Generated Patterns for Evaluating Machine Learning Model Efficiency in Computer Vision

James Piotrowski

February 1, 2024

Abstract

With the evergrowing need for parallelism in machine learning, programming languages using the Process-Oriented Paradigm (POP), such as ProcessJ, should be considered as they offer more intuitive and safe methods for implementing parallelism. To showcase ML and POP together, models are developed using both ProcessJ and modern popular languages (like C++) and compared with each other. Because the implementation of parallelism is a main consideration in evaluation, models of different magnitude need be considered to fully observe the parallel capabilities. Thus, datasets with a scalable magnitude should be used for consistent and reliable evaluations. Programatically generating simple image data for classification offers an effective solution.

Keywords: computer vision, image generation

1 Introduction

Efficient model design is a crucial aspect of machine learning (ML). With modern Big Data, ML models learning data could take days, weeks, or even months. The time spent training costs both energy and time. The faster a model can be trained, the quicker it can be evaluated and used. Researchers working with big data often have super computers with as many as 285,000 CPU cores and 10,000 GPUs [2] to not only handle the size of the data, but to expedite the training process. The energy resources to power such machines do not come cheap. Many businesses and researchers will look to lower both their energy bill and the carbon dioxide emissions produced by their hardware [1]. Given these facts, optimizing ML models for both speed and hardware usage is essential. With the continuing advancement in raw hardware speed being questionable [4], parallelization is a fundamental and obvious requirement for improving model efficiency.

Most modern programming languages implement parallelism in some way. Unfortunately, the implementation is often unintuitive and difficult to reason, resulting in a high probability to create unsafe code [5]. When picking a language to build ML models, the programmers knowledge of parallelism and the parallel constructs present in that language is important. Considering that the subject of ML is dense in statistics and mathematics, it is possible that the programmer does not have the programming knowledge necessary to build the most efficient model in their chosen language. For this reason and more, Process-Oriented programming languages that provide intuitive and safe methods to implement parallelism, like Occam and ProcessJ, should be considered for models that require a high level of parallelism.

As of the time of this paper, the Process-Oriented paradigm is not popular nor widely developed. To explore the viability of these languages in machine learning, large data sets are needed to truly evaluate their efficiency when compared with popular modern day languages such as C++, C#, and Java. Generated Patterns offers a simple, effective and scalable solution to programmers looking to evaluate the efficiency and effectiveness of their ML models.

2 Scalable Data Challenges & Requirements

When comparing the parallel effectiveness of two or more ML models, the magnitude of both the data and model are a crucial consideration. Each model may exceed in different aspects, thus it is important to provide multiple environments to illuminate each models strengths and weaknesses. Varying the size of the data set is important to fully observe how the models behave.

Finding data sets with variable size would be sufficient for testing, though there are two major challenges associated with this solution. First, when acquiring data sets it would be important to verify that the set meets requirements in the following areas:

- Sufficient volume of samples.
- Sufficient uniqueness of samples.
- Balanced number of samples per class.
- Sufficient sample dimensionality.

Many data sets may meet the mark on some of the requirements, but may not meet all of them. This would require the altering of the data set or the finding of a new one. The second challenge involves the comparison of results. When evaluating models of different magnitudes using different data sets, the results may vary significantly between evaluations. For example, a Convolutional Neural Network (CNN) used to learn three different data sets could result in accuracies of 99%, 72%, and 48%. Additionally, because the CNN is being compared with another CNN on the same data set, the first CNN may exceed the other on some data sets but not the others. This may make it difficult to draw confident conclusions on which may be better. For these reasons, a single data set with scaling capabilities would be preferred as the requirements need only be met for one data set and hypothetically, the results should remain relatively consistent despite the variability in model magnitude.

2.1 Requirements

The goal is to have a data set that is optimal for evaluating model efficiency, thus, a data set with the following requirements would be the most valuable.

- Not overly complex to ensure a high accuracy is achievable.
- Significant volume of unique samples.
- Samples distributed evenly over all classes.
- Scalability.

A dataset like MNIST [3] meets many of these requirements. The handwritten digits are easily recognized, the data can be simply represented as an array of bits or numbers, each sample is unique, and the number of samples per class is nicely balanced. The only requirement in question is the scalability.

3 Methodology

There are only two methods for acquiring a scalable data set. Either each sample in an already existent data set is altered to meet the required scale, or a data set it generated to meet the required scale.

3.1 Scaling Images

As mentioned in section 2.1, an already existent data set could be acquired that meets all requirements except for scalability. Therefore, to meet this requirement the samples of the data set must be scaled. It is possible to simply scale the images into larger or smaller dimensions, though the preservation of data quality as each sample is stretched or compacted is a concern. Depending on the original image dimensions, a basic upward scaling operation on the image may result in a rather choppy new image. Considering MNIST as an example, the original images are 28x28. To simply scale a sample without smoothing the edges may result in a rather distorted image. Down-scaling an image could result in loss of information. Additionally, if we were to scale the image into a different representation in which the ratio is not the same as the original, the dimensions of the image would be stretched, which may result in a somewhat misrepresented image. When dealing with large datasets containing thousands of images, it will be difficult to verify that the scaling safely translated all the samples into the proper scaled images. An example of this would be an image of the digit "1" scaled horizontally would look very much like a "7".

