Contents

A DEEP LEARNING APPROACH TO THE UNKNOTTING PROBLEM WITH MOSAIC DIAGRAMS

JESSICA CHILDRESS

Abstract. Mosaic knot theory creates a way to represent knots using square $n \times n$ diagrams. In such a tabular form, knots can easily be encoded as 2D matrices. In this paper, we seek to develop a solution to the unknotting problem using a convolutional neural network (CNN) which specializes in working with 2D matrices as input data. We provide details on the architecture of a CNN including some of the mathematical functions involved in various layers. We then discuss the results of our high-performing CNN models, in particular, we compare each model's accuracy on the testing set to its accuracy on 400 instances that were not included in the original data set. We conclude with ideas for future work and a brief reflection on the impact of this project.

1. Background: Knot Theory

A *knot* or *component* is defined as a closed loop in 3-space. A *link* is a union of two or more disjoint components. The *unknot* consists of a single component with zero crossings in its simplest form and is depicted in Figure 1. In this paper, we explore neural network applications to the *unknotting problem* which seeks to find an algorithm that can determine if any knot diagram represents the unknot. In his final publication in 1954 titled *Solvable and Unsolvable Problems*, Alan Turing remarked:"No systematic method is yet known by which one can tell whether two knots are the same...A decision problem that might

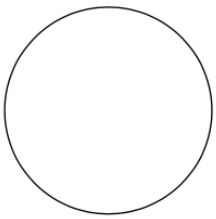

FIGURE 1. An unknot

FIGURE 2. The 11 tiles used to construct mosaic diagrams

well be unsolvable." [9] The first solution was found by Haken in 1961. Since then, other algorithms have been developed by Birman-Hirsch in 1998, Haas-Lagarias in 2001, and Manolescu, Ozsváth & Sarkar 2009.

The field of mosaic knot theory originates from the work of Lomonaco and Kauffman in 2008 [8]. A *mosaic knot* is a representation of a knot or link on an $n \times n$ grid using the mosaic tiles from Figure 2. Although originally defined as a tool to study quantum knots, Kuriya and Shehab showed in 2014 that mosaic knot theory is equivalent to tame knot theory [6]. We will be using square $n \times n$ mosaic diagrams in our research to find a machine learning model whose algorithm can be used to solve the unknotting problem.

In the early 20th century, a German mathematician named Kurt Reidemeister rigorously proved that there exists knots distinct from the unknot [4]. He also expressed the manipulation of knots in terms of three 'moves' that now bear his name. The twist, poke, and slide are known as the *Reidemeister moves* and are depicted in Figure 3. In Section 3 Their mosaic representations can be found in Figure 5.

2. Background: Machine Learning & Neural Networks

In this section we will provide an overview of relevant topics pertaining to machine learning and a brief history on neural networks. To begin, *machine learning* is a process that uses statistical methods to train algorithms that can make classifications or predictions. Where *classification* refers to a model trained to assign a label to data that is not numeric (usually called *categorical data*). An example of classification is a model that is given an image and must "classify" it as either a cat or dog. In contrast, *regression* refers to a model trained to

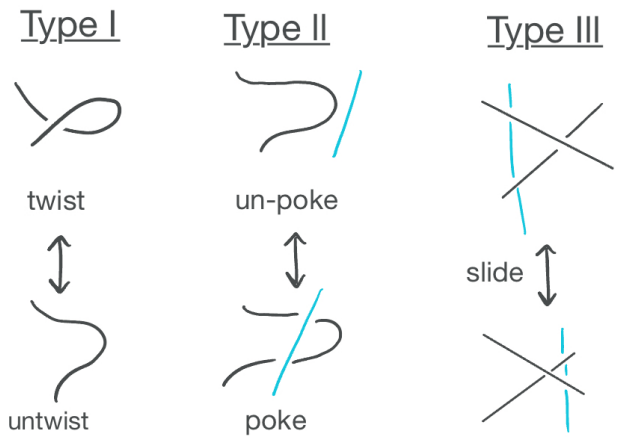

FIGURE 3. Reidemeister Moves

predict a numeric value (usually called *continuous data*). An example of regression would be a model that is given the square footage of a house and predicts what the value of the house is. When building a machine learning model, the data set is partitioned into two sets. The *training set* usually consists of about 60 - 75% of the data and is used to build the model. The *testing set* contains the rest of the data and is used to test how well the model performs on data that it was not trained on.

When working with data, it is important to minimize *bias* in a model to ensure the integrity of the data is being maintained. We define bias as a phenomenon that skews results in a machine learning model in favor or against a certain value. This often occurs when features in the data set have varied numeric ranges. We can *normalize* features that introduce bias by mapping all numeric features to a value between 0 and 1. This method maintains the integrity of the data while also ensuring that all features carry the same weight when being used in a model. In addition to bias, we must also be wary of *overfitting* in our model. This occurs when a machine learning model is so specialized to the partition of the data set known as the training set, that when new instances of data are used to test the model's performance, (often from a partition of the original data set called the testing set), the model performs worse than it did on the training data. The two metrics we used to gauge model performance were accuracy and loss. *Accuracy* is the percentage of correct predictions, and *loss* is typically

the Mean Squared Error (MSE). When a model is overfit, is has essentially "memorized" the training data. Moreover, the poor performance on the testing set tells us that the training set was not an accurate representation of the population. In general, we can detect overfitting in a machine learning model as the continued decrease in training loss after validation loss has leveled out or started to increase [7]. In other words, overfitting occurs when fewer and fewer predictions on the training data are wrong, but the number of wrong predictions on the testing set has leveled off started to increase.

An *artificial neural network* (ANN), as its name suggests, is a type of machine learning model whose architecture and function is intended to mimic the human brain. In 1943, neurophyisiologist Warren McCulloh and mathematician Walter Pitts used electrical circuits to model the way neurons would work in a neural network. Then Widrow and Hoff developed models called 'ADALINE' and 'MADALINE' ({Multiple} ADAptive LINear Elements) at Stanford in 1959 [2]. The next 50 years saw the rapid growth and development of computer technology, opening the door to neural network software as we know it today. In this paper, we will be working with *convolutional neural networks* (CNNs), which take 2-dimensional matrices as input and are primarily used for image classification [3].

3. Developing the Data

In this section, we will discuss the process used to develop the data set we used to train and test our neural network. The type of data we generated is called *synthetic data*. This means the data was not extracted from the real-world, but rather developed from scratch [1]. We wanted to make sure the knots and links we used were complex enough that they could not be easily distinguished by the human eye. Since neural networks require a high volume of training data, this task was not feasible by hand so we developed a program in python that could generate complex knots and links for us.

The program to generate these diagrams consisted of three main parts: (1) a list of possible moves that could be performed on a 2×2 or 3×3 submatrix of any given knot diagram, (2) a way to detect what moves were possible and select one such move, and (3) a way to visually represent the newly generated knot diagram.

In order to represent the mosaic tiles in 2D matrices, we assigned each of the 11 tiles from Figure 2 a number (0 to 10 from left to right). These numbers were used to encode the mosaic diagrams. An example

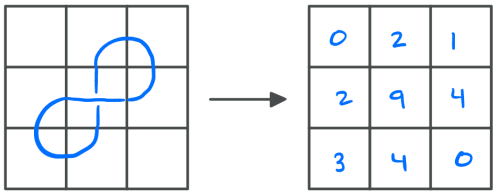

FIGURE 4. An example of a 3×3 mosaic diagram (left) and its matrix representation (right)

matrix representation of for a simple 3×3 mosaic diagram is depicted in Figure 4.

