Using Fonts to Train Neural Networks for Handwritten Character Recognition

To what extent can computer fonts be used to create a dataset to train machine leaning models to classify handwritten characters and digits?"

Computer Science Extended Essay

Word Count: 3624

Table of Contents

1. Introduction

Machine learning (field of computer science that aims to teach computers how to learn and act without being explicitly programmed[1]) is field in computer science that has gained substantial popularity in the last decades. Applications of machine learning can be seen across all aspects of life: digital assistants; website recommendations; spam detectors; medical image analysis; and recently self-driving cars[2]. With large amounts of data getting more accessible and computing becoming more powerful and affordable, the field of machine learning will continue to grow becoming ever more integrated into our personal and work lives[3].

Due to the growing popularity in this subject area many newcomers will look for a place to start. Most will begin with the MNIST dataset, a set of 70.000 small, greyscale images of handwritten digits from students and employees of the US Census Bureau. Programming a classification algorithm to identify these digits and characters has become a rite of passage for anyone interested in the field, often referred to as the "Hello World" of machine learning[4]. In 2017 an extension to the MNIST dataset called EMNIST was published containing 280.000 images consisting of both handwritten digits and characters[5].

Such large, labeled datasets are often difficult to come by as it is often difficult to collect and accurately evaluate such large amounts of data. Especially with handwritten digits and characters it is an arduous task to collect images of these glyphs, format them, and correctly label them. Though getting access to large amounts of different labeled characters and digits may not be as difficult as it seems.

[1] (DeepAI)
[2] (IBM Cloud Education)
[3] Ibid.
[4] (Géron)
[5] (Cohen, Afshar and Tapson)

When one loads up any modern word processing software, the first thing one usually changes is the font. Most systems come with a large number of preinstalled typefaces and if none of them are of your liking one can find countless other fonts online. Many websites such as Google Fonts, DaFont and Adobe Fonts, to name a few, offer numerous different fonts of varying styles. Certain fonts are also made to mimic handwritten letters and numbers and are often based on real handwritten glyphs.

Fig. 1 ***Examples of fonts that mimic handwritten glyphs***

This means that there are large amounts of labeled handwritten character and digit data readily available through computer fonts. Thus, this paper hopes to answer the question: **"To what extent can computer fonts be used to create a dataset to train machine leaning models to classify handwritten characters and digits?"**

2. Background Information

2.1 Data in Machine Learning

Machine Learning is the study of computer algorithms that improve automatically through experience[6]. It describes the construction of models that can learn and make predictions on data[7]. In the field of computer vision, a subcategory of machine learning and the focus of this

[6] (Mitchell)
[7] (Kohavi and Provost)

paper, data will come in form of images. These images could depict many things, animals, shapes, clothing, etc. A common task in computer vison would be to create a model to classify these things. Let us say we intent to create an algorithm to identify cats, to do so we would create a model and feed it many images of cats. Over time the model would recognize certain patterns and characteristics that the images have in common, this could be the general silhouette of the animal, the color of the fur, the size of the eyes, etc. The more data that is fed into the model, the more characteristics the model can recognize. Given enough data we will eventually have a model that can accurately recognize images of cats. This is the main reason why data plays such a fundamental role in machine learning; it gives the model information on what common attributes and characteristics it should focus on to make an accurate prediction.

Data plays a fundamental role in the field of machine learning as it used to train models to complete certain tasks.

2.2 Dataset Structure

The right end format for deep learning is generally a tensor, or a multi-dimensional array[8]. Therefore, data that is used in deep learning will generally be converted into vectors or tensors, to which linear algebra operations can be applied[9]. One can imagine that all this data is stored in a large table. Each row represents a single entry in the dataset, these are referred to as instances (or entry, sample, case). The columns represent the properties of each instance, there are referred to as attributes (or features, variables)[10]. For example, if a dataset stores the addresses of each client, each client would be its own instance. Every one of these instances would include attributes such as the name of the client, the street name, street number, country etc. All this data would be stored in tensors.

[8] (A.I. Wiki)
[9] Ibid.
[10] (Brownlee, Why Data Preparation Is So Important in Machine Learning)

Certain columns/tensors are defined as input and output variables. The input tensors are the columns the dataset provides to a machine learning model to make a prediction. The output tensor or column contains the target that the model should predicted[11].

In the font dataset each entry has two attributes, the image label, and the image itself. The images are converted into 2D tensors so that they can be inputted into the classification model. The image tensors are the input columns and the labels the output columns.

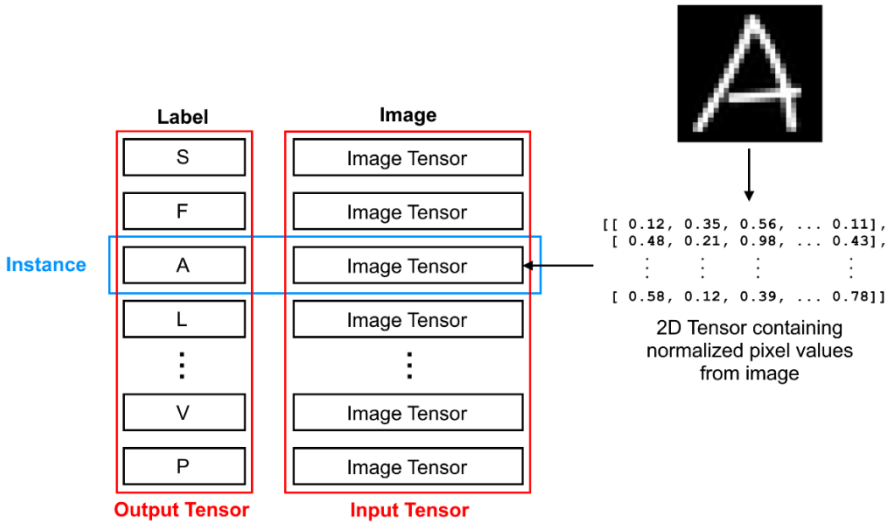

Fig. 2 ***Font dataset structure***

2.3 Training, Validation and Testing Data

In the process of creating supervised learning models, it is common practice to first partition the dataset into three sets: training set; validation set; and testing set.

Training set include the data samples that are used to fit (or train) the model[12]. This is usually the largest set.

[11] Ibid.
[12] (Brownlee, What is the Difference Between Test and Validation Datasets?)

Validation set is the data sample that is used to provide an unbiased evaluation of the model[13]**.** During the training process it shows how well the model is able to classify new data that the model has not seen before. The validation data plays no part in the training of the dataset, rather it offers a measure of how well the model is performing.

Testing set is used in the final step of developing a classification model. The training set provides an unbiased evaluation of the final model[14]. While the validation set may play a role in the development of the model, the testing set does not. The testing set is only used once the final model has been chosen and is measure of how well the model performs on data that it has never seen before.

2.4 Types of Machine Learning Data

Data in the field of machine learning comes in four primary types: numerical data; categorical data; time series data; and text data[15].

Numerical data (or quantitative data) consists of measurable data.[16] Some examples would be the height of buildings in New York or the amount of snowfall in Nebraska. Numerical data can either be discrete or continuous. Continuous data can take any value within a range (Example: height of a person) while discrete data has distinct values (Example: number of hedgehogs in a region)[17].