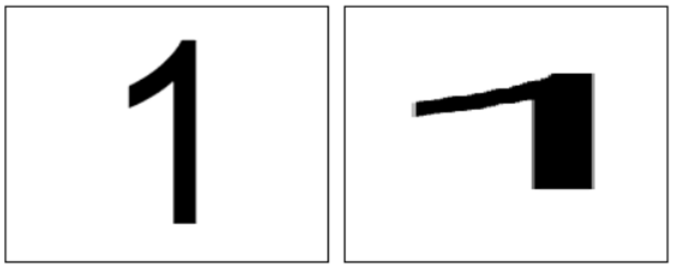


Figure 3.1: *An image of a "1" represented as the original (left) and with a scaled alteration (right).*

When applying scaling to a data set, it may be in the programmers best interest to spot-check all the samples to make sure the information is still being properly communicated, which is rather time consuming and tedious. For these reasons, scaling already existing data sets was not considered as a solution.

3.2 Generating Images

Instructing a computer to generate an image given the required dimensions easily solves the scaling problem described above. Many shapes can easily be plotted in an image using computational geometry. The computer would plot an image of a circle with a radius of 20 pixels just as easily as one with 40 pixels. Because the mathematics remain consistent despite varying dimensions, resulting images can be guaranteed to most accurately represent their class without the worry of distortion or misrepresentation.

With the scalability problem resolved, the other requirements must still be met. Because the data is being generated, there are many solutions to achieving the requirements. The most apparent solution is to use geometric shapes and symbols. These images can be easily classified. They can also be simply & uniquely represented. Additionally, because the data is being generated, balancing the sample per class distribution is completely configurable.

3.2.1 Simplicity

The data set should contain samples that are simple and easily classified. Because the images are being generated, it is important to ensure that the images are straightforward to produce. Basic shapes and symbols such as circles, triangles, and hexagons are some of the best images to utilize in meeting this requirement. They can be easily recognized by past and modern ML models. These shapes can also be represented as an array of bits (*if using black and white*), integers, or decimals. Additionally, many shapes and symbols have well defined modules to generate images of them.

3.2.2 Uniqueness

At first glance, there are not many ways to uniquely represent shapes like a circle or a square. A good starting place is to generate shapes of different sizes within the same sized image. In a 50x50 image, a circle with a radius of 20 pixels could plotted, as well as a circle with a 21 pixel radius or a 22 pixel radius. By plotting shapes of different sizes within a canvas of constant dimension, we are able to generate multiple unique representations of the class. But this is not enough scale. The best data sets have a significant number of samples per class. To reach large volumes of unique samples using shapes, we plot multiple shapes on the same image. For example, assuming a small circle, a medium circle, and large circle are generated, each of these unique instances can be combined into many unique images. Consider creating an image with up to 4 circles in it. Each of the unique circles could be chosen for each spot, resulting in 3^4 possible unique images. We can also control where each circle appears in the image, allowing the generation of even more unique images. For more distinct shapes, we can also configure how each shape is rotated within the image. With the exponential nature of this method, it is clear that generating significant unique sample volume is no longer a concern.

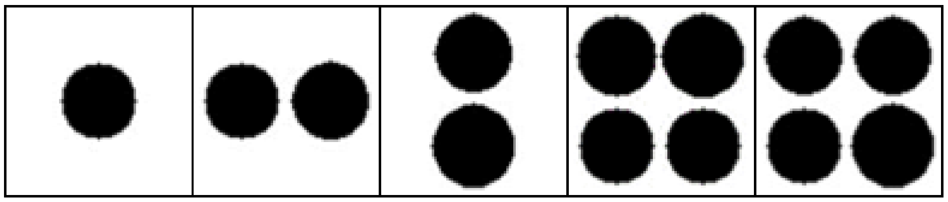


Figure 3.2.2: *Unique samples of images containing circles.*

3.2.3 Balanced Sample per Class Distribution

This requirement is simple to achieve by controlling the number of images generated per class. The only consideration regarding this requirement is how many ways a class can be uniquely represented. Depending on the shape, one class may have more unique representations than the other. If this is the case, the data set's volume will be limited by the class with the least amount of unique representations.

4 Method Generalization & Definition

4.1 Pattern-Units

To begin, for the desired number of classifications in the data set, program modules to generate a **Pattern-Unit** representing that class must be developed. Pattern-Units are images used to create larger images. A pattern-unit must have predetermined dimensions. Once determined, the program modules must take at least one configurable parameter called a **scale** that dictates the size of the shape within the pattern-unit. This is done using a percentage, which informs the module how much space the shape can occupy on the 2-dimensional grid. For example, in a predetermined grid size of 50x50, a circle generated using 80% as the scale will result in a circle having a diameter that is $(0.8 * 50) = 40$ pixels. All pattern-units will have the same dimension throughout the entire data set, though the size of the shapes within the units will vary. Multiple scales can be provided to a module if the shape that is plotted in the unit has the capability to be scaled in multiple ways. For example, a rectangle has two scales: the width and the height. Regardless of the height and width of the rectangle, it will still always be a rectangle, thus, a pattern-unit can contain a rectangle that occupies 20% of the width and 90% of the height. This only works for some shapes. A circle and a square by definition can only be scaled one way. Rectangles and ovals have two scales. More complicated shapes, such as an octagon, have more than just 2 scales.