We encoded six different types of legal moves that could be performed on either a 2×2 or 3×3 submatrix of a mosaic diagram. We will refer to these as "move matrices" from now on. As depicted in the table below, the move matrices encode the three types of Reidemeister moves, as well as "balloon", "mingle" and "lava lamp" moves. The dashed arcs in the corners of the move matrices represent possible arcs that could exist in a submatrix of a knot diagram. Each possible combination of arcs in these corners is included in the list of possible moves. Moreover, each of the possible moves shown in the table only take into account a "bottom left" orientation. Thus we also included in our list of possible moves the "top left", "top right", and "bottom right" orientations of these move matrices (90° rotations).

We then developed a function called `find_moves()`that iterates over every possible 2×2 and 3×3 submatrix on a given mosaic knot diagram. At each submatrix, it checks to see if there is a match in the list of possible moves and if there is, the possible move(s) that can be performed are stored based on the location of the top left corner of the submatrix within the mosaic diagram. Figure 6 shows an 8×8 mosaic diagram with a few matching moves in it that would be identified by `find_moves()`. As an example of how this function works, let's use the balloon move labeled (5) in Figure 6. After identifying that this 3×3 submatrix spanning down three and to the right three from the corner labeled (5), we store the matching move and the location where the matching submatrix can be found. Since we are storing the top left corner of the submatrix, we would store $(5, 4)$ because the "origin" of our diagram is the top left corner.

We then took this list of possible moves and used a random number generator to select one move that is then passed into a `perform_moves()` function which swaps the identified submatrix with its matching move and returns the new matrix after the move is performed. In keeping

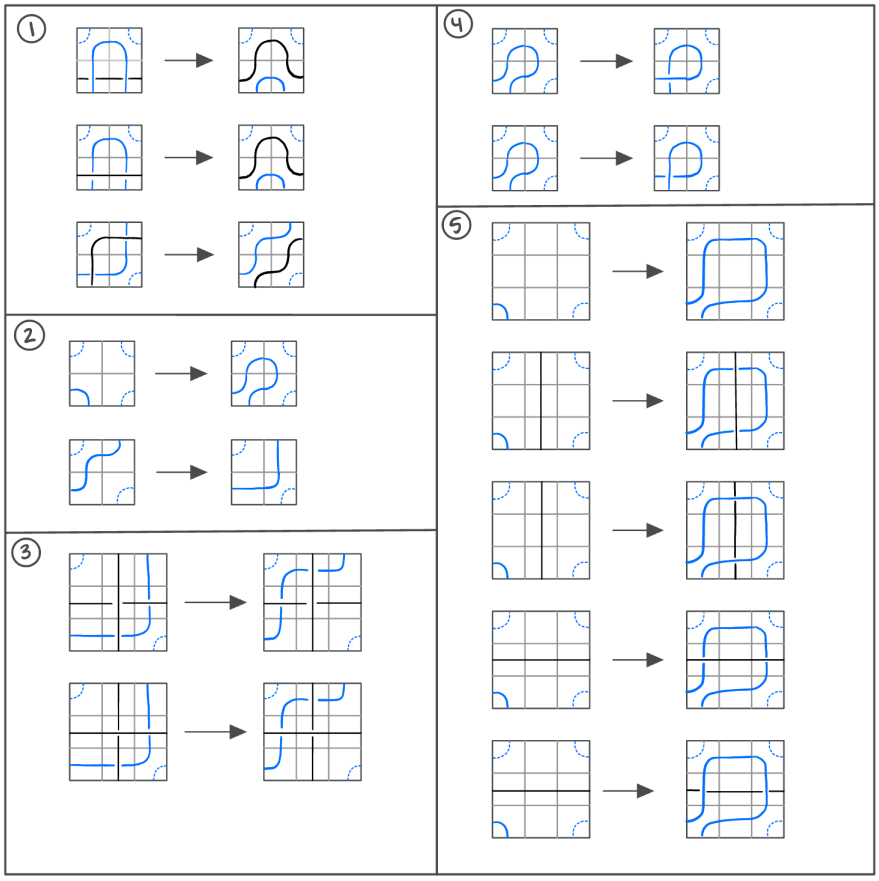

FIGURE 5. An example of each of the six types of moves we check for in our `find_moves()` function in their bottom left orientation. The dashed lines denote possible variations of a move.

with the example from Figure 6, the balloon move that was identified as the match with this submatrix at (4,5) is passed into the function along with the submatrix location, which is where the swap occurs and the new matrix is returned. For a more in-depth look at how this process occurs, the reader is encouraged to look at the code for this project that has been posted to OneDrive here.

To ensure that everything was running smoothly, we used the Pillow library to generate images based off of the matrix representation of our diagrams. We created a dictionary that mapped each of the 11 different numeric tile values to a 50 pixel by 50 pixel image representation of the mosaic tile. We could then visualize any 2D matrix diagram by simply

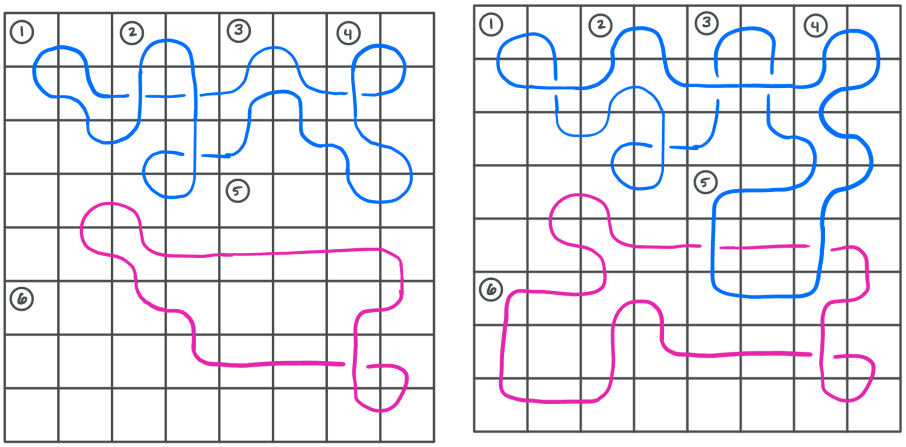

FIGURE 6. An example of some matrix moves on a mosaic diagram before (left) and after (right) applying `perform_moves()`. (1) Reidemeister type I (2 & 3) Reidemeister type II (4) lava lamp (5 & 6) balloon

creating a square image with $(50n)^2$ pixels where n is the dimension of the matrix diagram, and then mapping each matrix value to its corresponding tile image from the dictionary.

To determine the best grid size for producing complex knots without compromising too much on computational time, we tried $n = 25, 50, 100$ for the matrix diagrams. We also tried various numbers of moves performed on them. The following table summarizes our results. We settled on using a 25×25 matrix, because we were able to achieve a complex mosaic diagram with twice as many moves in just 2.4 minutes per knot.

2D Matrix Dimensions	Number of Moves Performed	Time (Minutes)
100×100	200	5
100×100	400	8.85
50×50	1000	5.3
50×50	2000	10.9
50×50	3000	21
25×25	2000	2.4
25×25	3000	4.5

We decided to start out with training data that consisted of unknots, hopf links, unlinks, and trefoils. To make sure there was plenty of data to train our neural network on, and that the data was not going to

lead to a biased diagram, we saved each manipulation of the matrix diagram between move 1 and move 1000. This means that for every move performed, a new diagram was saved to the data set. Since we performed 1000 moves on each knot, we got 1000 data points for each knot type in our neural network. Additionally, some thought was put into where the knots should be placed to mitigate bias in the model caused by a non-diverse data set. We settled on starting each of the 4 knots/links in 9 different locations on the grid. This allowed us to generate 1000 datapoints for $4 \times 9 = 36$ different original knot diagrams for a total of 36,000 knots in our data set. Figure 7 shows an example of an unknot mosaic diagram after 5, 25, 125 and 625 moves have been performed on it.

