you know if you have enough data. They can help you decide if you need to gather more data or not, or if your models have learned all they can from the data you have.

## Batches/Batch size

While training a neural network, instead of sending the entire input in one go, we randomly divide the input into several chunks of equal size. Training the data in *batches* makes the model more generalized as compared to the model built when the entire data set is fed to the network in one go. The batch size is the total number of training examples present in a single batch.

## Epochs

An epoch is defined as a single training iteration of all batches in both forward and back-propagation. This means one epoch is a single forward and backward pass of the entire input data.

There are some considerations you must make when defining the number of epochs you will use. It is highly probable that the more epochs you use the higher accuracy you will achieve. By the same token, an increased number of epochs will result in

an increased time for the network to converge. Finally, be careful as an increased number of epochs could also result in the network overfitting.

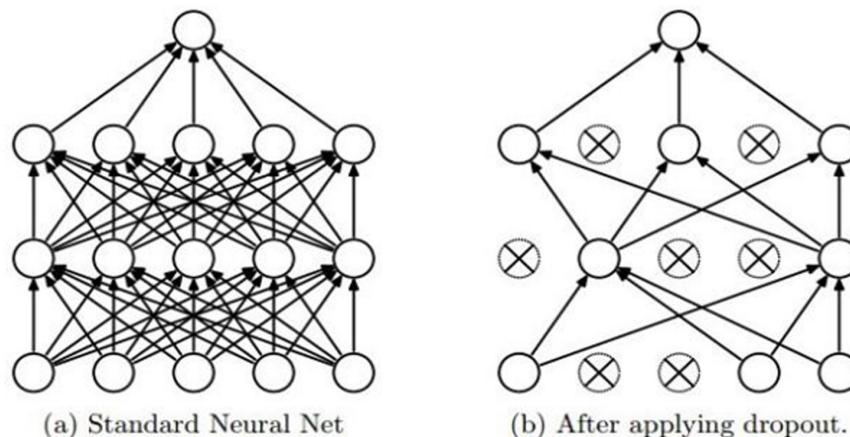
## Iterations

Iterations are the number of batches needed to complete one epoch.

## Dropout

Dropout is a regularization technique which prevents overfitting of the network. As the name suggests, during training, a certain number of neurons in the hidden layer is randomly dropped. This means that the training happens on several architectures of the neural network on different combinations of the neurons. You can think of drop out as an ensemble technique, where the output of multiple networks is then used to produce the final output.

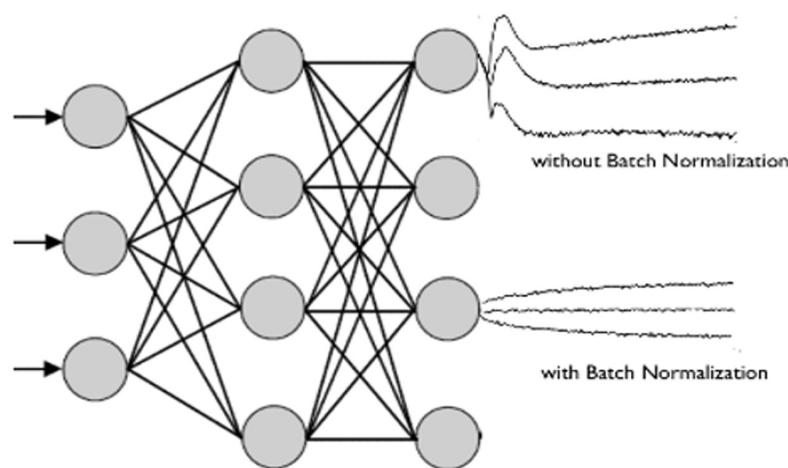
Dropout works on a neural network layer by masking a random subset of its outputs (zeroing them) for every input with probability  $p$  and scaling up the rest of the outputs by  $1/(1 - p)$ . Dropout is normally used during training. Masking prevents gradient back-propagation through the masked outputs. The method thus selects a random subset of the neural network to train on any example. This can be thought of as training a model ensemble to solve the task, with the individual models sharing parameters:



## Batch Normalization

Batch normalization works by normalizing layer outputs to a running mean and variance. This speeds up training and improves the final performance of the model. The running statistics are fixed at test time. In order to increase the stability of a

neural network, batch normalization normalizes the output of a previous activation layer by subtracting the batch mean and dividing by the batch standard deviation. After this occurs, the weights in the next layer are no longer optimal. Stochastic gradient descent undoes this normalization if there is a way for it to minimize the loss function. Therefore, the batch normalization process needs to add two trainable parameters to each layer. So the normalized output can be multiplied by a standard deviation (*gamma*), and a mean (*beta*). In this regard, stochastic gradient descent does the denormalization by changing the gamma and beta weights for each activation. This helps prevent stability loss on the network due to the changing of all the weights:



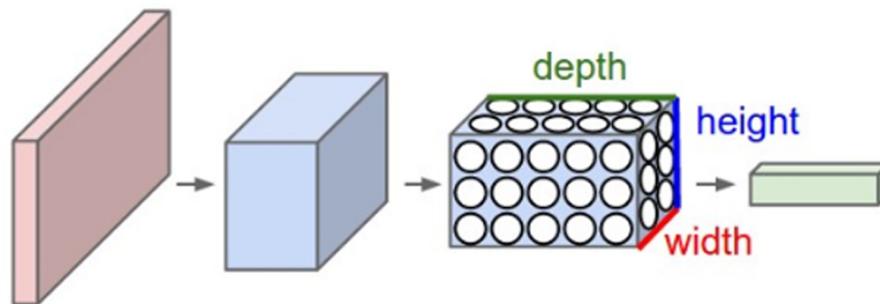
## CNN (Convolutional Neural Network)

Convolutional neural networks are very similar to ordinary neural networks described earlier which are made up of neurons that have learnable weights and bias. Each neuron receives some inputs, performs a dot product and optionally follows it with some form of non-linearity. The difference is that CNN architectures make the explicit assumption that the inputs are images. This assumption allows us to encode certain properties into the architecture itself, which makes the forward function more efficient to implement while vastly reducing the number of parameters required in the network. Pretty neat, huh? The whole network still expresses a single differentiable score function, and they still have a loss function (for example, SVM/Softmax) on the last (fully-connected) layer. All the tips/tricks we developed for learning regular neural networks still apply as well.

A CNN is a sequence of layers, and every layer transforms one volume of activations to another through what is known as a differentiable function. We use three main types of layers to build CNN architectures:

- Convolutional layer
- Pooling layer
- Fully-connected layer (as seen in regular neural networks)

These layers are stacked to form a full CNN architecture.



Convolutional neural networks are generally used on image data. Suppose we have an input image of size  $(28 \times 28 \times 3)$ . If we use a normal neural network, there would be 2352 parameters  $(28 \times 28 \times 3)$ . And as the size of the image increases, so does the number of parameters. In order to reduce the number of parameters, we convolve the images. As we slide the filter over the width and height of the input volume, we will produce a two-dimensional activation map that gives the output of that filter at every position. We will stack these activation maps along the depth dimension and produce the output volume.

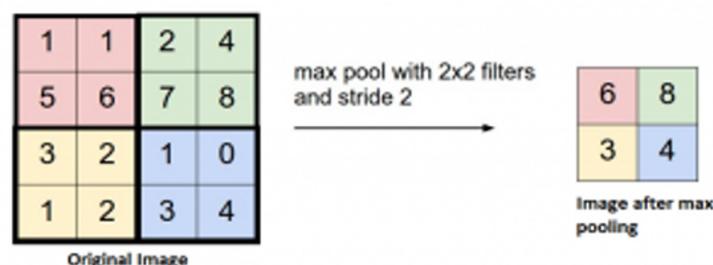
Summarily,

- A CNN architecture is, in the simplest form, a list of layers that transform the image volume into an output volume (for example, holding the class scores).
- There are a few distinct types of layers (for example, CONV/FC/RELU/POOL are by far the most popular).
- Each layer accepts an input 3D volume and transforms it to an output 3D volume through a differentiable function.
- Each layer may or may not have parameters (for example, CONV/FC do, RELU/POOL do not).
- Each layer may or may not have additional hyperparameters (for example, CONV/FC/POOL do, RELU doesn't).

## Pooling

CNNs (described earlier) typically feature pooling layers. A pooling layer basically summarizes the activities of local patches of neurons in convolutional layers. Essentially, a pooling layer takes as input the output of a convolutional layer and in turn, subsamples it. A pooling layer consists of pooling units which are laid out topographically and connected to a local neighborhood of convolutional unit outputs from the same bank. Each pooling unit then computes some functions of the bank's output in that neighborhood.

One common approach is to periodically introduce pooling layers in between the convolution layers. This is basically done in order to reduce several parameters and prevent over-fitting. The most common type of pooling is a pooling layer of filter size 2x2, using the MAX operation. This then takes the maximum of each 4x4 matrix of the original image. Please note that while the MAX operation has shown to work better in practice, there is nothing preventing you from using other pooling operations such as AVG and so on:



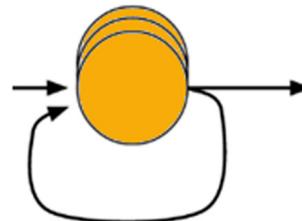
## Padding

Padding refers to adding extra layer of zeros across the images so that the output image has the same size as the input. This is also known as **sample padding**. The following image is an example of the same:

0	0	0	0	0	0
0	35	19	25	6	0
0	13	22	16	53	0
0	4	3	7	10	0
0	9	8	1	3	0
0	0	0	0	0	0

After the application of filters, the convolved layer has the size equal to the actual image. Valid padding refers to keeping the image and having all the pixels of the image which are actual or valid. In this case after the application of filters, the size of the length and the width of the output keep getting reduced at each convolutional layer.

## Recurrent neuron

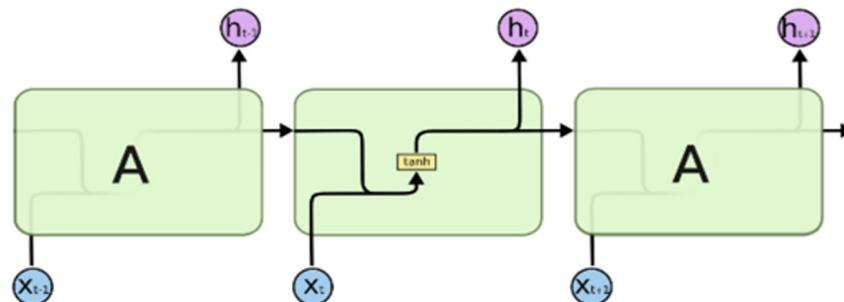


A recurrent neuron is one in which the output of the neuron is sent back to it (repeated) for  $t$  timestamps. The basic advantage of this neuron is that it gives a more generalized output. Here is a basic overview of how a recurrent neuron performs its work:

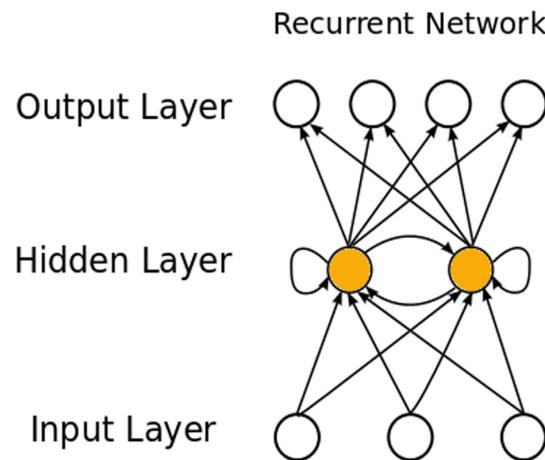
1. A single time step of the input is supplied to the network ( $x_t$ ).
2. Next, we calculate its current state using a combination of the current input and the previous state ( $h_t$ ).

3. The current  $h_t$  becomes  $h_{t-1}$  for the next time step.
4. We perform this as many time steps as the problem demands and combine the information from all the previous states.
5. Once all the time steps are completed, the final current state is used to calculate the output ( $y_t$ ).
6. The output is then compared to the actual output and the error is generated.
7. The error is then back propagated to the network to update the weights and the network is trained.

## RNN (Recurrent Neural Network)



Recurrent neural networks are used mostly for sequential data where the previous output is used to predict the next one. In this case, the networks have loops within them. The loops within the hidden neuron give them the capability to store information about the previous words for some time to be able to predict the output:



The output of the hidden layer is sent again to the hidden layer for  $t$  timestamps. The output of the recurrent neuron goes to the next layer only after completing all the timestamps. The output sent is more generalized and the previous information is retained for a longer period. The error is then back propagated according to the unfolded network to update the weights. This is known as **back-propagation through time (BPTT)**.

## Vanishing gradient problem

The vanishing gradient problem arises in cases where the gradient of the activation function is very small. During back-propagation, when the weights are multiplied with these low gradients, they tend to become very small and vanish as they go further deep into the network. This generally becomes a problem in cases of recurrent neural networks where long-term dependencies are very important for the network to remember. This can be solved by approaches such as:

- Leaky activation functions such as ReLU
- Hessian free optimizer with structural dumping
- Vanishing gradient regularization
- Long short-term memory
- Gated recurrent unit
- Orthogonal initialization

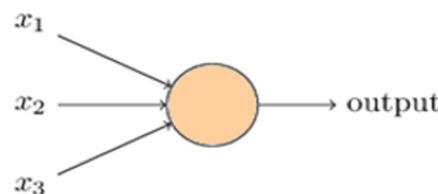
## Exploding gradient problem

This is exactly the opposite of the vanishing gradient problem described above. In the case of the exploding gradient problem, the gradient of the activation function is too large. During back-propagation, it makes the weight of a node very high with respect to the others, therefore rendering them insignificant. This can be easily solved by approaches such as:

- Clipping the gradient so that it doesn't exceed a certain value
- Truncated Backpropagation Through Time (TBPTT)
- L1 and L2 penalty on the recurrent weights
- Teacher forcing
- Echo state networks

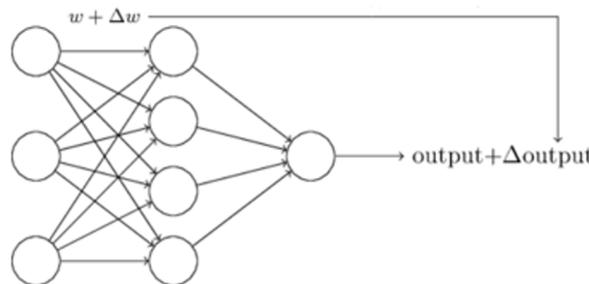
## Logistic Neurons

A **logistic neuron** (also called a **Sigmoid Neuron**) is designed to solve a unique purpose. It is similar to a regular neuron except that it is modified such that small changes in weights and bias cause only a small change in their output. This results in a network of logistic neurons being able to learn. Let us talk about this a little more in detail to understand why we need to do this. Let us start by looking at a regular neuron as it is represented in the following image:



Just like a regular neuron, the sigmoid neuron has inputs,  $x_1$ ,  $x_2$ , and  $x_3$ . But instead of taking values 0 or 1, these inputs can also take any values between 0 and 1. This means that values such as 0.44, 0.4453, 0.9, and 0.00032...0.03445 are valid inputs for a sigmoid neuron. Also, just like a regular neuron, the sigmoid neuron has weights for each input,  $w_1$ ,  $w_2$ , and  $w_3$ , and an overall bias,  $b$ . But the output is not 0 or 1. Instead, it is  $(w \cdot x + b) / (w \cdot x + b)$ , where  $w \cdot x + b$  is called the sigmoid function.

By using the actual sigmoid function, we get a smoothed out neuron. Indeed, it is the smoothness of the sigmoid function that is the crucial fact, not its detailed form. The smoothness of the sigmoid function means that small changes in the weights and in the bias will produce a small change in the output from the neuron:



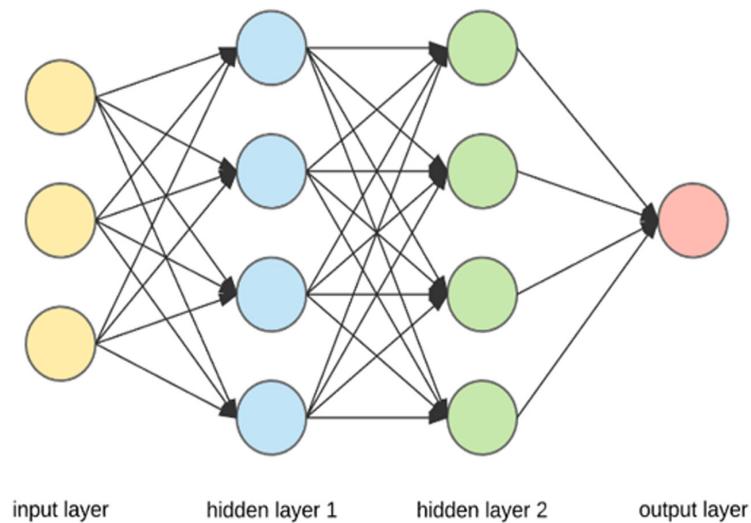
*Small change in weight (weight plus delta) results in a small change in output*

So, while sigmoid neurons have much of the same qualitative behavior as neurons, they make it much easier to figure out how changing the weights and biases will change the output.

How should we interpret the output from a sigmoid neuron? Obviously, one big difference between neurons and sigmoid neurons is that sigmoid neurons don't just output 0 or 1. But suppose we want the output from the network to indicate either '*the input image is a 9'* or '*the input image is not a 9*'. Obviously, this would be the easiest to do if the output was a 0 or a 1, as in a normal neuron. But in practice, we can set up a convention to deal with this; for example, by deciding to interpret any output of at least 0.50 indicates a 9 and any output less than 0.50 does not indicate a 9.

## Hidden layers

Now, if we enhance what we have been discussing and turn this into a complex network of neurons (as follows), we can make even more informed and complex decisions concerning our data. This would be a neural network. The following image is one such network architecture:

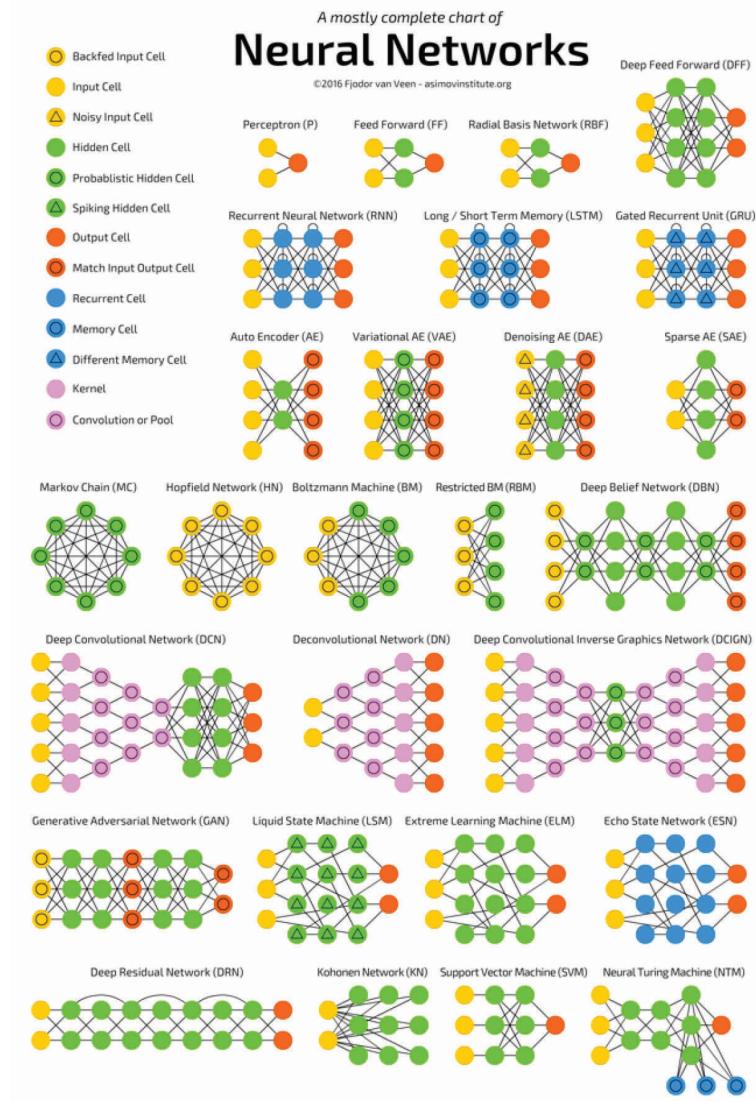


The circles between the inputs and outputs are neurons and are known as **hidden layers**. They are the key to our discussion of deep learning (more on that in *Chapter 4, Kelp.Net Reference*). We may have very large number of layers in our network, or we may have only a few. Generally speaking, anything more than one hidden layer is considered a deep learning/belief network by the general public nowadays. But let me caution you that more is not always better in this regard. The number of hidden layers available is not directly proportional to the complexity of problem you can solve.

Since we brought up the concept of hidden layers, what are some good rules to follow for hidden layers? One such rule we should always adhere to is that the size of a hidden layer, or more concretely, the number of neurons that it has should be close to the dimension of the space which we want to map our inputs to. So, if we have an output set with three layers, we must have at least one hidden layer with at least three neurons. If we do not, we might not be able to separate our data regardless of the depth (number of hidden layers) we use.

## Types of neural networks

The following are some of the types of neural networks that are known today:



*Reprinted with permission, Copyright Asimov Institute  
[\(http://www.asimovinstitute.org/neural-network-zoo/\)](http://www.asimovinstitute.org/neural-network-zoo/)*

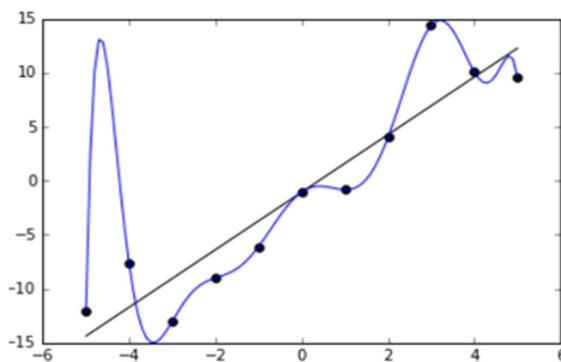
## Generalization

Generalization refers to your model's ability to adapt properly to new, previously unseen data, drawn from the same distribution as the one used to create the model. Testing for generalization is one of the reasons why we split the original dataset into test and training batches. If your model generalizes well, your algorithm is learning. If it does not, then you may be entering the brutal world of underfitting and overfitting, as discussed below. The goal is to have a model that generalizes as well under most conditions. It is iterative, and it a process. Remember that the goal is to get a model that can accurately predict labels on data that has yet to be seen. This is generalization at its simplest.

## Regularization

Regularization is any modification that we might make to an algorithm with the purpose of reducing the generalization error, but not the training error. Regularization is an important technique, right up there with general optimization. One of the ways in which we can accomplish regularization is by adding a parameter to our model called a regularizer. This parameter would be added to the cost function. There are many types of regularizers one can use to accomplish their goals.

Let us take a look at the following plot:



This plot shows underfitting (the solid straight line) and overfitting (the curvy line). In order to address underfitting, you can usually add additional features from your data. To address overfitting, you can sometimes reduce the number of features, or you can use regularization, which is easier and more productive than trying to remove features.

## Loss

When it comes to a neural network, the least is the loss the better the model. The loss is calculated on training and validation data, and its interpretation is how well the model is doing for these two sets. Loss is a summation of the errors made for each example in training or validation sets.

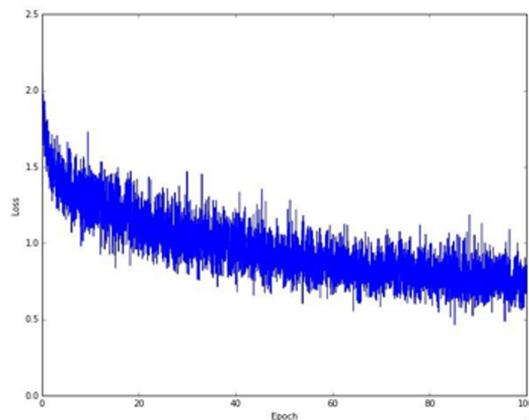
The main objective in a learning model is to reduce (minimize) the loss function's value with respect to the model's parameters by changing the weight vector values through different optimization methods, such as backpropagation in neural networks.

The loss value implies how well or poorly a certain model behaves after each iteration of optimization. Ideally, one would expect the reduction of loss after each, or several, iteration(s).

The accuracy of a model is usually determined after the model parameters are learned and fixed and no learning is taking place. Then, the test samples are fed to the model and the number of mistakes the model makes is recorded, after comparing the true targets. Then, the percentage of misclassification is calculated.

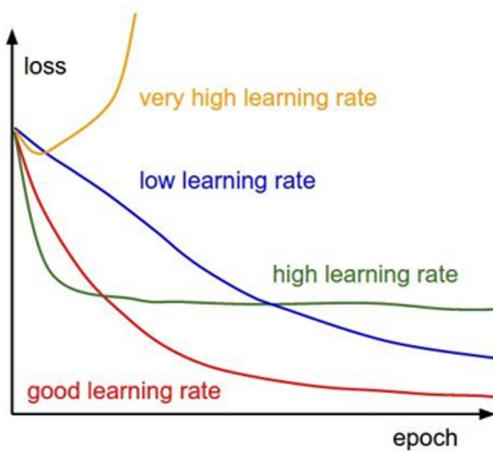
## Loss over time

The first quantity that is useful to track during training is the loss, as it is evaluated on the individual batches during the forward pass. Here is a diagram that shows what the loss over time might look like:



## Loss versus learning curve

Here is a diagram which depicts the different shapes of a loss curve and how these shapes relate to the learning rate (covered earlier):



## Supervised learning

These types of machine learning models are used to predict the outcome based on the data and instructions presented to it. The instructions provided are explicit and detailed, or at least should be, which is why we label it **supervised**.

In supervised learning, we are basically learning a function which maps an input to an output based on input and output pairs. This function is inferred from the training data which is called **labeled**, which specifically tells the function what it expects. In supervised learning, there is always an input and desired output value. More formally, this type of algorithm uses a technique known as inductive bias to accomplish this, which basically means that there are a set of assumptions which the algorithm will use to predict the outputs, given inputs it may or may not have previously seen.

In supervised learning, we typically have access to a set of  $x$  features ( $X_1, X_2, X_3, \dots, X_n$ ), measured on  $n$  observations, and a response  $Y$ , also measured on these same  $n$  observations. We then try and predict  $Y$  using  $X_1, X_2, X_3, \dots, X_n$ .

Some examples of supervised learning are as follows:

- Support Vector Machines
- Linear regression
- Naïve Bayes
- Tree-based methods

Next, let us briefly discuss a few things which we need to learn when it comes to supervised learning. They are in no particular order:

- Bias-Variance Trade-off
- Amount of data
- Input space dimensionality
- Incorrect output values
- Data heterogeneity

## Bias-Variance Trade-off (overfitting and underfitting)

Before we talk about the bias-variance trade-off, it would only make sense if we first make sure that you are familiar with the individual terms.

### Bias

Bias refers to an error from incorrect assumptions in the learning algorithm. High bias causes what is known as underfitting, a phenomenon which causes the algorithm to miss relevant feature-output layer relationships in the data.

### Variance

Variance, on the other hand, is a sensitivity error to small fluctuations in the training set. High variance can cause your algorithm to model random noise rather than the actual intended outputs, a phenomenon known as overfitting.

### Overfitting

The word overfitting refers to a model that models the training data too well. Instead of learning the general distribution of the data, the model learns the expected output for every data point:



More concretely, your algorithm memorizes the data and learns too much from it, which is similar to memorizing the answers for a test. If you were in school and

you memorized the answers for a test, consider the implications of what happens if someone changes a few questions in that test? This is the same scenario when you move from your training to your test sets. With overfitting, your algorithm may start to map shapes and curves that do not exist in the data. When you change the data, then your results become erratic and unpredictable.

There is a trade-off between bias and variance that every machine learning developer needs to understand. It has a direct correlation to under and over fitting of your data. We say that a learning algorithm has a high variance for an input if it predicts a different output result when used on a different training set, and that of course is not good and not what we want.

A machine learning algorithm with **low bias** must be flexible enough so that it can fit the data well. If the algorithm designed is too flexible, each training and test dataset will fit differently, resulting in high variance. Your algorithm must be flexible enough to adjust this trade-off either by inherent algorithmic knowledge or a parameter which can be user adjusted.

## Is your model overfitting or underfitting?

How can we determine if a model is underfitting or overfitting? The answer is by looking at the prediction error in the training and evaluation data. Our model will be underfitting when the model performs poorly in the training data. It is unable to achieve a sufficiently low error value. This is because the model cannot capture the relationship between the input and target values.

Our model will be overfitting when we see that the model performs well in the training data but does not perform well in the test data. This is due to the model memorizing the data and therefore it makes it difficult for it to generalize any unseen data. The gap between the test and training error is too large.

Poor performance can also be because the model is too simple, which means that the input features are not fully expressed. If this is the case, the input features cannot describe the target well enough to classify it. Your option here is to add new features, increase n-gram size, and more. Alternatively, you can decrease the amount of regularization that is being used.

If the model is overfitting, you must do the opposite which is to reduce the model flexibility. You can do so by using fewer feature combinations, decrease n-gram size, decrease the number of attributes, or increase the amount of regularization being used.

Finally, always consider increasing the amount of training data, or in some cases, consider increasing the number of passes over the existing training data. Poor performance could be due to the learning algorithm not having enough data from which to learn from.

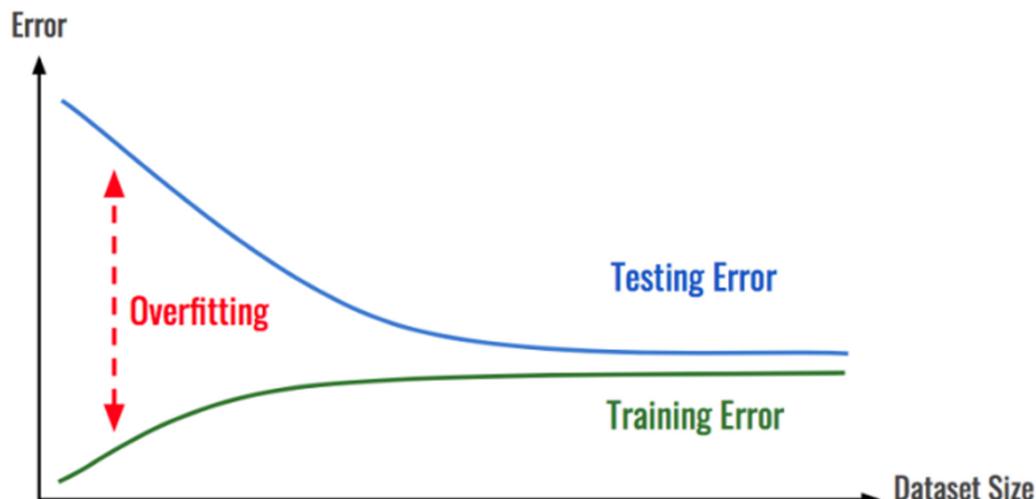
Here are some graphs showing the different types of fitting that may occur, and how each one looks like when plotted against the data. The middle image (an ideal fit) is our target goal:



## Prevention of overfitting and underfitting

Let us go through the prevention of underfitting and overfitting again since it is very important for everyone to understand.

Gather more data (if possible). This should always be your first choice of action. Why? This is because your model can only store so much information. This means that the more training data you feed it with, the less likely it is to overfit. The reason is that as you add more data, the model becomes unable to overfit all the samples, so it is forced to generalize to make progress. Gathering more data also has a by-product of increasing the accuracy of your model:

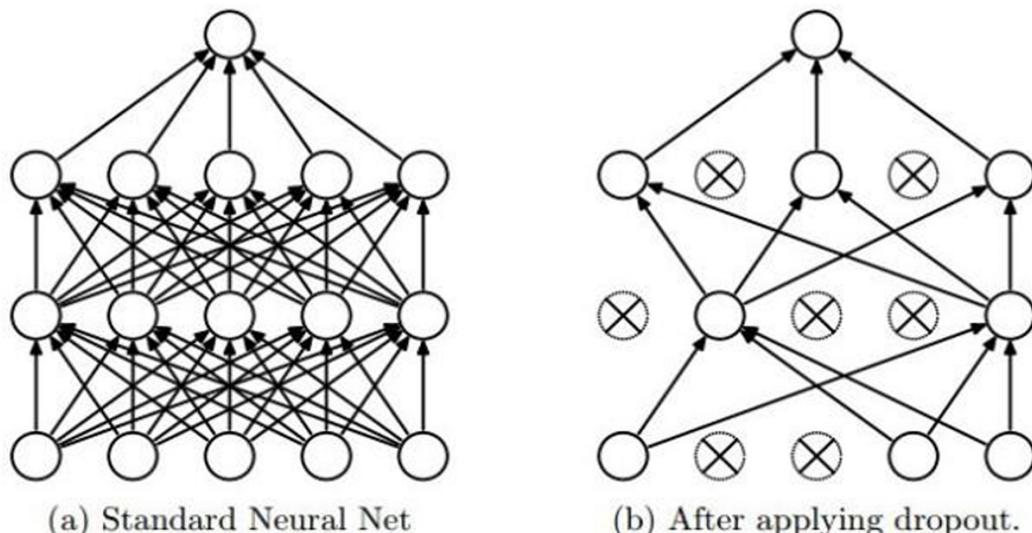


Next, use a process which is known as regularization. **Regularization** is a process of constraining the learning of the model to reduce overfitting. It can take many different forms. One of the most powerful and well-known techniques of regularization is to add a penalty to the loss function. The most common penalties are L1 (1) and L2 (2) where:

- The L1 penalty aims to minimize the absolute value of the weights.
- The L2 penalty aims to minimize the squared magnitude of the weights.

With the penalty, the model is forced to make compromises on its weights, as it can no longer make them arbitrarily large. This makes the model more general, which helps combat overfitting.

We can also try some advanced techniques if you have a deep learning network. These techniques are known as DropOut and DropConnect. These techniques are extremely effective and rely on the fact that the neural network processes information from one layer to the next. The idea then is to randomly deactivate either the neurons (**DropOut**) or the connections (**DropConnect**) during the training process. This forces the network to become what is known as **redundant**, as it can no longer rely on specific neurons or connections in order to extract specific features. Once the training is complete, all neurons and connections are restored to their original locations. The following image shows how this will look like:



## Amount of training data

As we have said repeatedly, there simply is no substitute for having enough data to get the job done correctly and completely. This directly correlates to the complexity of your learning algorithm. A less complex algorithm with high bias and low variance can learn better from a smaller amount of data. However, if our learning algorithm is complex (many input features, parameters, and so on), then you will need a much larger training set from which to learn from with low bias and high variance. You should always spend the first part of your project determining how much data you have access to, where it is located, how clean the data is, any missing data, and from where you can get more data if required.

## Input space dimensionality

With every learning problem, our input is going to be in the form of a vector. The feature vector, which means the features of the data, can affect the learning algorithm greatly. If the input feature vectors are very large, that is, what we call high dimensionality, then learning can be more difficult even if you only need just a few of those features. Sometimes, the extra dimensions confuse your learning algorithm, which results in high variance. This, in turn, means that you will have to tune your algorithm to have lower variance and higher bias. It is sometimes easier, if applicable, to remove the extra features from your data, thus improving your learning function accuracy.

Having said this, a technique known as dimensionality reduction can be accomplished by several machine learning algorithms such as the **Principal Components Analysis (PCA)** family of algorithms. These algorithms will identify and remove irrelevant features if you need them.

## Incorrect output values

One question we must ask ourselves is how much error exists in the desired output from our machine learning algorithm. If we experience this, the learning algorithm may be attempting to fit the data too well, resulting in something we mentioned previously, overfitting. Overfitting can result from incorrect data or a learning algorithm which is too complex for the task at hand. If this happens, we need to either tune our algorithm or look for one which will provide us with higher bias and lower variance. Incorrect output values are important to look out for as they are an indication that something is amiss in your algorithm.

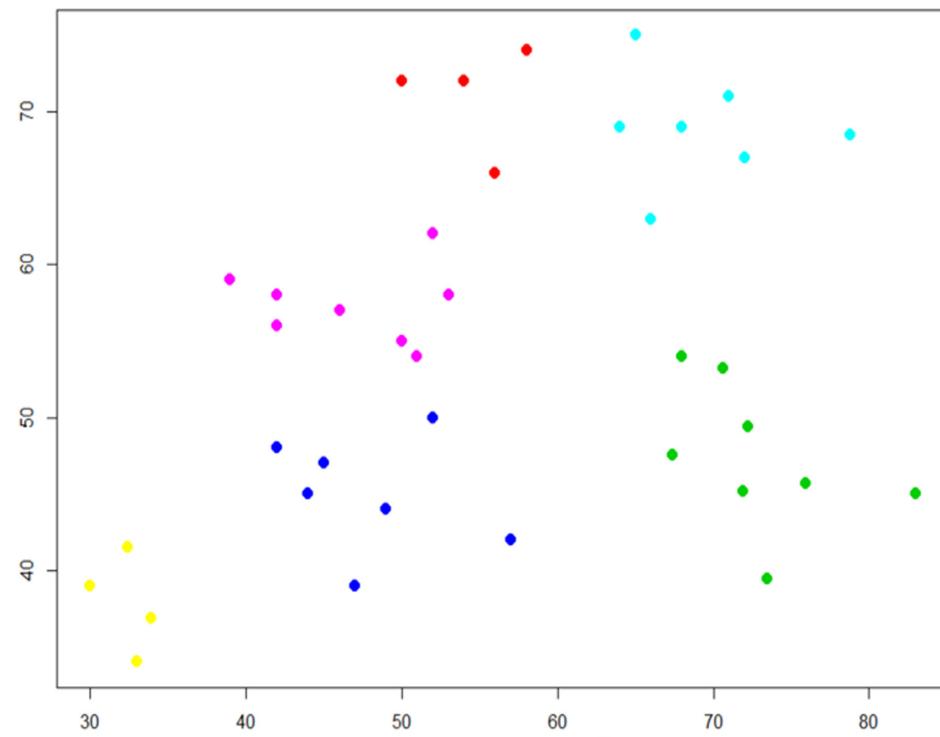
## Data heterogeneity

Heterogeneity, according to Webster's dictionary, means the quality or state of consisting of dissimilar or diverse elements: the quality or state of being heterogeneous. To us this means that the feature vectors include features of many different kinds. If this applies to our application, then it may be better for us to apply a different learning algorithm for the task. Some learning algorithms also require that our data is scaled to fit within certain ranges such as [0 – 1], [-1 – 1], and so on. As we get into learning algorithms that utilize distance functions as their basis such as the nearest neighbor and support vector methods, you will see that they are exceptionally sensitive to this. On the other hand, algorithms such as tree-based (decision trees, and more) handle this phenomenon quite well.

We will end this discussion by saying that we should always start with the least complex and most appropriate algorithm and ensure our data is collected and prepared correctly. From there, we can always experiment with different learning algorithms and tune them to see which one works best for our situation. Make no mistake, tuning algorithms may not be a simple task, and in the end, consume a lot more time than we have available. Always ensure the appropriate amount of data is available first!

## Unsupervised learning

Contrary to supervised learning, unsupervised usually has more leeway in how the outcome is determined. The data is treated such that, to the algorithm, there is no single feature more important than any other in the dataset. These algorithms learn from datasets of input data without the expected output data being labeled. K-Means Clustering (Cluster Analysis) is an example of an unsupervised model. It is very good at finding patterns in the data that have the meaning relative to the input data. The big difference between what we learned in the supervised section and here is that we now have  $x$  features  $X_1, X_2, X_3, \dots, X_x$  measured on  $n$  observations. But we no longer are interested in the prediction of  $Y$  because we no longer have  $Y$ . Our only interest is to discover data patterns over the features that we have.



In the preceding diagram, you can see that data such as this lends itself much more to a non-linear approach, where the data appears to be in clusters relative to the importance (grouped by colour for easier assimilation). It is non-linear because there is no way we will get a straight line to accurately separate and categorize the data. Unsupervised learning allows us to approach a problem with little to no idea what the results will, or should, look like. The structure is derived from the data itself versus supervised rules applied to output labels. This structure is usually derived by clustering relationships of data.

For the corporate world, this type of algorithm/problem/architecture will present you with yet another problem you will have to solve: explaining what the algorithm is doing, what it found and why. Most higher-up management does expect to ask these questions from time to time and usually are unkind to 'I'm not sure' responses. So proper debugging (see the *ReflectInsight* section in *Chapter 3, Deep Instrumentation using ReflectInsight*) will be key to helping you understand what your algorithm is doing and why.

For example, let us say we have 108 genes from a genomic data science experiment. We would like to group this data into similar segments such as hair color, lifespan, weight, and so on.

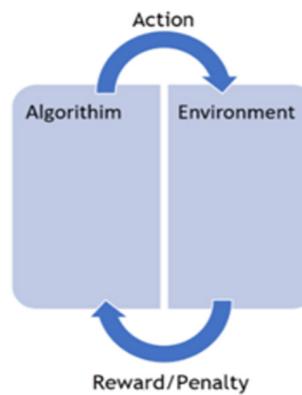
On the other end of the spectrum is what is famously known as the cocktail party effect, which basically refers to the brain's auditory ability to focus attention on one thing and filter out the noise around it.

Both the examples can use clustering to accomplish their goals.

## Reinforcement learning

Reinforcement learning is a case where the machine is trained for a specific outcome with the sole purpose of maximizing efficiency and/or performance. The algorithm is rewarded for making the correct decisions and penalized for making incorrect ones. Continual training is used to constantly improve performance. The continual learning process means less human intervention. Markov models are an example of reinforcement learning and self-driving autonomous automobiles are a great example of such an application. It constantly interacts with its environments, watches for obstacles, speed limits, distance, pedestrians, and so on to (hopefully) make the correct decisions.

Our biggest difference with reinforcement learning is that we do not deal with correct input and output data. The focus here is on the performance, which means somehow finding a balance between unseen data and what the algorithms already have learned. The algorithm applies an action to its environment, receives a reward or a penalty based on what it has done and repeats, and so on. You can just imagine how many times per second this happens in that little autonomous taxi that just picked you up at the hotel:



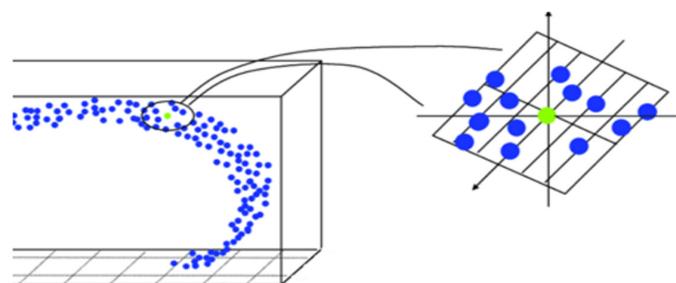
## Manifold learning

The dimensionality of a dataset is the number of variables used to represent it. For example, if we are interested in describing people in terms of their height and weight, our people dataset would have two dimensions. If instead we had a dataset

of images, and each image is a million pixels, then the dimensionality of the dataset would be one million. In fact, in many modern machine learning applications, the dimensionality of a dataset could be massive.

When dimensionality is very large (larger than the number of the samples in the dataset), we can run into some serious problems. Consider a simple classification algorithm that seeks to find a set of weights  $w$  such that when dotted with a sample  $x$ , gives a negative number for one class and a positive number for another.  $w$  will have a length equal to the dimensionality of the data, so it will have more parameters than the samples in the entire dataset. This means that a learner will be able to overfit the data, and consequently, the learner will not generalize well to other samples unseen during the training.

A manifold is an object of dimensionality  $d$  that is embedded in some higher dimensional space. Less formally, a manifold is a *connected region*. Imagine a set of points on a sheet of paper. If we crinkle up the paper, the points are now in three dimensions (poor example but hopefully you get the point). Many manifold learning algorithms seek to uncrinkle the sheet of paper to put the data back into two dimensions. A somewhat better example, perhaps, is the planet we live on. To us it looks flat, but it is a sphere. So, you could look at it as though it is a 2D manifold embedded in the 3D space. The following image shows this concept:



Even if we are not concerned with overfitting our model, a manifold learner can produce a space that makes classification and regression problems a lot easier. It does this basically by assuming that most of the data points are invalid, and that the data points that we should be most interested should be along manifolds, or when we move between manifolds.

## Types of manifolds in deep learning

There are a few types of manifold learning that I will briefly mention here. It is beyond the scope of this book to get too far into the weeds on them; it is enough that you know about these types of algorithmic approaches.

## Topological

A topological manifold allows you to talk about surfaces as topological objects, that is, you care about their general shape, but you do not care about things like distances, or angles, or corners. From the point of view of topology, there is no difference between a cube and a sphere.

## Differentiable

A differentiable manifold is a little more restrictive (any differentiable manifold is a topological manifold, but the reverse does not always hold true). On a differentiable manifold, it still does not make sense to talk about distances and angles, but there is this notion of smoothness—a cube is no longer the same as a sphere since a cube has edges and corners. However, a sphere and an ellipsoid are essentially the same. This gives us enough structure to talk about differential forms and set up essentially most of calculus.

## Riemannian

A Riemannian manifold is the most restrictive one. The addition of a Riemannian metric allows you to make sense of angles and distances. It is the correct setting for talking about geometry and curvature. A sphere and ellipsoid are different objects from the point of view of Riemannian geometry. One has constant curvature and the other curvature varies depending on where you are on the surface.

## Principal Component Analysis (PCA)

Principal Component Analysis is a method of extracting important variables (as components) from a larger set of variables within a dataset. It extracts low-dimensional features from high-dimensional datasets to capture as much information as possible. With fewer variables, the hope is that visualizing the data becomes more intuitive and meaningful. PCA is most useful when dealing with three or higher dimensional data.

The aim of PCA is to reduce the number of dimensions of a dataset where dimensions are not completely decorrelated. PCA provides us with a new set of dimensions, the principal components (PC). The first PC is the dimension having the largest variance. Each additional PC is orthogonal to the preceding one (remember that the orthogonal vectors means that their dot product is equal to 00). This means that each PC is decorrelated to the preceding one. It is way better than the feature selection in which we risk losing a considerable amount of information.

## Hyperparameter training

A model hyperparameter is a configuration that is external to the model and whose value cannot be estimated from data. No doubt you will spend a considerable amount of time tuning hyperparameters. The choice of the hyperparameters will affect the duration of the training and the accuracy of the predictions. For most datasets, only a few of the hyperparameters really matter.

Some hyperparameters are more common than others. The most common are as follows:

- The learning rate
- Momentum
- Number of hidden layers
- Mini-batch size

The learning rate is arguably the most important hyperparameter you will need to deal with, at least initially. This parameter controls what is known as the **effective capacity** of the model. A network with more layers and more neurons per hidden layer has a higher capacity, which means that it can represent more complicated functions. Even though more complicated functions can be represented, this does not necessarily mean that they are capable of being learned effectively.

The effective capacity is affected by the ability of the learning algorithm to minimize the cost function used to train the model. When the learning rate is too high, you can see an increase in the gradient descent rather than a decrease in the training error. When the learning rate is too low, training will be slower and can permanently be stuck. You may also encounter higher than normal training errors. The effective capacity is important to understand as well as how it is affected by hyperparameters.

There are two ways in which we can tune hyperparameters: automatic and manual. Manual tuning requires a deeper understanding of each hyperparameter and its effect on the system. Automatic tuning negates this requirement, but the flipside is that it is much more expensive both in time and in computation. Automatic tuning is performed by using algorithms which automatically infer a potential set of hyperparameters and attempt to optimize them. Grid and Random search are two methods of performing automatic tuning.

## Approaches to hyperparameter tuning

The following sections describe some of the many approaches to hyperparameter tuning:

### Grid search

The traditional way of performing hyperparameter optimization is basically what is also known as a parameter sweep and is an exhaustive search through a manually specified set of the hyperparameter space. We simply build a model for each possible combination of all of the hyperparameter values provided, evaluate each model, and select the architecture which produces the best results.

### Random search

Random search replaces the exhaustive enumeration of grid search by selecting hyperparameters randomly. Random search differs from grid search in such a way that we no longer provide a discrete set of values to explore for each hyperparameter; rather, we provide a statistical distribution for each hyperparameter from which values may be randomly sampled.

### Bayesian optimization

Bayesian optimization has been shown to obtain better results in fewer evaluations compared to grid and random search due to the ability to reason out the quality of experiments before they run.

### Gradient-based optimization

For specific learning algorithms, it is possible to compute the gradient with respect to hyperparameters and then optimize the hyperparameters using gradient descent.

### Evolutionary optimization

Evolutionary optimization uses evolutionary algorithms to search the space of hyperparameters for a given algorithm.

The following table summarizes some of the important aspects of hyperparameters and their effect on effective capacity:

Hyperparameter	Reason	Notes
<b>Total hidden units</b>	Increasing total hidden units increases capacity	Increasing total hidden units increases the total memory and operational time cost.
<b>Learning rate</b>	Too high or too low results in low effective capacity.	
<b>Zero padding</b>	Adding zeros before the convolution stage keeps the size large	Increases both time and memory cost.
<b>Weight decay</b>	Decreasing allows the model parameters to become larger	
<b>Dropout rate</b>	Dropping less neurons allows more opportunity for the neurons to fit the training set	Usually searched in non-logarithmic scale.

Larger neural networks such as those found in deep learning scenarios typically require a long time to train. Therefore, performing hyperparameter search can take many days or even weeks. It is important to keep this in mind since it influences choices made to your design.

## Summary

In this section, we covered a lot of machine and deep learning terms and concepts. For those of you who have not had the educational background in this field, I hope you will now feel a little more comfortable the next time you encounter these terms.

In the next chapter, we will deep dive into Kelp.Net.

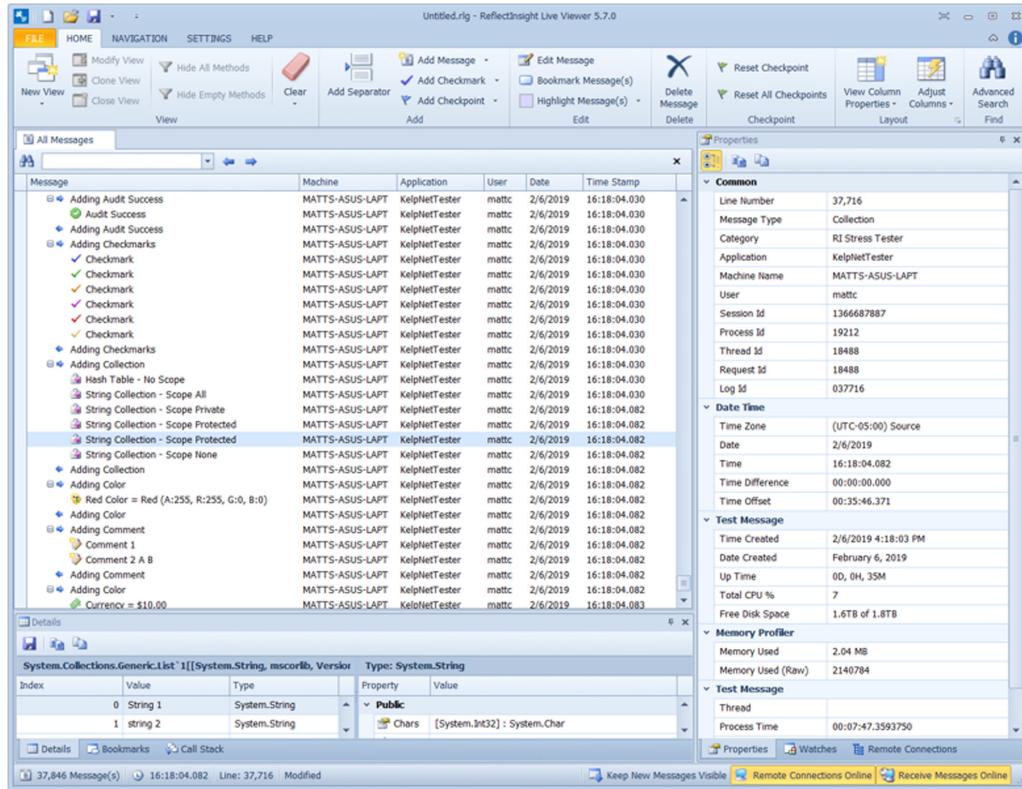
## References

- <http://mathworld.wolfram.com/L1-Norm.html>
- <http://mathworld.wolfram.com/L2-Norm.html>

# CHAPTER 3

# Deep Instrumentation Using ReflectInsight

One of the things I strongly feel is to have a good suite of tools at your disposal in order to accomplish your job. Logging is just one of the tools. The best logging tool is the one that you do not even realize is working, and ReflectInsight from ReflectSoftware is such a tool. It is one of the few, if not the only, production-ready, enterprise grade loggers available. It is incredibly powerful, flexible, intuitive, and above all, fast. It can log in to multiple sources via its Router Windows Service and comes with both live and historical loggers. The machine learning developer needs to know what is going on inside their algorithm and ReflectSoftware has the richest logging abilities around. I encourage you to download your copy and try it out. You will never look at logging the same way again. Here is a screenshot to show you exactly what I mean:



I cannot stress enough on the importance and benefits of logging, especially in deep learning. For instrumenting Kelp.Net, and my choice for deep learning debugging in general is ReflectInsight (RI). Kelp.Net uses RI for its entire model testing so that you, the end user, can see what is going on as each model is tested and verified. Having this information can make a difference between identifying and solving complex problems quickly, or not!

Here are a few of the features and benefits of ReflectInsight:

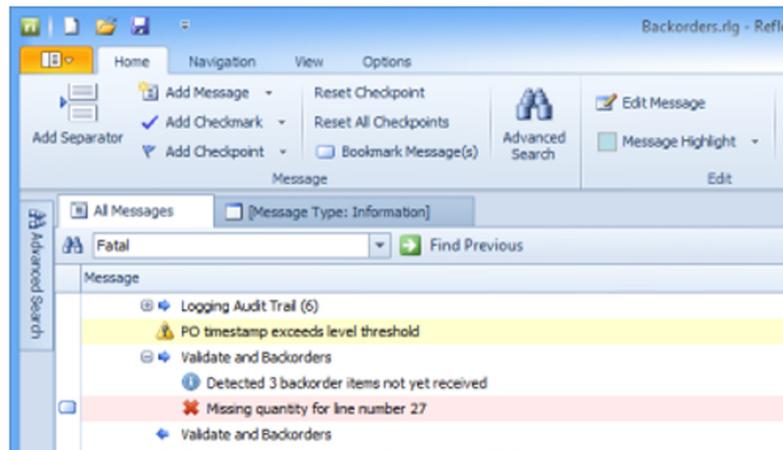
- Advance searching by using simple or regular expression criteria.
- Message filtering/categorization using multiple **User-Defined Views (UDV)**.
- Request and call stack traceability views.
- Bookmark, line, and message type highlighting.
- PostSharp AOP for automatic method and property tracing support.

- Message source-time to local or to any other UTC time toggle conversion.
- Rich visual indication of message types.
- Ability to handle very large number of messages.

Let us talk about some of the individual components of the system.

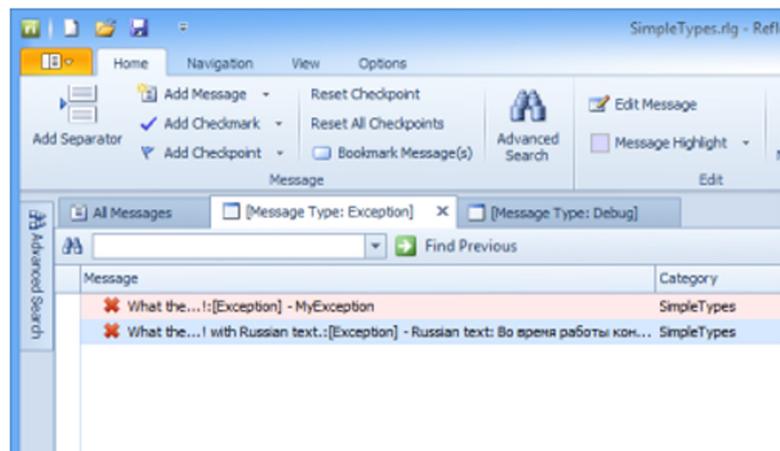
## Next generation logging viewers

ReflectInsight comes with both live and historical viewers that include a wide variety of tools specifically designed for analyzing ReflectInsight messages. High-performance logging allows you to monitor instrumented applications in real-time by displaying log messages in the live viewer. You can log incredibly rich details such as exceptions, objects, datasets, images, process and thread information, well-formatted XML, and so much more. You can quickly navigate and trace through applications to find the information you need with ease.



## Message log

The message log panel is the central area where all messages logged are displayed. This area provides the ability to create multiple **User-Defined Views (UDV)**. These views can be customized to display specific information that allows you to filter out unwanted noise and only focus on messages that are of interest to your immediate need:



## Message details

The message details panel displays the extended details of the selected message. The details can be as simple as the message itself or complex data such as an object, dataset, binary blob, image, process and thread Information, the contents of a collection, and so on. Syntax highlighting is available for select message types such as SQL, XML, and HTML-related messages. The full Unicode support is also included, as shown in the following screenshot:

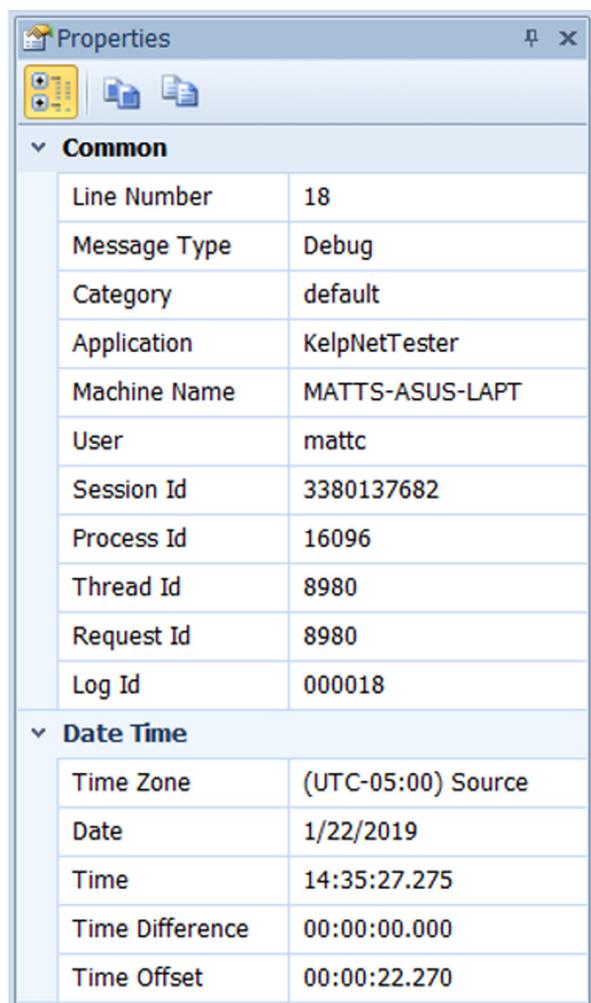
```

<?xml version="1.0" standalone="yes"?>

<!--<NewDataSet>
  <xs:schema id="NewDataSet" xmlns="" xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="NewDataSet" msdata:IsDataSet="1" msdata:Locale="en-US">
      <xs:complexType>
        <xs:choice minOccurs="0" maxOccurs="unbounded">
          <xs:element name="Table">
            <xs:complexType>
              <xs:sequence>
                <xs:element name="Table" type="xs:string"/>
              </xs:sequence>
            </xs:complexType>
          </xs:element>
        </xs:choice>
      </xs:complexType>
    </xs:element>
  </xs:schema-->
```

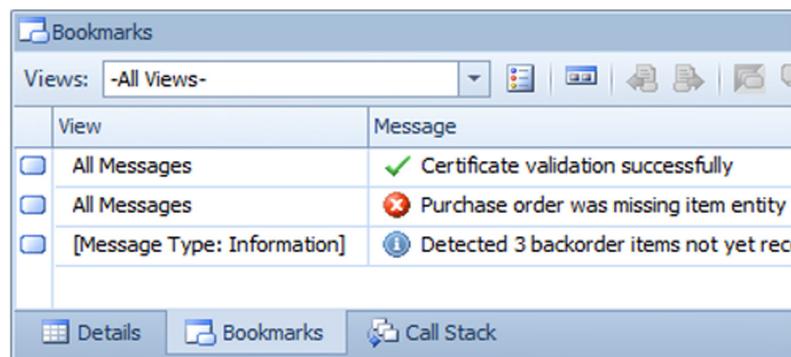
## Message properties

The message properties panel allows you to further inspect a selected message. We can view date/time values, time zone, process ID, thread ID, request ID, category, machine, and so on. We can also extend the message properties panel by programmatically attaching user-defined properties to one or more messages during our logging:



## Bookmarks

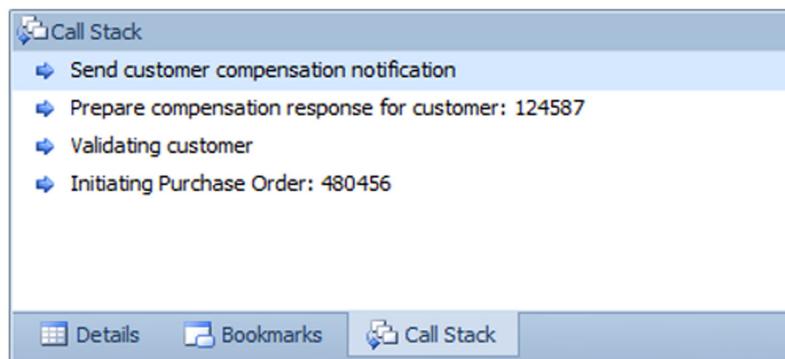
The **Bookmarks** panel allows you to view bookmarks for the current logging session and can be persisted with your log file for later retrieval. You can filter bookmarks for the active view, a given view, or see all bookmarks across all views. Navigating to any bookmark will immediately activate the view and select where the bookmarked message is located:



*Bookmarks can also be managed using the navigation ribbon menu and/or shortcut keys.*

## Call Stack

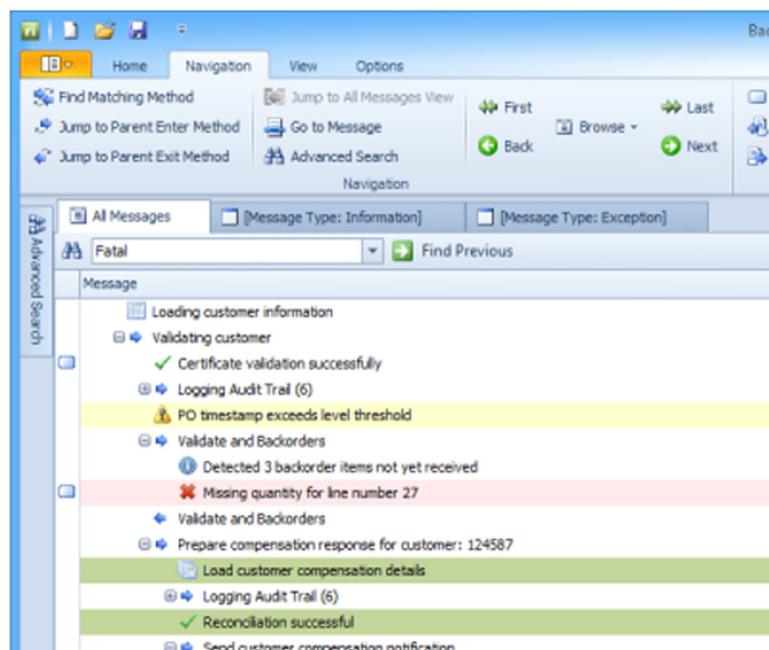
The **Call Stack** panel displays the call stack level of the current selected message. Call stack entries are generated programmatically using the enter/exit method, or if the message was contained within the `TraceMethod` using block. You can easily navigate the call stack by simply double-clicking on a call stack entry, which will then take you to the top of the enter/exit message block within the active message log panel:



## Message Navigation

ReflectInsight supports many ways in which you can navigate through your logged messages:

- Find matching enter/exit method block
- Jump to parent enter/exit method block
- Jump from any message in a user-defined view to the **All Messages** view
- **Go to Message** by line number
- **Advanced Search**
- Quick search (active view only)
- Message type browse navigator
- **Bookmarks**



## Advanced Search

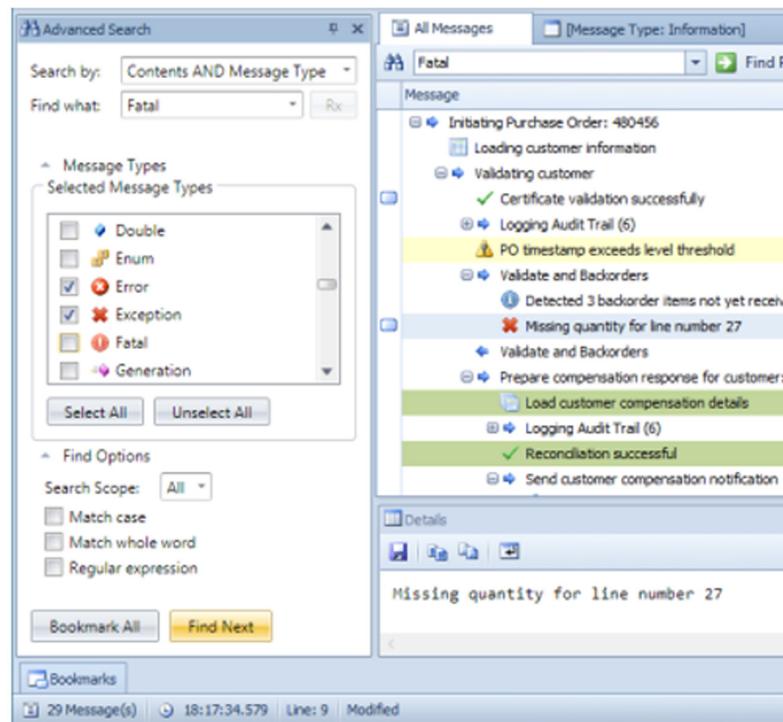
The viewer provides two ways to search messages by criteria:

- Quick search: This is mainly used for simple quick text-based searching.
- Advanced search: This is primarily used to search messages where a more complex search criterion is needed.

A search criterion can include a combination of the following:

- Message content
- Message type
- Message contents AND type
- Message contents OR type
- In addition to a regular expression

The **Advanced Search** view provides the ability to either navigate to the search result or bookmark it:

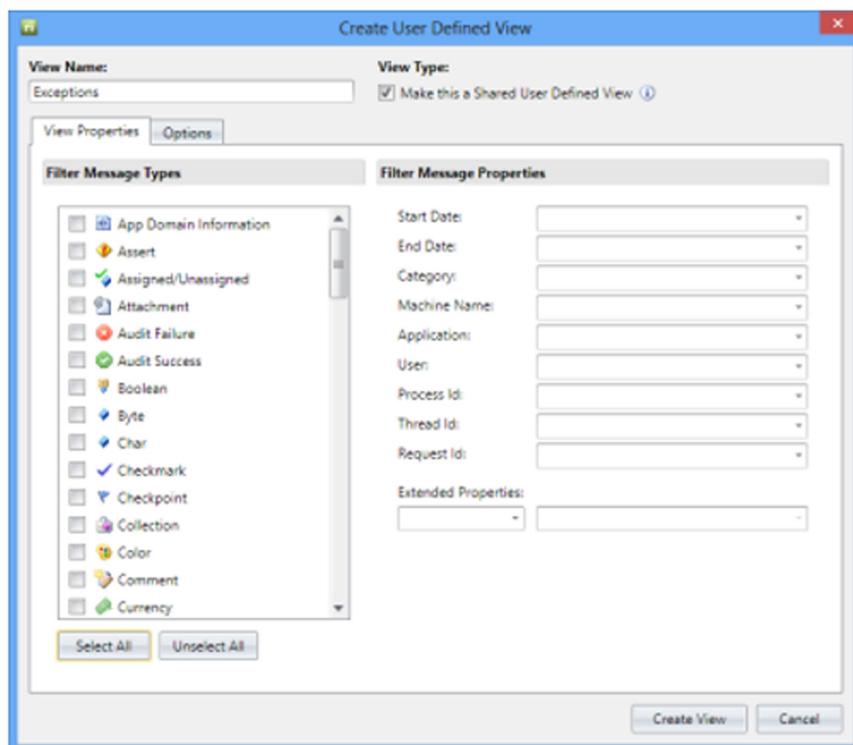


## User-Defined Views and Filtering

User-Defined Views (UDV) allows you to filter unwanted noise by only viewing messages that matter the most. You can create UDV's by applying one of the following methods:

- Clicking on a message and selecting either its message type or one of its properties.
- By creating a new UDV and specifying a filter criterion.
- By cloning an existing UDV.

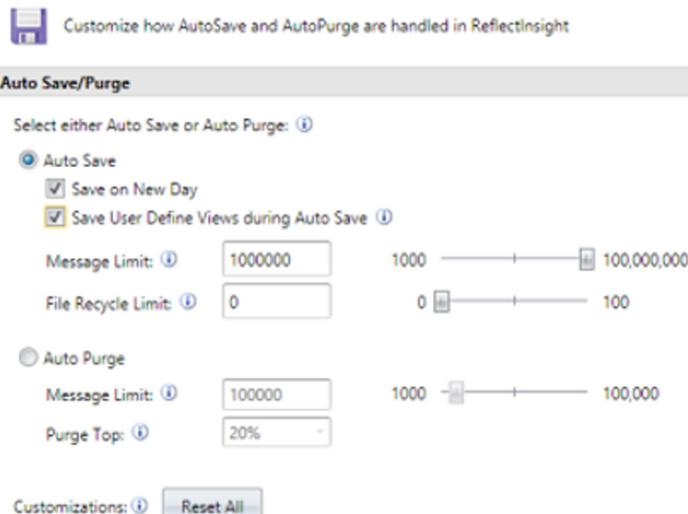
You also have the option to persist UDV's for the active log file.