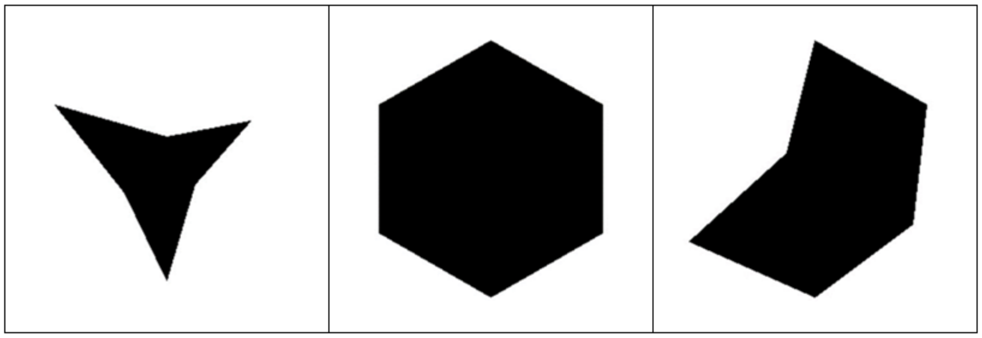


Figure 4.1.1: *A hexagon has 6 possible scales which when configured differently may result in very diverse images.*

The modules may also take an additional **rotation** parameter that instructs the module how to rotate the shape when generating it. This parameter is not implemented in this paper for reasons described in ***Section 4.2.4***.

20 pattern-unit modules were included in the first iteration of this data set. Most are polygons and were developed using computational geometry methods. A few of the shapes are more complex or are not polygons, thus the modules developed to generate them are custom methods. The 20 pattern units are listed below:

0. Square	1. Rectangle	2. Diamond	3. Triangle
4. Circle	5. Hexagon	6. Pentagon	7. Heptagon
8. Star	9. Octogon	10.Trapezoid	11.Heart
12.Cross	13.Crescent	14.Spike	15.Arrow
16.Tilde	17.Zigzag	18.Cane	19.Cat

Examples of these pattern-units are observable in ***section 5.1****.*

4.2 Patterns

Once the pattern-unit modules are developed and ready to use, the creation of **Patterns**, which will be the images for the data set, is ready to begin. The first step is to specify a two-dimension image size that is either the same size or larger than the pattern-units. Next, generate the possible variations of the pattern-units and place them within the pattern. To generate many unique patterns, each individual sample must have the pattern-units placed in a different way. When choosing how to place pattern-units within the larger pattern, there are four main options:

- **Clipping -** Pattern-units only partially represented due to dimension limitations.
- **Scaling -** Pattern-units ability to be uniquely represented by adjusting their scale(s).
- **Rotation -** Pattern-units ability to be uniquely represented by rotation.
- **Placement -** Patterns ability to be uniquely represented by the varying placement of pattern-units.

If these options are not sufficiently considered and limited, the number of unique representations per class pattern-unit may be too large.

4.2.1 Volume of Combinations

With each level of variation used to transform a pattern-unit, the possible number of unique samples grows exponentially. There are two main considerations when choosing the ways a pattern-unit can be altered.

Processing Power: A data set should be generated in a reasonable amount of time. Whether that amount of time is a few seconds or a few weeks, the data set should be generated within enough time to use it. With N being the number of unique variations a pattern-unit can be placed in a larger image and P being the max amount of pattern-units being placed in that image, the approximation of all unique possible images is $\sum_{i=1}^{P} N^i$. This approximation does not consider edge cases that may result in non-unique images, such as larger variations of pattern-units being placed on top of smaller ones which would result in the smaller shape being obscured. Regardless, the difference is negligible when considering this level of exponentiation.

Class Balance: The second consideration is how each pattern-unit can be uniquely represented. Certain shapes when transformed in some way will not result in a new unique representation of that shape. Consider a circle, a square, and a shape that has no symmetries (*such as a cane*). If we rotate each of these shapes 1 degree for 360 total degrees, the circle will still only have one unique representation. The square will have 90 unique variations, and the non-symmetrical shape will have 360 unique variations. Considering the level of exponentiation described above, if we do not restrict the ways in which a pattern-unit can transform, the resulting data set will be heavily unbalanced. Using the example of the circle, square, and cane, if we were to not restrict rotation on the pattern-units, depending on the other variation methods present, the non-symmetrical shape would generate enough unique images to represent over 99.99% of samples in the data set. Almost any model designed to classify this data set would be able to achieve almost perfect accuracy just by classifying the non-symmetrical class for each sample.

4.2.2 Clipping

Clipping occurs when only a partial pattern-unit is placed due to size restrictions of the pattern. Consider a pattern that has a height 320 pixels and a width of 420 pixels. If the pattern-unit has dimensions that do not evenly divide the larger dimensions, like 100 by 100, some of the pattern-units will be cut off when generating the larger image.