4. Convolutional Neural Network Architecture

Once we had all of our data ready to go, it was time to build our CNN. In this section, we will discuss the parameters that make up the architecture of a convolutional network as well as the architecture we used for our binary and multi-class CNNs.

A convolutional neural network, CNN, in its simplest form, consists of a series of four layers. First, a *convolutional layer* is applied to the input data. The convolutional layer is considered the core building block of a CNN [3]. In the convolutional layer, each filter applies a 2-dimensional array of weights to the input data as a dot product. Typically, the weights are represented on a 3×3 matrix, which becomes the size of the "receptive field", meaning the submatrix that a given dot product "sees". After the dot product is applied to a 3×3 submatrix of the input data, the result gets passed into an *output array* also called a *feature map*. This process is shown in Figure 8. Each convolutional layer consists of many filters that apply their own weights to the input data and produce a unique feature map. It is common for the number of filters in a convolutional layer to increase as new layers get added. This is because the convolutional layers can be thought of as a hierarchical learning process where each layer is able to identify more complex patterns in the data. Figure 9 represents this idea.

There are three hyper-parameters that affect the size of the feature map:

(1) **Number of filters:** since each filter yields its own feature map, increasing the number of filters will lead to more neurons getting passed into the fully connected layer (more on that below).
(2) **Stride:** the distance between each submatrix on the input image where the filter is applied. For example, if the stride is

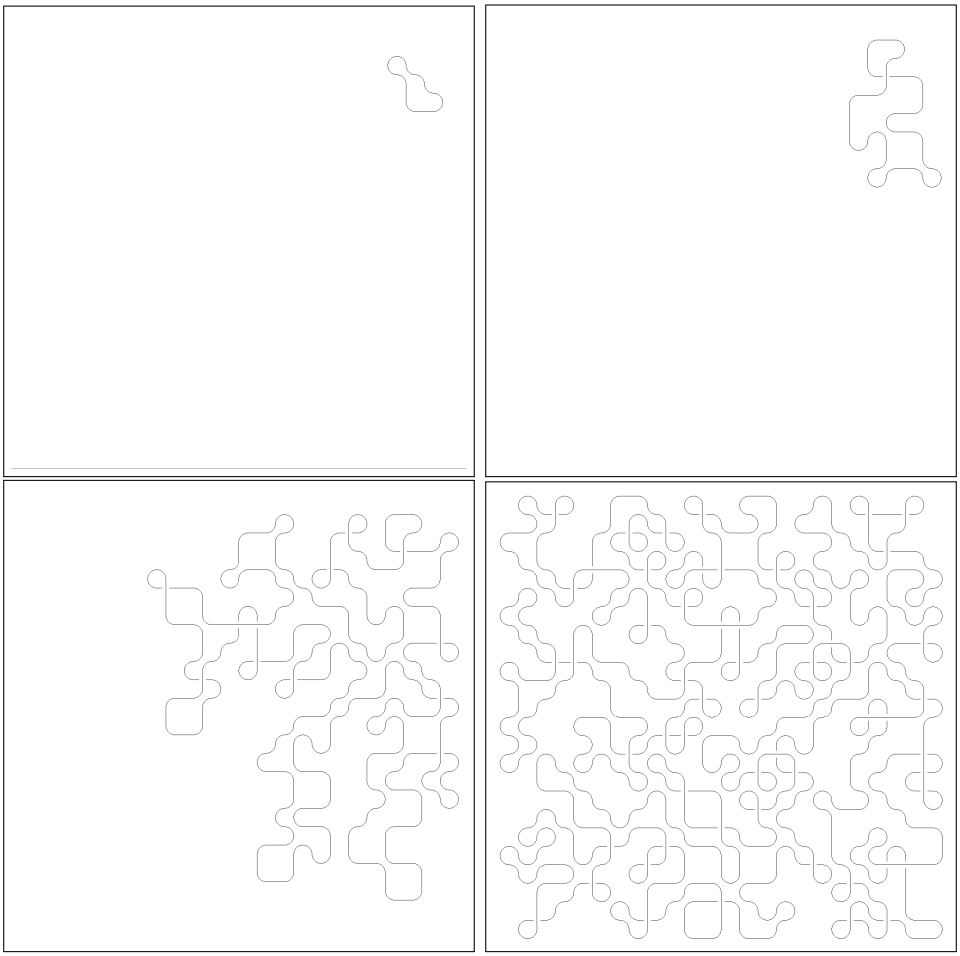

FIGURE 7. An unknot mosaic diagram after 5 moves (top left), 25 moves (top right), 125 moves(bottom left), and 625 moves (bottom right) have been performed.

1, then the filter "slides" over one unit before applying a dot product on the new receptive field. It is rare for this number to be larger than 2. As the stride increases, the size of the output decreases.

(3) **Zero-padding:** used when filters do not fit the input image. Sets all elements that fall outside of the input matrix to zero.

After each convolution operation, a rectified linear unit activation function, as shown in Equation 1 is applied to each feature map. This step introduces nonlinearity into the model.

$$f(x) = max\{0, x\} \tag{1}$$

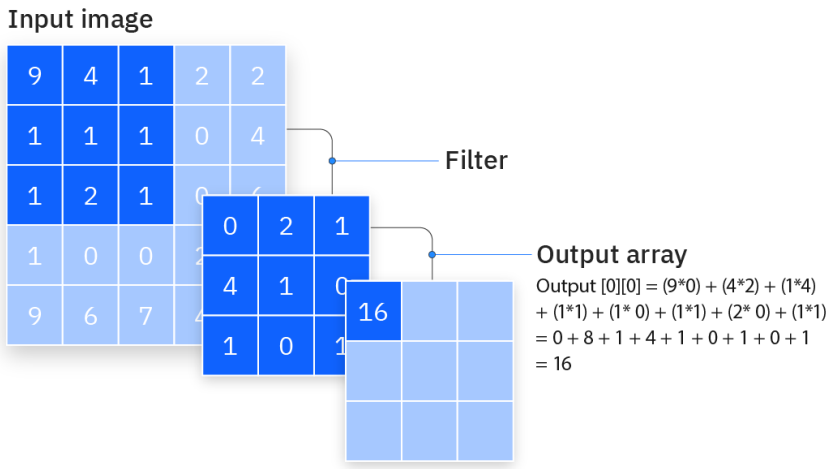

FIGURE 8. Generating a Feature Map During the Convolutional Layer [3].

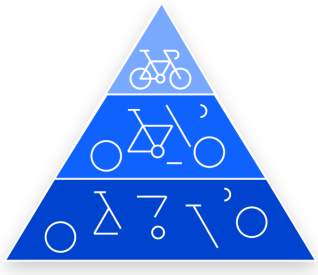

FIGURE 9. A visual representation of how a neural network learns patterns as it gets deeper into the convolutional layers [3].

After the activation function is applied, the data gets passed into a *pooling layer*. The purpose of the pooling layer is to reduce the dimensionality of the data. This is beneficial because it helps ensure that the model is able to pick up on more generalized patterns that exist regardless of orientation (an image could be upside down, sideways, zoomed in, zoomed out, etc.). By applying a pooling layer, we ensure the model is trained broadly on the characteristics of the data, which means it will perform better on unseen instances of the data. The generalizability also minimizes overfitting. The pooling layer works by sweeping a filter across the entire feature map without weights.