Categorical data is data that represents certain characteristics[18], this could include gender, social class, country, to name a few. These classifications do not have any mathematical meaning.

[13] Ibid.
[14] Ibid.
[15] (Algorithma)
[16] Ibid.
[17] (Zhang)
[18] Ibid.

Time series data refers to data that has a timestamp attached to it. These datapoints are often collected at regular intervals and allows models to analyze certain trends in time[19]. One example would be the price of gold in the last 10 years.

Text data simply consists of words, sentences, or paragraphs. Most of the time these words are turned into numerical data that a model can work with such as word frequency or average word length[20].

The dataset that this paper will discuss will consist of categorical data, images of handwritten glyphs will be assigned labels on what they represent (a letter from *a* to *z* or a number from 0 to 9). None of these labels have any mathematical meaning, thus the data is categorical.

2.5 Metrics for Data Quality

Not all datasets are created equal, they come in different qualities. When working with data one differentiates between clean and messy data. Messy data refers to data that is not standardized or data that contains errors such as invalid fields, missing values, additional values etc.[21]. Data frequently needs to be normalized, standardized, and cleaned before it can be used for machine learning[22]. There are six main dimensions that determine the quality of a dataset: accuracy; completeness; uniqueness; timeliness; validity; and consistency[23].

Accuracy – The degree to which data correctly describes the "real world" object or event being described[24]. This refers to the “correctness” of the data. For example, if an image is labeled as an apple it would be accurate if the image actually represented an apple.

[19] Ibid.
[20] (Algorithma)
[21] (Jones)
[22] (A.I. Wiki)
[23] (Farnworth)
[24] (CDC)

Completeness – The proportion of stored data against the potential of "100% complete"[25]. To evaluate how complete a dataset is one would check if all required information is available, for example if a dataset that stores addresses from customers is missing street names or countries, it would not be entirely complete.

Uniqueness – Nothing will be recorded more than once based upon how that thing is identified[26]. This is rather self-explanatory; the dataset should not include duplicate entries.

Timeliness – The degree to which data represent reality from the required point in time[27]. This means that data is accessible when it is expected and needed. Timeliness can be measured as the time between when information is expected and when it is readily available for use[28]. For example, if a yearly report is given five months too late it would not be very timely.

Validity – Data are valid if it conforms to the syntax (format, type, range) of its definition[29]. An example would be if all date formats were the same (DD/MM/YYYY). If a multiple different data formats are used such as (MM/DD/YYYY) or if the date is written out (2nd July 2013) them the data would be invalid.

Consistency – Consistency refers to data values in one data set being consistent with values in another data set[30]. One example if one collects weather data in a country, you can compare your data with the weather data from other sources to see if they correspond. If they do the data is consistent.

[25] Ibid.
[26] Ibid.
[27] Ibid.
[28] (Loshin)
[29] (CDC)
[30] (Loshin)

3. Experimental Methodology

3.1 Obtaining fonts

To begin with a large quantity of fonts was required to create the dataset. These were acquired from the font archive "DaFont" using a web scraper. Fonts under the "script section" were used as this was one of the larger subcategories (thus offering a lot training data) and because these fonts most closely resembled handwritten glyphs. After downloading these I ended up with 1.680 font files all with varying filetypes (otf, ttf, woff, etc…).

3.2 Dataset Pre-processing

For these fonts to be useful for machine learning purposes they must first be converted into image files. To do so I used the open-source font editor "FontForge", as the software had a python interpreter. This allowed me to quicky process the large amount of font files that I had. Using the interpreter, I converted the 1.680 font files into a total of 96.660 separate image files (png). These images would now have to be further formatted due to different image resolutions and glyphs often not properly filling the entire image.

The image formatting process used is quite similar to how the EMNIST dataset converted their glyph images[31]. The original greyscale images have varied resolutions (a). First the image colors are inverted (b), then the region of interest (ROI) is extracted, this is the region around the actual glyph (c). Then the ROI is rescaled to a maximum of 24 × 24 pixels while preserving

[31] (Cohen, Afshar and Tapson)

the aspect ratio (d). Finally, the ROI is centered and padded to a 28 × 28 pixel greyscale image (e).

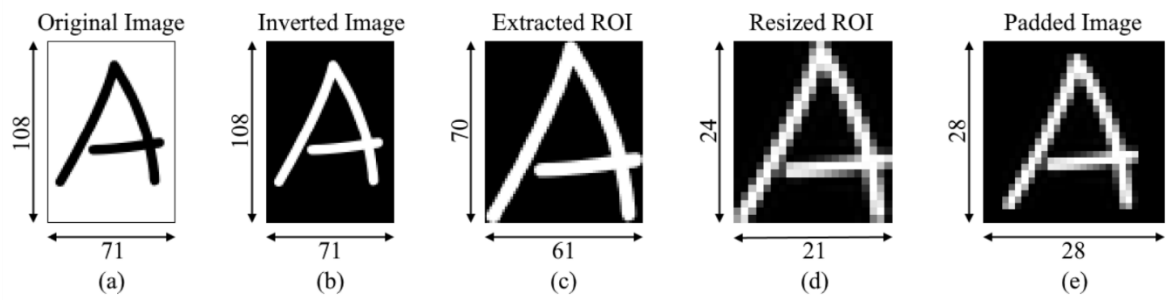

Fig. 3. ***Diagram of formatting process.*** *Example of capital 'A' from the font "Wagnasty".*

Now duplicate glyphs are filtered out, this often occurs between uppercase and lowercase characters as some fonts will use the same symbols for both. Additionally, fonts that are using placeholder symbols as substitutes for certain characters are also removed (this is done is many demo versions).

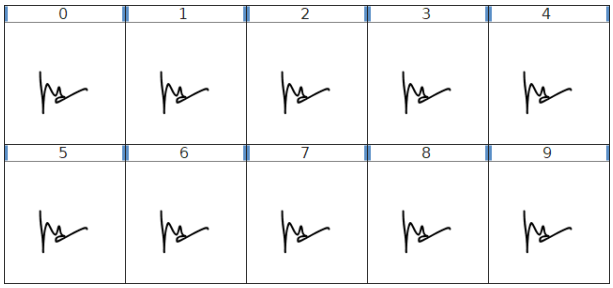

Fig. 4. ***Diagram of placeholder glyphs.*** *Example of placeholder glyphs for numbers in the font "jack no fruit".*

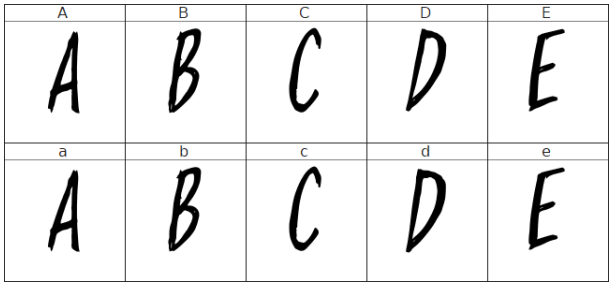

Fig. 5. ***Diagram of duplicate glyphs.*** *Example of both uppercase and lowercase characters representing the same glyphs in the font "Travelling"*

To remove duplicates one element from the uppercase and lowercase groups of the same font are compared, if they are the same the entire uppercase group for is removed (it does not matter if the uppercase or lowercase group is removed as they are the same). The placeholder glyphs are identified by comparing the first two elements for each character group (numbers, lowercase, uppercase). If the elements from a group are same, then the respective group is removed. This ended up removing 2.126 elements.

