# Scrabble score maximization from image input using corner keypoint warping and neural network letter recognition

Wei Jie Li, Jaehyun Shim, Kevin Wang, William Wang
University of Michigan
lweijie@umich.edu, jaeday@umich.edu, kwwangkw@umich.edu, willruiz@umich.edu

## Introduction

The motivation of this project is to provide a helper tool for people who would like additional assistance with Scrabble - a common English word-based board game. The idea for this project is to allow for a user to easily upload a photograph of their board and immediately be able to receive the highest possible scoring move they could make.

Although the motivation seems rather constrained to just entertainment purposes, the tool can be repurposed to more pertinent applications such as automating inferences from standardized document scanning and visual metric collection, where we can significantly eliminate tedious manual visual labor.

The problem is able to be solved because it is known Scrabble boards follow a standardized format and therefore allows us to confidently infer where tiles are instead of implementing dynamic tile detection, which is prone to being fragile to edge cases.

We know we can do board corner detection to obtain keypoint to fit a homography and flatten image, however we cannot guarantee that the detected corners are the board corners, hence we will still require some manual correction by the user.

We know that Scrabble tiles are all capital letters in the Eurostile font. Therefore, we can train a neural network with a known classifier count to learn letter recognition given that we acquire an appropriate dataset to train it on. If we are not successfully able to train the neural network model, we can use other models as fallback. (Google Cloud, Pytesseract, etc)

Related pieces of work include *The World's Fastest Scrabble Program* by Appel and Jacobson [1].

Currently, the method we use begins by using corner detection, homography transforms, and letter recognition. This is used to turn a scrabble board image into a digitized matrix that is then run with a scrabble score maximization algorithm to find the next best possible move for the player.

## 1. Approach

Initially, we used PyTesseract for the letter recognition of the tiles. Despite experimenting with the different parameters, we were not able to get very good results—accuracy remained below 70% even with a very legible scrabble board image.

Instead, we looked into training our own neural network. We used a dataset of characters in natural scenes [4] and trained multiple different variations of neural networks (2, 3, 4 convolutional layers) and hyperparameters. However, we were not able to get good performance when testing with scrabble tiles—accuracy was below 50%.

We also tried using pretrained models such as MobileNetV2 [5] and modifying the top layer to predict for characters. This approach was also unsuccessful, with an accuracy lower than 50%.

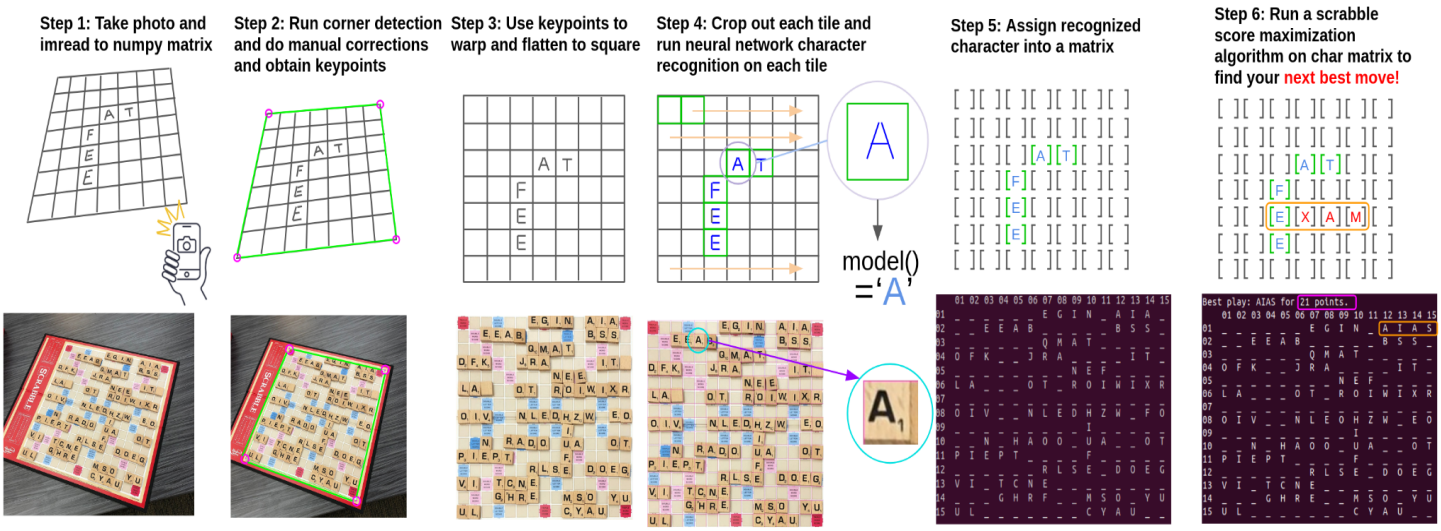
We then trained a new set on scrabble tile pictures. We first collect a dataset of labeled images of scrabble tile images. The process of this will be described in the following 'Data' section.