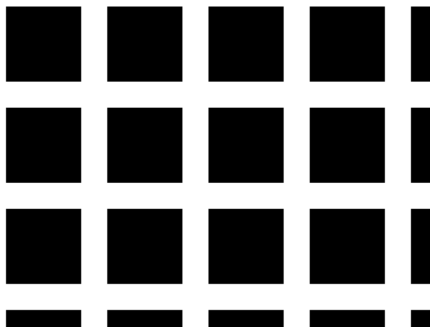


Figure 4.2.2.1: *An example of clipping. The squares on the right and bottom do not have enough space to be fully represented.*

Clipping poses a challenge to automatic classification because not all pattern-units within the pattern represent the class accurately. In *Figure*

4.2.2.1 some of the pattern-units are not squares, but rather rectangles. The image is made up mostly of squares so it may be trivial to identify that this image belongs to the square class, but consider samples where the ratio of whole pattern-units to partial pattern-units is 1:1, or where there are more clipped units than whole units. If there are other classes present in the data set that represent the clipped units (*like rectangles in the above example*), it will be more difficult to properly classify those samples.

It may be useful in some cases to have samples that are difficult to identify, though it is not important for evaluating efficiency. Because of this, clipping is avoided in the data sets generated for this paper, though it is an option in the program that is available on GitHub.

4.2.3 Scaling

Scaling a pattern-unit is a simple and effective way to generate unique samples. As mentioned in ***Section 4.1***, certain shapes may have more scaling capabilities than others. Procedural generation of all unique representations of a shape where the shape is able to be scaled multiple ways will result in exponentially more unique images than that of shapes with only a single scaling option. As emphasized in ***Section 4.2.1***, the exponential nature of multiple scales will result in the need for more processing power and yield a final data set with an uneven distribution of samples among the many classes, unless restricted in some way. Additionally, multiple scales allow for a much wider variation in the unique representations of that shape (*consider the hexagons in figure* ***Figure 4.1.1***). The wider variation in the shape representation will pose a challenge to automatic classification that may be desired in certain situations. However, great care must be taken when choosing this route because certain shapes when scaled in specific ways will result in images of another class. Consider that all squares are rectangles. When the width and height scale are equal to each other for a rectangle, an image is generated that is both a rectangle and a square. If asking the model to properly classify images with these properties, it will be impossible for the model to discern which image is a square or a rectangle. To avoid this, either the square class must be completely omitted from the data set, or all rectangles that are also squares must be removed from the rectangle class. There are also cases where a shape may be scaled in such a way that it is no longer representative of it's class. For example, a hexagon could be scaled such that the resulting image is a triangle. To visualize this, observe ***Figure 4.2.3.1*** below.

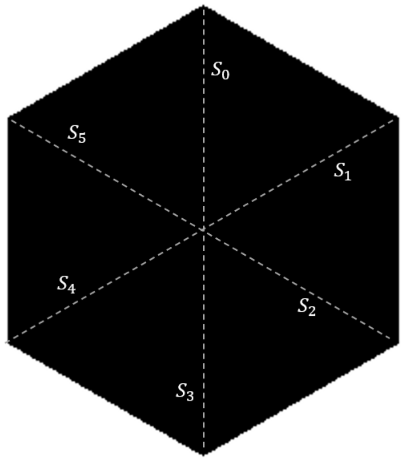


Figure 4.2.3.1: *Six lines are drawn from the center of the hexagon to where it's edges meet. The length of these lines are determined by scales used to generate the image.*

In any case where $S_0 = S_2 = S_4$, $S_1 = S_3 = S_5$, and $S_0 = 2S_1$ the resulting shape is a triangle.

Figure 4.2.3.2: *A triangle generated using the Hexagon module.*

As previously described, training a model with data samples like this is undesired. To avoid this, each module created to generate images must come with functionality to prevent the generation of an incorrect representation.

Allowing pattern-units to scale in multiple ways will yield many unique and effective samples for a data set, yet it requires a considerable amount of design and care. For these reasons, all pattern-units used in the final data sets for this paper were only scaled with one parameter. This gives complete control over balancing the data set as well as generating accurately represented classes.

4.2.4 Rotation

Rotation is a simple transformation that can be used on pattern-units to generate more unique samples for data sets. It also adds significant complexity. As with all transformations to pattern-units, rotating the shapes will require am exponential demand for more processing power. Also, as mentioned in ***Section 4.2.1***, certain shapes may have more unique representations than others depending on how they are rotated, making the volume of class distributions difficult to control. Lastly, rotating a shape may result in a representation of another shape. In this paper, diamonds and squares are both used as classes. Diamonds can be rotated into squares, and vice-versa. This is very similar to the multiple scales issue described in ***Section 4.2.3***. Rotation would provide another level of classification difficulty, but it is not needed to evaluate model efficiency. Due to the complications rotation presents, it was not implemented for this paper.

4.2.5 Location

The location in which pattern-units are placed within the larger pattern is perhaps the most powerful variation that allows the generation of so many unique samples. Additionally, the number of pattern-units placed plays a key role. The main consideration when placing a pattern-unit is if it will be correctly represented. If clipping is allowed and the center of the unit be placed close to the edge of the larger image, the unit may clip on the edge. Also, if a unit is placed such that it overlaps with a previously placed unit, units may be completely omitted or combined into a shape that looks nothing like the represented class. To avoid misrepresentation, overlapping pattern-units is not allowed.

Even with the restriction of no overlap, generating all possible variations of an image with varying representations of pattern-units, numbers of units, and locations of units is exponentially infeasible. To combat this, a simple system is designed to iteratively generate reasonable amounts of unique sample data. When generating a larger image, two parameters named the

Horizontal Offset and **Vertical Offset** are used to control how many pixels pattern-units must be away from each other. These offsets are constant throughout the entire image. So, if there are multiple units in an image, all units will be equally spaced horizontally and vertically.