## Auto Save/Purge rolling log files

Apart from the library's ability to **Auto Save/Purge** rolling log files, the live viewer has similar capabilities in addition to auto purging the top portion of a rolling log file. You can configure the live viewer to either **Auto Save/Auto Purge** by applying one of the following methods:

- **Auto Save:** This method forces the live viewer to save files once a specific criterion has been met (that is, on new day and/or message limit).
- **Auto Purge:** This method forces the live viewer to purge the top portion of the logged messages, based on predefined size percentages of the current log file.



## Watches

The watch panel allows you to programmatically track the values of many different variables throughout the lifetime of your application. In deep learning, once the algorithms start to do their magic, this panel becomes invaluable as insight into what is taking place.

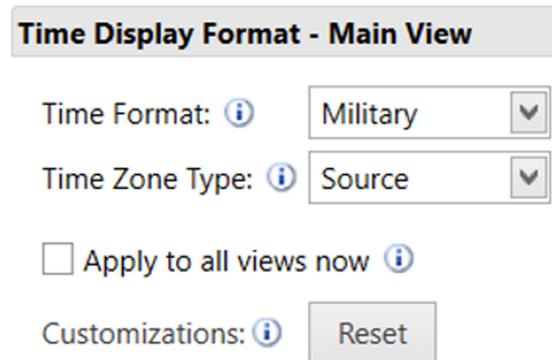
Label	Value
Prod Watch 6	asasdffasaadfd
Prod Watch 7	asasdffasaadfd
Prod Watch 8	7:53:17 AM
Prod Watch 9 {0:f}	2/7/2019 7:53:17 AM
Prod Watch 10 {0:N0} nanoseconds	571218797961642500
Prod Watch 11 {0:N0} ticks	5712187979616425
Prod Watch 12 {0:N2} seconds	571218797.961643
Prod Watch 13 {0:N2} minutes	9520313.29936071
Prod Watch 14 {0:N0} days, {1} hours, {2} minutes, {3} secon...	6611
Prod Outbound - Total CPU %	5
Prod Outbound - Free Disk Space	1.7TB of 1.8TB
DR Watch 1	asasdffasaadfd
DR Watch 2	asasdffasaadfd
DR Watch 3	asasdffasaadfd
DR Watch 4	asasdffasaadfd
DR Watch 5	asasdffasaadfd
DR Watch 6	asasdffasaadfd
DR Watch 7	asasdffasaadfd
DR Watch 8	7:53:17 AM
DR Watch 9 {0:f}	2/7/2019 7:53:17 AM
DR Watch 10 {0:N0} nanoseconds	571218797961642500
DR Watch 11 {0:N0} ticks	5712187979616425
DR Watch 12 {0:N2} seconds	571218797.961643
DR Watch 13 {0:N2} minutes	9520313.29936071
DR Watch 14 {0:N0} days, {1} hours, {2} minutes, {3} seconds	6611
DR Outbound - Total CPU %	13
DR Outbound - Free Disk Space	1.7TB of 1.8TB
Watch 3	Status 1 - Still More Status
Watch 4	Status 2 - More Status

Properties

Watches

Remote Connections

## Time zone formatting



We can display our time details in either standard or military time formats. Select the **Time Zone Type** that best suits your location such as **Source**, **Local**, **UTC**, or **Customize** by choosing from one of the available system time zones.

Now let us cover the four major components of the system:

- Router
- Log viewer
- Live viewer
- Configuration editor

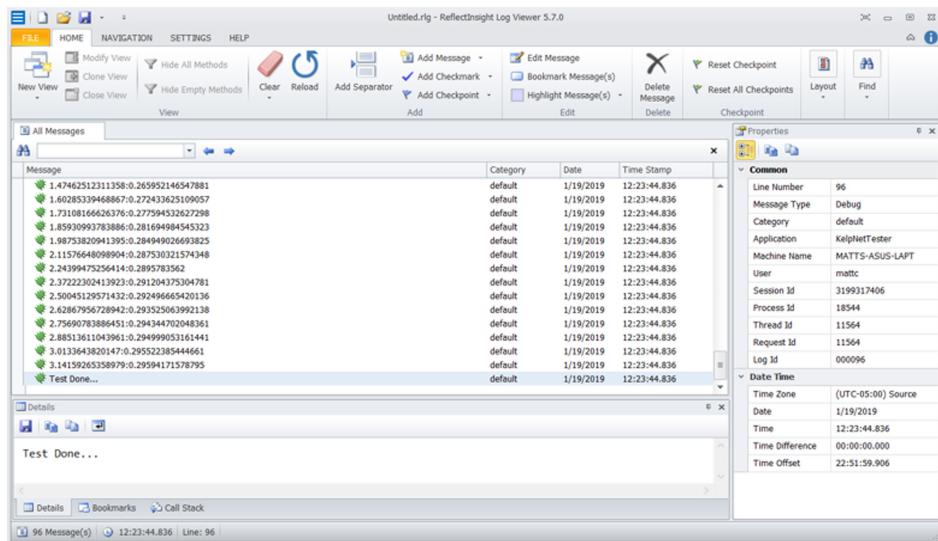
## Router

The router is the central part of the ReflectInsight (RI) logging system. This is a regular Windows service that constantly runs and directs all received log messages from various client libraries (bound to your application). These messages are then distributed to 'listeners' such as viewers, text files, binary files, event logs, databases, and so on. In a production environment, you would typically have the router installed on a separate machine from the rest of your logging system, but you certainly do not have to. Once installed and configured (the configuration out of the box is usually suitable for most situations), the router runs as a Windows service and has no user interface or desktop interaction.

## Log viewer

The log viewer is designed to view historical log files that have been saved either manually or automatically from the router/viewer configuration. If you are streaming a high number of messages through the system, you will no doubt collect a lot of log files which may need to be viewed. Historical logs can be invaluable when problems arise; I always set the system to keep the last seven days of log files just in case. I need to point out that, at the time of writing, watches and their values are not saved to historical files and these items are live streaming only.

The following screenshot is an example of the log viewer viewing a saved file:



## Live viewer

The live viewer is what you will use most of the time to view your real-time logging. The capabilities of the live viewer are extensive to say the least. If we take a look at the following screenshot, we can see that the amount of information to be gathered from our algorithms and applications is huge:

All Messages	Message	Machine	Application	User	Date	Time Stamp
Dropout	Forward [Cpu] : 5,287µs	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/23/2019	13:55:47.406
	Backward[Cpu] : 2,994µs	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/23/2019	13:55:47.406
Dropout		MATTS-ASUS-LAPT	KelpNetTester	mattc	1/23/2019	13:55:47.406
ArcSinH	Forward [Cpu] : 4,538µs	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/23/2019	13:55:47.406
	Backward[Cpu] : 2,657µs	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/23/2019	13:55:47.406
ArcSinH		MATTS-ASUS-LAPT	KelpNetTester	mattc	1/23/2019	13:55:47.406
ELU	Forward [Cpu] : 884µs	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/23/2019	13:55:47.440
	Backward[Cpu] : 1,848µs	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/23/2019	13:55:47.440
ELU		MATTS-ASUS-LAPT	KelpNetTester	mattc	1/23/2019	13:55:47.440
LeakyReLUShifted	Forward [Cpu] : 2,323µs	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/23/2019	13:55:47.440
	Backward[Cpu] : 2,872µs	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/23/2019	13:55:47.445
LeakyReLUShifted		MATTS-ASUS-LAPT	KelpNetTester	mattc	1/23/2019	13:55:47.445
LogisticFunction	Forward [Cpu] : 2,407µs	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/23/2019	13:55:47.445
	Backward[Cpu] : 2,854µs	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/23/2019	13:55:47.445
LogisticFunction		MATTS-ASUS-LAPT	KelpNetTester	mattc	1/23/2019	13:55:47.445
MaxMinusOne	Forward [Cpu] : 1,298µs	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/23/2019	13:55:47.445
	Backward[Cpu] : 3,122µs	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/23/2019	13:55:47.474
MaxMinusOne		MATTS-ASUS-LAPT	KelpNetTester	mattc	1/23/2019	13:55:47.474
ScaledELU	Forward [Cpu] : 3,741µs	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/23/2019	13:55:47.474
	Backward[Cpu] : 2,922µs	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/23/2019	13:55:47.474
ScaledELU		MATTS-ASUS-LAPT	KelpNetTester	mattc	1/23/2019	13:55:47.474
Sine	Forward [Cpu] : 4,508µs	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/23/2019	13:55:47.480
	Backward[Cpu] : 2,876µs	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/23/2019	13:55:47.480
Sine		MATTS-ASUS-LAPT	KelpNetTester	mattc	1/23/2019	13:55:47.480
Test Done...		MATTS-ASUS-LAPT	KelpNetTester	mattc	1/23/2019	13:55:47.480

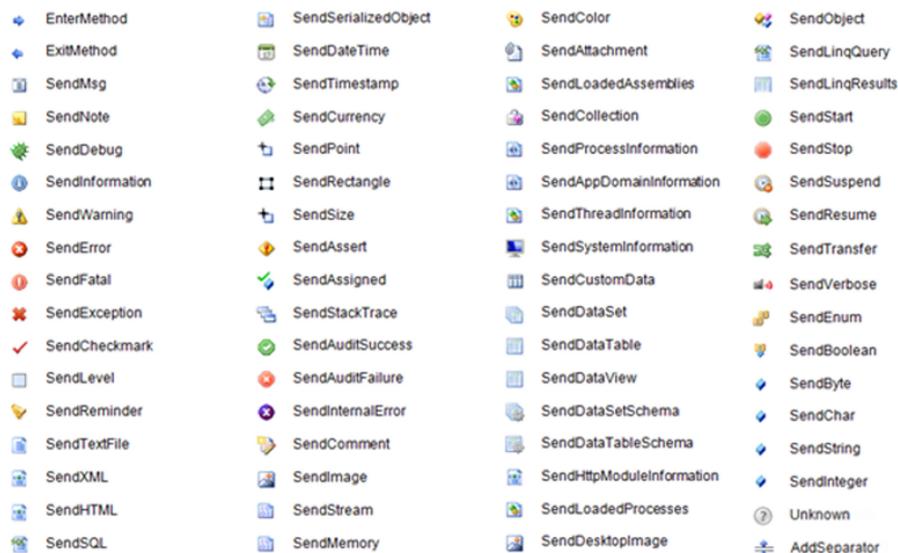
We have just mentioned how valuable a tool such as this can be when it comes to machine learning, so it is only fair that we show you exactly what we mean. The following screenshot shows the actual machine learning algorithm outputting data to the live viewer. Without this information in real-time, we would be lost to the efficacy and performance of our application!

All Messages	Message	Machine	Application	User	Date	Time Stamp
█ Create ComputeContext(2339339888256) in Thread(1).	MATT-S-USAS-LAPT	KelpNetTester	mattc		1/18/2019	13:31:44.930
█ Create ComputeCommandQueue(2339360277728) in Thread(1).	MATT-S-USAS-LAPT	KelpNetTester	mattc		1/18/2019	13:31:45.392
█ Training...	MATT-S-USAS-LAPT	KelpNetTester	mattc		1/18/2019	13:31:45.400
█ Test Start...	MATT-S-USAS-LAPT	KelpNetTester	mattc		1/18/2019	13:32:10.226
█ [ 2.26594662 -3.85293982]	MATT-S-USAS-LAPT	KelpNetTester	mattc		1/18/2019	13:32:10.231
█ 0 xor 0 = 0 [ 2.26594662 -3.85293982]	MATT-S-USAS-LAPT	KelpNetTester	mattc		1/18/2019	13:32:10.235
█ [-3.41753866 2.45219502]	MATT-S-USAS-LAPT	KelpNetTester	mattc		1/18/2019	13:32:10.235
█ 1 xor 0 = 1 [-3.41753866 2.45219502]	MATT-S-USAS-LAPT	KelpNetTester	mattc		1/18/2019	13:32:10.235
█ [-3.29385582 2.57134337]	MATT-S-USAS-LAPT	KelpNetTester	mattc		1/18/2019	13:32:10.235
█ 0 xor 1 = 1 [-3.29385582 2.57134337]	MATT-S-USAS-LAPT	KelpNetTester	mattc		1/18/2019	13:32:10.235
█ [ 2.34085338 -3.93792795]	MATT-S-USAS-LAPT	KelpNetTester	mattc		1/18/2019	13:32:10.235
█ 1 xor 1 = 0 [ 2.34085338 -3.93792795]	MATT-S-USAS-LAPT	KelpNetTester	mattc		1/18/2019	13:32:10.235
█ Test Start...	MATT-S-USAS-LAPT	KelpNetTester	mattc		1/18/2019	13:32:10.409
█ [ 2.26594662 -3.85293982]	MATT-S-USAS-LAPT	KelpNetTester	mattc		1/18/2019	13:32:10.516
█ 0 xor 0 = 0 [ 2.26594662 -3.85293982]	MATT-S-USAS-LAPT	KelpNetTester	mattc		1/18/2019	13:32:10.516
█ [-3.41753866 2.45219502]	MATT-S-USAS-LAPT	KelpNetTester	mattc		1/18/2019	13:32:10.516
█ 1 xor 0 = 1 [-3.41753866 2.45219502]	MATT-S-USAS-LAPT	KelpNetTester	mattc		1/18/2019	13:32:10.516
█ [-3.29385582 2.57134337]	MATT-S-USAS-LAPT	KelpNetTester	mattc		1/18/2019	13:32:10.516
█ 0 xor 1 = 1 [-3.29385582 2.57134337]	MATT-S-USAS-LAPT	KelpNetTester	mattc		1/18/2019	13:32:10.516
█ [ 2.34085338 -3.93792795]	MATT-S-USAS-LAPT	KelpNetTester	mattc		1/18/2019	13:32:10.516
█ 1 xor 1 = 0 [ 2.34085338 -3.93792795]	MATT-S-USAS-LAPT	KelpNetTester	mattc		1/18/2019	13:32:10.516
█ Test Done...	MATT-S-USAS-LAPT	KelpNetTester	mattc		1/18/2019	13:32:10.517
█ Create ComputeContext(2285891270160) in Thread(1).	MATT-S-USAS-LAPT	KelpNetTester	mattc		1/19/2019	12:23:13.914
█ Create ComputeCommandQueue(2285913778992) in Thread(1).	MATT-S-USAS-LAPT	KelpNetTester	mattc		1/19/2019	12:23:13.971
█ loss:0.240353813413972	MATT-S-USAS-LAPT	KelpNetTester	mattc		1/19/2019	12:23:14.235
█	MATT-S-USAS-LAPT	KelpNetTester	mattc		1/19/2019	12:23:14.235
█ loss:0.240353813413972	MATT-S-USAS-LAPT	KelpNetTester	mattc		1/19/2019	12:23:17.476
█	MATT-S-USAS-LAPT	KelpNetTester	mattc		1/19/2019	12:23:17.476
█ loss:0.240353813413972	MATT-S-USAS-LAPT	KelpNetTester	mattc		1/19/2019	12:23:20.446
█	MATT-S-USAS-LAPT	KelpNetTester	mattc		1/19/2019	12:23:20.446
█ loss:0.240353813413972	MATT-S-USAS-LAPT	KelpNetTester	mattc		1/19/2019	12:23:23.593
█	MATT-S-USAS-LAPT	KelpNetTester	mattc		1/19/2019	12:23:23.593
█ loss:0.240353813413972	MATT-S-USAS-LAPT	KelpNetTester	mattc		1/19/2019	12:23:26.612

## SDK

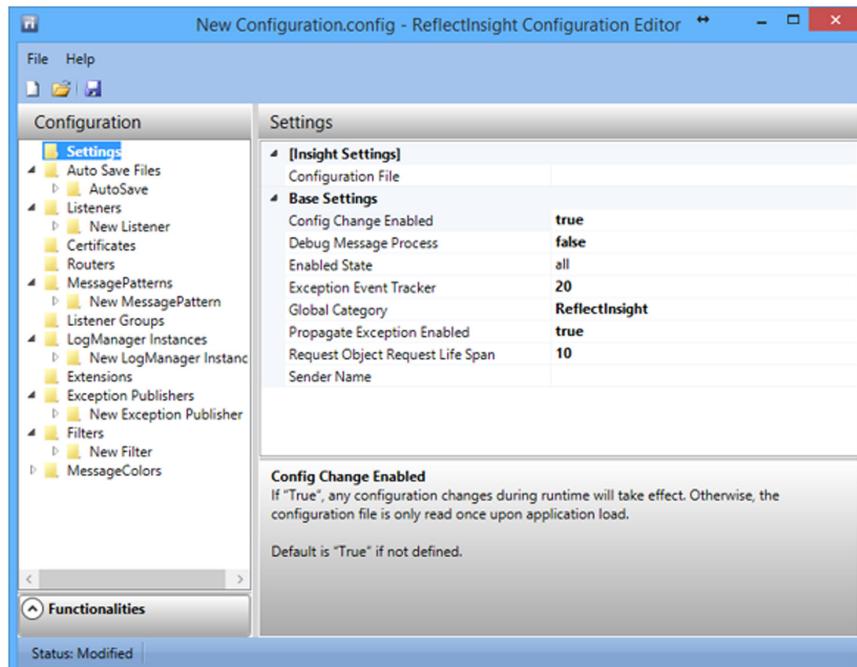
The **Software Development Kit** (SDK) is what allows you to connect ReflectInsight to your applications. We will see as we get further along just how easy it is to do this. The beautiful thing about how this SDK works compared to others is the rich image set assigned to each message. When you have thousands of messages streaming per second, colors and images can be a huge help in focusing your attention on just what you need to see.

The following is a screenshot that shows exactly what I mean. As an example, if we use the **SendException** message, then the red X will be displayed in the icon panel along the far-left-hand side of the live/log viewer. The same applies for all the other messages you see in the following screenshot. You can find more detailed information in the *Logging different message types* section of this chapter.



## Configuration editor

The configuration editor puts a graphical editor on top of the conventional XML file manipulation. It makes it much easier to modify your values using an intuitive graphical user interface as follows:



## Overview

We can use the XML-based configuration file with our applications to make the ReflectInsight viewer behave the way we want it to. There are a few configuration categories available from auto save and filtering, to message coloring, and many more.

## XML configuration

ReflectInsight is configured using an XML configuration file. The configuration information can be embedded within other XML configuration files such as the application or web .config file, or in a separate file. The configuration is easily readable and updateable while retaining the flexibility to express all configurations.

Alternatively, ReflectInsight can be configured programmatically. We will use a combination of both throughout this book, with the main configuration usually done via the app.config file.

## Dynamic configuration

ReflectInsight automatically monitors its configuration file for changes and dynamically applies these changes when made. In many cases, it is possible to diagnose application issues without terminating the process in question. This can be a very valuable tool in investigating issues from our deployed applications.

## Configuration editor

The ReflectInsight's configuration editor helps in easily creating config files through the visual interface, but advance users can work with XML.

The tool is very useful for editing settings, defining message patterns/formats, defining extensions, defining listeners, associating colors with message types, and many more.

- Easy to understand layout
- Remembers recent files list
- Pre-defined selections and dynamic section lookups
- Key values pop-up editor
- Message pattern pop-up editor
- Method types popup editor
- Color definition and message color pop-up editor

## Extensions

No doubt that you currently have some kind of logging framework in use. That framework is the result of countless hours of integration work. Wouldn't it be great if we did not have to rewrite our code base just to use a new tool? As fate would have it, ReflectSoftware developed an open-source set of library extensions that are simple to implement with no code change. It is just a matter of configuration.

These extensions allow you to see your logging messages in real-time in the live and log viewers. In addition, you can easily develop your own custom extensions as well.

Here are a few of the extensions currently available at the time of writing this book:

-  .NET Diagnostic Trace/Debug
-  HTTP Module
-  Enterprise Library - Logging Application Block
-  Enterprise Library - Semantic Logging Application Block
-  SourceForge - Common.Logging
-  Log4net
-  NLog
-  PostSharp

## Message type logging reference

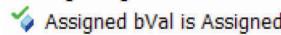
For your convenience, since there is not a lot of documentation as to how to work with many of the message types that RI supports, I have put together this reference section that will show you how to use the logging function. I have organized it by the message type for easy quick reference. You can take a look in the model tests section and you will find the fully functional code for what you see in the RITest.cs file.

## Assertions

Adding Assert  
💡 1 != 2

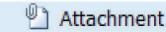
```
RILogManager.Get("Test").SendAssert(1 == 2, "1 != 2");
RILogManager.Get("Test").SendAssert(1 == 1, "1 = 1");
```

## Assigned variables



```
bool bVal = false;
RILogManager.Get("Test").SendAssigned("Assigned bVal", bVal);
```

## Attachments



```
RILogManager.Get("Test").SendAttachment("Attachment", @"C:\Program Files\ReflectSoftware\ReflectInsight\Bin\License.rtf");
```

## Audit failure and success

- ✉️ Adding Audit Failure
- ✖️ Audit Failure
- ➡️ Adding Audit Failure
- ✉️ Adding Audit Success
- ✅ Audit Success
- ➡️ Adding Audit Success

```
RILogManager.Get("Test").SendAuditFailure("Audit Failure");
RILogManager.Get("Test").SendAuditSuccess("Audit Success");
```

## Checkmarks

- ✓ Checkmark

```
RILogManager.Get("Test").SendCheckmark("Checkmark", Common.Checkmark.Blue);
RILogManager.Get("Test").SendCheckmark("Checkmark", Common.Checkmark.Green);
RILogManager.Get("Test").SendCheckmark("Checkmark", Common.Checkmark.Orange);
RILogManager.Get("Test").SendCheckmark("Checkmark", Common.Checkmark.Purple);
RILogManager.Get("Test").SendCheckmark("Checkmark", Common.Checkmark.Red);
RILogManager.Get("Test").SendCheckmark("Checkmark", Common.Checkmark.Yellow);
```

## Checkpoints

```
RILogManager.Get("Test").AddCheckpoint(ReflectSoftware.Insight.Common.Checkpoint.Blue);
RILogManager.Get("Test").AddCheckpoint(ReflectSoftware.Insight.Common.Checkpoint.Green);
RILogManager.Get("Test").AddCheckpoint(ReflectSoftware.Insight.Common.Checkpoint.Orange);
RILogManager.Get("Test").AddCheckpoint(ReflectSoftware.Insight.Common.Checkpoint.Purple);
RILogManager.Get("Test").AddCheckpoint(ReflectSoftware.Insight.Common.Checkpoint.Red);
RILogManager.Get("Test").AddCheckpoint(ReflectSoftware.Insight.Common.Checkpoint.Yellow);
```

## Collections

Adding Collection

Hash Table - No Scope	MATTS-ASUS-LAPT	KelpNetTester	mattc	2
String Collection - Scope All	MATTS-ASUS-LAPT	KelpNetTester	mattc	2
String Collection - Scope Private	MATTS-ASUS-LAPT	KelpNetTester	mattc	2
String Collection - Scope Protected	MATTS-ASUS-LAPT	KelpNetTester	mattc	2
String Collection - Scope Protected	MATTS-ASUS-LAPT	KelpNetTester	mattc	2

Details

System.Collections.Hashtable

Key	Value	Type
Enju	351-8765	System.String
Leah	922-5699	System.String
Nelly	110-5699	System.String
James	010-4077	System.String
Molly	221-5678	System.String
John	123-4567	System.String

```
List<string> sList = new List<string>();
sList.Add("String 1");
sList.Add("string 2");
sList.Add("string 3");

Hashtable phones = new Hashtable();
phones.Add("John", "123-4567");
phones.Add("Enju", "351-8765");
phones.Add("Molly", "221-5678");
phones.Add("James", "010-4077");
```

```
phones.Add("Nelly", "110-5699");
phones.Add("Leah", "922-5699");

RILogManager.Get("Test").SendCollection("Hash Table - No Scope", phones);
RILogManager.Get("Test").SendCollection("String Collection - Scope All", sList.
AsEnumerable<string>(), ReflectSoftware.Insight.Common.ObjectScope.All);
RILogManager.Get("Test").SendCollection("String Collection - Scope Private", sList.
AsEnumerable<string>(), ReflectSoftware.Insight.Common.ObjectScope.Private);
RILogManager.Get("Test").SendCollection("String Collection - Scope Protected", sList.
AsEnumerable<string>(), ReflectSoftware.Insight.Common.ObjectScope.Protected);
RILogManager.Get("Test").SendCollection("String Collection - Scope Protected", sList.
AsEnumerable<string>(), ReflectSoftware.Insight.Common.ObjectScope.Public);
RILogManager.Get("Test").SendCollection("String Collection - Scope None", sList.
AsEnumerable<string>(), ReflectSoftware.Insight.Common.ObjectScope.None);
```

## Comments

-  Comment 1
-  Comment 2 A B

```
RILogManager.Get("Test").SendComment("Comment 1");
RILogManager.Get("Test").SendComment("Comment 2 {0} {1}", "A", "B");
```

## Currency

-  Currency = \$10.00
-  Currency (no value) = \$0.00

```
decimal dVal1 = 10.0M;
decimal? dVal2 = null;
RILogManager.Get("Test").SendCurrency("Currency", dVal1);

RILogManager.Get("Test").SendCurrency("Currency (no value)", dVal2.HasValue ? dVal2.Value :
0.0M);
```

Data

		MATTSS-ASUS-LAPT	KelpNetTester	mattc	2/7/2019	07:37:25.807
↳	Adding DataSet	MATTSS-ASUS-LAPT	KelpNetTester	mattc	2/7/2019	07:37:25.807
↳	DataSet	MATTSS-ASUS-LAPT	KelpNetTester	mattc	2/7/2019	07:37:25.807
↳	Adding DataSet	MATTSS-ASUS-LAPT	KelpNetTester	mattc	2/7/2019	07:37:25.807
↳	Adding DataSet Schema	MATTSS-ASUS-LAPT	KelpNetTester	mattc	2/7/2019	07:37:25.807
↳	DataSet Schema	MATTSS-ASUS-LAPT	KelpNetTester	mattc	2/7/2019	07:37:25.807
↳	Test DataSet	MATTSS-ASUS-LAPT	KelpNetTester	mattc	2/7/2019	07:37:25.807
↳	Adding DataSet Schema	MATTSS-ASUS-LAPT	KelpNetTester	mattc	2/7/2019	07:37:25.807
↳	Adding DataTable	MATTSS-ASUS-LAPT	KelpNetTester	mattc	2/7/2019	07:37:25.807
↳	DataTable	MATTSS-ASUS-LAPT	KelpNetTester	mattc	2/7/2019	07:37:25.807
↳	Test DataTable	MATTSS-ASUS-LAPT	KelpNetTester	mattc	2/7/2019	07:37:25.807
↳	Adding DataTable	MATTSS-ASUS-LAPT	KelpNetTester	mattc	2/7/2019	07:37:25.807
↳	Adding DataTable Schema	MATTSS-ASUS-LAPT	KelpNetTester	mattc	2/7/2019	07:37:25.807
↳	DataTable Schema	MATTSS-ASUS-LAPT	KelpNetTester	mattc	2/7/2019	07:37:25.807
↳	Test DataTable	MATTSS-ASUS-LAPT	KelpNetTester	mattc	2/7/2019	07:37:25.807
↳	Adding DataTable Schema	MATTSS-ASUS-LAPT	KelpNetTester	mattc	2/7/2019	07:37:25.807
↳	Adding DataView	MATTSS-ASUS-LAPT	KelpNetTester	mattc	2/7/2019	07:37:25.807
↳	DataTable	MATTSS-ASUS-LAPT	KelpNetTester	mattc	2/7/2019	07:37:25.807
↳	Test DataView	MATTSS-ASUS-LAPT	KelpNetTester	mattc	2/7/2019	07:37:25.807
↳	Adding DataView	MATTSS-ASUS-LAPT	KelpNetTester	mattc	2/7/2019	07:37:25.807
↳	Adding DateTime	MATTSS-ASUS-LAPT	KelpNetTester	mattc	2/7/2019	07:37:25.807

```
DataSet ds = new DataSet("Test DataSet");
DataTable dt = new DataTable("Test DataTable");

dt.Columns.Add("Col1");
dt.Columns.Add("Col2");
dt.Columns.Add("Col3");
dt.Columns.Add("Col4");

for (int x = 0; x < 25; x++)
{
    DataRow drNew = dt.NewRow();
    drNew[0] = (x + 1).ToString();
    drNew[1] = (x + 2).ToString();
    drNew[2] = (x + 3).ToString();
    drNew[3] = (x + 4).ToString();
    dt.Rows.Add(drNew);
}
ds.Tables.Add(dt);
```

## DataSet

```
RILogManager.Get("Test").SendDataSet("DataSet", ds);
```

## DataSetSchema

```
RILogManager.Get("Test").SendDataSetSchema("DataSet Schema", ds);  
RILogManager.Get("Test").SendDataSetSchema(ds);
```

## DataTable

```
RILogManager.Get("Test").SendDataTable("DataTable", dt);  
RILogManager.Get("Test").SendDataTable(dt);
```

## DataTableSchema

```
RILogManager.Get("Test").SendDataTableSchema("DataTable Schema", dt);
```

## DataView

```
RILogManager.Get("Test").SendDataView(dt.DefaultView);  
RILogManager.Get("Test").SendDataView("DataView", dt.DefaultView);
```

## Date/Time

```
DateTime = 2/7/2019 7:37:25 AM  
DateTime (null) = null  
DateTime (Current Culture) = 2/7/2019 7:37:25 AM  
DateTime (Current Culture with null date) = null
```

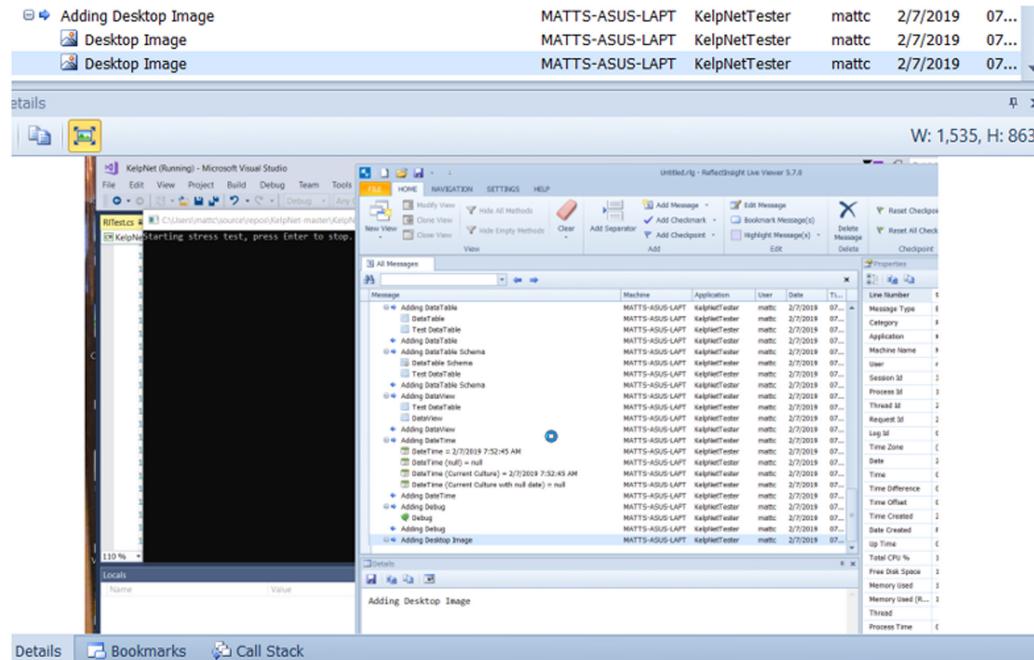
```
RILogManager.Get("Test").SendDateTime("DateTime", DateTime.Now);  
RILogManager.Get("Test").SendDateTime("DateTime (null)", null);  
  
RILogManager.Get("Test").SendDateTime("DateTime (Current Culture)", DateTime.Now, System.  
Globalization.CultureInfo.CurrentCulture);  
RILogManager.Get("Test").SendDateTime("DateTime (Current Culture with null date)", null,  
System.Globalization.CultureInfo.CurrentCulture);
```

## Debug



```
RILogger.Get("Test").SendDebug("Debug", dVal1);
```

## Desktop Image



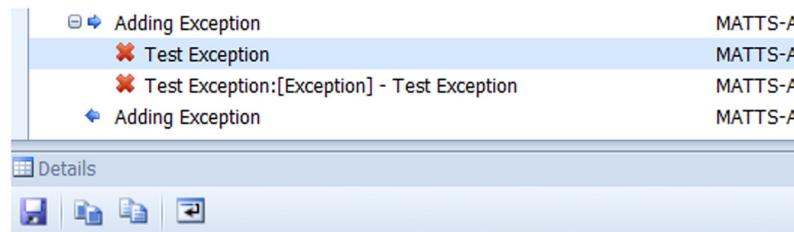
```
RILogger.Get("Test").SendDesktopImage("Desktop Image");
```

## Errors

- ✖ Error
- ✖ Error False

```
RILogger.Get("Test").SendError("Error {0}", false);
```

## Exceptions



### Test Exception

The following exception occurred:

#### General Information

#### Additional Info:

```
TrackingId: a1aa1758-2ad3-4217-837d-a71d93291078
MachineName: MATTSS-ASUS-LAPT
UTC: 2/7/2019 12:52:48 PM
```

#### 1) Exception Information

```
Exception Type: System.Exception
Message: Test Exception
Data: System.Collections.ListDictionaryInternal
TargetSite: NULL
HelpLink: NULL
```

```
Exception ex = new Exception("Test Exception");
RILogManager.Get("Test").SendException(ex);
RILogManager.Get("Test").SendException("Test Exception", ex);
```

## Fatal Errors

- Fatal
- Fatal 1

```
Exception ex = new Exception("Test Exception");
RILogManager.Get("Test").SendFatal("Fatal {0}", ex, 1);
```

## Generations

≡ethyst Generation = 0  
≡ethyst Generation = 2

```
RILogManager.Get("Test").SendGeneration("Generation", ex);  
RILogManager.Get("Test").SendGeneration("Generation", new WeakReference(phones, true));
```

## Images

```
RILogManager.Get("Test").SendImage("Icon", @"C:\Program Files\ReflectSoftware\ReflectInsight\Bin\ri_logo_green.ico");
```

## Information

ⓘ Information 1, 2, 3

```
RILogManager.Get("Test").SendInformation("Information {0}, {1}, {2}", 1, 2, 3);
```

## Levels



```
RILogManager.Get("Test").SendLevel("Level", LevelType.Blue);  
RILogManager.Get("Test").SendLevel("Level", LevelType.Cyan);  
RILogManager.Get("Test").SendLevel("Level", LevelType.Green);  
RILogManager.Get("Test").SendLevel("Level", LevelType.Magenta);  
RILogManager.Get("Test").SendLevel("Level", LevelType.Orange);  
RILogManager.Get("Test").SendLevel("Level", LevelType.Purple);  
RILogManager.Get("Test").SendLevel("Level", LevelType.Red);  
RILogManager.Get("Test").SendLevel("Level", LevelType.Yellow);
```

## Linq queries and results

⊕ ↗ Adding IQueryable  
 ⓘ This is message 1 1185523708  
 ⓘ this is another long message 1185523708  
 ⓘ this is an even longer message 1185523708  
 ⓘ one more long message about nothing 1185523708  
 ⓘ generating test data for current process 1185523708  
 ⓘ updating all watches 1185523708

```
List<string> fruits = new List<string> { "apple", "passionfruit", "banana", "mango", "orange",  

    "blueberry", "grape", "strawberry" };  

RILogManager.Get("Test").EnterMethod("Adding IQueryable");  

RILogManager.Get("Test").SendLinqQuery(fruits.AsQueryable().Where(f => f.Length < 6));  

RILogManager.Get("Test").SendLinqQuery("IQueryable", fruits.AsQueryable().Where(f => f.Length  

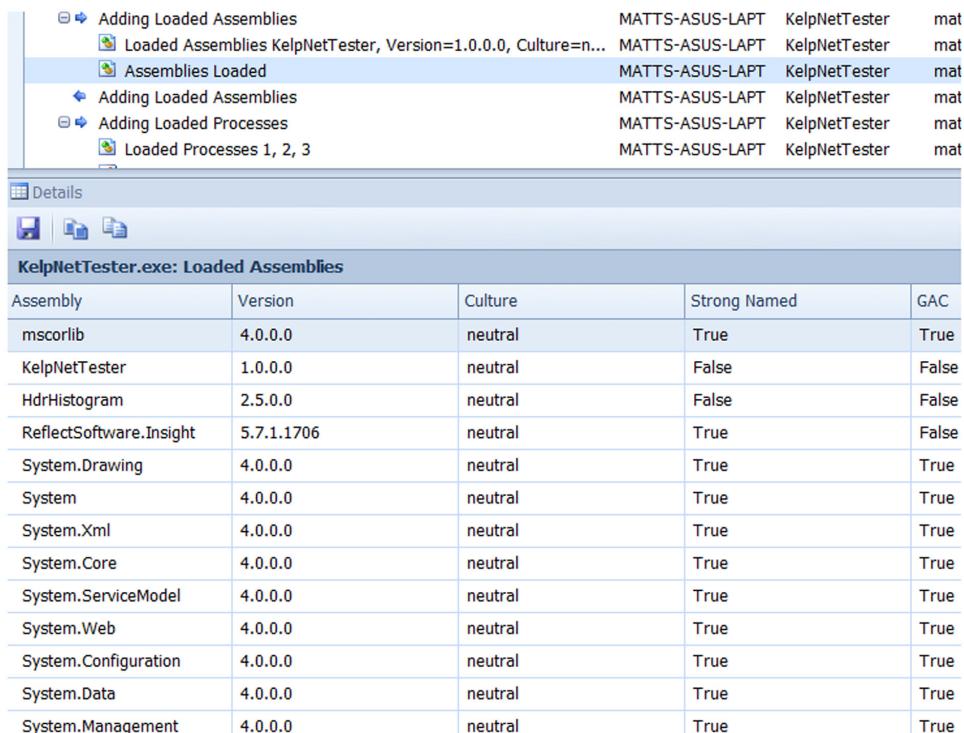
< 6));  

RILogManager.Get("Test").SendLinqResults(fruits.AsQueryable().Where(fruit => fruit.Length < 6));  

RILogManager.Get("Test").SendLinqResults("IQueryable Results", fruits.AsQueryable().Where(f =>  

f.Length < 6));
```

## Loaded assemblies



Assembly	Version	Culture	Strong Named	GAC
mscorlib	4.0.0.0	neutral	True	True
KelpNetTester	1.0.0.0	neutral	False	False
HdrHistogram	2.5.0.0	neutral	False	False
ReflectSoftware.Insight	5.7.1.1706	neutral	True	False
System.Drawing	4.0.0.0	neutral	True	True
System	4.0.0.0	neutral	True	True
System.Xml	4.0.0.0	neutral	True	True
System.Core	4.0.0.0	neutral	True	True
System.ServiceModel	4.0.0.0	neutral	True	True
System.Web	4.0.0.0	neutral	True	True
System.Configuration	4.0.0.0	neutral	True	True
System.Data	4.0.0.0	neutral	True	True
System.Management	4.0.0.0	neutral	True	True

```
RILogManager.Get("Test").SendLoadedAssemblies("Loaded Assemblies {0}", Assembly.  
GetEntryAssembly().FullName);  
RILogManager.Get("Test").SendLoadedAssemblies();
```

## Loaded processes

Image Name	Mem Usage	Virtual Mem Usage	Peak Mem Usage	Thread Count
ACMON	476 KB	2,928 KB	32,324 KB	1
AppleMobileDeviceProcess	10,628 KB	3,704 KB	12,896 KB	8
ApplicationFrameHost	29,152 KB	32,960 KB	42,132 KB	5
AsLdrSrv	5,064 KB	1,524 KB	6,036 KB	5
AsPatchTouchPanel64	268 KB	1,296 KB	5,688 KB	1
ASUSHelloBG	340 KB	1,700 KB	6,832 KB	1
ATKOSD2	7,464 KB	1,960 KB	9,036 KB	2
audiogd	20,536 KB	19,440 KB	31,128 KB	4
bdagent	5,544 KB	23,600 KB	34,724 KB	24
bdredline	7,652 KB	3,020 KB	8,684 KB	2
bdservicehost	13,548 KB	5,352 KB	17,700 KB	20
bdservicehost	35,260 KB	15,876 KB	40,384 KB	33
BdVpnApp	14,168 KB	5,400 KB	17,232 KB	16

```
RILogManager.Get("Test").SendLoadedProcesses("Loaded Processes {0}, {1}, {2}", 1, 2, 3);  
RILogManager.Get("Test").SendLoadedProcesses();
```

## Memory status

Max Generation	2
Total Memory	3,889 KB
Memory Load	48%
Total Physical	16,657,528 KB
Total Page File	22,686,840 KB
Total Virtual	137,438,953,344 KB
Available Physical	8,554,640 KB
Available Page File	11,963,264 KB
Available Virtual	137,434,010,780 KB

```
RILogManager.Get("Test").SendMemoryStatus("Memory Status");
RILogManager.Get("Test").SendMemoryStatus();
```

## Messages

-  Test Message 1
-  Test Message 2
-  Test Message 3 A B C

```
RILogManager.Get("Test").SendMessage("Test Message 1");
RILogManager.Get("Test").SendMessage("Test Message 2", "A", "B", "C");
RILogManager.Get("Test").SendMessage("Test Message 3 {0} {1} {2}", "A", "B", "C")
```

## Notes

-  Note 1
-  Note 2
-  Note 3 A B C

```
RILogManager.Get("Test").SendNote("Note 1");
RILogManager.Get("Test").SendNote("Note 2", "A", "B", "C");
RILogManager.Get("Test").SendNote("Note 3 {0} {1} {2}", "A", "B", "C");
```

## Objects

-  Object 2 = (Hashtable)
-  Object 3 (Scope All) = (Hashtable)
-  Object 3 (Scope Private) = (Hashtable)
-  Object 3 (Scope Protected) = (Hashtable)
-  Object 3 (Scope Public) = (Hashtable)

```
Hashtable phones = new Hashtable();
phones.Add("John", "123-4567");
phones.Add("Enju", "351-8765");
phones.Add("Molly", "221-5678");
phones.Add("James", "010-4077");
phones.Add("Nelly", "110-5699");
phones.Add("Leah", "922-5699");
RILogManager.Get("Test").SendObject("Object 1", phones);
RILogManager.Get("Test").SendObject("Object 2", phones, true);
RILogManager.Get("Test").SendObject("Object 3 (Scope All)", phones, ObjectScope.All);
RILogManager.Get("Test").SendObject("Object 3 (Scope Private)", phones, ObjectScope.Private);
```

```
RILogManager.Get("Test").SendObject("Object 3 (Scope Protected)", phones, ObjectScope.Protected);
RILogManager.Get("Test").SendObject("Object 3 (Scope Public)", phones, ObjectScope.Public);
```

## Process Information

The screenshot shows a software interface for viewing process information. At the top, there's a header bar with tabs for 'Details' and other icons. Below it is a section titled 'Process Information' with a tree view. Under 'Process Information', there are two expanded sections: 'Process Information' and 'Priority Information'. The 'Process Information' section contains the following data:

Process Id	13772
Process Name	KelpNetTester
Image	C:\Users\mattc\source\repos\KelpNet-master\KelpNetTester\bin\Debug\KelpNetTester.exe
Machine Name	MATTS-ASUS-LAPT
Main Window Handle	201142
Main Window Title	C:\Users\mattc\source\repos\KelpNet-master\KelpNetTester\bin\Debug\KelpNetTester.exe
Handle	1716
Open Handles	424
Number of Modules	70

The 'Priority Information' section contains the following data:

Base Priority	8
Priority	Normal
Priority Boost	True

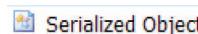
```
RILogManager.Get("Test").SendProcessInformation(System.Diagnostics.Process.GetCurrentProcess());
```

## Reminders

💡 This is Reminder 2 Don't do this

```
RILogManager.Get("Test").SendReminder("This is Reminder 2 {0}", "Don't do this");
```

## Serialized Objects



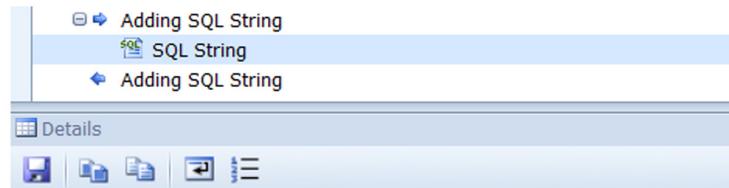
```
[Serializable()]
public class Employee : ISerializable
{
    public int EmplId;
```

```

    public string EmpName;
}
RILogManager.Get("Test").SendSerializedObject("Serialized Object", new Employee());

```

## SQL strings



WITH DataBase\_Size (SqlServerInstance,DatabaseName

```

readonly static string strSQLQuery = "WITH DataBase_Size (SqlServerInstance,DatabaseN
ame,DatabaseSize,LogSize,TotalSize) " + "AS (SELECT @@SERVERNAME SqlServerInstance,
db.name AS DatabaseName, SUM(CASE WHEN af.groupid = 0 THEN 0 ELSE af.size / 128.0E
END) AS DatabaseSize, " + "SUM(CASE WHEN af.groupid = 0 THEN af.size / 128.0E ELSE 0 END)
AS LogSize, SUM(af.size / 128.0E) AS TotalSize FROM master..sysdatabases AS db " + "INNER
JOIN master..sysaltfiles AS af ON af.[dbid] = db.[dbid] WHERE db.name NOT IN ('distribution',
'Resource', 'master', 'tempdb', 'model', 'msdb') " + "AND db.name NOT IN ('Northwind', 'pubs',
'AdventureWorks', 'AdventureWorksDW') GROUP BY db.name) " + "SELECT * FROM DataBase_
Size order by TotalSize desc";

```

```
RILogManager.Get("Test").SendSQLString("SQL String", strSQLQuery);
```

## Stack Traces



```

try
{
    RILogManager.Get("Test").SendStream("Stream (byte)", File.ReadAllBytes(@"C:\License.rtf"));
}
catch (Exception ex8)
{
    RILogManager.Get("Test").SendStackTrace("Stack Trace");
}

```

# Streams

```
RILogManager.Get("Test").SendStream("Stream (byte)", File.ReadAllBytes(@"C:\License.rtf"));
RILogManager.Get("Test").SendStream("Stream (Stream)", File.OpenRead(@"C:\ License.rtf"));
RILogManager.GetCurrentClassLogger().SendStream("Stream (Filename)", @"C:\License.rtf");
```

## System Information

The screenshot shows the ReflectInsight configuration interface. At the top, there's a navigation bar with icons for adding system information and a back arrow. Below it is a toolbar with icons for details, file operations, and a search bar. The main area is titled "System Information" and contains three expandable sections: "Environment", "Platform", and "Screen Information".

Computer Name	MATTS-ASUS-LAPT
User Name	mattc

OS	Longhorn
CPU Size	64 bits
Version	6.2
Build Number	9200
Revision	0

Resolution	1536 x 864
Color	True Color (32-bit)
Pixels Per Inch	120
Small Fonts	False

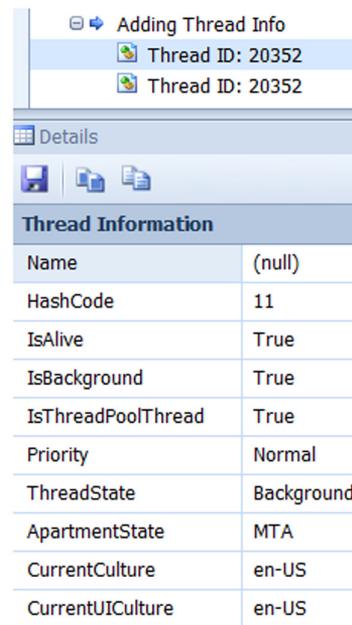
```
RILogManager.Get("Test").SendSystemInformation();
```

## Text files

The screenshot shows the ReflectInsight configuration interface. At the top, there's a navigation bar with icons for adding text files and a back arrow. Below it is a toolbar with icons for file operations. The main area is titled "Text File" and contains three options: "Text File (stream)", "Text File (stream)", and "Text File (FileName)".

```
RILogManager.Get("Test").SendTextFile("Text File (stream)", File.OpenRead(@"C:\Program Files\ReflectSoftware\ReflectInsight\Bin\License.rtf"));
RILogManager.Get("Test").SendTextFile("Text File (FileName)", @"C:\Program Files\ReflectSoftware\ReflectInsight\Bin\License.rtf");
```

## Thread Information



The screenshot shows a log viewer interface with two entries for 'Adding Thread Info'. Both entries show 'Thread ID: 20352'. Below this, there is a 'Details' section with icons for file operations. Under 'Thread Information', there is a table with the following data:

Name	(null)
HashCode	11
IsAlive	True
IsBackground	True
IsThreadPoolThread	True
Priority	Normal
ThreadState	Background
ApartmentState	MTA
CurrentCulture	en-US
CurrentUICulture	en-US

```
RILoggerManager.Get("Test").SendThreadInformation();
RILoggerManager.Get("Test").SendThreadInformation(Thread.CurrentThread);
```

## Time-zone information

- ⌚ Timestamp: 2019-52-07T07:52:50.945 (UTC-05:00) Eastern Time (US & Canada)
- ⌚ TimeStamp (TimeZone): 2019-52-07T07:52:50.945 (UTC-05:00) Eastern Time (US & Canada)
- ⌚ TimeStamp (TimeZoneId): 2019-52-07T07:52:50.945 (UTC-05:00) Eastern Time (US & Canada)
- ⌚ Timestamp: 2019-52-07T07:52:50.945 (UTC-05:00) Eastern Time (US & Canada)
- ⌚ TimeStamp (TimeZone): 2019-52-07T07:52:50.945 (UTC-05:00) Eastern Time (US & Canada)
- ⌚ TimeStamp (TimeZoneId): 2019-52-07T07:52:50.945 (UTC-05:00) Eastern Time (US & Canada)
- ⌚ TimeStamp (String): 2019-52-07T07:52:50.945 (UTC-05:00) Eastern Time (US & Canada)
- ⌚ TimeStamp (String): 2019-52-07T07:52:50.945 (UTC-05:00) Eastern Time (US & Canada)

```
RILoggerManager.Get("Test").SendTimestamp();
RILoggerManager.Get("Test").SendTimestamp("TimeStamp (TimeZone)", TimeZoneInfo.Local);
RILoggerManager.Get("Test").SendTimestamp("TimeStamp (TimeZoneld)", TimeZoneInfo.Local.Id);
RILoggerManager.Get("Test").SendTimestamp("TimeStamp (String)");
```

## Typed collections

```
List<IEnumerable> ienums = new List<IEnumerable>();
ienums.Add(phones);
ienums.Add(sList.AsEnumerable<string>());

RILogManager.Get("Test").SendTypedCollection("Hashtable", ienums);
```

## Warning



Warning this failed

```
RILogManager.Get("Test").SendWarning("Warning {0}", "this failed");
```

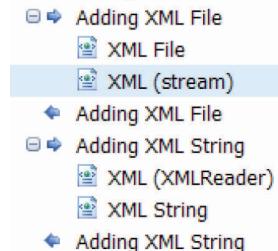
## XML

```
<configuration>
  <appSettings>
    <add key="eventTracker" value="20" />
    <add key="colCategoryDefaultWidth" value="160" />
    <add key="colMachineNameDefaultWidth" value="120" />
    <add key="colApplicationDefaultWidth" value="120" />
    <add key="colUserDefaultWidth" value="100" />
    <add key="colSessionIdDefaultWidth" value="75" />
    <add key="colProcessIdDefaultWidth" value="65" />
    <add key="colThreadIdDefaultWidth" value="65" />
    <add key="colRequestIdDefaultWidth" value="75" />
    <add key="colDateDefaultWidth" value="65" />
    <add key="colTimeDefaultWidth" value="100" />
    <add key="colLogIdDefaultWidth" value="65" />
    <add key="ClientSettingsProvider.ServiceUri" value="" />
  </appSettings>
  <runtime>
    <generatePublisherEvidence enabled="false" />
```

```
try
{
  TextReader tr = new StreamReader(@"C:\ReflectInsight.exe.config");
```

```
RILogManager.Get("Test").SendXML("XML (stream)", File.OpenRead(@"C:\ exe.config"));
XmlReader xmlR = XmlReader.Create(tr);
RILogManager.Get("Test").SendXML("XML (XMLReader)", xmlR);
 XmlDocument doc = new XmlDocument();
doc.Load(@"C:\ReflectInsight.exe.config");
 XmlNode child = doc.SelectSingleNode("/configuration");
if (child != null)
    RILogManager.Get("Test").SendXML("XML (XmlNode)", child);
}
catch (Exception ex3)
{
    RILogManager.Get("Test").SendException("SendXml Exception", ex3);
    RILogManager.Get("Test").SendStackTrace("Stack Trace");
}
```

## XML files



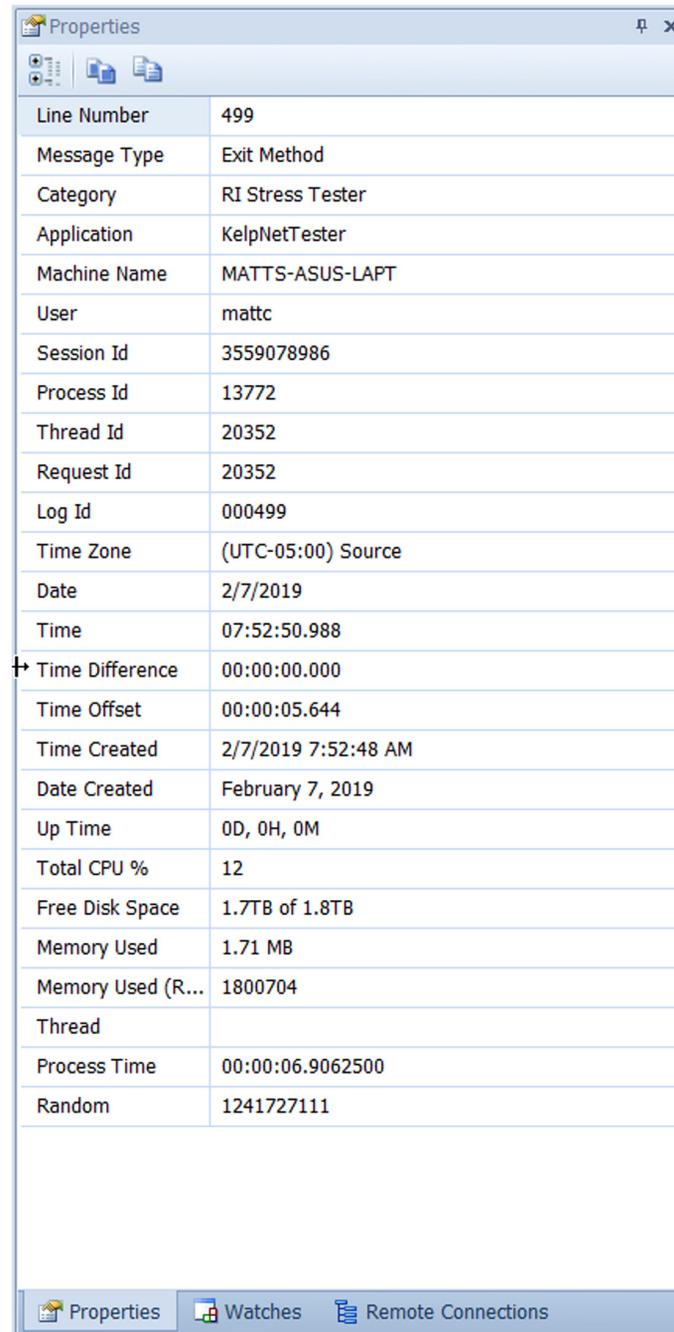
```
RILogManager.Get("Test").SendXMLFile("XML File", @"C:\ RI.exe.config");
```

## Tracing method calls

Trace one two three

```
RILogManager.Get("Test").TraceMethod("Trace Method");
RILogManager.Get("Test").ExitMethod("Trace Method");
RILogManager.Get("Test").TraceMethod(MethodBase.GetCurrentMethod());
RILogManager.Get("Test").ExitMethod(MethodBase.GetCurrentMethod());
RILogManager.Get("Test").TraceMethod(MethodBase.GetCurrentMethod(), true);
RILogManager.Get("Test").ExitMethod(MethodBase.GetCurrentMethod(), true);
```

## Attaching message properties



The screenshot shows the 'Properties' window of ReflectInsight. The window title is 'Properties'. At the top, there are tabs for 'Properties', 'Watches', and 'Remote Connections', with 'Properties' being the active tab. The main area displays a list of message properties in a table format.

Line Number	499
Message Type	Exit Method
Category	RI Stress Tester
Application	KelpNetTester
Machine Name	MATTS-ASUS-LAPT
User	mattc
Session Id	3559078986
Process Id	13772
Thread Id	20352
Request Id	20352
Log Id	000499
Time Zone	(UTC-05:00) Source
Date	2/7/2019
Time	07:52:50.988
Time Difference	00:00:00.000
Time Offset	00:00:05.644
Time Created	2/7/2019 7:52:48 AM
Date Created	February 7, 2019
Up Time	0D, 0H, 0M
Total CPU %	12
Free Disk Space	1.7TB of 1.8TB
Memory Used	1.71 MB
Memory Used (R...)	1800704
Thread	
Process Time	00:00:06.9062500
Random	1241727111

## To one request

```
int nNext = r.Next(104, Int32.MaxValue);
RIExtendedMessageProperty.AttachToRequest("Test Message", "Thread", System.Threading.Thread.CurrentThread.Name);
RIExtendedMessageProperty.AttachToRequest("Test Message", "Process Time", System.Diagnostics.Process.GetCurrentProcess().TotalProcessorTime.ToString());
RIExtendedMessageProperty.AttachToRequest("Test Message", "Random", nNext.ToString());
```

## To all requests

```
RIExtendedMessageProperty.AttachToAllRequests("Test Message", "Time Created", DateTime.Now.ToString());
RIExtendedMessageProperty.AttachToAllRequests("Test Message", "Date Created", DateTime.Now.ToString("yyyy-MM-dd HH:mm:ss"));
TimeSpan ts = DateTime.Now - Process.GetCurrentProcess().StartTime;
RIExtendedMessageProperty.AttachToAllRequests("Test Message", "Up Time", String.Format("{0} D, {1}H, {2}M", ts.Days, ts.Hours, ts.Minutes));
RIExtendedMessageProperty.AttachToAllRequests("Test Message", "Total CPU %", GetTotalCPU("RIS Stress.exe").ToString());
RIExtendedMessageProperty.AttachToAllRequests("Test Message", "Free Disk Space", GetDiskSpace("C:\\"));
RIExtendedMessageProperty.AttachToAllRequests("Memory Profiler", "Memory Used", FormatBytes(GC.GetTotalMemory(true)));
RIExtendedMessageProperty.AttachToAllRequests("Memory Profiler", "Memory Used (Raw)", GC.GetTotalMemory(true).ToString());
```

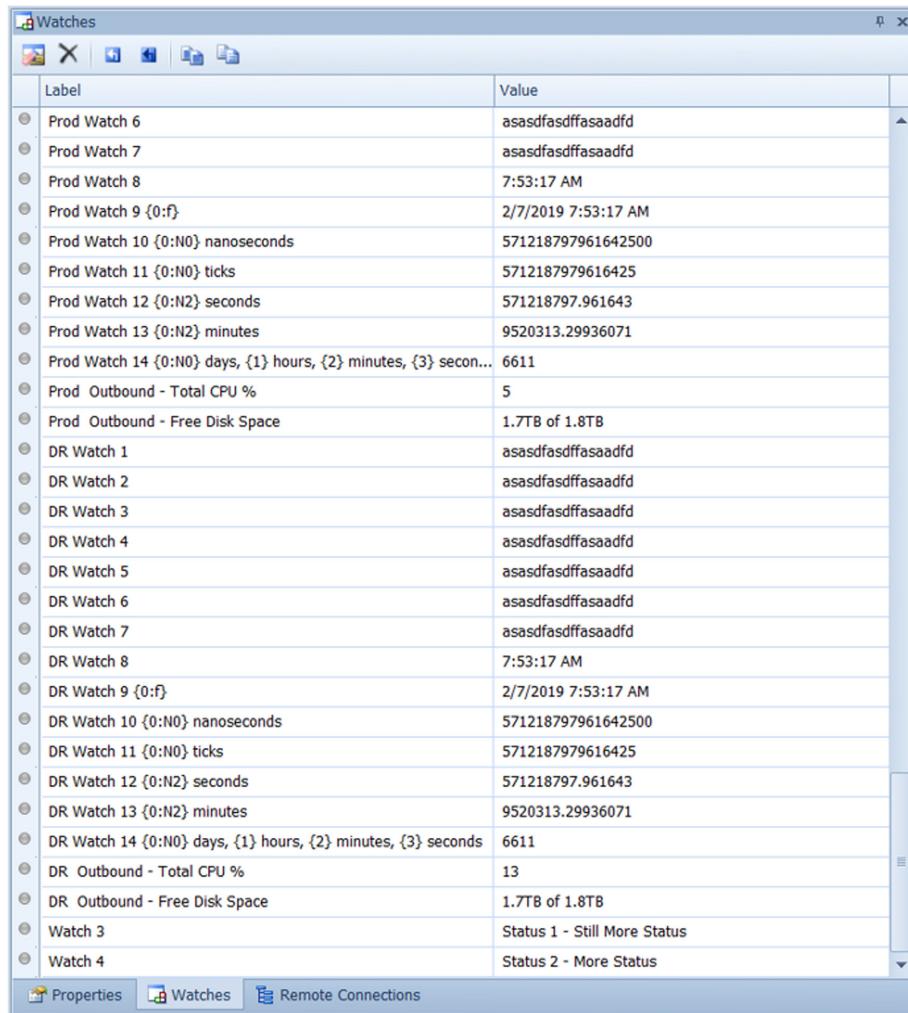
## To a single message

```
for (Int32 i = 0; i < 10000; i++)
{
    RIExtendedMessageProperty.AttachToSingleMessage("Company Information", "ClientID", "1234");
    RIExtendedMessageProperty.AttachToSingleMessage("Company Information", "CompanyID", "56789");
    RIExtendedMessageProperty.AttachToSingleMessage("Company Information", "State", "CA");
    ri.SendMessage("Hello");
}
```

## Watches

Watches are powerful components in ReflectInsight and are added to the live viewer by using the `ViewerSendWatch` function. The first parameter is the text label of what the watch parameter is and the second parameter is the value itself.

```
DateTime centuryBegin = new DateTime(2001, 1, 1);
DateTime currentDate = DateTime.Now;
long elapsedTicks = currentDate.Ticks - centuryBegin.Ticks;
TimeSpan elapsedSpan = new TimeSpan(elapsedTicks);
RILogger.Get("Test").ViewerSendWatch(s + "Watch 8", currentDate.ToString());
RILogger.Get("Test").ViewerSendWatch(s + "Watch 9 {0:f}", currentDate);
RILogger.Get("Test").ViewerSendWatch(s + "Watch 10 {0:N0} nanoseconds", elapsedTicks * 100);
RILogger.Get("Test").ViewerSendWatch(s + "Watch 11 {0:N0} ticks", elapsedTicks);
RILogger.Get("Test").ViewerSendWatch(s + "Watch 12 {0:N2} seconds", elapsedSpan.TotalSeconds);
RILogger.Get("Test").ViewerSendWatch(s + "Watch 13 {0:N2} minutes", elapsedSpan.TotalMinutes);
RILogger.Get("Test").ViewerSendWatch(s + "Watch 14 {0:N0} days, {1} hours, {2} minutes, {3} seconds", elapsedSpan.Days.ToString(), elapsedSpan.Hours, elapsedSpan.Minutes, elapsedSpan.Seconds);
RILogger.Get("Test").ViewerSendWatch(s + " Outbound - Total CPU %", GetTotalCPU("RISstress.exe"));
RILogger.Get("Test").ViewerSendWatch(s + " Outbound - Free Disk Space", GetDiskSpace("C:\\"));
```



The screenshot shows the Visual Studio 'Watches' window. The title bar says 'Watches'. The window contains a table with two columns: 'Label' and 'Value'. There are approximately 30 rows of data, mostly labeled 'Prod Watch' or 'DR Watch' followed by a number. Some values are dates and times, others are numerical values like 5 or 6611, and some are strings like 'asasdffasdfsadfd'. At the bottom of the table, there are two rows labeled 'Watch 3' and 'Watch 4' with values 'Status 1 - Still More Status' and 'Status 2 - More Status' respectively. Below the table, there are three tabs: 'Properties', 'Watches' (which is selected), and 'Remote Connections'.

Label	Value
Prod Watch 6	asasdffasdfsadfd
Prod Watch 7	asasdffasdfsadfd
Prod Watch 8	7:53:17 AM
Prod Watch 9 {0:f}	2/7/2019 7:53:17 AM
Prod Watch 10 {0:N0} nanoseconds	571218797961642500
Prod Watch 11 {0:N0} ticks	5712187979616425
Prod Watch 12 {0:N2} seconds	571218797.961643
Prod Watch 13 {0:N2} minutes	9520313.29936071
Prod Watch 14 {0:N0} days, {1} hours, {2} minutes, {3} secon...	6611
Prod Outbound - Total CPU %	5
Prod Outbound - Free Disk Space	1.7TB of 1.8TB
DR Watch 1	asasdffasdfsadfd
DR Watch 2	asasdffasdfsadfd
DR Watch 3	asasdffasdfsadfd
DR Watch 4	asasdffasdfsadfd
DR Watch 5	asasdffasdfsadfd
DR Watch 6	asasdffasdfsadfd
DR Watch 7	asasdffasdfsadfd
DR Watch 8	7:53:17 AM
DR Watch 9 {0:f}	2/7/2019 7:53:17 AM
DR Watch 10 {0:N0} nanoseconds	571218797961642500
DR Watch 11 {0:N0} ticks	5712187979616425
DR Watch 12 {0:N2} seconds	571218797.961643
DR Watch 13 {0:N2} minutes	9520313.29936071
DR Watch 14 {0:N0} days, {1} hours, {2} minutes, {3} seconds	6611
DR Outbound - Total CPU %	13
DR Outbound - Free Disk Space	1.7TB of 1.8TB
Watch 3	Status 1 - Still More Status
Watch 4	Status 2 - More Status

## Using custom data

Custom data is very powerful and allows you to add custom data objects to ReflectInsight. There are several RICustomXXX objects you can work with to define your data. Here is a quick sample code showing how you can use this powerful feature:

```
List<RICustomDataColumn> columns = new List<RICustomDataColumn>();
columns.Add(new RICustomDataColumn("Property"));
columns.Add(new RICustomDataColumn("Value"));
```

```
RICustomData rcd = new RICustomData(strNote, columns, false, true)
{ Expanded = false, AllowSort = true };
RICustomDataCategory cat = null;
foreach (TestObject c in objects)
{
if (c != null)
{
    cat = rcd.AddCategory("Test Object");
    cat.Expanded = false;
    cat.AddRow("str1", c.str1);
    cat.AddRow("str2", c.str2);
    cat.AddRow("str3", c.str3);
    cat.AddRow("str4", c.str4);
    cat.AddRow("str5", c.str5);
    cat.AddRow("str6", c.str6);
    cat.AddRow("str7", c.str7);
    cat.AddRow("str8", c.str8);
    cat.AddRow("str9", c.str9);
    cat.AddRow("str10", c.str10);
    if (c.obj1 != null || c.obj2 != null)
        cat = cat.AddCategory("Test Object Sub Object");
    if (c.obj1 != null)
        cat.AddRow("obj1", c.obj1.ToString());
    if (c.obj2 != null)
        cat.AddRow("obj2", c.obj2.ToString());
}
}
RILogManager.Get(strInstanceName).SendCustomData(strNote, rcd);
```

## Output

Here is what our custom object looks like when displayed within the live viewer.

The screenshot shows a software interface titled "Custom Data". At the top, there are four icons: a magnifying glass, a folder, a file, and a refresh symbol. Below this is a toolbar with a "Details" button, a "Test Object" dropdown, and other buttons. The main area is titled "Custom Data" and contains a tree view with a node expanded to show "Test Object". This node has ten entries labeled str1 through str10, each with a value. Below this is another node titled "Test Object Sub Object" with one entry labeled "obj2" and the value "2".

Custom Data	
Details	
Custom Data	
▼	Test Object
str1	asfsd
str2	aetrer
str3	asdgg
str4	sdbhjkahsjkd
str5	abfdlkjhkjhe
str6	baklshdrjkhere
str7	adsblkhhkjer
str8	jjlbkl akjsdhr ; klh
str9	b;ahsldhf krth lsk lkh
str10	baseklrh
▼	Test Object Sub Object
obj2	2

## Summary

In this chapter, we learned about ReflectInsight and the incredible benefits it can provide you. We saw how it can help the machine learning developer see exactly what is going on inside their algorithm. I encourage you to download your copy and try it out. You will never look at logging the same way again. In the next chapter, we will deep dive into the world of machine and deep learning by discussing topics, terms and concepts.

ReflectInsight is copyright ReflectSoftware. All images, text, logos are used with permission.

You can download a trial copy of ReflectInsight from [www.reflectsoftware.com](http://www.reflectsoftware.com). You need to mention the book code BPBDL201901 when purchasing and get a discount on the retail price!

# CHAPTER 4

# Kelp.Net

# Reference

In this chapter, we will discuss many features and functionalities present in Kelp. Net at the time of writing this book. As always, you should download the source code from the GitHub location provided after purchasing this book to view the latest features and functionalities.

