# Deep Learning Methods for Optimization of VLSI Chip Layout

Nathaniel Ogilvie[1,2,+,*] and Peter Larsen[1,3,-,*]

[1]University of Vermont, Burlington, VT
[2]IBM
[3]Complex Systems Center, Farrell Hall, UVM
[+]nogilvie@uvm.edu, Student ID: 953471207
[-]plarsen@uvm.edu, Student ID: 954371861
[*]These authors contributed equally to this report

March 20 2019

## 1 Introduction

There are already many algorithms that can design chip layouts efficiently. However, these algorithms are very rigid, and cannot expand too far beyond their specified size constraints. Thus, companies are forced to either use the algorithms at a sub-optimal accuracy level, and potentially receive inefficient chip designs which can cost money, or to rely on human engineers that, while placing nigh-perfectly due to expertise and knowledge of previous chip design generations, cost the company time. The goal of this project is to see whether various Deep Learning architectures can be utilized to develop these chip layouts similar to the human engineers in a fraction of the time. We believe that there will be one or multiple that are relatively successful at creating functional chip layouts because complex tasks normally requiring human involvement are ideal problems for machine-learning. If one of these methods succeeds, it would mean that a chip company could save time and money by creating chip layouts better and more efficiently than the sub-optimal algorithms or the human engineers.

## 2 Problem Definition and Algorithm

### 2.1 Task Definition

VLSI (Very Large Scale Integration) requires the integration and placement of many macros (blocks of smaller variants) into a larger chip. Currently, there are many algorithms that have been created to optimize the construction of the layouts for these macros, based on connections between blocks and physical size. However, many of those algorithms struggle with unconstrained placements, such as when chip constraints are of a large-enough size, because there are so many permutations available for the algorithms to try. At present, companies are forced to either use sub-optimal algorithms (made for smaller constraints) or to lay out large blocks by hand. Using either of these options is both expensive in terms of time and money. Laying out the blocks by hand also requires an expert in the field that often has knowledge of previous chip generations. If one could train a neural network to act as the expert, and feed into it the previous generations, one could potentially lower costs, and create a more efficient design process. Thus, the task is to create an architecture or architectures that simulate this process for any set of constraints in a quick and efficient manner.

### 2.2 Algorithm Definition: LSTM RNN

We initially decided that the best architecture for constructing these chip layouts could be a recurrent neural network(RNN). We chose this architecture because, in the real world, macros are added to the empty layout sequentially, and each macro is placed using the information from the previously placed macros. This felt very similar to how an RNN is used for sentence construction or sentiment analysis. This RNN would take in several inputs: {name of macro, height of macro, width of macro, macro that it is connected to, Cartesian bounds for total chip containing that macro} and would produce an output consisting of Cartesian coordinates for the lower

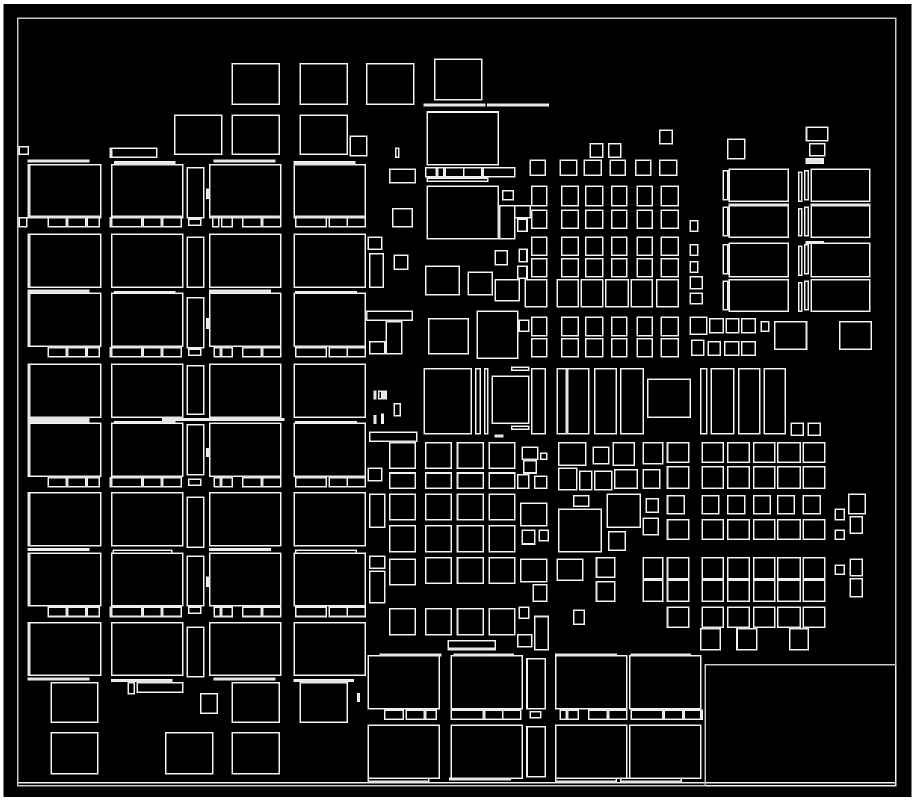

Figure 1: Example of Actual Chip Layout with Perfect Macro Placement

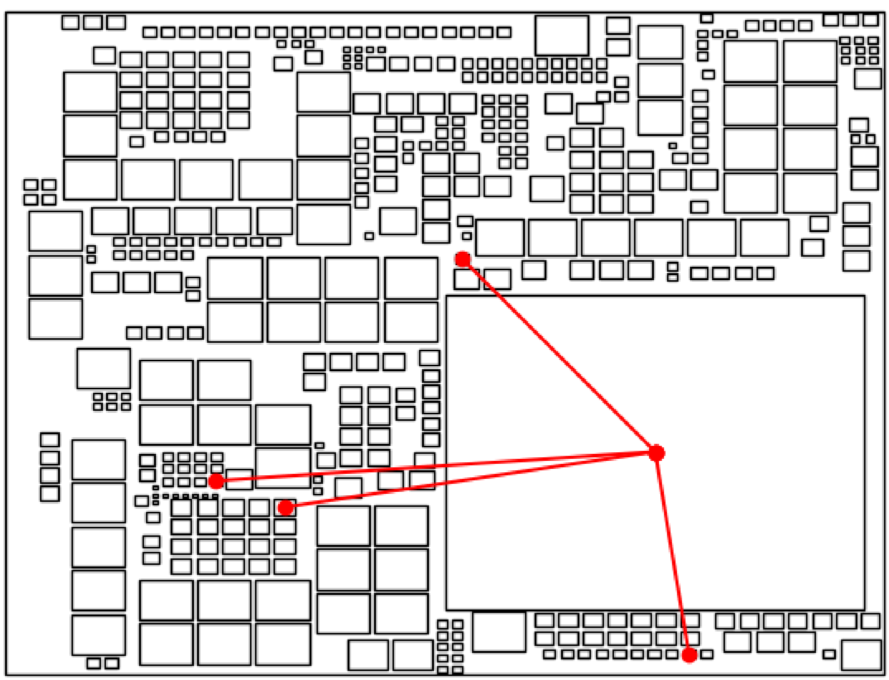

Figure 2: Example of Randomly-Generated Chip For Use in Dataset

left-hand corner of that macro, where the origin (0,0) was the lower left-hand corner of the chip as a whole. We decided that a long short-term memory (LSTM) RNN would be effective, because they can be used for human behavior/action recognition, and we believed that this problem was very much a machine attempting to emulate human action. However, this RNN did not perform as we would have liked, producing results as listed in section 3.2, which indicated to us that either an algorithm change or an architecture change was necessary.