Additionally, centering the resulting grid of pattern-units is enforced to control the number of images generated. In cases where just one unit is being placed in a much larger image, that pattern-unit could be moved just one pixel in either direction to generate a new image where the same unit appears in a different location. Considering the size of the data being generated, that single unit may be shifted thousands of times to create thousands of images, in which the process is repeated for all unique representations of that pattern-unit. This is a simple method to generate tons of sample data, though it will require tons of memory and processing power. Considering all of the other methods used to vary images, the volume generated by allowing this is not necessary.

4.3 Final Method

First, insert z classes to generate into a set P. Then, choose the pixel width and height dimensions for the pattern-units y and x. Next, choose the pixel width and height dimensions for the pattern h and w where $y < h$ and $x < w$. Now, choose a **minimum scale** m which restricts how small the pattern-unit can be, a **maximum scale** l that restricts how large the pattern-unit can be, and a **scale step** s which controls how much to increase the scale when generating the variations of pattern-units. Lastly, choose a percentage of combinations to keep c which helps mitigate the volume of samples generated if needed.

The next step is to determine the number of vertical offsets v and horizontal offsets g that will be used. Using the above parameters, the numbers can be determined with the following equations:

$$v = \begin{cases} \lceil (\frac{h}{2} + 1) - y \rceil, & \text{if } \frac{h}{y} \geq 2 \\ 0, & \text{otherwise} \end{cases}$$

$$g = \begin{cases} \lceil (\frac{w}{2} + 1) - x \rceil, & \text{if } \frac{w}{x} \geq 2 \\ 0, & \text{otherwise} \end{cases}$$

For all output classes, for all unique representations of each class's pattern-unit, for all vertical offsets from 0 to v, for all horizontal offsets from 0 to g, then for c of all possible combinations of pattern-units using the offsets, generate a pattern for the final data set. This process is summarized below in C++:

```
// set to hold samples
array<pattern*> dataSet;
// For all output classes
for(patternType p in P){
  // Create a set to hold all unique representations of pattern-unit.
  array<patternUnit*> uniquePatternUnitSet;
  // for all scales between m and l(L) incrementing by s
  for(double cs = m; cs <= l; cs += s){
    // Generate a pattern-unit at the current scale
    patternUnit *newPatternUnit = new patternUnit(p, cs);
    // Add it to the set
    uniquePatternUnitSet.Add(newPatternUnit);
  }
  // For all vertical offsets from 0 to v
  for(vO = 0; vO <= v; vO += 1){
    // Compute the number of pattern-units that can fit vertically in the
  pattern
    int numVerticalUnits = h / (y + vO);
    // For all horizontal offsets from 0 to g
    for (hO = 0; hO <= g; hO += 1) {
      // Compute the number of pattern-units that can fit horizontally in
  the pattern
      int numHorizontalUnits = w / (x + hO);
      // Determine number of pattern-units in the pattern
      int n =  numVerticalUnits * numHorizontalUnits;
      // Determine total number of unique pattern-units
      int u = uniquePatternUnitSet.size();
      // Generate all possible combinations of n with p possible pattern-
  units. Only keep c combinations
      array<combination> combinations = getCombinations(n, p, c);
      // For all combinations
      for (combination k in combinations) {
        // Generate pattern using the combination, patternType, vertical
  and horizontal offsets, and the unique pattern unit set.
        Pattern p = generatePattern(p, vO, hO, k, uniquePatternUnitSet);
        // Add to data set
        dataSet.Add(p);
      }
    }
  }
}

```

The size of the data set d can be estimated by:

$$d = z \sum_{i=0}^{v} \sum_{j=0}^{g} \left(\frac{l-m}{s}\right)^{\left(\left(\frac{h}{y+i}\right)\left(\frac{w}{x+j}\right)\right)}$$

Due to certain implementation restrictions, the number calculated using the equation above may not result in the exact number generated by the program available on github. This is because pattern-units need a minimum number of pixels to generate a reasonably discernible pattern. If the minimum scale m chosen will result in pattern-units smaller than 30 pixels in any dimension, m will be increased to meet the 30 pixel minimum. Additionally, the combinations to keep percentage c is slightly adjusted to $\frac{1}{[1 \div c]}$ (*for example,* $c = 0.3$ *is adjusted to* $c = \frac{1}{3}$). c is used to alleviate the computational strain of computing all combinations. Altering c so that it can be represented as 1 over some number greatly simplifies the implementation of generating some combinations whilst also maintaining a fair mix of the combinations (*for example, only compute 1 out of every 3 combinations*).