Furthermore, all fonts had wildly varying weights ("thickness" of a font) for their glyphs. I intended to filter out images that were either too thin or too thick so the final dataset would only consist of similarly weighted glyphs.

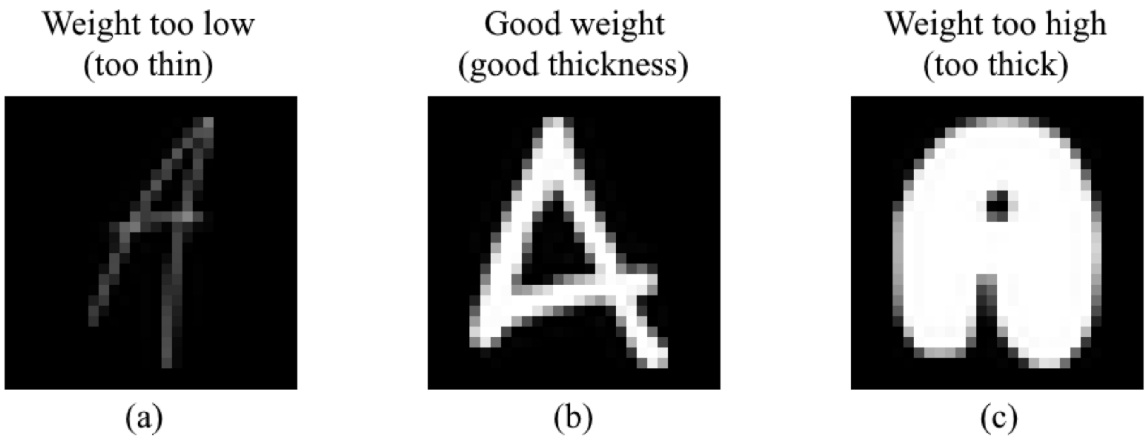

Fig. 6. ***Different glyph weights.*** *Examples of capital 'A' from the fonts "Sugeng Rawuh" (a), "TENA" (b), and "Bubble Writing" (c).*

To evaluate the weights of each glyph I calculated the sum of all the greyscale pixel values in the image.

$$W = \sum_{i=1}^{n} \sum_{j=1}^{m} p(i,j)$$

Eq. 1. ***Mathematical expression for the sum of image values to evaluate the weight of the glyph.*** *Elements n and m represent the number of rows and columns in the image and p(i,j) represents the greyscale value of the pixel at row i and column j. W is the weight of the image.*

The image sums are calculated for each glyph in the dataset and using these the distribution of image weights for each different glyph were determined. Using a box plot we can depict the distribution of weights and then determine upper and lower limits. Any glyphs with weights outside these limits are then removed from the dataset as these are deemed either too thin or thick. The following calculation was used to determine the upper and lower weight limits.

$$L_u = Q3 + IQR \times k$$

Eq. 2. ***Mathematical calculation of the upper limit.*** *Q3 represents the third quartile and IQR the interquartile range (Q3-Q1).*

$$L_l = Q1 - IQR \times k$$

Eq. 3. ***Mathematical calculation of the lower limit.*** *Q1 represents the first quartile.*

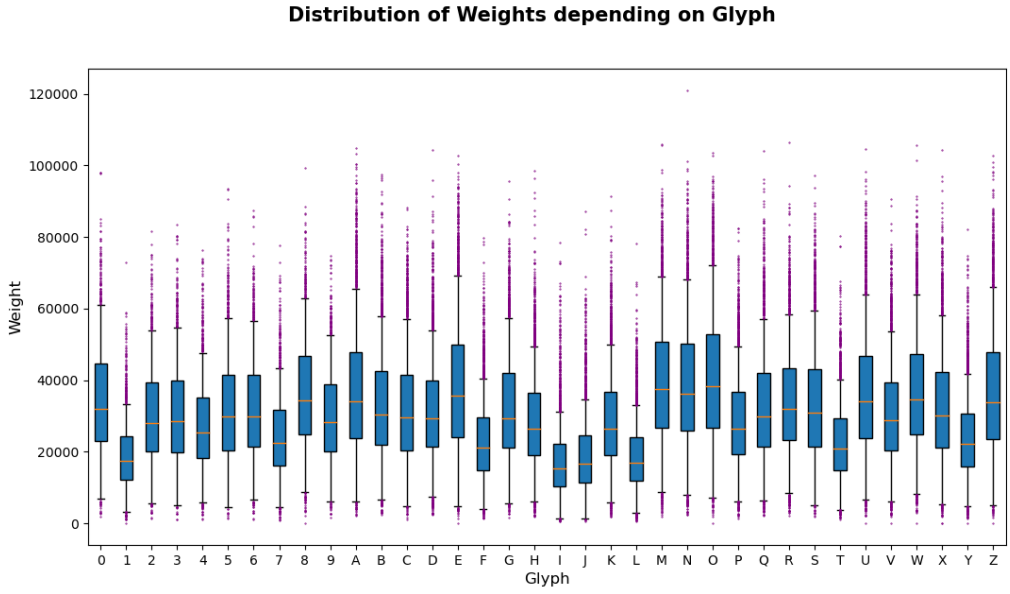

Fig. 7. ***Box plot depicting the distribution of weights for each glyph.*** *Capital and lowercase letters were grouped together as many fonts would use these interchangeably, therefore trying to keep these separate proved near impossible. Here the value k=0.75 is used in the limit calculations. The whiskers depict the upper and lower limits, the purple dots the values outside these limits and the blue boxes the IQR.*

This resulted in 8.313 elements being removed, thus the final font dataset consists of 85.222 samples. The final dataset contains of 71.708 different character samples and 13.708 different number samples.

3.3 Evaluation Model

To evaluate the quality and applicability of dataset with regarding machine learning, with the task of character and digit recognition, we must test it using a classification model. As character and digit recognition lies within the area of computer vison, we will be using a convolutional neural network (CNN), widely used in this field[32].

Two evaluation CNN's will be used to evaluate the font dataset with their architectures only differing at their output layers. The first CNN is for the recognition of handwritten numbers, while the second CNN is for the recognition of the handwritten alphabet (both uppercase and lowercase characters are grouped into the same classes). The chosen evaluation model consists of 2 convolutional layers separated by 2 max pooling layers. The final pooling layer (p2) is flattened into vector with the size of 6400, this is then inputted into two fully connected dense layers followed by the output layer. The output layer of the first CNN has the dimension of 10 (digits 0 to 9) while the second CNN had the dimension of 26 (letters *a* to *z*).