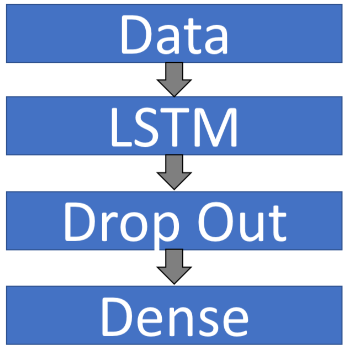


Figure 3: A representation of the LSTM RNN model

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_1 (LSTM)                (None, 1, 8)              544
_________________________________________________________________
dropout_1 (Dropout)          (None, 1, 8)              0
_________________________________________________________________
dense_1 (Dense)              (None, 1, 2)              18
=================================================================
Total params: 562
Trainable params: 562
Non-trainable params: 0
```

Figure 4: A quick summary of some of the elements of the LSTM RNN model

### 2.3 Algorithm Definition: Actor Critic

After testing with LSTM, we decided that examining each macro by itself was insufficient for creating proper chip layouts. Thus, we decided that looking at every macro simultaneously would be better, as it would allow for constant updates to the position of a macro based on the macros it is connected to. This lead us to the Actor Critic model. Within the Actor Critic model, there are two machine-learning algorithms that run to generate the full program: the Actor and the Critic. The Actor is responsible for generating the next action the computer will take based on the Environment's current state and the Q function. The Critic then evaluates the state of the Environment and the reward function from the Environment and updates the Q function, in a cyclical loop of updating actions and environments. Here, the reward function is based on a score calculated using a combination of the the total connection distance between that macro and the others it has a connection with, the number of overlaps, as well as cases for violations of the boundary conditions for the chip. The Actor has 9 input values: {x, y, xmax, ymax, width, height, total connection distance, score, num_overlaps}, with each set of 9 inputs representing one macro. The initial x and y are the lower left-hand corner coordinates, as in the LSTM architecture. Width and height are the x and y length values for the sides of the macro, respectively. xmax and ymax are the boundary limits that the coordinate can be placed, being calculated by the total width or height of the chip minus the width or height of the macro. Thus, combined with the origin, the xmax and ymax form the boundary that the chip can be placed within where it doesn't violate DRC boundary conditions. The total connection distance is the sum of all the distances of the connections with that macro. The score is generated by the function in Section 3.1, and num_overlaps is the number of overlaps for that macro. The Actor processes these nine values and generates an action to affect the Environment, with its output being two integers: $\{\Delta x, \Delta y\}$. These two integers are added to the x and y coordinates in the macro's environment, so effectively it will be:

$$
\begin{aligned}
x_2 &= x_1 + \Delta x \\
y_2 &= y_1 + \Delta y
\end{aligned}
$$

The Critic takes in 11 inputs, the first 9 being from the Environment with the same variables as the Actor inputs, but after the action has been applied to the Environment. The final 2 inputs are the output of the Actor, being $\{\Delta x, \Delta y\}$. The Actor's output is then merged into the Environment network, which the Critic outputs as a new Q function. The Q function is then used to update the Actor model weights, and the Actor produces a new action, creating a cycle.

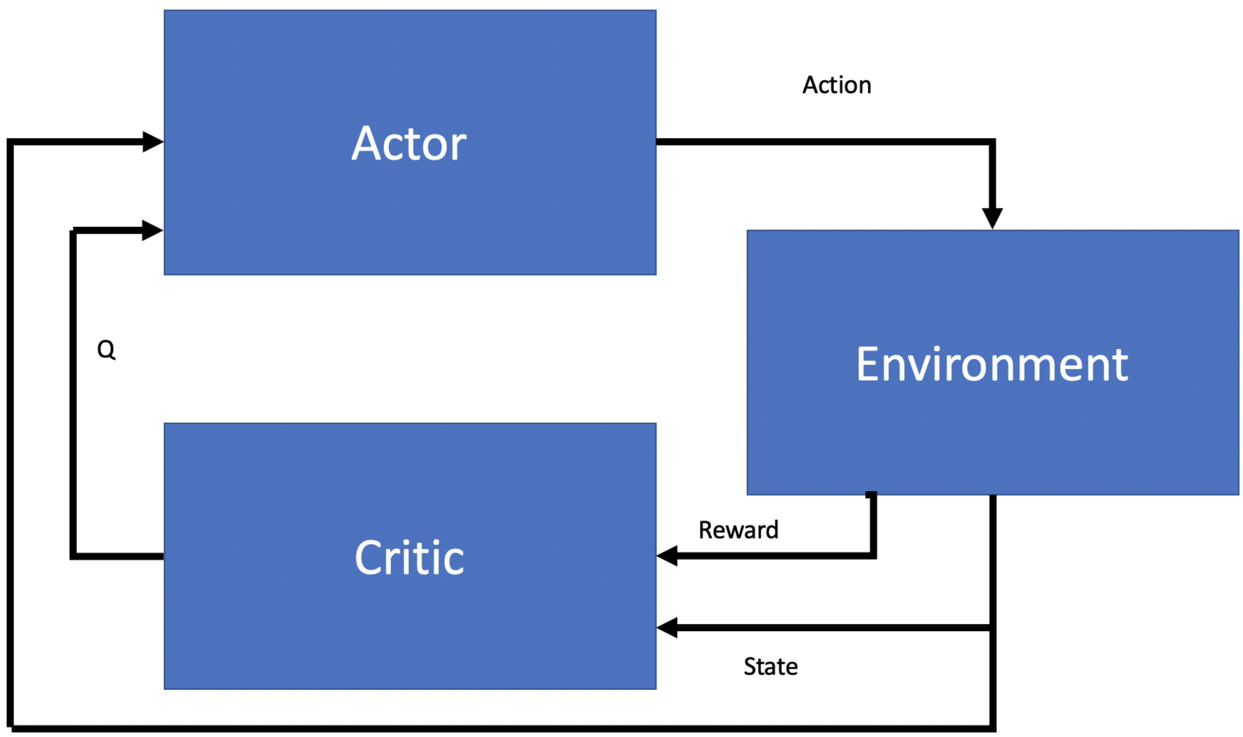


Figure 5: A representation of the Actor Critic model

```
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 8)                 0
_________________________________________________________________
dense_1 (Dense)              (None, 1024)              9216
_________________________________________________________________
dense_2 (Dense)              (None, 1024)              1049600
_________________________________________________________________
dense_3 (Dense)              (None, 512)               524800
_________________________________________________________________
dense_4 (Dense)              (None, 2)                 1026
=================================================================
Total params: 1,584,642
Trainable params: 1,584,642
Non-trainable params: 0
```

Figure 6: A quick summary of the elements of the Actor model