Instead, a possibly nonlinear aggregation function is performed. The type of aggregation function used defines the pooling layer:

(1) **Max pooling:** a summary of the features in a receptive field represented by the maximum value in that filter region.
(2) **Min pooling:** a summary of the features in a receptive field represented by the minimum value in that filter region.
(3) **Average pooling:** a summary of the features in a receptive field represented by the average value in that filter region.
(4) **Global pooling:** Each feature map is summarized by a single value that is either the maximum, minimum or average value of the entire feature map.

After multiple iterations of the last three layers are performed, the data is flattened and passed into a *fully connected layer*, where each node in the output layer connects directly to a node in the previous layer. This is where the model performs the classification task based on the features it extracted through the previous layers and their different filters. It is common for the fully connected layer to use a sigmoid activation function as shown in Equation 2 for any input $x \in \mathbb{R}$. The sigmoid function produces a value between 0 and 1 that represents the probability for each of the class options. The highest probability becomes the model's prediction. Figure 10 shows the architecture of a CNN from start to finish.

The number of times or *epochs* the training data is passed through a neural network can help a model learn more about the data set. This is a parameter that affects how much a neural network can learn. The goal is to maximize learning by selecting a sufficiently large number of epochs without causing the model to become overfit from too many pass throughs. There is also a mechanism called the *optimization function* which adjusts the weights on the convolutional layer filters up or down based on that epoch's calculated loss function. There are multiple commonly used optimization algorithms, including stochastic gradient descent (SGD), Adaptive Learning Rate (ADADELTA), Adaptive Momentum Estimation (Adam), and others; all of which are designed to figure out how to adjust the weights and biases to minimize loss [7]. We chose to use the Adam optimizer for our project because it is specifically designed for deep neural networks.

$$S(x) = \frac{1}{1 + e^{-x}} \tag{2}$$

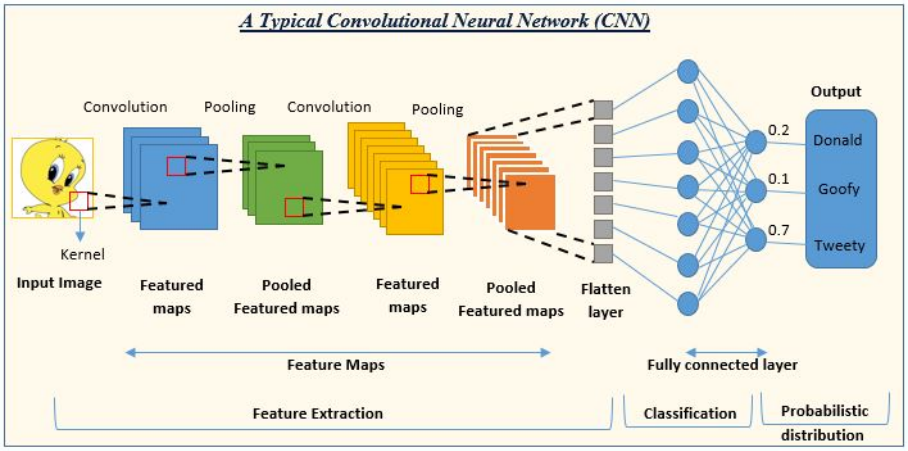

FIGURE 10. A diagram representation of the CNN architecture [10]

The overall architecture of a CNN varies depending on the task that the network is being used to solve. For the purpose of our research, having an increasing number of filters in each convolutional layer was a priority for getting the model to pick up on patterns and hopefully identify characteristics of the legal moves that are used in knot theory.

The architecture we used for our CNN was inspired by data scientist Harrison Kinsley's YouTube Playlist "Deep Learning basics with Python, TensorFlow and Keras" [5]. It consists of four convolutional layers. Before each convolutional layer is applied, we apply a *dropout* layer that randomly hides 20% of the training data from the layer. Since we cannot easily rotate, reflect, or otherwise manipulate our mosaic diagram matrices using the built-in tensorflow package, this is an optimal alternative to help prevent overfitting in the model. The first convolutional layer consists of 8 filters that generate corresponding feature maps about the input data. The feature maps are then passed into a ReLU activation function and then a max pooling layer with pool size set to 2×2. This means the max value is taken from each 2×2 submatrix across the feature maps. The second convolutional layer is the same only with 16 filters. As the neural network gets deeper, it is common to increase the number of filters to help the network learn more complex features. The same activation function and pooling layer is used each time, leading us to the third convolutional layer which consists of 32 filters. The fourth and final convolutional layer has 48 filters to truly maximize the network's potential to pick up on complex patterns within the feature maps. We then flatten the output and feed

it into a fully connected layer with 48 neurons, one for every feature map from the fourth layer. A ReLU activation function from Equation 1 is applied one last time after the fully connected layer and then passed through one more fully connected layer with only one neuron to perform the binary classification. Our classification is then passed through a sigmoid function from Equation 2 which converts the neural network's prediction to a value between 0 and 1. In binary classification models, if a value is less than 0.5, it is predicted as the 0 class label and if it is greater than or equal to 0.5, it is predicted as the 1 class label. For our CNN, 0 is used to represent 'unknot', and 1 represents 'un-unknot' (meaning it is either a link or a knot but it is **not** a single unknot).

We used the same architecture for our Multi-class CNN, with two minor adjustments to be able to perform classification with more than two categories. First, after the ReLU function is applied and the 48 neurons are ready to be passed into the second fully connected layer, there are four neurons instead of one because there are four possible categories for the input to be classified as. Second, instead of using a sigmoid function, a softmax function is used to convert each prediction to a probability between 0 and 1 for each of the four classes. Again, the class with the highest probability is the model's prediction. The softmax function is depicted in Equation 3

$$\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}} \tag{3}$$

5. Binary CNN Results

In developing a "good" model, we sought to optimize two parameters of our neural network: batch size and number of epochs. We ran numerous trials with varying parameters to determine which models had the highest performance for our data. The level of performance was measured using the accuracy and loss from the testing/validation set.

Number of Epochs	Batch Size	Final Validation Loss	Final Validation Accuracy
10	100	0.1894	0.9593
10	150	0.2436	0.9468
10	200	0.3100	0.9315
10	250	0.2290	0.9500
10	**300**	**0.2202**	**0.9574**
15	100	0.2587	0.9010
15	150	0.2898	0.8694
15	200	0.3092	0.8570
15	250	0.2975	0.9118
15	300	0.3171	0.9008
20	100	0.2014	0.9166
20	**150**	**0.1490**	**0.9627**
20	200	0.4634	0.7369
20	**250**	**0.1738**	**0.9556**
20	300	0.2499	0.9473

Of the 15 combinations of epochs and batch size represented in the table above, there were three that stuck out to us due to their high performance. They are depicted in pink text. The first model was trained on 10 epochs with a batch size of 300. The performance of this model is represented by Figures 11, 12. Recall that overfitting is indicated by a continued decrease in training loss after validation loss has leveled out or started to increase. We can use the loss graph in Figure 12 to determine how much overfitting may have occurred in this first model. Notice that the validation loss represented by the pink line does not appear to level out before the training ended. This tells us that the model has a low chance of being overfit, and it is probably more likely that the model did not train long enough to perform as well as it possible could. In particular, we see that the validation accuracy modeled by the pink line in Figure 11 has not begun to level out yet.

The second model was trained on 20 epochs with a batch size of 150 and its performance is shown in Figures 13, 14. From these graphs, we can see that the validation loss shown by the purple line has not quite leveled off by the end of training. Again, we have little reason to be concerned about overfitting. It is important to note that the overall loss was able to reach an all-time low of 0.1490 while the accuracy was at an all-time high of 0.9627.