Next we construct and train a neural network model. Images are first preprocessed by converting them to grayscale, normalizing them with mean 0 and variance 1, and padding the image to a fixed 32x32 size. The neural network is constructed with four convolutional layers of kernel size 3, 5, 3, and 3 respectively. The channel is amplified to 32 from 1 by the first convolutional layer. Convolutional layers were arranged into two groups, with a dropout of p=0.1 applied after each group to prevent overfitting. Between the groups, there are also 3 mean pooling layers with factor 4 and stride 2. Batch normalization was also added between the groups to prevent overfitting and to speed up the convergence of the model. At the top of the network, we add 2 fully-connected layers to produce 27 classes (26 English alphabets + 1 for an empty tile). The model is trained on a training dataset with n=7258. For the hyperparameters, we have a batch size of 32, learning rate of 2e-03, weight decay of 1e-04, and 18 epochs. After training, the model achieved an accuracy of 0.969 on the test dataset (n=3629). To obtain predictions from the model output, we take the softmax of the output and return the class with the largest softmax score.

The user then inputs an image of their board: a picture containing the complete board is sufficient for our algorithm to work.

The next step is using a corner detection algorithm to find board corners for automatic keypoint detection. In order to detect the corners of the board and then use a homography transform to crop the image and warp the picture into a top-down view, a few different algorithms and approaches were experimented with and developed. The original approach was a contour finding algorithm. The goal was to use Opencv's findContours to draw out the lines of the board and then pinpoint the maxima and minima of the points to find the corners. This approach worked extremely well with clear, well-illuminated images, but struggled to define the contours of any boards in grainy photographs. The next method of corner detection that was attempted was a template detection method using SIFT. However, with this method, a similar problem occurred with detection in that clear photos were identified extremely well, but photos taken in non-ideal lighting or with grain/distortions were not able to be fit to the template image.

The final method that was explored was using a Harris Corner Detector to detect the corners in the image, eliminate outliers, and then choose the four corner corners that were detected. Just using a Harris Corner Detector did not return the




most optimal of results, so an addition of corner detection with subPixel accuracy was added [6]. With this strategy from OpenCV documentation, the corners were found with maximum accuracy by using OpenCV's cornerSubPix function, which refines corners detected with sub-pixel accuracy. This methodology returned the best results on all test images and is the implementation that is used in the final code base.

The next step in the process is a manual corner adjustment done by the user using a GUI drag interface that we designed in order to pinpoint the keypoints and confirm the corner detection from above. In order to do this, we visualize the keypoints using Opencv drawing functions cv2.circle and cv2.line so the user sees what our corner detection thinks the board corners are. We implemented a click and drag function using in the visualization where the user can manually adjust where the board corners truly are. We want manual adjustment since our corner detection is far from perfect as described in the previous section. Once the user is satisfied with the visualization of the board corners, the user can use a cv2.waitKey press to continue to the next process of our algorithm.

Next, we do a homography transformation to warp the image to a flattened view of the board. Using an approach very similar to our HW3, we take the four keypoints from corner detection and do a homography transform that crops+flattens the image, and also resizes it to have square dimensions.

The next step is to loop through and crop out each tile and run the letter detection model on each tile to derive the character in the square as a chartype. Now that the image has been cropped, flattened, we use the assumption that a scrabble board contains 15x15 tiles and crop the image into individual tiles and run our letter recognition on each cropped tile section.

Then, an array representation of the board is constructed. Once we run the letter recognition model and obtain the recognized char, we insert the char into a numpy char matrix at its corresponding [x][y] coordinates from the board tile grid.