```
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
input_3 (InputLayer)            (None, 8)            0
__________________________________________________________________________________________________
dense_9 (Dense)                 (None, 1024)         9216        input_3[0][0]
__________________________________________________________________________________________________
input_4 (InputLayer)            (None, 2)            0
__________________________________________________________________________________________________
dense_10 (Dense)                (None, 1024)         1049600     dense_9[0][0]
__________________________________________________________________________________________________
dense_11 (Dense)                (None, 1024)         3072        input_4[0][0]
__________________________________________________________________________________________________
add_1 (Add)                     (None, 1024)         0           dense_10[0][0]
                                                                 dense_11[0][0]
__________________________________________________________________________________________________
dense_12 (Dense)                (None, 512)          524800      add_1[0][0]
__________________________________________________________________________________________________
dense_13 (Dense)                (None, 1)            513         dense_12[0][0]
==================================================================================================
Total params: 1,587,201
Trainable params: 1,587,201
Non-trainable params: 0
```

Figure 7: A quick summary of the elements of the Critic model

# 3 Experimental Evaluation

## 3.1 Methodology

We started off with a simple question: Can an RNN replicate or replace human involvement in the chip design process to the point of optimizing placement of macros on an unconstrained chip? To answer this question, we used a table of roughly 1000 chips with the aforementioned inputs. This data was created by a Python script that takes in common designer practices and randomizes sizes, quantities, and placements. The reason for generating these is, although we have access to the actual layouts of chips at IBM through Mr. Ogilvie, the amount of data would have been much less than what we currently have, and in theory, we could generate a nigh-infinite amount of random chips using the data generation functions. (Side note: there are also a lot of hoops and legal guidelines to jump through concerning IBM's proprietary information, and we thought it best to avoid those altogether). Also, upon demonstrating some of the randomly generated chip designs to engineers at IBM, we were informed that they were viable as layouts. The way we evaluated our model was by using the MSE (mean squared error) accuracy metric, which compares the generated x and y coordinates with the actual labelled ones from the test data. The lower the MSE, the better the model did at placing the piece. An MSE of 0 would indicate perfect placement, or 100% accuracy. We didn't believe that the given accuracy metric was effective at describing whether the chip is DRC clean (Design Rule Checker, indicates violations) or not, as it did not take into account perfect placement relative to the macros that the placed macro is connected to (one of the most important parts to proper layouts in chip design). We decided to implement a new model that we believe to have a better accuracy metric that does take into account whether the chip was DRC clean, being an Actor Critic model. This model was evaluated by using the final score of the entire chip, designated as $score_{chip}$ and calculated like so:

$$
\begin{aligned}
boundary\_error &= \begin{cases} 1,000,000 + (x - xMax) * 100x & x > xMax \\ 1,000,000 + (y - yMax) * 100y & y > yMax \\ 2,000,000 + (x - xMax) * 100x + (y - yMax) * 100y & x > xMax \ \& \ y > yMax \\ 1,000,000 + x * 100x & x < 0 \\ 1,000,000 + y * 100y & y < 0 \\ 2,000,000 + x * 100x + y * 100y & x < 0 \ \& \ y < 0 \\ 0 & x < xmax \ \& \ y < ymax \ \& \ x > 0 \ \& \ y > 0 \end{cases} \\
left\_Point &= \begin{cases} X_1 & X_1 > X_2 \\ X_2 & X_1 < X_2 \end{cases} \\
right\_Point &= \begin{cases} X_1 + W_1 & X_1 + W_1 > X_2 + W_2 \\ X_2 + W_2 & X_1 + W_1 < X_2 + W_2 \end{cases} \\
bottom\_Point &= \begin{cases} Y_1 & Y_1 > Y_2 \\ Y_2 & Y_1 < Y_2 \end{cases} \\
top\_Point &= \begin{cases} Y_1 + H_1 & Y_1 + H_1 > Y_2 + H_2 \\ Y_2 + H_2 & Y_1 + H_1 < Y_2 + H_2 \end{cases} \\
height_{overlap} &= top\_Point - bottom\_Point \\
width_{overlap} &= right\_Point - left\_Point \\
overlap_{area} &= width_{overlap} * height_{overlap} \\
score_i &= \left[ \sum_{j=1}^{n=num\_connections} dist_j \right] + overlap_{area} * 1,000 + boundary\_error \\
score_{chip} &= \sum_{i=1}^{n=num\_macros} score_i
\end{aligned}
$$

The score of the chip is then compared with the initial score from the unaltered chip (called $score_{init}$) and the average score of the training chips (called $score_{fed}$). The way this is done is by taking the difference between between $score_{init}$ and $score_{fed}$. Our accuracy is then calculated like so:

$$
accuracy = \begin{cases} 0 & score_{chip} \geq score_{init} \\ 1 & score_{chip} \geq score_{fed} \\ \frac{score_{init} - score_{chip}}{score_{init} - score_{fed}} & score_{fed} < score_{chip} < score_{init} \end{cases}
$$

### 3.2 Results

Our results as they stand are an information loss of 8032885.37, with an accuracy of 50.3% using the MSE (mean squared error) between the generated x and y coordinate and the labelled x and y coordinate. These numbers were produced using a table of 6,660,890 lines (the test data) and 10 epochs.

Below are the results from our simulations of the Actor Critic model. Many changes occurred, where we increased batch size, number of parameters, and eventually number of chips trained on. Thus, in the end, after training on 10 chips, we achieved an accuracy of roughly 3%. While this number is very low, we are encouraged by the

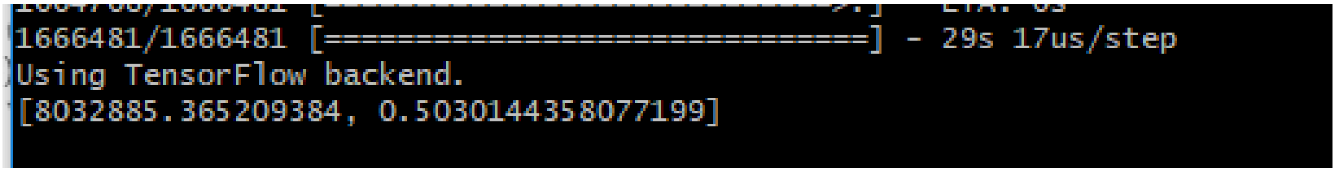


Figure 8: The produced results from our first run of the machine

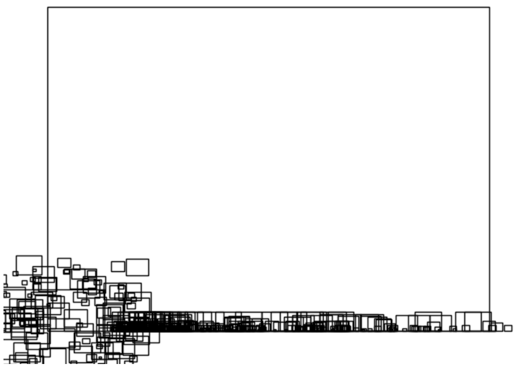

Figure 9: 2 Chips: 0.0021

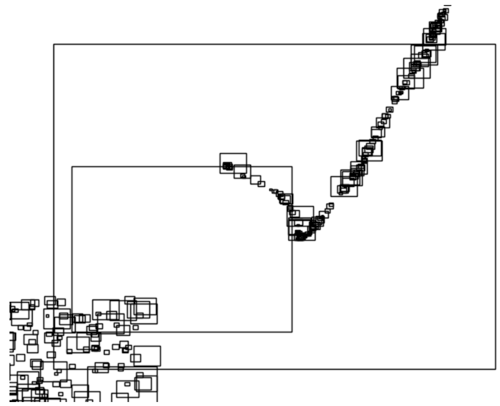

Figure 10: 2 chips: 0.0044

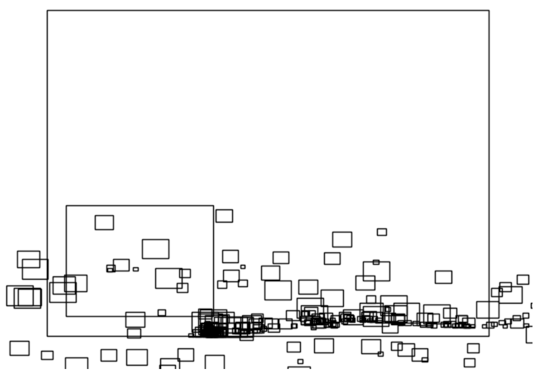

Figure 11: 10 chips: 0.0321

image created, as it indicates increased accuracy from older attempts (Figures 9-11 represent the final placements of macros during both runs with 2 chips and the run with 10 chips). Thus, with some work, we believe that this architecture will improve significantly and produce better, more accurate results.

| Trained Chips | Accuracy Score | Trainable Parameters | Batch Size |
|---|---|---|---|
| 1 | -0.1543 | 8102 | 4 |
| 1 | -0.1072 | 8102 | 32 |
| 1 | -0.9389 | 8102 | 128 |
| 1 | -0.1332 | 8102 | 128 |
| 1 | -0.0322 | 3,171,843 | 128 |
| 2 | 0.0021 | 3,171,843 | 128 |
| 2 | 0.0044 | 3,171,843 | 128 |
| 10 | 0.0321 | 3,171,843 | 128 |
| 1000 | DNF | 3,171,843 | 128 |

### 3.3 Discussion

From the results above, it appears that our hypothesis is not fully supported, as a roughly 50% accuracy from the LSTM RNN and a 3% accuracy from the Actor-Critic model is not anywhere close to what a human designer can do, and so the saved time does not behoove saved money through sub-optimal chip layouts. However, the Actor-Critic does appear to be heading in a better direction, and so we will continue to improve the model.

## 4 Related Work

At the current time, there is actually no real work being done in this area because: A) most of the data is often proprietary information (companies own it, and so they are less likely to put it out for free usage), B) most people working in this field don't have the necessary knowledge of machine learning that a problem such as this would require, and C) most people who do have the machine learning knowledge don't have the data. However, we will continue to look for work that could potentially aid us in our project. While there are multiple uses of these Deep Learning architectures that have been studied, it appears that no one has used them for anything close to chip layout (the closest to our Actor Critic model being the AlphaGo project that used an Actor Critic model to play the strategy game Go).

## 5 Code and Dataset

The original plan was to use a data-set provided by IBM. This would have given us between 200 and 300 chips to evaluate. After looking through the restrictions imposed upon us by using their proprietary data set, we decided to generate our own. We were able to generate similar designs to what a designer at IBM would generate, our generated results were checked by IBM designers. Not only does generating our own data give us flexibility to copy and post our data-set anywhere, it also allows us expand upon the limited size data-set that would have been provided. The tool is able to generate thousands of random chip designs in hours. The generated chips are all approximately what a designer would do, but with just a random number of macros, random sizes, and random connections. This dataset of generated chips was used to train the LSTM RNN, and a single-chip dataset was generated for the Actor Critic.

See the full code in Appendices A to E.

The LSTM dataset is located at this URL address (one line):

https://drive.google.com/file/d/1hFfNmViYhCZ0lzo9nwd2ONwWcxNX9KpD/
view?fbclid=IwAR1rEu-Cbn5jprdUfQRaL6Gm3WuZNPVkjDgdi5R7Pc9DN3Jpu-I_Zttriak

The Actor Critic dataset is located at this URL address:

https://drive.google.com/drive/folders/1UeWr7hIzpjuFuktJGzjW05EXPT9NyLm7?usp=sharing

## 6 Conclusion

Based on our initial findings using the MSE metric, we achieved an accuracy of 50.3%, which did not support our initial hypothesis. We believed this metric to be flawed, but chose to implement a new architecture instead because MSE was inherent to the LSTM RNN structure. Our findings with the Actor Critic model yielded different results, but did not improve on the LSTM RNN's results, though signs appear promising of potential with further testing.

## 7 Bibliography

M. Baccouche, F. Mamalet, C Wolf, C. Garcia, A. Baskurt. Sequential Deep Learning for Human Action Recognition. 2nd International Workshop on Human Behavior Understanding (HBU), A.A. Salah, B. Lepri ed. Amsterdam, Netherlands. pp. 29–39. Lecture Notes in Computer Science 7065. Springer. 2011

Donges, Niklas. “How to Build a Neural Network with Keras.” Towards Data Science, Towards Data Science, 3 Apr. 2018, towardsdatascience.com/how-to-build-a-neural-network-with-keras-e8faa33d0ae4.

“How to Check-Point Deep Learning Models in Keras.” Machine Learning Mastery, 4 Dec. 2018, machinelearningmastery.com/che point-deep-learning-models-keras/.

“Keras: The Python Deep Learning Library.” Home - Keras Documentation, keras.io/.

prottiprotti, et al. “Dimension of Shape in conv1D.” Stack Overflow, stackoverflow.com/questions/43396572/dimension-of-shape-in-conv1d/4339930843399308.

“AlphaGo.” DeepMind, Google, 7 Dec. 2018, deepmind.com/research/alphago/.

Brownlee, Jason. “Save and Load Your Keras Deep Learning Models.” Machine Learning Mastery, 10 Mar. 2018, machinelearningmastery.com/save-load-keras-deep-learning-models/.

“Estimators — TensorFlow Core — TensorFlow.” TensorFlow, www.tensorflow.org/guide/estimators.

Patel, Yash. “Reinforcement Learning w/ Keras + OpenAI: Actor-Critic Models.” Towards Data Science, Towards Data Science, 31 July 2017, towardsdatascience.com/reinforcement-learning-w-keras-openai-actor-critic-models-f084612cfd69.

# 8 Appendix A: Data Generator