The third model was trained on 20 epochs with a batch size of 250. This model's performance is shown in Figures 15, 16. We can see from the blue validation line in the loss graph from Figure 16 that there is no

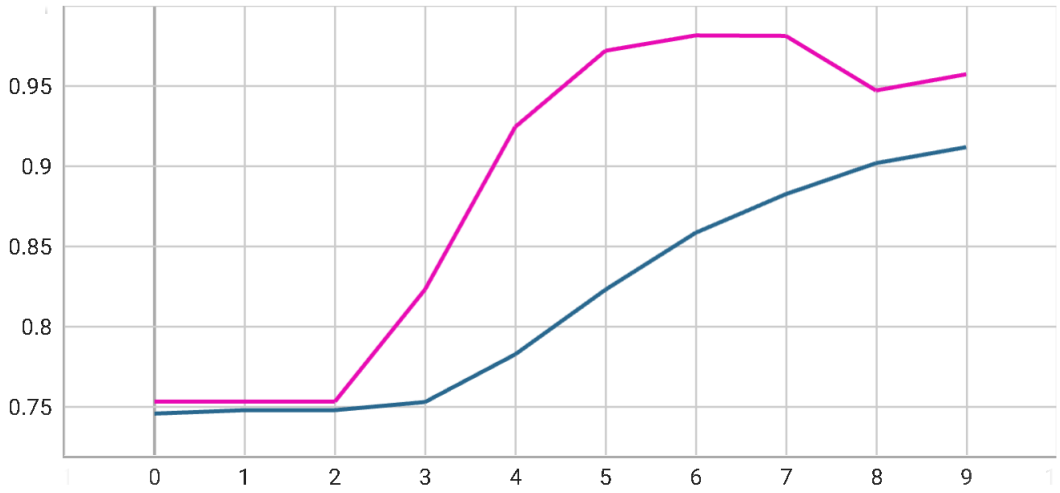

FIGURE 11. Accuracy by epoch for both the training (blue) and validation (pink) sets of the binary CNN trained on 10 epochs with a batch size of 300.

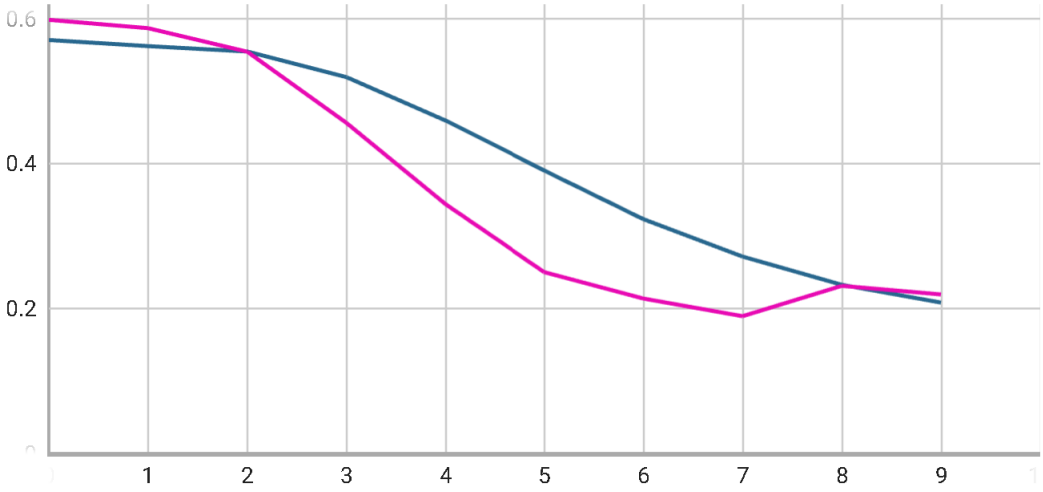

FIGURE 12. Loss by epoch for both the training (blue) and validation (pink) sets of the binary CNN trained on 10 epochs with a batch size of 300

leveling out before the model is fully trained. This is a good sign that the model did not experience any overfitting and is confirmed by the fact that the validation accuracy is higher than the training accuracy in Figure 15. We know from Section 2 that having a higher training

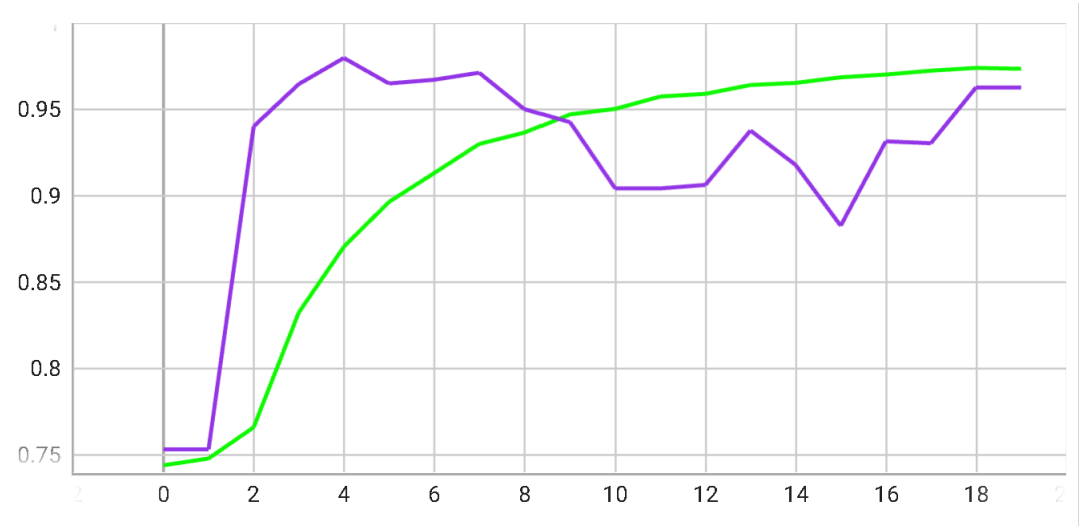

FIGURE 13. Accuracy by epoch for both the training (green) and validation (purple) sets of the binary CNN trained on 20 epochs with a batch size of 150.

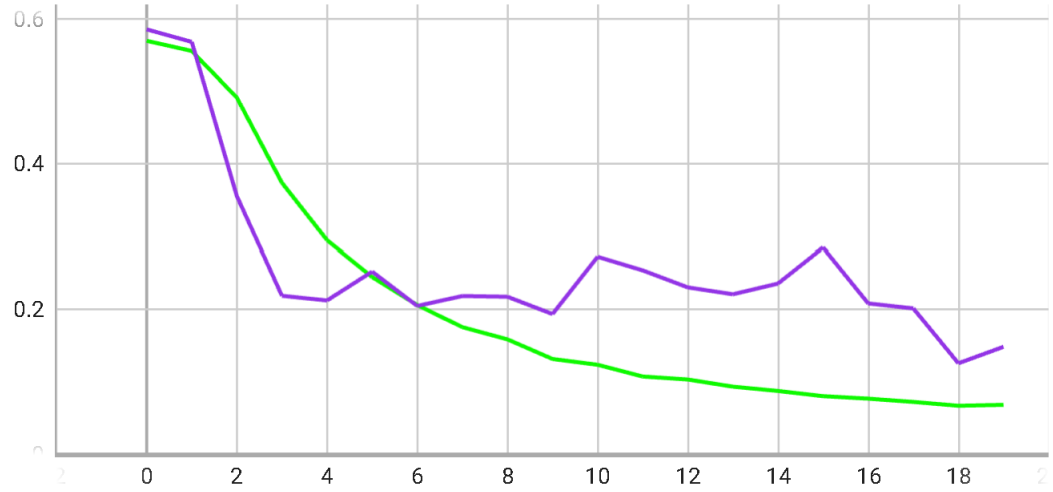

FIGURE 14. Loss by epoch for both the training (green) and validation (purple) sets of the binary CNN trained on 20 epochs with a batch size of 150.

accuracy can indicate memorization/overfitting in a model. Since the only difference between this model and the second binary model is the batch size, we can conclude that the batch size has an affect on the overall performance of the model. In fact, having a higher batch size

resulted in a lower accuracy and a higher loss. The table above reflects these findings. Of course, the dropout layers also play a critical role in the overall performance of the model because they affect which data points the model is actually trained on. It is possible that a poorly performing model is a consequence of too many critical matrices being dropped out, i.e. the model trained on 20 epochs with a batch size of 200.