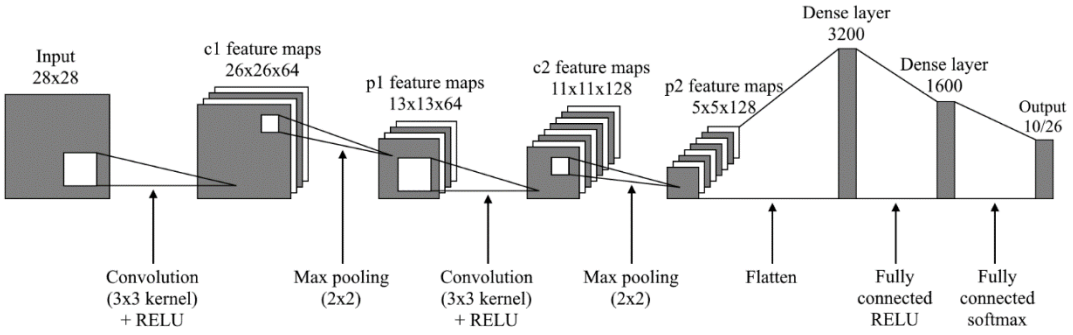

Fig. 8. ***Structure of evaluation convolutional neural network.*** *(c1) + (c2) are convolutional layers and (p1) + (p2) are pooling layers. Output has dimension of 10 when evaluating numbers and 26 when evaluating letters. Total of 5.212.106 trainable parameters.*

[32] (Zhou, Greenspan and Shen)

3.4 Comparison Dataset

To evaluate how well the font dataset performs at training neural networks to recognizing handwritten characters and digits it will be compared with another existing dataset. The one we will use is the previously mentioned EMNIST dataset, a "variant of the full NIST dataset", "which contains digits, uppercase and lowercase handwritten letters"[33]. Additionally, the font dataset used a similar formatting technique as used for the EMNIST dataset, thus making it easier to compare them both. The font dataset will be split into two sets, one for training and one for testing. The CNN will be trained using the training set and then validated with the training set and the EMNIST dataset. By validating the model with EMNIST, we can see how well a model, trained with the font dataset, performs at recognizing real handwritten characters and digits that the model has never seen before. This allows us to evaluate how accurately the font dataset represents real handwritten glyphs.

[33] (Cohen, Afshar and Tapson)

4. Experimental Results

4.1 Performance Accuracies

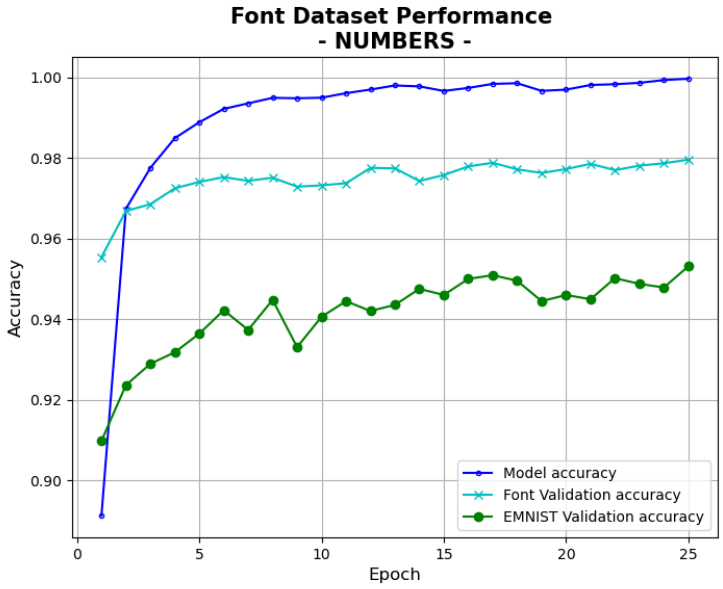

Fig. 9 ***Font dataset performance on digits.*** *Accuracies averaged over 10 trails using CNN described in 3.3 (batch size of 64 over 25 epochs).*

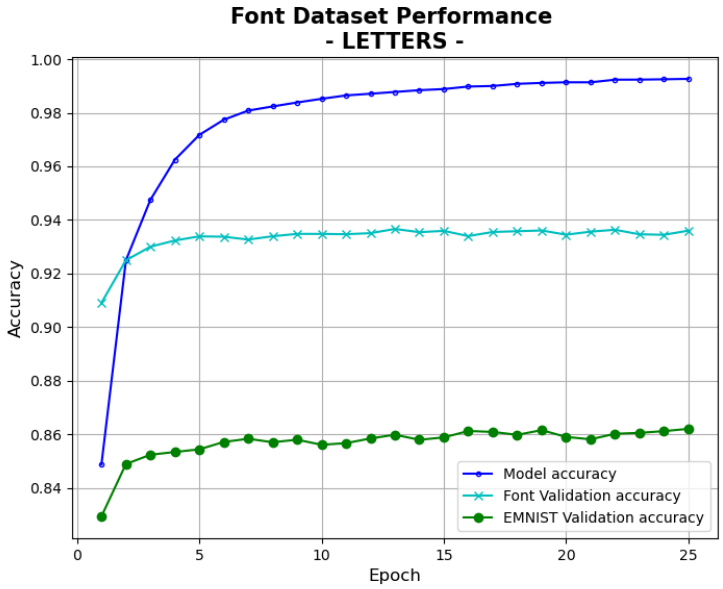

Fig. 10 ***Font dataset performance on characters.*** *Accuracies averaged over 10 trails using CNN described in 3.3 (batch size of 64 over 25 epochs).*

4.2 Confusion Matrix

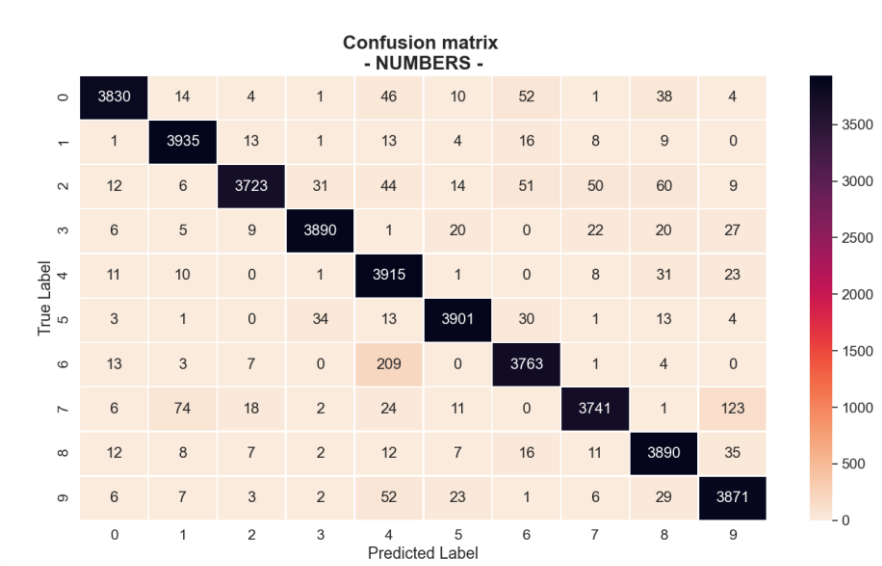

Fig. 11 ***Confusion matrix on digits.*** *Confusion matrix for single trail using CNN described in 3.3 (batch size of 64 over 25 epochs).*