## Let us be honest

Let us start this chapter by talking about where we are today relative to deep learning. We have come a long way in the past few years regarding research, computer processing power, and the general know how and testing of deep learning networks. Implementations of deep learning abound. The academic world should be given a great pat on the back for not only the advancement of knowledge regarding deep learning, but also for the great work that has been done with it. For readers who work on software projects in the real-world, you must be all familiar with the shipping date of a product (remember our product manager from Chapter 1). This is the date that the marketing team has decided the product will be available to the general public. It is not without any notice that in some companies, come hell or high water, on this date the product will ship, ready or not. With deep learning now out in the corporate world, the technology that once lived in a pristine, temperature-controlled environment is now subject to the same rules as we use for other software. Sure, some of our software is mission critical and very important. But does it compare to say an autonomous driving vehicle? Someone uses deep learning in a project, and it is due to go into production at some point. Come hell or high water, the project will have to ship on a certain date. Nothing different from the normal you might think, right?

Let us say our deep learning product is an autonomous driving vehicle. The vehicle could be a car, truck, etc., and it has a deep learning system in it that helps power its autonomous driving mode. How does each individual drive the car takes get debugged? Does the car need to get certified for driving like a human does in order to drive on the same roads? Are there differences by state? What new kinds of questions will we need to ask to get our new generation of software into production? Also, in this case, the deep learning software is a component of a bigger product (the vehicle), which also has a production date itself.

Although deep learning is an exciting area of research, as I mentioned earlier, once it enters the corporate world, it becomes a product where marketing and technology take over, and many things can happen differently than they do inside the pristine R&D think tank.

And the cold hard truth is that there still is a lot we do not know about deep learning. For instance, a neural network is a model of the human brain. You must understand that we know and understand very little about how the human brain actually works. We think we know some of what is going on, but the theory on this continues to evolve daily. Without this understanding, how can we design effective deep learning systems that will solve today's greatest problems? So, we are releasing software into the world, based on something we know little about, and we are expected to debug the software and ship a full feature set to meet production 'specs', which may not be the same as how the model would have matured otherwise. If the car is ready but the software is not, then what is the next step?

We can do all kinds of things with deep learning as you will see in this book. However, solving an XOR problem and perfecting an autonomous vehicle are two different things. Both are important, but both also require understanding of what the system is doing. An XOR problem has to determine whether the output is a 1 or a 0. The autonomous driving vehicle has to determine if what it sees is a pedestrian, and then make downstream decisions based on that. With my XOR problem, I can employ reinforcement learning if it gets the problem wrong. Not so sure of the appetite for reinforcement learning when it comes to hitting a pedestrian in a crosswalk!

I would like to make another point here and that is deep learning is entering the corporate world. At least in America, sooner or later, regulation and oversight steps in. As for deep learning, it has not done so in full force yet, but inevitably, it will. In the meantime, think about this. If I want to drive a car in any state, I need to get a driver's license, which requires taking a test, both written and driving, as well as an eye exam. Maybe I even need to re-certify often based on the state I am in. That is oversight and regulation. What about these autonomous vehicles driving around? Are they certified, where did they take their driving test, and if so how? The answer is no, at least not yet. But these autonomous vehicles are, in fact, massive experiments in deep learning. It is great as a tool to help us understand this particular field of work

better, but do not forget that this experiment, which is what it is, has human lives at stake as a part of their execution. Who debugs autonomous vehicles to see if their image and facial recognition software is working correctly? Does anyone review the images and results from a particular drive? It has been proven that interfering with even a single pixel on an image recognition screen can affect images in their totality, so how do we know pixel interference will not affect us on our next drive? These are some things we need to think about in the philosophical discussion related to deep learning.

My point in all of this? You have a great piece of software at your hands that you can use to answer these and many more questions. Embellish it, make it better, and make your deep learning experiments better and more powerful. Embrace more powerful machines and more data as this is an important key to making deep learning work better and more accurate. Be the one to find the answers to all these and more questions. Be the one to make a difference!

## Downloading Kelp.Net

Kelp.Net can be downloaded from its GitHub repository. Please refer to the information given when you purchase this book.

Branch: master ▾		New pull request	Find file	Clone or download ▾
 mattcolefla	Added SortedList and SortedFunctionStack. Added more documentation.		Latest commit 05de4a3 on May 7, 2018	
 <a href="#">CIFARLoader</a>	Adding in ReflectInsight		9 months ago	
 <a href="#">CaffemodelLoader</a>	Fixed GPU weaver errors on benchmarking. Benchmarking now runs with GPU		9 months ago	
 <a href="#">ChainerModelLoader</a>	Updated tests, translations, argument validation, added memory tester		9 months ago	
 <a href="#">Cloo</a>	Added SortedList and SortedFunctionStack. Added more documentation		9 months ago	
 <a href="#">KelpNet</a>	Added SortedList and SortedFunctionStack. Added more documentation.		9 months ago	
 <a href="#">KelpNetTester</a>	Added SortedList and SortedFunctionStack. Added more documentation.		9 months ago	
 <a href="#">KelpNetWaifu2x</a>	Updated tests, translations, argument validation, added memory tester		9 months ago	
 <a href="#">MNISTLoader</a>	Adding in ReflectInsight		9 months ago	
 <a href="#">MemoryTester</a>	Updated tests, translations, argument validation, added memory tester		9 months ago	
 <a href="#">TestDataManager</a>	Fixed GPU weaver errors on benchmarking. Benchmarking now runs with GPU		9 months ago	
 <a href="#">VocabularyMaker</a>	Adding in ReflectInsight		9 months ago	
 <a href="#">.gitattributes</a>	Initial commit		9 months ago	
 <a href="#">.gitignore</a>	Initial commit		9 months ago	
 <a href="#">ExcelCNN.xls</a>	Initial commit		9 months ago	
 <a href="#">KelpNet.sln</a>	Updated tests, translations, argument validation, added memory tester		9 months ago	
 <a href="#">LICENSE</a>	Initial commit		9 months ago	
 <a href="#">README.md</a>	Initial commit		9 months ago	

## Building the source code

Once the solution is downloaded, do a rebuild all on the project to ensure it builds properly. Prior to doing so, please make sure you have downloaded and installed the community version (or paid version if you have it) of Microsoft Visual Studio 2017. When rebuilding all the projects, please make sure that you have restored any NuGet packages, which are being used first. The building of this software is no different from the projects you have already done using Microsoft Visual Studio.

## What is Kelp.Net?

Kelp.Net is a flexible and powerful framework used to create and test deep learning neural network models. It allows you to write complex architectures simply and intuitively by adopting a define-by-run paradigm. This means that the neural network is defined dynamically via the actual forward computation versus the programming of it. More precisely, Kelp.Net stores the history of computation instead of the programming logic. This makes it easy to write multi-GPU parallelization, since logic comes closer to network manipulation.

For those of you who are familiar with Define-by-Run, let me tell you that it recently became famous because of the building blocks from which Chainer was built. Chainer was the first deep learning framework to introduce the Define-by-Run approach.

The traditional procedure to train a neural network before Define-by-Run was done in two phases: define the fixed connections between mathematical operations (such as matrix multiplication and non-linear activations) in the network, and then run the actual training calculation. Theano and TensorFlow are among the notable frameworks that took this approach. In contrast, in the Define-by-Run, or dynamic-graph approach, the connection in a network is not determined when the training starts. The network is determined during the training when the actual calculation is performed.

One of the advantages of this approach is that it is intuitive and flexible. If the network has complicated control flows such as conditional and loop statements in the Define-by-Run approach, then specially designed operations for such constructs are needed. On the other hand, in the Define-by-Run approach, the programming language's native constructs such as if statements and for loops can be used to describe such a flow.

Another advantage is ease of debugging, which we do a ton of! In the Define-by-Run approach, if an error (such as numeric error) occurs in the training calculation, it is often difficult to inspect the fault because the code written to define the network and the actual place of the error are separated. In the Define-by-Run approach, you can just suspend the calculation with the language's built-in debugger and inspect the data that flows on your code of the network.

As you can see, Kelp.Net will put your machine and deep learning efforts at the same starting stage as if you were writing your project in one of the newer data science languages.

**The original version of Kelp.Net has been converted to English dramatically enhanced, and released back into the open source community for the writing of this book. Many new tests have been added, extensive instrumentation has been provided, and new features and functionality has been added. All the credit goes to the original author for the vision and effort it took to create this wonderful software package.**

## Functions

In Kelp.Net, functions are the base objects used in many classes. Similar in concept to the Microsoft .NET System.Object, they are defined as follows:

```
[Serializable]
public abstract class Function : IComparable
```

Functions are the basic building blocks of a Kelp.Net neural network. Single functions are chained together within FunctionStack to create powerful and possibly complex neural network chains. Functions are also chained together when they are loaded from disks.

There are four primary types of functions you need to know about, and their purposes should be self-explanatory:

- Single-input functions
- Dual-input functions
- Multi-input functions
- Multi-output functions

Each function has a Forward and Backward method that you will be implementing when you create functions of your own, which we will discuss in a separate chapter:

```
public abstract NdArray[] Forward(params NdArray[] xs);
public virtual void Backward([CanBeNull]) params NdArray[] xs);
```

The complete definition for the class is as follows:

```
[Serializable]
public abstract class Function : IComparable
{
```

```
/// <summary> The name. </summary>
public string Name;

///////////////////////////////
/// <summary> Gets/Sets a value indicating whether this object is GPU enabled. </summary>
///
/// <value> True if GPU enable, false if not. </value>
///////////////////////////////

public bool GpuEnable { get; protected set; }

/// <summary> Options for controlling the operation. </summary>
public NdArray[] Parameters = { };
/// <summary> The optimizers. </summary>
public Optimizer[] Optimizers = { };

/// <summary> The previous inputs. </summary>
[NonSerialized]
public List<NdArray[]> PrevInputs = new List<NdArray[]>();

///////////////////////////////
/// <summary> Forwards using the variable-length NdArray parameter. </summary>
///
/// <param name="xs"> A variable-length parameter of type NdArray. </param>
///
/// <returns> A NdArray[]. </returns>
///////////////////////////////

public abstract NdArray[] Forward(bool verbose, params NdArray[] xs);

///////////////////////////////
/// <summary> Backwards using the variable-length NdArray parameter. </summary>
///
/// <param name="ys"> A variable-length parameter of type NdArray. </param>
///////////////////////////////

public virtual void Backward(bool verbose, [CanBeNull] params NdArray[] ys){}
```

```
/// <summary> List of names of the inputs. </summary>
public string[] InputNames;
/// <summary> List of names of the outputs. </summary>
public string[] OutputNames;

///////////////////////////////
/// <summary>
/// Initializes a new instance of the KelpNet.Common.Functions.Function class.
/// </summary>
///
/// <param name="name"> The name. </param>
/// <param name="inputNames"> (Optional) List of names of the inputs. </param>
/// <param name="outputNames"> (Optional) List of names of the outputs. </param>
///////////////////////////////

protected Function([CanBeNull] string name, [CanBeNull] string[] inputNames = null,
[CanBeNull] string[] outputNames = null)
{
    Name = name;
    if (inputNames != null)
    {
        InputNames = inputNames.ToArray();
    }
    if (outputNames != null)
    {
        OutputNames = outputNames.ToArray();
    }
}

///////////////////////////////
/// <summary> Sets an optimizer. </summary>
///
/// <param name="optimizers"> A variable-length parameters list containing optimizers. </param>
/////////////////////////////
public virtual void SetOptimizer([NotNull] params Optimizer[] optimizers)
{
    Optimizers = optimizers;
```

```
foreach (Optimizer optimizer in optimizers)
{
    optimizer?.AddFunctionParameters(Parameters);
}

/// <summary> Function to call when updating parameters) </summary>
protected void BackwardCountUp()
{
    foreach (NdArray parameter in Parameters)
    {
        parameter.CountUp();
    }
}

///////////////////////////////
/// <summary> Evaluation function. </summary>
///
/// <param name="input"> A variable-length parameters list containing input. </param>
///
/// <returns> A NdArray[]. </returns>
/////////////////////////////
[CanBeNull]
public virtual NdArray[] Predict(bool verbose = true, [CanBeNull] params NdArray[] input)
{
    return Forward(verbose, input);
}

/// <summary> Updates this object. </summary>
public virtual void Update(bool verbose = true)
{
    foreach (Optimizer optimizer in Optimizers)
    {
        optimizer.Update(verbose);
    }
}

/// <summary> Initialize input data that could not be used up by RNN etc. </summary>
```

```
public virtual void ResetState()
{
    PrevInputs = new List<NdArray>();
}

/////////// <summary> Returns a string that represents the current object. </summary>
///
/// <returns> A string that represents the current object. </returns>
///
/// <seealso cref="M:System.Object.ToString()">
/////////// <summary> Makes a deep copy of this object. </summary>
///
/// <returns> A copy of this object. </returns>
/////////// [CanBeNull]
public Function Clone()
{
    return DeepCopyHelper.DeepCopy(this);
}

/////////// <inheritdoc />
/// <summary>
/// Compares the current instance with another object of the same type and returns an integer
/// that indicates whether the current instance precedes, follows, or occurs in the same position
/// in the sort order as the other object.
/// </summary>
/// <exception cref="T:System.ArgumentException"> .</exception>
/// <param name="obj"> An object to compare with this instance. </param>
/// <returns>
```

```
/// A value that indicates the relative order of the objects being compared. The return value has
/// these meanings: Value Meaning Less than zero This instance precedes <paramref
name="obj" />
/// in the sort order. Zero This instance occurs in the same position in the sort order as
/// <paramref name="obj" />. Greater than zero This instance follows <paramref name="obj" />
in
/// the sort order.
/// </returns>
/// <seealso cref="M:System.IComparable.CompareTo(object)" />

public int CompareTo([NotNull] object obj)
{
    Function f = (Function) obj;
    if (f.Name == Name)
        return 0;
    if (InputNames != null && OutputNames != null && f.InputNames != null && f.OutputNames != null)
    {
        if (InputNames.Length == f.InputNames.Length && OutputNames.Length == f.OutputNames.Length)
            return (0);
    }
    return (-1);
}
```

As you can see, this is a very simple class definition. Kelp.Net has many classes which inherit from the base Function object.

## N-dimensional arrays

I want to talk briefly on an object that you will see been used quite heavily in Kelp.Net. This object is an n-dimensional array and the class used for this is called NdArray. These serializable n-dimensional arrays are the basic parameter types used in most functions. The data used for testing, training and validating, in fact, will lie within these arrays. They are small, fast, and powerful when it comes to representing your data.

Here are some of the properties which NdArray maintains:

- Data: This is the actual data being manipulated or used.
- Gradient: This is the gradient.
- Shape: This is the size of each dimension of the array.
- Length: The length is calculated from the shape. This is different from the length of Data.
- UseCount: This is the number of times the object has been used by the function. This helps with the timing of the Backward operation.
- ParentFunc: This is the parent of this object.
- BatchCount: This is the number of batches executed collectively in each function. This information is used in the discount in the Loss function.
- TrainCount: This is the number of backward operations executed without updating. This is useful when running the optimizer.

Most of the common mathematical operators can be used on NdArray, as well as operations such as Shape, Concatenate, Sum, and more.

## FunctionStack

FunctionStack are arrays of Function objects used to build Kelp.Net models. You can have as many, or as few, as you like in your model. There is no direct correlation between bigger neural networks, which means more hidden layers and neurons being more powerful than smaller ones. FunctionStack are truly the power and the beauty of Kelp.Net. They allow you to assemble functional groups (layers) that allow you to perform various levels of complex processing. When completed, they will form the basis of your deep learning model and can be saved and loaded from disks as required.

Here is an example of a simple FunctionStack.

```
FunctionStack nn = new FunctionStack(
    new Linear(2, 2, name: "l1 Linear"),
    new Sigmoid(name: "l1 Sigmoid"),
    new Linear(2, 2, name: "l2 Linear")
);
```

This stack contains three layers, two linear and one sigmoid. The stack given above is, as should be the case, created in the order in which you are planning to have each network layer been utilized and used as it moves forward (or backwards as the case may allow) through the architecture. As forward and backward progress occurs across the stack, it will occur in the order in which each layer was added.

Let us take a look at another example. See how easy it is to create a massive deep learning network model with 10,000 layers. These are for illustration purpose only; please do not try running this and then wonder why it does not complete in just a few seconds!

```
for (int x=0; x< numLayers; x++)
{
    functions.Add(new Linear(false, neuronCount * neuronCount, N, name: $"l{x} Linear"));
    functions.Add(new BatchNormalization(false, N, name: $"l{x} BatchNorm"));
    functions.Add(new ReLU(name: $"l{x} ReLU"));
};
```

The final example shows you some of the different ways of putting together functions to create your model. This example is an image processing model with Convolution and MaxPooling layers:

```
FunctionStack model = new FunctionStack(
    new Convolution2D(1, 32, 5, pad: 2, name: "I1 Conv2D", gpuEnable: true),
    new ReLU(name: "I1 ReLU"),
    new MaxPooling(2, 2, name: "I1 MaxPooling", gpuEnable: true),
    new Convolution2D(32, 64, 5, pad: 2, name: "I2 Conv2D", gpuEnable: true),
    new ReLU(name: "I2 ReLU"),
    new MaxPooling(2, 2, name: "I2 MaxPooling", gpuEnable: true),
    new Linear(7 * 7 * 64, 1024, name: "I3 Linear", gpuEnable: true),
    new ReLU(name: "I3 ReLU"),
    new Dropout(name: "I3 DropOut"),
    new Linear(1024, 10, name: "I4 Linear", gpuEnable: true)
);
```

At this point, please note that none of the models we have created are yet complete. We have added our functions for execution, but we still need to add an optimizer for it to be a model. We will do so by simply telling the model to set the optimizer to one of the many available optimizers.

## Optimizers

In Kelp.Net, telling the model to set the optimizer is a one function call. Using the model created above, we will simply call the SetOptimizer method as follows:

```
model.SetOptimizer(new MomentumSGD());
```

The SetOptimizer function takes an Optimizer class object, which is defined as follows:

```
[Serializable]
public abstract class Optimizer
{
    /// <summary> Number of updates. </summary>
    public long UpdateCount = 1;
    /// <summary> Options for controlling the optimizer. </summary>
    protected List<OptimizerParameter> OptimizerParameters = new List<OptimizerParameter>();
    /////////////////////////////////  

    /// <summary> Adds a function parameters. </summary>
    ///
    /// <param name="functionParameters"> Options for controlling the function. </param>
    /////////////////////////////////  

    internal abstract void AddFunctionParameters(NdArray[] functionParameters);
    /// <summary> Updates this object. </summary>
    public void Update()
    {
        bool isUpdated = false;
        foreach (var t in OptimizerParameters)
        {
            // Run discount of slope and check if there was update
            if (t.FunctionParameter.Reduce())
            {
                t.UpdateFunctionParameters();
                t.FunctionParameter?.ClearGrad();
                isUpdated = true;
            }
        }
        if (isUpdated)
        {
            UpdateCount++;
        }
    }

    /// <summary> Resets the parameters. </summary>
    public void ResetParams()
    {
        foreach (var t in OptimizerParameters)
        {
```

```

        t.FunctionParameter?.ClearGrad();
    }
    UpdateCount = 0;
}
}

```

**If you omit the optimizer declaration, the default SGD(0.1) will be used.**

The following optimizers are included with the base Kelp.Net:

Optimizer	Notes
AdaDelta	An extension of AdaGrad that seeks to reduce its aggressive, monotonically decreasing learning rate.
AdaGrad	Adapts the learning rate to the parameters, performs smaller updates (that is, low learning rates) for parameters associated with frequently occurring features, and larger updates (that is, high learning rates) for parameters associated with infrequent features. A downside of AdaGrad is that in case of deep learning, the monotonic learning rate usually proves too aggressive and stops learning too early.
Adam	Adaptive moment estimation computes adaptive learning rates for each parameter. In practice, Adam is currently recommended as the default algorithm to use, and often works slightly better than RMSProp.
GradientClipping	Gradient clipping is a technique to prevent exploding gradients in very deep networks, usually in recurrent neural networks.
MomentumSGD	Momentum Stochastic Gradient Descent.
RMSProp	Uses a moving average of squared gradients to normalize the gradient itself. The RMSProp update adjusts the Adagrad method in a very simple way to reduce its aggressive, monotonically decreasing learning rate.
SGD	Efficient and easy to use.

## AdaDelta

AdaDelta combines scaling of the learning rate based on the historical gradient (taking into account the recent time window only), and uses a component that serves as an acceleration term, accumulating historical updates. The end result is an improvement over AdaGrad object's weakness of the learning rate converging to zero with an increase in time. Adadelta is a gradient descent-based learning algorithm that adapts the learning rate per parameter over time.

This is the code for the AdaDelta object:

```
[Serializable]
public class AdaDelta : Optimizer
{
    /// <summary> The rho. </summary>
    public Real Rho;
    /// <summary> The epsilon. </summary>
    public Real Epsilon;
    /////////////////////////////////////////////////
    ///
    /// <summary> Initializes a new instance of the KelpNet.Optimizers.AdaDelta class. </summary>
    ///
    /// <param name="rho"> (Optional) The rho. </param>
    /// <param name="epsilon"> (Optional) The epsilon. </param>
    ///////////////////////////////////////////////
    public AdaDelta(double rho = 0.95, double epsilon = 1e-6)
    {
        Rho = rho;
        Epsilon = epsilon;
    }
    ///////////////////////////////////////////////
    /// <summary> Adds a function parameters. </summary>
    ///
    /// <param name="functionParameters"> Options for controlling the function. </param>
    ///
    /// <seealso cref="M:KelpNet.Common.Optimizers.Optimizer.AddFunctionParameters(NdArray[])" />
    ///////////////////////////////////////////////
```

```
internal override void AddFunctionParameters([NotNull] NdArray[] functionParameters)
{
    foreach (NdArray functionParameter in functionParameters)
    {
        OptimizerParameters.Add(new AdaDeltaParameter(functionParameter, this));
    }
}
```

## AdaGrad

AdaGrad gives frequently occurring features very low learning rates, and infrequent features high learning rates. The purpose of this is to make a note of this each time an infrequent feature is seen. So, in a nutshell, this facilitates finding and identifying very predictive but, for all intents and purposes, rare, features. AdaGrad can be used in lieu of SGD, and it is particularly helpful when dealing with sparse data, where it assigns higher learning rates to infrequently updated parameters.

This is the code for the AdaGrad object:

```
[Serializable]
public class AdaGrad : Optimizer
{
    /// <summary> The learning rate. </summary>
    public Real LearningRate;
    /// <summary> The epsilon. </summary>
    public Real Epsilon;
    /////////////////////////////////
    /// <summary> Initializes a new instance of the KelpNet.Optimizers.AdaGrad class. </summary>
    ///
    /// <param name="learningRate"> (Optional) The learning rate. </param>
    /// <param name="epsilon"> (Optional) The epsilon. </param>
    ///////////////////////////////
    public AdaGrad(double learningRate = 0.01, double epsilon = 1e-8)
    {
        LearningRate = learningRate;
        Epsilon = epsilon;
    }
    ///////////////////////////////
    /// <summary> Adds a function parameters. </summary>
```

```
///  
/// <param name="functionParameters"> Options for controlling the function. </param>  
///  
/// <seealso cref="M:KelpNet.Common.Optimizers.Optimizer.AddFunctionParameters(NdArray[])" />  
/////////////////////////////////////////////////////////////////////////  
internal override void AddFunctionParameters([NotNull] NdArray[] functionParameters)  
{  
    foreach (NdArray functionParameter in functionParameters)  
    {  
        OptimizerParameters.Add(new AdaGradParameter(functionParameter, this));  
    }  
}  
}
```

Adam

Adam has become an increasingly popular optimization algorithm. You can think of Adam as a generalized AdaGrad if you want, which means that Adam has certain parameter choices pre-made for its optimality. So, Adam is AdaDelta plus momentum, and sometimes reverts to AdaDelta under certain hyperparameter settings. If you disable the first-order smoothing in Adam, you have AdaDelta. If you disable the second-order rescaling, you have SGD + momentum. Regardless, your efforts will still be towards tweaking the learning rate. More formally, algorithmic updates are directly estimated using a running average of the first and second moment of the gradient and include a bias correction term.

This is the code for the Adam object:

```
/// <summary> Initializes a new instance of the KelpNet.Optimizers.Adam class. </summary>
///
/// <param name="alpha"> (Optional) The alpha. </param>
/// <param name="beta1"> (Optional) The first beta. </param>
/// <param name="beta2"> (Optional) The second beta. </param>
/// <param name="epsilon"> (Optional) The epsilon. </param>
///////////////////////////////
public Adam(double alpha = 0.001, double beta1 = 0.9, double beta2 = 0.999, double epsilon =
1e-8)
{
    Alpha = alpha;
    Beta1 = beta1;
    Beta2 = beta2;
    Epsilon = epsilon;
}
///////////////////////////////
/// <summary> Adds a function parameters. </summary>
///
/// <param name="functionParameters"> Options for controlling the function. </param>
///
/// <seealso cref="M:KelpNet.Common.Optimizers.Optimizer.AddFunctionParameters(NdArray["
])"/>
///////////////////////////////

internal override void AddFunctionParameters([NotNull] NdArray[] functionParameters)
{
    foreach (NdArray functionParameter in functionParameters)
    {
        OptimizerParameters.Add(new AdamParameter(functionParameter, this));
    }
}
```

## GradientClipping

GradientClipping happens mostly in Recurrent Neural Networks. Here is a brief explanation as to why.

When gradients are propagated back in time, they can vanish because they are continuously multiplied by numbers smaller than 1. This is known as the **vanishing**

**gradient problem.** Conversely, when the gradients get to be exponentially large from being multiplied by numbers greater than 1, you have what is known as the **exploding gradient problem**.

GradientClipping will clip the gradients between two numbers in order to prevent them from getting too large.

This is the code for the GradientClipping object:

```
[Serializable]
public class GradientClipping : Optimizer
{
    /// <summary> The threshold. </summary>
    public Real Threshold;
    /////////////////////////////////
    /// <summary>
    /// Initializes a new instance of the KelpNet.Optimizers.GradientClipping class.
    /// </summary>
    ///
    /// <param name="threshold"> The threshold. </param>
    /////////////////////////////////

    public GradientClipping(double threshold)
    {
        Threshold = threshold;
    }
    /////////////////////////////////
    /// <summary> Adds a function parameters. </summary>
    ///
    /// <param name="functionParameters"> Options for controlling the function. </param>
    ///
    /// <seealso cref="M:KelpNet.Common.Optimizers.Optimizer.AddFunctionParameters(NdArray[])" />
    /////////////////////////////////

    internal override void AddFunctionParameters([NotNull] NdArray[] functionParameters)
    {
        foreach (NdArray functionParameter in functionParameters)
        {
            OptimizerParameters.Add(new GradientClippingParameter(functionParameter, this));
        }
    }
}
```

```
    }  
}  
}
```

## MomentumSGD

**Stochastic Gradient Descent (SGD)** is dependent upon the given instance or instances of the present data iteration. Therefore, it tends to have unstable update steps per iteration, convergence takes more time, and your model appears to be stuck into a poor local minima. In order to address this problem, momentum (more concretely, Nesterov Momentum) is applied to keep the history of the previous update steps, combine this information with the next gradient step to keep the resulting updates more stable and smoother. In simpler terms, we are preventing chaotic jumps.

This is the code for the MomentumSGD object:

```
[Serializable]  
public class MomentumSGD : Optimizer  
{  
    /// <summary> The learning rate. </summary>  
    public Real LearningRate;  
    /// <summary> The momentum. </summary>  
    public Real Momentum;  
    /////////////////////////////////  
    /// <summary>  
    /// Initializes a new instance of the KelpNet.Optimizers.MomentumSGD class.  
    /// </summary>  
    ///  
    /// <param name="learningRate"> (Optional) The learning rate. </param>  
    /// <param name="momentum"> (Optional) The momentum. </param>  
    /////////////////////////////////  
  
    public MomentumSGD(double learningRate = 0.01, double momentum = 0.9)  
    {  
        LearningRate = learningRate;  
        Momentum = momentum;  
    }  
    /////////////////////////////////  
    /// <summary> Adds a function parameters. </summary>
```

```
///  
/// <param name="functionParameters"> Options for controlling the function. </param>  
///  
/// <seealso cref="M:KelpNet.Common.Optimizers.Optimizer.AddFunctionParameters(NdArr  
ay[])" />  
/////////////////////////////////////////////////////////////////////////  
  
internal override void AddFunctionParameters([NotNull] NdArray[] functionParameters)  
{  
    foreach (NdArray functionParameter in functionParameters)  
    {  
        OptimizerParameters.Add(new MomentumSGDParameter(functionParameter, this));  
    }  
}
```

RMSPROP

RMSprop was developed in order to resolve AdaGrad's radically diminishing learning rates. RMSprop divides the learning rate by an exponentially decaying average of squared gradients.

This is the code for the RMSprop object

```
[Serializable]
public class RMSprop : Optimizer
{
    /// <summary> The learning rate. </summary>
    public Real LearningRate;
    /// <summary> The alpha. </summary>
    public Real Alpha;
    /// <summary> The epsilon. </summary>
    public Real Epsilon;
    /////////////////////////////////////////////////
    /// <summary> Initializes a new instance of the KelpNet.Optimizers.RMSprop class. </summary>
    ///
    /// <param name="learningRate"> (Optional) The learning rate. </param>
    /// <param name="alpha">      (Optional) The alpha. </param>
    /// <param name="epsilon">     (Optional) The epsilon. </param>
    ///////////////////////////////////////////////
```

```
public RMSprop(double learningRate = 0.01, double alpha = 0.99, double epsilon = 1e-8)
{
    LearningRate = learningRate;
    Alpha = alpha;
    Epsilon = epsilon;
}
///////////////////////////////  

/// <summary> Adds a function parameters. </summary>  

///  

/// <param name="functionParameters"> Options for controlling the function. </param>  

///  

/// <seealso cref="M:KelpNet.Common.Optimizers.Optimizer.AddFunctionParameters(NdArr  

ay[])" />  

///////////////////////////////  

internal override void AddFunctionParameters([NotNull] NdArray[] functionParameters)
{
    foreach (NdArray functionParameter in functionParameters)
    {
        OptimizerParameters.Add(new RMSpropParameter(functionParameter, this));
    }
}
```

## SGD

**Stochastic Gradient Descent (SGD)** is an algorithm in which the batch size is one. This means that SGD relies on just a single example, chosen uniformly at random from your dataset, and is used to calculate an estimate of the gradient at each step.

This is the code for the SGD (Stochastic Gradient Descent) object:

```
[Serializable]
public class SGD : Optimizer
{
    /// <summary> The learning rate. </summary>
    public Real LearningRate;
}
///////////////////////////////  

/// <summary> Initializes a new instance of the KelpNet.Optimizers.SGD class. </summary>
```

```
///
/// <param name="learningRate"> (Optional) The learning rate. </param>
//////////////////////////////  
  
public SGD(double learningRate = 0.1)  
{  
    LearningRate = learningRate;  
}  
//////////////////////////////  
/// <summary> Adds a function parameters. </summary>  
///  
/// <param name="functionParameters"> Options for controlling the function. </param>  
///  
/// <seealso cref="M:KelpNet.Common.Optimizers.Optimizer.AddFunctionParameters(NdArr  
ay[])" />  
//////////////////////////////  
  
internal override void AddFunctionParameters([NotNull] NdArray[] functionParameters)  
{  
    foreach (NdArray functionParameter in functionParameters)  
    {  
        OptimizerParameters.Add(new SGDParameter(functionParameter, this));  
    }  
}
```

## Poolings

Pooling layers are most commonly inserted between successive convolutional layers in a **Convolutional Neural Network (CNN)** in order to progressively reduce the width and height of the data and to control overfitting. Kelp.Net provides the two most common pooling layers, which are the max and the average operators. These layers perform downsampling operations along the height and width of the input data.

The most common setup for a pooling layer is to apply 2x2 filters with a stride of 2. These will downsample each depth slice in the input volume by a factor of two on the spatial dimensions (width and height). With a 2x2 filter size, the max() operation takes the largest of the four numbers in the filter area. This means that if the input image is 32x32, the output image would then become say, 16x16. This operation does

not affect the depth dimension.

Pooling layers do not have parameters for the layer because they compute a fixed function. They do, however, have additional hyperparameters.

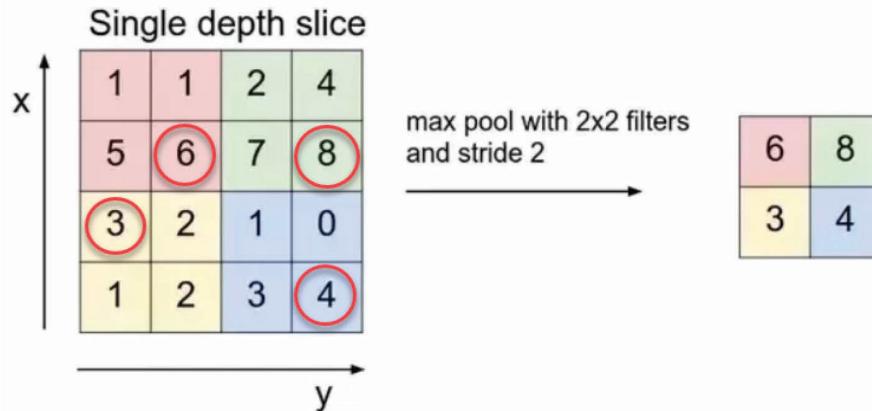
The following pooling layers are available in Kelp.Net:

Layer	Notes
MaxPooling	This layer takes the largest of numbers in the feature area and extracts most important features.
AveragePooling	This layer takes the average of numbers in the feature area and extracts features smoothly (less extreme edges).

## MaxPooling

With max pooling, the goal is to downsample the input data, thereby reducing its dimensionality and allowing feature assumptions to be made. It also reduces the computational cost by reducing the number of parameters to learn. This is accomplished by applying a max filter to sub regions of the initial data. These sub regions are usually non-overlapping but do not have to be.

For example, let us say we have a 4x4 matrix of simple data as our input, as shown in the following diagram. Our max pooling filter is a 2x2 filter, which we will use as the input. We will have a stride of 2, which means that the  $dx$  and  $dy$  for stepping over our input will be (2, 2) and will not overlap any regions. For each region that is represented by our filter, we will take the max value of that region and create a new output matrix, where each element is the max of a region from the original input. As you can see in the following diagram, our result is a 2x2 matrix with each element being a max element from the input regions:



This is the code for the MaxPooling object:

```
////////////////////////////////////////////////////////////////
/// <summary> Maximum pooling. </summary>
///
/// <seealso cref="T:KelpNet.Common.Functions.Type.SingleInputFunction"/>
/// <seealso cref="T:KelpNet.Common.Functions.IParallelizable"/>
////////////////////////////////////////////////////////////////
[Serializable]
public class MaxPooling : SingleInputFunction, IParallelizable
{
    /// <summary> Name of the function. </summary>
    const string FUNCTION_NAME = "MaxPooling";
    /// <summary> The width. </summary>
    private int _kWidth;
    /// <summary> The height. </summary>
    private int _kHeight;
    /// <summary> The pad x coordinate. </summary>
    private int _padX;
    /// <summary> The pad y coordinate. </summary>
    private int _padY;
    /// <summary> The stride x coordinate. </summary>
    private int _strideX;
    /// <summary> The stride y coordinate. </summary>
    private int _strideY;

    /// <summary> List of output indices. </summary>
```

```
private readonly List<int[]> _outputIndicesList = new List<int[]>();  
  
/// <summary> The forward kernel. </summary>  
[NonSerialized]  
[DebuggerBrowsable(DebuggerBrowsableState.Never)]  
public ComputeKernel ForwardKernel;  
////////////////////////////////////////////////////////////////////////  
/// <summary>  
/// Initializes a new instance of the KelpNet.Functions.Poolings.MaxPooling class.  
/// </summary>  
///  
/// <param name="ksize"> The ksize. </param>  
/// <param name="stride"> (Optional) The stride. </param>  
/// <param name="pad"> (Optional) The pad. </param>  
/// <param name="gpuEnable"> (Optional) True if GPU enable. </param>  
/// <param name="name"> (Optional) The name. </param>  
/// <param name="inputNames"> (Optional) List of names of the inputs. </param>  
/// <param name="outputNames"> (Optional) List of names of the outputs. </param>  
////////////////////////////////////////////////////////////////////////  
public MaxPooling(int ksize, int stride = 1, int pad = 0, bool gpuEnable = false, [CanBeNull]  
string name = FUNCTION_NAME, [CanBeNull] string[] inputNames = null, [CanBeNull] string[]  
outputNames = null) : base(name, inputNames, outputNames)  
{  
    _kHeight = ksize;  
    _kWidth = ksize;  
    _padY = pad;  
    _padX = pad;  
    _strideX = stride;  
    _strideY = stride;  
    SetGpuEnable(gpuEnable);  
    SingleOutputBackward = BackwardCpu;  
}  
////////////////////////////////////////////////////////////////////////  
/// <summary> Sets GPU enable. </summary>  
///  
/// <param name="enable"> True to enable, false to disable. </param>
```

```

///<summary>
///<param name="ksize"> The ksize. </param>
///<param name="stride"> (Optional) The stride. </param>
///<param name="pad"> (Optional) The pad. </param>
///<param name="gpuEnable"> (Optional) True if GPU enable. </param>
///<param name="name"> (Optional) The name. </param>
///<param name="inputNames"> (Optional) List of names of the inputs. </param>
///<param name="outputNames"> (Optional) List of names of the outputs. </param>
public MaxPooling(Size ksize, Size stride = new Size(), Size pad = new Size(), bool gpuEnable
= false, [CanBeNull] string name = FUNCTION_NAME, [CanBeNull] string[] inputNames = null,
[CanBeNull] string[] outputNames = null) : base(name, inputNames, outputNames)
{
    if (pad == Size.Empty)
        pad = new Size(0, 0);
}

```

```
if (stride == Size.Empty)
    stride = new Size(1, 1);
_kHeight = ksize.Height;
_kWidth = ksize.Width;
_padY = pad.Height;
_padX = pad.Width;
_strideX = stride.Width;
_strideY = stride.Height;
if (SetGpuEnable(gpuEnable))
{
    CreateKernel();
    SingleInputForward = ForwardGpu;
}
else
{
    SingleInputForward = ForwardCpu;
}

SingleOutputBackward = BackwardCpu;
}
///////////
/// <summary> Creates the kernel. </summary>
///
/// <seealso cref="M:KelpNet.Common.Functions.IParallelizable.CreateKernel()"/>
///////////

public void CreateKernel()
{
    if (GpuEnable)
        ForwardKernel = Weaver.CreateProgram(Weaver.GetKernelSource(FUNCTION_NAME)).Create
Kernel("MaxPoolingForward");
}

///////////
/// <summary> Forward CPU. </summary>
///
/// <param name="input"> The input. </param>
///
```

```
/// <returns> A NdArray. </returns>
[CanBeNull]
private NdArray ForwardCpu([NotNull] NdArray input)
{
    int outputHeight = (int)Math.Floor((input.Shape[1] - _kHeight + _padY * 2.0) / _strideY) + 1;
    int outputWidth = (int)Math.Floor((input.Shape[2] - _kWidth + _padX * 2.0) / _strideX) + 1;
    int[] outputIndices = new int[input.Shape[0] * outputHeight * outputWidth * input.
BatchCount];
    for (int b = 0; b < input.BatchCount; b++)
    {
        int resultIndex = b * input.Shape[0] * outputHeight * outputWidth;
        for (int i = 0; i < input.Shape[0]; i++)
        {
            int inputIndexOffset = b * input.Length + i * input.Shape[1] * input.Shape[2];
            for (int y = 0; y < outputHeight; y++)
            {
                int dyOffset = y * _strideY + -_padY < 0 ? 0 : y * _strideY + -_padY;
                int dyLimit = _kHeight + dyOffset < input.Shape[1] ? _kHeight + dyOffset : input.Shape[1];
                for (int x = 0; x < outputWidth; x++)
                {
                    int dxOffset = x * _strideX - _padX < 0 ? 0 : x * _strideX - _padX;
                    int dxLimit = _kWidth + dxOffset < input.Shape[2] ? _kWidth + dxOffset : input.Shape[2];
                    outputIndices[resultIndex] = inputIndexOffset + dyOffset * input.Shape[2] + dxOffset;
                    Real maxVal = input.Data[outputIndices[resultIndex]];
                    for (int dy = dyOffset; dy < dyLimit; dy++)
                    {
                        for (int dx = dxOffset; dx < dxLimit; dx++)
                        {
                            int inputIndex = inputIndexOffset + dy * input.Shape[2] + dx;
                            if (maxVal < input.Data[inputIndex])
                            {
                                maxVal = input.Data[inputIndex];
                                outputIndices[resultIndex] = inputIndex;
                            }
                        }
                    }
                }
            }
        }
    }
    resultIndex++;
}
```

```
        }
    }
}
}

return GetForwardResult(input, outputIndices, outputWidth, outputHeight);
}

//////////<summary> Forward GPU. </summary>
///
/// <param name="input"> The input. </param>
///
/// <returns> A NdArray. </returns>
//////////<summary> Forward GPU. </summary>

[CanBeNull]
private NdArray ForwardGpu([NotNull] NdArray input)
{
    int outputHeight = (int)Math.Floor((input.Shape[1] - _kHeight + _padY * 2.0) / _strideY) + 1;
    int outputWidth = (int)Math.Floor((input.Shape[2] - _kWidth + _padX * 2.0) / _strideX) + 1;
    int[] outputIndices = new int[input.Shape[0] * outputHeight * outputWidth * input.BatchCount];

    using (ComputeBuffer<Real> gpuX = new ComputeBuffer<Real>(Weaver.Context,
        ComputeMemoryFlags.ReadOnly | ComputeMemoryFlags.CopyHostPointer, input.Data))
    {
        using (ComputeBuffer<int> gpuYIndex = new ComputeBuffer<int>(Weaver.Context,
            ComputeMemoryFlags.WriteOnly | ComputeMemoryFlags.AllocateHostPointer, outputIndices.
            Length))
        {
            ForwardKernel?.SetMemoryArgument(0, gpuX);
            ForwardKernel?.SetMemoryArgument(1, gpuYIndex);
            ForwardKernel?.SetValueArgument(2, outputHeight);
            ForwardKernel?.SetValueArgument(3, outputWidth);
            ForwardKernel?.SetValueArgument(4, input.Shape[0]);
            ForwardKernel?.SetValueArgument(5, input.Shape[1]);
            ForwardKernel?.SetValueArgument(6, input.Shape[2]);
            ForwardKernel?.SetValueArgument(7, _kHeight);
            ForwardKernel?.SetValueArgument(8, _kWidth);
            ForwardKernel?.SetValueArgument(9, _strideX);
        }
    }
}
```

```
ForwardKernel?.SetValueArgument(10, _strideY);
ForwardKernel?.SetValueArgument(11, _padY);
ForwardKernel?.SetValueArgument(12, _padX);

Weaver.CommandQueue?.Execute(ForwardKernel, null, new long[] { input.BatchCount *
input.Shape[0], outputHeight, outputWidth }, null, null);
Weaver.CommandQueue?.Finish();
Weaver.CommandQueue?.ReadFromBuffer(gpuYIndex, ref outputIndices, true, null);
}

}

return GetForwardResult(input, outputIndices, outputWidth, outputHeight);
}

///////////////////////////////  

/// <summary> Gets forward result. </summary>
///  

/// <param name="input"> The input. </param>
/// <param name="outputIndices"> The output indices. </param>
/// <param name="outputWidth"> Width of the output. </param>
/// <param name="outputHeight"> Height of the output. </param>
///  

/// <returns> The forward result. </returns>
///////////////////////////////  
  

[NotNull]
NdArray GetForwardResult([NotNull] NdArray input, [NotNull] int[] outputIndices, int
outputWidth, int outputHeight)
{
    Real[] result = new Real[outputIndices.Length];
    for (int i = 0; i < result.Length; i++)
    {
        result[i] = input.Data[outputIndices[i]];
    }

    _outputIndicesList?.Add(outputIndices);
    return NdArray.Convert(result, new[] { input.Shape[0], outputHeight, outputWidth }, input.
BatchCount, this);
}
```

```
}

//////////  
/// <summary> Backward CPU. </summary>  
///  
/// <param name="y"> A NdArray to process. </param>  
/// <param name="x"> A NdArray to process. </param>  
//////////  
  
private void BackwardCpu([NotNull] NdArray y, [CanBeNull] NdArray x)  
{  
    Ensure.Argument(y).NotNull();  
    Ensure.Argument(x).NotNull();  
  
    int[] outputIndices = _outputIndicesList[_outputIndicesList.Count - 1];  
    _outputIndicesList.RemoveAt(_outputIndicesList.Count - 1);  
    for (int i = 0; i < y.Grad.Length; i++)  
    {  
        x.Grad[outputIndices[i]] += y.Grad[i];  
    }  
}
```

## AveragePooling

The AveragePooling function is similar to the MaxPooling function, except for the fact that the Average function is used for the image filter instead of the Max. Please take a look at the MaxPooling function for more information.

This is the code for the AveragePooling object:

```
//////////  
/// <summary> Average pooling. </summary>  
///  
/// <seealso cref="T:KelpNet.Common.Functions.Type.SingleInputFunction"/>  
//////////
```

```
[Serializable]  
public class AveragePooling : SingleInputFunction  
{
```

```
/// <summary> Name of the function. </summary>
const string FUNCTION_NAME = "AveragePooling";
/// <summary> The height. </summary>
private int _kHeight;
/// <summary> The width. </summary>
private int _kWidth;
/// <summary> The pad y coordinate. </summary>
private int _padY;
/// <summary> The pad x coordinate. </summary>
private int _padX;
/// <summary> The stride x coordinate. </summary>
private int _strideX;
/// <summary> The stride y coordinate. </summary>
private int _strideY;

///////////////////////////////
/// <summary>
/// Initializes a new instance of the KelpNet.Functions.Poolings.AveragePooling class.
/// </summary>
///
/// <param name="ksize"> The ksize. </param>
/// <param name="stride"> (Optional) The stride. </param>
/// <param name="pad"> (Optional) The pad. </param>
/// <param name="name"> (Optional) The name. </param>
/// <param name="inputNames"> (Optional) List of names of the inputs. </param>
/// <param name="outputNames"> (Optional) List of names of the outputs. </param>
///////////////////////////////

public AveragePooling(int ksize, int stride = 1, int pad = 0, [CanBeNull] string name = FUNCTION_NAME, [CanBeNull] string[] inputNames = null, [CanBeNull] string[] outputNames = null) :
base(name, inputNames, outputNames)
{
    _kWidth = ksize;
    _kHeight = ksize;
    _padY = pad;
    _padX = pad;
    _strideX = stride;
    _strideY = stride;
```

```
SingleInputForward = NeedPreviousForwardCpu;
SingleOutputBackward = NeedPreviousBackwardCpu;
}

///////////
/// <summary>
/// Initializes a new instance of the KelpNet.Functions.Poolings.AveragePooling class.
/// </summary>
///
/// <param name="ksize"> The ksize. </param>
/// <param name="stride"> (Optional) The stride. </param>
/// <param name="pad"> (Optional) The pad. </param>
/// <param name="name"> (Optional) The name. </param>
/// <param name="inputNames"> (Optional) List of names of the inputs. </param>
/// <param name="outputNames"> (Optional) List of names of the outputs. </param>
///////////

public AveragePooling(Size ksize, Size stride = new Size(), Size pad = new Size(), [CanBeNull]
string name = FUNCTION_NAME, [CanBeNull] string[] inputNames = null, [CanBeNull] string[]
outputNames = null) : base(name, inputNames, outputNames)
{
    if (pad == Size.Empty)
        pad = new Size(0, 0);
    if (stride == Size.Empty)
        stride = new Size(1, 1);

    _kWidth = ksize.Width;
    _kHeight = ksize.Height;
    _padY = pad.Height;
    _padX = pad.Width;
    _strideX = stride.Width;
    _strideY = stride.Height;
    SingleInputForward = NeedPreviousForwardCpu;
    SingleOutputBackward = NeedPreviousBackwardCpu;
}

///////////
/// <summary> Need previous forward CPU. </summary>
```

```
///<param name="input"> The input. </param>
///<returns> A NdArray. </returns>
////////////////////////////////////////////////////////////////////////

[NotNull]
protected NdArray NeedPreviousForwardCpu([NotNull] NdArray input)
{
    int outputHeight = (int)Math.Floor((input.Shape[1] - _kHeight + _padY * 2.0) / _strideY) + 1;
    int outputWidth = (int)Math.Floor((input.Shape[2] - _kWidth + _padX * 2.0) / _strideX) + 1;
    Real[] result = new Real[input.Shape[0] * outputHeight * outputWidth * input.BatchCount];
    Real m = _kHeight * _kWidth;

    for (int b = 0; b < input.BatchCount; b++)
    {
        int resultIndex = b * input.Shape[0] * outputHeight * outputWidth;
        for (int i = 0; i < input.Shape[0]; i++)
        {
            int inputIndexOffset = i * input.Shape[1] * input.Shape[2];
            for (int y = 0; y < outputHeight; y++)
            {
                int dyOffset = y * _strideY - _padY < 0 ? 0 : y * _strideY - _padY;
                int dyLimit = _kHeight + dyOffset < input.Shape[1] ? _kHeight + dyOffset : input.Shape[1];
                for (int x = 0; x < outputWidth; x++)
                {
                    int dxOffset = x * _strideX - _padX < 0 ? 0 : x * _strideX - _padX;
                    int dxLimit = _kWidth + dxOffset < input.Shape[2] ? _kWidth + dxOffset : input.Shape[2];
                    for (int dy = dyOffset; dy < dyLimit; dy++)
                    {
                        for (int dx = dxOffset; dx < dxLimit; dx++)
                        {
                            int inputindex = inputIndexOffset + dy * input.Shape[2] + dx;
                            result[resultIndex] += input.Data[inputindex + input.Length * b] / m;
                        }
                    }
                    resultIndex++;
                }
            }
        }
    }
}
```

```
        }
    }
}

return NdArray.Convert(result, new[] { input.Shape[0], outputHeight, outputWidth }, input.
BatchCount, this);
}

///////////////////////////////<summary> Need previous backward CPU. </summary>
///
/// <param name="y"> A NdArray to process. </param>
/// <param name="x"> A NdArray to process. </param>
///////////////////////////////<summary> Need previous backward CPU. </summary>
protected void NeedPreviousBackwardCpu([NotNull] NdArray y, [CanBeNull] NdArray x)
{
    Real m = _kHeight * _kWidth;
    for (int b = 0; b < y.BatchCount; b++)
    {
        int gyIndex = b * y.Length;
        for (int i = 0; i < x.Shape[0]; i++)
        {
            int resultIndexOffset = b * x.Length + i * x.Shape[1] * x.Shape[2];
            for (int posY = 0; posY < y.Shape[1]; posY++)
            {
                int dyOffset = posY * _strideY - _padY < 0 ? 0 : posY * _strideY - _padY;
                int dyLimit = _kHeight + dyOffset < x.Shape[1] ? _kHeight + dyOffset : x.Shape[1];
                for (int posX = 0; posX < y.Shape[2]; posX++)
                {
                    int dxOffset = posX * _strideX - _padX < 0 ? 0 : posX * _strideX - _padX;
                    int dxLimit = _kWidth + dxOffset < x.Shape[2] ? _kWidth + dxOffset : x.Shape[2];
                    Real gyData = y.Grad[gyIndex] / m;
                    for (int dy = dyOffset; dy < dyLimit; dy++)
                    {
                        for (int dx = dxOffset; dx < dxLimit; dx++)
                        {

```

## Containers

Containers in Kelp.Net are a very important concept. They basically hold all the deep learning functions that you will be using. Perhaps, the most used container is the FunctionStack, as described above and below.

The following containers are available in Kelp.Net:

- FunctionStack
  - FunctionDictionary
  - SplitFunction
  - SortedList
  - SortedFunctionStack

## FunctionStack

`FunctionStack` are basic but important principles in Kelp.Net. They are layers of functions that are executed simultaneously in one forward, backward, or update pass. You will always use `FunctionStack` to create your models, and there is more than one way to use them.

This is the code for the FunctionStack object:



```
{  
    Functions = new[] { function };  
}  
  
//////////  
/// <summary>  
/// Initializes a new instance of the KelpNet.Common.Functions.Container.FunctionStack class.  
/// </summary>  
///  
/// <param name="functions"> A variable-length parameters list containing functions. </param>  
/////////  
  
public FunctionStack([CanBeNull] params Function[] functions) : base(FUNCTION_NAME)  
{  
    Functions = new Function[]{};  
    Add(functions);  
}  
  
//////////  
/// <summary>  
/// It is not an efficient implementation because it is not assumed to be used frequently.  
/// </summary>  
///  
/// <param name="function"> A variable-length parameters list containing function. </param>  
/////////  
  
public void Add([CanBeNull] params Function[] function)  
{  
    if (function != null && function.Length > 0)  
    {  
        List<Function> functionList = new List<Function>();  
        if (Functions != null)  
        {  
            functionList.AddRange(Functions);  
        }  
        functionList.AddRange(function.Where(t => t != null));  
        Functions = functionList.ToArray();  
    }  
}
```

```
InputNames = Functions[0].InputNames;
OutputNames = Functions[Functions.Length - 1].OutputNames;
}

}

/// <summary> Compress this object. </summary>
public void Compress()
{
    List<Function> functionList = new List<Function>(Functions);
    // compress layer
    for (int i = 0; i < functionList.Count - 1; i++)
    {
        if (functionList[i] is CompressibleFunction)
        {
            if (functionList[i + 1] is CompressibleActivation)
            {
                ((CompressibleFunction)functionList[i]).SetActivation((CompressibleActivation)functionList[i + 1]);
                functionList.RemoveAt(i + 1);
            }
        }
    }
    Functions = functionList.ToArray();
}

///////////////////////////////
/// <summary> Forward. </summary>
///
/// <param name="xs"> A variable-length parameters list containing xs. </param>
///
/// <returns> A NdArray[]. </returns>
///
/// <seealso cref="M:KelpNet.Common.Functions.Function.Forward(params NdArray[])" />
/////////////////////////////
[CanBeNull]
public override NdArray[] Forward([CanBeNull] params NdArray[] xs)
{
    return Functions.Aggregate(xs, (current, t) => t.Forward(current));
```

```
}

///////////
/// <summary> Backward. </summary>
///
/// <param name="ys"> A variable-length parameters list containing ys. </param>
///
/// <seealso cref="M:KelpNet.Common.Functions.Function.Backward(params NdArray[])" />
///////////

public override void Backward([NotNull] params NdArray[] ys)
{
    NdArray.Backward(ys[0]);
}

///////////
/// <summary> Weight update process. </summary>
///
/// <seealso cref="M:KelpNet.Common.Functions.Function.Update()" />
///////////

public override void Update()
{
    foreach (var function in Functions)
    {
        function.Update();
    }
}

///////////
/// <summary>
/// Process to restore specific data to initial value after executing certain processing.
/// </summary>
///
/// <seealso cref="M:KelpNet.Common.Functions.Function.ResetState()" />
///////////

public override void ResetState()
```

```
{  
    foreach (Function function in Functions)  
    {  
        function.ResetState();  
    }  
}  
  
//////////  
/// <summary> Run the forecast. </summary>  
///  
/// <param name="xs"> A variable-length parameters list containing xs. </param>  
///  
/// <returns> A NdArray[]. </returns>  
///  
/// <seealso href="M:KelpNet.Common.Functions.Function.Predict(params NdArray[])">  
//////////  
  
[CanBeNull]  
public override NdArray[] Predict([CanBeNull] params NdArray[] xs)  
{  
    return Functions.Aggregate(xs, (current, t) => t.Predict(current));  
}  
  
//////////  
/// <summary> Sets an optimizer. </summary>  
///  
/// <param name="optimizers"> A variable-length parameters list containing optimizers. </param>  
///  
/// <seealso href="M:KelpNet.Common.Functions.Function.SetOptimizer(params Optimizer[])">  
//////////  
  
public override void SetOptimizer([CanBeNull] params Optimizer[] optimizers)  
{  
    foreach (Function function in Functions)  
    {  
        function.SetOptimizer(optimizers);  
    }  
}
```

```

    }
}
}

```

## FunctionDictionary

A FunctionDictionary is a serializable dictionary of functions used to create Caffe style data models. It is used in the Caffe data model loaders when creating new functions from data stored on disks. For those not familiar, Caffe is a deep learning framework made with expression, speed, and modularity in mind. It was developed by Berkeley AI Research as well as several community contributors. Kelp.Net supports working with Caffe models. Tests 15 and 17 demonstrate this functionality.

This is the code for the FunctionDictionary object:

```

///////////
/// <inheritdoc />
/// <summary> (Serializable) dictionary of functions. </summary>
/// <seealso cref="T:KelpNet.Common.Functions.Function" />

[Serializable]
public sealed class FunctionDictionary : Function
{
    /// <summary> Name of the function. </summary>
    const string FUNCTION_NAME = "FunctionDictionary";

    /// <summary> Manage with FunctionRecord unit with I / O key added to function. </summary>
    public Dictionary<string, FunctionStack> FunctionBlockDictionary = new Dictionary<string, FunctionStack>();
    /// <summary> a dictionary holding the name of the partitioning function. </summary>
    public Dictionary<string, FunctionStack> SplitedFunctionDictionary = new Dictionary<string, FunctionStack>();
    /// <summary> dictionary execution order list. </summary>
    public List<FunctionStack> FunctionBlocks = new List<FunctionStack>();
    /// <summary> True to compress. </summary>
    private readonly bool _compress = false;
/////////
/// <summary>
/// Initializes a new instance of the KelpNet.Common.Functions.Container.FunctionDictionary
/// class.

```



```
FunctionStack splitFunction = SplitedFunctionDictionary[function.InputNames[0]];
for (int i = 0; i < splitFunction.OutputNames.Length; i++)
{
    if (splitFunction.OutputNames[i] == function.InputNames[0])
    {
        splitFunction.OutputNames[i] = function.OutputNames[0];
        if (!SplitedFunctionDictionary.ContainsKey(function.OutputNames[0]))
        {
            SplitedFunctionDictionary.Add(function.OutputNames[0], splitFunction);
        }
    }
}
}

if (!(function is MultiOutputFunction) && // If the output branches, do not register and cut
the link
    !FunctionBlockDictionary.ContainsKey(function.OutputNames[0])) // Do not register if
already registered
{
    // Add link to dictionary
    FunctionBlockDictionary.Add(function.OutputNames[0], FunctionBlockDictionary[functio
n.InputNames[0]]);
}
else if (function is SplitFunction splitFunction)
{
    var splitFunctions = splitFunction.SplitedFunctions;
    for (int i = 0; i < splitFunctions.Length; i++)
    {
        // Add internal FunctionStack to link dictionary
        FunctionBlockDictionary.Add(splitFunction.OutputNames[i], splitFunctions[i]);
        // Add to SplitFunction's list
        SplitedFunctionDictionary.Add(splitFunction.OutputNames[i], FunctionBlockDictionary[s
plitFunction.InputNames[0]]);
    }
}

return;
}
```

```
// less processing less compression, or processing for MultiInput, DualInput
// whether the block is registered in the dictionary
if (FunctionBlockDictionary.ContainsKey(function.OutputNames[0]))
{
    // Concatenate to block if block already registered as dictionary
    FunctionBlockDictionary[function.OutputNames[0]].Add(function);
}
else
{
    // Create a new block if not registered
    FunctionStack functionRecord = new FunctionStack(function, function.Name, function.
InputNames, function.OutputNames);
    FunctionBlocks.Add(functionRecord);
    FunctionBlockDictionary.Add(function.Name, functionRecord);
}
}

///////////
//  

/// <summary> Forward. </summary>
///  

/// <param name="xs"> A variable-length parameters list containing xs. </param>
///  

/// <returns> A NdArray[]. </returns>
///  

/// <seealso cref="M:KelpNet.Common.Functions.Function.Forward(params NdArray[])" />
/////////
///  
  

[CanBeNull]
public override NdArray[] Forward([CanBeNull] params NdArray[] xs)
{
    NdArray[] result = xs;
    Dictionary<string, NdArray> outPuts = new Dictionary<string, NdArray>();
    // Register the first data in the dictionary
    for (int i = 0; i < FunctionBlocks[0].InputNames.Length; i++)
    {
        outPuts.Add(FunctionBlocks[0].InputNames[i], xs[i]);
    }
}
```

```
}

// Run in order of registration
foreach (var t in FunctionBlocks)
{
    string[] inputBlockNames = t.InputNames;
    // collect the input data
    // Implement the function
    result = t.Forward(inputBlockNames.Select(t1 => outPuts[t1]).ToArray());
    // Register the output data in the dictionary
    for (int j = 0; j < result.Length; j++)
    {
        outPuts.Add(t.OutputNames[j], result[j]);
    }
}
return result;
}

///////////////////////////////
// 
/// <summary> Backward. </summary>
///
/// <param name="ys"> A variable-length parameters list containing ys. </param>
///
/// <seealso cref="M:KelpNet.Common.Functions.Function.Backward(params NdArray[])" />
/////////////////////////////
//


public override void Backward([NotNull] params NdArray[] ys)
{
    NdArray.Backward(ys[0]);
}

/////////////////////////////
// 
/// <summary> Weight update process. </summary>
///
/// <seealso cref="M:KelpNet.Common.Functions.Function.Update()"/>
```

```
//  
  
public override void Update()  
{  
    foreach (var functionBlock in FunctionBlocks)  
    {  
        functionBlock.Update();  
    }  
}  
  
//  
/// <summary>  
/// Process to restore specific data to initial value after executing certain processing.  
/// </summary>  
///  
/// <seealso cref="M:KelpNet.Common.Functions.Function.ResetState()"/>  
//  
  
public override void ResetState()  
{  
    foreach (var functionBlock in FunctionBlocks)  
    {  
        functionBlock.ResetState();  
    }  
}  
  
//  
/// <summary> Run the forecast. </summary>  
///  
/// <param name="xs"> A variable-length parameters list containing xs. </param>  
///  
/// <returns> A NdArray[]. </returns>  
///  
/// <seealso cref="M:KelpNet.Common.Functions.Function.Predict(params NdArray[])"/>  
//
```

```
//  
  
[CanBeNull]  
public override NdArray[] Predict([CanBeNull] params NdArray[] xs)  
{  
    NdArray[] result = xs;  
    // dictionary of output data  
    Dictionary<string, NdArray> outPuts = new Dictionary<string, NdArray>();  
    // Register the output data in the dictionary  
    for (int j = 0; j < FunctionBlocks[0].InputNames.Length; j++)  
    {  
        outPuts.Add(FunctionBlocks[0].InputNames[j], xs[j]);  
    }  
  
    foreach (var t in FunctionBlocks)  
    {  
        string[] inputBlockNames = t.InputNames;  
        // collect the input data  
        // Implement the function  
        result = t.Predict(inputBlockNames.Select(t1 => outPuts[t1]).ToArray());  
        // Register the output data in the dictionary  
        for (int j = 0; j < result.Length; j++)  
        {  
            outPuts.Add(t.OutputNames[j], result[j]);  
        }  
    }  
  
    return result;  
}  
  
/////////////////////////////////////////////////////////////////////////  
//  
/// <summary> Sets an optimizer. </summary>  
///  
/// <param name="optimizers"> A variable-length parameters list containing optimizers. </param>  
///  
/// <seealso href="M:KelpNet.Common.Functions.Function.SetOptimizer(params
```

```
Optimizer[])" />
////////////////////////////////////////////////////////////////
//  
  
public override void SetOptimizer([CanBeNull] params Optimizer[] optimizers)
{
    foreach (var functionBlock in FunctionBlocks)
    {
        functionBlock.SetOptimizer(optimizers);
    }
}
```

## SplitFunction

A SplitFunction is a function that has multiple outputs.

This is the code for the SplitFunction object:

```
////////////////////////////////////////////////////////////////
/// <summary> (Serialize) a split function. </summary>
///  
/// <seealso cref="T:KelpNet.Common.Functions.Type.MultiOutputFunction"/>
////////////////////////////////////////////////////////////////  
  
[Serializable]
public class SplitFunction : MultiOutputFunction
{
    /// <summary> Name of the function. </summary>
    const string FUNCTION_NAME = "SplitFunction";
    /// <summary> The split number. </summary>
    private readonly int _splitNum;  
  
    /// <summary> The splited functions. </summary>
    public FunctionStack[] SplitedFunctions;  
  
    /// <summary>
    /// Initializes a new instance of the KelpNet.Common.Functions.Container.SplitFunction class.
    /// </summary>
```

```
///  
/// <param name="splitNum"> (Optional) The split number. </param>  
/// <param name="name"> (Optional) The name. </param>  
/// <param name="inputNames"> (Optional) List of names of the inputs. </param>  
/// <param name="outputNames"> (Optional) List of names of the outputs. </param>  
||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||  
  
public SplitFunction(int splitNum = 2, [CanBeNull] string name = FUNCTION_NAME, [CanBeNull]  
string[] inputNames = null, [CanBeNull] string[] outputNames = null) : base(name, inputNames,  
outputNames)  
{  
    _splitNum = splitNum;  
    SplitedFunctions = new FunctionStack[splitNum];  
    for (int i = 0; i < SplitedFunctions.Length; i++)  
    {  
        SplitedFunctions[i] = new FunctionStack(new Function[] { }, name + i, new[] { inputNames[0] },  
new[] { outputNames[i] });  
    }  
  
    SingleInputForward = ForwardCpu;  
    SingleOutputBackward = BackwardCpu;  
}  
||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||  
/// <summary> Forward CPU. </summary>  
///  
/// <param name="x"> A NdArray to process. </param>  
///  
/// <returns> A NdArray[]. </returns>  
||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||  
  
[NotNull]  
private NdArray[] ForwardCpu([CanBeNull] NdArray x)  
{  
    NdArray[] result = new NdArray[_splitNum];  
    for (int i = 0; i < result.Length; i++)  
    {  
        result[i] = SplitedFunctions[i].Forward(x)[0];  
    }  
}
```

```
    return result;
}

///////////
/// <summary> Backward CPU. </summary>
///
/// <param name="ys"> The ys. </param>
/// <param name="x"> A NdArray to process. </param>
///////////

private void BackwardCpu([CanBeNull] NdArray[] ys, [CanBeNull] NdArray x)
{
}

///////////
/// <summary> Evaluation function. </summary>
///
/// <param name="xs">A variable-length parameters list containing xs. </param>
///
/// <returns> A NdArray[]. </returns>
///
/// <seealso cref="M:KelpNet.Common.Functions.Function.Predict(params NdArray[])" />
///////////

[NotNull]
public override NdArray[] Predict([CanBeNull] params NdArray[] xs)
{
    NdArray[] result = new NdArray[_splitNum];
    for (int i = 0; i < result.Length; i++)
    {
        result[i] = SplitedFunctions[i].Predict(xs[0])[0];
    }
    return result;
}
```

## SortedList

A SortedList object is exactly as the name implies. It contains a list of sorted Function objects.