Manual correction of misidentified tiles follows the letter recognition model. Although our letter recognition managed to achieve around >95% accuracy on modern scrabble boards, errors need to be corrected. Furthermore, poor corner setting during the corner detection step of the program or even a very skewed picture of the board taken by a user, could result in skewing the board's tiles for letter recognition and further increasing room for error.

Therefore, we aimed to develop an interface that would allow the user to correct any misidentified tiles manually. We ideated a functionality to provide a very simple yet very effective display of the board that the program has, so the user can easily examine the display, identify the misidentified tile and location, and change that tile to the correct letter. In order to do this, the program outputs the board that the system has identified in a very human-friendly way. We have clearly labeled the x and y coordinates of each location of the board, allowing the user to simply identify where a misidentified tile is located. The user will be able to tell the program where the misidentified tile is by inputting the corresponding coordinates as outputted by the board. Afterwards, the user will input the correct letter, and the program will output the board in the same format, reflecting the changes made. This interface is kept running until all misidentified tiles on the board have been fixed; every time the user fixes a tile, the user will be prompted again to check the board and see if all misidentified tiles are fixed. We have used Python's REGEX library for pattern findall function; if the user inputs invalid form of input, the user will be prompted to fix it to a valid format. Once the user can see that the board contains the right letters at the right location, the program will continue and use the final representation to calculate the best move and score.

The final user input will be the input of the user's hand. In addition to the user inputting the current progress of the game (the board), they must be able to also input the set of letters they have for the next move. We have also created an interface that allows the user to input their letters directly into the program; mainly a while loop that counts to the number of letters the user has, and only allows the user to input alphabets. It's very error safe, since the domain of input for this particular part of the program is restricted to any uppercase alphabets, if inputted any non-alphanumeric character or any number, the program will disregard the input and prompt the user to input a valid letter. Lowercase alphabets are also valid forms of input; the program will automatically convert them to uppercase for later score calculation. Again, we have used Python's REGEX library for efficient input validity checks.

```
Here is your processed board.
    01 02 03 04 05 06 07 08 09 10 11 12 13 14 15
01  _  _  _  _  _  _  E  G  I  N  _  A  I  A  _
02  _  _  E  E  A  B  _  _  _  _  _  B  S  S  _
03  _  _  _  _  _  _  Q  M  A  T  _  _  _  _  _
04  D  F  K  _  _  J  R  A  _  _  _  I  T  _  _
05  _  _  _  _  _  _  _  _  N  E  E  _  _  _  _
06  L  A  _  _  _  O  T  _  R  O  I  W  I  X  R
07  _  _  _  _  _  _  _  _  _  _  _  _  _  _  _
08  O  I  V  _  _  N  L  E  D  H  Z  W  _  F  O
09  _  _  _  _  _  _  _  _  I  _  _  _  _  _  _
10  _  _  N  _  H  A  O  O  _  U  A  _  _  O  T
11  P  I  E  P  T  _  _  _  _  F  _  _  _  _  _
12  _  _  _  _  _  _  R  L  S  E  _  D  O  E  G
13  V  I  _  T  C  N  E  _  _  _  _  _  _  _  _
14  _  _  _  G  H  R  E  _  _  M  S  O  _  Y  U
15  U  L  _  _  _  _  _  _  _  C  Y  A  U  _  _

Does this look correct? (Y/N)Y
We will now return the most optimal gameplay for you.
Please enter the number of tiles on your hand: 7
Please enter a letter in your hand: A
Please enter a letter in your hand: E
Please enter a letter in your hand: L
Please enter a letter in your hand: O
Please enter a letter in your hand: F
Please enter a letter in your hand: D
Please enter a letter in your hand: S
```

After the preprocessing and letter detection is run and an array of chars representing the scrabble board is created, the next step is running a score maximization algorithm on the board and the user hand input. In order to accomplish this, the "World's Fastest Scrabble Program" algorithm [1] was used, with the majority of the implementation provided by the paper on GitHub. This algorithm is described in depth in the research paper, so will not go into depth here. In order to suit this code base to our specific use case, a new entry point was coded for the algorithm, and a new board representation was created. Part of this was reconstructing the board using new data structures after searching the regular and transposed boards for each word alongside the index of the starting character of each word. The algorithm was further altered to better utilize and calculate scores using the "bonus" markers on the board such as double letter score or triple word score.