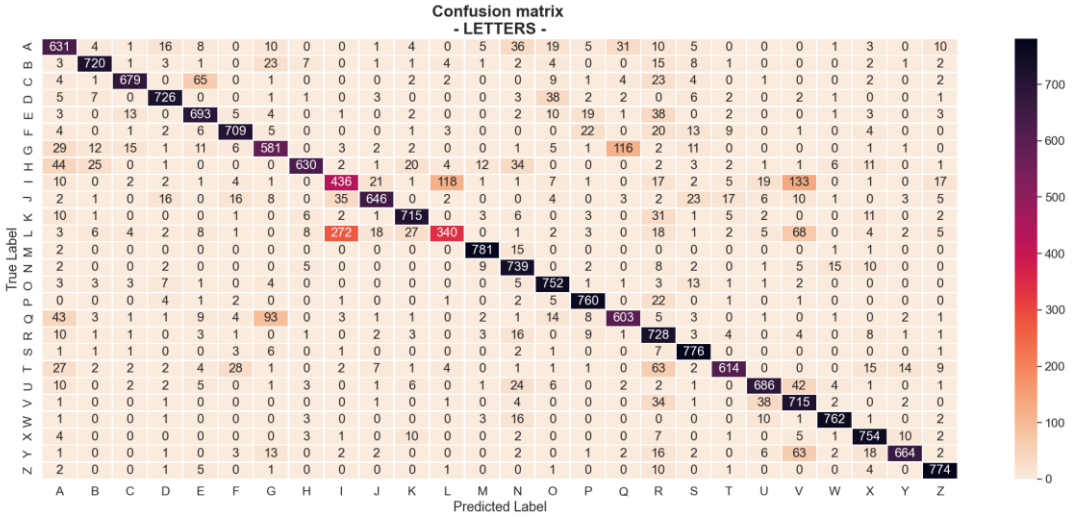

Fig. 12 ***Confusion matrix on characters.*** *Confusion matrix for single trail using CNN described in 3.3 (batch size of 64 over 25 epochs). The matrix shows noteworthy confusions between classes L and I, and classes Q and G. Instances of I are also occasionally confused with the class V.*

5. Analysis and Evaluation

5.1 Evaluation of Data Quality Metrics

Accuracy

To evaluate the accuracy of datasets it is "common to use 3rd party reference data from sources which are deemed trustworthy and of the same chronology"[34]. In this case the EMNIST dataset was used as it is a trustworthy dataset of real handwritten glyphs. The CNN's that were trained with the font dataset and validated with the EMNSIT dataset were able to achieve a validation accuracy of ~95% when classifying numbers and ~86% when classifying letters. This shows that the model was able to accurately recognize real handwritten glyphs from EMNIST even though it was trained using the font dataset. This means that the font dataset is accurate enough to train a model to classify "real world" handwritten glyphs.

Completeness

As the dataset only includes two columns (label and image) it was relatively easy to ensure completeness. All labels have a corresponding valid image and any images that were empty were also discarded. Additionally, all of the labels had a value that defined what their respective glyph represented. One can say the dataset is relatively complete.

Uniqueness

During the formatting process duplicate glyphs in each font group were removed. Any fonts with the same names were also removed before the formatting process. Therefore, most duplicate glyphs should have been filtered out.

[34] (CDC)

Timeliness

This dimension is not relevant for the font dataset as the data does not include any aspects regarding time. Thus, we may disregard this metric.

Validity

All instances are valid and each image has a respective label. As all image files went through the same formatting process (described in 3.2) they are all the same size (28 × 28 pixels). Additionally, none of the labels include any invalid symbols. Thus, all entries of the font dataset are valid.

Consistency

This dimension is also irrelevant to our dataset as there are no other datasets (that I am aware of) that collects images of font glyphs. Therefore, there is no other dataset that we can compare the font dataset with, so this metric can be disregarded.

5.2 Limitations and Possible Improvements

While font dataset, for the most part, fulfils the metrics that define a quality dataset, it is far from perfect. The dataset still has many limitations that must be addressed.

To begin with, a notable limitation of the font dataset is its comparatively small size. The dataset consists of a total of 85.222 elements, 71.708 character samples and 13.708 digit samples. Compared to the EMNIST dataset's 814.255 total entries, consisting of 411.302 character samples and 402.953 digit samples[35]. Thus, the EMNIST has around 10 times more total entries compared to the font dataset. This may be problematic as there is less information

[35] (Cohen, Afshar and Tapson)

to train a given model with, therefore the model will be less accurate and may have issues at correctly identifying new data.

One way of circumventing this problem would be to through the process of "Data Augmentation". Data augmentation allows you to increase the diversity of data for training without actually collecting new data[36]. This is done by shifting or distorting existing data to create new data that is still recognizable. For example, we could alter an existing image of the letter *a* by slanting it, changing its line thickness or by rotating it slightly. Therefore, it is possible to create multiple data entries from a single image.

Alternatively, one could just collect more data by finding more fonts, as there are many other sites where fonts can be downloaded. These sites can be web-scraped to gain access to further training data to be used in the dataset.

Another issue that the experimental results showed is that the trained models have difficulty distinguishing certain letters. This is can be seen in the confusion matrix (4.2), the model repeatedly confused *i*'s and *l*'s, and *g*'s and *q*'s.

Though this is likely not an issue regarding the dataset, rather a problem concerning the Latin alphabet. The previously mentioned letters show rather similar characteristics and can often be ambiguous, therefore they can be easily confused. Furthermore, these confusions were also evident in the EMNIST dataset, with the confusion matrix generated from their data showing similar results[37].

Furthermore, another limitation of our dataset is the grouping of capital and lowercase letters into a single class. Unlike the EMNIST dataset that has separate groups for both uppercase and lowercase letters. This means that any classification model trained using the font dataset would

[36] (Kausar)
[37] (Cohen, Afshar and Tapson)

be unable to differentiate between upper and lowercase characters. This grouping was done as many fonts will either use uppercase letters for lowercase letters or vice versa, often for stylistic reasons.

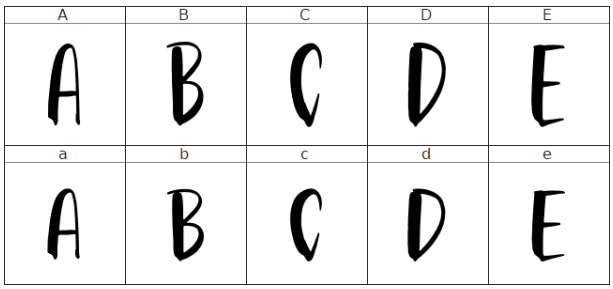

Fig. 13 ***Diagram of upper and lowercase glyphs.*** *Example of uppercase characters used for the lowercase characters in the font "Breadness".*

There was no practical way to get around this issue as one would have had to manually look through each font to determine in which of these this occurred. Therefore, the easiest solution to this problem was to group uppercase and lowercase characters together.

6. Conclusion

The investigation aimed at creating a dataset using computer fonts, that was able to accurately train machine learning models to recognize handwritten letters and numbers. After creating the dataset and evaluating its performance compared with that of the EMNIST dataset I am led to conclude that the font dataset was a relative success. Firstly, the dataset fulfills most aspects that determine a quality dataset. Furthermore, the dataset is able to achieve accuracies of ~98% when recognizing numbers (~95% validated with EMNIST) and ~93% when recognizing characters (~86% validated with EMNIST). The font dataset is a viable option to train models to classify handwritten glyphs.