This is the code for the SortedList object:

```
///////////  
/// <inheritdoc />  
/// <summary> List of sorted functions. </summary>  
/// <typeparam name="T"> Generic type parameter. </typeparam>  
/// <seealso cref="T:System.Collections.Generic.ICollection{T}" />  
  
public sealed class SortedList<T> : ICollection<T>  
{  
    /// <summary> List of inners. </summary>  
    private readonly List<T> m_innerList;  
    /// <summary> The comparer. </summary>  
    private readonly Comparer<T> m_comparer;  
///////////  
/// <inheritdoc />  
/// <summary>  
/// Initializes a new instance of the KelpNet.Common.Functions.Container.SortedFunctionList<T>;  
/// T&gt; class.  
/// </summary>  
  
public SortedList() : this(Comparer<T>.Default)  
{  
}  
  
///////////  
/// <summary>  
/// Initializes a new instance of the KelpNet.Common.Functions.Container.SortedFunctionList<T>;  
/// T&gt; class.  
/// </summary>  
///  
/// <param name="comparer"> The comparer. This may be null. </param>  
///////////  
public SortedList([CanBeNull] Comparer<T> comparer)  
{
```

```
m_innerList = new List<T>();
m_comparer = comparer;
}

///////////
/// <inheritdoc />
/// <summary>
/// Adds an item to the <see cref="T:System.Collections.Generic.ICollection`1" />.
/// </summary>
/// <exception cref="T:System.NotSupportedException"> The
///           <see cref="T:System.Collections.Generic.ICollection`1" />
///           is read-only. </exception>
/// <param name="item"> The object to add to the
///           <see cref="T:System.Collections.Generic.ICollection`1" />. </param>
/// <seealso cref="M:System.Collections.Generic.ICollection{T}.Add(T)" />

public void Add(T item)
{
    int insertIndex = FindIndexForSortedInsert(m_innerList, m_comparer, item);
    m_innerList.Insert(insertIndex, item);
}

///////////
/// <summary> Adds a range. </summary>
///
/// <param name="item"> The object to add to the
///           <see cref="T:System.Collections.Generic.ICollection`1" />. </param>
///////////

public void AddRange(T[] item)
{
    foreach (T thing in item)
    {
        m_innerList?.Insert(FindIndexForSortedInsert(m_innerList, m_comparer, thing), thing);
    }
}

/////////
```

```
///<inheritdoc />
///<summary>
/// Determines whether the <see cref="T:System.Collections.Generic.ICollection`1" /> contains
a
/// specific value.
///</summary>
///<param name="item">The object to locate in the
///      <see cref="T:System.Collections.Generic.ICollection`1" />. </param>
///<returns>
///<see langword="true" /> if <paramref name="item" /> is found in the
/// <see cref="T:System.Collections.Generic.ICollection`1" />; otherwise,
///<see langword="false" />.
///</returns>
///<seealso cref="M:System.Collections.Generic.ICollection{T}.Contains(T)" />

public bool Contains(T item)
{
    return IndexOf(item) != -1;
}

///////////////////////////////
///<summary>
/// Searches for the specified object and returns the zero-based index of the first occurrence
/// within the entire SortedList<T>;
///</summary>
///
///<param name="item">The item. This may be null. </param>
///
///<returns> An int. </returns>
///////////////////////////////

public int IndexOf([CanBeNull] T item)
{
    int insertIndex = FindIndexForSortedInsert(m_innerList, m_comparer, item);
    if (insertIndex == m_innerList.Count)
    {
        return -1;
    }
}
```

```
if (m_comparer.Compare(item, m_innerList[insertIndex]) == 0)
{
    int index = insertIndex;
    while (index > 0 && m_comparer.Compare(item, m_innerList[index - 1]) == 0)
    {
        index--;
    }
    return index;
}
return -1;
}

///////////////
/// <inheritdoc />
/// <summary>
/// Removes the first occurrence of a specific object from the
/// <see cref="T:System.Collections.Generic.ICollection`1" />.
/// </summary>
/// <exception cref="T:System.NotSupportedException"> The
///           <see cref="T:System.Collections.Generic.ICollection`1" />
///           is read-only. </exception>
/// <param name="item"> The object to remove from the
///           <see cref="T:System.Collections.Generic.ICollection`1" />. </param>
/// <returns>
/// <see langword="true" /> if <paramref name="item" /> was successfully removed from the
/// <see cref="T:System.Collections.Generic.ICollection`1" />; otherwise,
/// <see langword="false" />. This method also returns <see langword="false" /> if
/// <paramref name="item" /> is not found in the original
/// <see cref="T:System.Collections.Generic.ICollection`1" />.
/// </returns>
/// <seealso cref="M:System.Collections.Generic.ICollection{T}.Remove(T)" />

public bool Remove(T item)
{
    int index = IndexOf(item);
    if (index >= 0)
    {
```

```
m_innerList?.RemoveAt(index);
return true;
}

return false;
}

///////////
/// <summary> Removes at described by index. </summary>
///
/// <param name="index"> Zero-based index of the. </param>
///////////

public void RemoveAt(int index)
{
    m_innerList?.RemoveAt(index);
}

/////////
/// <summary>
/// Copies the elements of the <see cref="T:System.Collections.Generic.ICollection`1" /> to an
/// <see cref="T:System.Array" />, starting at a particular <see cref="T:System.Array" /> index.
/// </summary>
///
/// <param name="array"> . </exception> </param>
/////////

public void CopyTo([NotNull] T[] array)
{
    m_innerList?.CopyTo(array);
}

/////////
/// <inheritdoc />
/// <summary>
/// Copies the elements of the <see cref="T:System.Collections.Generic.ICollection`1" /> to an
/// <see cref="T:System.Array" />, starting at a particular <see cref="T:System.Array" /> index.
/// </summary>
```

```
///<exception cref="T:System.ArgumentNullException"> .</exception>
///<exception cref="T:System.ArgumentOutOfRangeException"> .</exception>
///<exception cref="T:System.ArgumentException"> The number of elements in the source
///          <see cref="T:System.Collections.Generic.ICollection`1" />
///          is greater than the available space
///          from <paramref name="arrayIndex" />
///          to the end of the destination
///          <paramref name="array" />. </exception>
///<param name="array"> The one-dimensional <see cref="T:System.Array" /> that is the
///          destination of the elements copied from
///          <see cref="T:System.Collections.Generic.ICollection`1" />. The
///          <see cref="T:System.Array" /> must have zero-based indexing. </param>
///<param name="arrayIndex"> The zero-based index in <paramref name="array" /> at which
///          copying begins. </param>
///<seealso cref="M:System.Collections.Generic.ICollection{T}.CopyTo(T[],int)" />

public void CopyTo(T[] array, int arrayIndex)
{
    m_innerList?.CopyTo(array, arrayIndex);
}

///////////////
///<inheritdoc />
///<summary>
/// Removes all items from the <see cref="T:System.Collections.Generic.ICollection`1" />.
///</summary>
///<exception cref="T:System.NotSupportedException"> The
///          <see cref="T:System.Collections.Generic.ICollection`1" />
///          is read-only. </exception>
///<seealso cref="M:System.Collections.Generic.ICollection{T}.Clear()" />

public void Clear()
{
    m_innerList?.Clear();
}

/////////////
///<summary>
```

```
/// Indexer to get or set items within this collection using array index syntax.  
/// </summary>  
///  
/// <param name="index"> Zero-based index of the entry to access. </param>  
///  
/// <returns> The indexed item. This may be null. </returns>  
|||||||  
  
[CanBeNull]  
public T this[int index] => m_innerList[index];  
|||||||  
/// <inheritdoc />  
/// <summary> Gets the enumerator. </summary>  
/// <returns> The enumerator. </returns>  
  
public IEnumerator<T> GetEnumerator()  
{  
    return m_innerList.GetEnumerator();  
}  
  
|||||||  
/// <inheritdoc />  
/// <summary> Gets the enumerator. </summary>  
/// <returns> The enumerator. </returns>  
  
IEnumerator IEnumerable.GetEnumerator()  
{  
    return m_innerList.GetEnumerator();  
}  
  
|||||||  
/// <inheritdoc />  
/// <summary>  
/// Gets or sets the number of elements contained in the  
/// <see cref="T:System.Collections.Generic.ICollection`1" />.  
/// </summary>  
/// <value>  
/// The number of elements contained in the
```

```
/// <see cref="T:System.Collections.Generic.ICollection`1" />
/// </value>
/// <seealso cref="P:System.Collections.Generic.ICollection{T}.Count" />

public int Count => m_innerList.Count;

///////////////////////////////
/// <inheritdoc />
/// <summary>
/// Gets or sets a value indicating whether the
/// <see cref="T:System.Collections.Generic.ICollection`1" /> is read-only.
/// </summary>
/// <value>
/// <see langword="true" /> if the <see cref="T:System.Collections.Generic.ICollection`1" /> is
/// read-only; otherwise, <see langword="false" />.
/// </value>
/// <seealso cref="P:System.Collections.Generic.ICollection{T}.IsReadOnly" />

public bool IsReadOnly => false;

///////////////////////////////
/// <summary> Searches for the first index for sorted insert. </summary>
///
/// <param name="list"> The list. This cannot be null. </param>
/// <param name="comparer"> The comparer. This may be null. </param>
/// <param name="item"> The item. This may be null. </param>
///
/// <returns> The found index for sorted insert. </returns>
///////////////////////////////

public static int FindIndexForSortedInsert([NotNull] List<T> list, [CanBeNull] Comparer<T>
comparer, [CanBeNull] T item)
{
    if (list.Count == 0)
    {
        return 0;
    }
```

```
int lowerIndex = 0;
int upperIndex = list.Count - 1;
int comparisonResult;
while (lowerIndex < upperIndex)
{
    int middleIndex = (lowerIndex + upperIndex) / 2;
    T middle = list[middleIndex];
    comparisonResult = comparer.Compare(middle, item);
    if (comparisonResult == 0)
    {
        return middleIndex;
    }
    if (comparisonResult > 0) // middle > item
    {
        upperIndex = middleIndex - 1;
    }
    else // middle < item
    {
        lowerIndex = middleIndex + 1;
    }
}

// At this point any entry following 'middle' is greater than 'item',
// and any entry preceding 'middle' is lesser than 'item'.
// So we either put 'item' before or after 'middle'.
comparisonResult = comparer.Compare(list[lowerIndex], item);
if (comparisonResult < 0) // middle < item
{
    return lowerIndex + 1;
}
return lowerIndex;
}
```

## SortedFunctionStack

A SortedFunctionStack is a FunctionStack (stack of Function objects) which are sorted.

This is the code for the SortedFunctionStack object:

```
public class SortedFunctionStack : Function
{
    /// <summary> Name of the function. </summary>
    const string FUNCTION_NAME = "SortedFunctionStack";

    /////////////////////////////////
    /// <summary> All layers are stored here as Function class. </summary>
    ///
    /// <value> The functions. </value>
    /////////////////////////////////

    public Function[] Functions { get; private set; }

    /////////////////////////////////
    /// <inheritdoc />
    /// <summary>
    /// Initializes a new instance of the KelpNet.Common.Functions.Container.FunctionStack class.
    /// </summary>
    /// <param name="functions"> The functions. </param>
    /// <param name="name"> (Optional) The name. </param>
    /// <param name="inputNames"> (Optional) List of names of the inputs. </param>
    /// <param name="outputNames"> (Optional) List of names of the outputs. </param>

    public SortedFunctionStack([CanBeNull] Function[] functions, [CanBeNull] string name =
FUNCTION_NAME,
    [CanBeNull] string[] inputNames = null, [CanBeNull] string[] outputNames = null) : base(name,
inputNames, outputNames)
    {
        Functions = functions;
    }

    /////////////////////////////////
    /// <inheritdoc />
    /// <summary>
    /// Initializes a new instance of the KelpNet.Common.Functions.Container.FunctionStack class.
    /// </summary>
    /// <param name="function"> The function. </param>
```

```
///<param name="name"> (Optional) The name. </param>
///<param name="inputNames"> (Optional) List of names of the inputs. </param>
///<param name="outputNames"> (Optional) List of names of the outputs. </param>

public SortedFunctionStack([CanBeNull] Function function, [CanBeNull] string name =
FUNCTION_NAME,
[CanBeNull] string[] inputNames = null, [CanBeNull] string[] outputNames = null) : base(name,
inputNames, outputNames)
{
    Functions = new[] {function};
}

///////////////
///<inheritdoc />
///<summary>
/// Initializes a new instance of the KelpNet.Common.Functions.Container.FunctionStack class.
///</summary>
///<param name="functions"> A variable-length parameters list containing functions. </param>

public SortedFunctionStack([CanBeNull] params Function[] functions) : base(FUNCTION_NAME)
{
    Functions = new Function[] { };
    Add(functions);
}

///////////////
///<summary>
/// It is not an efficient implementation because it is not assumed to be used frequently.
///</summary>
///
///<param name="function"> A variable-length parameters list containing function. </param>
///

public void Add([CanBeNull] params Function[] function)
{
    if (function != null && function.Length > 0)
{
```

```
SortedList<Function> functionList = new SortedList<Function>();
if (Functions != null)
{
    functionList.AddRange(Functions);
}
IEnumerable<Function> funcs = function.Where(t => t != null);
functionList.AddRange(funcs.ToArray());
Functions = functionList.ToArray();
InputNames = Functions[0]?.InputNames;
OutputNames = Functions[Functions.Length - 1]?.OutputNames;
}

/// <summary> Compress this object. </summary>
public void Compress()
{
    List<Function> functionList = new List<Function>(Functions);

    // compress layer
    for (int i = 0; i < functionList.Count - 1; i++)
    {
        if (functionList[i] is CompressibleFunction)
        {
            if (functionList[i + 1] is CompressibleActivation)
            {
                ((CompressibleFunction) functionList[i]).SetActivation(
                    (CompressibleActivation) functionList[i + 1]);
                functionList.RemoveAt(i + 1);
            }
        }
    }
    Functions = functionList.ToArray();
}

///////////////////////////////
/// <inheritdoc />
/// <summary> Forward. </summary>
/// <param name="xs"> A variable-length parameters list containing xs. </param>
```

```
/// <returns> A NdArray[]. </returns>
/// <seealso cref="M:KelpNet.Common.Functions.Function.Forward(params NdArray[])" />

[CanBeNull]
public override NdArray[] Forward([CanBeNull] params NdArray[] xs)
{
    return Functions.Aggregate(xs, (current, t) => t.Forward(current));
}

///////////
/// <inheritdoc />
/// <summary> Backward. </summary>
/// <param name="ys"> A variable-length parameters list containing ys. </param>
/// <seealso cref="M:KelpNet.Common.Functions.Function.Backward(params NdArray[])" />

public override void Backward([NotNull] params NdArray[] ys)
{
    NdArray.Backward(ys[0]);
}

///////////
/// <inheritdoc />
/// <summary> Weight update process. </summary>
/// <seealso cref="M:KelpNet.Common.Functions.Function.Update()" />

public override void Update()
{
    foreach (var function in Functions)
    {
        function.Update();
    }
}

///////////
/// <inheritdoc />
/// <summary>
/// Process to restore specific data to initial value after executing certain processing.
/// </summary>
```

```
/// </summary>
/// <seealso cref="M:KelpNet.Common.Functions.Function.ResetState()" />

public override void ResetState()
{
    foreach (Function function in Functions)
    {
        function.ResetState();
    }
}

///////////////////////////////
/// <inheritdoc />
/// <summary> Run the forecast. </summary>
/// <param name="xs"> A variable-length parameters list containing xs. </param>
/// <returns> A NdArray[]. </returns>
/// <seealso cref="M:KelpNet.Common.Functions.Function.Predict(params NdArray[])" />

[CanBeNull]
public override NdArray[] Predict([CanBeNull] params NdArray[] xs)
{
    return Functions.Aggregate(xs, (current, t) => t.Predict(current));
}

///////////////////////////////
/// <inheritdoc />
/// <summary> Sets an optimizer. </summary>
/// <param name="optimizers"> A variable-length parameters list containing optimizers. </param>
/// <seealso cref="M:KelpNet.Common.Functions.Function.SetOptimizer(params Optimizer[])" />

public override void SetOptimizer([CanBeNull] params Optimizer[] optimizers)
{
    foreach (Function function in Functions)
    {
        function.SetOptimizer(optimizers);
    }
}
```

```
}
```

```
}
```

```
}
```

## Activation Functions

Basically, an activation function is an abstraction that represents the rate of firing potential. In its simplest form, this function will be binary, which means that the function (neuron) will either fire (be activated, be on), or not (not activated, be off). Please see *Chapter 3, Deep Instrumentation using ReflectInsight* for more information regarding Activation Functions in more detail.

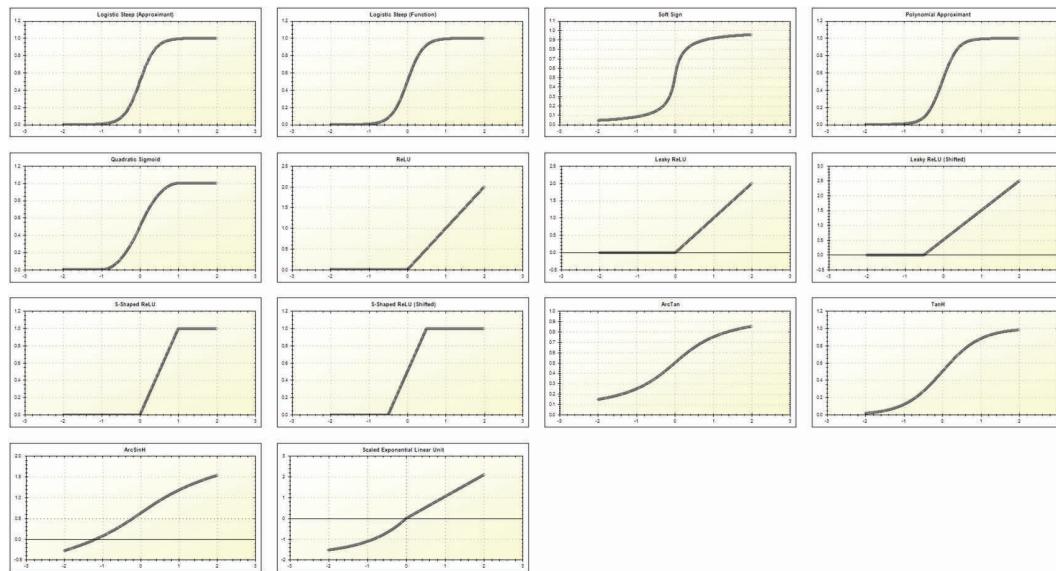
The following activation functions come with Kelp.Net:

- ArcSinH
- ArcTan
- ELU
- Gaussian
- LeakyReLU
- LeakyReLUShifted
- LogisticFunction
- MaxMinusOne
- PolynomialApproximantSteep
- QuadraticSigmoid
- RbfGaussian
- ReLU
- ReLuTanh
- ScaledELU
- Sigmoid
- Sine
- Softmax
- Softplus
- SReLU
- SReLUShifted

- Swish
- Tanh

## Activation plots

Here is a screenshot that shows how the different activation functions look like graphically when plotted:



## ArcSinH

This is the code for the ArcSinH object:

```
[Serializable]
public class ArcSinH : CompressibleActivation
{
    /// <summary> Name of the function. </summary>
    const string FUNCTION_NAME = "ArcSinH";

    /////////////////////////////////
    /// <summary>
    /// Initializes a new instance of the KelpNet.Functions.Activations.ArcSinH class.
    /// </summary>
    ///
    /// <param name="name"> (Optional) The name. </param>
```



```
/// <seealso cref="M:KelpNet.Common.Functions.CompressibleActivation.  
BackwardActivate(Real,Real)" />  
||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||  
  
internal override Real BackwardActivate(Real gy, Real y)  
{  
    return gy * (1 - y * y);  
}  
}
```

## ArcTan

This is the code for the ArcTan object:

```
[Serializable]  
public class ArcTan : CompressibleActivation  
{  
    /// <summary> Name of the function. </summary>  
    const string FUNCTION_NAME = "ArcTan";  
  
    //////////////////////////////////////////////////////////////////  
    /// <summary>  
    /// Initializes a new instance of the KelpNet.Functions.Activations.ArcTan class.  
    /// </summary>  
    //////////////////////////////////////////////////////////////////  
    /// <param name="name"> (Optional) The name. </param>  
    /// <param name="inputNames"> (Optional) List of names of the inputs. </param>  
    /// <param name="outputNames"> (Optional) List of names of the outputs. </param>  
    /// <param name="gpuEnable"> (Optional) True if GPU enable. </param>  
    //////////////////////////////////////////////////////////////////  
  
    public ArcTan([CanBeNull] string name = FUNCTION_NAME, [CanBeNull] string[] inputNames =  
        null, [CanBeNull] string[] outputNames = null, bool gpuEnable = false) : base(FUNCTION_NAME,  
        null, name, inputNames, outputNames, gpuEnable)  
    {  
    }  
  
    internal override Real ForwardActivate(Real x, [CanBeNull] Real[] args)  
    {
```

```
    return x;
}

///////////////
/// <summary> Activate virtual function used in // .Net. </summary>
///
/// <param name="x"> A Real to process. </param>
///
/// <returns> A Real. </returns>
///
/// <seealso cref="M:KelpNet.Common.Functions.CompressibleActivation.
ForwardActivate(Real)" />
///////////////

internal override Real ForwardActivate(Real x)
{
    return Math.Atan(x);
}

///////////////
/// <summary> Backward activate. </summary>
///
/// <param name="gy"> The gy. </param>
/// <param name="y"> A Real to process. </param>
///
/// <returns> A Real. </returns>
///
/// <seealso cref="M:KelpNet.Common.Functions.CompressibleActivation.
BackwardActivate(Real,Real)" />
///////////////

internal override Real BackwardActivate(Real gy, Real y)
{
    return gy * (1 - y * y);
```

```
    }  
}
```

## ELU

This is the code for the ELU object:

```
[Serializable]  
public class ELU : SingleInputFunction  
{  
    ///<summary> Name of the function. </summary>  
    const string FUNCTION_NAME = "ELU";  
  
    ///<summary> The alpha. </summary>  
    private readonly Real _alpha;  
  
    /////////////////////////////////  
    ///<summary>  
    /// Initializes a new instance of the KelpNet.Functions.Activations.ELU class.  
    ///</summary>  
    ///  
    ///<param name="alpha"> (Optional) The alpha. </param>  
    ///<param name="name"> (Optional) The name. </param>  
    ///<param name="inputNames"> (Optional) List of names of the inputs. </param>  
    ///<param name="outputNames"> (Optional) List of names of the outputs. </param>  
    /////////////////////////////////  
  
    public ELU(double alpha = 1, [CanBeNull] string name = FUNCTION_NAME, [CanBeNull]  
    string[] inputNames = null, [CanBeNull] string[] outputNames = null) : base(name, inputNames,  
    outputNames)  
    {  
        _alpha = alpha;  
        SingleInputForward = NeedPreviousForwardCpu;  
        SingleOutputBackward = NeedPreviousBackwardCpu;  
    }  
  
    /////////////////////////////////  
    ///<summary> Need previous forward CPU. </summary>
```

```
///<param name="x"> A NdArray to process. </param>
///
///<returns> A NdArray. </returns>
//////////  
  
[NotNull]
private NdArray NeedPreviousForwardCpu([NotNull] NdArray x)
{
    Real[] result = new Real[x.Data.Length];
    for (int i = 0; i < x.Data.Length; i++)
    {
        if (x.Data[i] >= 0)
            result[i] = x.Data[i];
        else
            result[i] = _alpha * (Math.Exp(x.Data[i]) - 1);
    }
    return NdArray.Convert(result, x.Shape, x.BatchCount, this);
}  
  
//////////  
///<summary> Need previous backward CPU. </summary>
///
///<param name="y"> A NdArray to process. </param>
///<param name="x"> A NdArray to process. </param>
//////////  
  
private void NeedPreviousBackwardCpu([NotNull] NdArray y, [CanBeNull] NdArray x)
{
    for (int i = 0; i < y.Grad.Length; i++)
    {
        if (x.Data[i] >= 0)
            x.Grad[i] += y.Grad[i];
        else
            x.Grad[i] += y.Grad[i] * _alpha * Math.Exp(x.Data[i]);
    }
}
```

```
}
```

## Gaussian

This is the code for the Gaussian object:

```
[Serializable]
public class Gaussian : CompressibleActivation
{
    /// <summary> Name of the function. </summary>
    const string FUNCTION_NAME = "Gaussian";

    /////////////////////////////////
    /// <summary>
    /// Initializes a new instance of the KelpNet.Functions.Activations.Gaussian class.
    /// </summary>
    ///
    /// <param name="name"> (Optional) The name. </param>
    /// <param name="inputNames"> (Optional) List of names of the inputs. </param>
    /// <param name="outputNames"> (Optional) List of names of the outputs. </param>
    /// <param name="gpuEnable"> (Optional) True if GPU enable. </param>
    /////////////////////////////////

    public Gaussian([CanBeNull] string name = FUNCTION_NAME, [CanBeNull] string[] inputNames
        = null, [CanBeNull] string[] outputNames = null, bool gpuEnable = false) : base(FUNCTION_NAME,
        null, name, inputNames, outputNames, gpuEnable)
    {
    }

    internal override Real ForwardActivate(Real x, [CanBeNull] Real[] args)
    {
        return x;
    }

    /////////////////////////////////
    /// <summary> Activate virtual function used in // .Net. </summary>
    ///
    /// <param name="x"> A Real to process. </param>
```

```
///<summary> A Real. </summary>
///<seealso cref="M:KelpNet.Common.Functions.CompressibleActivation.
ForwardActivate(Real)"/>
////////////////////////////////////////////////////////////////

internal override Real ForwardActivate(Real x)
{
    return Math.Exp(-Math.Pow(x * 2.5, 2.0));
}

////////////////////////////////////////////////////////////////
///<summary> Backward activate. </summary>
///<param name="gy"> The gy. </param>
///<param name="y"> A Real to process. </param>
///<returns> A Real. </returns>
///<seealso cref="M:KelpNet.Common.Functions.CompressibleActivation.
BackwardActivate(Real,Real)"/>
////////////////////////////////////////////////////////////////

internal override Real BackwardActivate(Real gy, Real y)
{
    return gy * (1 - y * y);
}
```

## LeakyReLU

This is the code for the LeakyReLU object:

```
[Serializable]  
public class LeakyReLU : CompressibleActivation  
{  
    /// <summary> Name of the function. </summary>  
    const string FUNCTION_NAME = "LeakyReLU";
```

```
/// <summary> Name of the parameter. </summary>
private const string PARAM_NAME = “/*slope*/”;

/// <summary> The slope. </summary>
private readonly Real _slope;

///////////////////////////////
/// <summary>
/// Initializes a new instance of the KelpNet.Functions.Activations.LeakyReLU class.
/// </summary>
///
/// <param name="slope"> (Optional) The slope. </param>
/// <param name="name"> (Optional) The name. </param>
/// <param name="inputNames"> (Optional) List of names of the inputs. </param>
/// <param name="outputNames"> (Optional) List of names of the outputs. </param>
/// <param name="gpuEnable"> (Optional) True if GPU enable. </param>
///////////////////////////////

public LeakyReLU(double slope = 0.2, [CanBeNull] string name = FUNCTION_NAME, [CanBeNull]
string[] inputNames = null, [CanBeNull] string[] outputNames = null, bool gpuEnable = false)
: base(FUNCTION_NAME, new[] { new KeyValuePair<string, string>(PARAM_NAME, slope.
ToString()) }, name, inputNames, outputNames, gpuEnable)
{
    _slope = slope;
}

internal override Real ForwardActivate(Real x, [CanBeNull] Real[] args)
{
    return x;
}

///////////////////////////////
/// <summary> Activate virtual function used in // .Net. </summary>
///
/// <param name="x"> A Real to process. </param>
///
/// <returns> A Real. </returns>
///
```

```
/// <seealso cref="M:KelpNet.Common.Functions.CompressibleActivation.  
ForwardActivate(Real)" />  
||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||  
  
internal override Real ForwardActivate(Real x)  
{  
    return x < 0 ? (Real)(x * _slope) : x;  
}  
  
||||||||||||||||||||||||||||||||||||||||||||||||||||||||  
/// <summary> Backward activate. </summary>  
///  
/// <param name="gy"> The gy. </param>  
/// <param name="y"> A Real to process. </param>  
///  
/// <returns> A Real. </returns>  
///  
/// <seealso cref="M:KelpNet.Common.Functions.CompressibleActivation.  
BackwardActivate(Real,Real)" />  
||||||||||||||||||||||||||||||||||||||||||||||||||||||||  
  
internal override Real BackwardActivate(Real gy, Real y)  
{  
    return y <= 0 ? (Real)(y * _slope) : gy;  
}  
}
```

## LeakyReLUShifted

This is the code for the LeakyReLUShifted object

```
/// Initializes a new instance of the KelpNet.Functions.Activations.LeakyReLUShifted class.  
/// </summary>  
///  
/// <param name="name"> (Optional) The name. </param>  
/// <param name="inputNames"> (Optional) List of names of the inputs. </param>  
/// <param name="outputNames"> (Optional) List of names of the outputs. </param>  
/// <param name="gpuEnable"> (Optional) True if GPU enable. </param>  
/////////////////////////////////////////////////////////////////////////  
  
public LeakyReLUShifted([CanBeNull] string name = FUNCTION_NAME, [CanBeNull] string[]  
inputNames = null, [CanBeNull] string[] outputNames = null, bool gpuEnable = false) :  
base(FUNCTION_NAME, null, name, inputNames, outputNames, gpuEnable)  
{  
}  
  
internal override Real ForwardActivate(Real x, [CanBeNull] Real[] args)  
{  
    return x;  
}  
  
/////////////////////////////////////////////////////////////////////////  
/// <summary> Activate virtual function used in // .Net. </summary>  
///  
/// <param name="x"> A Real to process. </param>  
///  
/// <returns> A Real. </returns>  
///  
/// <seealso cref="M:KelpNet.Common.Functions.CompressibleActivation.  
ForwardActivate(Real)" />  
/////////////////////////////////////////////////////////////////////////  
  
internal override Real ForwardActivate(Real x)  
{  
    const double a = 0.001;  
    const double offset = 0.5;  
    double y;  
    if (x + offset > 0.0)  
        y = x + offset;
```

```

    else
        y = (x + offset) * a;
    return y;
}

///////////////////////////////  

/// <summary> Backward activate.</summary>  

///  

/// <param name="gy"> The gy. </param>  

/// <param name="y"> A Real to process. </param>  

///  

/// <returns> A Real. </returns>  

///  

/// <seealso cref="M:KelpNet.Common.Functions.CompressibleActivation.  

BackwardActivate(Real,Real)">  

///////////////////////////////
internal override Real BackwardActivate(Real gy, Real y)
{
    return gy * (1 - y * y);
}
}

```

## LogisticFunction

This is the code for the LogisticFunction object:

```

[Serializable]
public class LogisticFunction : CompressibleActivation
{
    /// <summary> Name of the function. </summary>
    const string FUNCTION_NAME = "LogisticFunction";

///////////////////////////////
/// <summary>
/// Initializes a new instance of the KelpNet.Functions.Activations.LogisticFunction class.
/// </summary>
///  

/// <param name="name"> (Optional) The name. </param>

```



# MaxMinusOne

This is the code for the MaxMinusOne object:

```
[Serializable]
public class MaxMinusOne : CompressibleActivation
{
    /// <summary> Name of the function. </summary>
    const string FUNCTION_NAME = "MaxMinusOne";

    /////////////////////////////////////////////////
    /// <summary>
    /// Initializes a new instance of the KelpNet.Functions.Activations.MaxMinusOne class.
    /// </summary>
    ///
    /// <param name="name"> (Optional) The name. </param>
    /// <param name="inputNames"> (Optional) List of names of the inputs. </param>
    /// <param name="outputNames"> (Optional) List of names of the outputs. </param>
    /// <param name="gpuEnable"> (Optional) True if GPU enable. </param>
    /////////////////////////////////////////////////

    public MaxMinusOne([CanBeNull] string name = FUNCTION_NAME, [CanBeNull] string[]
        inputNames = null, [CanBeNull] string[] outputNames = null, bool gpuEnable = false) :
        base(FUNCTION_NAME, null, name, inputNames, outputNames, gpuEnable)
    {
    }

    internal override Real ForwardActivate(Real x, [CanBeNull] Real[] args)
```

```
{  
    return x;  
}  
  
|||||||  
/// <summary> Activate virtual function used in // .Net. </summary>  
///  
/// <param name="x"> A Real to process. </param>  
///  
/// <returns> A Real. </returns>  
///  
/// <seealso cref="M:KelpNet.Common.Functions.CompressibleActivation.  
ForwardActivate(Real)"/>  
|||||||  
  
internal override Real ForwardActivate(Real x)  
{  
    Real y;  
    if (x > -1)  
        y = x;  
    else  
        y = -1;  
    return y;  
}  
  
|||||||  
/// <summary> Backward activate. </summary>  
///  
/// <param name="gy"> The gy. </param>  
/// <param name="y"> A Real to process. </param>  
///  
/// <returns> A Real. </returns>  
///  
/// <seealso cref="M:KelpNet.Common.Functions.CompressibleActivation.  
BackwardActivate(Real,Real)"/>  
|||||||  
  
internal override Real BackwardActivate(Real gy, Real y)
```

```
{
    return gy * (1 - y * y);
}
}
```

## PolynomialApproximantSteep

This is the code for the PolynomialApproximantSteep object:

```
public class PolynomialApproximantSteep : CompressibleActivation
{
    /// <summary> Name of the function. </summary>
    const string FUNCTION_NAME = "PolynomialApproximantSteep";
    /// <summary> Name of the parameter. </summary>
    private const string PARAM_NAME = "/*slope*/";
    /// <summary> The slope. </summary>
    private readonly Real _slope = 0.00001;

    /////////////////////////////////
    /// <summary>
    /// Initializes a new instance of the KelpNet.Functions.Activations.PolynomialApproximantSteep
    /// class.
    /// </summary>
    ///
    /// <param name="slope"> (Optional) The slope. </param>
    /// <param name="name"> (Optional) The name. </param>
    /// <param name="inputNames"> (Optional) List of names of the inputs. </param>
    /// <param name="outputNames"> (Optional) List of names of the outputs. </param>
    /// <param name="gpuEnable"> (Optional) True if GPU enable. </param>
    /////////////////////////////////

    public PolynomialApproximantSteep(double slope = 0.00001, [CanBeNull] string name =
FUNCTION_NAME, [CanBeNull] string[] inputNames = null, [CanBeNull] string[] outputNames =
null, bool gpuEnable = false)
        : base(FUNCTION_NAME, new[] { new KeyValuePair<string, string>(PARAM_NAME, slope.
ToString()) }, name, inputNames, outputNames, gpuEnable)

    {
        _slope = slope;
    }
}
```

```
internal override Real ForwardActivate(Real x, [CanBeNull] Real[] args)
{
    return x;
}

///////////////////////////////  

/// <summary> Activate virtual function used in // .Net. </summary>
///  

/// <param name="x"> A Real to process. </param>
///  

/// <returns> A Real. </returns>
///  

/// <seealso cref="M:KelpNet.Common.Functions.CompressibleActivation.
ForwardActivate(Real)"/>
///////////////////////////////  
  

internal override Real ForwardActivate(Real x)
{
    x = x * 4.9;
    double x2 = x * x;
    double e = 1.0 + Math.Abs(x) + (x2 * 0.555) + (x2 * x2 * 0.143);
    double f = (x > 0) ? (1.0 / e) : e;
    return 1.0 / (1.0 + f);
}  
  

///////////////////////////////  

/// <summary> Backward activate. </summary>
///  

/// <param name="gy">The gy. </param>
/// <param name="y">A Real to process. </param>
///  

/// <returns> A Real. </returns>
///  

/// <seealso cref="M:KelpNet.Common.Functions.CompressibleActivation.
BackwardActivate(Real,Real)"/>
///////////////////////////////
```

```

internal override Real BackwardActivate(Real gy, Real y)
{
    y = y * 4.9;
    double x2 = y * y;
    double e = 1.0 + Math.Abs(y) + (x2 * 0.555) + (x2 * x2 * 0.143);
    double f = (y > 0) ? (1.0 / e) : e;
    return 1.0 / (1.0 + f);
}
}

```

## QuadraticSigmoid

This is the code for the QuadraticSigmoid object:

```

[Serializable]
public class QuadraticSigmoid : CompressibleActivation
{
    /// <summary> Name of the function. </summary>
    const string FUNCTION_NAME = "QuadraticSigmoid";

    /////////////////////////////////
    /// <summary>
    /// Initializes a new instance of the KelpNet.Functions.Activations.QuadraticSigmoid class.
    /// </summary>
    ///
    /// <param name="name"> (Optional) The name. </param>
    /// <param name="inputNames"> (Optional) List of names of the inputs. </param>
    /// <param name="outputNames"> (Optional) List of names of the outputs. </param>
    /// <param name="gpuEnable"> (Optional) True if GPU enable. </param>
    /////////////////////////////////

    public QuadraticSigmoid([CanBeNull] string name = FUNCTION_NAME, [CanBeNull] string[]
        inputNames = null, [CanBeNull] string[] outputNames = null, bool gpuEnable = false) :
        base(FUNCTION_NAME, null, name, inputNames, outputNames, gpuEnable)
    {
    }

    internal override Real ForwardActivate(Real x, [CanBeNull] Real[] args)
    {

```

```
    return x;
}

///////////////////////////////  

/// <summary> Activate virtual function used in // .Net. </summary>  

///  

/// <param name="x"> A Real to process. </param>  

///  

/// <returns> A Real. </returns>  

///  

/// <seealso cref="M:KelpNet.Common.Functions.CompressibleActivation.  
ForwardActivate(Real)" />  

///////////////////////////////  
  

internal override Real ForwardActivate(Real x)  
{  

    const double t = 0.999;  

    const double a = 0.00001;  

    double sign = Math.Sign(x);  

    x = Math.Abs(x);  

    double y = 0;  

    if (x >= 0 && x < t)  

        y = t - ((x - t) * (x - t));  

    else  

        y = t + (x - t) * a;  

    return (y * sign * 0.5) + 0.5;  

}  
  

///////////////////////////////  

/// <summary> Backward activate. </summary>  

///  

/// <param name="gy"> The gy. </param>  

/// <param name="y"> A Real to process. </param>  

///  

/// <returns> A Real. </returns>  

///  

/// <seealso cref="M:KelpNet.Common.Functions.CompressibleActivation.  
BackwardActivate(Real,Real)" />
```

.....

```
internal override Real BackwardActivate(Real gy, Real y)
{
    return gy * (1 - y * y);
}
```

## RbfGaussian

This is the code for the RbfGaussian object

```
[Serializable]
public class RbfGaussian : CompressibleActivation
{
    /// <summary> Name of the function. </summary>
    const string FUNCTION_NAME = "RbfGaussian";

    /////////////////////////////////////////////////
    /// <summary>
    /// Initializes a new instance of the KelpNet.Functions.Activations.RbfGaussian class.
    /// </summary>
    ///
    /// <param name="name"> (Optional) The name. </param>
    /// <param name="inputNames"> (Optional) List of names of the inputs. </param>
    /// <param name="outputNames"> (Optional) List of names of the outputs. </param>
    /// <param name="gpuEnable"> (Optional) True if GPU enable. </param>
    /////////////////////////////////////////////////

    public RbfGaussian([CanBeNull] string name = FUNCTION_NAME, [CanBeNull] string[]
        inputNames = null, [CanBeNull] string[] outputNames = null, bool gpuEnable = false) :
        base(FUNCTION_NAME, null, name, inputNames, outputNames, gpuEnable)
    {
    }

    internal override Real ForwardActivate(Real x)
    {
        return x;
    }
}
```

```
///////////  
/// <summary> Activate virtual function used in // .Net. </summary>  
///  
/// <param name="x"> A Real to process. </param>  
///  
/// <returns> A Real. </returns>  
///  
/// <seealso cref="M:KelpNet.Common.Functions.CompressibleActivation.  
ForwardActivate(Real)"/>  
///////////  
  
internal override Real ForwardActivate(Real x, [NotNull] Real[] args)  
{  
    // auxArgs[0] - RBF center.  
    // auxArgs[1] - RBF Gaussian epsilon.  
    double d = (x - args[0]) * Math.Sqrt(args[1]) * 4.0;  
    return Math.Exp(-(d * d));  
}  
  
///////////  
/// <summary> Backward activate. </summary>  
///  
/// <param name="gy"> The gy. </param>  
/// <param name="y"> A Real to process. </param>  
///  
/// <returns> A Real. </returns>  
///  
/// <seealso cref="M:KelpNet.Common.Functions.CompressibleActivation.  
BackwardActivate(Real,Real)"/>  
///////////  
  
internal override Real BackwardActivate(Real gy, Real y)  
{
```

```
    return gy * (1 - y * y);
}
}
```

## ReLU

This is the code for the ReLU object:

```
[Serializable]
public class ReLU : CompressibleActivation
{
    /// <summary> Name of the function. </summary>
    const string FUNCTION_NAME = "ReLU";

    /////////////////////////////////
    /// <summary>
    /// Initializes a new instance of the KelpNet.Functions.Activations.ReLU class.
    /// </summary>
    ///
    /// <param name="name"> (Optional) The name. </param>
    /// <param name="inputNames"> (Optional) List of names of the inputs. </param>
    /// <param name="outputNames"> (Optional) List of names of the outputs. </param>
    /// <param name="gpuEnable"> (Optional) True if GPU enable. </param>
    /////////////////////////////////

    public ReLU([CanBeNull] string name = FUNCTION_NAME, [CanBeNull] string[] inputNames =
        null, [CanBeNull] string[] outputNames = null, bool gpuEnable = false) : base(FUNCTION_NAME,
        null, name, inputNames, outputNames, gpuEnable)
    {
    }

    internal override Real ForwardActivate(Real x, [CanBeNull] Real[] args)
    {
        return x;
    }

    /////////////////////////////////
    /// <summary> Activate virtual function used in // .Net. </summary>
    ///
```

```
/// <param name="x"> A Real to process. </param>
///
/// <returns> A Real. </returns>
///
/// <seealso cref="M:KelpNet.Common.Functions.CompressibleActivation.
ForwardActivate(Real)" />
////////////////////////////////////////////////////////////////////////

internal override Real ForwardActivate(Real x)
{
    return x < 0 ? 0 : x;
}

////////////////////////////////////////////////////////////////////////
/// <summary> Backward activate. </summary>
///
/// <param name="gy"> The gy. </param>
/// <param name="y"> A Real to process. </param>
///
/// <returns> A Real. </returns>
///
/// <seealso cref="M:KelpNet.Common.Functions.CompressibleActivation.
BackwardActivate(Real,Real)" />
////////////////////////////////////////////////////////////////////////

internal override Real BackwardActivate(Real gy, Real y)
{
    return y <= 0 ? 0 : gy;
}
```

## ReLU Tanh

This is the code for the ReLuTanh object:

```
[Serializable]
public class ReLuTanh : CompressibleActivation
{
    /// <summary> Name of the function. </summary>
```

```
const string FUNCTION_NAME = "ReLU Tanh";
/// <summary> Name of the parameter. </summary>
private const string PARAM_NAME = /*slope*/;
/// <summary> The slope. </summary>
private readonly Real _slope = 0.2;

///////////////////////////////
/// <summary>
/// Initializes a new instance of the KelpNet.Functions.Activations.ReLuTanh class.
/// </summary>
///
/// <param name="slope"> (Optional) The slope. </param>
/// <param name="name"> (Optional) The name. </param>
/// <param name="inputNames"> (Optional) List of names of the inputs. </param>
/// <param name="outputNames"> (Optional) List of names of the outputs. </param>
/// <param name="gpuEnable"> (Optional) True if GPU enable. </param>
///////////////////////////////

public ReLuTanh(double slope = 0.2, [CanBeNull] string name = FUNCTION_NAME, [CanBeNull]
string[] inputNames = null, [CanBeNull] string[] outputNames = null, bool gpuEnable = false)
: base(FUNCTION_NAME, new[] { new KeyValuePair<string, string>(PARAM_NAME, slope.
ToString()) }, name, inputNames, outputNames, gpuEnable)
{
    _slope = slope;
}

internal override Real ForwardActivate(Real x, [CanBeNull] Real[] args)
{
    return x;
}

///////////////////////////////
/// <summary> Activate virtual function used in // .Net. </summary>
///
/// <param name="x"> A Real to process. </param>
///
/// <returns> A Real. </returns>
///
```

```
/// <seealso cref="M:KelpNet.Common.Functions.CompressibleActivation.  
ForwardActivate(Real)" />  
||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||  
  
internal override Real ForwardActivate(Real x)  
{  
    return x < 0 ? (Real)(x * _slope) * MathNet.Numerics.SpecialFunctions.Logistic(x) : x * MathNet.  
Numerics.SpecialFunctions.Logistic(x);  
}  
  
||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||  
/// <summary> Backward activate. </summary>  
///  
/// <param name="gy"> The gy. </param>  
/// <param name="y"> A Real to process. </param>  
///  
/// <returns> A Real. </returns>  
///  
/// <seealso cref="M:KelpNet.Common.Functions.CompressibleActivation.  
BackwardActivate(Real,Real)" />  
||||||||||||||||||||||||||||||||||||||||||||||||||||||||  
  
internal override Real BackwardActivate(Real gy, Real y)  
{  
    return y <= 0 ? (Real)(y * _slope) * MathNet.Numerics.SpecialFunctions.Logistic(y) : gy *  
MathNet.Numerics.SpecialFunctions.Logistic(y);  
}  
}
```

## ScaledELU

This is the code for the ScaledELU object:

```
[Serializable]  
public class ScaledELU : CompressibleActivation  
{  
    /// <summary> Name of the function. </summary>  
    const string FUNCTION_NAME = "ScaledELU";  
}  
||||||||||||||||||||||||||||||||||||||||||||||||||||
```

```
/// <summary>
/// Initializes a new instance of the KelpNet.Functions.Activations.ScaledELU class.
/// </summary>
/// <param name="name"> (Optional) The name. </param>
/// <param name="inputNames"> (Optional) List of names of the inputs. </param>
/// <param name="outputNames"> (Optional) List of names of the outputs. </param>
/// <param name="gpuEnable"> (Optional) True if GPU enable. </param>
////////////////////////////////////////////////////////////////////////

public ScaledELU([CanBeNull] string name = FUNCTION_NAME, [CanBeNull] string[] inputNames =
null, [CanBeNull] string[] outputNames = null, bool gpuEnable = false) : base(FUNCTION_NAME,
null, name, inputNames, outputNames, gpuEnable)
{
}

internal override Real ForwardActivate(Real x, [CanBeNull] Real[] args)
{
    return x;
}

////////////////////////////////////////////////////////////////////////
/// <summary> Activate virtual function used in // .Net. </summary>
///
/// <param name="x"> A Real to process. </param>
///
/// <returns> A Real. </returns>
///
/// <seealso cref="M:KelpNet.Common.Functions.CompressibleActivation.
ForwardActivate(Real)"/>
////////////////////////////////////////////////////////////////////////

internal override Real ForwardActivate(Real x)
{
    Real alpha = 1.6732632423543772848170429916717;
    Real scale = 1.0507009873554804934193349852946;
    Real y;
    if (x >= 0)
```

```
y = scale * x;
else
    y = scale * (alpha * Math.Exp(x) - alpha);
return y;
}

///////////////////////////////  

/// <summary> Backward activate. </summary>
///  

/// <param name="gy"> The gy. </param>
/// <param name="y"> A Real to process. </param>
///  

/// <returns> A Real. </returns>
///  

/// <seealso cref="M:KelpNet.Common.Functions.CompressibleActivation.
BackwardActivate(Real,Real)" />
///////////////////////////////  
  

internal override Real BackwardActivate(Real gy, Real y)
{
    return gy * (1 - y * y);
}
```

## Sigmoid

This is the code for the Sigmoid object:

```
[Serializable]
public class Sigmoid : CompressibleActivation
{
    /// <summary> Name of the function. </summary>
    const string FUNCTION_NAME = "Sigmoid";

///////////////////////////////
/// <summary>
/// Initializes a new instance of the KelpNet.Functions.Activations.Sigmoid class.
/// </summary>
///
```

```
///<param name="name"> (Optional) The name. </param>
///<param name="inputNames"> (Optional) List of names of the inputs. </param>
///<param name="outputNames"> (Optional) List of names of the outputs. </param>
///<param name="gpuEnable"> (Optional) True if GPU enable. </param>
//////////////////////////////  
  
public Sigmoid([CanBeNull] string name = FUNCTION_NAME, [CanBeNull] string[] inputNames = null, [CanBeNull] string[] outputNames = null, bool gpuEnable = false) : base(FUNCTION_NAME, null, name, inputNames, outputNames, gpuEnable)  
{  
}  
  
internal override Real ForwardActivate(Real x, [CanBeNull] Real[] args)  
{  
    return x;  
}  
  
//////////////////////////////  
///<summary> Activate virtual function used in // .Net. </summary>
///<param name="x"> A Real to process. </param>
///<returns> A Real. </returns>
///<seealso cref="M:KelpNet.Common.Functions.CompressibleActivation.  
ForwardActivate(Real)"/>
//////////////////////////////  
  
internal override Real ForwardActivate(Real x)  
{  
    return 1 / (1 + Math.Exp(-x));  
}  
  
//////////////////////////////  
///<summary> Backward activate. </summary>
///<param name="gy"> The gy. </param>
///<param name="y"> A Real to process. </param>
```

```
///  
/// <returns> A Real. </returns>  
///  
/// <seealso cref="M:KelpNet.Common.Functions.CompressibleActivation.  
BackwardActivate(Real,Real)" />  
/////////////////////////////////////////////////////////////////////////  
  
internal override Real BackwardActivate(Real gy, Real y)  
{  
    return gy * y * (1 - y);  
}  
}
```

Sine

This is the code for the Sine object:

```
[Serializable]
public class Sine : CompressibleActivation
{
    /// <summary> Name of the function. </summary>
    const string FUNCTION_NAME = "Sine";

    /////////////////////////////////////////////////
    /// <summary>
    /// Initializes a new instance of the KelpNet.Functions.Activations.Sine class.
    /// </summary>
    ///
    /// <param name="name"> (Optional) The name. </param>
    /// <param name="inputNames"> (Optional) List of names of the inputs. </param>
    /// <param name="outputNames"> (Optional) List of names of the outputs. </param>
    /// <param name="gpuEnable"> (Optional) True if GPU enable. </param>
    /////////////////////////////////////////////////

    public Sine([CanBeNull] string name = FUNCTION_NAME, [CanBeNull] string[] inputNames =
        null, [CanBeNull] string[] outputNames = null, bool gpuEnable = false) : base(FUNCTION_NAME,
        null, name, inputNames, outputNames, gpuEnable)
    {
    }
}
```

```
internal override Real ForwardActivate(Real x, [CanBeNull] Real[] args)
{
    return x;
}
///////////////////////////////
/// <summary> Activate virtual function used in // .Net. </summary>
///
/// <param name="x"> A Real to process. </param>
///
/// <returns> A Real. </returns>
///
/// <seealso cref="M:KelpNet.Common.Functions.CompressibleActivation.
ForwardActivate(Real)"/>
///////////////////////////////

internal override Real ForwardActivate(Real x)
{
    return Math.Sin(2.0 * x);
}

///////////////////////////////
/// <summary> Backward activate. </summary>
///
/// <param name="gy"> The gy. </param>
/// <param name="y"> A Real to process. </param>
///
/// <returns> A Real. </returns>
///
/// <seealso cref="M:KelpNet.Common.Functions.CompressibleActivation.
BackwardActivate(Real,Real)"/>
///////////////////////////////

internal override Real BackwardActivate(Real gy, Real y)
{
```

```
    return gy * (1 - y * y);
}
}
```

## Softmax

This is the code for the Softmax object:

```
[Serializable]
public class Softmax : SingleInputFunction
{
    /// <summary> Name of the function. </summary>
    private const string FUNCTION_NAME = "Softmax";

    /////////////////////////////////
    /// <summary>
    /// Initializes a new instance of the KelpNet.Functions.Activations.Softmax class.
    /// </summary>
    ///
    /// <param name="name"> (Optional) The name. </param>
    /// <param name="inputNames"> (Optional) List of names of the inputs. </param>
    /// <param name="outputNames"> (Optional) List of names of the outputs. </param>
    /////////////////////////////////

    public Softmax([CanBeNull] string name = FUNCTION_NAME, [CanBeNull] string[] inputNames =
        null, [CanBeNull] string[] outputNames = null) : base(name, inputNames, outputNames)
    {
        SingleInputForward = NeedPreviousForwardCpu;
        SingleOutputBackward = NeedPreviousBackwardCpu;
    }

    /////////////////////////////////
    /// <summary> Need previous forward CPU. </summary>
    ///
    /// <param name="x"> A NdArray to process. </param>
    ///
    /// <returns> A NdArray. </returns>
    ///////////////////////////////
```

```
[NotNull]
protected NdArray NeedPreviousForwardCpu([NotNull] NdArray x)
{
    Real[] y = new Real[x.Data.Length];
    int indexOffset = 0;

    for (int b = 0; b < x.BatchCount; b++)
    {
        Real maxval = x.Data[indexOffset];
        for (int i = 1; i < x.Length; i++)
        {
            if (maxval < x.Data[indexOffset + i])
                maxval = x.Data[indexOffset + i];
        }

        Real sumval = 0;
        for (int i = 0; i < x.Length; i++)
        {
            y[indexOffset + i] = Math.Exp(x.Data[indexOffset + i] - maxval);
            sumval += y[indexOffset + i];
        }
        for (int i = 0; i < x.Length; i++)
            y[indexOffset + i] /= sumval;
        indexOffset += x.Length;
    }

    return NdArray.Convert(y, x.Shape, x.BatchCount, this);
}

///////////////
/// <summary> Need previous backward CPU. </summary>
///
/// <param name="y"> A NdArray to process. </param>
/// <param name="x"> A NdArray to process. </param>
///////////////

protected void NeedPreviousBackwardCpu([NotNull] NdArray y, [NotNull] NdArray x)
{
    Real[] gx = new Real[y.Grad.Length];
```

```
int indexOffset = 0;
for (int b = 0; b < y.BatchCount; b++)
{
    Real sumdx = 0;
    for (int i = 0; i < y.Length; i++)
    {
        gx[indexOffset + i] = y.Data[indexOffset + i] * y.Data[indexOffset + i];
        sumdx += gx[indexOffset + i];
    }
    for (int i = 0; i < y.Length; i++)
        gx[indexOffset + i] -= y.Data[indexOffset + i] * sumdx;
    indexOffset += y.Length;
}
for (int i = 0; i < x.Grad.Length; i++)
    x.Grad[i] += gx[i];
}
```

## Softplus

This is the code for the Softplus object:

```
[Serializable]
public class Softplus : SingleInputFunction
{
    /// <summary> Name of the function. </summary>
    const string FUNCTION_NAME = "Softplus";

    /// <summary> The beta. </summary>
    private readonly Real _beta;
    /// <summary> The beta inverse. </summary>
    private readonly Real _betaInv;

    /////////////////////////////////
    /// <summary>
    /// Initializes a new instance of the KelpNet.Functions.Activations.Softplus class.
    /// </summary>
    ///
}
```

```
///<param name="beta"> (Optional) The beta. </param>
///<param name="name"> (Optional) The name. </param>
///<param name="inputNames"> (Optional) List of names of the inputs. </param>
///<param name="outputNames"> (Optional) List of names of the outputs. </param>
////////////////////////////////////////////////////////////////////////

public Softplus(double beta = 1, [CanBeNull] string name = FUNCTION_NAME, [CanBeNull]
string[] inputNames = null, [CanBeNull] string[] outputNames = null) : base(name, inputNames,
outputNames)
{
    _beta = beta;
    _betaInv = 1 / _beta;
    SingleInputForward = NeedPreviousForwardCpu;
    SingleOutputBackward = NeedPreviousBackwardCpu;
}

////////////////////////////////////////////////////////////////////////

///<summary> Need previous forward CPU. </summary>
///
///<param name="x"> A NdArray to process. </param>
///
///<returns> A NdArray. </returns>
////////////////////////////////////////////////////////////////////////

[NotNull]
protected NdArray NeedPreviousForwardCpu([NotNull] NdArray x)
{
    Real[] y = new Real[x.Data.Length];
    for (int b = 0; b < x.BatchCount; b++)
    {
        for (int i = 0; i < x.Length; i++)
            y[i + b * x.Length] = x.Data[i + b * x.Length] * _beta;
        Real maxval = y[b * x.Length];
        for (int i = 1; i < x.Length; i++)
        {
            if (maxval < y[i + b * x.Length])
                maxval = y[i + b * x.Length];
        }
    }
}
```

```
    for (int i = 0; i < x.Length; i++)
        y[i + b * x.Length] = (maxval + Math.Log(1.0 + Math.Exp(-Math.Abs(x.Data[i + b * x.Length]
* _beta)))) * _betaInv;
    }
    return NdArray.Convert(y, x.Shape, x.BatchCount, this);
}

///////////
/// <summary> Need previous backward CPU. </summary>
///
/// <param name="y"> A NdArray to process. </param>
/// <param name="x"> A NdArray to process. </param>
///////////

protected void NeedPreviousBackwardCpu([CanBeNull] NdArray y, [NotNull] NdArray x)
{
    for (int i = 0; i < x.Grad.Length; i++)
        x.Grad[i] += (1 - 1 / (1 + Math.Exp(_beta * y.Data[i]))) * y.Grad[i];
}
```

## SReLU

This is the code for the SReLU object:

```
[Serializable]
public class SReLU : CompressibleActivation
{
    /// <summary> Name of the function. </summary>
    const string FUNCTION_NAME = "SReLU";

/////////
/// <summary>
/// Initializes a new instance of the KelpNet.Functions.Activations.SReLU class.
/// </summary>
///
/// <param name="name"> (Optional) The name. </param>
/// <param name="inputNames"> (Optional) List of names of the inputs. </param>
/// <param name="outputNames"> (Optional) List of names of the outputs. </param>
```

```
/// <param name="gpuEnable"> (Optional) True if GPU enable. </param>
///////////////////////////// public SReLU([CanBeNull] string name = FUNCTION_NAME, [CanBeNull] string[] inputNames =
null, [CanBeNull] string[] outputNames = null, bool gpuEnable = false) : base(FUNCTION_NAME,
null, name, inputNames, outputNames, gpuEnable)
{
}

internal override Real ForwardActivate(Real x, [CanBeNull] Real[] args)
{
    return x;
}

/////////////////////////////
/// <summary> Activate virtual function used in // .Net. </summary>
///
/// <param name="x"> A Real to process. </param>
///
/// <returns> A Real. </returns>
///
/// <seealso cref="M:KelpNet.Common.Functions.CompressibleActivation.
ForwardActivate(Real)" />
/////////////////////////////

internal override Real ForwardActivate(Real x)
{
    const double tl = 0.001; // threshold (left).
    const double tr = 0.999; // threshold (right).
    const double a = 0.00001;
    double y;
    if (x > tl && x < tr)
        y = x;
    else if (x <= tl)
        y = tl + (x - tl) * a;
    else
        y = tr + (x - tr) * a;
    return y;
}
```

```
}

///////////
/// <summary> Backward activate. </summary>
///
/// <param name="gy"> The gy. </param>
/// <param name="y"> A Real to process. </param>
///
/// <returns> A Real. </returns>
///
/// <seealso cref="M:KelpNet.Common.Functions.CompressibleActivation.BackwardActivate(Real,Real)" />
///////////

internal override Real BackwardActivate(Real gy, Real y)
{
    return gy * (1 - y * y);
}
```

## SReLUShifted

This is the code for the SReLUShifted object:

```
[Serializable]
public class SReLUShifted : CompressibleActivation
{
    /// <summary> Name of the function. </summary>
    const string FUNCTION_NAME = "SReLUShifted";

/////////
/// <summary>
/// Initializes a new instance of the KelpNet.Functions.Activations.SReLUShifted class.
/// </summary>
///
/// <param name="name"> (Optional) The name. </param>
/// <param name="inputNames"> (Optional) List of names of the inputs. </param>
/// <param name="outputNames"> (Optional) List of names of the outputs. </param>
/// <param name="gpuEnable"> (Optional) True if GPU enable. </param>
```

.....

```
public SReLUShifted([CanBeNull] string name = FUNCTION_NAME, [CanBeNull] string[] inputNames = null, [CanBeNull] string[] outputNames = null, bool gpuEnable = false) : base(FUNCTION_NAME, null, name, inputNames, outputNames, gpuEnable)
{
}
```

```
internal override Real ForwardActivate(Real x, [CanBeNull] Real[] args)
{
    return x;
}
```

```
internal override Real ForwardActivate(Real x)
{
    const double tl = 0.001; // threshold (left).
    const double tr = 0.999; // threshold (right).
    const double a = 0.00001;
    const double offset = 0.5;
    double y;
    if (x + offset > tl && x + offset < tr)
        y = x + offset;
    else if (x + offset <= tl)
        y = tl + ((x + offset) - tl) * a;
    else
        y = tr + ((x + offset) - tr) * a;
    return y;
}
```

```
}

///////////
/// <summary> Backward activate. </summary>
///
/// <param name="gy"> The gy. </param>
/// <param name="y"> A Real to process. </param>
///
/// <returns> A Real. </returns>
///
/// <seealso href="M:KelpNet.Common.Functions.CompressibleActivation.BackwardActivate(Real,Real)">
///////////

internal override Real BackwardActivate(Real gy, Real y)
{
    return gy * (1 - y * y);
}
```

## Swish

This is the code for the Swish object:

```
[Serializable]
public class Swish : SingleInputFunction
{
    /// <summary> Name of the function. </summary>
    const string FUNCTION_NAME = "Swish";

/////////
/// <summary>
/// Initializes a new instance of the KelpNet.Functions.Activations.Swish class.
/// </summary>
///
/// <param name="name"> (Optional) The name. </param>
/// <param name="inputNames"> (Optional) List of names of the inputs. </param>
/// <param name="outputNames"> (Optional) List of names of the outputs. </param>
```

```
public Swish([CanBeNull] string name = FUNCTION_NAME, [CanBeNull] string[] inputNames = null, [CanBeNull] string[] outputNames = null) : base(name, inputNames, outputNames)
{
}

/////////////////////////////// <summary> Need previous forward CPU. </summary>
///
/// <param name="x"> A NdArray to process. </param>
///
/// <returns> A NdArray. </returns>
///////////////////////////////
```

[NotNull]

```
protected NdArray NeedPreviousForwardCpu([NotNull] NdArray x)
{
    Real[] result = new Real[x.Data.Length];
    for (int i = 0; i < x.Data.Length; i++)
        result[i] = x.Data[i] / (1 + Math.Exp(-x.Data[i]));
    return NdArray.Convert(result, x.Shape, x.BatchCount, this);
}
```

/////////////////////////////// <summary> Need previous backward CPU. </summary>

```
///
/// <param name="y"> A NdArray to process. </param>
/// <param name="x"> A NdArray to process. </param>
///////////////////////////////
```

```
protected void NeedPreviousBackwardCpu([NotNull] NdArray y, [CanBeNull] NdArray x)
{
    for (int i = 0; i < y.Grad.Length; i++)
        for (int j = 0; j < y.Grad[i].Length; j++)
            y.Grad[i][j] = x.Data[i] * y.Grad[i][j];
}
```

```
    x.Grad[i] += y.Grad[i] * (y.Data[i] + y.Data[i] / x.Data[i] * (1 - y.Data[i]));  
}  
}
```

## Tanh

This is the code for the Tanh object:

```
[Serializable]  
public class Tanh : CompressibleActivation  
{  
    /// <summary> Name of the function. </summary>  
    const string FUNCTION_NAME = "Tanh";  
  
    /////////////////////////////////  
    /// <summary>  
    /// Initializes a new instance of the KelpNet.Functions.Activations.Tanh class.  
    /// </summary>  
    ///  
    /// <param name="name"> (Optional) The name. </param>  
    /// <param name="inputNames"> (Optional) List of names of the inputs. </param>  
    /// <param name="outputNames"> (Optional) List of names of the outputs. </param>  
    /// <param name="gpuEnable"> (Optional) True if GPU enable. </param>  
    ///////////////////////////////  
  
    public Tanh([CanBeNull] string name = FUNCTION_NAME, [CanBeNull] string[] inputNames =  
        null, [CanBeNull] string[] outputNames = null, bool gpuEnable = false) : base(FUNCTION_NAME,  
        null, name, inputNames, outputNames, gpuEnable)  
    {  
    }  
  
    internal override Real ForwardActivate(Real x, [CanBeNull] Real[] args)  
    {  
        return x;  
    }  
  
    ///////////////////////////////  
    /// <summary> Activate virtual function used in // .Net. </summary>  
    ///
```

```
/// <param name="x"> A Real to process. </param>
///
/// <returns> A Real. </returns>
///
/// <seealso cref="M:KelpNet.Common.Functions.CompressibleActivation.
ForwardActivate(Real)" />
////////////////////////////////////////////////////////////////////////

internal override Real ForwardActivate(Real x)
{
    return Math.Tanh(x);
}

////////////////////////////////////////////////////////////////////////
/// <summary> Backward activate. </summary>
///
/// <param name="gy"> The gy. </param>
/// <param name="y"> A Real to process. </param>
///
/// <returns> A Real. </returns>
///
/// <seealso cref="M:KelpNet.Common.Functions.CompressibleActivation.
BackwardActivate(Real,Real)" />
////////////////////////////////////////////////////////////////////////

internal override Real BackwardActivate(Real gy, Real y)
{
    return gy * (1 - y * y);
}
```

## Connections

The following connections are available in Kelp.Nets:

- Convolution2D
  - Deconvolution2D
  - EmbedID

- Linear
- LSTM

## Convolution2D

This is the code for the Convolution2D object:

```
[Serializable]
public class Convolution2D : CompressibleFunction
{
    /// <summary> Name of the function. </summary>
    const string FUNCTION_NAME = "Convolution2D";
    /// <summary> Name of the parameter. </summary>
    private const string PARAM_NAME = "/*ForwardActivate*/";
    /// <summary> . </summary>
    private const string PARAM_VALUE = "localResult = ForwardActivate(localResult);";
    /// <summary> The weight. </summary>
    public NdArray Weight;
    /// <summary> The bias. </summary>
    public NdArray Bias;
    /// <summary> True to no bias. </summary>
    public readonly bool NoBias;
    /// <summary> The width. </summary>
    private readonly int _kWidth;
    /// <summary> The height. </summary>
    private readonly int _kHeight;
    /// <summary> The stride x coordinate. </summary>
    private readonly int _strideX;
    /// <summary> The stride y coordinate. </summary>
    private readonly int _strideY;
    /// <summary> The pad x coordinate. </summary>
    private readonly int _padX;
    /// <summary> The pad y coordinate. </summary>
    private readonly int _padY;
    /// <summary> Number of inputs. </summary>
    public readonly int InputCount;
    /// <summary> Number of outputs. </summary>
    public readonly int OutputCount;
```

```
public Convolution2D(int inputChannels, int outputChannels, int kSize, int stride = 1, int pad = 0,
bool noBias = false, [CanBeNull] Array initialW = null, [CanBeNull] Array initialb = null, [CanBeNull]
CompressibleActivation activation = null, [CanBeNull] string name = FUNCTION_NAME,
[CanBeNull] string[] inputNames = null, [CanBeNull] string[] outputNames = null, bool gpuEnable
= false) : base(FUNCTION_NAME, activation, new[] { new KeyValuePair<string, string>(PARAM_
NAME, PARAM_VALUE) }, name, inputNames, outputNames, gpuEnable)
{
    _kWidth = kSize;
    _kHeight = kSize;
    _strideX = stride;
    _strideY = stride;
    _padX = pad;
    _padY = pad;
    NoBias = noBias;
    Parameters = new NdArray[noBias ? 1 : 2];
    OutputCount = outputChannels;
    InputCount = inputChannels;
    Initialize(initialW, initialb);
}
```

```
public Convolution2D(int inputChannels, int outputChannels, Size kSize, Size stride = new  
Size(), Size pad = new Size(), bool noBias = false, [CanBeNull] Array initialW = null, [CanBeNull]  
Array initialb = null, [CanBeNull] CompressibleActivation activation = null, [CanBeNull] string[]  
name = FUNCTION_NAME, [CanBeNull] string[] inputNames = null, [CanBeNull] string[]  
outputNames = null, bool gpuEnable = false) : base(FUNCTION_NAME, activation, new[] { new  
KeyValuePair<string, string>(PARAM_NAME, PARAM_VALUE) }, name, inputNames, outputNames,  
gpuEnable)  
{  
    if (pad == Size.Empty)  
        pad = new Size(0, 0);  
  
    if (stride == Size.Empty)  
        stride = new Size(1, 1);  
  
    _kWidth = kSize.Width;  
    _kHeight = kSize.Height;  
    _strideX = stride.Width;  
    _strideY = stride.Height;
```

```
_padX = pad.Width;
_padY = pad.Height;
NoBias = noBias;
Parameters = new NdArray[noBias ? 1 : 2];
OutputCount = outputChannels;
InputCount = inputChannels;
Initialize(initialW, initialb);
}

///////////////
/// <summary>
/// Initializes a new instance of the KelpNet.Functions.Connections.Convolution2D class.
/// </summary>
///
/// <param name="linear"> The linear. </param>
///////////////

public Convolution2D([NotNull] Linear linear) : base(FUNCTION_NAME, linear.Activator, new[]
{ new KeyValuePair<string, string>(PARAM_NAME, PARAM_VALUE) }, linear.Name, linear.
InputNames, linear.OutputNames, linear.GpuEnable)
{
    _kWidth = 1;
    _kHeight = 1;
    _strideX = 1;
    _strideY = 1;
    _padX = 0;
    _padY = 0;
    Parameters = linear.Parameters;
    Weight = linear.Weight;
    Weight.Reshape(OutputCount, InputCount, _kHeight, _kWidth);
    Bias = linear.Bias;
    NoBias = linear.NoBias;
}

///////////////
/// <summary> Initializes this object. </summary>
///
/// <param name="initialW"> (Optional) The initial w. </param>
```

```
/// <param name="initialb"> (Optional) The initialb. </param>
//////////////////////////////  
  
void Initialize([CanBeNull] Array initialW = null, [CanBeNull] Array initialb = null)  
{  
    Weight = new NdArray(OutputCount, InputCount, _kHeight, _kWidth)  
    {  
        Name = Name + " Weight"  
    };  
    if (initialW == null)  
        Initializer.InitWeight(Weight);  
    else  
        Weight.Data = Real.GetArray(initialW);  
    Parameters[0] = Weight;  
    if (!NoBias)  
    {  
        Bias = new NdArray(OutputCount)  
        {  
            Name = Name + " Bias"  
        };  
        if (initialb != null)  
            Bias.Data = Real.GetArray(initialb);  
        Parameters[1] = Bias;  
    }  
}  
  
//////////////////////////////  
/// <summary> Need previous forward CPU. </summary>  
///  
/// <param name="input"> The input. </param>  
///  
/// <returns> A NdArray. </returns>  
///  
/// <seealso href="M:KelpNet.Common.Functions.CompressibleFunction.NeedPreviousForwardCpu(NdArray)">/>  
//////////////////////////////
```

```
protected override NdArray NeedPreviousForwardCpu([NotNull] NdArray input)
{
    int outputHeight = (int)Math.Floor((input.Shape[1] - _kHeight + _padY * 2.0) / _strideY) + 1;
    int outputWidth = (int)Math.Floor((input.Shape[2] - _kWidth + _padX * 2.0) / _strideX) + 1;
    Real[] result = new Real[OutputCount * outputHeight * outputWidth * input.BatchCount];
    for (int batchCounter = 0; batchCounter < input.BatchCount; batchCounter++)
    {
        int resultIndex = batchCounter * OutputCount * outputHeight * outputWidth;
        for (int och = 0; och < OutputCount; och++)
        {
            //For W index
            int outChOffset = och * InputCount * _kHeight * _kWidth;
            for (int oy = 0; oy < outputHeight * _strideY; oy += _strideY)
            {
                int kyStartIndex = oy - _padY < 0 ? 0 : oy - _padY;
                int kyLimit = _kHeight + oy - _padY < input.Shape[1] ? _kHeight + oy - _padY : input.
Shape[1];
                for (int ox = 0; ox < outputWidth * _strideX; ox += _strideX)
                {
                    int kxStartIndex = ox - _padX < 0 ? 0 : ox - _padX;
                    int kxLimit = _kWidth + ox - _padX < input.Shape[2] ? _kWidth + ox - _padX : input.
Shape[2];
                    for (int ich = 0; ich < InputCount; ich++)
                    {
                        // for W index
                        int inChOffset = ich * _kHeight * _kWidth;
                        // for input index
                        int inputOffset = ich * input.Shape[1] * input.Shape[2];
                        for (int ky = kyStartIndex; ky < kyLimit; ky++)
                        {
                            for (int kx = kxStartIndex; kx < kxLimit; kx++)
                            {
                                int wIndex = outChOffset + inChOffset + (ky - oy + _padY) * _kWidth + kx - ox + _padX;
                                int inputIndex = inputOffset + ky * input.Shape[2] + kx + batchCounter * input.Length;
                                result[resultIndex] += input.Data[inputIndex] * Weight.Data[wIndex];
                            }
                        }
                    }
                }
            }
        }
    }
}
```

```
        resultIndex++;
    }
}
}
}
if (Activator != null && !NoBias)
{
    for (int batchCounter = 0; batchCounter < input.BatchCount; batchCounter++)
    {
        int resultIndex = batchCounter * OutputCount * outputHeight * outputWidth;
        for (int och = 0; och < OutputCount; och++)
        {
            for (int location = 0; location < outputHeight * outputWidth; location++)
            {
                result[resultIndex] += Bias.Data[och];
                result[resultIndex] = Activator.ForwardActivate(result[resultIndex]);
                resultIndex++;
            }
        }
    }
}
else if (!NoBias)
{
    for (int batchCounter = 0; batchCounter < input.BatchCount; batchCounter++)
    {
        int resultIndex = batchCounter * OutputCount * outputHeight * outputWidth;
        for (int och = 0; och < OutputCount; och++)
        {
            for (int location = 0; location < outputHeight * outputWidth; location++)
            {
                result[resultIndex] += Bias.Data[och];
                resultIndex++;
            }
        }
    }
}
else if (Activator != null)
{
```

```

for (int batchCounter = 0; batchCounter < input.BatchCount; batchCounter++)
{
    int resultIndex = batchCounter * OutputCount * outputHeight * outputWidth;
    for (int och = 0; och < OutputCount; och++)
    {
        for (int location = 0; location < outputHeight * outputWidth; location++)
        {
            result[resultIndex] = Activator.ForwardActivate(result[resultIndex]);
            resultIndex++;
        }
    }
}
return NdArray.Convert(result, new[] { OutputCount, outputHeight, outputWidth }, input.
BatchCount, this);
}

///////////////
/// <summary> Need previous forward GPU. </summary>
///
/// <param name="input"> The input. </param>
///
/// <returns> A NdArray. </returns>
///
/// <seealso cref="M:KelpNet.Common.Functions.CompressibleFunction.NeedPreviousForward
Gpu(NdArray)" />
///////////////

[NotNull]
protected override NdArray NeedPreviousForwardGpu([NotNull] NdArray input)
{
    int outputHeight = (int)Math.Floor((input.Shape[1] - _kHeight + _padY * 2.0) / _strideY) + 1;
    int outputWidth = (int)Math.Floor((input.Shape[2] - _kWidth + _padX * 2.0) / _strideX) + 1;
    Real[] result = new Real[OutputCount * outputHeight * outputWidth * input.BatchCount];

    using (ComputeBuffer<Real> gpuX = new ComputeBuffer<Real>(Weaver.Context,
ComputeMemoryFlags.ReadOnly | ComputeMemoryFlags.CopyHostPointer, input.Data))
    {