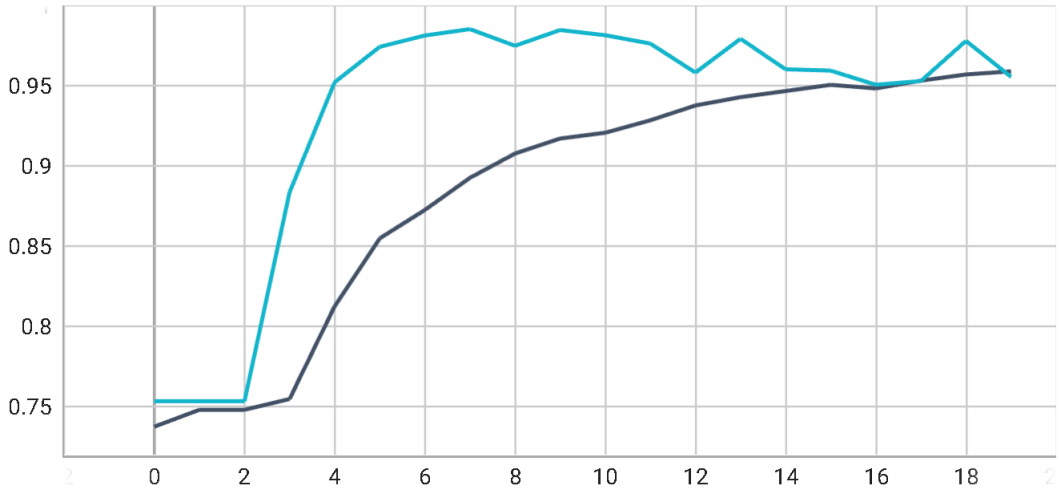

FIGURE 15. Accuracy by epoch for both the training (black) and the validation (blue) sets of the binary CNN trained on 20 epochs with a batch size of 250.

6. Multi-class CNN Results

The same parameters, batch size and number of epochs, were tested for the Multi-Class CNN. Of the 15 combinations, there were only two that demonstrated high performance. First was the model trained with 15 epochs and a batch size of 300. The results of this model are displayed in Figures 17, 18. We can use these figures to determine the extent to which overfitting may have occured during model training. In Figure 18, the purple line representing the validation loss appears to level out around epoch 8 or 9. Since the training loss continues to decrease for the next 5 or so epochs, there is a chance that this model is slightly overfit. Since the validation accuracy in Figure 17 is still higher than the training accuracy, overfitting is not a large concern.

The second model with high performance was trained using 20 epochs and a batch size of 100. The performance of this model is depicted in

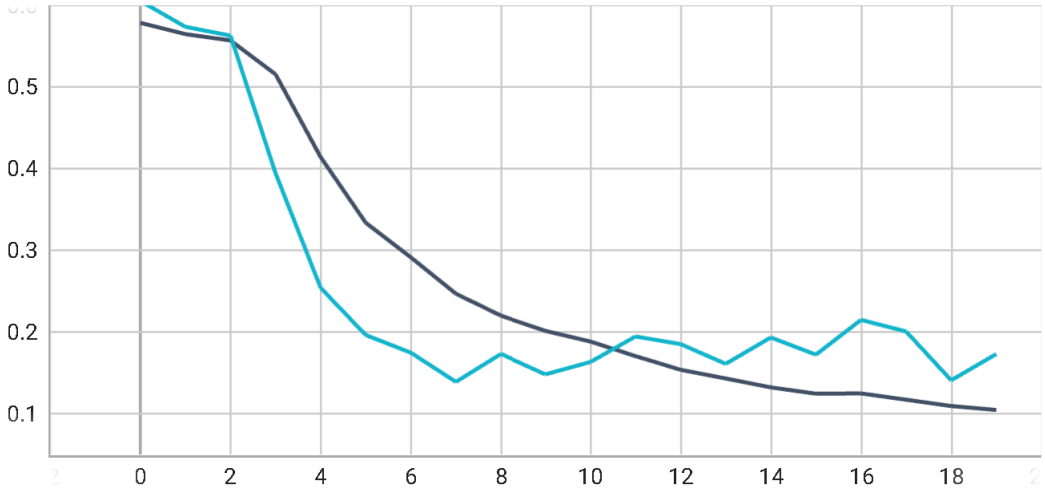

FIGURE 16. Loss by epoch for both the training (black) and validation (blue) sets of the binary CNN trained on 20 epochs with a batch size of 250.

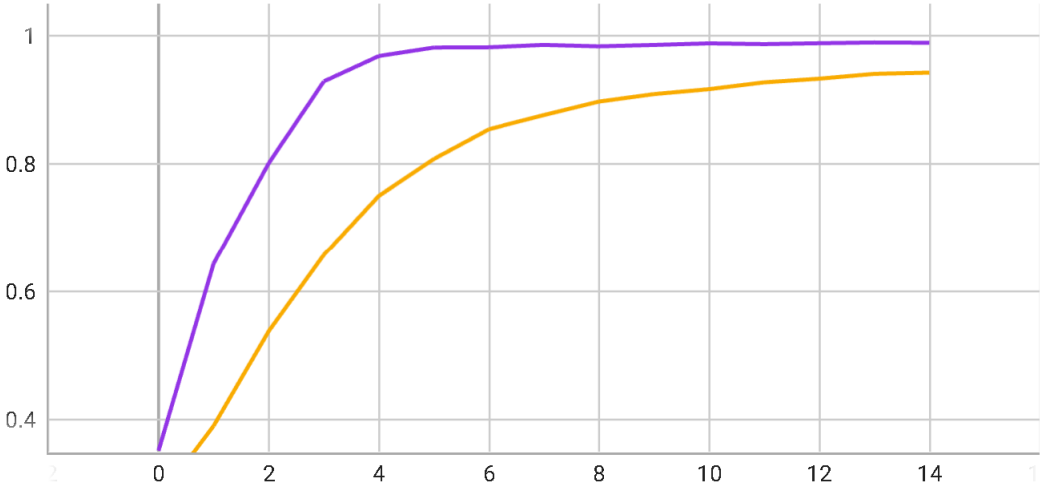

FIGURE 17. Accuracy per epoch for both the training (orange) and validation (purple) sets of the Multi-Class CNN with 15 epochs and batch size of 300.