7. Works Cited

A.I. Wiki. *Datasets and Machine Learning*. n.d. <https://wiki.pathmind.com/datasets-ml>.

Algorithma. *The importance of machine learning data*. 26 March 2020. <https://algorithmia.com/blog/the-importance-of-machine-learning-data>.

Brownlee, Jason. *What is the Difference Between Test and Validation Datasets?* 14 July 2017. <https://machinelearningmastery.com/difference-test-validation-datasets/>.

—. *Why Data Preparation Is So Important in Machine Learning*. 15 June 2020. <https://machinelearningmastery.com/data-preparation-is-important/>.

CDC. *The Six Dimensionsof EHDI Data Quality Assessment*. n.d. <https://www.cdc.gov/ncbddd/hearingloss/documents/dataqualityworksheet.pdf>.

Cohen, Gregory, et al. "EMNIST: an extension of MNIST to handwrittenletters." 17 Febuary 2017. <https://arxiv.org/abs/1702.05373>.

DeepAI. *Machine Learning*. n.d. <https://deepai.org/machine-learning-glossary-and-terms/machine-learning>.

Farnworth, Richard. *The Six Dimensions of Data Quality — and how to deal with them*. 28 June 2020. <https://towardsdatascience.com/the-six-dimensions-of-data-quality-and-how-to-deal-with-them-bdcf9a3dba71>.

Géron, Aurélien. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow,*. 2nd. O'Reilly Media, Inc, 2019.

IBM Cloud Education. *Machine Learning*. 15 July 2020. <https://www.ibm.com/cloud/learn/machine-learning>.

Jones, M. *Working with messy data*. 14 December 2017. <https://developer.ibm.com/technologies/analytics/tutorials/ba-cleanse-process-visualize-data-set-1/>.

Kausar, Aqsa. *Dealing with Small Datasets — Get More From Less — TensorFlow 2.0 — Part 1*. 27 April 2020. <https://medium.com/@aqsakausar30/dealing-with-small-datasets-get-more-from-less-tensorflow-2-0-10472e65a7f6>.

Kohavi, Ron and Foster Provost. "Glossary of Terms." *Machine Learning* (n.d.). <https://ai.stanford.edu/~ronnyk/glossary.html>.

Loshin, David. *Master Data Management*. Morgan Kaufmann, 2008.

Mitchell, Tom. *Machine Learning*. McGraw Hill, 1997. <http://www.cs.cmu.edu/~tom/mlbook.html>.

Zhang, Alina. *Data Types From A Machine Learning Perspective With Examples*. 16 August 2018. <https://towardsdatascience.com/data-types-from-a-machine-learning-perspective-with-examples-111ac679e8bc>.

Zhou, S. Kevin, Hayit Greenspan and Dinggang Shen. *Deep Learning for Medical Image Analysis*. Academic Press, 2017.

8. Apendix

8.1 Raw experimental data

Font Dataset Performance (Numbers)