```

```
using (ComputeBuffer<Real> gpuW = new ComputeBuffer<Real>(Weaver.Context,
ComputeMemoryFlags.ReadOnly | ComputeMemoryFlags.CopyHostPointer, Weight.Data))
{
    using (ComputeBuffer<Real> gpub = new ComputeBuffer<Real>(Weaver.Context,
ComputeMemoryFlags.ReadOnly | ComputeMemoryFlags.CopyHostPointer, NoBias ? new
Real[OutputCount] : Bias.Data))
    {
        using (ComputeBuffer<Real> gpuY = new ComputeBuffer<Real>(Weaver.Context,
ComputeMemoryFlags.WriteOnly | ComputeMemoryFlags.AllocateHostPointer, result.Length))
        {
            ForwardKernel.SetMemoryArgument(0, gpuX);
            ForwardKernel.SetMemoryArgument(1, gpuW);
            ForwardKernel.SetMemoryArgument(2, gpub);
            ForwardKernel.SetMemoryArgument(3, gpuY);
            ForwardKernel.SetValueArgument(4, input.Shape[1]);
            ForwardKernel.SetValueArgument(5, input.Shape[2]);
            ForwardKernel.SetValueArgument(6, input.Length);
            ForwardKernel.SetValueArgument(7, outputWidth);
            ForwardKernel.SetValueArgument(8, outputHeight);
            ForwardKernel.SetValueArgument(9, _strideX);
            ForwardKernel.SetValueArgument(10, _strideY);
            ForwardKernel.SetValueArgument(11, _padX);
            ForwardKernel.SetValueArgument(12, _padY);
            ForwardKernel.SetValueArgument(13, _kHeight);
            ForwardKernel.SetValueArgument(14, _kWidth);
            ForwardKernel.SetValueArgument(15, OutputCount);
            ForwardKernel.SetValueArgument(16, InputCount);

            Weaver.CommandQueue.Execute(ForwardKernel, null, new long[] { input.BatchCount *
OutputCount, outputHeight, outputWidth }, null, null);
            Weaver.CommandQueue.Finish();
            Weaver.CommandQueue.ReadFromBuffer(gpuY, ref result, true, null);
        }
    }
}

return NdArray.Convert(result, new[] { OutputCount, outputHeight, outputWidth }, input.
BatchCount, this);
}
```



```
int gyIndex = 0;
for (int batchCounter = 0; batchCounter < batchCount; batchCounter++)
{
    for (int och = 0; och < gyShape[0]; och++)
    {
        for (int olocation = 0; olocation < gyShape[1] * gyShape[2]; olocation++)
        {
            Bias.Grad[och] += gy[gyIndex];
            gyIndex++;
        }
    }
}
}

///////////////////////////////
/// <summary> Need previous backward CPU. </summary>
///
/// <param name="y"> A NdArray to process. </param>
/// <param name="x"> A NdArray to process. </param>
///
/// <seealso href="M:KelpNet.Common.Functions.CompressibleFunction.NeedPreviousBackwar
dCpu(NdArray,NdArray)">
///////////////////////////////

protected override void NeedPreviousBackwardCpu([NotNull] NdArray y, [CanBeNull] NdArray x)
{
    Real[] activatedgy = Activator != null ? GetActivatedGY(y) : y.Grad;
    if (!NoBias) CalcBiasGrad(activatedgy, y.Shape, y.BatchCount);
    for (int batchCounter = 0; batchCounter < y.BatchCount; batchCounter++)
    {
        for (int och = 0; och < y.Shape[0]; och++)
        {
            //For gW index
            int outChOffset = och * InputCount * _kHeight * _kWidth;
            for (int oy = 0; oy < y.Shape[1] * _strideY; oy += _strideY)
            {
                // Jump due to omission of calculation
                int kyStartIndex = _padY - oy < 0 ? 0 : _padY - oy;
```

```
int kyLimit = _kHeight < x.Shape[1] - oy + _padY ? _kHeight : x.Shape[1] - oy + _padY;
for (int ox = 0; ox < y.Shape[2] * _strideX; ox += _strideX)
{
    // Jump due to omission of calculation
    int kxStartIndex = _padX - ox < 0 ? 0 : _padX - ox;
    int kxLimit = _kWidth < x.Shape[2] - ox + _padX ? _kWidth : x.Shape[2] - ox + _padX;
    int gyIndex = batchCounter * y.Length + och * y.Shape[1] * y.Shape[2] + oy * y.Shape[2] +
ox;
    Real gyData = activatedgy[gyIndex];
    for (int ich = 0; ich < x.Shape[0]; ich++)
    {
        // for gW index
        int inChOffset = ich * _kHeight * _kWidth;
        //input index
        int inputOffset = ich * x.Shape[1] * x.Shape[2] + batchCounter * x.Length;
        for (int ky = kyStartIndex; ky < kyLimit; ky++)
        {
            for (int kx = kxStartIndex; kx < kxLimit; kx++)
            {
                // The shapes of W and gW are equal
                int wIndex = outChOffset + inChOffset + ky * _kWidth + kx;
                // The shapes of x and gx are equal
                int inputIndex = inputOffset + (ky + oy - _padY) * x.Shape[2] + kx + ox - _padX;
                Weight.Grad[wIndex] += x.Data[inputIndex] * gyData;
                x.Grad[inputIndex] += Weight.Data[wIndex] * gyData;
            }
        }
    }
}
}

////////////////////////////////////////////////////////////////////////
/// <summary> Need previous backward GPU. </summary>
///
/// <param name="y"> A NdArray to process. </param>
```

```
/// <param name="x"> A NdArray to process. </param>
///
/// <seealso cref="M:KelpNet.Common.Functions.CompressibleFunction.NeedPreviousBackwar
dGpu(NdArray,NdArray)" />
////////////////////////////////////////////////////////////////////////

protected override void NeedPreviousBackwardGpu([NotNull] NdArray y, [NotNull] NdArray x)
{
    Real[] gx = new Real[x.Data.Length];
    Real[] activatedgy = Activator != null ? GetActivatedGY(y) : y.Grad;
    if (!NoBias) CalcBiasGrad(activatedgy, y.Shape, y.BatchCount);
    // gy is used in common
    using (ComputeBuffer<Real> gpugY = new ComputeBuffer<Real>(Weaver.Context,
        ComputeMemoryFlags.ReadOnly | ComputeMemoryFlags.CopyHostPointer, activatedgy))
    {
        using (ComputeBuffer<Real> gpugW = new ComputeBuffer<Real>(Weaver.Context,
            ComputeMemoryFlags.ReadWrite | ComputeMemoryFlags.CopyHostPointer, Weight.Grad))
        {
            using (ComputeBuffer<Real> gpuX = new ComputeBuffer<Real>(Weaver.Context,
                ComputeMemoryFlags.ReadOnly | ComputeMemoryFlags.CopyHostPointer, x.Data))
            {
                BackwardgWKernel.SetValueArgument(0, gpugY);
                BackwardgWKernel.SetValueArgument(1, gpugW);
                BackwardgWKernel.SetValueArgument(2, gpugY);
                BackwardgWKernel.SetValueArgument(3, y.BatchCount);
                BackwardgWKernel.SetValueArgument(4, InputCount);
                BackwardgWKernel.SetValueArgument(5, y.Shape[0]);
                BackwardgWKernel.SetValueArgument(6, y.Shape[1]);
                BackwardgWKernel.SetValueArgument(7, y.Shape[2]);
                BackwardgWKernel.SetValueArgument(8, x.Shape[1]);
                BackwardgWKernel.SetValueArgument(9, x.Shape[2]);
                BackwardgWKernel.SetValueArgument(10, x.Length);
                BackwardgWKernel.SetValueArgument(11, _strideX);
                BackwardgWKernel.SetValueArgument(12, _strideY);
                BackwardgWKernel.SetValueArgument(13, _padX);
                BackwardgWKernel.SetValueArgument(14, _padY);
                BackwardgWKernel.SetValueArgument(15, _kHeight);
                BackwardgWKernel.SetValueArgument(16, _kWidth);
```

```
    Weaver.CommandQueue.Execute(BackwardgWKernel, null, new long[] { OutputCount *
InputCount, _kHeight, _kWidth },
    null, null);
    Weaver.CommandQueue.Finish();
    Weaver.CommandQueue.ReadFromBuffer(gpugW, ref Weight.Grad, true, null);
}
}

using (ComputeBuffer<Real> gpugX = new ComputeBuffer<Real>(Weaver.Context,
ComputeMemoryFlags.WriteOnly | ComputeMemoryFlags.AllocateHostPointer, gx.Length))
{
    using (ComputeBuffer<Real> gpuW = new ComputeBuffer<Real>(Weaver.Context,
ComputeMemoryFlags.ReadOnly | ComputeMemoryFlags.CopyHostPointer, Weight.Data))
    {
        BackwardgXKernel.SetMemoryArgument(0, gpugY);
        BackwardgXKernel.SetMemoryArgument(1, gpuW);
        BackwardgXKernel.SetMemoryArgument(2, gpugX);
        BackwardgXKernel.SetValueArgument(3, OutputCount);
        BackwardgXKernel.SetValueArgument(4, InputCount);
        BackwardgXKernel.SetValueArgument(5, y.Shape[0]);
        BackwardgXKernel.SetValueArgument(6, y.Shape[1]);
        BackwardgXKernel.SetValueArgument(7, y.Shape[2]);
        BackwardgXKernel.SetValueArgument(8, x.Shape[1]);
        BackwardgXKernel.SetValueArgument(9, x.Shape[2]);
        BackwardgXKernel.SetValueArgument(10, x.Length);
        BackwardgXKernel.SetValueArgument(11, _strideX);
        BackwardgXKernel.SetValueArgument(12, _strideY);
        BackwardgXKernel.SetValueArgument(13, _padX);
        BackwardgXKernel.SetValueArgument(14, _padY);
        BackwardgXKernel.SetValueArgument(15, _kHeight);
        BackwardgXKernel.SetValueArgument(16, _kWidth);

        Weaver.CommandQueue.Execute(BackwardgXKernel, null, new long[] { y.BatchCount *
x.Shape[0], x.Shape[1], x.Shape[2] },
        null, null);
        Weaver.CommandQueue.Finish();
        Weaver.CommandQueue.ReadFromBuffer(gpugX, ref gx, true, null);
    }
}
```

```
    }
    for (int i = 0; i < x.Grad.Length; i++)
        x.Grad[i] += gx[i];
}
}
```

## Deconvolution2D

This is the code for the Deconvolution2D object:

```
[Serializable]
public class Deconvolution2D : CompressibleFunction
{
    ///<summary> Name of the function. </summary>
    const string FUNCTION_NAME = "Deconvolution2D";
    ///<summary> Name of the parameter. </summary>
    private const string PARAM_NAME = /*ForwardActivate*/;
    ///<summary> . </summary>
    private const string PARAM_VALUE = "result = ForwardActivate(result);"

    ///<summary> The weight. </summary>
    public NdArray Weight;
    ///<summary> The bias. </summary>
    public NdArray Bias;

    ///<summary> True to no bias. </summary>
    public readonly bool NoBias;

    ///<summary> The width. </summary>
    private readonly int _kWidth;
    ///<summary> The height. </summary>
    private readonly int _kHeight;
    ///<summary> The sub sample x coordinate. </summary>
    private readonly int _subSampleX;
    ///<summary> The sub sample y coordinate. </summary>
    private readonly int _subSampleY;
    ///<summary> The trim x coordinate. </summary>
    private readonly int _trimX;
    ///<summary> The trim y coordinate. </summary>
```

```
private readonly int _trimY;
/// <summary> Number of inputs. </summary>
public readonly int InputCount;
/// <summary> Number of outputs. </summary>
public readonly int OutputCount;

///////////////////////////////
/// <summary>
/// Initializes a new instance of the KelpNet.Functions.Connections.Deconvolution2D class.
/// </summary>
///
/// <param name="inputChannels"> The input channels. </param>
/// <param name="outputChannels"> The output channels. </param>
/// <param name="kSize"> The size. </param>
/// <param name="subSample"> (Optional) The sub sample. </param>
/// <param name="trim"> (Optional) The trim. </param>
/// <param name="noBias"> (Optional) True to no bias. </param>
/// <param name="initialW"> (Optional) The initial w. </param>
/// <param name="initialb"> (Optional) The initialb. </param>
/// <param name="activation"> (Optional) The activation. </param>
/// <param name="name"> (Optional) The name. </param>
/// <param name="inputNames"> (Optional) List of names of the inputs. </param>
/// <param name="outputNames"> (Optional) List of names of the outputs. </param>
/// <param name="gpuEnable"> (Optional) True if GPU enable. </param>
///////////////////////////////

public Deconvolution2D(int inputChannels, int outputChannels, int kSize, int subSample = 1, int
trim = 0, bool noBias = false, [CanBeNull] Array initialW = null, [CanBeNull] Array initialb = null,
[CanBeNull] CompressibleActivation activation = null, [CanBeNull] string name = FUNCTION_
NAME, [CanBeNull] string[] inputNames = null, [CanBeNull] string[] outputNames = null, bool
gpuEnable = false) : base(FUNCTION_NAME, activation, new []{new KeyValuePair<string,
string>(PARAM_NAME, PARAM_VALUE)}, name, inputNames, outputNames, gpuEnable)
{
    _kWidth = kSize;
    _kHeight = kSize;
    _trimX = trim;
    _trimY = trim;
    _subSampleX = subSample;
    _subSampleY = subSample;
```

```
NoBias = noBias;

Parameters = new NdArray[noBias ? 1 : 2];
OutputCount = outputChannels;
InputCount = inputChannels;
Initialize(initialW, initialb);
}

///////////////
/// <summary>
/// Initializes a new instance of the KelpNet.Functions.Connections.Deconvolution2D class.
/// </summary>
///
/// <param name="inputChannels"> The input channels. </param>
/// <param name="outputChannels"> The output channels. </param>
/// <param name="kSize"> The size. </param>
/// <param name="subSample"> (Optional) The sub sample. </param>
/// <param name="trim"> (Optional) The trim. </param>
/// <param name="noBias"> (Optional) True to no bias. </param>
/// <param name="initialW"> (Optional) The initial w. </param>
/// <param name="initialb"> (Optional) The initialb. </param>
/// <param name="activation"> (Optional) The activation. </param>
/// <param name="name"> (Optional) The name. </param>
/// <param name="inputNames"> (Optional) List of names of the inputs. </param>
/// <param name="outputNames"> (Optional) List of names of the outputs. </param>
/// <param name="gpuEnable"> (Optional) True if GPU enable. </param>
///////////////

public Deconvolution2D(int inputChannels, int outputChannels, Size kSize, Size subSample = new
Size(), Size trim = new Size(), bool noBias = false, [CanBeNull] Array initialW = null, [CanBeNull]
Array initialb = null, [CanBeNull] CompressibleActivation activation = null, [CanBeNull] string
name = FUNCTION_NAME, [CanBeNull] string[] inputNames = null, [CanBeNull] string[]
outputNames = null, bool gpuEnable = false) : base(FUNCTION_NAME, activation, new []{new
KeyValuePair<string, string>(PARAM_NAME, PARAM_VALUE)}, name, inputNames, outputNames,
gpuEnable)
{
    if (subSample == Size.Empty)
        subSample = new Size(1, 1);
    if (trim == Size.Empty)
```

```
trim = new Size(0, 0);

_kWidth = kSize.Width;
_kHeight = kSize.Height;
_trimX = trim.Width;
_trimY = trim.Height;
NoBias = noBias;
_subSampleX = subSample.Width;
_subSampleY = subSample.Height;
Parameters = new NdArray[noBias ? 1 : 2];
OutputCount = outputChannels;
InputCount = inputChannels;
Initialize(initialW, initialb);
}

///////////////
/// <summary> Initializes this object. </summary>
///
/// <param name="initialW"> (Optional) The initial w. </param>
/// <param name="initialb"> (Optional) The initialb. </param>
///////////////

void Initialize([CanBeNull] Array initialW = null, [CanBeNull] Array initialb = null)
{
    Weight = new NdArray(OutputCount, InputCount, _kHeight, _kWidth);
    Weight.Name = Name + " Weight";

    if (initialW == null)
    {
        Initializer.InitWeight(Weight);
    }
    else
    {
        Weight.Data = Real.GetArray(initialW);
    }
    Parameters[0] = Weight;
    if (!NoBias)
    {
```

```
Bias = new NdArray(OutputCount) {Name = Name + " Bias"};
if (initialb != null)
{
    Bias.Data = Real.GetArray(initialb);
}
Parameters[1] = Bias;
}

///////////////
/// <summary> Need previous forward CPU. </summary>
///
/// <param name="input"> The input. </param>
///
/// <returns> A NdArray. </returns>
///
/// <seealso cref="M:KelpNet.Common.Functions.CompressibleFunction.NeedPreviousForward
Cpu(NdArray)" />
///////////////

[NotNull]
protected override NdArray NeedPreviousForwardCpu([NotNull] NdArray input)
{
    int outputHeight = (input.Shape[1] - 1) * _subSampleY + _kHeight - _trimY * 2;
    int outputWidth = (input.Shape[2] - 1) * _subSampleX + _kWidth - _trimX * 2;
    Real[] result = new Real[input.BatchCount * OutputCount * outputWidth * outputHeight];
    int outSizeOffset = outputWidth * outputHeight;
    int inputSizeOffset = input.Shape[1] * input.Shape[2];
    int kSizeOffset = Weight.Shape[2] * Weight.Shape[3];

    for (int batchCount = 0; batchCount < input.BatchCount; batchCount++)
    {
        for (int och = 0; och < OutputCount; och++)
        {
            for (int oy = _trimY; oy < outputHeight + _trimY; oy++)
            {
                int iyLimit = oy / _subSampleY + 1 < input.Shape[1] ? oy / _subSampleY + 1 : input.Shape[1];
                int iyStart = oy - Weight.Shape[2] < 0 ? 0 : (oy - Weight.Shape[2]) / _subSampleY + 1;
```

```
for (int ox = _trimX; ox < outputWidth + _trimX; ox++)
{
    int ixLimit = ox / _subSampleX + 1 < input.Shape[2] ? ox / _subSampleX + 1 : input.
Shape[2];
    int ixStart = ox - Weight.Shape[3] < 0 ? 0 : (ox - Weight.Shape[3]) / _subSampleX + 1;
    int outputIndex = batchCount * OutputCount * outSizeOffset + och * outSizeOffset + (oy
- _trimY) * outputWidth + ox - _trimX;
    for (int ich = 0; ich < input.Shape[0]; ich++)
    {
        int inputIndexOffset = batchCount * input.Length + ich * inputSizeOffset;
        int kernelIndexOffset = och * Weight.Shape[1] * kSizeOffset + ich * kSizeOffset;
        for (int iy = iyStart; iy < iyLimit; iy++)
        {
            for (int ix = ixStart; ix < ixLimit; ix++)
            {
                int inputIndex = inputIndexOffset + iy * input.Shape[2] + ix;
                int kernelIndex = kernelIndexOffset + (oy - iy * _subSampleY) * Weight.Shape[3] + (ox -
ix * _subSampleX);
                result[outputIndex] += input.Data[inputIndex] * Weight.Data[kernelIndex];
            }
        }
    }
}
if (Activator != null && !NoBias)
{
    for (int batchCount = 0; batchCount < input.BatchCount; batchCount++)
    {
        for (int och = 0; och < OutputCount; och++)
        {
            for (int oy = _trimY; oy < outputHeight + _trimY; oy++)
            {
                for (int ox = _trimX; ox < outputWidth + _trimX; ox++)
                {
                    int outputIndex = batchCount * OutputCount * outSizeOffset + och * outSizeOffset + (oy
- _trimY) * outputWidth + ox - _trimX;
                    result[outputIndex] += Bias.Data[och];
                }
            }
        }
    }
}
```

```
        result[outputIndex] = Activator.ForwardActivate(result[outputIndex]);
    }
}
}
}
}
else if (!NoBias)
{
    for (int batchCount = 0; batchCount < input.BatchCount; batchCount++)
    {
        for (int och = 0; och < OutputCount; och++)
        {
            for (int oy = _trimY; oy < outputHeight + _trimY; oy++)
            {
                for (int ox = _trimX; ox < outputWidth + _trimX; ox++)
                {
                    int outputIndex = batchCount * OutputCount * outSizeOffset + och * outSizeOffset + (oy - _trimY) * outputWidth + ox - _trimX;
                    result[outputIndex] += Bias.Data[och];
                }
            }
        }
    }
}
else if (Activator != null)
{
    for (int batchCount = 0; batchCount < input.BatchCount; batchCount++)
    {
        for (int och = 0; och < OutputCount; och++)
        {
            for (int oy = _trimY; oy < outputHeight + _trimY; oy++)
            {
                for (int ox = _trimX; ox < outputWidth + _trimX; ox++)
                {
                    int outputIndex = batchCount * OutputCount * outSizeOffset + och * outSizeOffset + (oy - _trimY) * outputWidth + ox - _trimX;
                    result[outputIndex] = Activator.ForwardActivate(result[outputIndex]);
                }
            }
        }
    }
}
```

```
        }
    }
}
}

return NdArray.Convert(result, new[] { OutputCount, outputHeight, outputWidth }, input.
BatchCount, this);
}

///////////////////////////////  

/// <summary> Need previous forward GPU. </summary>
///  

/// <param name="input"> The input. </param>
///  

/// <returns> A NdArray. </returns>
///  

/// <seealso cref="M:KelpNet.Common.Functions.CompressibleFunction.NeedPreviousForward
Gpu(NdArray)"/>  

///////////////////////////////

[NotNull]
protected override NdArray NeedPreviousForwardGpu([NotNull] NdArray input)
{
    int outputHeight = (input.Shape[1] - 1) * _subSampleY + _kHeight - _trimY * 2;
    int outputWidth = (input.Shape[2] - 1) * _subSampleX + _kWidth - _trimX * 2;
    Real[] result = new Real[input.BatchCount * OutputCount * outputWidth * outputHeight];

    using (ComputeBuffer<Real> gpuX = new ComputeBuffer<Real>(Weaver.Context,
ComputeMemoryFlags.ReadOnly | ComputeMemoryFlags.CopyHostPointer, input.Data))
    {
        using (ComputeBuffer<Real> gpuW = new ComputeBuffer<Real>(Weaver.Context,
ComputeMemoryFlags.ReadOnly | ComputeMemoryFlags.CopyHostPointer, Weight.Data))
        {
            using (ComputeBuffer<Real> gpub = new ComputeBuffer<Real>(Weaver.Context,
ComputeMemoryFlags.ReadOnly | ComputeMemoryFlags.CopyHostPointer, NoBias ? new
Real[OutputCount] : Bias.Data))
            {
                using (ComputeBuffer<Real> gpuY = new ComputeBuffer<Real>(Weaver.Context,
ComputeMemoryFlags.WriteOnly | ComputeMemoryFlags.AllocateHostPointer, result.Length))
                {

```

```
        ForwardKernel.SetMemoryArgument(0, gpuX);
        ForwardKernel.SetMemoryArgument(1, gpuW);
        ForwardKernel.SetMemoryArgument(2, gpub);
        ForwardKernel.SetMemoryArgument(3, gpuY);
        ForwardKernel.SetValueArgument(4, input.Shape[1]);
        ForwardKernel.SetValueArgument(5, input.Shape[2]);
        ForwardKernel.SetValueArgument(6, input.Length);
        ForwardKernel.SetValueArgument(7, outputWidth);
        ForwardKernel.SetValueArgument(8, outputHeight);
        ForwardKernel.SetValueArgument(9, _subSampleX);
        ForwardKernel.SetValueArgument(10, _subSampleY);
        ForwardKernel.SetValueArgument(11, _trimX);
        ForwardKernel.SetValueArgument(12, _trimY);
        ForwardKernel.SetValueArgument(13, _kHeight);
        ForwardKernel.SetValueArgument(14, _kWidth);
        ForwardKernel.SetValueArgument(15, OutputCount);
        ForwardKernel.SetValueArgument(16, InputCount);

        Weaver.CommandQueue.Execute(ForwardKernel, null, new long[] { input.BatchCount *
OutputCount, outputHeight, outputWidth }, null, null);
        Weaver.CommandQueue.Finish();
        Weaver.CommandQueue.ReadFromBuffer(gpuY, ref result, true, null);
    }
}
}

return NdArray.Convert(result, new[] { OutputCount, outputHeight, outputWidth }, input.
BatchCount, this);
}

///////////////////////////////
/// <summary> Gets an activatedgy. </summary>
///
/// <param name="y"> A NdArray to process. </param>
///
/// <returns> An array of real. </returns>
/////////////////////////////
```

```
[NotNull]
Real[] GetActivatedgy([NotNull] NdArray y)
{
    int gyIndex = 0;
    Real[] activatedgy = new Real[y.Grad.Length];
    for (int batchCounter = 0; batchCounter < y.BatchCount; batchCounter++)
    {
        for (int och = 0; och < y.Shape[0]; och++)
        {
            for (int olocation = 0; olocation < y.Shape[1] * y.Shape[2]; olocation++)
            {
                activatedgy[gyIndex] = Activator.BackwardActivate(y.Grad[gyIndex], y.Data[gyIndex]);
                gyIndex++;
            }
        }
    }
    return activatedgy;
}
```

```
////////////////////////////////////////////////////////////////////////
/// <summary> Calculates the bias graduated. </summary>
///
/// <param name="gy"> The gy. </param>
/// <param name="gyShape"> The gy shape. </param>
/// <param name="batchCount"> Number of batches. </param>
////////////////////////////////////////////////////////////////////////
```

```
void CalcBiasGrad([CanBeNull] Real[] gy, [CanBeNull] int[] gyShape, int batchCount)
{
    int gyIndex = 0;
    for (int batchCounter = 0; batchCounter < batchCount; batchCounter++)
    {
        for (int och = 0; och < gyShape[0]; och++)
        {
            for (int olocation = 0; olocation < gyShape[1] * gyShape[2]; olocation++)
            {
                Bias.Grad[och] += gy[gyIndex];
                gyIndex++;
            }
        }
    }
}
```

```
        }

    }

}

}

//////////////////////////////////////////////////////////////////
/// <summary> Need previous backward CPU. </summary>
///
/// <param name="y"> A NdArray to process. </param>
/// <param name="x"> A NdArray to process. </param>
///
/// <seealso cref="M:KelpNet.Common.Functions.CompressibleFunction.NeedPreviousBackwar-
dCpu(NdArray,NdArray)" />
//////////////////////////////////////////////////////////////////

protected override void NeedPreviousBackwardCpu([NotNull] NdArray y, [CanBeNull] NdArray x)
{
    //Real[] gx = new Real[x.Data.Length];
    Real[] activatedgy = Activator != null ? GetActivatedgy(y) : y.Grad;
    if (!NoBias) CalcBiasGrad(activatedgy, y.Shape, y.BatchCount);
    // Original logic
    for (int batchCount = 0; batchCount < y.BatchCount; batchCount++)
    {
        for (int och = 0; och < OutputCount; och++)
        {
            int outChOffset = och * Weight.Shape[1] * Weight.Shape[2] * Weight.Shape[3];
            int inputOffset = och * y.Shape[1] * y.Shape[2];
            for (int oy = _trimY; oy < y.Shape[1] + _trimY; oy++)
            {
                int iyLimit = oy / _subSampleY + 1 < x.Shape[1] ? oy / _subSampleY + 1 : x.Shape[1];
                int iyStart = oy - Weight.Shape[2] < 0 ? 0 : (oy - Weight.Shape[2]) / _subSampleY + 1;
                for (int ox = _trimX; ox < y.Shape[2] + _trimX; ox++)
                {
                    int ixLimit = ox / _subSampleX + 1 < x.Shape[2] ? ox / _subSampleX + 1 : x.Shape[2];
                    int ixStart = ox - Weight.Shape[3] < 0 ? 0 : (ox - Weight.Shape[3]) / _subSampleX + 1;
                    int gyIndex = batchCount * y.Length + inputOffset + (oy - _trimY) * y.Shape[2] + ox - _trimX;
                    Real gyData = activatedgy[gyIndex];
                    for (int ich = 0; ich < InputCount; ich++)
```

```

{
    int inChOffset = outChOffset + ich * Weight.Shape[2] * Weight.Shape[3];
    int pinputOffset = batchCount * x.Length + ich * x.Shape[1] * x.Shape[2];
    for (int iy = iyStart; iy < iyLimit; iy++)
    {
        for (int ix = ixStart; ix < ixLimit; ix++)
        {
            int pInIndex = pinputOffset + iy * x.Shape[2] + ix;
            int gwIndex = inChOffset + (oy - iy * _subSampleY) * Weight.Shape[3] + (ox - ix *
            _subSampleX);
            Weight.Grad[gwIndex] += x.Data[pInIndex] * gyData;
            x.Grad[pInIndex] += Weight.Data[gwIndex] * gyData;
        }
    }
}

///////////////////////////////
/// <summary> Need previous backward GPU. </summary>
///
/// <param name="y"> A NdArray to process. </param>
/// <param name="x"> A NdArray to process. </param>
///
/// <seealso cref="M:KelpNet.Common.Functions.CompressibleFunction.NeedPreviousBackwar
dGpu(NdArray,NdArray)" />
///////////////////////////////

protected override void NeedPreviousBackwardGpu([NotNull] NdArray y, [NotNull] NdArray x)
{
    Real[] gx = new Real[x.Data.Length];
    Real[] activatedgy = Activator != null ? GetActivatedgy(y) : y.Grad;
    if (!NoBias) CalcBiasGrad(activatedgy, y.Shape, y.BatchCount);
    // gy is used in common
    using (ComputeBuffer<Real> gpubY = new ComputeBuffer<Real>(Weaver.Context,

```

```
ComputeMemoryFlags.ReadOnly | ComputeMemoryFlags.CopyHostPointer, activatedgy))
{
    using (ComputeBuffer<Real> gpugW = new ComputeBuffer<Real>(Weaver.Context,
ComputeMemoryFlags.ReadWrite | ComputeMemoryFlags.CopyHostPointer, Weight.Grad))
    {
        using (ComputeBuffer<Real> gpuX = new ComputeBuffer<Real>(Weaver.Context,
ComputeMemoryFlags.ReadOnly | ComputeMemoryFlags.CopyHostPointer, x.Data))
        {
            BackwardgWKernel.SetMemoryArgument(0, gpugY);
            BackwardgWKernel.SetMemoryArgument(1, gpuX);
            BackwardgWKernel.SetMemoryArgument(2, gpugW);
            BackwardgWKernel.SetValueArgument(3, y.BatchCount);
            BackwardgWKernel.SetValueArgument(4, InputCount);
            BackwardgWKernel.SetValueArgument(5, y.Length);
            BackwardgWKernel.SetValueArgument(6, y.Shape[1]);
            BackwardgWKernel.SetValueArgument(7, y.Shape[2]);
            BackwardgWKernel.SetValueArgument(8, x.Shape[1]);
            BackwardgWKernel.SetValueArgument(9, x.Shape[2]);
            BackwardgWKernel.SetValueArgument(10, x.Length);
            BackwardgWKernel.SetValueArgument(11, _subSampleX);
            BackwardgWKernel.SetValueArgument(12, _subSampleY);
            BackwardgWKernel.SetValueArgument(13, _trimX);
            BackwardgWKernel.SetValueArgument(14, _trimY);
            BackwardgWKernel.SetValueArgument(15, _kHeight);
            BackwardgWKernel.SetValueArgument(16, _kWidth);

            Weaver.CommandQueue.Execute(BackwardgWKernel, null, new long[] { OutputCount *
InputCount, _kHeight, _kWidth },
                null, null);
            Weaver.CommandQueue.Finish();
            Weaver.CommandQueue.ReadFromBuffer(gpugW, ref Weight.Grad, true, null);
        }
    }
}

using (ComputeBuffer<Real> gpugX = new ComputeBuffer<Real>(Weaver.Context,
ComputeMemoryFlags.WriteOnly | ComputeMemoryFlags.AllocateHostPointer, gx.Length))
{
```

```
using (ComputeBuffer<Real> gpuW = new ComputeBuffer<Real>(Weaver.Context,
ComputeMemoryFlags.ReadOnly | ComputeMemoryFlags.CopyHostPointer, Weight.Data))
{
    BackwardgXKernel.SetMemoryArgument(0, gpugY);
    BackwardgXKernel.SetMemoryArgument(1, gpuW);
    BackwardgXKernel.SetMemoryArgument(2, gpugX);
    BackwardgXKernel.SetValueArgument(3, OutputCount);
    BackwardgXKernel.SetValueArgument(4, InputCount);
    BackwardgXKernel.SetValueArgument(5, y.Length);
    BackwardgXKernel.SetValueArgument(6, y.Shape[1]);
    BackwardgXKernel.SetValueArgument(7, y.Shape[2]);
    BackwardgXKernel.SetValueArgument(8, x.Shape[1]);
    BackwardgXKernel.SetValueArgument(9, x.Shape[2]);
    BackwardgXKernel.SetValueArgument(10, x.Length);
    BackwardgXKernel.SetValueArgument(11, _subSampleX);
    BackwardgXKernel.SetValueArgument(12, _subSampleY);
    BackwardgXKernel.SetValueArgument(13, _trimX);
    BackwardgXKernel.SetValueArgument(14, _trimY);
    BackwardgXKernel.SetValueArgument(15, _kHeight);
    BackwardgXKernel.SetValueArgument(16, _kWidth);

    Weaver.CommandQueue.Execute(BackwardgXKernel, null, new long[] { y.BatchCount *
x.Shape[0], x.Shape[1], x.Shape[2] },
    null, null);
    Weaver.CommandQueue.Finish();
    Weaver.CommandQueue.ReadFromBuffer(gpugX, ref gx, true, null);
}
}
}
for (int i = 0; i < x.Grad.Length; i++)
{
    x.Grad[i] += gx[i];
}
}
```

```
}
```

## EmbedID

EmbedID is used to learn embeddings of one-hot vector inputs. This is the code for the EmbedID object:

```
[Serializable]
public class EmbedID : SingleInputFunction
{
    /// <summary> Name of the function. </summary>
    const string FUNCTION_NAME = "EmbedID";

    /// <summary> The weight. </summary>
    public NdArray Weight;

    /// <summary> Number of inputs. </summary>
    public readonly int InputCount;
    /// <summary> Number of outputs. </summary>
    public readonly int OutputCount;

    /////////////////////////////////
    /// <summary>
    /// Initializes a new instance of the KelpNet.Functions.Connections.EmbedID class.
    /// </summary>
    ///
    /// <param name="inputCount"> Number of inputs. </param>
    /// <param name="outputCount"> Number of outputs. </param>
    /// <param name="initialW"> (Optional) The initial w. </param>
    /// <param name="name"> (Optional) The name. </param>
    /// <param name="inputNames"> (Optional) List of names of the inputs. </param>
    /// <param name="outputNames"> (Optional) List of names of the outputs. </param>
    ///////////////////////////////

    public EmbedID(int inputCount, int outputCount, [CanBeNull] Real[,] initialW = null, [CanBeNull]
        string name = FUNCTION_NAME, [CanBeNull] string[] inputNames = null, [CanBeNull] string[]
        outputNames = null) : base(name, inputNames, outputNames)
    {
        InputCount = inputCount;
        OutputCount = outputCount;
```

```
Weight = new NdArray(inputCount, outputCount);
Weight.Name = Name + " Weight";
if (initialW == null)
{
    Initializer.InitWeight(Weight);
}
else
{
    // Do not simply substitute for checking the size
    Weight.Data = Real.GetArray(initialW);
}
Parameters = new[] { Weight };
SingleInputForward = NeedPreviousForwardCpu;
SingleOutputBackward = NeedPreviousBackwardCpu;
}

///////////////////////////////  

/// <summary> Need previous forward CPU. </summary>
///  

/// <param name="x"> A NdArray to process. </param>
///  

/// <returns> A NdArray. </returns>
///////////////////////////////  
  

[NotNull]
protected NdArray NeedPreviousForwardCpu([NotNull] NdArray x)
{
    Real[] result = new Real[x.Data.Length * OutputCount];
    for (int b = 0; b < x.BatchCount; b++)
    {
        for (int i = 0; i < x.Length; i++)
        {
            for (int j = 0; j < OutputCount; j++)
            {
                result[i * OutputCount + j + b * x.Length * OutputCount] = Weight.Data[(int)x.Data[i + b * x.Length] * OutputCount + j];
            }
        }
    }
}
```

```
    }

    return NdArray.Convert(result, new[] { x.Length, OutputCount }, x.BatchCount, this);
}

//////////  
/// <summary> Need previous backward CPU. </summary>
///  
/// <param name="y"> A NdArray to process. </param>  
/// <param name="x"> A NdArray to process. </param>
//////////  
  
protected void NeedPreviousBackwardCpu([NotNull] NdArray y, [CanBeNull] NdArray x)
{
    for (int b = 0; b < y.BatchCount; b++)
    {
        for (int i = 0; i < x.Length; i++)
        {
            for (int j = 0; j < OutputCount; j++)
            {
                Weight.Grad[(int)x.Data[i + b * x.Length] * OutputCount + j] += y.Grad[i + j + b * y.Length];
            }
        }
    }
}
```

## Linear

This is the code for the Linear object:

```
[Serializable]
public class Linear : CompressibleFunction
{
    /// <summary> Name of the function. </summary>
    const string FUNCTION_NAME = "Linear";
    /// <summary> Name of the parameter. </summary>
    private const string PARAM_NAME = /*ForwardActivate*/;
    /// <summary> . </summary>
    private const string PARAM_VALUE = "gpuYSum = ForwardActivate(gpuYSum);";
```

```
/// <summary> The weight. </summary>
public NdArray Weight;
/// <summary> The bias. </summary>
public NdArray Bias;
/// <summary> True to no bias. </summary>
public readonly bool NoBias;
/// <summary> Number of inputs. </summary>
public readonly int InputCount;
/// <summary> Number of outputs. </summary>
public readonly int OutputCount;

///////////////////////////////
/// <summary>
/// Initializes a new instance of the KelpNet.Functions.Connections.Linear class.
/// </summary>
///
/// <param name="inputCount"> Number of inputs. </param>
/// <param name="outputCount"> Number of outputs. </param>
/// <param name="noBias"> (Optional) True to no bias. </param>
/// <param name="initialW"> (Optional) The initial w. </param>
/// <param name="initialb"> (Optional) The initialb. </param>
/// <param name="activation"> (Optional) The activation. </param>
/// <param name="name"> (Optional) The name. </param>
/// <param name="inputNames"> (Optional) List of names of the inputs. </param>
/// <param name="outputNames"> (Optional) List of names of the outputs. </param>
/// <param name="gpuEnable"> (Optional) True if GPU enable. </param>
///////////////////////////////

public Linear(int inputCount, int outputCount, bool noBias = false, [CanBeNull] Array initialW
= null, [CanBeNull] Array initialb = null, [CanBeNull] CompressibleActivation activation = null,
[CanBeNull] string name = FUNCTION_NAME, [CanBeNull] string[] inputNames = null, [CanBeNull]
string[] outputNames = null, bool gpuEnable = false) : base(FUNCTION_NAME, activation,
new[] { new KeyValuePair<string, string>(PARAM_NAME, PARAM_VALUE) }, name, inputNames,
outputNames, gpuEnable)
{
    OutputCount = outputCount;
    InputCount = inputCount;
    Weight = new NdArray(outputCount, inputCount) {Name = Name + " Weight"};
    NoBias = noBias;
```

```
Parameters = new NdArray[noBias ? 1 : 2];

if (initialW == null)
    Initializer.InitWeight(Weight);
else
    Weight.Data = Real.GetArray(initialW);
Parameters[0] = Weight;
if (!noBias)
{
    Bias = new NdArray(outputCount) {Name = Name + " Bias"};
    if (initialb != null)
        Bias.Data = Real.GetArray(initialb);
    Parameters[1] = Bias;
}
}

///////////
/// <summary> Gets biased value. </summary>
///
/// <param name="batchCount"> Number of batches. </param>
///
/// <returns> An array of real. </returns>
///////////

[NotNull]
Real[] GetBiasedValue(int batchCount)
{
    Real[] y = new Real[OutputCount * batchCount];
    for (int i = 0; i < batchCount; i++)
        Array.Copy(Bias.Data, 0, y, i * OutputCount, Bias.Data.Length);
    return y;
}

///////////
/// <summary> Need previous forward CPU. </summary>
///
/// <param name="x"> A NdArray to process. </param>
///
```



```
[NotNull]
protected override NdArray NeedPreviousForwardGpu([NotNull] NdArray x)
{
    Real[] y = NoBias ? new Real[OutputCount * x.BatchCount] : GetBiasedValue(x.BatchCount);

    using (ComputeBuffer<Real> gpuX = new ComputeBuffer<Real>(Weaver.Context,
        ComputeMemoryFlags.ReadOnly | ComputeMemoryFlags.CopyHostPointer, x.Data))
    {
        using (ComputeBuffer<Real> gpuW = new ComputeBuffer<Real>(Weaver.Context,
            ComputeMemoryFlags.ReadOnly | ComputeMemoryFlags.CopyHostPointer, Weight.Data))
        {
            using (ComputeBuffer<Real> gpuY = new ComputeBuffer<Real>(Weaver.Context,
                ComputeMemoryFlags.ReadWrite | ComputeMemoryFlags.CopyHostPointer, y))
            {
                ForwardKernel.SetMemoryArgument(0, gpuX);
                ForwardKernel.SetMemoryArgument(1, gpuW);
                ForwardKernel.SetMemoryArgument(2, gpuY);
                ForwardKernel.SetValueArgument(3, OutputCount);
                ForwardKernel.SetValueArgument(4, InputCount);

                Weaver.CommandQueue.Execute(ForwardKernel, null, new long[] { OutputCount,
                    x.BatchCount }, null, null);
                Weaver.CommandQueue.Finish();
                Weaver.CommandQueue.ReadFromBuffer(gpuY, ref y, true, null);
            }
        }
    }

    return NdArray.Convert(y, new[] { OutputCount }, x.BatchCount, this);
}

///////////////////////////////  

/// <summary> Gets an activatedgy. </summary>
///  

/// <param name="y"> A NdArray to process. </param>
///  

/// <returns> An array of real. </returns>
///////////////////////////////
```

[NotNull]

```
Real[] GetActivatedgy([NotNull] NdArray y)
{
    Real[] activateddgY = new Real[y.Grad.Length];
    for (int batchCount = 0; batchCount < y.BatchCount; batchCount++)
    {
        for (int i = 0; i < OutputCount; i++)
        {
            int index = batchCount * OutputCount + i;
            activateddgY[index] = Activator.BackwardActivate(y.Grad[index], y.Data[index]);
        }
    }
    return activateddgY;
}

///////////
/// <summary> Calculates the bias graduated. </summary>
///
/// <param name="gy"> The gy. </param>
/// <param name="batchCount"> Number of batches. </param>
///////////

void CalcBiasGrad([CanBeNull] Real[] gy, int batchCount)
{
    for (int batchCounter = 0; batchCounter < batchCount; batchCounter++)
    {
        for (int i = 0; i < OutputCount; i++)
            Bias.Grad[i] += gy[batchCounter * OutputCount + i];
    }
}

///////////
/// <summary> Need previous backward CPU. </summary>
///
/// <param name="y"> A NdArray to process. </param>
/// <param name="x"> A NdArray to process. </param>
///
/// <seealso cref="M:KelpNet.Common.Functions.CompressibleFunction.NeedPreviousBackwar
dCpu(NdArray,NdArray)"/>
```

```
||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
```

```
protected override void NeedPreviousBackwardCpu([NotNull] NdArray y, [CanBeNull] NdArray x)
{
    Real[] activatedgy = Activator != null ? GetActivatedgy(y) : y.Grad;
    if (!NoBias)
        CalcBiasGrad(activatedgy, y.BatchCount);
    for (int batchCount = 0; batchCount < y.BatchCount; batchCount++)
    {
        for (int i = 0; i < OutputCount; i++)
        {
            Real gyData = activatedgy[i + batchCount * OutputCount];
            for (int j = 0; j < InputCount; j++)
            {
                Weight.Grad[i * InputCount + j] += x.Data[j + batchCount * InputCount] * gyData;
                x.Grad[j + batchCount * InputCount] += Weight.Data[i * InputCount + j] * gyData;
            }
        }
    }
}
```

```
||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
```

```
/// <summary> Need previous backward GPU. </summary>
///
/// <param name="y"> A NdArray to process. </param>
/// <param name="x"> A NdArray to process. </param>
///
/// <seealso href="M:KelpNet.Common.Functions.CompressibleFunction.NeedPreviousBackwar
dGpu(NdArray,NdArray)">
||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
```

```
protected override void NeedPreviousBackwardGpu([NotNull] NdArray y, [NotNull] NdArray x)
{
    Real[] gx = new Real[x.Data.Length];
    Real[] activatedgy = Activator != null ? GetActivatedgy(y) : y.Grad;
    if (!NoBias)
        CalcBiasGrad(activatedgy, y.BatchCount);
```

```
using (ComputeBuffer<Real> gpugY = new ComputeBuffer<Real>(Weaver.Context,
ComputeMemoryFlags.ReadOnly | ComputeMemoryFlags.CopyHostPointer, activatedgy))
{
    using (ComputeBuffer<Real> gpugW = new ComputeBuffer<Real>(Weaver.Context,
ComputeMemoryFlags.ReadWrite | ComputeMemoryFlags.CopyHostPointer, Weight.Grad))
    {
        using (ComputeBuffer<Real> gpuX = new ComputeBuffer<Real>(Weaver.Context,
ComputeMemoryFlags.ReadOnly | ComputeMemoryFlags.CopyHostPointer, x.Data))
        {
            BackwardgWKernel.SetMemoryArgument(0, gpugY);
            BackwardgWKernel.SetMemoryArgument(1, gpuX);
            BackwardgWKernel.SetMemoryArgument(2, gpugW);
            BackwardgWKernel.SetValueArgument(3, y.BatchCount);
            BackwardgWKernel.SetValueArgument(4, OutputCount);
            BackwardgWKernel.SetValueArgument(5, InputCount);

            Weaver.CommandQueue.Execute(BackwardgWKernel, null, new long[] { InputCount,
OutputCount }, null, null);
            Weaver.CommandQueue.Finish();
            Weaver.CommandQueue.ReadFromBuffer(gpugW, ref Weight.Grad, true, null);
        }
    }
}

using (ComputeBuffer<Real> gpugX = new ComputeBuffer<Real>(Weaver.Context,
ComputeMemoryFlags.WriteOnly | ComputeMemoryFlags.AllocateHostPointer, gx.Length))
{
    using (ComputeBuffer<Real> gpuW = new ComputeBuffer<Real>(Weaver.Context,
ComputeMemoryFlags.ReadOnly | ComputeMemoryFlags.CopyHostPointer, Weight.Data))
    {
        BackwardgXKernel.SetMemoryArgument(0, gpugY);
        BackwardgXKernel.SetMemoryArgument(1, gpuW);
        BackwardgXKernel.SetMemoryArgument(2, gpugX);
        BackwardgXKernel.SetValueArgument(3, y.BatchCount);
        BackwardgXKernel.SetValueArgument(4, OutputCount);
        BackwardgXKernel.SetValueArgument(5, InputCount);

        Weaver.CommandQueue.Execute(BackwardgXKernel, null, new long[] { InputCount,
y.BatchCount }, null, null);
        Weaver.CommandQueue.Finish();
    }
}
```

```
        Weaver.CommandQueue.ReadFromBuffer(gpugX, ref gx, true, null);
    }
}
}
for (int i = 0; i < x.Grad.Length; i++)
    x.Grad[i] += gx[i];
}

///////////////////////////////  

/// <summary> Converts this object to a convolution 2D. </summary>
///  

/// <returns> A Convolution2D. </returns>
///////////////////////////////

[NotNull]
public Convolution2D AsConvolution2D()
{
    return new Convolution2D(this);
}
```

## LSTM

**Long Short-Term Memory** networks (**LSTM**) are a special kind of Recurrent Neural Network capable of learning long-term dependencies. They are explicitly designed to avoid any long-term dependency problems. Remembering information for long periods of time is their inherent behavior.

There are several variations of LSTMs such as peep-hole LSTMs, Gated Recurrent Units, coupled forget and input gate LSTMs, and so on.

This is the code for the LSTM object:

```
[Serializable]
public class LSTM : SingleInputFunction
{
    /// <summary> Name of the function. </summary>
    const string FUNCTION_NAME = "LSTM";

    /// <summary> The upward 0. </summary>
    public Linear upward0;
```

```
///<summary> The first upward. </summary>
public Linear upward1;
///<summary> The second upward. </summary>
public Linear upward2;
///<summary> The third upward. </summary>
public Linear upward3;

///<summary> The lateral 0. </summary>
public Linear lateral0;
///<summary> The first lateral. </summary>
public Linear lateral1;
///<summary> The second lateral. </summary>
public Linear lateral2;
///<summary> The third lateral. </summary>
public Linear lateral3;
///<summary> The parameter. </summary>
private List<Real[]> aParam;
///<summary> Zero-based index of the parameter. </summary>
private List<Real[]> iParam;
///<summary> The parameter. </summary>
private List<Real[]> fParam;
///<summary> The parameter. </summary>
private List<Real[]> oParam;
///<summary> The parameter. </summary>
private List<Real[]> cParam;

///<summary> The parameter. </summary>
private NdArray hParam;

///<summary> The gx previous 0. </summary>
NdArray gxPrev0;
///<summary> The first gx previous. </summary>
NdArray gxPrev1;
///<summary> The second gx previous. </summary>
NdArray gxPrev2;
///<summary> The third gx previous. </summary>
NdArray gxPrev3;
///<summary> The GC previous. </summary>
```

```
Real[] gcPrev;
/// <summary> Number of inputs. </summary>
public readonly int InputCount;
/// <summary> Number of outputs. </summary>
public readonly int OutputCount;

///////////////////////////////
/// <summary>
/// Initializes a new instance of the KelpNet.Functions.Connections.LSTM class.
/// </summary>
///
/// <param name="inSize"> Size of the in. </param>
/// <param name="outSize"> Size of the out. </param>
/// <param name="initialUpwardW"> (Optional) The initial upward w. </param>
/// <param name="initialUpwardb"> (Optional) The initial upwardb. </param>
/// <param name="initialLateralW"> (Optional) The initial lateral w. </param>
/// <param name="name"> (Optional) The name. </param>
/// <param name="inputNames"> (Optional) List of names of the inputs. </param>
/// <param name="outputNames"> (Optional) List of names of the outputs. </param>
/// <param name="gpuEnable"> (Optional) True if GPU enable. </param>
///////////////////////////////

public LSTM(int inSize, int outSize, [CanBeNull] Real[,] initialUpwardW = null, [CanBeNull]
Real[] initialUpwardb = null, [CanBeNull] Real[,] initialLateralW = null, [CanBeNull] string name =
FUNCTION_NAME, [CanBeNull] string[] inputNames = null, [CanBeNull] string[] outputNames =
null, bool gpuEnable = false) : base(name, inputNames, outputNames)
{
    InputCount = inSize;
    OutputCount = outSize;
    List<NdArray> functionParameters = new List<NdArray>();

    upward0 = new Linear(inSize, outSize, noBias: false, initialW: initialUpwardW, initialb:
initialUpwardb, name: "upward0", gpuEnable: gpuEnable);
    upward1 = new Linear(inSize, outSize, noBias: false, initialW: initialUpwardW, initialb:
initialUpwardb, name: "upward1", gpuEnable: gpuEnable);
    upward2 = new Linear(inSize, outSize, noBias: false, initialW: initialUpwardW, initialb:
initialUpwardb, name: "upward2", gpuEnable: gpuEnable);
    upward3 = new Linear(inSize, outSize, noBias: false, initialW: initialUpwardW, initialb:
initialUpwardb, name: "upward3", gpuEnable: gpuEnable);
```

```
functionParameters.AddRange(upward0.Parameters);
functionParameters.AddRange(upward1.Parameters);
functionParameters.AddRange(upward2.Parameters);
functionParameters.AddRange(upward3.Parameters);

// lateral does not have Bias
lateral0 = new Linear(outSize, outSize, noBias: true, initialW: initialLateralW, name: "lateral0",
gpuEnable: gpuEnable);
lateral1 = new Linear(outSize, outSize, noBias: true, initialW: initialLateralW, name: "lateral1",
gpuEnable: gpuEnable);
lateral2 = new Linear(outSize, outSize, noBias: true, initialW: initialLateralW, name: "lateral2",
gpuEnable: gpuEnable);
lateral3 = new Linear(outSize, outSize, noBias: true, initialW: initialLateralW, name: "lateral3",
gpuEnable: gpuEnable);

functionParameters.AddRange(lateral0.Parameters);
functionParameters.AddRange(lateral1.Parameters);
functionParameters.AddRange(lateral2.Parameters);
functionParameters.AddRange(lateral3.Parameters);

Parameters = functionParameters.ToArray();
SingleInputForward = ForwardCpu;
SingleOutputBackward = BackwardCpu;
}

///////////////////////////////
/// <summary> Forward CPU. </summary>
///
/// <param name="x"> A NdArray to process. </param>
///
/// <returns> A NdArray. </returns>
///////////////////////////////

[NotNull]
public NdArray ForwardCpu([NotNull] NdArray x)
{
    Real[][] upwards = new Real[4][];
    upwards[0] = upward0.Forward(x)[0].Data;
```

```
upwards[1] = upward1.Forward(x)[0].Data;
upwards[2] = upward2.Forward(x)[0].Data;
upwards[3] = upward3.Forward(x)[0].Data;

int outputDataSize = x.BatchCount * OutputCount;
if (hParam == null)
{
    // Initialize if there is no value
    aParam = new List<Real[]>();
    iParam = new List<Real[]>();
    fParam = new List<Real[]>();
    oParam = new List<Real[]>();
    cParam = new List<Real[]>();
}
else
{
    Real[] lateral0 = lateral0.Forward(hParam)[0].Data;
    Real[] lateral1 = lateral1.Forward(hParam)[0].Data;
    Real[] lateral2 = lateral2.Forward(hParam)[0].Data;
    Real[] lateral3 = lateral3.Forward(hParam)[0].Data;
    hParam.UseCount -= 4; // Correct number of times RFI

    for (int i = 0; i < outputDataSize; i++)
    {
        upwards[0][i] += lateral0[i];
        upwards[1][i] += lateral1[i];
        upwards[2][i] += lateral2[i];
        upwards[3][i] += lateral3[i];
    }
}

if (cParam.Count == 0)
{
    cParam.Add(new Real[outputDataSize]);
}

Real[] la = new Real[outputDataSize];
Real[] li = new Real[outputDataSize];
```

```
Real[] lf = new Real[outputDataSize];
Real[] lo = new Real[outputDataSize];
Real[] cPrev = cParam[cParam.Count - 1];
Real[] cResult = new Real[cPrev.Length];
Real[] lhParam = new Real[outputDataSize];

for (int b = 0; b < x.BatchCount; b++)
{
    //Reconfigure
    for (int j = 0; j < OutputCount; j++)
    {
        int index = j * 4;
        int batchIndex = b * OutputCount + j;
        la[batchIndex] = Math.Tanh(upwards[index / OutputCount][index % OutputCount + b * OutputCount]);
        li[batchIndex] = Sigmoid(upwards[++index / OutputCount][index % OutputCount + b * OutputCount]);
        lf[batchIndex] = Sigmoid(upwards[++index / OutputCount][index % OutputCount + b * OutputCount]);
        lo[batchIndex] = Sigmoid(upwards[++index / OutputCount][index % OutputCount + b * OutputCount]);
        cResult[batchIndex] = la[batchIndex] * li[batchIndex] + lf[batchIndex] * cPrev[batchIndex];
        lhParam[batchIndex] = lo[batchIndex] * Math.Tanh(cResult[batchIndex]);
    }
}

//Backward用
cParam.Add(cResult);
aParam.Add(la);
iParam.Add(li);
fParam.Add(lf);
oParam.Add(lo);
hParam = new NdArray(lhParam, new[] { OutputCount }, x.BatchCount, this);
return hParam;
}

///////////////////////////////
/// <summary> Backward CPU. </summary>
///
```



```
Real[] cPrev = cParam[cParam.Count - 1];
for (int i = 0; i < y.BatchCount; i++)
{
    Real[] gParam = new Real[InputCount * 4];
    for (int j = 0; j < InputCount; j++)
    {
        int prevOutputIndex = j + i * OutputCount;
        int prevInputIndex = j + i * InputCount;
        double co = Math.Tanh(lcParam[prevOutputIndex]);
        gcPrev[prevInputIndex] += y.Grad[prevOutputIndex] * loParam[prevOutputIndex] *
        GradTanh(co);
        gParam[j + InputCount * 0] = gcPrev[prevInputIndex] * liParam[prevOutputIndex] * GradTanh
        (laParam[prevOutputIndex]);
        gParam[j + InputCount * 1] = gcPrev[prevInputIndex] * laParam[prevOutputIndex] * GradSig
        moid(liParam[prevOutputIndex]);
        gParam[j + InputCount * 2] = gcPrev[prevInputIndex] * cPrev[prevOutputIndex] * GradSigmoi
        d(lfParam[prevOutputIndex]);
        gParam[j + InputCount * 3] = y.Grad[prevOutputIndex] * co * GradSigmoid(loParam[prevOut
        putIndex]);
        gcPrev[prevInputIndex] *= lfParam[prevOutputIndex];
    }
    Real[] resultParam = new Real[OutputCount * 4];
    // rearrangement
    for (int j = 0; j < OutputCount * 4; j++)
    {
        // Implicitly truncated
        int index = j / OutputCount;
        resultParam[j % OutputCount + index * OutputCount] = gParam[j / 4 + j % 4 * InputCount];
    }
    for (int j = 0; j < OutputCount; j++)
    {
        gxPrev0.Grad[i * OutputCount + j] = resultParam[0 * OutputCount + j];
        gxPrev1.Grad[i * OutputCount + j] = resultParam[1 * OutputCount + j];
        gxPrev2.Grad[i * OutputCount + j] = resultParam[2 * OutputCount + j];
        gxPrev3.Grad[i * OutputCount + j] = resultParam[3 * OutputCount + j];
    }
}
upward0.Backward(gxPrev0);
upward1.Backward(gxPrev1);
```

```
upward2.Backward(gxPrev2);
upward3.Backward(gxPrev3);
}

///////////
/// <summary> Initialize input data that could not be used up by RNN etc. </summary>
///
/// <seealso cref="M:KelpNet.Common.Functions.Function.ResetState()"/>
///////////

public override void ResetState()
{
    base.ResetState();
    gcPrev = null;
    hParam = null;
}

///////////
/// <summary> Sigmoids. </summary>
///
/// <param name="x"> The x coordinate. </param>
///
/// <returns> A double. </returns>
///////////

static double Sigmoid(double x)
{
    return 1 / (1 + Math.Exp(-x));
}

///////////
/// <summary> Graduated sigmoid. </summary>
///
/// <param name="x"> The x coordinate. </param>
///
/// <returns> A double. </returns>
/////////
```

```

static double GradSigmoid(double x)
{
    return x * (1 - x);
}

///////////////////////////////  

/// <summary> Graduated hyperbolic tangent. </summary>
///  

/// <param name="x"> The x coordinate. </param>
///  

/// <returns> A double. </returns>
///////////////////////////////

```

```

static double GradTanh(double x)
{
    return 1 - x * x;
}

```

## Normalization

The following normalization functions are available in Kelp.Net:

- BatchNormalization
- LRN

## BatchNormalization

BatchNormalization works by normalizing the inputs in a neural network in such a way that we no longer need to worry about the scale of the input features being drastically different. This, in turn, can reduce oscillations when we are approaching the minimum point. In theory, this should provide faster convergence. The use of BatchNormalization can also reduce the impact of the earlier layers by keeping their mean and variance fixed, again theoretically providing faster convergence.