```
#import tools used for data creation
import numpy as np
import random
import matplotlib.pyplot as plt
import matplotlib.patches as patches
import operator
import math

#needed for storage
import pickle
import sys
sys.setrecursionlimit(10000)

from dataStructures import *


def is_over_lap_or_out_bounds(rectList, newRect, exteriorRect):
        intersection = False
        buffer = 20 #amount around border
        #check intersection between macros
        for item in rectList:
                intersection = item.is_intersect(newRect, buffer)
                if (intersection):
                        #print ("There is an intersection")
                        return True
        #check if inside bounding box
        #If any of the sides from A are outside of B
        #return true when there is overlap
        if( newRect.get_minx() <= exteriorRect.get_minx() ):
                return True;

        if( newRect.get_miny() <= exteriorRect.get_miny() ):
                return True;

        if( newRect.get_maxx() >= exteriorRect.get_maxx() ):
                return True;

        if( newRect.get_maxy() >= exteriorRect.get_maxy() ):
                return True;

        return intersection

def add_random_blocks(iterations, nameCount):
        for iter in range(iterations):
                min_x = np.random.randint(low=0, high=5000)
                min_y = np.random.randint(low=0, high=5000)
                width = np.random.randint(low=200, high=1000)
                height = np.random.randint(low=200, high=1000)
                max_x = min_x + width
                max_y = min_y + height
                tmp_rect = Rectangle_struct(str(nameCount), min_x, max_x, min_y, max_y )
                nameCount = nameCount + 1
                if (not is_over_lap_or_out_bounds(rect_obj, tmp_rect, extRect)):
                        rect_obj.append(tmp_rect)
#create recantangle

def create_new_rectangle(chipName):
        # build a rectangle in axes coords units in nanometers
        left, width = 0, np.random.normal(5000, 50)
        bottom, height = 0, np.random.normal(5000, 50)
        right = left + width
        top = bottom + height
        extRect = Rectangle_struct("ExteriorRect", left, width, bottom, height)

        buffer_macros = 30

        #Create List of rectangle objects
        rect_obj = []

        #Randomly shrink item either by subtraction of division
        ops = {'-':operator.sub, '/':operator.truediv}
        #randomly choose if square of rectangular
        rec_squ = ['square', 'rectangle']

                #iterate 100 times
        for iter in range(100):
                #other VARIABLES
                nameCount = 0

                #Intitial startup to make smaller
                div_width = np.random.normal(0, 3)
                div_width = abs(div_width)
                if (div_width < 2):
                        div_width = 2
                macro_width = width / div_width
                rec_squ_choice = np.random.choice(rec_squ)
                if (rec_squ_choice == 'square'):
                        macro_height = macro_width
                else:
                        selector = np.random.normal(0, 0.5)
                        #select whether it should be larger or smaller based
                        #on positive or negative number then divide according
                        div = False
                        if (selector < 0):
                                div = True
                        selector = abs(selector)
                        if (selector < 1):
                                macro_height = macro_width
                        else:
                                if (div):
                                        macro_height = macro_width / selector
                                else:
                                        macro_height = macro_width * selector

                currentX = (np.random.randint(low=1, high=int((abs(width - macro_width) + 10) / 10)) * 10)
                currentY = (np.random.randint(low=1, high=int((abs(height - macro_height) + 10) / 10)) * 10)

                topy = currentY + macro_height
```

```
topx = currentX + macro_width

tmp_rect = Rectangle_struct(str(nameCount), currentX, topx, currentY, topy)
nameCount = nameCount + 1
if (not is_over_lap_or_out_bounds(rect_obj, tmp_rect, extRect)):
        rect_obj.append(tmp_rect)

nameCount = nameCount + 1

currentX = (np.random.randint(low=0, high=500) * 10)
currentY = (np.random.randint(low=0, high=500) * 10)

#REMOVE this later
DEBUG = True

#lets loop this sucker
while (macro_width > 100 or macro_height > 100):
        #print("Macro Width: " + str(macro_width))
        #Division is half as likely
        opChoice = np.random.choice(list(ops.keys()))
        #print(opChoice)
        #Subtraction by gausian distribution of half width
        if (opChoice == '-'):
                sub_width = np.random.normal(50, 10)
                if (sub_width < 0):
                        sub_width = 0
                #print("Sub width: " + str(sub_width))
                macro_width = macro_width - sub_width
                rec_squ_choice = np.random.choice(rec_squ)
                if (rec_squ_choice == 'square'):
                        macro_height = macro_width
                else:
                        selector = np.random.normal(0, 0.5)
                        #select whether it should be larger or smaller based
                        #on positive or negative number then divide according
                        div = False
                        if (selector < 0):
                                div = True
                                #print("Divide ")
                        selector = abs(selector)
                        #print("Selector multi: " + str(selector))
                        if (selector < 1):
                                macro_height = macro_width
                        else:
                                if (div):
                                        macro_height = macro_width / selector
                                else:
                                        macro_height = macro_width * selector
        else:
                div_width = np.random.normal(0, 2)
                div_width = abs(div_width)
                if (div_width < 2):
                        div_width = 2
                #print("Divide width: " + str(div_width))
                macro_width = macro_width / div_width
                rec_squ_choice = np.random.choice(rec_squ)
                if (rec_squ_choice == 'square'):
                        macro_height = macro_width
                else:
                        selector = np.random.normal(0, 0.5)
                        #select whether it should be larger or smaller based
                        #on positive or negative number then divide according
                        div = False
                        if (selector < 0):
                                div = True
                        selector = abs(selector)
                        if (selector < 1):
                                macro_height = macro_width
                        else:
                                if (div):
                                        macro_height = macro_width / selector
                                else:
                                        macro_height = macro_width * selector

        #Random number created
        shape_multiplier = np.random.normal(0, 2)
        shape_multiplier = abs(shape_multiplier)
        #Round to nearest integer
        shape_multiplier = round(shape_multiplier)
        #make even by multiplying by 2
        shape_multiplier = 2 * int(shape_multiplier)

        #Test Squares so set shape multiplier to 4
        shape_multiplier = 20

        #if its less than 0.5 then make it 1
        #check if perfect sqaure
        if (shape_multiplier <= 1):
                shape_multiplier = 1
                is_square = False
        elif (int(shape_multiplier**0.5)**2 == int(shape_multiplier)):
                is_square = True
        else:
                is_square = False