Finally, the program prints out the highest possible scoring word along with the score of that word. A visual representation of the new board is also created and printed for easy understanding for the next move, as shown below.

```
Best play: COALED for 42 points.
    01 02 03 04 05 06 07 08 09 10 11 12 13 14 15
01  _  _  C  O  A  L  E  D  _  _  _  _  _  _  _
02  _  F  I  R  E  _  _  _  _  _  _  _  B  _  _
03  _  _  N  _  _  _  _  _  O  D  A  I  R  _  _
04  _  _  N  _  _  _  _  _  _  _  R  _  E  _  _
05  _  H  A  Y  M  I  T  C  H  _  E  _  A  _  _
06  _  _  _  _  O  _  _  _  _  _  N  _  D  _  _
07  _  _  _  _  C  _  _  _  _  _  A  _  _  _  _
08  _  _  _  _  K  A  T  N  I  S  S  _  _  _  _
09  _  _  _  _  I  _  _  _  _  _  _  _  _  _  _
10  _  _  H  U  N  G  E  R  G  A  M  E  S  _  _
11  _  R  _  _  G  _  F  _  _  _  U  _  N  _  _
12  _  U  _  _  J  _  F  _  _  _  T  _  O  _  P
13  P  E  E  T  A  _  I  _  _  _  T  _  W  A  R
14  _  _  _  _  Y  _  E  _  _  _  _  _  _  _  I
15  _  _  _  _  _  _  _  _  _  _  _  S  E  A  M
```

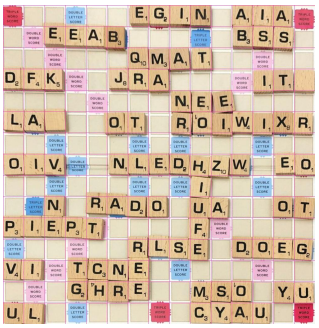
## 3. Experiments

We've done a variety of photo angles and scrabble set versions, both from our own scrabble board images and scrabble board images from the internet.

We describe both qualitative (images that the user sees) and quantitative results (accuracy percentages and prediction confidence values) in the following metrics.

### Comprehensive test:

For our comprehensive test experiment. We used a complete board using all scrabble tiles provided in a 2021 release version set of Scrabble™ to assess the full capabilities of our character recognition in a realistic scenario if players were to use our application on a modern normal game of scrabble.



```
_ _ _ _ _ _ E G I N _ A I A _
_ _ E E A B _ _ _ _ _ B S S _
_ _ _ _ _ _ Q M A T _ _ _ _ _
O F K _ _ J R A _ _ _ _ I T _
_ _ _ _ _ _ _ _ N E F _ _ _ _
L A _ _ _ O T _ R O I W I X R
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _
O I V _ _ N L E D H Z W _ F O
_ _ _ _ _ _ _ _ _ I _ _ _ _ _
_ _ N _ H A O O _ U A _ _ O T
P I E P T _ _ _ _ F _ _ _ _ _
_ _ _ _ _ _ R L S E _ D O E G
V I _ T C N E _ _ _ _ _ _ _ _
_ _ _ G H R F _ _ M S O _ Y U
U L _ _ _ _ _ _ _ C Y A U _ _
```

Accuracy **(only characters)**: 91/97 = **93.8%**
Accuracy **(all tiles):** 249/255 = **97.7%**

Confidence values of prediction:

__ = predicted wrong character
__ = correct prediction but low confidence

```
[0.99, 0.99, 0.99, 1.0 , 0.99, 1.0 , 0.9 , 1.0 , 1.0 , 1.0 , 1.0 , 0.99, 0.97, 0.94, 1.0 ]
[1.0 , 1.0 , 1.0 , 1.0 , 0.98, 0.92, 1.0 , 1.0 , 1.0 , 1.0 , 0.99, 1.0 , 1.0 , 1.0 , 1.0 ]
[1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 0.98, 1.0 , 0.99, 1.0 , 1.0 , 1.0 , 0.99]
[0.9 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 0.98, 1.0 , 1.0 , 1.0 , 1.0 ]
[1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 0.69, 0.95, 1.0 , 1.0 , 1.0 ]
[1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 ]
[1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 0.99, 1.0 , 1.0 , 1.0 , 0.99, 1.0 , 1.0 , 1.0 ]
[1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 0.85, 1.0 , 1.0 , 1.0 , 0.99, 1.0 ]
[1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 0.99, 1.0 , 1.0 , 1.0 , 1.0 , 1.0 ]
[1.0 , 1.0 , 0.99, 1.0 , 0.68, 1.0 , 0.83, 1.0 , 1.0 , 0.99, 0.95, 1.0 , 1.0 , 1.0 , 1.0 ]
[1.0 , 0.75, 0.86, 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 ]
[1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 0.93, 1.0 , 1.0 , 1.0 , 1.0 , 0.84, 1.0 , 0.66, 1.0 ]
[1.0 , 0.96, 1.0 , 0.99, 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 ]
[1.0 , 1.0 , 1.0 , 1.0 , 0.99, 1.0 , 0.54, 1.0 , 1.0 , 0.99, 1.0 , 1.0 , 1.0 , 1.0 , 1.0 ]
[0.99, 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 0.99, 1.0 , 1.0 , 1.0 ]
```

**Average confidence (only characters): 0.9693**
**Average confidence (all spaces): 0.9861**
**Average confidence of wrong predictions: 0.7717**

### Analysis of our character prediction model:

It appears that given a complete 2021 version of Scrabble™, we had around a 97% accuracy on character recognition. Our model particularly struggled with distinguishing between characters 'E' vs 'F' and 'D' vs 'O', as these characters in Eurostile font share many geometrical similarities. Overall, we are very satisfied with the results of our character recognition model and provide means for manual correction by the player if wrong predictions are given, allowing the app to still effectively accomplish its purpose of providing the next move with the maximum possible score.

### Testing the limitations - edge cases:

We used a variety of images to stress test our application, as long as they held the assumption that the image contained a 15x15 board with scrabble tiles. As expected, our character recognition model accuracy drastically went down as we used

board images that differ from the 2021 release version set of Scrabble™.

Using `oddtest2.jpg`

```
V _ _ _ _ _ _ I _ _ _ _ G _ I 4 1
_ _ _ _ _ _ _ _ A W V U _ _ _ 4 3
_ _ _ _ _ _ _ _ _ E _ _ U _ _ 2 0
_ _ _ _ _ _ _ V _ N _ _ E _ _ 3 1
_ _ _ _ _ _ _ L A V E _ _ _ _ 4 3
_ _ _ _ _ _ _ D _ E _ _ _ _ _ 2 1
_ _ _ _ _ _ _ V _ _ _ _ _ _ _ 1 0
I _ _ _ _ U D V _ _ _ _ _ _ _ 4 0
_ _ _ _ A A U _ _ _ _ _ _ _ _ 3 2
_ _ _ _ _ D _ _ _ _ _ _ _ _ _ 1 1
_ _ _ _ _ U U E _ _ _ _ _ _ _ 3 2
_ _ B E B V _ _ _ _ _ _ _ _ _ 4 2
_ _ _ M _ _ _ _ _ _ _ _ _ _ _ 1 0
W A V E _ _ _ _ _ _ _ _ _ _ _ 4 4
I _ _ _ _ _ I _ _ _ _ _ _ _ _ 2 0
```

Accuracy **(only characters)**: 20/37 = **54.1%**
Accuracy **(all spaces):** 233/255 = **91.4%**

Confidence values of prediction:

___ = predicted wrong character
___ = correct prediction but low confidence

```
[0.88, 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 0.88, 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 0.9 ]
[1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 0.84, 0.78, 0.71, 0.99, 0.69, 0.99, 1.0 ]
[1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 0.99, 1.0 , 1.0 , 0.94, 1.0 , 1.0 ]
[1.0 , 1.0 , 1.0 , 0.99, 1.0 , 1.0 , 1.0 , 0.83, 1.0 , 0.94, 1.0 , 0.98, 0.55, 1.0 , 0.99]
[1.0 , 1.0 , 1.0 , 1.0 , 0.98, 1.0 , 1.0 , 0.95, 0.69, 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 ]
[1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 0.99, 1.0 , 0.42, 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 0.81, 1.0 ]
[1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 0.95, 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 ]
[0.63, 1.0 , 1.0 , 0.99, 1.0 , 0.84, 0.41, 0.94, 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 0.36]
[1.0 , 1.0 , 1.0 , 1.0 , 0.48, 0.71, 0.54, 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 ]
[1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 0.96, 1.0 , 1.0 , 1.0 , 0.98, 1.0 , 1.0 , 1.0 , 0.96, 1.0 ]
[1.0 , 1.0 , 1.0 , 1.0 , 0.99, 0.58, 0.93, 0.93, 0.99, 1.0 , 0.96, 1.0 , 1.0 , 1.0 , 1.0 ]
[1.0 , 1.0 , 0.71, 0.77, 0.66, 0.84, 1.0 , 1.0 , 0.98, 1.0 , 1.0 , 0.94, 1.0 , 1.0 , 0.99]
[1.0 , 1.0 , 1.0 , 0.51, 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 0.99, 0.99, 0.98, 0.76, 1.0 , 1.0 ]
[1.0 , 0.98, 0.56, 0.9 , 1.0 , 0.99, 1.0 , 1.0 , 1.0 , 1.0 , 0.98, 1.0 , 1.0 , 0.96, 1.0 ]
[0.29, 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 0.84]
```

```
Average confidence (only characters):    0.9593
Average confidence (all spaces):         0.7991
Average confidence of wrong predictions: 0.7662
```

## 4. Implementation

To train our neural network, we first used the PyTorch library to define our model. We uploaded our code to Google Colaboratory and used the GPU on the platform to speed up the training of our model. After the training was complete, the code was exported into a save file, which contained the weights of our neural network. This save file is loaded later in our application to generate predictions for the scrabble tiles.

When writing code for the homography transform of the original board input image, after the keypoints are generated and calculated using the Harris Corner Detector, a general homography transform was used. In order to do the homography transform, two OpenCV functions (getPerspectiveTransform and warpPerspective) were used.

In order to accomplish score maximization, the "World's Fastest Scrabble Program" algorithm [1] was used, with the majority of the implementation provided by the paper on GitHub. In order to suit this code base to our specific use case, a new entry point was coded for the algorithm, and a new board representation was created. Part of this was reconstructing the board using new data structures after searching the regular and transposed boards for each word alongside the index of the starting character of each word. The algorithm was further altered to better utilize and calculate scores using the "bonus" markers on the board such as double letter score or triple word score.

## 5. Data

We created our own dataset pipeline to train letter recognition

1. First we collect sample boards, both images of boards that we have taken as well as boards from the internet uploaded by Scrabble players. We made sure to select images of scrabble boards that were uncommon (different fonts, boards, image quality) when we saw them.
2. Because of the number of tiles we had to label, we implemented a training program in Python to speed up this process. Our training program draws a box around each tile that is to be labeled, and the box moves in a left-to-right, up-to-down sequence as the user works through all the tiles on the board. In comparison to our initial approach of simply cropping out the tile and having the user label that, this approach allows the user to "look ahead" when labeling the different tiles, reducing human error when labeling and improving labeling speed. Moreover, we designed the program in a way where only a single keystroke is required to label each tile. These optimizations resulted in a very fast labeling process— a full board can be labeled in less than 2 minutes.
3. We saved the cropped tiles and added the labels for each tile in its filename. These tiles were all added to a folder in preparation for our next step.
4. The tiles and their filename tag was fed into a custom PyTorch dataset that parsed the image and extracted its label from the filename, which was then fed into our machine learning model to generate our neural network learning model.

## 7. References


[1] Appel, A. W., & Jacobson, G. J. (1988, May 1). The world's fastest Scrabble program. Retrieved April 20, 2022.
[2] Homography Transform
[3] Neural network letter recognition
[4] T. E. de Campos, B. R. Babu and M. Varma. Character recognition in natural images. In Proceedings of the International Conference on Computer Vision Theory and Applications (VISAPP), Lisbon, Portugal, February 2009.
[5] Sandler, Mark and Howard, Andrew and Zhu, Menglong and Zhmoginov, Andrey and Chen, Liang-Chieh. MobileNetV2: Inverted Residuals and Linear Bottlenecks
[6] https://opencv24-python-tutorials.readthedocs.io/en/latest/py_tutorials/py_feature2d/py_features_harris/py_features_harris.html