This is the code for the BatchNormalization object:

```

[Serializable]
public class BatchNormalization : SingleInputFunction
{
    /// <summary> Name of the function. </summary>

```

```
const string FUNCTION_NAME = "BatchNormalization";
/// <summary> True if this object is train. </summary>
public bool IsTrain;
/// <summary> The gamma. </summary>
public NdArray Gamma;
/// <summary> The beta. </summary>
public NdArray Beta;
/// <summary> The average mean. </summary>
public NdArray AvgMean;
/// <summary> The average variable. </summary>
public NdArray AvgVar;
/// <summary> The decay. </summary>
private readonly Real Decay;
/// <summary> The EPS. </summary>
private readonly Real Eps;
/// <summary> The standard. </summary>
private Real[] Std;
/// <summary> The xhat. </summary>
private Real[] Xhat;
/// <summary> The mean. </summary>
private Real[] Mean;
/// <summary> The variance. </summary>
private Real[] Variance;
/// <summary> Size of the channel. </summary>
private readonly int ChannelSize;

///////////////////////////////
/// <summary>
/// Initializes a new instance of the KelpNet.Functions.Normalization.BatchNormalization
/// class.
/// </summary>
///
/// <param name="channelSize"> Size of the channel. </param>
/// <param name="decay"> (Optional) The decay. </param>
/// <param name="eps"> (Optional) The EPS. </param>
/// <param name="initialAvgMean"> (Optional) The initial average mean. </param>
/// <param name="initialAvgVar"> (Optional) The initial average variable. </param>
/// <param name="isTrain"> (Optional) True if this object is train. </param>
```



```
};

if (initialAvgMean != null)
    AvgMean.Data = Real.GetArray(initialAvgMean);

if (initialAvgVar != null)
    AvgVar.Data = Real.GetArray(initialAvgVar);

if (!IsTrain)
{
    Parameters[2] = AvgMean;
    Parameters[3] = AvgVar;
}
SingleInputForward = ForwardCpu;
SingleOutputBackward = BackwardCpu;
}

///////////////////////////////  

/// <summary> Forward CPU. </summary>  

///  

/// <param name="x"> A NdArray to process. </param>  

///  

/// <returns> A NdArray. </returns>  

///////////////////////////////  
  

[NotNull]
private NdArray ForwardCpu([NotNull] NdArray x)
{
    // Acquire parameters for calculation
    if (IsTrain)
    {
        // Set Mean and Variance of member
        Variance = new Real[ChannelSize];
        for (int i = 0; i < Variance.Length; i++)
            Variance[i] = 0;
        Mean = new Real[ChannelSize];
        for (int i = 0; i < Mean.Length; i++)
        {
```

```
for (int index = 0; index < x.BatchCount; index++)
    Mean[i] += x.Data[i + index * x.Length];
    Mean[i] /= x.BatchCount;
}
for (int i = 0; i < Mean.Length; i++)
{
    for (int index = 0; index < x.BatchCount; index++)
        Variance[i] += (x.Data[i + index * x.Length] - Mean[i]) * (x.Data[i + index * x.Length] -
Mean[i]);
    Variance[i] /= x.BatchCount;
}

for (int i = 0; i < Variance.Length; i++)
    Variance[i] += Eps;
}
else
{
    Mean = AvgMean.Data;
    Variance = AvgVar.Data;
}

Std = new Real[Variance.Length];
for (int i = 0; i < Variance.Length; i++)
    Std[i] = Math.Sqrt(Variance[i]);

// Calculate result
Xhat = new Real[x.Data.Length];
Real[] y = new Real[x.Data.Length];

int dataSize = 1;
for (int i = 1; i < x.Shape.Length; i++)
    dataSize *= x.Shape[i];

for (int batchCount = 0; batchCount < x.BatchCount; batchCount++)
{
    for (int i = 0; i < ChannelSize; i++)
    {
        for (int location = 0; location < dataSize; location++)
```

```

{
    int index = batchCount * ChannelSize * dataSize + i * dataSize + location;
    Xhat[index] = (x.Data[index] - Mean[i]) / Std[i];
    y[index] = Gamma.Data[i] * Xhat[index] + Beta.Data[i];
}
}

// Update parameters
if (IsTrain)
{
    int m = x.BatchCount;
    Real adjust = m / Math.Max(m - 1.0, 1.0); // unbiased estimation
    for (int i = 0; i < AvgMean.Data.Length; i++)
    {
        AvgMean.Data[i] *= Decay;
        Mean[i] *= 1 - Decay; // reuse buffer as a temporary
        AvgMean.Data[i] += Mean[i];
        AvgVar.Data[i] *= Decay;
        Variance[i] *= (1 - Decay) * adjust; // reuse buffer as a temporary
        AvgVar.Data[i] += Variance[i];
    }
}
return NdArray.Convert(y, x.Shape, x.BatchCount, this);
}

///////////////
/// <summary> Backward CPU. </summary>
///
/// <param name="y"> A NdArray to process. </param>
/// <param name="x"> A NdArray to process. </param>
///////////////

private void BackwardCpu([CanBeNull] NdArray y, [CanBeNull] NdArray x)
{
    Beta.ClearGrad();
    Gamma.ClearGrad();
}

```

```

for (int i = 0; i < ChannelSize; i++)
{
    for (int j = 0; j < y.BatchCount; j++)
    {
        Beta.Grad[i] += y.Grad[i + j * y.Length];
        Gamma.Grad[i] += y.Grad[i + j * y.Length] * Xhat[j * ChannelSize + i];
    }
}

if (IsTrain)
{
    // with learning
    int m = y.BatchCount;
    for (int i = 0; i < ChannelSize; i++)
    {
        Real gs = Gamma.Data[i] / Std[i];
        for (int j = 0; j < y.BatchCount; j++)
        {
            Real val = (Xhat[j * ChannelSize + i] * Gamma.Grad[i] + Beta.Grad[i]) / m;
            x.Grad[i + j * ChannelSize] += gs * (y.Grad[i + j * y.Length] - val);
        }
    }
}
else
{
    // No learning
    for (int i = 0; i < ChannelSize; i++)
    {
        Real gs = Gamma.Data[i] / Std[i];
        AvgMean.Grad[i] = -gs * Beta.Grad[i];
        AvgVar.Grad[i] = -0.5 * Gamma.Data[i] / AvgVar.Data[i] * Gamma.Grad[i];
        for (int j = 0; j < y.BatchCount; j++)
            x.Grad[i + j * ChannelSize] += gs * y.Grad[i + j * y.Length];
    }
}
}

```

## Local Response Normalization

**Local Response Normalization (LRN)** normalizes the input in a local region across or within feature maps.

This is the code for the LRN object:

```
[Serializable]  
public class LRN : SingleInputFunction  
{  
    /// <summary> Name of the function. </summary>  
    const string FUNCTION_NAME = "LRN";
```

```
///<summary> An int to process. </summary>
private int n;
///<summary> A Real to process. </summary>
private Real k;
///<summary> The alpha. </summary>
private Real alpha;
///<summary> The beta. </summary>
private Real beta;
///<summary> The unit scale. </summary>
private Real[] unitScale;
///<summary> The scale. </summary>
private Real[] scale;

///////////////////////////////
///<summary>
/// Initializes a new instance of the KelpNet.Functions.Normalization.LRN class.
///</summary>
///
///<param name="n"> (Optional) An int to process. </param>
///<param name="k"> (Optional) A double to process. </param>
///<param name="alpha"> (Optional) The alpha. </param>
///<param name="beta"> (Optional) The beta. </param>
///<param name="name"> (Optional) The name. </param>
///<param name="inputNames"> (Optional) List of names of the inputs. </param>
///<param name="outputNames"> (Optional) List of names of the outputs. </param>
///////////////////////////////

public LRN(int n = 5, double k = 2, double alpha = 1e-4, double beta = 0.75, [CanBeNull]
string name = FUNCTION_NAME, [CanBeNull] string[] inputNames = null, [CanBeNull] string[]
outputNames = null) : base(name, inputNames, outputNames)
{
    this.n = n;
    this.k = (Real)k;
    this.alpha = (Real)alpha;
    this.beta = (Real)beta;

    SingleInputForward = NeedPreviousForwardCpu;
    SingleOutputBackward = NeedPreviousBackwardCpu;
```

```
}

///////////
/// <summary> Need previous forward CPU. </summary>
///
/// <param name="input"> The input. </param>
///
/// <returns> A NdArray. </returns>
///////////

[NotNull]
private NdArray NeedPreviousForwardCpu([NotNull] NdArray input)
{
    int nHalf = n / 2;
    Real[] result = new Real[input.Data.Length];
    Real[] x2 = new Real[input.Data.Length];
    Real[] sumPart = new Real[input.Data.Length];
    unitScale = new Real[input.Data.Length];
    scale = new Real[input.Data.Length];

    for (int i = 0; i < x2.Length; i++)
        x2[i] = input.Data[i] * input.Data[i];
    Array.Copy(x2, sumPart, x2.Length);
    for (int b = 0; b < input.BatchCount; b++)
    {
        for (int ich = 0; ich < input.Shape[0]; ich++)
        {
            for (int location = 0; location < input.Shape[1] * input.Shape[2]; location++)
            {
                int baseIndex = b * input.Length + ich * input.Shape[1] * input.Shape[2] + location;

                for (int offsetCh = 1; offsetCh < nHalf; offsetCh++)
                {
                    if (ich - offsetCh > 0)
                    {
                        int offsetIndex = b * input.Length + (ich - offsetCh) * input.Shape[1] * input.Shape[2] +
location;
                        sumPart[baseIndex] += x2[offsetIndex];
                    }
                }
            }
        }
    }
}
```

```
        }

        if (ich + offsetCh < input.Shape[0])
        {
            int offsetIndex = b * input.Length + (ich + offsetCh) * input.Shape[1] * input.Shape[2] +
location;
            sumPart[baseIndex] += x2[offsetIndex];
        }
    }
}

// Take the average of places with n channels before and after
for (int i = 0; i < sumPart.Length; i++)
{
    unitScale[i] = k + alpha * sumPart[i];
    scale[i] = Math.Pow(unitScale[i], -beta);
    result[i] *= scale[i];
}
return NdArray.Convert(result, input.Shape, input.BatchCount, this);
}

///////////////////////////////
/// <summary> Need previous backward CPU. </summary>
///
/// <param name="y"> A NdArray to process. </param>
/// <param name="x"> A NdArray to process. </param>
///////////////////////////////

private void NeedPreviousBackwardCpu([NotNull] NdArray y, [NotNull] NdArray x)
{
    int nHalf = n / 2;
    Real[] summand = new Real[y.Grad.Length];
    Real[] sumPart = new Real[y.Grad.Length];

    for (int i = 0; i < y.Grad.Length; i++)
        summand[i] = y.Data[i] * y.Grad[i] / unitScale[i];
```

```
Array.Copy(summand, sumPart, summand.Length);

for (int b = 0; b < y.BatchCount; b++)
{
    for (int ich = 0; ich < y.Shape[0]; ich++)
    {
        for (int location = 0; location < y.Shape[1] * y.Shape[2]; location++)
        {
            int baseIndex = b * y.Length + ich * y.Shape[1] * y.Shape[2] + location;
            for (int offsetCh = 1; offsetCh < nHalf; offsetCh++)
            {
                if (ich - offsetCh > 0)
                {
                    int offsetIndex = b * y.Length + (ich - offsetCh) * y.Shape[1] * y.Shape[2] + location;
                    sumPart[baseIndex] += summand[offsetIndex];
                }
                if (ich + offsetCh < y.Shape[0])
                {
                    int offsetIndex = b * y.Length + (ich + offsetCh) * y.Shape[1] * y.Shape[2] + location;
                    sumPart[baseIndex] += summand[offsetIndex];
                }
            }
        }
    }
}

for (int i = 0; i < x.Grad.Length; i++)
    x.Grad[i] += y.Grad[i] * scale[i] - 2 * alpha * beta * y.Data[i] * sumPart[i];
}
```

## Noise

The following Noise functions are available in Kelp.Net:

- Dropout
- StochasticDepth

## Dropout

This is the code for the Dropout object:

```
[Serializable]
public class Dropout : SingleInputFunction, IParallelizable
{
    /// <summary> Name of the function. </summary>
    const string FUNCTION_NAME = "Dropout";

    /// <summary> The dropout ratio. </summary>
    private readonly Real dropoutRatio;
    /// <summary> Stack of masks. </summary>
    private readonly List<Real[]> maskStack = new List<Real[]>();

    /// <summary> The forward kernel. </summary>
    [NonSerialized]
    [DebuggerBrowsable(DebuggerBrowsableState.Never)]
    public ComputeKernel ForwardKernel;

    /// <summary> The backward kernel. </summary>
    [NonSerialized]
    [DebuggerBrowsable(DebuggerBrowsableState.Never)]
    public ComputeKernel BackwardKernel;

    /////////////////////////////////
    /// <summary>
    /// Initializes a new instance of the KelpNet.Functions.Noise.Dropout class.
    /// </summary>
    ///
    /// <param name="dropoutRatio"> (Optional) The dropout ratio. </param>
    /// <param name="name"> (Optional) The name. </param>
    /// <param name="inputNames"> (Optional) List of names of the inputs. </param>
    /// <param name="outputNames"> (Optional) List of names of the outputs. </param>
    /// <param name="gpuEnable"> (Optional) True if GPU enable. </param>
    ///////////////////////////////
    public Dropout(double dropoutRatio = 0.5, [CanBeNull] string name = FUNCTION_NAME,
        [CanBeNull] string[] inputNames = null, [CanBeNull] string[] outputNames = null, bool gpuEnable
```

```
= false) : base(name, inputNames, outputNames)
{
    this.dropoutRatio = dropoutRatio;
    SetGpuEnable(gpuEnable);
}

///////////////////////////////  

/// <summary> Sets GPU enable. </summary>  

///  

/// <param name="enable"> True to enable, false to disable. </param>  

///  

/// <returns> True if it succeeds, false if it fails. </returns>  

///  

/// <seealso cref="M:KelpNet.Common.Functions.IParallelizable.SetGpuEnable(bool)" />  

///////////////////////////////

public bool SetGpuEnable(bool enable)
{
    GpuEnable = enable & Weaver.Enable;
    CreateKernel();
    if (GpuEnable)
    {
        SingleInputForward = ForwardGpu;
        SingleOutputBackward = BackwardGpu;
    }
    else
    {
        SingleInputForward = ForwardCpu;
        SingleOutputBackward = BackwardCpu;
    }
    return GpuEnable;
}

/////////////////////////////  

/// <summary> Creates the kernel. </summary>  

///  

/// <seealso cref="M:KelpNet.Common.Functions.IParallelizable.CreateKernel()" />  

/////////////////////////////
```

```
public void CreateKernel()
{
    if (GpuEnable)
    {
        string kernelSource = Weaver.GetKernelSource(FUNCTION_NAME);
        ComputeProgram program = Weaver.CreateProgram(kernelSource);
        ForwardKernel = program.CreateKernel("DropoutForward");
        BackwardKernel = program.CreateKernel("DropoutBackward");
    }
}

///////////////////////////////
/// <summary> Makes a mask. </summary>
///
/// <param name="xLength"> The length. </param>
///
/// <returns> A Real[]. </returns>
///////////////////////////////

[NotNull]
private Real[] MakeMask(int xLength)
{
    Real[] mask = new Real[xLength];
    Real scale = 1 / (1 - dropoutRatio);
    for (int i = 0; i < mask.Length; i++)
        mask[i] = Mother.Dice.NextDouble() >= dropoutRatio ? scale : 0;
    maskStack.Add(mask);
    return mask;
}

///////////////////////////////
/// <summary> Forward CPU. </summary>
///
/// <param name="x"> A NdArray to process. </param>
///
/// <returns> A NdArray. </returns>
/////////////////////////////
```

```
[NotNull]
public NdArray ForwardCpu([NotNull] NdArray x)
{
    Real[] result = new Real[x.Data.Length];
    Real[] mask = MakeMask(x.Length);
    for (int i = 0; i < x.Data.Length; i++)
        result[i] = x.Data[i] * mask[i % mask.Length];
    return NdArray.Convert(result, x.Shape, x.BatchCount, this);
}

///////////////////////////////  

/// <summary> Forward GPU. </summary>
///  

/// <param name="x"> A NdArray to process. </param>
///  

/// <returns> A NdArray. </returns>
///////////////////////////////

[NotNull]
public NdArray ForwardGpu([NotNull] NdArray x)
{
    Real[] result = new Real[x.Data.Length];
    Real[] mask = MakeMask(x.Length);
    using (ComputeBuffer<Real> gpuX = new ComputeBuffer<Real>(Weaver.Context,
        ComputeMemoryFlags.ReadOnly | ComputeMemoryFlags.CopyHostPointer, x.Data))
    {
        using (ComputeBuffer<Real> gpuMask = new ComputeBuffer<Real>(Weaver.Context,
            ComputeMemoryFlags.ReadOnly | ComputeMemoryFlags.CopyHostPointer, mask))
        {
            using (ComputeBuffer<Real> gpuY = new ComputeBuffer<Real>(Weaver.Context,
                ComputeMemoryFlags.WriteOnly | ComputeMemoryFlags.AllocateHostPointer, result.Length))
            {
                ForwardKernel.SetMemoryArgument(0, gpuX);
                ForwardKernel.SetMemoryArgument(1, gpuMask);
                ForwardKernel.SetMemoryArgument(2, gpuY);
                ForwardKernel.SetValueArgument(3, mask.Length);
            }
        }
    }
}
```

```
Weaver.CommandQueue.Execute(ForwardKernel, null, new long[] { x.Data.Length }, null,
null);
    Weaver.CommandQueue.Finish();
    Weaver.CommandQueue.ReadFromBuffer(gpuY, ref result, true, null);
}
}
}

return NdArray.Convert(result, x.Shape, x.BatchCount, this);
}

///////////////////////////////  

/// <summary> Backward CPU. </summary>
///  

/// <param name="y"> A NdArray to process. </param>
/// <param name="x"> A NdArray to process. </param>
///////////////////////////////

public void BackwardCpu([NotNull] NdArray y, [NotNull] NdArray x)
{
    Real[] result = y.Grad.ToArray();
    Real[] mask = maskStack[maskStack.Count - 1];
    maskStack.RemoveAt(maskStack.Count - 1);
    for (int b = 0; b < y.BatchCount; b++)
    {
        for (int i = 0; i < mask.Length; i++)
            result[b * y.Length + i] *= mask[i];
    }
    for (int i = 0; i < x.Grad.Length; i++)
        x.Grad[i] += result[i];
}

///////////////////////////////
/// <summary> Backward GPU. </summary>
///  

/// <param name="y"> A NdArray to process. </param>
/// <param name="x"> A NdArray to process. </param>
/////////////////////////////
```

```
public void BackwardGpu([NotNull] NdArray y, [NotNull] NdArray x)
{
    Real[] result = y.Grad.ToArray();
    Real[] mask = maskStack[maskStack.Count - 1];
    maskStack.RemoveAt(maskStack.Count - 1);

    using (ComputeBuffer<Real> gpuMask = new ComputeBuffer<Real>(Weaver.Context,
        ComputeMemoryFlags.ReadOnly | ComputeMemoryFlags.CopyHostPointer, mask))
    {
        using (ComputeBuffer<Real> gpugX = new ComputeBuffer<Real>(Weaver.Context,
            ComputeMemoryFlags.ReadWrite | ComputeMemoryFlags.CopyHostPointer, result))
        {
            BackwardKernel.SetMemoryArgument(0, gpuMask);
            BackwardKernel.SetMemoryArgument(1, gpugX);
            BackwardKernel.SetValueArgument(2, y.Length);

            Weaver.CommandQueue.Execute(BackwardKernel, null, new long[] { mask.Length,
                y.BatchCount }, null, null);
            Weaver.CommandQueue.Finish();
            Weaver.CommandQueue.ReadFromBuffer(gpugX, ref result, true, null);
        }
    }

    for (int i = 0; i < x.Grad.Length; i++)
        x.Grad[i] += result[i];
}

///////////////////////////////  

/// <summary> I do not do anything when Predict. </summary>  

///  

/// <param name="input"> The input. </param>  

///  

/// <returns> A NdArray. </returns>  

///  

/// <seealso cref="M:KelpNet.Common.Functions.Type.SingleInputFunction.Predict(NdArray)" />
///////////////////////////////  
  

public override NdArray Predict(NdArray input)
```

```
{
    return input;
}
}
```

## StochasticDepth

StochasticDepth is a training procedure that enables a setup to train short networks and obtain deep networks in return (yes, this is not a typo!). We start with very deep networks but during training, for each mini-batch we process, we randomly drop a subset of layers and bypass them using the identify function. The resulting networks are short during training and deep during testing. It reduces the training time substantially and improves test errors on almost all datasets significantly.

This is the code for the StochasticDepth object:

```
[Serializable]
public class StochasticDepth : Function // Use to replace SplitFunction
{
    /// <summary> Name of the function. </summary>
    const string FUNCTION_NAME = "StochasticDepth";
    /// <summary> The pl. </summary>
    private readonly Real _pl;
    /// <summary> List of skips. </summary>
    private readonly List<bool> _skipList = new List<bool>();
    /// <summary> Skipped by probability. </summary>
    private readonly Function _function;
    /// <summary> Always executed. </summary>
    private readonly Function _resBlock;

    /////////////////////////////////
    /// <summary> Query if this object is skip. </summary>
    ///
    /// <returns> True if skip, false if not. </returns>
    /////////////////////////////////

    private bool IsSkip()
    {
        bool result = Mother.Dice.NextDouble() >= _pl;
        _skipList.Add(result);
    }
}
```

```
    return result;
}

///////////
/// <summary>
/// Initializes a new instance of the KelpNet.Functions.Noise.StochasticDepth class.
/// </summary>
///
/// <param name="function"> The function. </param>
/// <param name="resBlock"> (Optional) The resource block. </param>
/// <param name="pl"> (Optional) The pl. </param>
/// <param name="name"> (Optional) The name. </param>
/// <param name="inputNames"> (Optional) List of names of the inputs. </param>
/// <param name="outputNames"> (Optional) List of names of the outputs. </param>
///////////

public StochasticDepth([CanBeNull] Function function, [CanBeNull] Function resBlock = null,
double pl = 0.5, [CanBeNull] string name = FUNCTION_NAME, [CanBeNull] string[] inputNames =
null, [CanBeNull] string[] outputNames = null) : base(name, inputNames, outputNames)
{
    _function = function;
    _resBlock = resBlock;
    _pl = pl;
}

/////////
/// <summary> Forwards the given xs. </summary>
///
/// <param name="xs"> A variable-length parameters list containing xs. </param>
///
/// <returns> A NdArray[]. </returns>
///
/// <seealso cref="M:KelpNet.Common.Functions.Function.Forward(params NdArray[])"/>
/////////

[NotNull]
public override NdArray[] Forward([CanBeNull] params NdArray[] xs)
{
```

```
List<NdArray> resultArray = new List<NdArray>();
NdArray[] resResult = xs;
if (_resBlock != null)
    resResult = _resBlock.Forward(xs);

resultArray.AddRange(resResult);
if (!IsSkip())
{
    Real scale = 1 / (1 - _pl);
    NdArray[] result = _function.Forward(xs);
    foreach (var t in result)
    {
        for (int j = 0; j < t.Data.Length; j++)
            t.Data[j] *= scale;
    }
    resultArray.AddRange(result);
}
else
{
    NdArray[] result = new NdArray[resResult.Length];
    for (int i = 0; i < result.Length; i++)
        result[i] = new NdArray(resResult[i].Shape, resResult[i].BatchCount, resResult[i].ParentFunc);
    resultArray.AddRange(result);
}
return resultArray.ToArray();
}

///////////////////////////////
/// <summary> Backwards the given ys. </summary>
///
/// <param name="ys"> A variable-length parameters list containing ys. </param>
///
/// <seealso href="M:KelpNet.Common.Functions.Function.Backward(params NdArray[])">
///////////////////////////////

public override void Backward(params NdArray[] ys)
{
    _resBlock?.Backward(ys);
```

```
bool isSkip = _skipList[_skipList.Count - 1];
_skipList.RemoveAt(_skipList.Count - 1);
if (!isSkip)
{
    NdArray[] copyys = new NdArray[ys.Length];
    Real scale = 1 / (1 - _pl)
    for (int i = 0; i < ys.Length; i++)
    {
        copyys[i] = ys[i].Clone();
        for (int j = 0; j < ys[i].Data.Length; j++)
            copyys[i].Data[j] *= scale;
    }
    _function.Backward(copyys);
}
}

///////////////////////////////
/// <summary> Evaluation function. </summary>
///
/// <param name="xs"> A variable-length parameters list containing xs. </param>
///
/// <returns> A NdArray[]. </returns>
///
/// <seealso href="M:KelpNet.Common.Functions.Function.Predict(params NdArray[])">
///////////////////////////////

public override NdArray[] Predict(params NdArray[] xs)
{
    return _function.Predict(xs);
}
```

## Loss

The following loss functions are available in Kelp.Net:

- MeanSquaredError
- SoftmaxCrossEntropy

## MeanSquaredError

This is the code for the MeanSquaredError object:

```
public class MeanSquaredError : LossFunction
{
    /////////////////////////////////
    /// <summary> Evaluates. </summary>
    ///
    /// <exception cref="Exception"> Thrown when an exception error condition occurs. </exception>
    ///
    /// <param name="input"> The input. </param>
    /// <param name="teachSignal"> The teach signal. </param>
    ///
    /// <returns> A Real. </returns>
    ///
    /// <seealso href="M:KelpNet.Common.Loss.LossFunction.Evaluate(NdArray[],NdArray[])">
    /////////////////////////////////

    public override Real Evaluate([NotNull] NdArray[] input, [NotNull] NdArray[] teachSignal)
    {
        Real resultLoss = 0;
        #if DEBUG
            if (input.Length != teachSignal.Length) throw new Exception("Input and teacher signal size are different");
        #endif
        for (int k = 0; k < input.Length; k++)
        {
            Real sumLoss = 0;
            Real[] resultArray = new Real[input[k].Data.Length];
            for (int b = 0; b < input[k].BatchCount; b++)
            {
                Real localloss = 0;
                Real coeff = 2.0 / teachSignal[k].Length;
                int batchoffset = b * teachSignal[k].Length;
                for (int i = 0; i < input[k].Length; i++)
                {
                    Real result = input[k].Data[b * input[k].Length + i] - teachSignal[k].Data[batchoffset + i];
                    localloss += result * result;
                }
                sumLoss += localloss / coeff;
            }
            resultArray[k] = sumLoss;
        }
        return resultArray;
    }
}
```

```
    localLoss += result * result;
    resultArray[batchOffset + i] *= coeff;
}
sumLoss += localLoss / teachSignal[k].Length;
}
resultLoss += sumLoss / input[k].BatchCount;
input[k].Grad = resultArray;
}
resultLoss /= input.Length;
return resultLoss;
}
}
```

## SoftmaxCrossEntropy

This is the code for the SoftmaxCrossEntropy object:

```
public class SoftmaxCrossEntropy : LossFunction
{
    /////////////////////////////////
    /// <summary> Evaluates. </summary>
    ///
    /// <exception cref="Exception"> Thrown when an exception error condition occurs. </exception>
    ///
    /// <param name="input"> The input. </param>
    /// <param name="teachSignal"> A variable-length parameters list containing teach signal. </param>
    ///
    /// <returns> A Real. </returns>
    ///
    /// <seealso cref="M:KelpNet.Common.Loss.LossFunction.Evaluate(NdArray[],params NdArray[])" />
    /////////////////////////////////

    public override Real Evaluate([NotNull] NdArray[] input, [NotNull] params NdArray[] teachSignal)
    {
        Real resultLoss = 0;
        #if DEBUG
```

```
if (input.Length != teachSignal.Length) throw new Exception("Input and teacher signal size are different");
#endifif
for (int k = 0; k < input.Length; k++)
{
    Real localloss = 0;
    Real[] gx = new Real[input[k].Data.Length];
    for (int b = 0; b < input[k].BatchCount; b++)
    {
        Real maxIndex = 0;
        for (int i = 0; i < teachSignal[k].Length; i++)
        {
            if (maxIndex < teachSignal[k].Data[i + b * teachSignal[k].Length])
                maxIndex = teachSignal[k].Data[i + b * teachSignal[k].Length];
        }
        Real[] logY = new Real[input[k].Length];
        Real y = 0;
        Real m = input[k].Data[b * input[k].Length];
        for (int i = 1; i < input[k].Length; i++)
        {
            if (m < input[k].Data[i + b * input[k].Length])
                m = input[k].Data[i + b * input[k].Length];
        }
        for (int i = 0; i < input[k].Length; i++)
            y += Math.Exp(input[k].Data[i + b * input[k].Length] - m);
        m += Math.Log(y);
        for (int i = 0; i < input[k].Length; i++)
            logY[i] = input[k].Data[i + b * input[k].Length] - m;
        localloss += -logY[(int)maxIndex];
        for (int i = 0; i < logY.Length; i++)
            gx[i + b * input[k].Length] = Math.Exp(logY[i]);
        gx[(int)maxIndex + b * input[k].Length] -= 1;
    }
    resultLoss += localloss / input[k].BatchCount;
    input[k].Grad = gx;
}
resultLoss /= input.Length;
```

```

    return resultLoss;
}
}

```

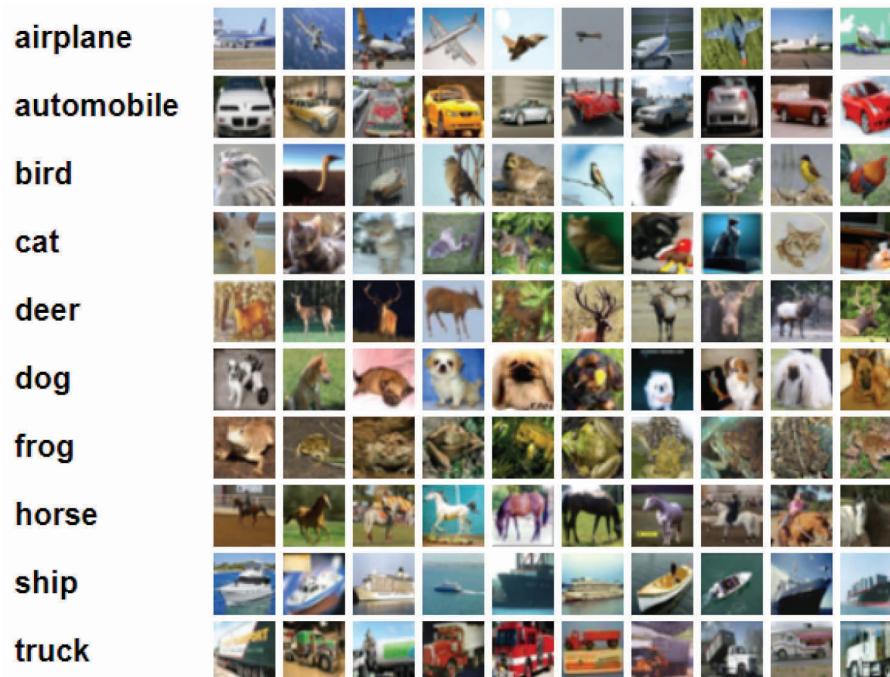
## Datasets

Throughout the example tests, we will create, try and stay consistent in the data that we use. To do this, we need to utilize the following machine learning datasets. When a dataset may not be supported in any test, then a declaration will be made under the main heading of that dataset.

### CIFAR-10

The CIFAR-10 dataset consists of 60,000 32x32 colour images in 10 classes, with 6,000 images per class. There are 50,000 training images and 10,000 test images. The dataset is divided into five training batches and one test batch, each with 10,000 images. The test batch contains exactly 1,000 randomly-selected images from each class. The training batches contain the remaining images in random order, but some training batches may contain more images from one class than another. Between them, the training batches contain exactly 5,000 images from each class.

Here are examples of the images contained in this dataset. The classes are completely mutually exclusive:



## CIFAR-100

The CIFAR-100 dataset is similar to the CIFAR-10 dataset, except that it has 100 classes containing 600 images each. There are 500 training images and 100 testing images per class. The 100 classes in the CIFAR-100 are grouped into 20 superclasses. Each image comes with a fine label (the class to which it belongs) and a coarse label (the superclass to which it belongs). Here is the list of classes in the CIFAR-100 dataset:

Superclass	Classes
Aquatic mammals	Beaver, dolphin, otter, seal, and whale
Fish	Aquarium fish, flatfish, ray, shark, and trout
Flowers	Orchids, poppies, roses, sunflowers, and tulips
Food containers	Bottles, bowls, cans, cups, and plates
Fruit and vegetables	Apples, mushrooms, oranges, pears, and sweet peppers
Household electrical devices	Clock, computer keyboard, lamp, telephone, and television
Household furniture	Bed, chair, couch, table, and wardrobe
Insects	Bee, beetle, butterfly, caterpillar, and cockroach
Large carnivores	Bear, leopard, lion, tiger, and wolf
Large man-made outdoor things	Bridge, castle, house, road, and skyscraper
Large natural outdoor scenes	Cloud, forest, mountain, plain, and sea
Large omnivores and herbivores	Camel, cattle, chimpanzee, elephant, and kangaroo
Medium-sized mammals	Fox, porcupine, possum, raccoon, and skunk
Non-insect invertebrates	Crab, lobster, snail, spider, and worm
People	Baby, boy, girl, man, and woman
Reptiles	Crocodile, dinosaur, lizard, snake, and turtle
Small mammals	Hamster, mouse, rabbit, shrew, and squirrel
Trees	Maple, oak, palm, pine, and willow
Vehicles 1	Bicycle, bus, motorcycle, pickup truck, and train
Vehicles 2	Lawn-mower, rocket, streetcar, tank, and tractor

## MNIST

The MNIST database is a large database of handwritten digits that is commonly used for training various image processing systems. The database is also widely used for training and testing in the field of machine learning. It has a training set of 60,000 examples and a test set of 10,000 examples. The digits have been size-normalized and centered in a fixed-size image, making it the standard of choice for people wanting to try various learning techniques without requiring the effort on preprocessing and formatting:



## Street View House Numbers (SVHN)

The SVHN dataset is a real-world image dataset similar to MNIST (the images are of small cropped digits), but incorporates an order of magnitude more labeled data (over 600,000 digit images) and comes from a significantly harder, unsolved, real-world problem (recognizing digits and numbers in natural scene images). SVHN is obtained from house numbers in Google Street View images.

At the time of writing this book, the full SVHN dataset was not yet supported in Kelp.Net. Various preprocessing techniques are suggested to work with the larger and more complicated dataset, and I am hard at work to support it. Keep checking the GitHub repository for the latest updates to Kelp.Net:



## Summary

In this chapter, we discussed each of the terms in the world of Kelp.Net. We also discussed the functional components of Kelp.Net and the datasets we will use in future examples.

Next, time to play!

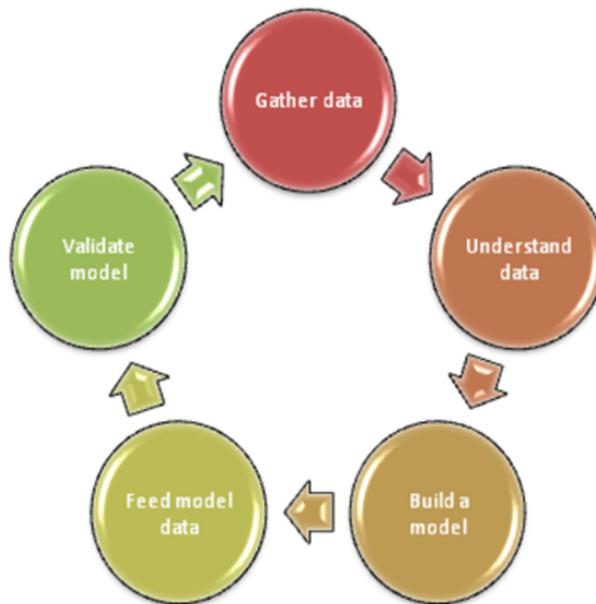
## References

1. Tokui, Seiya; et al. (2015). *Chainer: a next-generation open source framework for deep learning*. 29th Annual Conference on Neural Information Processing Systems (NIPS).
2. Shimada, Naoki (September 14, 2017). *Deep Learning with Chainer*. Gijutsu-Hyohron. p.61. ISBN 4774191868.
3. *Eager Execution: An imperative, Define-by-Run interface to TensorFlow* Google Research Blog.
4. *Deep Learning With Dynamic Computation Graphs* (ICLR 2017) Metadata.
5. Hido, Shohei (8 November 2016) *Complex neural networks made easy by Chainer* O'Reilly Media. Retrieved 26 June 2018.
6. Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, Andrew Y. Ng *Reading Digits in Natural Images with Unsupervised Feature Learning* NIPS Workshop on Deep Learning and Unsupervised Feature Learning 2011.

# CHAPTER 5

# Model Testing and Training

As you may know, a vast amount of your time will be spent testing and training your algorithm. Besides the loading and saving of models, this will probably be your most used functionality. It is a cycle that repeats over and over again.



Fortunately, Kelp.Net has made this a very easy process and provides many functions to help you in this endeavor. Kelp.Net has a `Trainer` class which encapsulates all the code required for training your model. The basic function is the `Train` method, which

is listed as follows:

```
public static Real Train([NotNull] FunctionStack functionStack, [CanBeNull] NdArray input,
[CanBeNull] NdArray teach, [NotNull] LossFunction lossFunction, bool isUpdate = true, bool
verbose = true)
{
    if (verbose)
    {
        RILogManager.Default?.EnterMethod("Training " + functionStack.Name);
        RILogManager.Default?.SendDebug("Forward propagation");
    }
    result = functionStack.Forward(verbose, input);
    if (verbose)
        RILogManager.Default?.SendDebug("Evaluating loss");
    Real sumLoss = lossFunction.Evaluate(verbose, result, teach);
    // Run Backward batch
    if (verbose)
        RILogManager.Default?.SendDebug("Backward propagation");
    functionStack.Backward(verbose, result);
    if (isUpdate)
    {
        if (verbose)
            RILogManager.Default?.SendDebug("Updating stack");
        functionStack.Update();
    }
    if (verbose)
    {
        RILogManager.Default?.ExitMethod("Training " + functionStack.Name);
        RILogManager.Default?.ViewerSendWatch("Local Loss", sumLoss.ToString(), sumLoss);
    }
    return sumLoss;
}
```

Besides the basic Train method, there are several more specific methods which can be of use and save your time in coding. These functions basically save the time of specifying the LossFunction to use for training. Based on the name of the function, the LossFunction is created internally for your convenience.

The additional training methods are as follows:

- TrainWithSoftmaxCrossEntropy

- TrainWithMeanSquaredError
- TrainWithRootMeanSquaredError

Once FunctionStack or SortedFunctionStack have been created and your optimizer(s) is/are set, you can call the Train method and perform your training.

Here is a simple example to see how we can use the TrainWithSoftmaxCrossEntropy method. In this example, we will create a small function stack consisting of linear and sigmoid layers as well as a stochastic gradient descent (momentum) optimizer. Once we create them, we will iterate over the data and train the algorithm. The Train algorithm takes a LossFunction to work with, and you can choose between the MeanSquaredError or SoftmaxCrossEntropy functions.

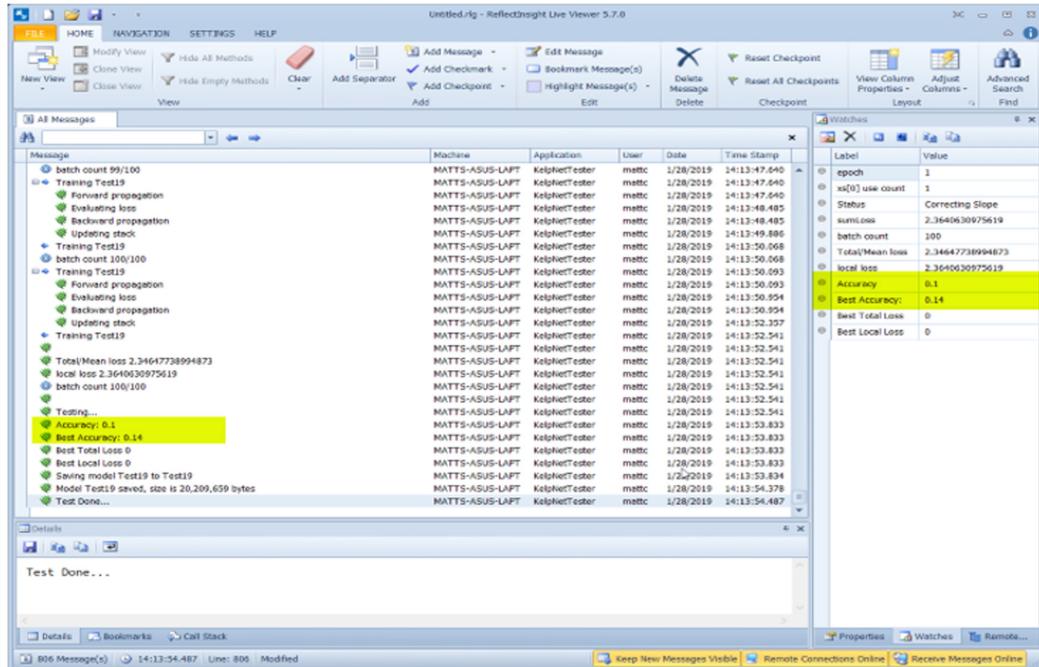
```
const int learningCount = 10000;
Real[][] trainData =
{
    new Real[] { 0, 0 },
    new Real[] { 1, 0 },
    new Real[] { 0, 1 },
    new Real[] { 1, 1 }
};
Real[][] trainLabel =
{
    new Real[] { 0 },
    new Real[] { 1 },
    new Real[] { 1 },
    new Real[] { 0 }
};
FunctionStack nn = CommonStacks.SimpleLinearSigmoidStack("Test1", 2, 2, new
MomentumSGD());
for (int i = 0; i < learningCount; i++)
    for (int j = 0; j < trainData.Length; j++)
        Trainer.TrainWithSoftmaxCrossEntropy(nn, trainData[j], trainLabel[j]);
```

## Accuracy

The other item that the Trainer object provides you with is a quick and easy way to get the accuracy of your model. Here is an example of a function call which tests the accuracy of a model:

```
Real accuracy = Trainer.Accuracy(nn, datasetY.Data, datasetY.Label);
```

The following screenshot shows ReflectInsight running during the execution of this function call, and you can see that accuracy is being tracked and reported:



This section would not be complete without providing you tips on how to improve the accuracy and predictive ability of your mode, thereby producing better results:

- Ensure that you have variety of data that covers almost all the scenarios and not biased to any one scenario. Instead of working on a limited data, ask for more data, which will improve the accuracy of the model.
- You must treat outliers and missing values properly to increase the accuracy. There are multiple methods to do this such as impute the mean, median or mode values in case of continuous features and for categorical features, use a class. For outliers, either delete them or perform some transformations.
- Finding the right features which will have maximum impact on the outcome is one of the key aspects of better accuracy. This will come from better domain knowledge and visualizations. It is imperative to consider as many relevant features and potential outcomes as possible prior to deploying a machine learning algorithm.

- Ensemble models are combining multiple models to improve the accuracy using bagging and boosting. This can improve the predictive performance more than any single model. Random forests are used many times for ensembling.
- Re-validate the model at proper time frequency. As your data changes, it is necessary to re-score the model with new data. If required, rebuild the models periodically with different techniques to challenge the model.

## Timing

As testing and training of your model occurs, it is important that you clearly see where the bottlenecks in your system are, as well as monitor your system and model performance. Kelp.Net provides exceptional debugging information for the developer, more so than any other deep learning framework. Many of the classes in Kelp.Net have extensive debug and timing information in them that will print out to the ReflectInsight live viewer (shown below) to help you understand how your algorithm is performing. This information is invaluable during all phases of execution.

The following screenshot depicts a deep learning model performing through a series of tests and displaying internal information. There is no need to instrument your code, Kelp.Net has it all built in and available without any additional work.

Message	Machine	Application	User	Date	Time Stamp
NdArray Reducing/Slope Correction took 5.9136 ms	MATTS-ASUS-LAPT	KelpNetTester	matte	1/29/2019	10:13:25.303
Adam Function Parameter Updating took 26.9246 ms	MATTS-ASUS-LAPT	KelpNetTester	matte	1/29/2019	10:13:25.326
NdArray Reducing/Slope Correction took 0.0061 ms	MATTS-ASUS-LAPT	KelpNetTester	matte	1/29/2019	10:13:25.434
Adam Function Parameter Updating took 0.0282 ms	MATTS-ASUS-LAPT	KelpNetTester	matte	1/29/2019	10:13:25.434
	MATTS-ASUS-LAPT	KelpNetTester	matte	1/29/2019	10:13:25.434
	MATTS-ASUS-LAPT	KelpNetTester	matte	1/29/2019	10:13:25.434
total loss 2.51737021747142	MATTS-ASUS-LAPT	KelpNetTester	matte	1/29/2019	10:13:25.434
local loss 2.47819201301621	MATTS-ASUS-LAPT	KelpNetTester	matte	1/29/2019	10:13:25.434
	MATTS-ASUS-LAPT	KelpNetTester	matte	1/29/2019	10:13:25.434
cDN12 total loss 0	MATTS-ASUS-LAPT	KelpNetTester	matte	1/29/2019	10:13:25.434
cDN13 total loss 0.00391803342945853	MATTS-ASUS-LAPT	KelpNetTester	matte	1/29/2019	10:13:25.434
	MATTS-ASUS-LAPT	KelpNetTester	matte	1/29/2019	10:13:25.434
cDN12 local loss 0	MATTS-ASUS-LAPT	KelpNetTester	matte	1/29/2019	10:13:25.434
cDN13 local loss 0.00385722407505512	MATTS-ASUS-LAPT	KelpNetTester	matte	1/29/2019	10:13:25.434
	MATTS-ASUS-LAPT	KelpNetTester	matte	1/29/2019	10:13:25.434
Trainer Accuracy took	MATTS-ASUS-LAPT	KelpNetTester	matte	1/29/2019	10:13:26.835
accuracy 0.13	MATTS-ASUS-LAPT	KelpNetTester	matte	1/29/2019	10:13:26.835
NdArray Reducing/Slope Correction took 4.3165 ms	MATTS-ASUS-LAPT	KelpNetTester	matte	1/29/2019	10:13:36.156
Adam Function Parameter Updating took 21.4792 ms	MATTS-ASUS-LAPT	KelpNetTester	matte	1/29/2019	10:13:36.188
NdArray Reducing/Slope Correction took 0.0056 ms	MATTS-ASUS-LAPT	KelpNetTester	matte	1/29/2019	10:13:36.188
Adam Function Parameter Updating took 0.0272 ms	MATTS-ASUS-LAPT	KelpNetTester	matte	1/29/2019	10:13:36.188
NdArray Reducing/Slope Correction took 1.3658 ms	MATTS-ASUS-LAPT	KelpNetTester	matte	1/29/2019	10:13:42.431
Adam Function Parameter Updating took 6.4585 ms	MATTS-ASUS-LAPT	KelpNetTester	matte	1/29/2019	10:13:42.439
NdArray Reducing/Slope Correction took 0.0056 ms	MATTS-ASUS-LAPT	KelpNetTester	matte	1/29/2019	10:13:42.439
Adam Function Parameter Updating took 0.0272 ms	MATTS-ASUS-LAPT	KelpNetTester	matte	1/29/2019	10:13:42.439

## Common stacks

Kelp.Net has the concept of *common stacks*, which are pre-defined FunctionStacks that can greatly reduce the amount of code you need to write. The following are several pre-defined function stacks and you can see how simple it would be to extend this class to include more of the most commonly used function stacks:

```
public static class CommonStacks
{
    ///////////////////////////////////////////////////
    /// <summary> Simple linear sigmoid stack. </summary>
    /// <param name="name"> The name. </param>
    /// <param name="verbose"> True to verbose. </param>
    /// <param name="inputs"> The inputs. </param>
    /// <param name="outputs"> The outputs. </param>
    /// <param name="optimizers"> A variable-length parameters list containing optimizers. </param>
    /// <returns> A FunctionStack. </returns>
    ///////////////////////////////////////////////////
    public static FunctionStack SimpleLinearSigmoidStack(string name, bool verbose, int inputs, int outputs, params Optimizer[] optimizers)
    {
        FunctionStack nn = new FunctionStack(name,
            new Linear(verbose, inputs, outputs, name: "l1 Linear"),
            new Sigmoid(name: "l1 Sigmoid", verbose: verbose),
            new Linear(verbose, inputs, outputs, name: "l2 Linear"));
        if (optimizers != null)
            nn.SetOptimizer(optimizers);
        return (nn);
    }
    ///////////////////////////////////////////////////
}

/// <summary> Simple linear ReLU stack. </summary>
/// <param name="name"> The name. </param>
/// <param name="verbose"> True to verbose. </param>
/// <param name="inputs"> The inputs. </param>
/// <param name="outputs"> The outputs. </param>
/// <param name="optimizers"> A variable-length parameters list containing optimizers. </param>
///
```

```
///<returns> A FunctionStack. </returns>
////////////////////////////////////////////////////////////////
public static FunctionStack SimpleLinearReLUStack(string name, bool verbose, int inputs, int
outputs, params Optimizer[] optimizers)
{
    FunctionStack nn = new FunctionStack(name,
        new Linear(verbose, inputs, outputs, name: "l1 Linear"),
        new ReLU(name: "l1 ReLU"),
        new Linear(verbose, inputs, outputs, name: "l2 Linear"));

    if (optimizers != null)
        nn.SetOptimizer(optimizers);
    return (nn);
}
////////////////////////////////////////////////////////////////
///<summary> MNIST capable stack. </summary>
///
///<param name="name"> The name. </param>
///<param name="verbose"> True to verbose. </param>
///<param name="inputs"> The inputs. </param>
///<param name="outputs"> The outputs. </param>
///<param name="optimizers"> A variable-length parameters list containing optimizers. </
param>
///
///<returns> A FunctionStack. </returns>
////////////////////////////////////////////////////////////////
public static FunctionStack MNISTCapableStack(string name, bool verbose, int inputs, int
outputs, params Optimizer[] optimizers)
{
    int N = 30;

    FunctionStack nn = new FunctionStack(name,
        new Linear(verbose, 28 * 28, N, name: "l1 Linear"), // L1
        new BatchNormalization(verbose, N, name: "l1 BatchNorm"),
        new ReLU(name: "l1 ReLU"),
        new Linear(verbose, N, N, name: "l2 Linear"), // L2
        new BatchNormalization(verbose, N, name: "l2 BatchNorm"),
        new ReLU(name: "l2 ReLU"),
```

```
new Linear(verbose, N, N, name: "l3 Linear"), // L3
new BatchNormalization(verbose, N, name: "l3 BatchNorm"),
new ReLU(name: "l3 ReLU"),
new Linear(verbose, N, N, name: "l4 Linear"), // L4
new BatchNormalization(verbose, N, name: "l4 BatchNorm"),
new ReLU(name: "l4 ReLU"),
new Linear(verbose, N, N, name: "l5 Linear"), // L5
new BatchNormalization(verbose, N, name: "l5 BatchNorm"),
new ReLU(name: "l5 ReLU"),
new Linear(verbose, N, N, name: "l6 Linear"), // L6
new BatchNormalization(verbose, N, name: "l6 BatchNorm"),
new ReLU(name: "l6 ReLU"),
new Linear(verbose, N, N, name: "l7 Linear"), // L7
new BatchNormalization(verbose, N, name: "l7 BatchNorm"),
new ReLU(name: "l7 ReLU"),
new Linear(verbose, N, N, name: "l8 Linear"), // L8
new BatchNormalization(verbose, N, name: "l8 BatchNorm"),
new ReLU(name: "l8 ReLU"),
new Linear(verbose, N, N, name: "l9 Linear"), // L9
new BatchNormalization(verbose, N, name: "l9 BatchNorm"),
new ReLU(name: "l9 ReLU"),
new Linear(verbose, N, N, name: "l10 Linear"), // L10
new BatchNormalization(verbose, N, name: "l10 BatchNorm"),
new ReLU(name: "l10 ReLU"),
new Linear(verbose, N, N, name: "l11 Linear"), // L11
new BatchNormalization(verbose, N, name: "l11 BatchNorm"),
new ReLU(name: "l11 ReLU"),
new Linear(verbose, N, N, name: "l12 Linear"), // L12
new BatchNormalization(verbose, N, name: "l12 BatchNorm"),
new ReLU(name: "l12 ReLU"),
new Linear(verbose, N, N, name: "l13 Linear"), // L13
new BatchNormalization(verbose, N, name: "l13 BatchNorm"),
new ReLU(name: "l13 ReLU"),
new Linear(verbose, N, N, name: "l14 Linear"), // L14
new BatchNormalization(verbose, N, name: "l14 BatchNorm"),
new ReLU(name: "l14 ReLU"),
new Linear(verbose, N, 10, name: "l15 Linear") // L15
```

```
 );  
  
if (optimizers != null)  
    nn.SetOptimizer(optimizers);  
return nn;  
}  
}
```

## Summary

In this chapter, we talked about testing and training of models. We also talked about measured accuracy and timing. Finally, we showed how to use CommonStacks in order to reduce the amount of coding required to create FunctionStacks.

In the next chapter, we will talk about how to load and save your models; something you will probably do in each test to ensure ease of reuse later on.



# CHAPTER 6

# Loading and Saving Models

Although short and brief, this chapter deals with how to load and save your models. The last thing you want to do, especially if your model is complex or time-consuming, is repeat the learning process each time you run it. Why not save and reload the model whenever necessary? This is exactly what we are going to discuss in this chapter.

Every deep learning model requires the ability to load and save its configuration to the disk. This is handled by the `ModelIO` class. The class itself is very small so the code is reproduced in its entirety here:

```
public class ModelIO
{
    public static void Save([NotNull] FunctionStack functionStack, [NotNull] string fileName)
    {
        Ensure.Argument(fileName).NotNullOrWhiteSpace("fileName is null");
        Ensure.Argument(functionStack).NotNull("functionStack is null");
        NetDataContractSerializer bf = new NetDataContractSerializer();
        RILogManager.Default?.SendDebug("Saving model " + functionStack.Name + " to " + fileName);

        try
        {
            using (Stream stream = File.OpenWrite(fileName))
            {
                bf.Serialize(stream, functionStack);
```

```
        }
    }
    catch (Exception ex)
    {
        RILogManager.Default?.SendException(ex.Message, ex);
    }
}

public static void Save([NotNull] SortedFunctionStack functionStack, [NotNull] string fileName)
{
    Ensure.Argument(fileName).NotNullOrWhiteSpace("fileName is null");
    Ensure.Argument(functionStack).NotNull("functionStack is null");

    NetDataContractSerializer bf = new NetDataContractSerializer();
    RILogManager.Default?.SendDebug("Saving model " + functionStack.Name + " to " + fileName);
    try
    {
        using (Stream stream = File.OpenWrite(fileName))
        {
            bf.Serialize(stream, functionStack);
        }
    }
    catch (Exception ex)
    {
        RILogManager.Default?.SendException(ex.Message, ex);
    }
}

[CanBeNull]
public static FunctionStack Load([NotNull] string fileName)
{
    Ensure.Argument(fileName).NotNullOrWhiteSpace("fileName is null");
    NetDataContractSerializer bf = new NetDataContractSerializer();
    FunctionStack result = null;
    RILogManager.Default?.SendDebug("Loading model from " + fileName);

    try
    {
        using (Stream stream = File.OpenRead(fileName))
```

```
{  
    result = (FunctionStack)bf.Deserialize(stream);  
}  
}  
}  
catch (Exception ex)  
{  
    RILogManager.Default?.SendException(ex.Message, ex);  
}  
  
if (result?.Functions != null)  
{  
    foreach (Function function in result?.Functions)  
    {  
        RILogManager.Default?.SendDebug("Configuring " + function.Name);  
        function.ResetState();  
  
        if (function.Optimizers != null)  
        {  
            RILogManager.Default?.SendDebug("Configuring " + function.Name + " Optimizers");  
            foreach (var t in function.Optimizers)  
                t.ResetParams();  
        }  
  
        if (function is IParallelizable parallelizable)  
        {  
            RILogManager.Default?.SendDebug("Configuring " + function.Name + " Kernel");  
            parallelizable.CreateKernel();  
        }  
    }  
}  
  
return result;  
}  
}
```

As you can see, saving a model to the disk requires serializing each function in the stack, which is a straightforward process. Loading a model, on the other hand, requires more work. First, a new `NetDataContractSerializer` object is created. The file is then opened and the function stack is deserialized into the memory. Each function

will have its state reset, and each function optimizer present will have its parameters reset. Finally, the OpenCL Kernel will be created for use if the function implements the IParallelizable interface.

## Loading models

Loading models can be accomplished via the ModelIO.Load method. Here is the complete code of this method:

```
[CanBeNull]
public static FunctionStack Load([NotNull] string fileName)
{
    Ensure.Argument(fileName).NotNullOrWhiteSpace("fileName is null");
    NetDataContractSerializer bf = new NetDataContractSerializer();
    FunctionStack result = null;
    RILogManager.Default?.SendDebug("Loading model from " + fileName);

    try
    {
        using (Stream stream = File.OpenRead(fileName))
        {
            result = (FunctionStack)bf.Deserialize(stream);
            RILogManager.Default?.SendDebug("Model " + result.Name + " loaded, size is " + stream.Length.ToString("N0") + " bytes");
        }
    }
    catch (Exception ex)
    {
        RILogManager.Default?.SendException(ex.Message, ex);
    }

    if (result?.Functions != null)
    {
        foreach (Function function in result?.Functions)
        {
            RILogManager.Default?.SendDebug("Resetting " + function.Name);
            function.ResetState();

            if (function.Optimizers != null)
```

```
{  
    RILogManager.Default?.SendDebug("Resetting " + function.Name + " Optimizers");  
    foreach (var t in function.Optimizers)  
        t.ResetParams();  
    }  
  
    if (function is IParallelizable parallelizable)  
    {  
        RILogManager.Default?.SendDebug("Creating " + function.Name + " Kernel");  
        parallelizable.CreateKernel();  
    }  
}  
}  
  
return result;  
}
```

```
FunctionStack testnn = ModelIO.Load("test.nn");
```

## Saving models

Saving a model is also a straightforward process and can be done via the ModelIO.Save method:

```
public static void Save([NotNull] FunctionStack functionStack, [NotNull] string fileName)  
{  
    Ensure.Argument(fileName).NotNullOrWhiteSpace("fileName is null");  
    Ensure.Argument(functionStack).NotNull("functionStack is null");  
  
    NetDataContractSerializer bf = new NetDataContractSerializer();  
    RILogManager.Default?.SendDebug("Saving model " + functionStack.Name + " to " + fileName);  
  
    try  
    {  
        using (Stream stream = File.OpenWrite(fileName))  
        {  
            bf.Serialize(stream, functionStack);  
        }  
    }
```

```
FileStream fs = File.OpenRead(fileName);
RILogManager.Default?.SendDebug("Model " + functionStack.Name + " saved, size is " +
fs.Length.ToString("N0") + " bytes");
}
catch (Exception ex)
{
    RILogManager.Default?.SendException(ex.Message, ex);
}
}
```

```
FunctionStack nn = new FunctionStack("Test1",
new Linear(2, 2, name: "l1 Linear"),
new Sigmoid(name: "l1 Sigmoid"),
new Linear(2, 2, name: "l2 Linear"));
ModelIO.Save(nn, "test.nn");
```

## Model size

You will notice that the size of your model when saved will differ, depending on where in your process you called the Save method. As we mentioned earlier, the computations are what are stored, not the operations, so as your model trains, it increases its memory footprint. As your model adds new layers and optimizers, it increases its memory footprint, and so on until you complete your testing and training. It is feasible to expect that your model which is depending on the complexity, may run into the MB size quite easily. Do not worry; saving to and loading from disk is very fast.

How much serialized information you need may vary based on your usage. Suppose if you are designing a visual layer to Kelp.Net and you want the user to be able to save the model empty (as soon as created). You may also elect to programmatically save the model information periodically throughout the user session, as a backup. Regardless of your approach, the Save and Load methods make this extremely easy to do.

The following instrumentation shows the size of each model when saved. This instrumentation is provided for you automatically via the Kelp.Net framework:

Testing...	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/28/2019	14:13:52.541
Accuracy: 0.1	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/28/2019	14:13:53.833
Best Accuracy: 0.14	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/28/2019	14:13:53.833
Best Total Loss 0	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/28/2019	14:13:53.833
Best Local Loss 0	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/28/2019	14:13:53.833
Saving model Test19 to Test19	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/28/2019	14:13:53.834
Model Test19 saved, size is 20,209,659 bytes	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/28/2019	14:13:54.378

## Summary

In this chapter, we learned how to load and save models. In the next chapter, we will talk about testing and training of models in detail.



# CHAPTER 7

# Sample Deep Learning Tests

In this chapter, we will examine some of the sample models and tests which accompany Kelp.Net. These tests have been created to show you the myriad ways of using Kelp.Net to solve problems. Please note that these were accurate at the time of writing this book, but I am always adding and improving so refer to the latest version of the software when and where you can.

These samples try and select various aspects of Kelp.Net such as very large FunctionStacks, Optimizers, activation functions, and more. In many cases, there are multiple ways to solve a problem, and I have tried to show this in these sections. Please feel free to create and/or modify any of these to fit your problem space, and please take these tests, try all combinations of things, write your own tests, and embellish what is here.

These samples also show how to programmatically configure ReflectInsight to clear out messages and watches, set a category label, and much more. It is a great reference for you to learn how to use this powerful logging tool. I have used this extensively in order to show you information relative to the tests we are doing.

## A simple XOR problem

This test is very simple and basic and is designed to create a small but efficient neural network (NN), and then train it on the data we provide. This NN will function as an exclusive OR gate, or XOR. Exclusive or (XOR, EOR, or EXOR) is a logical operator which outputs a true value when either of the operands are true (one is true and the other one is false) but both are not true, and both are not false.

Our training and label datasets look like this. As you can see, we have one set of training data and one set of training labels:

```
Real[][] trainData =  
{  
    new Real[] { 0, 0 },  
    new Real[] { 1, 0 },  
    new Real[] { 0, 1 },  
    new Real[] { 1, 1 }  
};  
  
Real[][] trainLabel =  
{  
    new Real[] { 0 },  
    new Real[] { 1 },  
    new Real[] { 1 },  
    new Real[] { 0 }  
};
```

## Complete source code

The complete source code is shown below, including comments. You can find this source code in the Test1 test of Kelp.Net tests:

```
class Test1  
{  
    public static void Run()  
    {  
        const int learningCount = 10000;  
  
        Real[][] trainData =  
        {  
            new Real[] { 0, 0 },  
            new Real[] { 1, 0 },  
            new Real[] { 0, 1 },  
            new Real[] { 1, 1 }  
        };  
  
        Real[][] trainLabel =  
        {
```

```
    new Real[] { 0 },
    new Real[] { 1 },
    new Real[] { 1 },
    new Real[] { 0 }
};
```

Here, we will create a new function stack that holds three different functions. From left to right, we have a linear function, a sigmoid, and finally, a second linear function:

```
FunctionStack nn = new FunctionStack("Test1",
    new Linear(2, 2, name: "l1 Linear"),
    new Sigmoid(name: "l1 Sigmoid"),
    new Linear(2, 2, name: "l2 Linear"));
//Set the optimizer as a new instance of the Momentum Stochastic Gradient Descent algorithm.
nn.SetOptimizer(new MomentumSGD());

//Perform the actual training.

for (int i = 0; i < learningCount; i++)
{
    for (int j = 0; j < trainData.Length; j++)
    {
        Trainer.Train(nn, trainData[j], trainLabel[j], new SoftmaxCrossEntropy());
    }
}

//Create some test data and run some predictions against it

foreach (Real[] input in trainData)
{
    NdArray result = nn.Predict(input)?[0];
    int resultIndex = Array.IndexOf(result?.Data, result.Data.Max());
    Info($"{input[0]} xor {input[1]} = {resultIndex} {result}");
}

//Save the model to disk.
ModelIO.Save(nn, "test1.nn");
```

```
//Load the model back in.
FunctionStack testnn = ModelIO.Load("test1.nn");

//Let the N describe the functions, optimizers, inputs and more.
Info(testnn.Describe());

//Run the tests again after reloading the model from disk to ensure consistency.
foreach (Real[] input in trainData)
{
    NdArray result = testnn?.Predict(input)?[0];
    int resultIndex = Array.IndexOf(result?.Data, result?.Data.Max());
    Info($"{input[0]} xor {input[1]} = {resultIndex} {result}");
}
```

## Output

The following screenshot shows the output after running the preceding test. As you can see, our algorithm was able to successfully train and recognize the data we provided. You will notice a lot of messages in the output. This is because most methods in Kelp.Net take a Boolean parameter that indicates if the function should be verbose during the operation or not. If true, then the extra information will be displayed to help you track the progress of your model:

Message	Category	Machine	Application	User	Date	Time Stamp
Saving Model...	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	04:54:17.014
Saving model Test1 to test.nn	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	04:54:17.018
Model Test1 saved, size is 23,004 bytes	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	04:54:17.024
Loading Model...	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	04:54:17.082
Loading model from test.nn	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	04:54:17.082
Model Test1 loaded, size is 23,004 bytes	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	04:54:17.134
Resetting I1 Linear	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	04:54:17.245
Resetting I1 Linear Optimizers	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	04:54:17.245
Creating I1 Linear Kernel	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	04:54:17.245
Resetting I1 Sigmoid	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	04:54:17.245
Resetting I1 Sigmoid Optimizers	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	04:54:17.245
Creating I1 Sigmoid Kernel	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	04:54:17.245
Resetting I2 Linear	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	04:54:17.245
Resetting I2 Linear Optimizers	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	04:54:17.245
Creating I2 Linear Kernel	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	04:54:17.245
- Function: I1 Linear	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	04:54:17.245
Test Start...	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	04:54:17.245
[2.26594662 -3.85293982]	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	04:54:17.245
0 xor 0 = [ 2.26594662 -3.85293982] ✓	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	04:54:17.245
[-3.41753866 2.45219502] ✓	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	04:54:17.245
1 xor 0 = [ -3.41753866 2.45219502] ✓	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	04:54:17.245
[-3.29385582 2.57134337] ✓	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	04:54:17.245
0 xor 1 = [-3.29385582 2.57134337] ✓	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	04:54:17.245
[ 2.34085338 -3.93792795] ✓	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	04:54:17.245
1 xor 1 = [ 2.34085338 -3.93792795] ✓	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	04:54:17.245
Test Done...	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	04:54:17.252

If you can recall that we used a Momentum Stochastic Gradient Descent optimizer for this test. It performed well and was fairly fast. In fact, it took 1 minute and 10 seconds to complete the testing, save the model, load it back in and verify it.

```

    0 xor 0 = 0 [ 2.26594662 -3.85293982]
    [-3.41753866 2.45219502]
    1 xor 0 = 1 [-3.41753866 2.45219502]
    [-3.29385582 2.57134337]
    0 xor 1 = 1 [-3.29385582 2.57134337]
    [ 2.34085338 -3.93792795]
    1 xor 1 = 0 [ 2.34085338 -3.93792795]
    Test took 1 minute, 10 seconds, 884 milliseconds

```

## A penny for your thoughts

Let us go ahead and make an adjustment to this model test. Let us find the line of code that sets the optimizer and change it to Adam. Just look for this line of code:

```
nn.SetOptimizer(new MomentumSGD());
```

When you change it, it will look like this:

```
nn.SetOptimizer(new Adam());
```

As you can see, this is a very simple and fast change to make. After running the model test again, we achieved the same end results, but the training took considerably longer. In fact, it took 22 minutes and 57 seconds!

```

    0 xor 0 = 0 [ 0.24281167 -1.80639227]
    [-1.37518631 0.42838749]
    1 xor 0 = 1 [-1.37518631 0.42838749]
    [-1.21995861 0.49628958]
    0 xor 1 = 1 [-1.21995861 0.49628958]
    [ 0.35427044 -1.97970915]
    1 xor 1 = 0 [ 0.35427044 -1.97970915]
    Test took 22 minutes, 57 seconds, 439 milliseconds

```

An extra 21 minutes all because we changed to an optimizer that has been heralded as one of the best to use. Why do you think that happened? Could it be one of our hyperparameters for the Adam optimizer is not the best?

Your exercise will be to work with the parameters which this method takes and see if you can improve the timing from what we see above. Try changing different hyperparameters and see the results that you get. Here is a brief overview of what they are:

- **Alpha** is also referred to as the learning rate or step size. Larger values result in faster initial learning before the update. Smaller values result in slower learning all the way down during training. Try adjusting this value first! As a hint, try the following learning rates and see what difference, if any, they make: 0.0005, 0.001, and 0.00146.
- Beta1 is the decay rate for the first moment estimates.
- Beta2 is the decay rate for the second moment estimates.
- Epsilon is a very small number whose only purpose is to prevent divide by zero errors.

Changing the hyperparameters will give a feel of just how easy it is to change parameters and re-run your tests; something you will do countless times.

## A simple XOR problem (part 2)

This test is designed to show you how you can apply different settings (functions, optimizers, and so on.) to train your algorithm. This sample will solve the XOR problem as well; however, we will use MeanSquaredError for the loss function, and we will train/update the model in batches. This test shows how you can approach the same problem in multiple ways sometimes, and how different approaches and their associated hyperparameters may have a positive or negative effect.

## Complete source code

The complete source code is shown below, including comments. You can find this source code in the Test2 test of Kelp.Net tests:

```
public static void Run()
{
    const int learningCount = 10000;

    Real[][] trainData =
    {
        new Real[] { 0, 0 },
        new Real[] { 1, 0 },
        new Real[] { 0, 1 },
        new Real[] { 1, 1 }
    };

    Real[][] trainLabel =
```

```
{  
    new Real[] { 0 },  
    new Real[] { 1 },  
    new Real[] { 1 },  
    new Real[] { 0 }  
};  
  
FunctionStack nn = new FunctionStack("Test2",  
    new Linear(2, 2, name: "l1 Linear"),  
    new ReLU(name: "l1 ReLU"),  
    new Linear(2, 1, name: "l2 Linear"));  
  
nn.SetOptimizer(new AdaGrad());  
  
RILogManager.Default?.SendDebug("Training...");  
for (int i = 0; i < learningCount; i++)  
{  
    //use MeanSquaredError for loss function  
    Trainer.Train(nn, trainData[0], trainLabel[0], new MeanSquaredError(), false);  
    Trainer.Train(nn, trainData[1], trainLabel[1], new MeanSquaredError(), false);  
    Trainer.Train(nn, trainData[2], trainLabel[2], new MeanSquaredError(), false);  
    Trainer.Train(nn, trainData[3], trainLabel[3], new MeanSquaredError(), false);  
  
    //If you do not update every time after training, you can update it as a mini batch  
    nn.Update();  
}  
  
RILogManager.Default?.SendDebug("Test Start...");  
foreach (Real[] val in trainData)  
{  
    NdArray result = nn.Predict(val)[0];  
    RILogManager.Default?.SendDebug($"'{val[0]} xor {val[1]} = {(result.Data[0] > 0.5 ? 1 : 0)}  
{result}'");  
}  
}
```

## Output

Message	Category	Machine	Application	User	Date	Time Stamp
Saving Model...	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	04:54:17.014
Saving model Test1 to test.nn	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	04:54:17.018
Model Test1 saved, size is 23,004 bytes	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	04:54:17.024
Loading Model...	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	04:54:17.082
Loading model from test.nn	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	04:54:17.082
Model Test1 loaded, size is 23,004 bytes	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	04:54:17.134
Resetting I1 Linear	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	04:54:17.245
Resetting I1 Linear Optimizers	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	04:54:17.245
Creating I1 Linear Kernel	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	04:54:17.245
Resetting I1 Sigmoid	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	04:54:17.245
Resetting I1 Sigmoid Optimizers	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	04:54:17.245
Creating I1 Sigmoid Kernel	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	04:54:17.245
Resetting I2 Linear	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	04:54:17.245
Resetting I2 Linear Optimizers	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	04:54:17.245
Creating I2 Linear Kernel	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	04:54:17.245
- Function: I1 Linear	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	04:54:17.245
Test Start...	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	04:54:17.245
[ 2.26594662 -3.85293982]	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	04:54:17.245
0 xor 0 = 0 [ 2.26594662 -3.85293982]	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	04:54:17.245
[ -3.41753866 2.45219502]	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	04:54:17.245
1 xor 0 = 1 [-3.41753866 2.45219502]	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	04:54:17.245
[ -3.29385582 2.57134337]	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	04:54:17.245
0 xor 1 = 1 [-3.29385582 2.57134337]	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	04:54:17.245
[ 2.34085338 -3.93792795]	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	04:54:17.245
1 xor 1 = 0 [ 2.34085338 -3.93792795]	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	04:54:17.245
Test Done...	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	04:54:17.252

## Recurrent Neural Network Language Models (RNNLM)

In this test, we will build and test a model that is designed to handle **Recurrent Neural Network Language Models (RNNLM)**. This uses a simple RNN architecture.

## Complete source code

The complete source code is shown below, including comments. You can find this source code in the Test9 test of Kelp.Net tests:

```
const int TRAINING_EPOCHS = 5;
const int N_UNITS = 100;
const string DOWNLOAD_URL = "https://raw.githubusercontent.com/wojzaremba/lstm/master/
data/";
const string TRAIN_FILE = "ptb.train.txt";
const string TEST_FILE = "ptb.test.txt";
public static void Run()
{
    //Build the vocabulary
```

```
Vocabulary vocabulary = new Vocabulary();
string trainPath = InternetFileDownloader.Download(DOWNLOAD_URL + TRAIN_FILE, TRAIN_FILE);
string testPath = InternetFileDownloader.Download(DOWNLOAD_URL + TEST_FILE, TEST_FILE);

//Load the data
int[] trainData = vocabulary.LoadData(trainPath);
int[] testData = vocabulary.LoadData(testPath);
int nVocab = vocabulary.Length;

//Create the network model. We will use a simple Recurrent Neural Network to do this.
FunctionStack model = new FunctionStack("Test9",
    new EmbedID(nVocab, N_UNITS, name: "I1 EmbedID"),
    new Linear(N_UNITS, N_UNITS, name: "I2 Linear"),
    new Tanh("I2 Tanh"),
    new Linear(N_UNITS, nVocab, name: "I3 Linear"),
    new Softmax("I3 Softmax")
);

model.SetOptimizer(new Adam());
List<int> s = new List<int>();
```

Start the training. We will use SoftmaxCrossEntropy for this model.

As the name suggests, the Softmax function is a soft version of the max function. Instead of selecting one maximum value, it breaks the whole (1) with maximal element getting the largest portion of the distribution, but other smaller elements getting some of it as well. Because of this property of the Softmax function that it outputs a probability distribution, we use it as the final layer in neural networks. For this, we need to calculate the derivative or gradient and pass it back to the previous layer during backpropagation.

Cross entropy indicates the distance between what the model believes the output distribution should be and what the original distribution really is.

```
SoftmaxCrossEntropy softmaxCrossEntropy = new SoftmaxCrossEntropy();
for (int epoch = 0; epoch < TRAINING_EPOCHS; epoch++)
{
    for (int pos = 0; pos < trainData.Length; pos++)
    {
        NdArray h = new NdArray(new Real[N_UNITS]);
```

```
int id = trainData[pos];
s.Add(id);

if (id == vocabulary.EosID)
{
    Real accumloss = 0;
    Stack<NdArray> tmp = new Stack<NdArray>();

    for (int i = 0; i < s.Count; i++)
    {
        int tx = i == s.Count - 1 ? vocabulary.EosID : s[i + 1];
```

We now use our network model a bit differently than the other tests I have shown. Instead of performing forward and backpropagation on the entire functionStack itself, the code given below shows you how you can directly propagate a specific layer in your network by going straight to the function itself.