5 Evaluation of Generated Data Sets

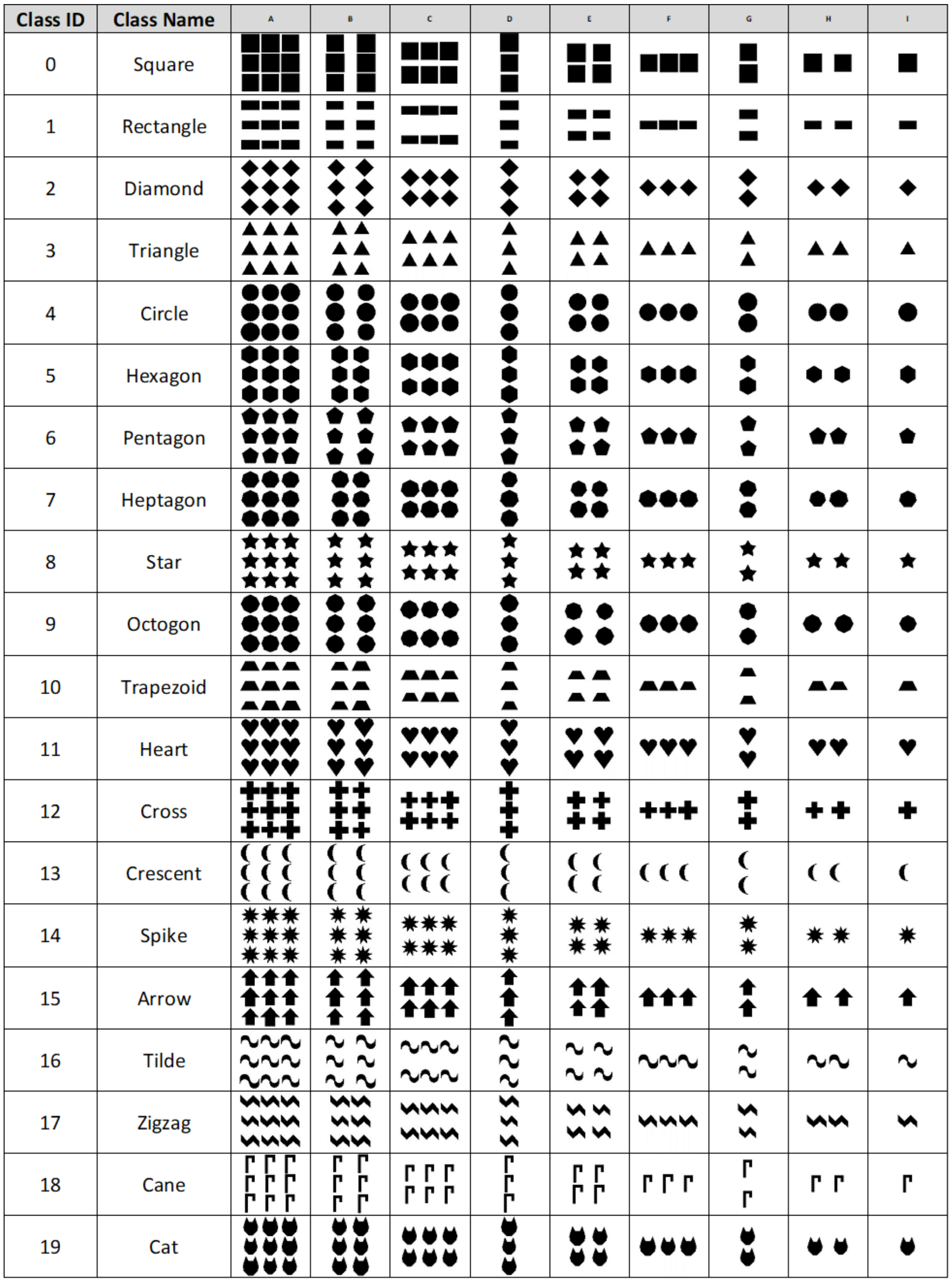
To verify that the generated pattern sets are a good solution to evaluating model efficiency, multiple data sets at different scales are generated and then evaluated using a simple model to prove that the data is automatically classifiable and consistent despite variations in scale.

5.1 Data

Three data sets were generated for this paper:

- [**75***x***75 Data Set**]: 35 by 35 Pattern-Units, 20 Classes, 50240 unique samples.
- [**110***x***110 Data Set**]: 35 by 35 Pattern-Units, 20 Classes, 288200 unique samples.
- [**65***x***465 Data Set**]: 50 by 50 Pattern-Units, 20 Classes, 60480 unique samples.

Each data set is composed of generated images from the 20 classes listed in ***Section 4.1***. The figure below details all the unique orientations of the generated data in the [110x110 Data Set].



Class ID	Class Name
0	Square
1	Rectangle
2	Diamond
3	Triangle
4	Circle
5	Hexagon
6	Pentagon
7	Heptagon
8	Star
9	Octogon
10	Trapezoid
11	Heart
12	Cross
13	Crescent
14	Spike
15	Arrow
16	Tilde
17	Zigzag
18	Cane
19	Cat

Figure 5.1.1: *All Possible Orientations for the [110x110 Data Set].*

The [75x75 Data Set] samples look similar to those depicted in *Figure 5.1.1* except the images are smaller and there are no orientations that contain 3 pattern-units in any dimension. The [65x465 Data Set] is longer with each image containing up to nine pattern-units length-wise and only one pattern-unit height-wise.