        #print("shape_multiplier: " + str(shape_multiplier))
        #Intial locations for squre of macros
        squareCounter = 0
        square_x_og_location = currentX
        square_y_og_location = currentY
        first_time_y_9 = True
        first_time_y_16 = True
        first_time = True

        #REMOVE THIS AFTER TESTING
        if (DEBUG):
                macro_height = 300
                macro_width = 300
                DEBUG = False

        #Decide number of columns
        num_columns = np.random.randint(low=1, high=8)
```

```
            #num_rows round up of total / columns
            num_rows = math.ceil(shape_multiplier / num_columns)

            #starts at one since updated after placement
            row_count = 1
            column_count = 0


            for x in range(shape_multiplier):
                if (is_square):
                    #Create Sub Rectangles
                    topy = currentY + macro_height
                    topx = currentX + macro_width
                    tmp_rect = Rectangle_struct(str(nameCount), currentX, topx, currentY, topy)
                    nameCount = nameCount + 1
                    if (not is_over_lap_or_out_bounds(rect_obj, tmp_rect, extRect)):
                        rect_obj.append(tmp_rect)
                    if (squareCounter == 0):
                        currentX = currentX + macro_width + buffer_macros
                    elif (squareCounter == 1):
                        currentY = currentY + macro_height + buffer_macros
                        currentX = currentX - macro_width - buffer_macros
                    elif (squareCounter == 2):
                        currentX = currentX + macro_width + buffer_macros
                    elif (squareCounter == 3):
                        currentX = square_x_og_location
                        currentY = currentY + macro_height + buffer_macros
                        #need to iterate + 1 to make my life easier later so its not leading
                        squareCounter = squareCounter + 1
                    if ((squareCounter**0.5) > 2 and (squareCounter**0.5) < 3):
                        if ((squareCounter**0.5) < 2.6):
                            currentX = currentX + macro_width + buffer_macros
                        else:
                            if (first_time_y_9):
                                currentY = square_y_og_location
                                first_time_y_9 = False
                            else:
                                currentY = currentY + macro_height + buffer_macros
                    elif ((squareCounter**0.5) >= 3 and (squareCounter**0.5) <= 4):
                        if (first_time):
                            currentX = square_x_og_location
                            currentY = square_y_og_location + 3 * ( macro_height + buffer_macros )
                            first_time = False
                        elif ((squareCounter**0.5) < 3.6):
                            currentX = currentX + macro_width + buffer_macros
                        else:
                            if (first_time_y_16):
                                currentY = square_y_og_location
                                first_time_y_16 = False
                            else:
                                currentY = currentY + macro_height + buffer_macros
                else:
                    #Create Sub Rectangles
                    topy = currentY + macro_height
                    topx = currentX + macro_width
                    tmp_rect = Rectangle_struct(str(nameCount), currentX, topx, currentY, topy)
                    nameCount = nameCount + 1
                    if (not is_over_lap_or_out_bounds(rect_obj, tmp_rect, extRect)):
                        rect_obj.append(tmp_rect)

                    if ( row_count < num_rows):
                        currentX = currentX + macro_width + buffer_macros
                        row_count = row_count + 1
                    else:
                        if ( column_count < num_columns):
                            currentY = currentY + macro_height + buffer_macros
                            column_count = column_count + 1
                            #reset rows for next column
                            currentX = square_x_og_location
                            row_count = 1
                squareCounter = squareCounter + 1
            currentX = (np.random.randint(low=0, high=500) * 10)
            currentY = (np.random.randint(low=0, high=500) * 10)

    #get number of objects in list
    numRectObj = len(rect_obj)

    #Create Random Connections
    for item in rect_obj:
        #Generate random number of connections
        num_connections = np.random.randint(low=1, high=(numRectObj/10))
        #Generate random sample of equal distance
        tmp_sample = random.sample(rect_obj, num_connections)
        finale_sample = tmp_sample
        #See Distance and determine if its a good one to use
        for connection in tmp_sample:
            #Calulate distance between item and connection
            distance = ( (connection.get_middle_x() - item.get_middle_x())**2
                + (connection.get_middle_y() - item.get_middle_y())**2 )**0.5
            #Determine how likely it would be connected
            max_distance = abs(np.random.normal((5000/1.5), 2000))
            if (max_distance < distance):
                finale_sample.remove(connection)
        item.set_connect(tmp_sample)
    #return the chip to be stored
    print("Made chip: " + chipName)
    return Chip(chipName, height, width, rect_obj)

#Create array to store in pickle
chip_array = []
for i in range (1000):
    chip_name = "chip_" + str(i)
    tmpChip = create_new_rectangle(chip_name)
    chip_array.append(tmpChip)
#store object in pickle Array of rectangles
pickle.dump( chip_array, open( "save_chips_1000.p", "wb" ) )
```

# 9 Appendix B: Data Structure Format for LSTM RNN

```python
import numpy as np
import random
import matplotlib.pyplot as plt
import matplotlib.patches as patches
import operator
import math

class Chip:
	def __init__(self, name, height, width, rectangle_list):
		self.name = name
		self.height = height
		self.width = width
		self.rectangle_list = rectangle_list

	def get_name(self):
		return self.name

	def get_height(self):
		return self.height

	def get_width(self):
		return self.width

	def get_rectangle_list(self):
		return self.rectangle_list

	def print_plot(self):
		#Setup the figure
		fig = plt.figure(self.name)
		ax = fig.add_axes([0,0,1,1])

		#Determine which is larger height or width and use it for making square axis
		if (self.width > self.height):
			image_buf = self.width * 0.1
			outerEdge = self.width
		else:
			image_buf = self.height * 0.1
			outerEdge = self.height

		#Change axis to view height and width plus buffer
		ax.set_xlim(0-image_buf, outerEdge+image_buf)
		ax.set_ylim(0-image_buf, outerEdge+image_buf)

		# axes coordinates are 0,0 is bottom left and 1,1 is upper right
		p = patches.Rectangle(
			(0, 0), self.width, self.height,
			fill=False, clip_on=False
			)

		ax.add_patch(p)

		for item in self.rectangle_list:
			minx = item.get_minx()
			miny = item.get_miny()
			maxx = item.get_maxx()
			maxy = item.get_maxy()
			width = maxx - minx
			height = maxy - miny
			p = patches.Rectangle((minx, miny), width, height,
				fill=False, clip_on=False)
			#ax.text((minx+width/2), (miny+height/2), item.get_name())
			ax.add_patch(p)
			#plot lines
			#get connections for item
			connect = item.get_connect()
		ax.set_axis_off()
		return ax
			#plt.show()


class Rectangle_struct:
	def __init__(self, name, min_x, max_x, min_y, max_y):
		self.name = name
		self.min_x = min_x
		self.max_x = max_x
		self.min_y = min_y
		self.max_y = max_y
		self.middle = (((min_x + max_x) / 2), ((min_y + max_y) / 2))
		self.connect = []

	def is_intersect(self, other, buffer):
		if (self.min_x < (other.max_x + buffer) and (self.max_x + buffer) > other.min_x
			and self.min_y < (other.max_y + buffer) and (self.max_y + buffer) > other.min_y):
			#print("Intersection between: " + str(self.name) + " and " + str(other.name))
			return True
		return False

	def get_width(self):
		return (self.max_x - self.min_x)

	def get_height(self):
		return (self.max_y - self.min_y)

	def get_minx(self):
		return self.min_x

	def get_maxx(self):
		return self.max_x

	def get_miny(self):
		return self.min_y

	def get_maxy(self):
		return self.max_y

	def get_middle(self):
		return self.middle
```

```python
    def get_name(self):
        return self.name

    def set_connect(self, listToSet):
        self.connect = listToSet

    def get_middle_x(self):
        return self.middle[0]

    def get_middle_y(self):
        return self.middle[1]

    def get_connect(self):
        return self.connect
```

# 10 Appendix C: Recurrent Neural Network Model

```
from keras import applications
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Flatten
from keras.layers import Conv1D, MaxPooling1D, LeakyReLU, PReLU
from keras.callbacks import CSVLogger, ModelCheckpoint

#try new LSTM
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.layers import Embedding
from keras.layers import LSTM

#Stuff needed for dataset
import numpy as np
import random
import matplotlib.pyplot as plt
import matplotlib.patches as patches
import operator
import math

#needed for storage
import pickle
import sys
sys.setrecursionlimit(10000)

from dataStructures import *

chip_array = pickle.load( open( "save_chips_1000.p", "rb" ) )

#make sure we have data
#Uncomment to print
initial_print = False
if (initial_print):
        for item in chip_array:
                item.print_plot()


        num_rectangles = 0
        #Get average number of rectangles
        for item in chip_array:
                rect_list = item.get_rectangle_list()
                tmp_rect_len = len(rect_list)
                num_rectangles = num_rectangles + tmp_rect_len

        average_num_rect = num_rectangles / len(chip_array)
        plt.show()