Access the EmbedID function and then perform forward propagation on it:

```
//l1 EmbedID
NdArray l1 = model.Functions[0].Forward(s[i])[0];
```

Access the Linear function and then perform forward propagation on it:

```
//l2 Linear
NdArray l2 = model.Functions[1].Forward(h)[0];
```

Now, add the results of the two arrays together:

```
//Add
NdArray xK = l1 + l2;
```

Access the Tanh function and then perform forward propagation on it:

```
//l2 Tanh
h = model.Functions[2].Forward(xK)[0];
```

Access the Linear function and then perform forward propagation on it:

```
//l3 Linear
NdArray h2 = model.Functions[3].Forward(h)[0];
```

Evaluate the loss with the softmaxCrossEntropy object:

```
Real loss = softmaxCrossEntropy.Evaluate(h2, tx);
tmp.Push(h2);
```

```
    accumloss += loss;
}
RILogManager.Default?.SendDebug(accumloss.ToString());
for (int i = 0; i < s.Count; i++)
{
```

Perform backward propagation on the entire model at once (all functions) versus individually:

```
    model.Backward(tmp.Pop());
}

model.Update();
s.Clear();
}

if (pos % 100 == 0)
{
    RILogManager.Default?.SendDebug(pos + "/" + trainData.Length + " finished");
}
}
}
```

Start testing:

```
RILogManager.Default?.SendDebug("Test Start.");
Real sum = 0;
int wnum = 0;
List<int> ts = new List<int>();
bool unkWord = false;

for (int pos = 0; pos < 1000; pos++)
{
    int id = testData[pos];
    ts.Add(id);

    if (id > trainData.Length)
{
```

We found an unknown word:

```
unkWord = true;
}

if (id == vocabulary.EosID)
{
    if (!unkWord)
    {
        RILogManager.Default?.SendDebug("pos: " + pos);
        RILogManager.Default?.SendDebug("tsLen: " + ts.Count);
        RILogManager.Default?.SendDebug("sum: " + sum);
        RILogManager.Default?.SendDebug("wnum: " + wnum);
        RILogManager.Default?.ViewerSendWatch("pos", pos);
        RILogManager.Default?.ViewerSendWatch("tsLen", ts.Count);
        RILogManager.Default?.ViewerSendWatch("sum", sum);
        RILogManager.Default?.ViewerSendWatch("wnum", wnum);
        sum += CalPs(model, ts);
        wnum += ts.Count - 1;
    }
    else
    {
        unkWord = false;
    }
    ts.Clear();
}
}

RILogManager.Default?.SendDebug(Math.Pow(2.0, sum / wnum).ToString());
}

static Real CalPs(FunctionStack model, List<int> s)
{
    Real sum = 0;
    NdArray h = new NdArray(new Real[N_UNITS]);
    for (int i = 1; i < s.Count; i++)
    {
```

Perform forward propagation on the first four individual functions:

```
//I1 Linear
NdArray xK = model.Functions[0].Forward(s[i])[0];
//I2 Linear
NdArray l2 = model.Functions[1].Forward(h)[0];
for (int j = 0; j < xK.Data.Length; j++)
    xK.Data[j] += l2.Data[j];
//I2 Tanh
h = model.Functions[2].Forward(xK)[0];
//I3 Softmax(I3 Linear)
NdArray yv = model.Functions[4].Forward(model.Functions[3].Forward(h))[0];
Real pi = yv.Data[s[i - 1]];
sum -= Math.Log(pi, 2);
}
return sum;
}
```

## Vocabulary

In this model, we will use vocabulary so let us talk about how this will happen. Loading of the vocabulary file is a simple process and handled by the LoadData method of the Vocabulary class as follows:

```
public List<string> Data = new List<string>();
public int EosID = -1;
public int Length => Data.Count;
public int[] LoadData(string fileName)
{
    int[] result;

    using (FileStream fs = new FileStream(fileName, FileMode.Open))
    {
        StreamReader sr = new StreamReader(fs);
        string strText = sr.ReadToEnd();
        sr.Close();

        string[] replace = strText.Replace("\r\n", "\n").Replace("\n", "<EOS>").Trim().Split(new[] { ' ' },
StringSplitOptions.RemoveEmptyEntries);
        Data.AddRange(replace);
    }
}
```

```

Data = new List<string>(Data.Distinct());
result = new int[replace.Length];
for (int i = 0; i < replace.Length; i++)
    result[i] = Data.IndexOf(replace[i]);
if (EosID == -1)
    EosID = Data.IndexOf("<EOS>");
}

return result;
}

```

## Output

Message	Category	Machine	Application	User	Date	Time Stamp
Adam Function Parameter Updating took 5.8458 ms	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	08:24:13.702
NdArray Reducing/Slope Correction took 0.0149 ms	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	08:24:13.702
Adam Function Parameter Updating took 0.0693 ms	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	08:24:13.702
NdArray Reducing/Slope Correction took 32.2415 ms	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	08:24:13.714
Adam Function Parameter Updating took 104.1553 ms	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	08:24:13.917
NdArray Reducing/Slope Correction took 0.8744 ms	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	08:24:14.040
Adam Function Parameter Updating took 6.0751 ms	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	08:24:14.074
222.294274648892	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	08:24:16.199
NdArray Reducing/Slope Correction took 22.7685 ms	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	08:24:20.165
Adam Function Parameter Updating took 99.0779 ms	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	08:24:20.269
NdArray Reducing/Slope Correction took 0.5628 ms	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	08:24:20.381
Adam Function Parameter Updating took 2.8123 ms	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	08:24:20.381
NdArray Reducing/Slope Correction took 0.0123 ms	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	08:24:20.381
Adam Function Parameter Updating took 0.02 ms	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	08:24:20.412
NdArray Reducing/Slope Correction took 29.9504 ms	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	08:24:20.436
Adam Function Parameter Updating took 92.6821 ms	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	08:24:20.510
NdArray Reducing/Slope Correction took 0.5849 ms	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	08:24:20.621
Adam Function Parameter Updating took 2.4957 ms	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	08:24:20.655
309.577715618118	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	08:24:23.505
NdArray Reducing/Slope Correction took 19.0591 ms	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	08:24:28.899
Adam Function Parameter Updating took 93.1704 ms	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	08:24:28.992
NdArray Reducing/Slope Correction took 0.166 ms	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	08:24:29.102
Adam Function Parameter Updating took 1.1113 ms	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	08:24:29.134
NdArray Reducing/Slope Correction took 0.0041 ms	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	08:24:29.134
Adam Function Parameter Updating took 0.019 ms	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	08:24:29.134
NdArray Reducing/Slope Correction took 22.897 ms	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	08:24:29.156

## Word prediction test

This network model is trained to predict a word, given the preceding word sequence. To do this, we create a simple neural network to train a language model, which is based on the word level Penn Tree Bank dataset. As a comparison, this test achieves a perplexity score of 115 for a small model in about 1 hour of processing time. It can achieve a perplexity score of 81 for a big model, given 24 hours of processing time.

## Complete source code

The complete source code is shown below, including comments. You can find this source code in the Test10 test of Kelp.Net tests:

```
const int N_EPOCH = 39;
const int N_UNITS = 650;
const int BATCH_SIZE = 20;
const int BPROP_LEN = 35;
const int GRAD_CLIP = 5;
const string DOWNLOAD_URL = "https://raw.githubusercontent.com/wojzaremba/lstm/master/
data/";
const string TRAIN_FILE = "ptb.train.txt";
const string VALID_FILE = "ptb.valid.txt";
const string TEST_FILE = "ptb.test.txt";

private const string LogPath = "Histogram.log";
private static HistogramLogWriter _logWriter;
private static FileStream _outputStream;
private static int _isCompleted = -1;
private static bool _running;

public static void Run()
{
    RILogManager.Add("Test10", "Test10");
    RILogManager.SetDefault("Test10");
    RILogManager.Default?.ViewerClearAll();
    _outputStream = File.Create(LogPath);
    _logWriter = new HistogramLogWriter(_outputStream);
    _logWriter.Write(DateTime.Now);

    var recorder = HistogramFactory
        .With64BitBucketSize()
        ?.WithValuesFrom(1)
        ?.WithValuesUpTo(2345678912345)
        ?.WithPrecisionOf(3)
        ?.WithThreadSafeWrites()
        ?.WithThreadSafeReads()
        ?.Create();
```

```
var accumulatingHistogram = new LongHistogram(2345678912345, 3);
var size = accumulatingHistogram.GetEstimatedFootprintInBytes();
RILogManager.Default?.ViewerSendWatch("Histogram Size", $"{size} bytes ({size / 1024.0 / 1024.0:F2} MB)");

accumulatingHistogram.OutputPercentileDistribution(Console.Out,
outputValueUnitScalingRatio: OutputScalingFactor.None, useCsvFormat: true);
Console.WriteLine();
accumulatingHistogram.OutputPercentileDistribution(Console.Out,
outputValueUnitScalingRatio: OutputScalingFactor.TimeStampToMicroseconds, useCsvFormat: true);
Console.WriteLine();
accumulatingHistogram.OutputPercentileDistribution(Console.Out,
outputValueUnitScalingRatio: OutputScalingFactor.TimeStampToMilliseconds, useCsvFormat: true);
Console.WriteLine();

accumulatingHistogram.OutputPercentileDistribution(Console.Out,
outputValueUnitScalingRatio: OutputScalingFactor.TimeStampToSeconds, useCsvFormat: true);
DocumentResults(accumulatingHistogram, recorder);

RILogManager.Default?.SendDebug("Building Vocabulary.");
DocumentResults(accumulatingHistogram, recorder);
Vocabulary vocabulary = new Vocabulary();
DocumentResults(accumulatingHistogram, recorder);
RILogManager.Default?.SendDebug("Downloading files.");
string trainPath = InternetFileDownloader.Download(DOWNLOAD_URL + TRAIN_FILE, TRAIN_FILE);
DocumentResults(accumulatingHistogram, recorder);

string validPath = InternetFileDownloader.Download(DOWNLOAD_URL + VALID_FILE, VALID_FILE);
DocumentResults(accumulatingHistogram, recorder);
string testPath = InternetFileDownloader.Download(DOWNLOAD_URL + TEST_FILE, TEST_FILE);
DocumentResults(accumulatingHistogram, recorder);
RILogManager.Default?.SendDebug("Loading Data.");
int[] trainData = vocabulary.LoadData(trainPath);
DocumentResults(accumulatingHistogram, recorder);
```

```
int[] validData = vocabulary.LoadData(validPath);
DocumentResults(accumulatingHistogram, recorder);
int[] testData = vocabulary.LoadData(testPath);
DocumentResults(accumulatingHistogram, recorder);
RILogManager.Default?.ViewerSendWatch("Vocabulary Len", vocabulary.Length);
int nVocab = vocabulary.Length;
bool verbose = true;

RILogManager.Default?.SendDebug($"Neural Network Initializing ({N_UNITS}) units.");
FunctionStack model = new FunctionStack("Test10",
    new EmbedID(verbose, nVocab, N_UNITS, name: "I1 EmbedID"),
    new Dropout(true),
    new LSTM(true, N_UNITS, N_UNITS, name: "I2 LSTM"),
    new Dropout(true),
    new LSTM(true, N_UNITS, N_UNITS, name: "I3 LSTM"),
    new Dropout(true),
    new Linear(true, N_UNITS, nVocab, name: "I4 Linear")
);
DocumentResults(accumulatingHistogram, recorder);

// Do not cease at the given threshold, correct the rate by taking the rate from L2Norm of all
parameters
RILogManager.Default?.ViewerSendWatch("Gradient Clipping", GRAD_CLIP);
GradientClipping gradientClipping = new GradientClipping(threshold: GRAD_CLIP);
SGD sgd = new SGD(learningRate: 1);
model.SetOptimizer(gradientClipping, sgd);
RILogManager.Default?.ViewerSendWatch("Learning Rate", sgd.LearningRate);
DocumentResults(accumulatingHistogram, recorder);

Real wholeLen = trainData.Length;
RILogManager.Default?.ViewerSendWatch("Training Data Len", Helpers.
BytesToString(wholeLen));
int jump = (int)Math.Floor(wholeLen / BATCH_SIZE);
RILogManager.Default?.ViewerSendWatch("Jump", jump);

int epoch = 0;
Stack<NdArray[]> backNdArrays = new Stack<NdArray[]>();
RILogManager.Default?.SendDebug("Starting Training")
```

```
double dVal;
NdArray x = new NdArray(new[] { 1 }, BATCH_SIZE, (Function)null);
NdArray t = new NdArray(new[] { 1 }, BATCH_SIZE, (Function)null);

for (int i = 0; i < jump * N_EPOCH; i++)
{
    for (int j = 0; j < BATCH_SIZE; j++)
    {
        x.Data[j] = trainData[(int)((jump * j + i) % wholeLen)];
        t.Data[j] = trainData[(int)((jump * j + i + 1) % wholeLen)];
    }

    NdArray[] result = model.Forward(true, x);
    Real sumLoss = new SoftmaxCrossEntropy().Evaluate(result, t);
    backNdArrays.Push(result);
    RILogManager.Default?.SendDebug("[{0}/{1}] Local Loss: {2}", i + 1, jump, sumLoss);
    RILogManager.Default?.ViewerSendWatch("Local Loss", sumLoss);
    RILogManager.Default?.ViewerSendWatch("Current Iteration", i+1);
    RILogManager.Default?.ViewerSendWatch("Jump", jump);

    //Run truncated BPTT
    RILogManager.Default?.SendDebug("Running truncated BPTT");
    if ((i + 1) % BPROP_LEN == 0)
    {
        for (int j = 0; backNdArrays.Count > 0; j++)
        {
            RILogManager.Default?.SendDebug("Backward" + backNdArrays.Count);
            model.Backward(true, backNdArrays.Pop());
        }
    }

    RILogManager.Default?.SendDebug("Updating Model");
    model.Update();
    model.ResetState();
}

if ((i + 1) % jump == 0)
{
    epoch++;
}
```

```
RILogManager.Default?.SendDebug("Evaluating Model");
dVal = Evaluate(model, validData);
RILogManager.Default?.SendDebug($"Validation Perplexity: {dVal}");
RILogManager.Default?.ViewerSendWatch("Validation Perplexity", dVal);
if (epoch >= 6)
{
    sgd.LearningRate /= 1.2;
    RILogManager.Default?.SendDebug("Adjusting Learning Rate to " + sgd.LearningRate);
}
}
DocumentResults(accumulatingHistogram, recorder);
}

RILogManager.Default?.SendDebug("Test Starting");
dVal = Evaluate(model, testData);
RILogManager.Default?.SendDebug("Test Perplexity: " + dVal);
RILogManager.Default?.ViewerSendWatch("Test Perplexity", dVal);
DocumentResults(accumulatingHistogram, recorder);
_logWriter.Dispose();
_outputStream.Dispose();
RILogManager.Default?.SendDebug("Log contents");
RILogManager.Default?.SendDebug(File.ReadAllText(LogPath));
Console.WriteLine();
RILogManager.Default?.SendDebug("Percentile distribution (values reported in milliseconds)");
accumulatingHistogram.OutputPercentileDistribution(Console.Out,
outputValueUnitScalingRatio: OutputScalingFactor.TimeStampToMilliseconds, useCsvFormat:
true);

RILogManager.Default?.SendDebug("Mean: " + Helpers.BytesToString(accumulatingHistogram.
GetMean()) + ", StdDev: " +
Helpers.BytesToString(accumulatingHistogram.GetStdDeviation()));
RILogManager.Default?.ViewerSendWatch("Mean", Helpers.BytesToString(accumulatingHistogram.
GetMean()));
RILogManager.Default?.ViewerSendWatch("StdDev", Helpers.BytesToString(accumulatingHistogram.
GetStdDeviation()));
}

static void DocumentResults(LongHistogram accumulatingHistogram, Recorder recorder)
{
```

```
recorder?.RecordValue(GC.GetTotalMemory(false));
RILogManager.Default?.ViewerSendWatch("Total Memory", GC.GetTotalMemory(false));
RILogManager.Default?.ViewerSendWatch($"Accumulated TotalCount", accumulatingHistogram.
TotalCount);
RILogManager.Default?.ViewerSendWatch("Mean", accumulatingHistogram.GetMean());
RILogManager.Default?.ViewerSendWatch("StdDev", accumulatingHistogram.GetStdDeviation());

var histogram = recorder?.GetIntervalHistogram();
accumulatingHistogram?.Add(histogram);
_logWriter?.Append(histogram);
RILogManager.Default?.SendDebug($"Accumulated.TotalCount = {accumulatingHistogram.
TotalCount,10:G}.");
RILogManager.Default?.SendDebug("Mean: " + accumulatingHistogram.GetMean() + ", StdDev:
" +
accumulatingHistogram.GetStdDeviation());
}

static double Evaluate(FunctionStack model, int[] dataset)
{
    FunctionStack predictModel = (FunctionStack)model.Clone();
    predictModel.ResetState();
    Real totalLoss = 0;
    long totalLossCount = 0;
    NdArray x = new NdArray(new[] { 1 }, BATCH_SIZE, (Function)null);
    NdArray t = new NdArray(new[] { 1 }, BATCH_SIZE, (Function)null);

    for (int i = 0; i < dataset.Length - 1; i++)
    {
        for (int j = 0; j < BATCH_SIZE; j++)
        {
            x.Data[j] = dataset[j + i];
            t.Data[j] = dataset[j + i + 1];
        }
        RILogManager.Default?.ViewerSendWatch("Validating Cross Entropy", i);
        Real sumLoss = new SoftmaxCrossEntropy().Evaluate(predictModel.Forward(true, x), t);
        totalLoss += sumLoss;
        totalLossCount++;
    }
}
```

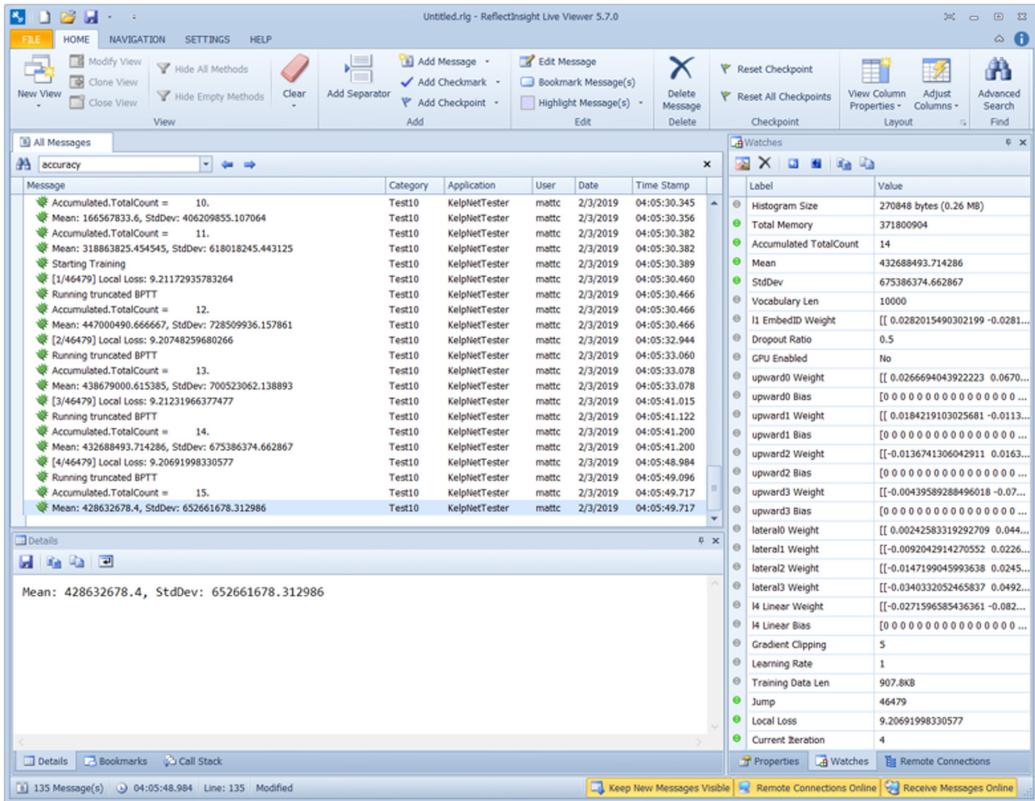
```

}

//calc perplexity
return Math.Exp(totalLoss / (totalLossCount - 1));
}

```

## Output



# Decoupled Neural Interfaces using Synthetic Gradients

This network model shows **Decoupled Neural Interfaces (DNI)** using **Synthetic Gradients** to learn **MNIST** (handwritten characters) by gradients. The network model that is created is fairly simple yet very powerful.

This test also takes a different approach to creating and using function stacks and functions. In this case, the functionStack object itself also becomes a function to another function stack. Thus, illustrating the immense power you have in how you create your models.

## Complete source code

The complete source code is shown below, including comments. You can find this source code in the Test11 test of Kelp.Net tests.

I will forewarn you; we are progressing away from smaller models and moving towards somewhat more complex ones. Take some time to see how this test begins to move in some exciting new directions:

```
const int BATCH_DATA_COUNT = 200;  
// Number of exercises per generation  
const int TRAIN_DATA_COUNT = 300; // = 60000 / 20  
// number of data at performance evaluation  
const int TEST_DATA_COUNT = 1000;  
  
public static void Run()  
{
```

Prepare the MNIST data:

```
RILogManager.Default?.SendDebug("MNIST Data Loading...");  
MnistData mnistData = new MnistData(28);  
RILogManager.Default?.SendDebug("Training Start...");
```

Write the network configuration in the FunctionStack:

```
FunctionStack Layer1 = new FunctionStack("Test11 Layer 1",  
    new Linear(28 * 28, 256, name: "I1 Linear"),  
    new BatchNormalization(256, name: "I1 Norm"),  
    new ReLU(name: "I1 ReLU")  
);
```

```
FunctionStack Layer2 = new FunctionStack("Test11 Layer 2",  
    new Linear(256, 256, name: "I2 Linear"),  
    new BatchNormalization(256, name: "I2 Norm"),  
    new ReLU(name: "I2 ReLU")  
);
```

```
FunctionStack Layer3 = new FunctionStack("Test11 Layer 3",
    new Linear(256, 256, name: "I3 Linear"),
    new BatchNormalization(256, name: "I3 Norm"),
    new ReLU(name: "I3 ReLU")
);
```

```
FunctionStack Layer4 = new FunctionStack("Test11 Layer 4",
    new Linear(256, 10, name: "I4 Linear")
);
```

The FunctionStack itself is also stacked as a function. This shows you how easy it is to access information in all neurons and layers of your network:

```
FunctionStack nn = new FunctionStack
("Test11",
Layer1,
Layer2,
Layer3,
Layer4
);
```

```
FunctionStack DNI1 = new FunctionStack("Test11 DNI1",
    new Linear(256, 1024, name: "DNI1 Linear1"),
    new BatchNormalization(1024, name: "DNI1 Nrom1"),
    new ReLU(name: "DNI1 ReLU1"),
    new Linear(1024, 1024, name: "DNI1 Linear2"),
    new BatchNormalization(1024, name: "DNI1 Nrom2"),
    new ReLU(name: "DNI1 ReLU2"),
    new Linear(1024, 256, initialW: new Real[1024, 256], name: "DNI1 Linear3")
);
```

```
FunctionStack DNI2 = new FunctionStack("Test11 DNI2",
    new Linear(256, 1024, name: "DNI2 Linear1"),
    new BatchNormalization(1024, name: "DNI2 Nrom1"),
    new ReLU(name: "DNI2 ReLU1"),
    new Linear(1024, 1024, name: "DNI2 Linear2"),
    new BatchNormalization(1024, name: "DNI2 Nrom2"),
    new ReLU(name: "DNI2 ReLU2"),
    new Linear(1024, 256, initialW: new Real[1024, 256], name: "DNI2 Linear3")
);
```

```
);  
  
FunctionStack DNI3 = new FunctionStack("Test11 DNI3",  
    new Linear(256, 1024, name: "DNI3 Linear1"),  
    new BatchNormalization(1024, name: "DNI3 Nrom1"),  
    new ReLU(name: "DNI3 ReLU1"),  
    new Linear(1024, 1024, name: "DNI3 Linear2"),  
    new BatchNormalization(1024, name: "DNI3 Nrom2"),  
    new ReLU(name: "DNI3 ReLU2"),  
    new Linear(1024, 256, initialW: new Real[1024, 256], name: "DNI3 Linear3")  
);
```

Set up the optimizers. In the instrumentation provided for this sample, you will see the difference between Adam and AdaXXX processing (Adam takes considerably longer):

```
Layer1.SetOptimizer(new Adam());  
Layer2.SetOptimizer(new Adam());  
Layer3.SetOptimizer(new Adam());  
Layer4.SetOptimizer(new Adam());  
DNI1.SetOptimizer(new Adam());  
DNI2.SetOptimizer(new Adam());  
DNI3.SetOptimizer(new Adam());
```

Let us learn:

```
for (int epoch = 0; epoch < 20; epoch++)  
{  
    Real totalLoss = 0;  
    Real DNI1totalLoss = 0;  
    Real DNI2totalLoss = 0;  
    Real DNI3totalLoss = 0;  
    long totalLossCount = 0;  
    long DNI1totalLossCount = 0;  
    long DNI2totalLossCount = 0;  
    long DNI3totalLossCount = 0;
```

Iterate over the batch:

```
for (int i = 1; i < TRAIN_DATA_COUNT + 1; i++)  
{
```

Get the data randomly from the training data:

```
TestDataSet datasetX = mnistData.GetRandomXSet(BATCH_DATA_COUNT, 28, 28);
```

Run the first layer with forward propagation:

```
NdArray[] layer1ForwardResult = Layer1.Forward(datasetX.Data);
```

Obtain the slope of the first layer:

```
NdArray[] DNI1Result = DNI1.Forward(layer1ForwardResult);
```

Apply the slope of the first layer:

```
layer1ForwardResult[0].Grad = DNI1Result[0].Data.ToArray();  
//Backwards propagation  
Layer1.Backward(layer1ForwardResult);  
layer1ForwardResult[0].ParentFunc = null; // Backward was executed and cut off calculation  
graph  
Layer1.Update();
```

//Layer 2

```
NdArray[] layer2ForwardResult = Layer2.Forward(layer1ForwardResult);
```

Get the inclination of the second layer:

```
NdArray[] DNI2Result = DNI2.Forward(layer2ForwardResult);
```

Apply the slope:

```
layer2ForwardResult[0].Grad = DNI2Result[0].Data.ToArray();  
//Backwards propagation  
Layer2.Backward(layer2ForwardResult);  
layer2ForwardResult[0].ParentFunc = null;
```

Run DNI learning for the first layer:

```
Real DNI1loss = new MeanSquaredError().Evaluate(DNI1Result, new  
NdArray(layer1ForwardResult[0].Grad, DNI1Result[0].Shape, DNI1Result[0].BatchCount));  
  
Layer2.Update();  
DNI1.Backward(DNI1Result);  
DNI1.Update();
```

```
DNI1totalLoss += DNI1loss;  
DNI1totalLossCount++;
```

Process the third layer:

```
NdArray[] layer3ForwardResult = Layer3.Forward(layer2ForwardResult);  
NdArray[] DNI3Result = DNI3.Forward(layer3ForwardResult);  
layer3ForwardResult[0].Grad = DNI3Result[0].Data.ToArray();  
Layer3.Backward(layer3ForwardResult);  
layer3ForwardResult[0].ParentFunc = null;
```

Run DNI learning for the second layer:

```
Real DNI2loss = new MeanSquaredError().Evaluate(DNI2Result, new  
NdArray(layer2ForwardResult[0].Grad, DNI2Result[0].Shape, DNI2Result[0].BatchCount));  
  
Layer3.Update();  
DNI2.Backward(DNI2Result);  
DNI2.Update();  
DNI2totalLoss += DNI2loss;  
DNI2totalLossCount++;
```

Process layer 4:

```
NdArray[] layer4ForwardResult = Layer4.Forward(layer3ForwardResult);  
Real sumLoss = new SoftmaxCrossEntropy().Evaluate(layer4ForwardResult, datasetX.Label);  
Layer4.Backward(layer4ForwardResult);  
layer4ForwardResult[0].ParentFunc = null;  
totalLoss += sumLoss;  
totalLossCount++;
```

Run DNI learning for layer 3:

```
Real DNI3loss = new MeanSquaredError().Evaluate(DNI3Result, new  
NdArray(layer3ForwardResult[0].Grad, DNI3Result[0].Shape, DNI3Result[0].BatchCount));  
  
Layer4.Update();  
DNI3.Backward(DNI3Result);  
DNI3.Update();  
DNI3totalLoss += DNI3loss;  
DNI3totalLossCount++;  
  
// Test the accuracy if you move the batch 20 times  
if (i % 20 == 0)  
{
```

```
RILogManager.Default?.SendDebug("Testing...");
```

Get random data:

```
TestDataSet datasetY = mnistData.GetRandomYSet(TEST_DATA_COUNT, 28);
```

Run the test and get the accuracy:

```
Real accuracy = Trainer.Accuracy(nn, datasetY.Data, datasetY.Label);
RILogManager.Default?.SendDebug("accuracy " + accuracy);
}
```

## Output

Message	Category	Machine	Application	User	Date	Time Stamp
NdArray Reducing/Slope Correction took 5.0099 ms	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	10:55:33.812
Adam Function Parameter Updating took 24.5039 ms	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	10:55:33.845
NdArray Reducing/Slope Correction took 0.0195 ms	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	10:55:33.845
Adam Function Parameter Updating took 0.0976 ms	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	10:55:33.845
NdArray Reducing/Slope Correction took 20.7082 ms	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	10:55:33.867
Adam Function Parameter Updating took 98.1855 ms	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	10:55:34.068
NdArray Reducing/Slope Correction took 0.0215 ms	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	10:55:34.184
Adam Function Parameter Updating took 0.1002 ms	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	10:55:34.184
NdArray Reducing/Slope Correction took 5.4376 ms	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	10:55:34.197
Adam Function Parameter Updating took 30.4017 ms	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	10:55:34.223
NdArray Reducing/Slope Correction took 0.0056 ms	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	10:55:34.223
Adam Function Parameter Updating took 0.0246 ms	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	10:55:34.223
batch count 8/300	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	10:55:34.223
total loss 2.4219902234754	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	10:55:34.223
local loss 2.33922170138672	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	10:55:34.223
DNI1 total loss 0	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	10:55:34.223
DNI2 total loss 0	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	10:55:34.223
DNI3 total loss 0.00380063590747217	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	10:55:34.223
DNI1 local loss 0	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	10:55:34.223
DNI2 local loss 0	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	10:55:34.224
DNI3 local loss 0.00365819891221584	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	10:55:34.224
Getting random X data (784) bytes	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	10:55:34.249
NdArray Reducing/Slope Correction took 3.8687 ms	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	10:55:46.434
Adam Function Parameter Updating took 26.3633 ms	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	10:55:46.476
NdArray Reducing/Slope Correction took 0.0128 ms	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	10:55:46.476
Adam Function Parameter Updating took 0.0663 ms	Common	MATTS-ASUS-LAPT	KelpNetTester	mattc	1/30/2019	10:55:46.476

## MNIST accuracy tester

This sample network model is designed around getting the optimal accuracy when training on MNIST digits. This is an excellent testbed for hyperparameter tuning, especially the optimizer and learning rate. We can specify different learning rates for our algorithm in order to see which one helps us to achieve optimal efficiency. Clearly, this is where the time will be spent when it comes to perfecting your test.

## Complete source code

The complete source code is shown below, including comments. You can find this source code in the Test21 test of Kelp.Net tests:

```
const int BATCH_DATA_COUNT = 20;
const int TRAIN_DATA_COUNT = 3000; // = 60000 / 20
const int TEST_DATA_COUNT = 200;
public static bool Passed = false;
// MNIST accuracy tester
public static void Run(double accuracyThreshold = .9979D)
{
    MnistData mnistData = new MnistData(28);
    Real maxAccuracy = 0;
    //Number of middle layers
    const int N = 30; //It operates at 1000 similar to the reference link but it is slow at the CPU
    ReflectInsight ri = new ReflectInsight("Test21");
    ri.Enabled = true;
    RILogManager.Add("Test21", "Test21");
    RILogManager.SetDefault("Test21");

    FunctionStack nn = new FunctionStack("Test7",
        new Linear(true, 28 * 28, N, name: "I1 Linear"), // L1
        new BatchNormalization(true, N, name: "I1 BatchNorm"),
        new ReLU(name: "I1 ReLU"),
        new Linear(true, N, N, name: "I2 Linear"), // L2
        new BatchNormalization(true, N, name: "I2 BatchNorm"),
        new ReLU(name: "I2 ReLU"),
        new Linear(true, N, N, name: "I3 Linear"), // L3
        new BatchNormalization(true, N, name: "I3 BatchNorm"),
        new ReLU(name: "I3 ReLU"),
        new Linear(true, N, N, name: "I4 Linear"), // L4
        new BatchNormalization(true, N, name: "I4 BatchNorm"),
        new ReLU(name: "I4 ReLU"),
        new Linear(true, N, N, name: "I5 Linear"), // L5
        new BatchNormalization(true, N, name: "I5 BatchNorm"),
        new ReLU(name: "I5 ReLU"),
        new Linear(true, N, N, name: "I6 Linear"), // L6
        new BatchNormalization(true, N, name: "I6 BatchNorm"),
```

```
new ReLU(name: "l6 ReLU"),
new Linear(true, N, N, name: "l7 Linear"), // L7
new BatchNormalization(true, N, name: "l7 BatchNorm"),
new ReLU(name: "l7 ReLU"),
new Linear(true, N, N, name: "l8 Linear"), // L8
new BatchNormalization(true, N, name: "l8 BatchNorm"),
new ReLU(name: "l8 ReLU"),
new Linear(true, N, N, name: "l9 Linear"), // L9
new BatchNormalization(true, N, name: "l9 BatchNorm"),
new ReLU(name: "l9 ReLU"),
new Linear(true, N, N, name: "l10 Linear"), // L10
new BatchNormalization(true, N, name: "l10 BatchNorm"),
new ReLU(name: "l10 ReLU"),
new Linear(true, N, N, name: "l11 Linear"), // L11
new BatchNormalization(true, N, name: "l11 BatchNorm"),
new ReLU(name: "l11 ReLU"),
new Linear(true, N, N, name: "l12 Linear"), // L12
new BatchNormalization(true, N, name: "l12 BatchNorm"),
new ReLU(name: "l12 ReLU"),
new Linear(true, N, N, name: "l13 Linear"), // L13
new BatchNormalization(true, N, name: "l13 BatchNorm"),
new ReLU(name: "l13 ReLU"),
new Linear(true, N, N, name: "l14 Linear"), // L14
new BatchNormalization(true, N, name: "l14 BatchNorm"),
new ReLU(name: "l14 ReLU"),
new Linear(true, N, 10, name: "l15 Linear") // L15
);

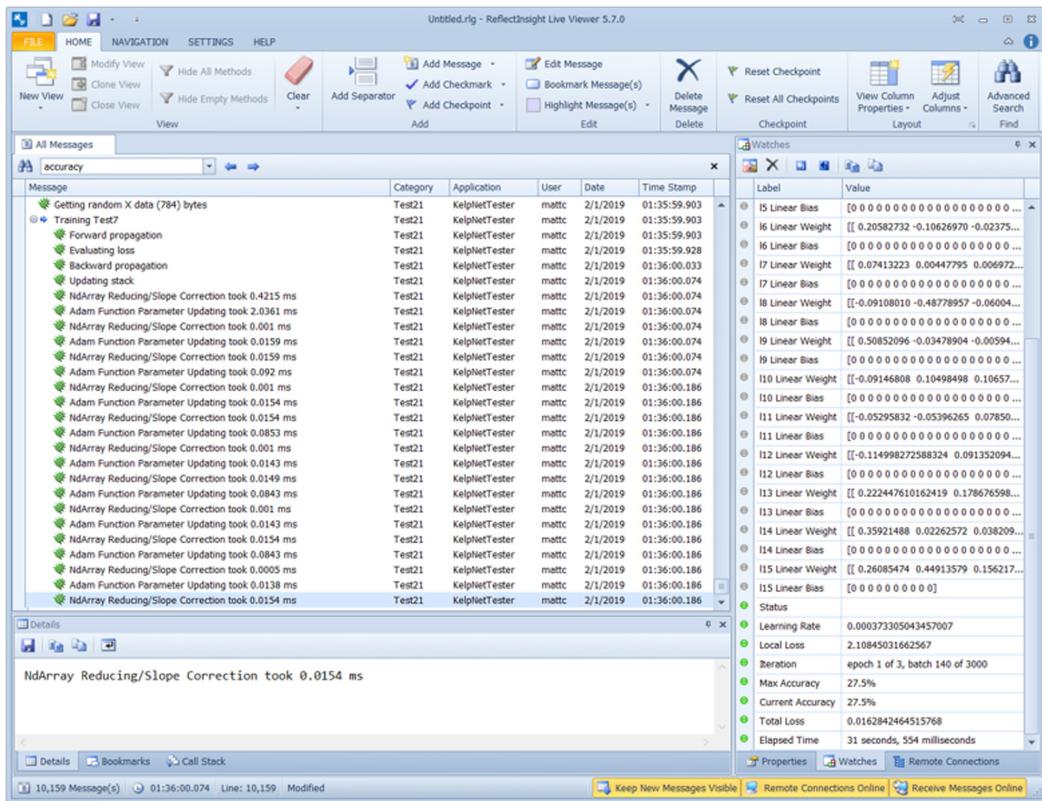
// 0.0005 - 97.5, 0.001, 0.00146
double alpha = 0.001;
double beta1 = 0.9D;
double beta2 = 0.999D;
double epsilon = 1e-8;
nn.SetOptimizer(new Adam("Adam21", alpha, beta1, beta2, epsilon));
Stopwatch sw = new Stopwatch();
sw.Start();

for (int epoch = 0; epoch < 5; epoch++)
```

```
{  
    Real totalLoss = 0;  
    long totalLossCount = 0;  
  
    for (int i = 1; i < TRAIN_DATA_COUNT + 1; i++)  
    {  
        TestDataSet datasetX = mnistData.GetRandomXSet(BATCH_DATA_COUNT, 28, 28);  
        Real sumLoss = Trainer.Train(nn, datasetX.Data, datasetX.Label, new SoftmaxCrossEntropy());  
        totalLoss = sumLoss;  
        totalLossCount++;  
  
        if (i % 20 == 0)  
        {  
            TestDataSet datasetY = mnistData.GetRandomYSet(TEST_DATA_COUNT, 28);  
            Real accuracy = Trainer.Accuracy(nn, datasetY.Data, datasetY.Label, false);  
            if (accuracy > maxAccuracy)  
                maxAccuracy = accuracy;  
            Passed = (accuracy >= accuracyThreshold);  
            sw.Stop();  
            ri.ViewerSendWatch("Iteration", "epoch " + (epoch + 1) + " of 3, batch " + i + " of " + TRAIN_  
DATA_COUNT);  
            ri.ViewerSendWatch("Max Accuracy", maxAccuracy * 100 + "%");  
            ri.ViewerSendWatch("Current Accuracy", accuracy * 100 + "%");  
            ri.ViewerSendWatch("Total Loss ", totalLoss / totalLossCount);  
            ri.ViewerSendWatch("Elapsed Time", Helpers.FormatTimeSpan(sw.Elapsed));  
            ri.ViewerSendWatch("Accuracy Threshold", Passed ? "Passed" : "Not Passed");  
            sw.Start();  
        }  
    }  
    sw.Stop();  
    ri.SendInformation("Total Processing Time: " + Helpers.FormatTimeSpan(sw.Elapsed));  
}
```

## Output

One of the things you will notice in this test is the increased usage of Watches in ReflectInsight. We do this so that we can log our individual weights and biases and observe them in real-time. Weights can be updated and biases should never change. We can make sure this holds true by monitoring the various weights and biases by levels in the deep learning model (the level is indicated by `lx` which precedes the bias and names of weights in each watch).



## Massively Deep Network Test

In this test, you will see how easy it is to create a massively scalable deep network with 1,000 layers. The goal of this test is to show you how easy it is to accomplish such a task with Kelp.Net, moreso than perfecting the hyperparameters which would be required for optimal efficiency. Please note that this test shows you what you could do. Bigger is not always better when it comes to deep learning, so determining the correct number of hidden layers is something you will need to do. 1,000 hidden layers do not mean things will work 1,000 times faster!

## Complete source code

The following is the complete source code, including comments. You can find this source code in the Test20 test of Kelp.Net tests:

```
int neuronCount = 28;
RILogManager.Add("Test20", "Test20");
RILogManager.SetDefault("Test20");
RILogManager.Default?.ViewerClearAll();
RILogManager.Default?.SendProcessInformation();
RILogManager.Default?.SendDebug("MNIST Data Loading...");
MnistData mnistData = new MnistData(neuronCount);
RILogManager.Default?.SendInformation("Training Start, creating function stack.");
SortedFunctionStack nn = new SortedFunctionStack();
SortedList<Function> functions = new SortedList<Function>();
ParallelOptions po = new ParallelOptions();
po.MaxDegreeOfParallelism = 4;

for (int x=0; x< numLayers; x++)
{
    functions.Add(new Linear(false, neuronCount * neuronCount, N, name: $"l{x} Linear"));
    functions.Add(new BatchNormalization(false, N, name: $"l{x} BatchNorm"));
    functions.Add(new ReLU(name: $"l{x} ReLU"));
    RILogManager.Default?.ViewerSendWatch("Total Layers", (x + 1));
};

RILogManager.Default?.SendInformation("Adding Output Layer");
nn.Add(new Linear(false, N, 10, noBias: false, name: $"l{numLayers + 1} Linear"));
RILogManager.Default?.ViewerSendWatch("Total Layers", numLayers);
RILogManager.Default?.SendInformation("Setting Optimizer to AdaGrad");
nn.SetOptimizer(new AdaGrad());
RunningStatistics stats = new RunningStatistics();
Real totalLoss = 0;
long totalLossCounter = 0;
Real highestAccuracy = 0;
Real bestLocalLoss = 0;
Real bestTotalLoss = 0;
LossFunction lossFunction = new SoftmaxCrossEntropy();
```

```
for (int epoch = 0; epoch < 3; epoch++)
{
    RILogManager.Default.ViewerSendWatch("epoch", (epoch + 1) + " of 3")

    for (int i = 1; i < TRAIN_DATA_COUNT + 1; i++)
    {
        DataSet datasetX = mnistData.GetRandomXSet(BATCH_DATA_COUNT,neuronCount,
neuronCount, false);
        Real sumLoss = Trainer.Train(nn, datasetX.Data, datasetX.Label, lossFunction, true, false);
        totalLoss += sumLoss;
        totalLossCounter++;
        stats.Push(sumLoss);
        if (sumLoss < bestLocalLoss && !double.IsNaN(sumLoss))
            bestLocalLoss = sumLoss;
        if (stats.Mean < bestTotalLoss && !double.IsNaN(sumLoss))
            bestTotalLoss = stats.Mean;
        if (i % 20 == 0)
        {
            RILogManager.Default?.ViewerSendWatch("Batch Count", i);
            RILogManager.Default?.ViewerSendWatch("Total/Mean loss", stats.Mean);
            RILogManager.Default?.ViewerSendWatch("Local loss", sumLoss);
            RILogManager.Default?.ViewerSendWatch("Status", "Testing");
            DataSet datasetY = mnistData.GetRandomYSet(TEST_DATA_COUNT, 28, false);
            Real accuracy = Trainer.Accuracy(nn, datasetY?.Data, datasetY.Label, false);
            if (accuracy > highestAccuracy)
                highestAccuracy = accuracy;
            RILogManager.Default?.ViewerSendWatch("Accuracy", accuracy);
            RILogManager.Default?.ViewerSendWatch("Best Accuracy", highestAccuracy);
            RILogManager.Default?.ViewerSendWatch("Best Total Loss", bestTotalLoss);
            RILogManager.Default?.ViewerSendWatch("Best Local Loss", bestLocalLoss);
        }
    }
}

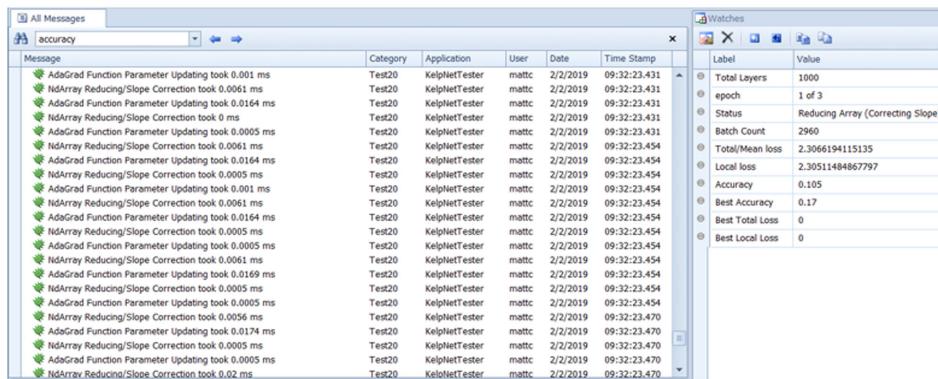
RILogManager.Default?.ViewerSendWatch("Status", "Testing");
DataSet ds = mnistData.GetRandomYSet(TEST_DATA_COUNT, 28, false);
Real acc = Trainer.Accuracy(nn, ds?.Data, ds.Label, false);
if (acc > highestAccuracy)
```

```

highestAccuracy = acc;
ModelIO.Save(nn, Application.StartupPath + "\\test20.nn");
RILogManager.Default?.ViewerSendWatch("Best Accuracy", highestAccuracy);
RILogManager.Default?.ViewerSendWatch("Best Total Loss", bestTotalLoss);
RILogManager.Default?.ViewerSendWatch("Best Local Loss", bestLocalLoss);
RILogManager.Default?.ViewerSendWatch("Status", "Complete");
}

```

## Output



## Image prediction test

In this test, we will create a model that can analyze an image and predict what it is.

### Example image

Our example image is that of a Boston terrier. It is very common when predicting between cats and dogs as this image can cause a lot of trouble due to the various features of the dog that resemble some cats. So we will see how good our image detection is.



## Complete source code

The complete source code is shown below, including comments. You can find this source code in the Test15 test of Kelp.Net tests:

```
private const string DOWNLOAD_URL = "http://www.robots.ox.ac.uk/~vgg/software/very_deep/caffe/VGG_ILSVRC_16_layers.caffemodel";  
private const string MODEL_FILE = "VGG_ILSVRC_16_layers.caffemodel";  
private const string CLASS_LIST_PATH = "Data/synset_words.txt";  
  
public static void Run()  
{  
    RILogManager.Add("Test15", "Test15");  
    RILogManager.SetDefault("Test15");  
    OpenFileDialog ofd = new OpenFileDialog { Filter = "Image Files(*.jpg;*.png;*.gif;*.bmp)| *.jpg;*.png;*.gif;*.bmp| All Files(*.*)| *.*" };  
  
    if (ofd.ShowDialog() == DialogResult.OK)  
    {  
        RILogManager.Default?.SendDebug("Model Loading.");  
        string modelFilePath = InternetFileDownloader.Download(DOWNLOAD_URL, MODEL_FILE);  
        List<Function> vgg16Net = CaffemodelDataLoader.ModelLoad(true, modelFilePath);  
        string[] classList = File.ReadAllLines(CLASS_LIST_PATH);  
  
        // Initialize the GPU  
        for (int i = 0; i < vgg16Net.Count - 1; i++)  
        {  
            if (vgg16Net[i] is Convolution2D || vgg16Net[i] is Linear || vgg16Net[i] is MaxPooling)  
            {  
                ((IParallelizable) vgg16Net[i]).SetGpuEnable(true);  
            }  
        }  
  
        FunctionStack nn = new FunctionStack(vgg16Net.ToArray());  
        // compress layer  
        RILogManager.Default?.SendDebug("Compressing Layer.");  
        nn.Compress();  
        RILogManager.Default?.SendDebug("Model Loading done.");
```

```
do
{
    RILogManager.Default?.SendDebug("Converting image resolution to 224x24x3");
    // Set the resolution to 224px x 224px x 3ch before entering the network
    Bitmap baseImage = new Bitmap(ofd.FileName);
    Bitmap resultImage = new Bitmap(224, 224, PixelFormat.Format24bppRgb);
    Graphics g = Graphics.FromImage(resultImage);
    g.DrawImage(baseImage, 0, 0, 224, 224);
    g.Dispose();
    Real[] bias = { -123.68, -116.779, -103.939 }; // The channel order of the correction value
    follows the input image
    NdArray imageArray = NdArrayConverter.Image2NdArray(resultImage, false, true, bias);
    RILogManager.Default?.SendDebug("Starting prediction...");
    Stopwatch sw = Stopwatch.StartNew();
    NdArray result = nn.Predict(true, imageArray)[0];
    sw.Stop();
    RILogManager.Default?.ViewerSendWatch("Result Time", (sw.ElapsedTicks / (Stopwatch.
    Frequency / (1000L * 1000L))).ToString("n0") + "μ s");
    RILogManager.Default?.SendDebug("Result Time : " +
    (sw.ElapsedTicks / (Stopwatch.Frequency / (1000L * 1000L))).ToString("n0") + "μ s");
    int maxIndex = Array.IndexOf(result.Data, result.Data.Max());
    RILogManager.Default?.SendDebug("[ " + result.Data[maxIndex] + " ] : " + classList[maxIndex]);
    RILogManager.Default?.ViewerSendWatch("Recognizer", "[ " + result.Data[maxIndex] + " ] : " +
    classList[maxIndex]);
}
while (ofd.ShowDialog() == DialogResult.OK);
}
```

## Output

Fortunately for us, our image detection algorithm was able to detect that the picture was a Boston terrier dog. You should try using other images to see just how accurate the detection is. First, try identifiable images such as cars, dogs, and more so that you can verify that your algorithm is working. Then, try more complicated images which are much harder to detect and correctly identify to make sure that your image detector works across as wide as a spectrum as possible.

Message	Category	Application	User	Date	Time Stamp
SetActivation -> Replacing /*ForwardActivate*/ with localResult = Forwa... Test15	Test15	KelpNetTester	mattc	2/2/2019	17:03:24.073
New KernelString is __kernel void Convolution2DForward(	Test15	KelpNetTester	mattc	2/2/2019	17:03:24.073
SetActivation -> Replacing /*ForwardActivate*/ with localResult = Forwa... Test15	Test15	KelpNetTester	mattc	2/2/2019	17:03:24.073
New KernelString is __kernel void Convolution2DForward(	Test15	KelpNetTester	mattc	2/2/2019	17:03:24.073
SetActivation -> Replacing /*ForwardActivate*/ with localResult = Forwa... Test15	Test15	KelpNetTester	mattc	2/2/2019	17:03:24.073
New KernelString is __kernel void Convolution2DForward(	Test15	KelpNetTester	mattc	2/2/2019	17:03:24.073
SetActivation -> Replacing /*ForwardActivate*/ with localResult = Forwa... Test15	Test15	KelpNetTester	mattc	2/2/2019	17:03:24.073
New KernelString is __kernel void Convolution2DForward(	Test15	KelpNetTester	mattc	2/2/2019	17:03:24.073
SetActivation -> Replacing /*ForwardActivate*/ with localResult = Forwa... Test15	Test15	KelpNetTester	mattc	2/2/2019	17:03:24.073
New KernelString is __kernel void Convolution2DForward(	Test15	KelpNetTester	mattc	2/2/2019	17:03:24.073
SetActivation -> Replacing /*ForwardActivate*/ with localResult = Forwa... Test15	Test15	KelpNetTester	mattc	2/2/2019	17:03:24.073
New KernelString is __kernel void Convolution2DForward(	Test15	KelpNetTester	mattc	2/2/2019	17:03:24.073
SetActivation -> Replacing /*ForwardActivate*/ with localResult = Forwa... Test15	Test15	KelpNetTester	mattc	2/2/2019	17:03:24.073
New KernelString is __kernel void Convolution2DForward(	Test15	KelpNetTester	mattc	2/2/2019	17:03:24.073
SetActivation -> Replacing /*ForwardActivate*/ with localResult = Forwa... Test15	Test15	KelpNetTester	mattc	2/2/2019	17:03:24.073
New KernelString is __kernel void Convolution2DForward(	Test15	KelpNetTester	mattc	2/2/2019	17:03:24.073
SetActivation -> Replacing /*ForwardActivate*/ with gpuYSum = Forwar... Test15	Test15	KelpNetTester	mattc	2/2/2019	17:03:24.073
New KernelString is __kernel void LinearForward(	Test15	KelpNetTester	mattc	2/2/2019	17:03:24.074
SetActivation -> Replacing /*ForwardActivate*/ with gpuYSum = Forwar... Test15	Test15	KelpNetTester	mattc	2/2/2019	17:03:24.074
New KernelString is __kernel void LinearForward(	Test15	KelpNetTester	mattc	2/2/2019	17:03:24.074
Model Loading done.	Test15	KelpNetTester	mattc	2/2/2019	17:03:24.074
Converting image resolution to 224x24x3	Test15	KelpNetTester	mattc	2/2/2019	17:03:24.074
Starting prediction...	Test15	KelpNetTester	mattc	2/2/2019	17:03:24.512
Result Time : 1,138,929,190µs	Test15	KelpNetTester	mattc	2/2/2019	17:13:10.293
[0.516563665290234] : n02096585 Boston bull, Boston terrier	Test15	KelpNetTester	mattc	2/2/2019	17:13:10.335

Details

[0.516563665290234] : n02096585 Boston bull, Boston terrier

## Function benchmarking

Function benchmarking is a model test that deals specifically with running many of the internal functions available to you. It is a one-stop shop to gain all your information on how the Kelp.Net internal functions perform. It can also then be used to help you notice when a change has negative performance results.

## Complete source code

The complete source code is shown below, including comments. You can find this source code in the SingleBenchmark test of Kelp.Net tests:

```
public static void Run(bool verbose)
{
    Stopwatch sw = new Stopwatch();
    NdArray inputArrayCpu = new NdArray(BenchDataMaker.GetRealArray(INPUT_SIZE));
    NdArray inputArrayGpu = new NdArray(BenchDataMaker.GetRealArray(INPUT_SIZE));
    Ensure.Argument(inputArrayGpu).NotNull();
    Ensure.Argument(inputArrayCpu).NotNull();
    RILogManager.Add("SingleBenchmark", "Benchmarking");
    RILogManager.SetDefault("SingleBenchmark");
```

```
//Linear
Linear linear = new Linear(verbose, INPUT_SIZE, OUTPUT_SIZE);
if (verbose)
    RILogManager.Default?.EnterMethod(linear.Name);
sw.Restart();
NdArray[] gradArrayCpu = linear.Forward(verbose, inputArrayCpu);
sw.Stop();
if (verbose)
    RILogManager.Default?.SendDebug("Forward [Cpu] : " + (sw.ElapsedTicks / (Stopwatch.
Frequency / (1000L * 1000L))).ToString("n0") + "μ s");

Ensure.Argument(gradArrayCpu).NotNull();
gradArrayCpu[0].Grad = gradArrayCpu[0].Data; // Use Data as Grad
sw.Restart();
linear.Backward(verbose, gradArrayCpu);
sw.Stop();
if (verbose)
    RILogManager.Default?.SendDebug("Backward[Cpu] : " + (sw.ElapsedTicks / (Stopwatch.
Frequency / (1000L * 1000L))).ToString("n0") + "μ s");

if (linear.SetGpuEnable(true))
{
    sw.Restart();
    NdArray[] gradArrayGpu = linear.Forward(verbose, inputArrayGpu);
    sw.Stop();
    if (verbose)
        RILogManager.Default?.SendDebug("Forward [Gpu] : " + (sw.ElapsedTicks / (Stopwatch.
Frequency / (1000L * 1000L))).ToString("n0") + "μ s");

    gradArrayGpu[0].Grad = gradArrayGpu[0].Data;
    sw.Restart();
    linear.Backward(verbose, gradArrayGpu);
    sw.Stop();
    if (verbose)
        RILogManager.Default?.SendDebug("Backward[Gpu] : " + (sw.ElapsedTicks / (Stopwatch.
Frequency / (1000L * 1000L))).ToString("n0") + "μ s");
}
```

```
RILogManager.Default?.ExitMethod(linear.Name);

//Tanh
Tanh tanh = new Tanh();
if (verbose)
    RILogManager.Default?.EnterMethod(tanh.Name);
sw.Restart();
gradArrayCpu = tanh.Forward(verbose, inputArrayCpu);
sw.Stop();
if (verbose)
    RILogManager.Default?.SendDebug("Forward [Cpu] : " + (sw.ElapsedTicks / (Stopwatch.
Frequency / (1000L * 1000L))).ToString("n0") + "μ s ");

gradArrayCpu[0].Grad = gradArrayCpu[0].Data;
sw.Restart();
tanh.Backward(verbose, gradArrayCpu);
sw.Stop();
if (verbose)
    RILogManager.Default?.SendDebug("Backward[Cpu] : " + (sw.ElapsedTicks / (Stopwatch.
Frequency / (1000L * 1000L))).ToString("n0") + "μ s ");

if (tanh.SetGpuEnable(true))
    HandleGPU(verbose, sw, tanh, inputArrayGpu);
if (verbose)
    RILogManager.Default?.ExitMethod(tanh.Name);

//Sigmoid
Sigmoid sigmoid = new Sigmoid();
if (verbose)
    RILogManager.Default?.EnterMethod(sigmoid.Name);
sw.Restart();
gradArrayCpu = sigmoid.Forward(verbose, inputArrayCpu);
sw.Stop();
if (verbose)
    RILogManager.Default?.SendDebug("Forward [Cpu] : " + (sw.ElapsedTicks / (Stopwatch.
Frequency / (1000L * 1000L))).ToString("n0") + "μ s ");

gradArrayCpu[0].Grad = gradArrayCpu[0].Data;
```

```
sw.Restart();
sigmoid.Backward(verbose, gradArrayCpu);
sw.Stop();
if (verbose)
    RILogManager.Default?.SendDebug("Backward[Cpu] : " + (sw.ElapsedTicks / (Stopwatch.
Frequency / (1000L * 1000L))).ToString("n0") + "μ s");

if (sigmoid.SetGpuEnable(true))
    HandleGPU(verbose, sw, sigmoid, inputArrayGpu);
if (verbose)
    RILogManager.Default?.ExitMethod(tanh.Name);

//Softmax
Softmax sm = new Softmax();
RILogManager.Default?.EnterMethod(sm.Name);
sw.Restart();
gradArrayCpu = sm.Forward(verbose, inputArrayCpu);
sw.Stop();
RILogManager.Default?.SendDebug("Forward [Cpu] : " + (sw.ElapsedTicks / (Stopwatch.
Frequency / (1000L * 1000L))).ToString("n0") + "μ s");

gradArrayCpu[0].Grad = gradArrayCpu[0].Data;
sw.Restart();
sm.Backward(verbose, gradArrayCpu);
sw.Stop();
if (verbose)
    RILogManager.Default?.SendDebug("Backward[Cpu] : " + (sw.ElapsedTicks / (Stopwatch.
Frequency / (1000L * 1000L))).ToString("n0") + "μ s");
if (verbose)
    RILogManager.Default?.ExitMethod(sm.Name);

//Softplus
Softplus sp = new Softplus();
if (verbose)
    RILogManager.Default?.EnterMethod(sp.Name);
sw.Restart();
gradArrayCpu = sp.Forward(verbose, inputArrayCpu);
sw.Stop();
```

```
RILogManager.Default?.SendDebug("Forward [Cpu] : " + (sw.ElapsedTicks / (Stopwatch.  
Frequency / (1000L * 1000L))).ToString("n0") + "μ s");  
  
gradArrayCpu[0].Grad = gradArrayCpu[0].Data;  
sw.Restart();  
sp.Backward(verbose, gradArrayCpu);  
sw.Stop();  
RILogManager.Default?.SendDebug("Backward[Cpu] : " + (sw.ElapsedTicks / (Stopwatch.  
Frequency / (1000L * 1000L))).ToString("n0") + "μ s");  
RILogManager.Default?.ExitMethod(sp.Name);  
  
//ReLU  
ReLU relu = new ReLU();  
RILogManager.Default?.EnterMethod(relu.Name);  
sw.Restart();  
gradArrayCpu = relu.Forward(verbose, inputArrayCpu);  
sw.Stop();  
RILogManager.Default?.SendDebug("Forward [Cpu] : " + (sw.ElapsedTicks / (Stopwatch.  
Frequency / (1000L * 1000L))).ToString("n0") + "μ s");  
  
gradArrayCpu[0].Grad = gradArrayCpu[0].Data;  
sw.Restart();  
relu.Backward(verbose, gradArrayCpu);  
sw.Stop();  
RILogManager.Default?.SendDebug("Backward[Cpu] : " + (sw.ElapsedTicks / (Stopwatch.  
Frequency / (1000L * 1000L))).ToString("n0") + "μ s");  
  
if (relu.SetGpuEnable(true))  
    HandleGPU(verbose, sw, relu, inputArrayGpu);  
RILogManager.Default?.ExitMethod(relu.Name);  
  
//LeakyReLU  
LeakyReLU leakyRelu = new LeakyReLU();  
RILogManager.Default?.EnterMethod(leakyRelu.Name);  
sw.Restart();  
gradArrayCpu = leakyRelu.Forward(verbose, inputArrayCpu);  
sw.Stop();  
RILogManager.Default?.SendDebug("Forward [Cpu] : " + (sw.ElapsedTicks / (Stopwatch.  
Frequency / (1000L * 1000L))).ToString("n0") + "μ s");
```

```
gradArrayCpu[0].Grad = gradArrayCpu[0].Data;
sw.Restart();
leakyRelu.Backward(verbose, gradArrayCpu);
sw.Stop();
RILogManager.Default?.SendDebug("Backward[Cpu] : " + (sw.ElapsedTicks / (Stopwatch.
Frequency / (1000L * 1000L))).ToString("n0") + "μ s");

if (leakyRelu.SetGpuEnable(true))
    HandleGPU(verbose, sw, leakyRelu, inputArrayGpu);
RILogManager.Default?.ExitMethod(leakyRelu.Name);

//ReLU
ReLU rth = new ReLU();
RILogManager.Default?.EnterMethod(rth.Name);
sw.Restart();
gradArrayCpu = rth.Forward(verbose, inputArrayCpu);
sw.Stop();
RILogManager.Default?.SendDebug("Forward [Cpu] : " + (sw.ElapsedTicks / (Stopwatch.
Frequency / (1000L * 1000L))).ToString("n0") + "μ s");

gradArrayCpu[0].Grad = gradArrayCpu[0].Data;
sw.Restart();
rth.Backward(verbose, gradArrayCpu);
sw.Stop();
RILogManager.Default?.SendDebug("Backward[Cpu] : " + (sw.ElapsedTicks / (Stopwatch.
Frequency / (1000L * 1000L))).ToString("n0") + "μ s");

if (rth.SetGpuEnable(true))
    HandleGPU(verbose, sw, rth, inputArrayGpu);
RILogManager.Default?.ExitMethod(rth.Name);

////Swish
//Swish swi = new Swish();
//RILogManager.Default?.SendDebug(swi.Name);
//swi.Restart();
//gradArrayCpu = swi.Forward(inputArrayCpu);
//swi.Stop();
```

```
//RILogManager.Default?.SendDebug("Forward [Cpu] : " + (sw.ElapsedTicks / (Stopwatch.
Frequency / (1000L * 1000L))).ToString("n0") + "μ s ");

//gradArrayCpu[0].Grad = gradArrayCpu[0].Data;
//sw.Restart();
//swi.Backward(gradArrayCpu);
//sw.Stop();
//RILogManager.Default?.SendDebug("Backward[Cpu] : " + (sw.ElapsedTicks / (Stopwatch.
Frequency / (1000L * 1000L))).ToString("n0") + "μ s ");

NdArray inputImageArrayGpu = new NdArray(BenchDataMaker.GetRealArray(3 * 256 * 256 * 5),
new[] { 3, 256, 256 }, 5);
NdArray inputImageArrayCpu = new NdArray(BenchDataMaker.GetRealArray(3 * 256 * 256 * 5),
new[] { 3, 256, 256 }, 5);

//MaxPooling
MaxPooling maxPooling = new MaxPooling(3);
RILogManager.Default?.EnterMethod(maxPooling.Name);
sw.Restart();
NdArray[] gradImageArrayCpu = maxPooling.Forward(verbose, inputImageArrayCpu);
sw.Stop();
RILogManager.Default?.SendDebug("Forward [Cpu] : " + (sw.ElapsedTicks / (Stopwatch.
Frequency / (1000L * 1000L))).ToString("n0") + "μ s ");

gradImageArrayCpu[0].Grad = gradImageArrayCpu[0].Data;
sw.Restart();
maxPooling.Backward(verbose, gradImageArrayCpu);
sw.Stop();
RILogManager.Default?.SendDebug("Backward[Cpu] : " + (sw.ElapsedTicks / (Stopwatch.
Frequency / (1000L * 1000L))).ToString("n0") + "μ s ");

if (maxPooling.SetGpuEnable(true))
{
    sw.Restart();
    maxPooling.Forward(verbose, inputImageArrayGpu);
    sw.Stop();
    RILogManager.Default?.SendDebug("Forward [Gpu] : " + (sw.ElapsedTicks / (Stopwatch.
Frequency / (1000L * 1000L))).ToString("n0") + "μ s ");
```

```
// There is no implementation for memory transfer only
RILogManager.Default?.SendDebug("Backward[Gpu] : None");
}

RILogManager.Default?.ExitMethod(maxPooling.Name);

//AvgPooling
AveragePooling avgPooling = new AveragePooling(3);
RILogManager.Default?.EnterMethod(avgPooling.Name);
sw.Restart();
gradImageArrayCpu = avgPooling.Forward(verbose, inputImageArrayCpu);
sw.Stop();

RILogManager.Default?.SendDebug("Forward [Cpu] : " + (sw.ElapsedTicks / (Stopwatch.
Frequency / (1000L * 1000L))).ToString("n0") + "μ s");

gradImageArrayCpu[0].Grad = gradImageArrayCpu[0].Data;
sw.Restart();
avgPooling.Backward(verbose, gradImageArrayCpu);
sw.Stop();

RILogManager.Default?.SendDebug("Backward[Cpu] : " + (sw.ElapsedTicks / (Stopwatch.
Frequency / (1000L * 1000L))).ToString("n0") + "μ s");

RILogManager.Default?.ExitMethod(avgPooling.Name);

//Conv2D
Convolution2D conv2d = new Convolution2D(verbose, 3, 3, 3);
RILogManager.Default?.EnterMethod(conv2d.Name);
sw.Restart();
gradImageArrayCpu = conv2d.Forward(verbose, inputImageArrayCpu);
sw.Stop();

RILogManager.Default?.SendDebug("Forward [Cpu] : " + (sw.ElapsedTicks / (Stopwatch.
Frequency / (1000L * 1000L))).ToString("n0") + "μ s");

gradImageArrayCpu[0].Grad = gradImageArrayCpu[0].Data;
sw.Restart();
conv2d.Backward(verbose, gradImageArrayCpu);
sw.Stop();

RILogManager.Default?.SendDebug("Backward[Cpu] : " + (sw.ElapsedTicks / (Stopwatch.
Frequency / (1000L * 1000L))).ToString("n0") + "μ s");

if (conv2d.SetGpuEnable(true))
```

```
HandleGPU(verbose, sw, conv2d, inputArrayGpu);
RILogManager.Default?.ExitMethod(conv2d.Name);

//Deconv2D
Deconvolution2D deconv2d = new Deconvolution2D(verbose, 3, 3, 3);
RILogManager.Default?.EnterMethod(deconv2d.Name);
sw.Restart();
gradImageArrayCpu = deconv2d.Forward(verbose, inputImageArrayCpu);
sw.Stop();
RILogManager.Default?.SendDebug("Forward [Cpu] : " + (sw.ElapsedTicks / (Stopwatch.
Frequency / (1000L * 1000L))).ToString("n0") + "μ s");

gradImageArrayCpu[0].Grad = gradImageArrayCpu[0].Data;
sw.Restart();
deconv2d.Backward(verbose, gradImageArrayCpu);
sw.Stop();
RILogManager.Default?.SendDebug("Backward[Cpu] : " + (sw.ElapsedTicks / (Stopwatch.
Frequency / (1000L * 1000L))).ToString("n0") + "μ s");

if (deconv2d.SetGpuEnable(true))
    HandleGPU(verbose, sw, deconv2d, inputArrayGpu);
RILogManager.Default?.ExitMethod(deconv2d.Name);

//Dropout
Dropout dropout = new Dropout();
RILogManager.Default?.EnterMethod(dropout.Name);
sw.Restart();
gradArrayCpu = dropout.Forward(verbose, inputArrayCpu);
sw.Stop();
RILogManager.Default?.SendDebug("Forward [Cpu] : " + (sw.ElapsedTicks / (Stopwatch.
Frequency / (1000L * 1000L))).ToString("n0") + "μ s");

gradArrayCpu[0].Grad = gradArrayCpu[0].Data;
sw.Restart();
dropout.Backward(verbose, gradArrayCpu);
sw.Stop();
RILogManager.Default?.SendDebug("Backward[Cpu] : " + (sw.ElapsedTicks / (Stopwatch.
Frequency / (1000L * 1000L))).ToString("n0") + "μ s");
```

```
if (dropout.SetGpuEnable(true))
{
    sw.Restart();
    NdArray[] gradArrayGpu = dropout.Forward(verbose, inputArrayGpu);
    sw.Stop();
    RILogManager.Default?.SendDebug("Forward [Gpu] : " + (sw.ElapsedTicks / (Stopwatch.
Frequency / (1000L * 1000L))).ToString("n0") + "μ s ");

    gradArrayGpu[0].Grad = gradArrayGpu[0].Data;
    sw.Restart();
    dropout.Backward(verbose, gradArrayGpu);
    sw.Stop();
    RILogManager.Default?.SendDebug("Backward[Gpu] : " + (sw.ElapsedTicks / (Stopwatch.
Frequency / (1000L * 1000L))).ToString("n0") + "μ s ");
}

RILogManager.Default?.ExitMethod(dropout.Name);

//ArcSinH
ArcSinH a = new ArcSinH();
RILogManager.Default?.EnterMethod(a.Name);
sw.Restart();
gradArrayCpu = a.Forward(verbose, inputArrayCpu);
sw.Stop();
RILogManager.Default?.SendDebug("Forward [Cpu] : " + (sw.ElapsedTicks / (Stopwatch.
Frequency / (1000L * 1000L))).ToString("n0") + "μ s ");

gradArrayCpu[0].Grad = gradArrayCpu[0].Data;
sw.Restart();
a.Backward(verbose, gradArrayCpu);
sw.Stop();
RILogManager.Default?.SendDebug("Backward[Cpu] : " + (sw.ElapsedTicks / (Stopwatch.
Frequency / (1000L * 1000L))).ToString("n0") + "μ s ");

if (a.SetGpuEnable(true))
    HandleGPU(verbose, sw, a, inputArrayGpu);
RILogManager.Default?.ExitMethod(a.Name);

//ELU
ELU e = new ELU();
```

```
RILogManager.Default?.EnterMethod(e.Name);
sw.Restart();
gradArrayCpu = e.Forward(verbose, inputArrayCpu);
sw.Stop();
RILogManager.Default?.SendDebug("Forward [Cpu] : " + (sw.ElapsedTicks / (Stopwatch.
Frequency / (1000L * 1000L))).ToString("n0") + "μ s");

gradArrayCpu[0].Grad = gradArrayCpu[0].Data;
sw.Restart();
e.Backward(verbose, gradArrayCpu);
sw.Stop();
RILogManager.Default?.SendDebug("Backward[Cpu] : " + (sw.ElapsedTicks / (Stopwatch.
Frequency / (1000L * 1000L))).ToString("n0") + "μ s");
RILogManager.Default?.ExitMethod(e.Name);

//LeakyReluShifted
LeakyReLUShifted lrs = new LeakyReLUShifted();
RILogManager.Default?.EnterMethod(lrs.Name);
sw.Restart();
gradArrayCpu = lrs.Forward(verbose, inputArrayCpu);
sw.Stop();
RILogManager.Default?.SendDebug("Forward [Cpu] : " + (sw.ElapsedTicks / (Stopwatch.
Frequency / (1000L * 1000L))).ToString("n0") + "μ s");

gradArrayCpu[0].Grad = gradArrayCpu[0].Data;
sw.Restart();
lrs.Backward(verbose, gradArrayCpu);
sw.Stop();
RILogManager.Default?.SendDebug("Backward[Cpu] : " + (sw.ElapsedTicks / (Stopwatch.
Frequency / (1000L * 1000L))).ToString("n0") + "μ s");

if (lrs.SetGpuEnable(true))
    HandleGPU(verbose, sw, lrs, inputArrayGpu);
RILogManager.Default?.ExitMethod(lrs.Name);

//Logistic
LogisticFunction lf = new LogisticFunction();
RILogManager.Default?.EnterMethod(lf.Name);
sw.Restart();
```

```
gradArrayCpu = lf.Forward(verbose, inputArrayCpu);
sw.Stop();
RILogManager.Default?.SendDebug("Forward [Cpu] : " + (sw.ElapsedTicks / (Stopwatch.
Frequency / (1000L * 1000L))).ToString("n0") + "μ s");

gradArrayCpu[0].Grad = gradArrayCpu[0].Data;
sw.Restart();
lf.Backward(verbose, gradArrayCpu);
sw.Stop();
RILogManager.Default?.SendDebug("Backward[Cpu] : " + (sw.ElapsedTicks / (Stopwatch.
Frequency / (1000L * 1000L))).ToString("n0") + "μ s");

if (lf.SetGpuEnable(true))
    HandleGPU(verbose, sw, lf, inputArrayGpu);
RILogManager.Default?.ExitMethod(lf.Name);

//MaxMinusOne
MaxMinusOne mmo = new MaxMinusOne();
RILogManager.Default?.EnterMethod(mmo.Name);
sw.Restart();
gradArrayCpu = mmo.Forward(verbose, inputArrayCpu);
sw.Stop();
RILogManager.Default?.SendDebug("Forward [Cpu] : " + (sw.ElapsedTicks / (Stopwatch.
Frequency / (1000L * 1000L))).ToString("n0") + "μ s");

gradArrayCpu[0].Grad = gradArrayCpu[0].Data;
sw.Restart();
mmo.Backward(verbose, gradArrayCpu);
sw.Stop();
RILogManager.Default?.SendDebug("Backward[Cpu] : " + (sw.ElapsedTicks / (Stopwatch.
Frequency / (1000L * 1000L))).ToString("n0") + "μ s");

if (mmo.SetGpuEnable(true))
    HandleGPU(verbose, sw, mmo, inputArrayGpu);
RILogManager.Default?.ExitMethod(mmo.Name);

//ScaledELU
ScaledELU se = new ScaledELU();
```

```
RILogManager.Default?.EnterMethod(se.Name);
sw.Restart();
gradArrayCpu = se.Forward(verbose, inputArrayCpu);
sw.Stop();
RILogManager.Default?.SendDebug("Forward [Cpu] : " + (sw.ElapsedTicks / (Stopwatch.
Frequency / (1000L * 1000L))).ToString("n0") + "μ s");

gradArrayCpu[0].Grad = gradArrayCpu[0].Data;
sw.Restart();
se.Backward(verbose, gradArrayCpu);
sw.Stop();
RILogManager.Default?.SendDebug("Backward[Cpu] : " + (sw.ElapsedTicks / (Stopwatch.
Frequency / (1000L * 1000L))).ToString("n0") + "μ s");

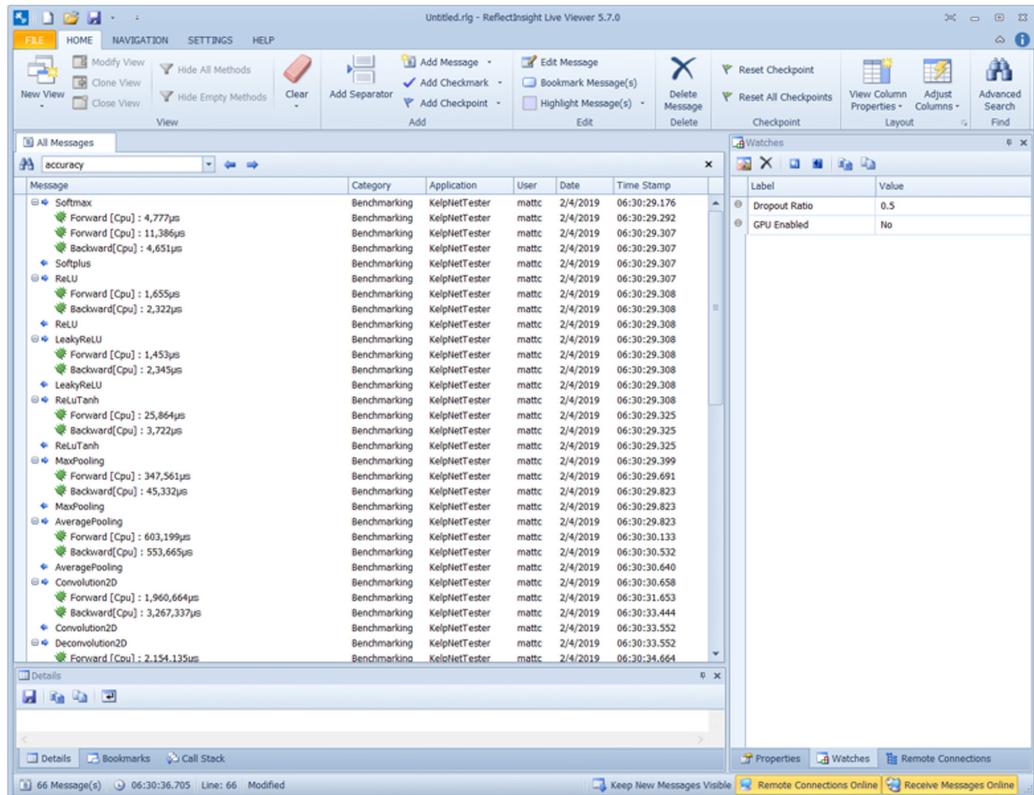
if (se.SetGpuEnable(true))
    HandleGPU(verbose, sw, se, inputArrayGpu);
RILogManager.Default?.ExitMethod(se.Name);

//Sine
Sine s = new Sine();
RILogManager.Default?.EnterMethod(s.Name);
sw.Restart();
gradArrayCpu = s.Forward(verbose, inputArrayCpu);
sw.Stop();
RILogManager.Default?.SendDebug("Forward [Cpu] : " + (sw.ElapsedTicks / (Stopwatch.
Frequency / (1000L * 1000L))).ToString("n0") + "μ s");

gradArrayCpu[0].Grad = gradArrayCpu[0].Data;
sw.Restart();
s.Backward(verbose, gradArrayCpu);
sw.Stop();
RILogManager.Default?.SendDebug("Backward[Cpu] : " + (sw.ElapsedTicks / (Stopwatch.
Frequency / (1000L * 1000L))).ToString("n0") + "μ s");

if (s.SetGpuEnable(true))
    HandleGPU(verbose, sw, s, inputArrayGpu);
RILogManager.Default?.ExitMethod(s.Name);
}
```

## Output



## MNIST (handwritten characters) learning test

This model tests the ability to use the MNIST handwritten character image sets and determine the accuracy by the network.