	Epoch →	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
	Run ↓																									
Model accuracy	1	0.894304	0.968467	0.978731	0.984834	0.988826	0.991046	0.994524	0.994154	0.996596	0.995708	0.997114	0.994894	0.997188	0.999112	0.99963	0.999704	0.999186	0.993266	0.99519	0.998076	0.999186	0.999704	0.999852	0.999852	0.999778
	2	0.884779	0.968744	0.977807	0.984927	0.991416	0.992378	0.99297	0.99667	0.996596	0.993044	0.996004	0.997114	0.996966	0.996966	0.9963	0.998224	0.998964	0.999482	0.999704	0.999704	0.999704	0.999852	0.999778	0.999778	0.999778
	3	0.891714	0.96486	0.978084	0.985944	0.99001	0.991046	0.992896	0.99556	0.995116	0.994154	0.996152	0.993562	0.998446	0.998816	0.999408	0.99926	0.99926	0.99889	0.99408	0.993488	0.994746	0.998964	0.99963	0.999778	0.999852
	4	0.892639	0.963751	0.977807	0.986037	0.992156	0.992452	0.99445	0.996448	0.995042	0.993784	0.996596	0.996004	0.996226	0.995782	0.997262	0.998002	0.996448	0.998002	0.99593	0.99815	0.998742	0.999556	0.999852	0.999778	0.999778
	5	0.893841	0.967912	0.975217	0.98206	0.988678	0.993636	0.994672	0.995782	0.994376	0.994376	0.997558	0.998002	0.996078	0.995856	0.99704	0.995856	0.998372	0.99926	0.997484	0.998742	0.998594	0.995042	0.996448	0.996522	0.998002
	6	0.889865	0.967727	0.978176	0.986129	0.98779	0.9926	0.993784	0.98964	0.994672	0.997262	0.996374	0.99593	0.995042	0.99889	0.998446	0.999408	0.999778	0.999778	0.999852	0.999852	0.999778	0.999778	0.999778	0.999852	0.999778
	7	0.893194	0.968559	0.980211	0.984002	0.989936	0.992156	0.992156	0.993932	0.996374	0.993932	0.996226	0.997114	0.998002	0.999334	0.999704	0.999704	0.999852	0.999778	0.999778	0.999778	0.999852	0.999778	0.999852	0.999852	0.999852
	8	0.895598	0.96745	0.977344	0.985944	0.988456	0.9889	0.99334	0.996448	0.996374	0.993858	0.996078	0.999186	0.998964	0.999482	0.999778	0.999778	0.999778	0.999778	0.999778	0.999852	0.999778	0.999852	0.999852	0.999778	0.999778
	9	0.888016	0.967727	0.976882	0.984372	0.989936	0.991416	0.99334	0.99593	0.995634	0.994598	0.997262	0.998668	0.995856	0.99519	0.998816	0.998964	0.99926	0.99963	0.999704	0.99963	0.999778	0.999778	0.999778	0.999778	0.999778
	10	0.889125	0.969207	0.97568	0.985482	0.991786	0.992526	0.99408	0.994894	0.993784	0.995634	0.997484	0.997632	0.99667	0.99445	0.996596	0.99889	0.999482	0.998668	0.999556	0.999704	0.999482	0.99963	0.996374	0.990528	0.997336
Font Validation accuracy	1	0.956667	0.967037	0.972222	0.971111	0.972222	0.968519	0.973704	0.977037	0.976296	0.971111	0.971852	0.977778	0.976296	0.977037	0.975185	0.975185	0.975556	0.977037	0.978889	0.98037	0.981111	0.981111	0.98037	0.98037	0.98037
	2	0.942593	0.971111	0.964074	0.972963	0.968148	0.976667	0.977407	0.971852	0.973704	0.975556	0.971481	0.975926	0.976667	0.972593	0.979259	0.978519	0.978148	0.977037	0.972963	0.974074	0.974815	0.964815	0.973333	0.976296	0.973333
	3	0.948889	0.972593	0.968889	0.972593	0.974444	0.974074	0.974074	0.972593	0.97	0.973333	0.972593	0.977037	0.977778	0.974444	0.978519	0.975926	0.97963	0.976296	0.974815	0.979259	0.978148	0.98	0.98	0.98037	0.981481
	4	0.966296	0.960741	0.968889	0.975556	0.978148	0.973704	0.975556	0.973333	0.971852	0.97	0.976667	0.977407	0.978148	0.98037	0.978148	0.98	0.981852	0.981481	0.981481	0.982222	0.981111	0.980741	0.980741	0.981111	0.980741
	5	0.947778	0.967778	0.971111	0.971111	0.974815	0.976296	0.978148	0.974074	0.975185	0.977407	0.977778	0.97963	0.98	0.978889	0.980741	0.981481	0.97963	0.964444	0.967778	0.971852	0.978148	0.98037	0.98	0.98037	0.980741
	6	0.954444	0.969259	0.968148	0.974815	0.977778	0.977407	0.971481	0.972963	0.977778	0.975556	0.976667	0.979259	0.979259	0.962963	0.972963	0.972963	0.975926	0.979259	0.979259	0.98	0.98	0.97963	0.98	0.98037	0.98
	7	0.954074	0.962222	0.97	0.974074	0.969259	0.976296	0.972593	0.973704	0.97	0.974074	0.976296	0.975185	0.978148	0.977037	0.978148	0.980741	0.978519	0.979259	0.977407	0.972963	0.98037	0.982963	0.981111	0.98	0.982222
	8	0.958148	0.961852	0.966667	0.97	0.978889	0.976667	0.973704	0.977037	0.977037	0.973704	0.976667	0.981111	0.978148	0.97037	0.967037	0.975926	0.977037	0.977407	0.972593	0.977037	0.977037	0.975926	0.974815	0.974815	0.98
	9	0.965556	0.972222	0.968889	0.973704	0.973704	0.975185	0.977778	0.97963	0.97	0.975556	0.958889	0.973704	0.974815	0.977778	0.965556	0.977778	0.978889	0.977037	0.975185	0.972222	0.977407	0.975556	0.978519	0.979259	0.979259
	10	0.96	0.964074	0.967037	0.969259	0.973704	0.978148	0.969259	0.979259	0.967407	0.966296	0.978889	0.978889	0.975556	0.971852	0.982593	0.981111	0.983333	0.982963	0.982963	0.982963	0.977778	0.969259	0.972593	0.974444	0.978148
EMNIST Validation accuracy	1	0.91575	0.924	0.941325	0.933	0.93835	0.936525	0.946425	0.93995	0.948825	0.936875	0.922	0.94635	0.9382	0.954425	0.95385	0.953875	0.948825	0.9423	0.9261	0.949375	0.93495	0.955825	0.954525	0.955175	0.956475
	2	0.900475	0.931625	0.9318	0.935625	0.936325	0.932725	0.932525	0.9435	0.92175	0.940175	0.945725	0.9307	0.939125	0.94885	0.9479	0.93565	0.951275	0.95195	0.9486	0.952525	0.9521	0.951175	0.951525	0.951675	0.9517
	3	0.904775	0.92875	0.9392	0.9382	0.940375	0.951	0.91885	0.9527	0.932925	0.95255	0.95165	0.953575	0.953225	0.954925	0.953125	0.9559	0.9443	0.9587	0.9383	0.909775	0.916975	0.9374	0.957	0.956075	0.957575
	4	0.90655	0.932775	0.907975	0.9287	0.95105	0.948675	0.94875	0.94625	0.91745	0.942725	0.948675	0.946125	0.9471	0.937975	0.942475	0.9515	0.9533	0.942725	0.928625	0.9443	0.949575	0.950175	0.949875	0.95155	0.951725
	5	0.87855	0.924075	0.939175	0.940025	0.926825	0.938425	0.94415	0.939875	0.9467	0.939175	0.950375	0.9528	0.944425	0.948275	0.91755	0.9434	0.9546	0.9439	0.9354	0.9424	0.932325	0.9421	0.934375	0.9127	0.954325
	6	0.91665	0.932425	0.940475	0.91155	0.948375	0.9383	0.9336	0.9485	0.944125	0.9493	0.9385	0.9066	0.94095	0.943825	0.943175	0.947875	0.9487	0.949725	0.950475	0.949125	0.949275	0.949925	0.950475	0.949725	0.949775
	7	0.9227	0.893775	0.9169	0.93185	0.9326	0.94875	0.930075	0.925825	0.936525	0.93415	0.944025	0.9483	0.952425	0.955075	0.95225	0.956775	0.955125	0.9539	0.95535	0.9555	0.9556	0.954925	0.955775	0.954225	0.95585
	8	0.9166	0.8931	0.93385	0.9358	0.9338	0.936275	0.930875	0.944	0.942175	0.932075	0.9461	0.946375	0.94825	0.945375	0.947875	0.946925	0.949225	0.94955	0.9486	0.947775	0.948575	0.9488	0.9472	0.948	0.948625
	9	0.915875	0.936775	0.92515	0.9384	0.938275	0.95045	0.947275	0.955875	0.938025	0.9354	0.946075	0.945725	0.91885	0.944625	0.955575	0.95595	0.947325	0.957075	0.95995	0.9573	0.958075	0.95815	0.958675	0.959275	0.959325
	10	0.920625	0.939375	0.9138	0.925075	0.9185	0.941475	0.9412	0.951825	0.9015	0.943925	0.95195	0.9442	0.953275	0.942275	0.946725	0.9525	0.9569	0.94585	0.953725	0.952375	0.9525	0.953775	0.928525	0.94065	0.9462

Font Dataset Performance (Letters)