#calculate value for test and training
#percentage for testing
perc_test = 0.8
total_entries = 0
test_num = int(len(chip_array) * perc_test)
#Setup Data with inputs as data and outputs as labels
#First level is chip
#Second level of array name, width, height, connection,
#         bounds_xmin, bounds_ymin, bounse_xmax, bounds_ymax
#second level for label is x cord, y cord
train_datas = []
train_labels = []
for chip in chip_array[:test_num]:
        #get rectangles
        rect_list = chip.get_rectangle_list()
        for rectangle in rect_list:
                #get connections for each rectangle
                tmp_connection_list = rectangle.get_connect()
                for connection in tmp_connection_list:
                        train_datas.append([rectangle.get_name(), rectangle.get_width(),
                                rectangle.get_height(), connection.get_name(), 0, 0,
                                chip.get_width(), chip.get_height()])
                        train_labels.append([rectangle.get_minx(), rectangle.get_miny()])

verf_datas = []
verf_labels = []
for chip in chip_array[test_num:]:
        #get rectangles
        rect_list = chip.get_rectangle_list()
        for rectangle in rect_list:
                #get connections for each rectangle
                tmp_connection_list = rectangle.get_connect()
                for connection in tmp_connection_list:
                        verf_datas.append([rectangle.get_name(), rectangle.get_width(),
                                rectangle.get_height(), connection.get_name(), 0, 0,
                                chip.get_width(), chip.get_height()])
                        verf_labels.append([rectangle.get_minx(), rectangle.get_miny()])


np_train_datas = np.array(train_datas)
np_train_labels = np.array(train_labels)
np_verf_datas = np.array(verf_datas)
np_verf_labels = np.array(verf_labels)
#shape of data
#since variable size step_size is none, features is width of data
step_size = 8
nb_features = len(train_datas[0])
batch_size = 128
epochs = 200

print(np_train_datas.shape[0])
print(np_verf_datas.shape[0])


#For LSTM
np_train_datas = np.reshape(np_train_datas,
        (np_train_datas.shape[0], 1, np_train_datas.shape[1]))
np_train_labels = np.reshape(np_train_labels,
        (np_train_labels.shape[0], 1, np_train_labels.shape[1]))
np_verf_datas = np.reshape(np_verf_datas,
        (np_verf_datas.shape[0], 1, np_verf_datas.shape[1]))
```

```
np_verf_labels = np.reshape(np_verf_labels,
        (np_verf_labels.shape[0], 1, np_verf_labels.shape[1]))

model = Sequential()

#model.add(Embedding(max_features, output_dim=256)) Shouldn't need this since its all standardized before

model.add(LSTM(8, input_shape=(1,8), return_sequences=True))
model.add(Dropout(0.5))
#2 is number of output layers for Dense
model.add(Dense(2, activation='sigmoid'))

model.summary()
model.compile(loss='mse',
              optimizer='nadam',
              metrics=['accuracy'])

model.fit(np_train_datas, np_train_labels, batch_size=1000, epochs=10)
score = model.evaluate(np_verf_datas, np_verf_labels, batch_size=128)

print(score)
```

# 11 Appendix D: Data Structure Format for Actor Critic

```
import numpy as np
import random
import matplotlib.pyplot as plt
import matplotlib.patches as patches
import operator
import math

class Chip:
        def __init__(self, name, height, width, rectangle_list):
                self.name = name
                self.height = height
                self.width = width
                self.rectangle_list = rectangle_list

        def get_name(self):
                return self.name

        def get_height(self):
                return self.height

        def get_width(self):
                return self.width

        def get_rectangle_list(self):
                return self.rectangle_list

        def update_rectangle_overlaps(self):
                compare_list = self.rectangle_list
                for rectangle in self.rectangle_list:
                        num_overlaps = 0
                        for comp_rect in compare_list:
                                if (rectangle.is_intersect(comp_rect, 10)):
                                        num_overlaps = num_overlaps + 1
                        #Need to update the actual list not just the retangle copy
                        indexTmp = self.rectangle_list.index(rectangle)
                        self.rectangle_list[indexTmp].set_num_overlap(num_overlaps)

        def update_rectangle_list_xy(self, x_cord, y_cord, iter):
                self.rectangle_list[iter].set_lower_left(x_cord, y_cord)

        def update_rectangle_list_x_max_y_max(self, x_bounds, y_bounds, iter):
                self.rectangle_list[iter].set_bounds(x_bounds, y_bounds)

        def get_num_macros(self):
                return len(self.rectangle_list)

        def get_chip_score(self):
                rectangle_list = self.get_rectangle_list()
                total_score = 0
                for rectangle in rectangle_list:
                        total_score = total_score + rectangle.calculate_score()
                return total_score

        #Print a chip into a graphic
        def print_plot(self):
                #Setup the figure
                fig = plt.figure(self.name)
                ax = fig.add_axes([0,0,1,1])

                #Determine which is larger height or width and use it for making square axis
                if (self.width > self.height):
                        image_buf = self.width * 0.1
                        outerEdge = self.width
                else:
                        image_buf = self.height * 0.1
                        outerEdge = self.height

                #Change axis to view height and width plus buffer
                ax.set_xlim(0-image_buf, outerEdge+image_buf)
                ax.set_ylim(0-image_buf, outerEdge+image_buf)

                # axes coordinates are 0,0 is bottom left and 1,1 is upper right
                p = patches.Rectangle(
                        (0, 0), self.width, self.height,
                        fill=False, clip_on=False
                        )

                ax.add_patch(p)

                for item in self.rectangle_list:
                        minx = item.get_minx()
                        miny = item.get_miny()
                        maxx = item.get_maxx()
                        maxy = item.get_maxy()
                        width = maxx - minx
                        height = maxy - miny
                        p = patches.Rectangle((minx, miny), width, height,
                                fill=False, clip_on=False)
                        ax.add_patch(p)
                        #plot lines
                        #get connections for item
                        connect = item.get_connect()
                        ax.set_axis_off()
                return ax


class Rectangle_struct:
        def __init__(self, name, min_x, max_x, min_y, max_y):
                self.name = name
                self.min_x = min_x
                self.max_x = max_x
                self.min_y = min_y
                self.max_y = max_y
                self.middle = (((min_x + max_x) / 2), ((min_y + max_y) / 2))
                self.height = max_y - min_y
                self.width = max_x - min_x
                self.lower_left = (min_x, min_y)
                self.connect = []
```

```
        self.num_overlap = 0
        self.score = 0
        self.bounds_x = 0
        self.bounds_y = 0

    #Does the rectangle strucutre intersect with others part
    def is_intersect(self, other, buffer):
        if (self.min_x < (other.max_x + buffer) and (self.max_x + buffer) > other.min_x
                and self.min_y < (other.max_y + buffer) and (self.max_y + buffer) > other.min_y):
                    return True
        return False

    def get_midpoint_dist(self, other):
        distance = ((self.middle[0] - other.middle[0])**2 + (self.middle[1] - other.middle[1])**2)**0.5
        return distance

    def get_width(self):
        return (self.max_x - self.min_x)

    def get_height(self):
        return (self.max_y - self.min_y)

    def get_minx(self):
        return self.min_x

    def get_maxx(self):
        return self.max_x

    def get_miny(self):
        return self.min_y

    def get_maxy(self):
        return self.max_y

    def get_middle(self):
        return self.middle

    def get_name(self):
        return self.name

    def get_middle_x(self):
        return self.middle[0]

    def get_middle_y(self):
        return self.middle[1]

    def get_connect(self):
        return self.connect

    def get_num_overlap(self):
        return self.num_overlap

    def get_total_conn_dist(self):
        connectList = self.get_connect()
        distance = 0
        for connection in connectList:
                distance = distance + self.get_midpoint_dist(connection)
        return distance