### Complete source code

The complete source code is shown below, including comments. You can find this source code in the Test4 test of Kelp.Net test:

```
const int BATCH_DATA_COUNT = 20;
const int TRAIN_DATA_COUNT = 3000; // = 60000 / 20
const int TEST_DATA_COUNT = 200;
```

```
public static void Run()
{
    RILogManager.Default?.SendDebug("MNIST Data Loading...");
    RILogManager.Default?.SendDebug("MNIST Data Loading...");
    MnistData mnistData = new MnistData(28);
    RILogManager.Default?.SendDebug("Training Start...");

    FunctionStack nn = new FunctionStack("Test4",
        new Linear(true, 28 * 28, 1024, name: "I1 Linear"),
        new Sigmoid(name: "I1 Sigmoid"),
        new Linear(true, 1024, 10, name: "I2 Linear")
    );
    nn.SetOptimizer(new MomentumSGD());

    for (int epoch = 0; epoch < 3; epoch++)
    {
        RILogManager.Default?.SendDebug("epoch " + (epoch + 1));
        Real totalLoss = 0;
        long totalLossCount = 0;

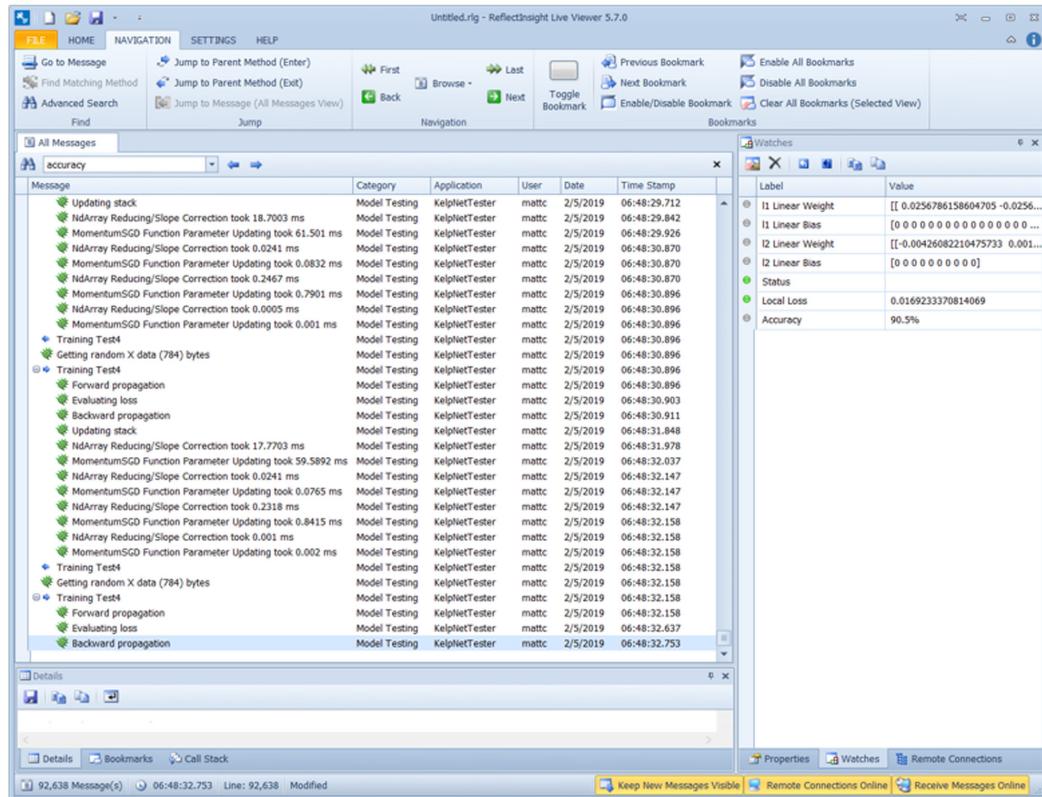
        for (int i = 1; i < TRAIN_DATA_COUNT + 1; i++)
        {
            //Get data randomly from training data
            DataSet datasetX = mnistData.GetRandomXSet(BATCH_DATA_COUNT, 28, 28);
            Real sumLoss = Trainer.Train(nn, datasetX.Data, datasetX.Label, new SoftmaxCrossEntropy());
            totalLoss = sumLoss;
            totalLossCount++;

            if (i % 20 == 0)
            {
                RILogManager.Default?.SendDebug("\nbatch count " + i + "/" + TRAIN_DATA_COUNT);
                RILogManager.Default?.SendDebug("total loss " + totalLoss / totalLossCount);
                RILogManager.Default?.SendDebug("local loss " + sumLoss);
                RILogManager.Default?.SendDebug("\nTesting...");

                //Get data randomly from test data
                DataSet datasetY = mnistData.GetRandomYSet(TEST_DATA_COUNT, 28);
                Real accuracy = Trainer.Accuracy(nn, datasetY.Data, datasetY.Label);
            }
        }
    }
}
```

```
RILogManager.Default?.SendDebug("accuracy " + accuracy);  
}  
}  
}  
}  
}
```

## Output



# LeakyReLu and PolynomialApproximantSteep Combination Network

This test will create a more complicated and thus a powerful neural network that shows you how to combine a `PolynomialApproximateSteep` function under `LeakyReLU`.

## Complete source code

The complete source code is shown below, including comments. You can find this source code in the Test19 test of Kelp.Net tests:

```
const int BATCH_DATA_COUNT = 128;
private const int TRAIN_DATA_COUNT = 100; //1000; //50000;
const int TEST_DATA_COUNT = 200;
private const int N = 60;
public static void Run()
{
    RILogManager.Default?.SendDebug("MNIST Data Loading...");
    MnistData mnistData = new MnistData(28);
    RILogManager.Default?.SendDebug("Training Start...");

    int neuronCount = 28;
    FunctionStack nn = new FunctionStack("Test19",
        new Linear(true, neuronCount * neuronCount, N, name: "I1 Linear"), // L1
        new BatchNormalization(true, N, name: "I1 BatchNorm"),
        new LeakyReLU(slope: 0.000001, name: "I1 LeakyReLU"),
        new Linear(true, N, N, name: "I2 Linear"), // L2
        new BatchNormalization(true, N, name: "I2 BatchNorm"),
        new LeakyReLU(slope: 0.000001, name: "I2 LeakyReLU"),
        new Linear(true, N, N, name: "I3 Linear"), // L3
        new BatchNormalization(true, N, name: "I3 BatchNorm"),
        new LeakyReLU(slope: 0.000001, name: "I3 LeakyReLU"),
        new Linear(true, N, N, name: "I4 Linear"), // L4
        new BatchNormalization(true, N, name: "I4 BatchNorm"),
        new LeakyReLU(slope: 0.000001, name: "I4 LeakyReLU"),
        new Linear(true, N, N, name: "I5 Linear"), // L5
        new BatchNormalization(true, N, name: "I5 BatchNorm"),
        new LeakyReLU(slope: 0.000001, name: "I5 LeakyReLU"),
        new Linear(true, N, N, name: "I6 Linear"), // L6
        new BatchNormalization(true, N, name: "I6 BatchNorm"),
        new LeakyReLU(slope: 0.000001, name: "I6 LeakyReLU"),
        new Linear(true, N, N, name: "I7 Linear"), // L7
        new BatchNormalization(true, N, name: "I7 BatchNorm"),
        new LeakyReLU(slope: 0.000001, name: "I7 ReLU"),
        new Linear(true, N, N, name: "I8 Linear"), // L8
```

```
new BatchNormalization(true, N, name: "l8 BatchNorm"),
new LeakyReLU(slope: 0.000001, name: "l8 LeakyReLU"),
new Linear(true, N, N, name: "l9 Linear"), // L9
new BatchNormalization(true, N, name: "l9 BatchNorm"),
new PolynomialApproximantSteep(slope: 0.000001, name: "l9 PolynomialApproximantSteep"),
new Linear(true, N, N, name: "l10 Linear"), // L10
new BatchNormalization(true, N, name: "l10 BatchNorm"),
new PolynomialApproximantSteep(slope: 0.000001, name: "l10
PolynomialApproximantSteep"),
new Linear(true, N, N, name: "l11 Linear"), // L11
new BatchNormalization(true, N, name: "l11 BatchNorm"),
new PolynomialApproximantSteep(slope: 0.000001, name: "l11
PolynomialApproximantSteep"),
new Linear(true, N, N, name: "l12 Linear"), // L12
new BatchNormalization(true, N, name: "l12 BatchNorm"),
new PolynomialApproximantSteep(slope: 0.000001, name: "l12
PolynomialApproximantSteep"),
new Linear(true, N, N, name: "l13 Linear"), // L13
new BatchNormalization(true, N, name: "l13 BatchNorm"),
new PolynomialApproximantSteep(slope: 0.000001, name: "l13
PolynomialApproximantSteep"),
new Linear(true, N, N, name: "l14 Linear"), // L14
new BatchNormalization(true, N, name: "l14 BatchNorm"),
new PolynomialApproximantSteep(slope: 0.000001, name: "l14
PolynomialApproximantSteep"),
new Linear(true, N, 10, name: "l15 Linear") // L15
);

nn.SetOptimizer(new Adam());
RunningStatistics stats = new RunningStatistics();
Histogram lossHistogram = new Histogram();
Histogram accuracyHistogram = new Histogram();
Real totalLoss = 0;
long totalLossCounter = 0;
Real highestAccuracy = 0;
Real bestLocalLoss = 0;
Real bestTotalLoss = 0;
// First skeleton save
ModelIO.Save(nn, nn.Name);
```

```
for (int epoch = 0; epoch < 1; epoch++)
{
    RILogManager.Default?.SendDebug("epoch " + (epoch + 1));
    RILogManager.Default?.ViewerSendWatch("epoch", (epoch + 1));

    for (int i = 1; i < TRAIN_DATA_COUNT + 1; i++)
    {
        RILogManager.Default?.SendInformation("batch count " + i + "/" + TRAIN_DATA_COUNT);
        DataSet datasetX = mnistData.GetRandomXSet(BATCH_DATA_COUNT, 28, 28);
        Real sumLoss = Trainer.Train(nn, datasetX.Data, datasetX.Label, new SoftmaxCrossEntropy());
        totalLoss += sumLoss;
        totalLossCounter++;
        stats.Push(sumLoss);
        lossHistogram.AddBucket(new Bucket(-10, 10));
        accuracyHistogram.AddBucket(new Bucket(-10.0, 10));

        if (sumLoss < bestLocalLoss && sumLoss != Double.NaN)
            bestLocalLoss = sumLoss;
        if (stats.Mean < bestTotalLoss && sumLoss != Double.NaN)
            bestTotalLoss = stats.Mean;

        try
        {
            lossHistogram.AddData(sumLoss);
        }
        catch (Exception)
        {
        }

        if (i % 20 == 0)
        {
            RILogManager.Default?.SendDebug("\nbatch count " + i + "/" + TRAIN_DATA_COUNT);
            RILogManager.Default?.SendDebug("Total/Mean loss " + stats.Mean);
            RILogManager.Default?.SendDebug("local loss " + sumLoss);
            RILogManager.Default?.SendInformation("batch count " + i + "/" + TRAIN_DATA_COUNT);
            RILogManager.Default?.ViewerSendWatch("batch count", i);
            RILogManager.Default?.ViewerSendWatch("Total/Mean loss", stats.Mean);
        }
    }
}
```

```
RILogManager.Default?.ViewerSendWatch("local loss", sumLoss);
RILogManager.Default?.SendDebug("");

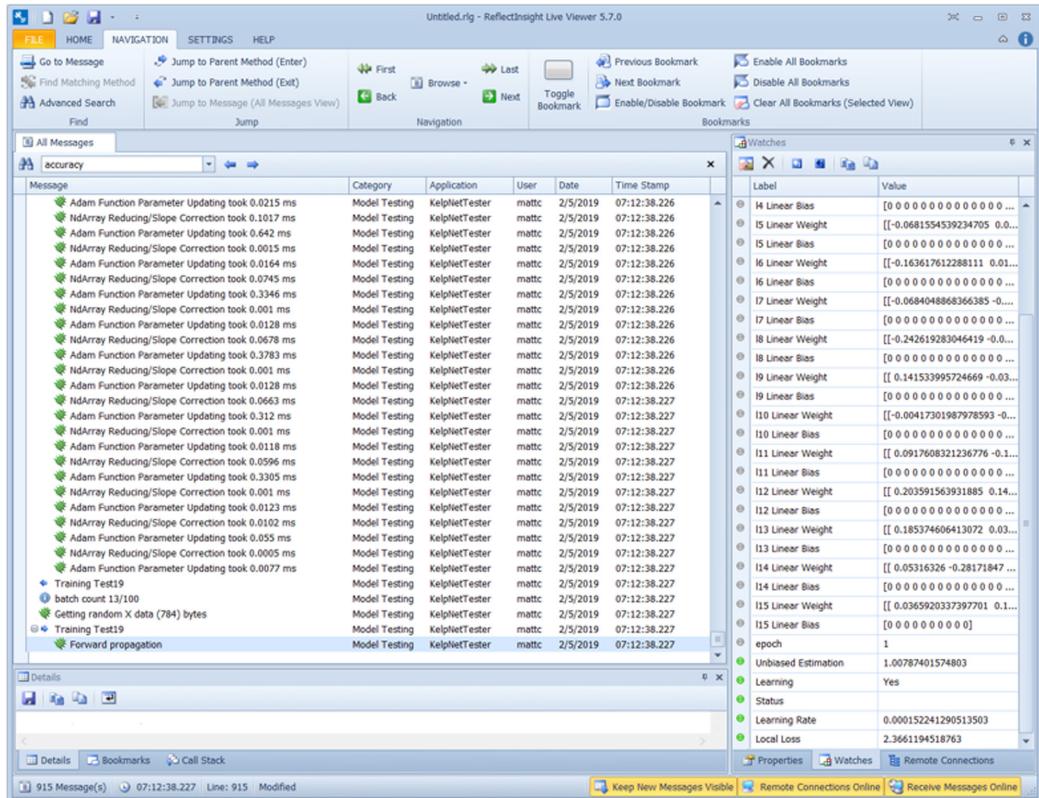
RILogManager.Default?.SendDebug("Testing...");
TestDataSet datasetY = mnistData.GetRandomYSet(TEST_DATA_COUNT, 28);
Real accuracy = Trainer.Accuracy(nn, datasetY.Data, datasetY.Label);
if (accuracy > highestAccuracy)
    highestAccuracy = accuracy;
RILogManager.Default?.SendDebug("Accuracy: " + accuracy);
RILogManager.Default?.ViewerSendWatch("Accuracy", accuracy);

try
{
    accuracyHistogram.AddData(accuracy);
}
catch (Exception)
{
}
}

RILogManager.Default?.SendDebug("Best Accuracy: " + highestAccuracy);
RILogManager.Default?.SendDebug("Best Total Loss " + bestTotalLoss);
RILogManager.Default?.SendDebug("Best Local Loss " + bestLocalLoss);
RILogManager.Default?.ViewerSendWatch("Best Accuracy:", highestAccuracy);
RILogManager.Default?.ViewerSendWatch("Best Total Loss", bestTotalLoss);
RILogManager.Default?.ViewerSendWatch("Best Local Loss", bestLocalLoss);

// Save all with training data
ModelIO.Save(nn, nn.Name);
}
```

## Output



## FunctionStack navigation tests

This test shows how you can navigate each function within a `FunctionStack` individually in both, the forward as well as backward directions.

## Complete source code

The complete source code is shown below, including comments. You can find this source code in the Test22 test of Kelp.Net tests:

```
const int TRAINING_EPOCHS = 5;
const int N_UNITS = 100;
const string DOWNLOAD_URL = "https://raw.githubusercontent.com/wojzaremba/lstm/master/
data/";
const string TRAIN_FILE = "ptb.train.txt";
const string TEST_FILE = "ptb.test.txt";
```

```
public static void Run()
{
    RILogManager.Default?.SendDebug("Building Vocabulary.");

    Vocabulary vocabulary = new Vocabulary();
    string trainPath = InternetFileDownloader.Download(DOWNLOAD_URL + TRAIN_FILE, TRAIN_FILE);
    string testPath = InternetFileDownloader.Download(DOWNLOAD_URL + TEST_FILE, TEST_FILE);
    int[] trainData = vocabulary.LoadData(trainPath);
    int[] testData = vocabulary.LoadData(testPath);
    int nVocab = vocabulary.Length;
    bool verbose = true;

    RILogManager.Default?.SendDebug("Network Initializing.");
    FunctionStack model = new FunctionStack("Test22",
        new EmbedID(verbose, nVocab, N_UNITS, name: "I1 EmbedID"),
        new Linear(true, N_UNITS, N_UNITS, name: "I2 Linear"),
        new Tanh("I2 Tanh"),
        new Linear(true, N_UNITS, nVocab, name: "I3 Linear"),
        new Softmax("I3 Softmax")
    );

    model.SetOptimizer(new Adam());
    List<int> s = new List<int>();
    RILogManager.Default?.SendDebug("Train Start.");
    SoftmaxCrossEntropy softmaxCrossEntropy = new SoftmaxCrossEntropy();
    for (int epoch = 0; epoch < TRAINING_EPOCHS; epoch++)
    {
        for (int pos = 0; pos < 1000; pos++)
        {
            NdArray h = new NdArray(new Real[N_UNITS]);
            int id = trainData[pos];
            s.Add(id);

            if (id == vocabulary.EosID)
            {
                Real accumloss = 0;
```

```
Stack<NdArray> tmp = new Stack<NdArray>();  
  
for (int i = 0; i < s.Count; i++)  
{  
    int tx = i == s.Count - 1 ? vocabulary.EosID : s[i + 1];  
  
    //l1 EmbedID  
    NdArray l1 = model.Functions[0].Forward(true, s[i])[0];  
  
    //l2 Linear  
    NdArray l2 = model.Functions[1].Forward(true, h)[0];  
  
    //Add  
    NdArray xK = l1 + l2;  
  
    //l2 Tanh  
    h = model.Functions[2].Forward(true, xK)[0];  
  
    //l3 Linear  
    NdArray h2 = model.Functions[3].Forward(true, h)[0];  
  
    Real loss = softmaxCrossEntropy.Evaluate(true, h2, tx);  
    tmp.Push(h2);  
    accumloss += loss;  
    RILogManager.Default?.ViewerSendWatch("Local Loss", loss);  
    RILogManager.Default?.ViewerSendWatch("Total Loss", accumloss);  
}  
for (int i = 0; i < s.Count; i++)  
{  
    model.Backward(true, tmp.Pop());  
}  
model.Update();  
s.Clear();  
}  
if (pos % 100 == 0)  
{  
    RILogManager.Default?.SendDebug(pos + "/" + trainData.Length + " finished");  
}
```

```
        }
    }

RILogManager.Default?.SendDebug("Test Start.");
Real sum = 0;
int wnum = 0;
List<int> ts = new List<int>();
bool unkWord = false;

for (int pos = 0; pos < 1000; pos++)
{
    int id = testData[pos];
    ts.Add(id);
    if (id > trainData.Length)
    {
        unkWord = true;
    }
    if (id == vocabulary.EosID)
    {
        if (!unkWord)
        {
            RILogManager.Default?.ViewerSendWatch("pos", pos);
            RILogManager.Default?.ViewerSendWatch("tsLen", ts.Count);
            RILogManager.Default?.ViewerSendWatch("sum", sum);
            RILogManager.Default?.ViewerSendWatch("wnum", wnum);
            sum += CalPs(model, ts);
            wnum += ts.Count - 1;
        }
        else
        {
            unkWord = false;
        }
        ts.Clear();
    }
}
```

```
static Real CalPs(FunctionStack model, List<int> s)
{
    Real sum = 0;
    NdArray h = new NdArray(new Real[N_UNITS]);

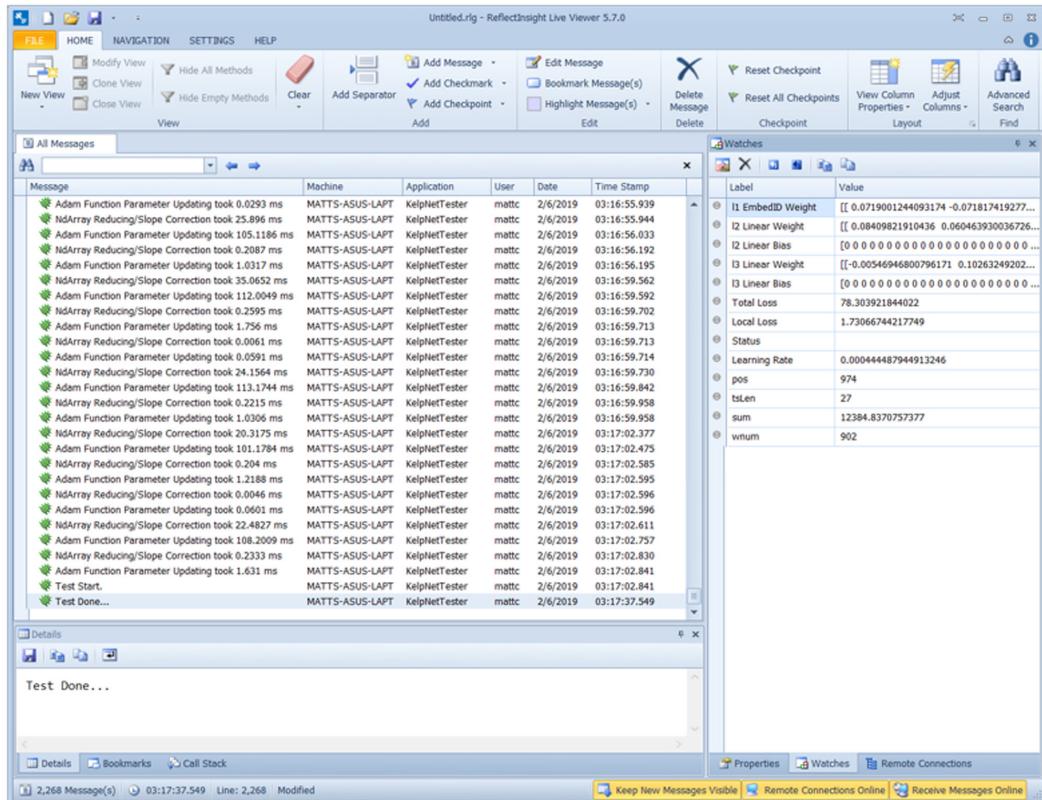
    for (int i = 1; i < s.Count; i++)
    {
        //I1
        NdArray xK = model.Functions[0].Forward(true, s[i])[0];

        //I2
        NdArray l2 = model.Functions[1].Forward(true, h)[0];
        for (int j = 0; j < xK.Data.Length; j++)
        {
            xK.Data[j] += l2.Data[j];
        }

        //I2
        h = model.Functions[2].Forward(true, xK)[0];

        //I3 I4
        NdArray yv = model.Functions[4].Forward(true, model.Functions[3].Forward(true, h))[0];
        Real pi = yv.Data[s[i - 1]];
        sum -= Math.Log(pi, 2);
    }
    return sum;
}
```

## Output



## Learning Rate Hyperparameter tester

This test helps you determine the best learning rate, in terms of elapsed time, for your model test. Basically, it iterates through several different learning rates, runs training and tests, and then tracks the time of each. It reports watches for each learning rate and the elapsed time, and in the end, reports back which was the fastest.

## Complete source code

The complete source code is shown below, including comments. You can find this source code in the Test23 test of Kelp.Net tests:

```
const int learningCount = 10000;
```

```
Real[][] trainData =
{
```

```
new Real[] { 0, 0 },
new Real[] { 1, 0 },
new Real[] { 0, 1 },
new Real[] { 1, 1 }
};

Real[][] trainLabel =
{
    new Real[] { 0 },
    new Real[] { 1 },
    new Real[] { 1 },
    new Real[] { 0 }
};

bool verbose = true;

RILogManager.Add("Test23", "Test23");
RILogManager.SetDefault("Test23");
Stopwatch sw = new Stopwatch();
sw.Start();
FunctionStack nn = new FunctionStack("Test23",
    new Linear(verbose, 2, 2, name: "I1 Linear"),
    new Sigmoid(name: "I1 Sigmoid"),
    new Linear(verbose, 2, 2, name: "I2 Linear"));
long lBest = 0L;
Decimal bestLearningRate = 0M;
TimeSpan bestElapsedTime = TimeSpan.Zero;

for (Decimal d = 0.0005M; d <= 1.0M; d+= 0.005M)
{
    nn.SetOptimizer(new RMSprop("RMSProp", Convert.ToDouble(d)));
    Info("Training with learning rate of " + d);
    for (int i = 0; i < learningCount; i++)
    {
        for (int j = 0; j < trainData.Length; j++)
            Trainer.Train(nn, trainData[j], trainLabel[j], new SoftmaxCrossEntropy());
    }
}
```

```
Info("Test Start...");  
  
foreach (Real[] input in trainData)  
{  
    NdArray result = nn.Predict(true, input)?[0];  
    int resultIndex = Array.IndexOf(result?.Data, result.Data.Max());  
    Info($"'{input[0]} xor {input[1]} = {resultIndex} {result}'");  
}  
sw.Stop();  
if (verbose)  
{  
    RILogManager.Default?.SendDebug("Test took " + Helpers.FormatTimeSpan(sw.Elapsed));  
    RILogManager.Default?.ViewerSendWatch(d + " time", Helpers.FormatTimeSpan(sw.Elapsed));  
}  
  
if (sw.ElapsedMilliseconds < lBest)  
{  
    lBest = sw.ElapsedMilliseconds;  
    bestLearningRate = d;  
    bestElapsed = sw.Elapsed;  
}  
}  
  
Info("Saving Model...");  
ModelIO.Save(nn, "test23.dn");  
Info("Best learning rate was " + bestLearningRate + " which took " + Helpers.FormatTimeSpan(bestElapsed));
```

## Output

0.0005 time	1 minute, 16 seconds, 387 milliseconds
0.0505 time	1 minute, 12 seconds, 969 milliseconds
0.1005 time	1 minute, 11 seconds, 176 milliseconds
0.1505 time	1 minute, 11 seconds, 882 milliseconds
0.2005 time	1 minute, 8 seconds, 845 milliseconds
0.2505 time	1 minute, 13 seconds, 218 milliseconds
0.3005 time	1 minute, 9 seconds, 728 milliseconds
0.3505 time	1 minute, 4 seconds, 509 milliseconds
0.4005 time	1 minute, 5 seconds, 962 milliseconds
0.4505 time	1 minute, 14 seconds, 184 milliseconds
0.5005 time	1 minute, 6 seconds, 654 milliseconds
0.5505 time	1 minute, 3 seconds, 926 milliseconds
0.6005 time	1 minute, 4 seconds, 827 milliseconds
0.6505 time	1 minute, 7 seconds, 531 milliseconds
0.7005 time	1 minute, 4 seconds, 706 milliseconds
0.7505 time	1 minute, 6 seconds, 705 milliseconds
0.8005 time	1 minute, 9 seconds, 416 milliseconds
0.8505 time	1 minute, 7 seconds, 963 milliseconds
0.9005 time	1 minute, 11 seconds, 746 milliseconds
0.9505 time	1 minute, 8 seconds, 575 milliseconds

As the tests go on, you will begin to see where one learning rate may exceed or fall behind the performance of others, as shown in the preceding screenshot, hence the *No Free Lunch* theorem that we discussed earlier.

## Model scoring

This test will help you obtain important scoring characteristics about the performance of your model.

## Complete source code

The complete source code is shown below, including comments. You can find this source code in the Test5 test of Kelp.Net tests:

```
// Describe each initial value
Real[,,,] initial_W1 =
{
```

```
    {{{1.0, 0.5, 0.0}, {0.5, 0.0, -0.5}, {0.0, -0.5, -1.0}}},  
    {{{0.0, -0.1, 0.1}, {-0.3, 0.4, 0.7}, {0.5, -0.2, 0.2}}}  
};  
Real[] initial_b1 = { 0.5, 1.0 };  
  
Real[,,] initial_W2 =  
{  
    {{{-0.1, 0.6}, {0.3, -0.9}}, {{ 0.7, 0.9}, {-0.2, -0.3}}},  
    {{{-0.6, -0.1}, {0.3, 0.3}}, {{-0.5, 0.8}, { 0.9, 0.1}}}  
};  
Real[] initial_b2 = { 0.1, 0.9 };  
  
Real[,] initial_W3 =  
{  
    {0.5, 0.3, 0.4, 0.2, 0.6, 0.1, 0.4, 0.3},  
    {0.6, 0.4, 0.9, 0.1, 0.5, 0.2, 0.3, 0.4}  
};  
Real[] initial_b3 = { 0.01, 0.02 };  
Real[,] initial_W4 = { { 0.8, 0.2 }, { 0.4, 0.6 } };  
Real[] initial_b4 = { 0.02, 0.01 };  
  
//Input data  
NdArray x = new NdArray(new Real[,]{  
    { 0.0, 0.0, 0.0, 0.0, 0.2, 0.9, 0.2, 0.0, 0.0, 0.0, 0.0},  
    { 0.0, 0.0, 0.0, 0.0, 0.2, 0.8, 0.9, 0.1, 0.0, 0.0, 0.0},  
    { 0.0, 0.0, 0.0, 0.1, 0.8, 0.5, 0.8, 0.1, 0.0, 0.0, 0.0},  
    { 0.0, 0.0, 0.0, 0.3, 0.3, 0.1, 0.7, 0.2, 0.0, 0.0, 0.0},  
    { 0.0, 0.0, 0.0, 0.1, 0.0, 0.1, 0.7, 0.2, 0.0, 0.0, 0.0},  
    { 0.0, 0.0, 0.0, 0.0, 0.0, 0.1, 0.7, 0.1, 0.0, 0.0, 0.0},  
    { 0.0, 0.0, 0.0, 0.0, 0.0, 0.4, 0.8, 0.1, 0.0, 0.0, 0.0},  
    { 0.0, 0.0, 0.0, 0.0, 0.8, 0.4, 0.1, 0.0, 0.0, 0.0, 0.0},  
    { 0.0, 0.0, 0.0, 0.0, 0.2, 0.8, 0.3, 0.0, 0.0, 0.0, 0.0},  
    { 0.0, 0.0, 0.0, 0.0, 0.1, 0.8, 0.2, 0.0, 0.0, 0.0, 0.0},  
    { 0.0, 0.0, 0.0, 0.0, 0.1, 0.7, 0.2, 0.0, 0.0, 0.0, 0.0},  
    { 0.0, 0.0, 0.0, 0.0, 0.0, 0.3, 0.0, 0.0, 0.0, 0.0, 0.0}  
});  
  
//teacher signal
```

```
Real[] t = { 0.0, 1.0 };

// If you want to check the contents of a layer, have an instance as a single layer
Convolution2D l2 = new Convolution2D(true, 1, 2, 3, initialW: initial_W1, initialb: initial_b1,
name: "l2 Conv2D");

// ReSharper disable once SuggestVarOrType_SimpleTypes

FunctionStack nn = new FunctionStack("Test5",
l2,
new ReLU(name: "l2 ReLU"),
new MaxPooling(2, 2, name: "l2 MaxPooling"),
new Convolution2D(true, 2, 2, 2, initialW: initial_W2, initialb: initial_b2, name: "l3 Conv2D"),
new ReLU(name: "l3 ReLU"),
new MaxPooling(2, 2, name: "l3 MaxPooling"),
new Linear(true, 8, 2, initialW: initial_W3, initialb: initial_b3, name: "l4 Linear"),
new ReLU(name: "l4 ReLU"),
new Linear(true, 2, 2, initialW: initial_W4, initialb: initial_b4, name: "l5 Linear")
);

// If you omit the optimizer declaration, the default SGD(0.1) will be used
// nn.SetOptimizer(new SGD());
// Training conducted
Real loss = Trainer.Train(nn, x, t, new MeanSquaredError(), false);

// If Update is executed, grad is consumed, so output the value first
RILogManager.Default?.SendDebug("gw1");
RILogManager.Default?.SendDebug(l2.Weight.ToString("Grad"));
RILogManager.Default?.SendDebug("gb1");
RILogManager.Default?.SendDebug(l2.Bias.ToString("Grad"));
RILogManager.Default?.SendDebug("Loss", loss);

//update
nn.Update();

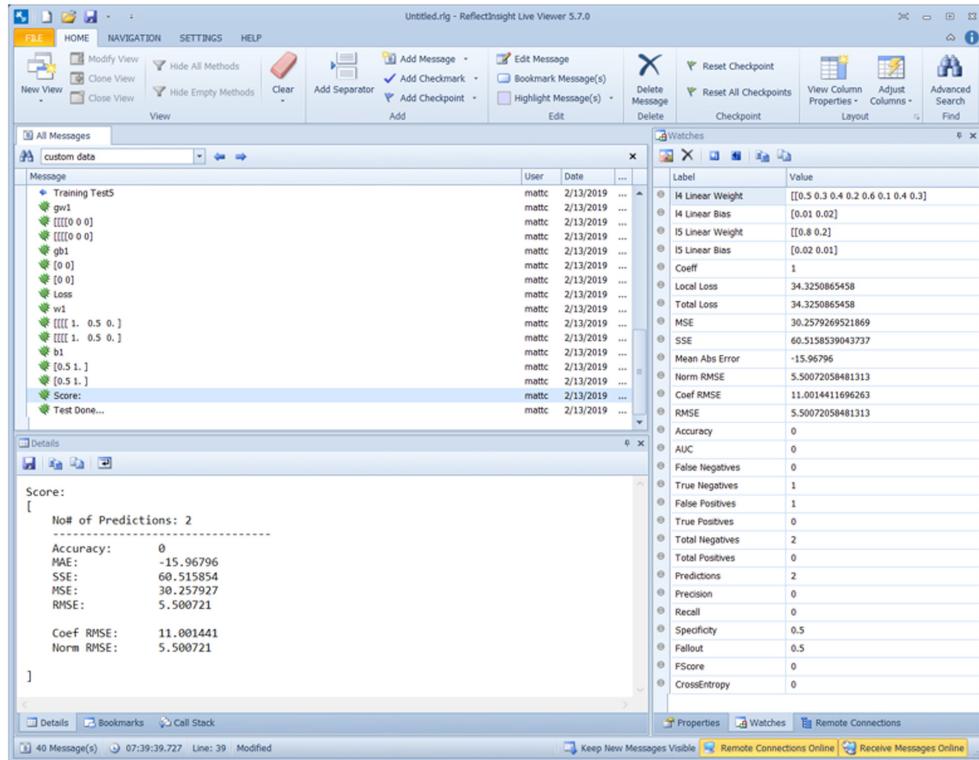
RILogManager.Default?.SendDebug("w1");
RILogManager.Default?.SendDebug(l2.Weight.ToString());
RILogManager.Default?.SendDebug("b1");
```

```
RILogManager.Default?.SendDebug(l2.Bias.ToString());
```

You will get the actual score values here. The scores will be printed out to ReflectInsight in the message and viewer watch windows.

```
NdArray[] results = nn.Predict(true, x);
ModelScorer scores = ModelScorer.ScorePredictions(t, results[0].Data);
RILogManager.Default?.SendDebug(scores.ToString());
}
```

## Output



## Summary

In this chapter, we discussed several deep learning models and showed various techniques of working with them. We discussed how to train and test them, save and load them, and talked about the detailed reviews of the code for each.

In the next chapter, we will show you how to create your own deep learning tests which I am sure you will be able to create in the future.

# CHAPTER 8

# Creating Your Own Deep Learning Tests

In this chapter, we will talk about how you can create your own models and the associated tests to train, test and run them. Doing all this work is not good if you cannot save and reuse your work and the description of the model is what we are after. Also, remember that no two problems are the same, and so, no two solutions are the same. After you create your model, you should be able to spend considerable time for testing and training, comparing evaluation metrics, tweaking, and repeating the process until you achieve the desired results.

You can write a model test in any which way you like. The pattern that you will see throughout Kelp.Net is the usage and exposure of a Run method, which does all the heavy lifting of our model. In the example given below, you will see one of these tests.

Creating a model consists basically of creating the function stack you wish to use, and then selecting and setting the optimizer of your choice. Once this is done, you can train and test your model as well as load and save it as desired. Let us take a quick look at the following example:

## Example

The following code is an example of what you need to implement in order to create your own tests.

```
class TestExample  
{  
}
```

## Implementing the Run function

Throughout most of the model tests, you will see that the standard method of executing a test is via the Run method. Although not set in stone or enforced, this is the paradigm that was established early on. You may feel free to implement your tests anyway you want. The important point is not how you perform your tests, but you perform your tests.

The Run method has the following signature:

```
public static void Run()  
{  
}
```

The following is a complete Run function which has been implemented for your review:

```
public static void Run()  
{  
    const int learningCount = 10000;  
    bool verbose = true;  
  
    Real[][] trainData =  
    {  
        new Real[] { 0, 0 },  
        new Real[] { 1, 0 },  
        new Real[] { 0, 1 },  
        new Real[] { 1, 1 }  
    };  
    Real[][] trainLabel =  
    {  
        new Real[] { 0 },  
        new Real[] { 1 },  
        new Real[] { 1 },  
        new Real[] { 0 }  
    };
```

## Create a FunctionStack with your functions

A FunctionStack is a list of functions chained together to perform forward, update and backward functions. The only other component of a model is the optimizer, which is covered next.

```
FunctionStack nn = new FunctionStack("Test1",
    new Linear(verbose, 2, 2, name: "l1 Linear"),
    new Sigmoid(name: "l1 Sigmoid"),
    new Linear(verbose, 2, 2, name: "l2 Linear"));
```

## Set the optimizer

The optimizer is the last component of the model and is always added after the function stack has been completely assembled. It is set by providing an object which derives itself from the Optimizer base class. Here is how the optimizer is set. Please keep in mind that you need to first create a model by declaring the FunctionStack and using the returned object as shown here:

```
nn.SetOptimizer(new MomentumSGD());
```

## Train the data

With the FunctionStack and Optimizer created and ready to go, you now need to train your model. Use the training dataset and labels you have set aside (remember a good thumb rule is using a single dataset, taking 80-90% for training and the remaining for testing. The higher the number of rows in your dataset, the higher the percentage can go). To train your model, simply call the Trainer.Train method.

```
for (int i = 0; i < learningCount; i++)
{
    for (int j = 0; j < trainData.Length; j++)
    {
        Trainer.Train(nn, trainData[j], trainLabel[j], new SoftmaxCrossEntropy());
    }
}
```

## Make your predictions

Next, with your model fully trained, you can make predictions based on the data as follows:

```
foreach (Real[] input in trainData)
{
    NdArray result = nn.Predict(true, input)?[0];
    int resultIndex = Array.IndexOf(result?.Data, result.Data.Max());
    Console.WriteLine($"{input[0]} xor {input[1]} = {resultIndex} {result}");
}
```

## Save the model

Models are saved via the ModelIO class and are done so using the Save and Load methods (overloaded where appropriate). One quick note on saving of models is that you will notice that the size of your model will vary depending on where in the pipeline you call the save method. This is due to the *Define-by-Run* approach we discussed earlier in the book. Saving the model after testing and training will result in a larger model typically than if done so just after setting your optimizer:

```
ModelIO.Save(nn, "test1.dn");  
}  
}
```

## Loading models

Models which have been saved to the disk using the ModelIO.Save function can also be loaded back into the memory using the ModelIO.Load functions. To load a model from the disk, follow the given instructions:

1. Open the file.
2. Deserialize the model which creates a FunctionStack object.
3. Reset the state of each function in the FunctionStack.
4. Reset the parameters of each Optimizer function. Please note that while typical deep networks consist of a single optimizer, there are no technical limits to using more than one.
5. If the function is parallelizable, an OpenCL Kernel will be created for execution.

## Summary

In this chapter, we discussed how to create your own deep learning tests. As you have learned, it is not a complicated process at all. The work, as it should be, centers around the test logic and the model itself. By following just a few, short principles, you should be up and running and creating your own tests in no time.

## Thank You

At this point I want to say a big Thank You for purchasing and reading this book. Hopefully, you have learned that both Kelp.Net and ReflectInsight are two very powerful tools which can make your work in deep learning a lot easier, and of course, more fun. I hope you enjoyed reading this book as much as I have writing it, and you are taking away a giant arsenal of practical tools and techniques that will help you in your programming endeavors. I wish you all the luck in creating your models and tests and hope to see the great work I know you are about to accomplish!

*Matt R. Cole*



# Appendix A

## Evaluation metrics

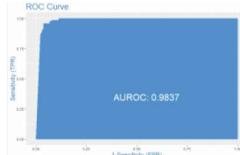
This section will briefly outline some terminologies and concepts you should be aware of when evaluating your deep learning models. It is not enough to simply produce a model; you need to have metrics against which you can compare the efficiency of that model:

		True condition			
		Condition positive	Condition negative	Prevalence = $\frac{\sum \text{Condition positive}}{\sum \text{Total population}}$	Accuracy (ACC) = $\frac{\sum \text{True positive} + \sum \text{True negative}}{\sum \text{Total population}}$
Predicted condition	Predicted condition positive	<b>True positive</b> , Power	<b>False positive</b> , Type I error	Positive predictive value (PPV), Precision = $\frac{\sum \text{True positive}}{\sum \text{Predicted condition positive}}$	False discovery rate (FDR) = $\frac{\sum \text{False positive}}{\sum \text{Predicted condition positive}}$
	Predicted condition negative	<b>False negative</b> , Type II error	<b>True negative</b>	False omission rate (FOR) = $\frac{\sum \text{False negative}}{\sum \text{Predicted condition negative}}$	Negative predictive value (NPV) = $\frac{\sum \text{True negative}}{\sum \text{Predicted condition negative}}$
		True positive rate (TPR), Recall, Sensitivity, probability of detection = $\frac{\sum \text{True positive}}{\sum \text{Condition positive}}$	False positive rate (FPR), Fall-out, probability of false alarm = $\frac{\sum \text{False positive}}{\sum \text{Condition negative}}$	Positive likelihood ratio (LR+) = $\frac{\text{TPR}}{\text{FPR}}$	Diagnostic odds ratio (DOR) = $\frac{\text{LR+}}{\text{LR-}} = \frac{\text{TPR}}{\text{FNR}} \cdot \frac{\text{FNR}}{\text{TNR}}$
		False negative rate (FNR), Miss rate = $\frac{\sum \text{False negative}}{\sum \text{Condition positive}}$	Specificity (SPC), Selectivity, True negative rate (TNR) = $\frac{\sum \text{True negative}}{\sum \text{Condition negative}}$	Negative likelihood ratio (LR-) = $\frac{\text{FNR}}{\text{TNR}}$	<b>F<sub>1</sub> score</b> = $\frac{1}{\frac{1}{\text{Recall}} + \frac{1}{\text{Precision}}} = \frac{2 \cdot \text{Recall} \cdot \text{Precision}}{\text{Recall} + \text{Precision}}$

## Metrics terminology

Some important terms are described as follows:

Term	Description
<b>True positives</b>	Predicted yes and the ground truth is yes.
<b>True negatives</b>	Predicted no and the ground truth is no.
<b>False positives</b>	Predicted yes but the ground truth is no. This is sometimes called a Type I error.

Term	Description
<b>False negatives</b>	Predicted no but the ground truth is yes. This is sometimes called a Type II error.
<b>Accuracy</b>	How often is the classifier correct? $(TP + TN) / \text{total}$ .
<b>Misclassification rate</b>	How often is the classifier wrong? $(FP + FN) / \text{total}$ . Also, equivalent to 1 minus the Accuracy and sometimes called the <b>error rate</b> .
<b>True positive rate</b>	When the ground truth is yes, how often is yes predicted? $(TP / \text{actual yes})$ . This is sometimes known as the <b>sensitivity</b> or <b>recall</b> .
<b>False positive rate</b>	When the ground truth is no, how often is yes predicted? $(FP / \text{actual no})$
<b>True negative rate</b>	When the ground truth is no, how often is no predicted? $(TN / \text{actual no})$ . Equal to $1 - \text{False Positive Rate}$ , also known as <b>specificity</b> .
<b>Precision</b>	When predicted yes, how often is it correct? $(TP / \text{predicted yes})$
<b>Prevalence</b>	How often does the yes condition occur in our sample? $(\text{actual yes} / \text{total})$
<b>Null error rate</b>	How often would you be wrong if you predicted the majority class?
<b>F score</b>	Weighted average of the true positive rate (recall) and precision.
<b>ROC curve</b> 	Performance summary over all possible thresholds. The true positive rate is the Y axis, and the false positive rate is the X axis.
<b>Detection rate</b>	Correctly predicted 1's as a percentage of the entire sample.
<b>Detection prevalence</b>	Percentage of the full sample that was predicted as 1.
<b>Balanced accuracy</b>	The balance between correctly predicting 1's and 0's.
<b>Recall</b>	The percentage of all 1's which were correctly predicted.

Term	Description
Cohens kappa	A measure of how well the classifier performed as compared to how well it would have performed simply by chance.
Concordance	Proportion of concordant pairs.
Somers D	Combination of concordance and discordance. Used to judge the efficiency of the model.
AUROC	Area under the ROC curve. Model's true performance, considering possible probability cut-offs.
Gini coefficient	How the model exceeded random predictions in terms of ROC.

## Confusion matrix

A confusion matrix is a table that describes the performance of a model that was used for classification. The model was created from a set of test data from which the true values were known.

Example of a confusion matrix:

n=165	Predicted:		60
	NO	YES	
Actual: NO	TN = 50	FP = 10	60
Actual: YES	FN = 5	TP = 100	105
		55	110

In the above matrix, we have two possible classes, **YES** and **NO**. True values or things that happened or are on would be **YES**, and the others would be **NO**. The total number of predictions made was **165** (total of all the rows and columns). Out of those **65** predictions, **110** resulted in a predicted value of **YES**, while **55** resulted in a predicted value of **NO**.



# Appendix B

## OpenCL

Kelp.Net makes heavy usage of the open computing language, or OpenCL. OpenCL views a computing system as consisting of several compute devices, which might be CPUs or GPUs attached to a CPU. Functions executed on an OpenCL device are called **kernels**. A single compute device typically consists of several compute units, which in turn comprise multiple processing elements. A single kernel execution can run on all or many of the processing elements in parallel.

In OpenCL, tasks are scheduled onto command queues. There is at least one command queue for each device. The OpenCL runtime breaks the scheduled data-parallel tasks into pieces and sends the tasks to the device processing element.

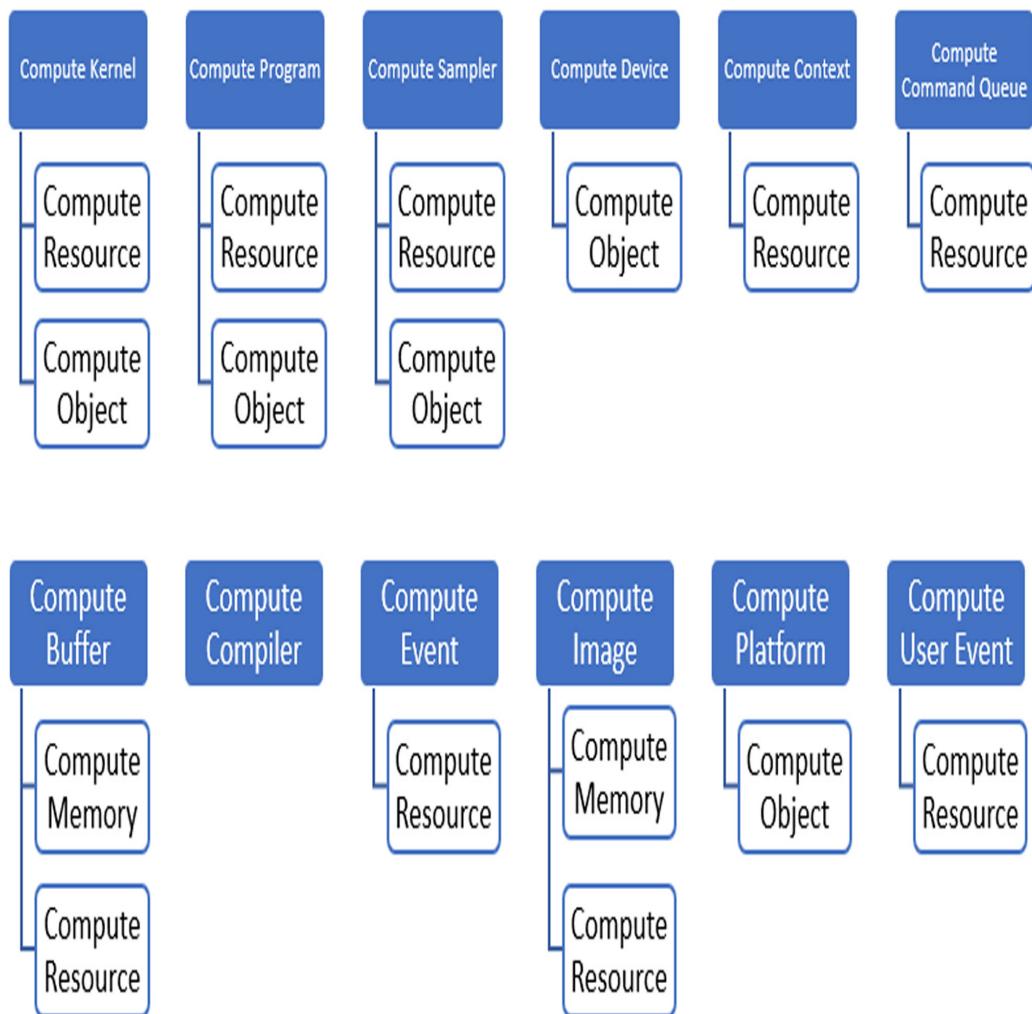
OpenCL defines a memory hierarchy with the following attributes:

- **Global:** Shared by all processing elements and has high latency
- **Read-only:** Smaller, of lower-latency, and writable by the host CPU but not compute devices
- **Local:** Shared by a process element group
- **Per-element:** Private memory

OpenCL also provides an **Application Programming Interface (API)** designed more towards math. This can be seen in the exposure of fixed-length vector types such as float4 (four vector of single-precision floats), available in lengths of 2, 3, 4, 8, and 16. As you gain more exposure to Kelp.Net and start to create your own functions, you will encounter OpenCL programming. For now, it is enough to know that it exists and is being used under the hood extensively.

# OpenCL hierarchy

In Kelp.Net, the hierarchy for various OpenCL resources is as shown here:



Let us describe these in more detail:

- **Compute kernel:** A kernel object encapsulates a specific kernel function declared in a program and the argument values to be used when executing this kernel function.

- **Compute program:** An OpenCL program consisting of a set of kernels. Programs may also contain auxiliary functions called by the kernel functions and constant data.
- **Compute sampler:** An object that describes how to sample an image when the image is read in the kernel. The image read functions take a sampler as an argument. The sampler specifies the image addressing mode, which means how out-of-range coordinates are handled, the filtering mode and whether the input image coordinate is a normalized or unnormalized value.
- **Compute device:** A compute device is a collection of compute units. A command queue is used to queue commands to a device. Examples of commands include executing kernels or reading/writing memory objects. OpenCL devices typically correspond to a GPU, a multi-core CPU, and other processors such as **Digital Signal Processor (DSP)** and the cell/B.E. processor.
- **Compute resource:** An OpenCL resource that can be created and deleted by the application.
- **Compute object:** An object which is identified by its handle in the OpenCL environment.
- **Compute context:** A compute context is the actual environment within which the kernels execute and the domain in which synchronization and memory management are defined.
- **Compute command queue:** A command queue is an object which holds commands that will be executed on a specific device. The command queue is created on a specific device within a context. Commands to a queue are queued in-order but may be executed either in-order or out of order.
- **Compute buffer:** A memory object that stores a linear collection of bytes. Buffer objects are accessible using a pointer in a kernel executing on a device.
- **Compute event:** An event encapsulates the status of an operation such as a command. It can be used to synchronize operations in a context.
- **Compute image:** A memory object that stores a 2D or 3D structured array. Image data can only be accessed with read and write functions. Read functions use a sampler.
- **Compute platform:** The host plus a collection of devices managed by the OpenCL framework that allows an application to share resources and execute kernels on devices on the platform.
- **Compute user event:** This represents a user-created event.



# Index

## A

activation function 19, 20, 171  
  Rectified Linear Unit (ReLU) 21  
  sigmoid 21  
  Softmax 22  
activation plots 172  
AdaDelta 119  
AdaGrad 120  
Adam 121  
Advanced Search 70  
Alpha 314  
Application Programming Interface (API) 387  
ArcSinH 172  
ArcTan 174  
Artificial Intelligence 10  
assertion 80  
assigned 80  
assigned varibales 80  
attachment 80  
audit failure 80  
audit success 80  
Auto Purge method 72

Auto Purge rolling log files 72  
Auto Save method 72  
Auto Save rolling log files 72  
AveragePooling 136

## B

back propagation 25, 26  
back-propagation through time (BPTT) 42  
batches/batch size 35  
batch normalization 36, 37  
BatchNormalization 261  
Bayesian optimization 61  
bias 19  
bias-variance trade-off 50  
bookmark panel 68

## C

Call Stack panel 68  
checkmarks 80  
checkpoint 80  
checkpoints 80

- CIFAR-10 dataset 286
- CIFAR-100 dataset 287
- collections 80
- comments 80
- common stacks 296
- complexity 12
- concurrency 80
- configuration editor 78, 79
- confusion matrix 385
- connections, Kelp.Net
  - about 213
  - Convolution2D 214
  - Deconvolution2D 228
  - EmbedID 242
  - Linear 244
- Long Short-Term Memory (LSTM)
  - 252
- containers, Kelp.Net
  - about 141
  - FunctionDictionary 147
  - FunctionStack 141
  - SortedFunctionStack 165
  - SortedList 157
  - SplitFunction 154
- Convolution2D 214
- convolutional neural network (CNN)
  - 37, 38
- Convolutional Neural Network (CNN) 127
- cost function 27, 28
- Curse of Dimensionality 27
- custom data
  - using 102
- D**
- data 80
  - training 54, 379
- data heterogeneity 54, 55
- data prediction
- creating 379
- dataset
  - about 286
  - CIFAR-10 dataset 286
  - CIFAR-100 dataset 287
  - MNIST database 288
  - SVHN dataset 288
- DataSet 80
- datasets
  - features 18
- DataSetSchema 80
- DataTable 80
- DataTableSchema 80
- DataView 80
- Date/Time 80
- Debug 102
- Deconvolution2D 228
- Decoupled Neural Interfaces (DNI)
  - about 329
  - output 335
  - source code 330, 331, 332, 333, 334, 335
- Synthetic Gradients, used 329, 330
- deep learning
  - about 17
  - manifold learning, types in 58
  - objectives 5, 6
  - overview 10, 11
  - versus machine learning 13, 14
- deep learning model
  - loading 304, 380
  - saving 305, 380
  - size 306, 307
- deep learning test
  - example 377
- deep network
  - output 342
  - source code 340
  - testing 339
- Desktop Image 102

differentiable manifold 59  
 Digital Signal Processor (DSP) 389  
 DropConnect 53  
 dropout 36  
 Dropout 273  
 DropOut 53  
 Dynamic configuration 79

**E**

effective capacity 60  
 ELU 176  
 EmbedID 242  
 epoch 35, 36  
 error rate 384  
 Errors 102  
 evaluation metrics 383  
 evolutionary optimization 61  
 Exceptions 102  
 exploding gradient problem 43, 123  
 extensions 80

**F**

Fatal Errors 102  
 filtering 71  
 forward propagation 25  
 fully-connected networks 18  
 function benchmarking  
   about 345  
   output 358  
   source code 345  
 FunctionDictionary 147  
 FunctionStack 116, 141  
   about 115, 116  
   creating, with function 378  
   example 115  
 FunctionStack navigation  
   output 370  
   source code 365  
   testing 365

**G**

GANs 23  
 Gaussian 178  
 generalization 46, 47  
 Generations 102  
 gradient-based optimization 61  
 GradientClipping 122  
 gradient descent 26, 29, 30, 31, 32  
 grid search 61  
 GRUs 23

**H**

hidden layer 25, 44, 45  
 hidden layers 10  
 hyperparameter  
   aspects 62  
   training 60, 61  
 hyperparameter tuning  
   approaches 61  
   Bayesian optimization 61  
   evolutionary optimization 61  
   gradient-based optimization 61  
   grid search 61  
   random search 61

**I**

image prediction  
   example 342  
   output 344  
   source code 343  
   test  
   ing 342  
 Images 102  
 incorrect output values 54  
 Information 102  
 input layer 25  
 iterations 36

## K

Kelp.Net  
 about 108, 109  
 downloading 107  
 function 109, 114  
 function, types 109  
 source code, building 108  
 kernels 387

## L

labeled 49  
 layers  
 versus neurons 27  
 LeakyReLu  
 combination network 360  
 output 365  
 source code 361  
 LeakyReLU 179  
 LeakyReLUShifted 181  
 learning rate 33, 34, 35  
 Levels 102  
 Linear 244  
 Linq queries 102  
 Linq results 102  
 live viewer 75, 76  
 Loaded assemblies 102  
 Loaded processes 102  
 Local Response Normalization (LRN)  
 268  
 logging viewers  
 65  
 LogisticFunction 183  
 logistic neuron 43, 44  
 log viewer 75  
 Long Short-Term Memory (LSTM)  
 252  
 loss  
 about 48  
 over time 48

versus learning curve 48  
 loss function, Kelp.Net  
 about 282  
 MeanSquaredError 283  
 SoftmaxCrossEntropy 284  
 low bias 51

## M

machine learning  
 overview 9, 10  
 versus deep learning 13, 14  
 manifold learning 57, 58  
 differentiable manifold 59  
 Riemannian manifold 59  
 topological manifold 59  
 types, in deep learning 58  
 MaxMinusOne 185  
 MaxPooling 128, 129  
 MeanSquaredError 283  
 Memory status 102  
 message details panel 66  
 message log panel 65  
 message navigation 69  
 message properties  
 attaching 80  
 message properties panel 67  
 Messages 102  
 message type logging reference 80  
 metrics terminology 383  
 MNIST 329  
 accuracy, testing 335  
 output 339  
 source code 336  
 MNIST database 288  
 MNIST handwritten character  
 learning 358  
 output 360  
 source code 358  
 model

accuracy 293, 294, 295  
 output 376  
 scoring 373  
 source code 373  
 timing 295  
**MomentumSGD** 124  
**multi-layer perceptron (MLP)** 17, 18

**N**

**NdArray**  
 about 114  
 properties 115  
**n-dimensional arrays** 114, 115  
**neural network** 22, 23  
 about 24  
 hidden layer 25  
 input layer 25  
 output layer 25  
 overview 7, 8, 9, 15, 16  
 types 45  
**neural network, components**  
 Axon 24  
 Dendrites 23  
 Soma 24  
**neuron** 16, 17  
**neurons**  
 versus layers 27  
**No Free Lunch theorem** 27  
**noise function, Kelp.Net**  
 about 272  
 Dropout 273  
 StochasticDepth 279  
**normalization funtion, Kelp.Net**  
 about 261  
 BatchNormalization 261  
 Local Response Normalization  
 (LRN) 268  
**Notes** 102

**O**

**Objects** 102  
**OpenCL**  
 about 387  
 global 387  
 local 387  
 per-element 387  
 read-only 387  
**OpenCL hierarchy**  
 about 388  
 compute buffer 389  
 compute command queue 389  
 compute context 389  
 compute device 389  
 compute event 389  
 compute image 389  
 compute kernel 388  
 compute object 389  
 compute platform 389  
 compute program 389  
 compute resource 389  
 compute sampler 389  
 compute user event 389  
**optimizer** 116  
 setting up 379  
 with Kelp.Net 118  
**Output** 104  
 output layer 25  
 overfitting 50, 51  
 preventing 52

**P**

padding 39, 40  
**perceptron** 16, 17  
**PolynomialApproximantSteep** 187  
 combination network 360  
 output 365  
 source code 361  
 pooling layer 39

pooling layers 127, 128  
Principal Component Analysis (PCA)  
54, 59  
Process Information 102  
Python 3

## Q

QuadraticSigmoid 189

## R

random search 61  
Rate Hyperparameter tester  
learning 370  
output 373  
source code 370  
RbfGaussian 191  
recall 384  
Rectified Linear Unit (ReLU) 21  
Recurrent Neural Network Language  
Models (RNNLM)  
about 316  
source code 316, 317, 318, 319, 320,  
321  
Recurrent Neural Network (RNN)  
23, 41, 42  
recurrent neuron 40, 41  
redundant 53  
ReflectInsight (RI)  
benefits 64  
features 64  
ReflectInsight (RI), component  
configuration editor 78  
live viewer 75, 76  
log viewer 75  
router 74  
Software Development kit (SDK) 77  
regularization 47, 53  
reinforcement learning 57  
ReLU 193

ReLU{Tanh} 194  
Reminders 102  
Riemannian manifold 59  
R language 3  
RMSPROP 125  
router 74  
Run function  
implementing 378

## S

sample padding 39  
ScaledELU 196  
SendException message 77  
sensitivity 384  
Serialized Objects 104  
sigmoid 21  
Sigmoid 198  
Sigmoid Neuron 43  
Sine 200  
Softmax 22, 202  
SoftmaxCrossEntropy 284  
Softplus 204  
Software Development Kit (SDK) 77  
SortedFunctionStack 165  
SortedList 157  
specificity 384  
SplitFunction 154  
SQL strings 104  
SReLU 206  
SReLUShifted 208  
Stack Traces 104  
StochasticDepth 279  
Stochastic Gradient Descent (SGD)  
124, 126  
Streams 104  
Street View House Numbers (SVHN)  
288  
supervised 49  
supervised learning 49  
Supervised Learning 10

Swish 210

### Synthetic Gradients

about 329

used, for Decoupled Neural Interfaces (DNI) 330

used, in Decoupled Neural Interfaces (DNI) 329

System Information 104

## T

Tanh 212

Text files 104

Thread Information 104

Time zone formatting

about 74

Time-zone information 104

To all request 80

topological manifold 59

To request 80

To single message 80

Tracing method calls 80

Typed collections 104

## U

underfitting 51

preventing 52

unsupervised learning 55, 56, 57

User-Defined Views (UDV) 64, 65, 71

## V

vanishing gradient problem 122

about 42

approaches 42

variance 50

vocabulary

about 321

output 321

## W

Warning 104

Watches 80

watch panel 73

weight factor 19

weights 17

word prediction test

about 322

output 322

source code 323

## X

XML 104

XML-based configuration file

configuration editor 79

Dynamic configuration 79

overview 79

XML configuration file 79

XML files 104

XOR problem

about 309, 310, 314

model, testing 313, 314

output 312, 313, 314

source code 310, 311, 314