Figures 19, 20. When checking for overfitting in this model, we can see that the validation loss as represented by the blue line in Figure 20 does not appear to level off until the final epoch. Meanwhile, the

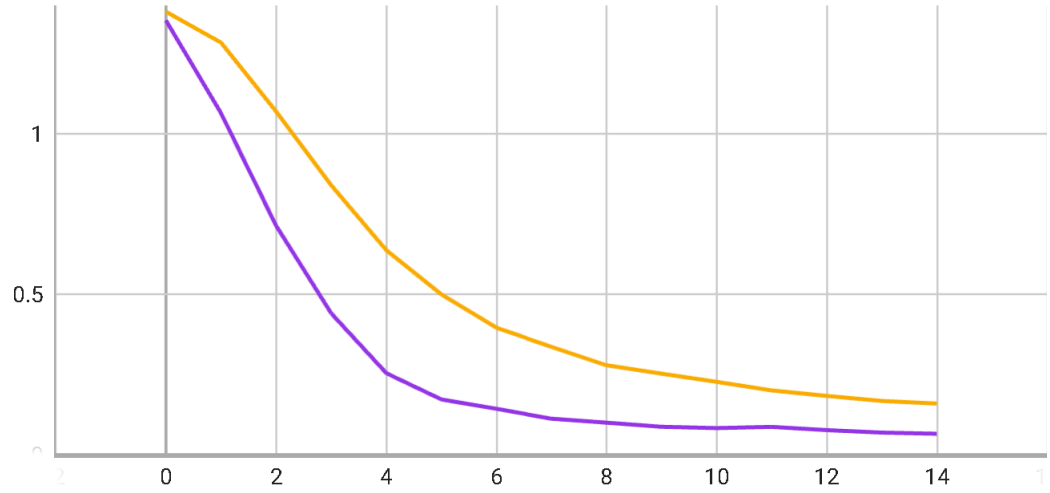

FIGURE 18. Loss per epoch for both the training (orange) and validation (purple) sets of the Multi-Class CNN with 15 epochs and batch size of 300.

training loss is steadily decreasing, so we can be more confident this time that the model has not been overfit.

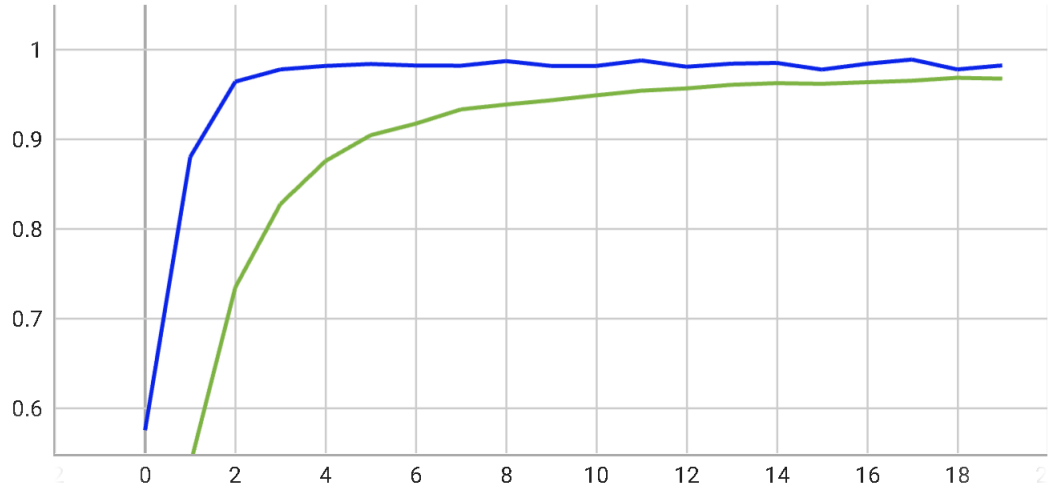

FIGURE 19. Accuracy per epoch for both the training (green) and validation (blue) sets of the Multi-Class CNN with 20 epochs and batch size of 100.

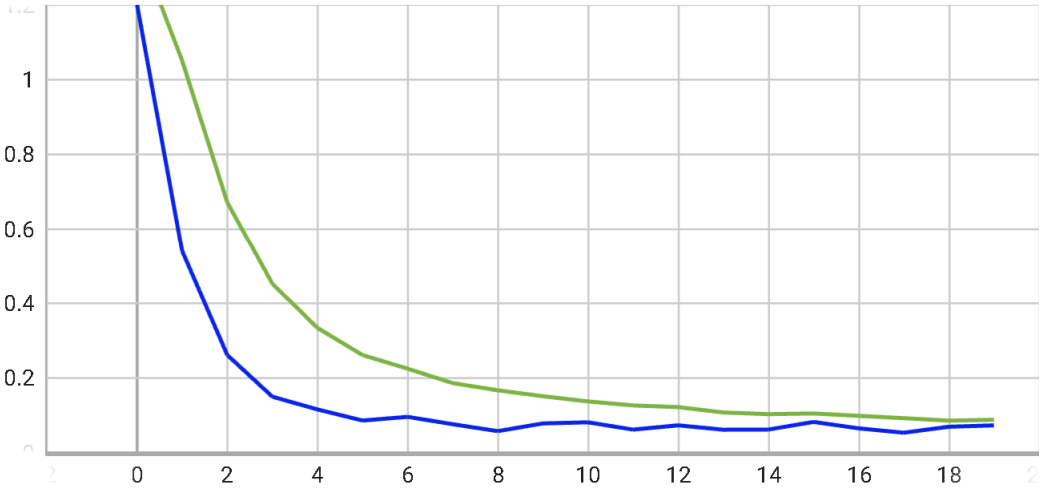

FIGURE 20. Loss per epoch for both the training (green) and validation (blue) sets of the Multi-Class CNN with 20 epochs and batch size of 100.

7. Conclusion

While all of the Binary and Multi-Class models that we saved had a good validation accuracy by the final epoch, their *deployability* is determined by testing the model on unseen instances that were not part of the original 36,000 mosaic knot dataset. The performance on these new instances tells us how generalized the model is and thus reflects a more accurate summary of how well the model would perform if deployed. To this end, we developed a new unknot, hopf link, unlink, and trefoil shown in Figure 21. The four new diagrams we selected were very simple representations of each of the four knots but in a very different manner from the data we used to train the model. After developing these very "stretched out" knots, and just feeding these four simple diagrams into the five models (three binary, two multi-class), we realized that the models were almost always classifying the diagrams as "unknot". This led us to perform 100 moves on each knot using the `perform_moves()` function from before, yielding 400 new data points to test our models on. The results of these tests are modeled in the table below.

Not surprisingly, the second binary CNN model that achieved the highest accuracy and lowest loss ended up performing the best on these unseen instances with an accuracy of approximately 41%. Digging into these results a little deeper, we see that the model accurately predicted

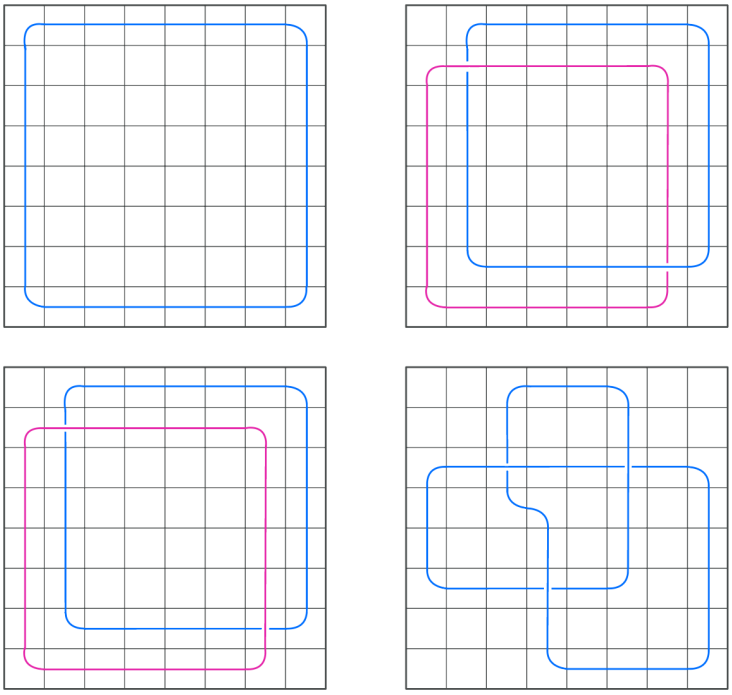

FIGURE 21. The new unknot (top left), hopf link (top right), unlink (bottom left), and trefoil (bottom right) mosaic diagrams for model testing. (not to scale)

64 out 100 of the unknot matrices, 61 out of 100 of the hopf links, 0 out of 100 of the unlinks, and 38 out of 100 of the trefoils. It is important to note that most of the non-unknots were identified past the 50 move range. This tells us that the model is more effective at classifying complex knot diagrams, which was our goal, however, the fact that none of the unlink diagrams were classified correctly tells us that there is still a lot of work that needs to be done before this approach to solving the unknotting problem can be taken seriously.