    #Set Data structure values
    #Want to restructure to make macros that are x,y points and then height and width
    #stored or at least just add that functionality
    def set_bounds(self, x_bound, y_bound):
        self.bounds_x = x_bound
        self.bounds_y = y_bound

    def set_num_overlap(self, overlaps):
        self.num_overlap = overlaps

    def set_connect(self, listToSet):
        self.connect = listToSet

    def set_lower_left(self, x, y):
        #When reseting lower left coordinate need to set
        #the midpoints as well as the upper left hand corner
        height = self.get_height()
        width = self.get_width()
        self.min_x = x
        self.min_y = y
        self.max_x = (x + width)
        self.max_y = (y + height)
        self.middle = (((self.min_x + self.max_x) / 2), ((self.min_y + self.max_y) / 2))

    def calculate_score(self):
        #Set the points for out of bounds to million
        pts_bounds = 10000
        distance = self.get_total_conn_dist()
        num_overlaps = self.get_num_overlap()
        score = (distance / 100) +  num_overlaps * 1000
        #Calculate if rectangle is in the bounds of chip add points accordingly
        if (self.bounds_x < self.min_x):
                score = score + pts_bounds
        if (self.bounds_y < self.min_y):
                score = score + pts_bounds
        if (0 > self.min_x):
                score = score + pts_bounds
        if (0 > self.min_y):
                score = score + pts_bounds
        self.score = score
        return score
```

# 12 Appendix E: Actor Critic Model

```
### Import Everything needed
from dataStructures import *

import numpy as np
from keras.models import Sequential, Model, load_model
from keras.layers import Dense, Dropout, Input
from keras.layers.merge import Add, Multiply
from keras.optimizers import Adam
import keras.backend as K

import tensorflow as tf

import random
from collections import deque

#needed for storage
import pickle
import sys
sys.setrecursionlimit(10000)

class ActorCritic:
	#In Init create the actor critic models (need to go through this to make sure its created right)
	def __init__(self, chip, sess, state):
		self.chip = chip
		self.sess = sess
		self.action = np.array([[0, 0]])

		self.score = 0
		self.num_macros = chip.get_num_macros()

		self.learning_rate = 0.001
		self.epsilon = 1.0
		self.epsilon_decay = .995
		self.gamma = .95
		self.tau   = .125

		self.cur_state = state

		# ===================================================================== #
		#                               Actor Model                             #
		# Chain rule: find the gradient of chaging the actor network params in  #
		# getting closest to the final value network predictions, i.e. de/dA    #
		# Calculate de/dA as = de/dC * dC/dA, where e is error, C critic, A act #
		# ===================================================================== #
		#Don't know what this does
		#Maybe remember how far back it needs to remember
		self.memory = deque(maxlen=5000)
		self.actor_state_input, self.actor_model = self.create_actor_model()
		_, self.target_actor_model = self.create_actor_model()

		#Feed in the number of actions should be 2 columns
		self.actor_critic_grad = tf.placeholder(tf.float32,
			[None, 2]) # where we will feed de/dC (from critic)

		actor_model_weights = self.actor_model.trainable_weights
		#made none negative since I do want it to be the minimum
		self.actor_grads = tf.gradients(self.actor_model.output,
			actor_model_weights, self.actor_critic_grad) # dC/dA (from actor)
		grads = zip(self.actor_grads, actor_model_weights)
		self.optimize = tf.train.AdamOptimizer(self.learning_rate).apply_gradients(grads)

		# ===================================================================== #
		#                              Critic Model                             #
		# ===================================================================== #

		self.critic_state_input, self.critic_action_input, \
			self.critic_model = self.create_critic_model()
		_, _, self.target_critic_model = self.create_critic_model()

		self.critic_grads = tf.gradients(self.critic_model.output,
			self.critic_action_input) # where we calcaulte de/dC for feeding above

		# Initialize for later gradient calculations
		self.sess.run(tf.global_variables_initializer())

	# ===================================================================== #
	#                            Calcualte Score                            #
	# ===================================================================== #

	def calc_score(self, buffer_size):
		#Iterate through all rectangles
		macroList = self.chip.get_rectangle_list()
		#reset score before calculating
		self.score = 0
		for macro in macroList:
			#Calculate Distance
			connectionList = macro.get_connect()
			#iterate through all connection centers to get total score
			for connection in connectionList:
				scoreAdd = macro.get_midpoint_dist(connection)
				self.score = self.score + scoreAdd

			#Determine if overlap and if there is add 0 points N^2 ouch...
			for macro_compare in macroList:
				#Remove possibility of comparing same macro because duh they overlap
				if (macro.is_intersect(macro_compare, buffer_size)):
					if (macro_compare != macro):
						self.score = self.score + 100000


	def get_score(self):
		return self.score



	# ========================================================================= #
	#                             Model Definitions                             #
	# ========================================================================= #
```

```
    #Need to figure out state_input and output DONE And for now is good?
    def create_actor_model(self):
        #Create observation space see webpage for details
        #Currently thinking (x, y) lower left height and width of macro plus all other macros, and chip bounds
        #Do it within the data structure
        #Shape will be (num_rectangles + bounds, 4)
        #where bounds is frist followed by macros each one needs max_x, max_y that x and y can't exceed
        #[[ x, y, width, height, max_x, max_y, totConnection, overlaps, score ], [ x, y, width, height, max_x, max_y, totConnecti
        #[ x, y, width, height, max_x, max_y, totConnection ...]... ]
        state_input = Input(shape=(9,))
        #try with high density first trying 4096, going down from there
        h1 = Dense(1024, activation='relu')(state_input)
        h2 = Dense(1024, activation='relu')(h1)
        h3 = Dense(512, activation='relu')(h2)
        #need dense for output which is the action in our case move 10, 100, 500, 1000 units in any direction maybe?
        #Should only be two numbers THIS NEEDS TO CHANGE TO ACCOMDATE ALL THE OUTPUTs?
        output = Dense(2, activation='relu')(h3)

        model = Model(input=state_input, output=output)
        adam  = Adam(lr=0.001)
        model.compile(loss="mse", optimizer=adam)
        model.summary()
        return state_input, model
    #Need to figure out state_input and output
    def create_critic_model(self):
        ##[[ x, y, width, height, max_x, max_y, totConnection], [ x, y, width, height, max_x, max_y, totConnect],... ] see above
        state_input = Input(shape=(9,))
        state_h1 = Dense(1024, activation='relu')(state_input)
        state_h2 = Dense(1024)(state_h1)

        #8 actions to input so make that shape.. Does this need to be 2 , None?
        action_input = Input(shape=(2,))
        action_h1    = Dense(1024)(action_input)

        merged    = Add()([state_h2, action_h1])
        merged_h1 = Dense(512, activation='relu')(merged)
        #This is going to be the score I believe
        output = Dense(1, activation='relu')(merged_h1)
        model  = Model(input=[state_input,action_input], output=output)

        adam  = Adam(lr=0.001)
        model.compile(loss="mse", optimizer=adam)
        model.summary()
        return state_input, action_input, model

    # ========================================================================= #
    #                              Save Models                                  #
    # ========================================================================= #

    def save_models(self):
        #Save all keras models
        self.critic_model.save('critic_model.h5')
        self.target_critic_model.save('target_critic_model.h5')
        self.target_actor_model.save('target_actor_model.h5')
        self.actor_model.save('actor_model.h5')

        #Save the tensorflow model
        saver = tf.train.Saver()
        self.saveMemory()
        saver.save(self.sess, './model/tf_model')

    def load_models(self):
        saver = tf.train.Saver()
        saver.restore(self.sess, './model/tf_model')
        self.critic_model = load_model('critic_model.h5')
        self.target_critic_model = load_model('target_critic_model.h5')
        self.target_actor_model = load_model('target_actor_model.h5')
        self.actor_model = load_model('actor_model.h5')


    # ========================================================================= #
    #                              Model Training                               #
    # ========================================================================= #

    def remember(self, cur_state, action, reward, new_state, done):
        self.memory.append([cur_state, action, reward, new_state, done])

    def _train_actor(self, samples, idx):
        for sample in samples:
            cur_state, action, reward, new_state, _ = sample
            #print(cur_state)
            predicted_action = self.actor_model.predict(cur_state[idx])
            grads = self.sess.run(self.critic_grads, feed_dict={
                self.critic_state_input:  cur_state[idx],
                self.critic_action_input: predicted_action
            })[0]

            self.sess.run(self.optimize, feed_dict={
                self.actor_state_input: cur_state[idx],
                self.actor_critic_grad: grads
            })

    def _train_critic(self, samples, idx):
        for sample in samples:
            cur_state, action, reward, new_state, done = sample
            if not done:
                #if (len(new_state) > idx):
                #       idx = len(new_state) - 100
                target_action = self.target_actor_model.predict(new_state[idx])
                #THIS USED TO BE TARGET_ACTION so change it back if you want the standard
                future_reward = self.target_critic_model.predict(
                    [new_state[idx], target_action])[0][0]
                #future_reward = random.randint(-1000,1000)
                reward += self.gamma * future_reward
            reward = np.array([reward])
            #print("Reward: ", reward)
            self.critic_model.fit([cur_state[idx], action], reward, verbose=0)

    def train(self, idx):
        batch_size = 128
        if len(self.memory) < batch_size:
            return
```