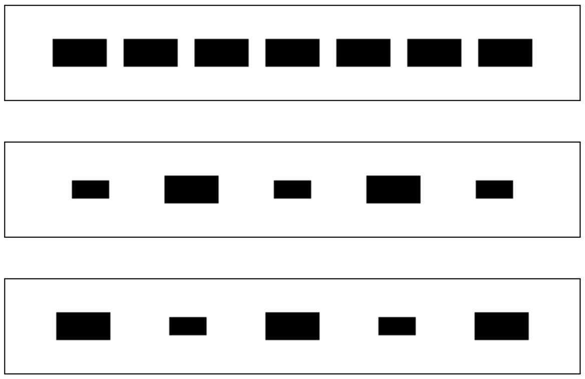


Figure 5.1.2: *Example samples from the [65x465 Data Set].*

5.2 Classification

Each data set is randomly split into 80% for training and 20% for testing where each class maintains an even distribution in each set. Three fully connected neural networks (FCNN) were trained to classify the three datasets. The classification program is available on GitHub. The FCNNs are implemented using C++ and posix threads to achieve maximum possible efficiency without the use of GPUs. Each FCNN trained on it's respective data set and then the training accuracy, testing accuracy, and speed were evaluated. Each FCNN was tested multiple times with different numbers of threads to determine the optimal number of threads to achieve the best speed. All FCNNs were evaluated on the same machine which is described below.

5.2.1 Hardware

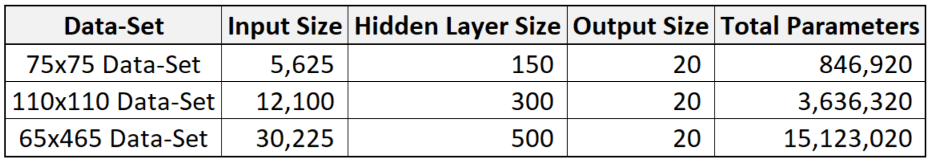
A machine named "Pikachu" running Ubuntu 22.04.3 LTS with two AMD EPYC 7452 32-Core Processors (*128 cores*) at 2.35GHz and 512 Gigabytes of memory was used to train and evaluate.

5.2.2 Optimizations

C++ STD 20 and Object Oriented programming were used to implement a FCNN that has configurable layers, activation functions, and posix threads. The executable is compiled using the g++ compiler with optimization level 2 (*”-O2”*).

5.2.3 FCNNs

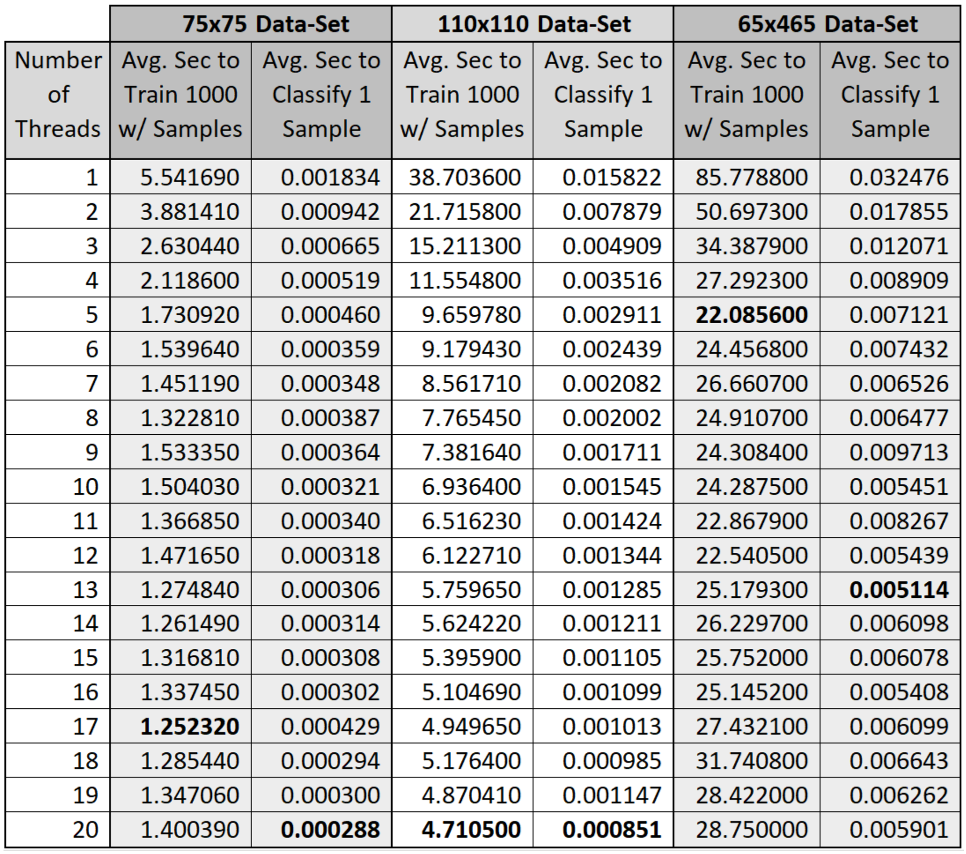
The layout for each data set and it’s respective FCNN is tabulated below:



Data-Set	Input Size	Hidden Layer Size	Output Size	Total Parameters
75x75 Data-Set	5,625	150	20	846,920
110x110 Data-Set	12,100	300	20	3,636,320
65x465 Data-Set	30,225	500	20	15,123,020

Figure 5.2.3.1: *Summary of FCNN Architecture for each Data Set.*

Each FCNN uses sigmoid as the activation function throughout all the layers. The results of all FCNNs speed when tested with different threads is tabulated below. Each FCNN was tested up to 20 threads.