To try and get a sense of what this neural net is "thinking" when we plug in one of the 400 different input knot diagrams, we generated "brain scans" of the convolutional layers for an unknot, hopf link, unlink, and trefoil after 58 moves had been performed on each. We selected 58 moves because this is a point where all but the unlink were correctly classified by the binary model as being an unknot (0) or not (1). A sample of the brain scans are shown in Figure 22. As fascinating as these images are, they do not lend themselves to a quick understanding of what is happening the neural network "under the hood". We

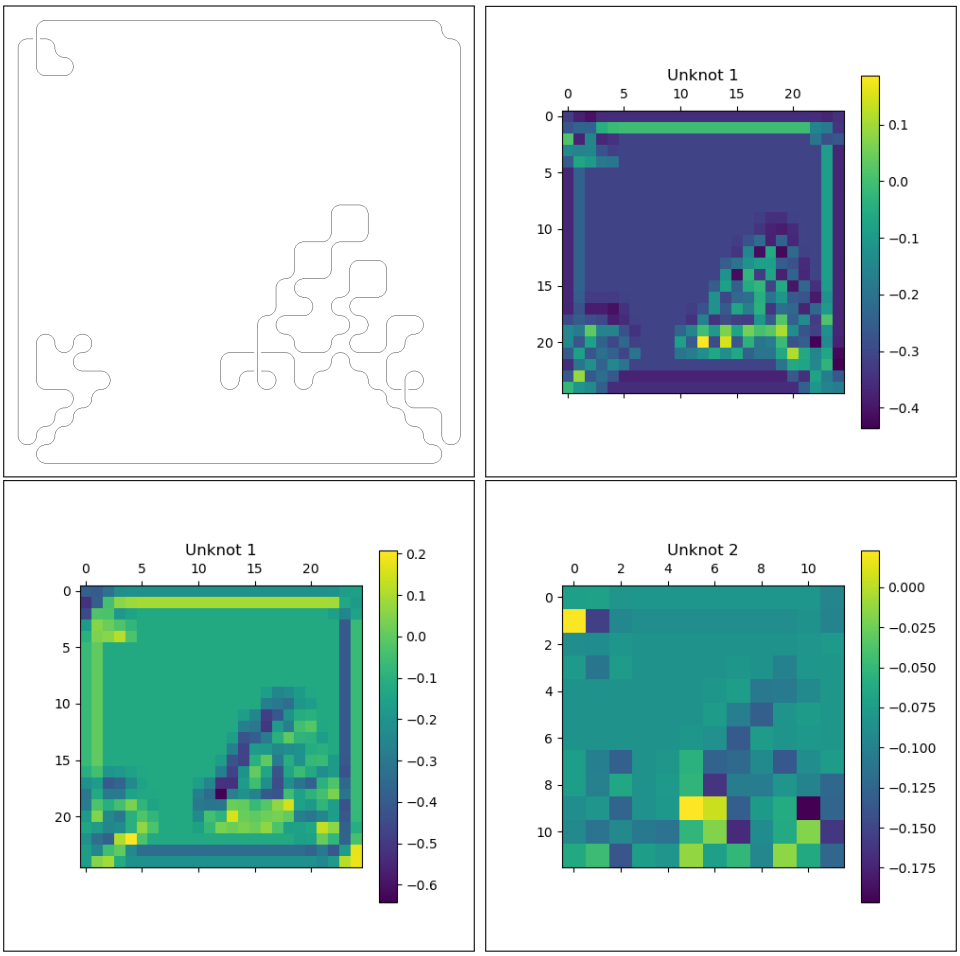

FIGURE 22. An unknot with 58 moves performed on it (top left), brain scans from the first convolutional layer (top right, bottom left) and a brain scan from the second convolutional layer (bottom right)

believe they spark some interesting questions about how the CNN is working to classify the knot diagrams. In particular, we are curious to know how the different weights applied by the various filters at each level of the convolutional layer affects what is being identified as important to in the diagrams. Moreover, we want to know what tiles, if any, are being singled out by these convolutional layers, i.e. a specific crossing tile or grouping of tiles that shows whether an unknot or not is present.

8. Project Reflection (TLDR)

The goal of this project was to apply machine learning to the very data-like structure of mosaic diagrams. To this end, we wanted to see if we could develop an algorithm using a convolutional neural network (CNN) that would successfully determine whether any given knot is an unknot or not. While at the outset this seemed like a feasible task for a CNN to accomplish, the more we worked with our data and began to scratch the surface of how a convolutional neural network "learns", we realized that the moves a knot theorist can so easily detect when looking at a mosaic knot diagram are not quite so straightforward for a computer. This being said, we developed many models, as demonstrated in the table above that show it is possible to train a neural network to be somewhat successful at distinguishing between different knot types. In particular, it was common for the model to predict the correct knot type more accurately when the knot increased in complexity.

From the perspective of a mathematician, the fact that our best model achieves upwards of 90% accuracy on our testing set and 44% accuracy on never before seen knot diagrams is exciting news for many reasons. For starters, these performance levels tell us that there is room for improvement, both in terms of the model itself, as well as in terms of the underlying technology. In 10 years when laptops have stronger GPU's and can handle more computationally intensive tasks, there is a high potential that a neural network could very easily be able to differentiate between an unknot and other knots. In the meantime, the news is also exciting in that it provides a little bit of security. I appreciate knowing there are still things in the field of mathematics that a computer cannot replace. This idea of computers being able to replace humans in the workplace is a thought that has become more prevalent as I begin to consider my own career trajectory. I hope to see a strong mutualistic relationship between humans and technology. In particular, I am optimistic that the evolution of technology will continue to be driven by human curiosity and innovation however with a stronger sense of what ethical best practices are surrounding ChatGPT and similar artificially intelligent large language models. The adage that not everything you hear on the internet is true is more important than ever when information is gleaned from a source like ChatGPT. For these reasons I am stressing a mutualistic relationship in which humans are gaining from computers just as much as computers are gaining from humans. If we develop too strong of a reliance on technology that we don't understand, what are we learning?

References

[1] Gerard Andrews. What is synthetic data?, 2021.
[2] Jimmy Pang Caroline Clabaugh, Dave Myszewski. Neural networks history: The 1940's to the 1970's, 2000.
[3] International Business Machines. What are Neural Networks?
[4] k. Reidemeister. Knotten und gruppen. *Abhandlungen aus dem Mathematischen Seminar der Universität Hamburg*, 1927.
[5] Harrison Kinsley. Deep learning basics with python, tensorflow and keras, 2018.
[6] Takahito Kuriya and Omar Shehab. The Lomonaco-Kauffman conjecture. *J. Knot Theory Ramifications*, 23(1):1450003, 20, 2014.
[7] Microsoft Learn. Machine learning challenge, 2023.
[8] Samuel J. Lomonaco and Louis H. Kauffman. Quantum knots and mosaics. *Quantum Information Processing*, 7(2-3):85, 2008.
[9] Derek Muller. How the most useless branch of math could save your life, Sept. 2023.
[10] Saily Shah. Convolutional neural network: An overview, 2022.