```python
            rewards = []
            samples = random.sample(self.memory, batch_size)
            self._train_critic(samples, idx)
            self._train_actor(samples, idx)

    # ========================================================================= #
    #                          Target Model Updating                            #
    # ========================================================================= #

    def _update_actor_target(self):
        actor_model_weights  = self.actor_model.get_weights()
        actor_target_weights = self.target_critic_model.get_weights()

        for i in range(len(actor_target_weights)):
            actor_target_weights[i] = actor_model_weights[i]
        self.target_critic_model.set_weights(actor_target_weights)

    def _update_critic_target(self):
        critic_model_weights  = self.critic_model.get_weights()
        critic_target_weights = self.critic_target_model.get_weights()

        for i in range(len(critic_target_weights)):
            critic_target_weights[i] = critic_model_weights[i]
        self.critic_target_model.set_weights(critic_target_weights)

    def update_target(self):
        self._update_actor_target()
        self._update_critic_target()

    # ========================================================================= #
    #                              Model Predictions                            #
    # ========================================================================= #

    def act(self, cur_state, macro_index):
        #pick a current chip (but maybe infuture move more than one?)
        self.epsilon *= self.epsilon_decay
        if np.random.random() < self.epsilon:
            #create random action
            #Should return something like this
            # [ [14, 45]] where x and y can be positive or negative
            return self.randomAct()
        return self.actor_model.predict(cur_state[macro_index])

    def randomAct(self):
        x = random.randint(-1000,1000)
        y = random.randint(-1000,1000)
        return np.array([[x, y]])

    def update_state(self, action, index):
        #get curr_stateiterator
        old_score = self.cur_state[index][0][8]
        #change x by action x
        #print("x before: ", self.cur_state[index][0])
        #First is index, second is inside [[ ]] then its the actuall array which is [0]
        self.cur_state[index][0][0] = self.cur_state[index][0][0] + action[0][0]
        #print("x after: ", self.cur_state[index][0])
        #change y by action y
        self.cur_state[index][0][1] = self.cur_state[index][0][1] + action[0][1]
        #Set x and y in the chip
        self.chip.update_rectangle_list_xy(self.cur_state[index][0][0], self.cur_state[index][0][1], index)
        #Update Overlaps in the chip
        self.chip.update_rectangle_overlaps()
        #Pull the new score
        score = self.chip.rectangle_list[index].calculate_score()
        #print("Score: ", score)
        self.cur_state[index][0][8] = score
        #WNAT TO MINIMIZE (I think check this later)
        score_dif = self.cur_state[index][0][8] - old_score
        return self.cur_state, score_dif

    def saveMemory(self):
        pickle.dump( self.memory, open( "save_memory_2000.p", "wb" ) )

    def loadMemory(self):
        self.memory = pickle.load( open( "save_memory_2000.p", "rb" ) )


def createStructure(chip):
    rectangleList = chip.get_rectangle_list()
    envirorment = []
    #[ x, y, width, height, max_x, max_y, totConnection ...]
    for count, rectangle in enumerate(rectangleList):
        #x = rectangle.get_minx()
        #y = rectangle.get_miny()
        x = -10
        y = -10
        width = rectangle.get_width()
        height = rectangle.get_height()
        max_x = chip.get_width() - width
        max_y = chip.get_height() - height
        connectDistance = rectangle.get_total_conn_dist()
        overlaps = rectangle.get_num_overlap()
        #Update rectangle with 0,0 and max_y max_x cord possible cord
        chip.update_rectangle_list_xy(x,y, count)
        chip.update_rectangle_list_x_max_y_max(max_x, max_y, count)

        score = rectangle.calculate_score()
        rectValue = np.array([[x, y, width, height, max_x, max_y, connectDistance, overlaps, score]])
        envirorment.append(rectValue)
    return envirorment, chip

def get_linear_score(ideal_score, intial_score, current_score):
    return (intial_score - current_score)/(intial_score - ideal_score)

def update_xy_max(chip):
    rectangleList = chip.get_rectangle_list()
    chip.update_rectangle_overlaps()
    for count, rectangle in enumerate(rectangleList):
        width = rectangle.get_width()
        height = rectangle.get_height()
        max_x = chip.get_width() - width
```

```
            max_y = chip.get_height() - height
            chip.update_rectangle_list_x_max_y_max(max_x, max_y, count)

def main():
        #Create Tensor flow session
        sess = tf.Session()
        K.set_session(sess)
        #create chip enviorment
        chip_array = pickle.load( open( "save_chips_10.p", "rb" ) )
        actor_critic_array = []
        #Get array of scores
        ideal_score_array = []
        post_score_array = []

        #Need to update the rectanlge list for x_max_y_max before hand
        for iterator, chip in enumerate(chip_array):
                update_xy_max(chip_array[iterator])

        #View Plot Before
        plot_name = "test_base_"
        chip_array[0].print_plot()
        plt.savefig(plot_name)
        plt.close()
        #print("Before: ", chip_array[0].get_chip_score())
        for iterator, chip in enumerate(chip_array):
                #Store Ideal Score
                ideal_score_array.append(chip.get_chip_score())
                #Create Envirorment for all chips
                cur_state, chip_array[iterator] = createStructure(chip)
                chip_array[iterator].update_rectangle_overlaps()
                actor_critic = ActorCritic(chip_array[iterator], sess, cur_state)
                actor_critic_array.append(actor_critic)


        #How many times we should move the macro when
        #iterating through all macros on chip
        move_single_mac = 3
        count = 0
        #while count < 1:
        current_best_linear_score = 0
        read_to_load = False
        for iterator, chip in enumerate(chip_array):
                actor_critic = actor_critic_array[iterator]
                cur_state,_ = createStructure(chip)
                if (read_to_load):
                        actor_critic.load_models()
                #cur_state = cur_state.reshape((1, env.shape[0]))
                #print(cur_state)
                #print(env.observation_space.shape)
                #action = action.reshape((1, env.action_space.shape[0]))
                #Take selected Action here
                #Iterate through all macros to move initially
                initial_score = actor_critic.chip.get_chip_score()
                for idx, macro in enumerate(chip.get_rectangle_list()):
                        for x in range(0, move_single_mac):
                                action = actor_critic.act(cur_state, idx)
                                #Need to mimic this below
                                score_change = 0
                                #new_state, reward, done, _ = env.step(action)
                                new_state, score_change = actor_critic.update_state(action, idx)
                                reward = score_change
                                done = 0
                                #new_state = new_state.reshape((1, env.observation_space.shape[0]))
                                #print("Action: ", action)
                                #print("Reward: ", reward)
                                actor_critic.remember(cur_state, action, reward, new_state, done)
                                actor_critic.train(idx)

                                #need to seperate but good start
                                cur_state = new_state

                        #for testing for now pick one item with highest score then once it doesn't improve after
                        #50 moves move on OR DO THIS EVERY 100 MOVES NEED TO FIGURE OUT
                        curMaxScore = 0
                        curMaxIter = 0

                #Iterate through the worst 100 to try to improve the score
                number =0
                while (number < 100):
                        curMaxScore = 0
                        for idx, rectangle in enumerate(cur_state):
                                if rectangle[0][8] > curMaxScore:
                                        curMaxScore = rectangle[0][8]
                                        curMaxIter = idx

                        #Change worst one
                        action = actor_critic.act(cur_state, idx)
                        #Need to mimic this below
                        score_change = 0
                        #new_state, reward, done, _ = env.step(action)
                        new_state, score_change = actor_critic.update_state(action, idx)
                        reward = score_change
                        done = 0
                        actor_critic.remember(cur_state, action, reward, new_state, done)
                        actor_critic.train(idx)
                        number = number + 1

                #need to seperate but good start
                cur_state = new_state
                #print(count)

                #Update the actor critic session and chip
                actor_critic_array[iterator] = actor_critic
                chip_array[iterator] = chip

                #get linear score chip_array[iterator].get_chip_score() = after trained score
                #ideal_score_array[iterator] = ideal score of original data
                #
                linear_score = get_linear_score(ideal_score_array[iterator], initial_score, chip_array[iterator].get_chip_score())
                if (linear_score > current_best_linear_score):
                        count = count + 1
                        read_to_load = True
```

```
                    plot_name = "new_best_" + str(count)
                    actor_critic.chip.print_plot()
                    plt.savefig(plot_name)
                    plt.close()
                    current_best_linear_score = linear_score
                    #Figure Out how to save the models here..
                    actor_critic.save_models()


        actor_critic.saveMemory()
        actor_critic.chip.print_plot()


if __name__ == "__main__":
        main()
```