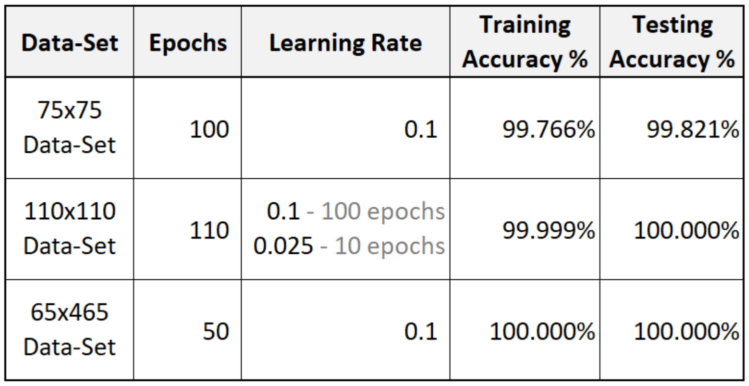
	75x75 Data-Set		110x110 Data-Set		65x465 Data-Set	
Number of Threads	Avg. Sec to Train 1000 w/ Samples	Avg. Sec to Classify 1 Sample	Avg. Sec to Train 1000 w/ Samples	Avg. Sec to Classify 1 Sample	Avg. Sec to Train 1000 w/ Samples	Avg. Sec to Classify 1 Sample
1	5.541690	0.001834	38.703600	0.015822	85.778800	0.032476
2	3.881410	0.000942	21.715800	0.007879	50.697300	0.017855
3	2.630440	0.000665	15.211300	0.004909	34.387900	0.012071
4	2.118600	0.000519	11.554800	0.003516	27.292300	0.008909
5	1.730920	0.000460	9.659780	0.002911	**22.085600**	0.007121
6	1.539640	0.000359	9.179430	0.002439	24.456800	0.007432
7	1.451190	0.000348	8.561710	0.002082	26.660700	0.006526
8	1.322810	0.000387	7.765450	0.002002	24.910700	0.006477
9	1.533350	0.000364	7.381640	0.001711	24.308400	0.009713
10	1.504030	0.000321	6.936400	0.001545	24.287500	0.005451
11	1.366850	0.000340	6.516230	0.001424	22.867900	0.008267
12	1.471650	0.000318	6.122710	0.001344	22.540500	0.005439
13	1.274840	0.000306	5.759650	0.001285	25.179300	**0.005114**
14	1.261490	0.000314	5.624220	0.001211	26.229700	0.006098
15	1.316810	0.000308	5.395900	0.001105	25.752000	0.006078
16	1.337450	0.000302	5.104690	0.001099	25.145200	0.005408
17	**1.252320**	0.000429	4.949650	0.001013	27.432100	0.006099
18	1.285440	0.000294	5.176400	0.000985	31.740800	0.006643
19	1.347060	0.000300	4.870410	0.001147	28.422000	0.006262
20	1.400390	**0.000288**	**4.710500**	**0.000851**	28.750000	0.005901

Figure 5.2.3.2: *Speed Test for each FCNN Architecture.*

Based on the results in the above table, the [75x75 Data Set] was trained with 17 threads, the [110x110 Data Set] was trained with 20 threads, and the [65x465 Data Set] was trained with 5 threads.

6 Results

Each FCNN was trained until reasonable accuracy was reached. The details of the training can be seen below:



Data-Set	Epochs	Learning Rate	Training Accuracy %	Testing Accuracy %
75x75 Data-Set	100	0.1	99.766%	99.821%
110x110 Data-Set	110	0.1 - 100 epochs 0.025 - 10 epochs	99.999%	100.000%
65x465 Data-Set	50	0.1	100.000%	100.000%

Figure 5.2.3.2: *Accuracy Test for each FCNN + Data Set.*

It is believed that each data set could easily reach 100% classification accuracy with proper tuning of the models, though the goal of this paper is to prove that the data can be automatically classified with reasonable accuracy. $99+\%$ absolutely solidifies that this generated data can be recognized, even when scaled in different ways.

7 Discussion & Future Work

The goal of this paper was to generate scalable data for automatic classification in computer vision and that goal was achieved. This generated data will be used to compare machine learning models built in process-oriented programming languages and models built in other widely used languages that are object-oriented. The data sets scalability and simplicity will allow our future work to focus on model efficiency whilst still proving effectiveness.

Regarding the work done to generate the data, there will likely be no more work done unless a need presents itself. If more work is to be done, here are some areas that should be explored:

- Development and evaluation of patterns with rotating pattern-units.
- Development and evaluation of patterns which contain pattern-units that are scaled with more than just a single scale.
- Development and evaluation of patterns that contain units with non-static spacing.

- Evaluation of much larger patterns.
- Evaluation of patterns with clipping.
- Evaluation of patterns with and without centering.
- Optimizing the generation of data with significant exponential factors.
- Generating patterns with a non-binary color scale.

References

[1] Omar Y. Al-Jarrah, Paul D. Yoo, Sami Muhaidat, George K. Karagiannidis, and Kamal Taha. Efficient machine learning for big data: A review. *Big Data Research*, 2(3):87–93, 2015. Big Data, Analytics, and High-Performance Computing.

[2] Jennifer Langston. Microsoft announces new supercomputer, lays out vision for future ai work, May 2020. [Online; posted 19-May-2020].

[3] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[4] Mark S. Lundstrom and Muhammad A. Alam. Moore's law: The journey ahead. *Science*, 378(6621):722–723, 2022.

[5] Matthew Sowders. Processj: A process-oriented programming language. Master's thesis, Digital Scholarship@UNLV, 2011.