	Epoch →	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
	Run ↓																									
Model accuracy	1	0.860141	0.930454	0.950856	0.964718	0.973587	0.978901	0.980686	0.984214	0.985302	0.986431	0.986808	0.987923	0.987881	0.989136	0.989276	0.990712	0.99028	0.990475	0.990852	0.992372	0.993097	0.991005	0.993055	0.992944	0.99392
	2	0.862582	0.932225	0.951874	0.965834	0.974466	0.979193	0.980825	0.983405	0.984297	0.985246	0.986961	0.987449	0.988244	0.988258	0.989694	0.989011	0.990991	0.989987	0.99173	0.99134	0.991242	0.992999	0.99226	0.992623	0.991507
	3	0.860253	0.929492	0.951777	0.965067	0.973629	0.978761	0.98215	0.983586	0.985162	0.985302	0.986417	0.988356	0.987644	0.989429	0.989109	0.989722	0.990378	0.990127	0.990935	0.992232	0.992218	0.991981	0.991953	0.992707	0.993083
	4	0.863	0.930733	0.952265	0.965889	0.973671	0.978315	0.982387	0.983196	0.983042	0.985859	0.986571	0.987393	0.987226	0.989123	0.989513	0.989666	0.990168	0.990601	0.991256	0.991786	0.991424	0.992595	0.991912	0.992609	0.993334
	5	0.86201	0.929994	0.950117	0.964118	0.974173	0.977632	0.980839	0.983335	0.983726	0.985539	0.98579	0.986515	0.987017	0.989039	0.989248	0.988495	0.989917	0.989736	0.990587	0.991159	0.990977	0.992051	0.991689	0.993334	0.992971
	6	0.861159	0.930147	0.951414	0.965053	0.972918	0.979012	0.981996	0.983977	0.983656	0.985804	0.986682	0.986082	0.989081	0.988885	0.988467	0.99028	0.989429	0.991521	0.989959	0.991089	0.991828	0.991563	0.992246	0.992693	0.992637
	7	0.858189	0.930426	0.950814	0.964216	0.973183	0.97904	0.980741	0.984144	0.984283	0.984883	0.98611	0.988397	0.988007	0.989471	0.989234	0.989401	0.989639	0.991033	0.991675	0.991075	0.991689	0.992428	0.991898	0.993069	0.993557
	8	0.860253	0.931263	0.951289	0.964718	0.974285	0.978203	0.982122	0.983558	0.985106	0.985553	0.987059	0.986933	0.988676	0.988579	0.989401	0.990113	0.989792	0.989973	0.99141	0.99173	0.991033	0.992609	0.992149	0.992665	0.992497
	9	0.86208	0.931765	0.950898	0.964648	0.974048	0.978482	0.98162	0.982568	0.984325	0.985957	0.986682	0.987449	0.988411	0.989109	0.989276	0.990657	0.989457	0.99067	0.990629	0.991382	0.991521	0.992037	0.992009	0.992985	0.993041
	10	0.859862	0.930175	0.950773	0.965304	0.972765	0.978482	0.981704	0.982987	0.984088	0.985943	0.987282	0.987449	0.98784	0.988439	0.989903	0.989388	0.99141	0.990935	0.990684	0.991549	0.991159	0.9924	0.992665	0.993683	0.991953
Font Validation accuracy	1	0.910857	0.924286	0.929429	0.935429	0.935071	0.937571	0.932786	0.934786	0.936857	0.930214	0.932214	0.936429	0.939571	0.934571	0.937357	0.932429	0.931286	0.938929	0.9375	0.936929	0.938	0.934143	0.935643	0.937929	0.936857
	2	0.906357	0.922429	0.933143	0.934143	0.935286	0.932786	0.933071	0.930571	0.934143	0.934143	0.931143	0.935	0.937857	0.9315	0.933357	0.933857	0.9345	0.938286	0.938214	0.935714	0.938143	0.934357	0.932857	0.936143	0.935071
	3	0.912429	0.926357	0.9335	0.930643	0.936786	0.934429	0.936286	0.934143	0.932357	0.932643	0.935571	0.935429	0.937071	0.937214	0.935714	0.937	0.934929	0.935929	0.934286	0.936429	0.935857	0.937071	0.936357	0.936286	0.9375
	4	0.900929	0.929857	0.928357	0.932929	0.930929	0.931571	0.936214	0.932214	0.935214	0.933643	0.932571	0.935214	0.935214	0.937143	0.934143	0.934071	0.932643	0.933	0.935	0.930571	0.934714	0.94	0.9315	0.935071	0.934643
	5	0.907571	0.923429	0.934286	0.933	0.9335	0.936857	0.934643	0.934143	0.935643	0.934714	0.937071	0.934857	0.936143	0.933286	0.937143	0.935	0.937929	0.936429	0.936571	0.935714	0.936214	0.937286	0.936214	0.933929	0.933429
	6	0.914357	0.919786	0.929286	0.930857	0.933357	0.933071	0.931214	0.935	0.933286	0.935286	0.9335	0.934929	0.937786	0.937857	0.933214	0.936571	0.9345	0.932857	0.932929	0.9335	0.934071	0.935143	0.934	0.933571	0.934
	7	0.915643	0.925286	0.927643	0.933143	0.932	0.936143	0.928857	0.935571	0.933429	0.939071	0.938357	0.936	0.934357	0.935786	0.939071	0.932643	0.935857	0.935571	0.939714	0.935214	0.934571	0.933643	0.934	0.932714	0.9355
	8	0.906143	0.925143	0.928	0.930429	0.935714	0.9325	0.923357	0.932643	0.931857	0.933429	0.929357	0.9325	0.937429	0.938571	0.934714	0.933929	0.935143	0.936429	0.932571	0.933143	0.933571	0.934214	0.935214	0.933286	0.9375
	9	0.906	0.927071	0.926571	0.930714	0.932643	0.931	0.935	0.936286	0.938857	0.934786	0.936143	0.936071	0.935429	0.936	0.938786	0.934	0.936857	0.937214	0.938643	0.933786	0.937571	0.9365	0.938929	0.9335	0.938643
	10	0.911929	0.924857	0.929071	0.934786	0.934429	0.935143	0.935214	0.9345	0.938143	0.935214	0.938143	0.935357	0.938214	0.931357	0.936929	0.9285	0.937	0.936143	0.936429	0.936214	0.935714	0.938571	0.932571	0.935429	0.9375
EMNIST Validation accuracy	1	0.830577	0.836971	0.853798	0.845481	0.858606	0.853125	0.858606	0.858269	0.859135	0.854663	0.855577	0.859615	0.860144	0.84101	0.853125	0.858221	0.863894	0.861971	0.864519	0.863798	0.858798	0.858846	0.862933	0.866202	0.863365
	2	0.849183	0.864615	0.856154	0.852212	0.850529	0.863221	0.853654	0.865529	0.873846	0.865096	0.864135	0.862885	0.866827	0.862644	0.859135	0.869519	0.857452	0.850625	0.861154	0.86851	0.85976	0.864712	0.859183	0.863365	0.858125
	3	0.833846	0.841058	0.85375	0.855865	0.844231	0.863029	0.860673	0.86351	0.863606	0.86274	0.844375	0.851298	0.859087	0.860192	0.851538	0.861635	0.865673	0.849231	0.854231	0.856923	0.86399	0.84976	0.855288	0.855865	0.861154
	4	0.843221	0.847692	0.863029	0.848269	0.863702	0.857356	0.867115	0.860865	0.860192	0.861827	0.860577	0.853077	0.865721	0.864183	0.86149	0.864087	0.856154	0.868413	0.868462	0.864423	0.859567	0.862692	0.855913	0.866394	0.86601
	5	0.827885	0.850769	0.848173	0.86	0.850865	0.854471	0.851827	0.855096	0.850577	0.850337	0.855817	0.86399	0.855529	0.854471	0.86851	0.866683	0.858702	0.86	0.864712	0.85601	0.859904	0.862596	0.856202	0.85226	0.86851
	6	0.834087	0.850865	0.83625	0.855817	0.858942	0.861394	0.858942	0.857115	0.855913	0.857115	0.864519	0.856298	0.864135	0.862212	0.868269	0.856635	0.86149	0.866971	0.851298	0.854183	0.856683	0.865433	0.865048	0.862981	0.863702
	7	0.824231	0.856058	0.856202	0.858654	0.867067	0.868894	0.857212	0.851923	0.856635	0.85375	0.859712	0.852308	0.85976	0.858942	0.86024	0.861442	0.865769	0.859519	0.867115	0.856298	0.860288	0.859375	0.863317	0.85226	0.866635
	8	0.835048	0.848221	0.867452	0.840144	0.850913	0.846971	0.85	0.862788	0.860673	0.84726	0.851106	0.857452	0.849471	0.86101	0.863317	0.850288	0.861202	0.861875	0.860962	0.858606	0.862644	0.857163	0.867212	0.864183	0.861298
	9	0.822885	0.852885	0.84625	0.858269	0.839519	0.8525	0.863173	0.848413	0.847019	0.843702	0.84625	0.856298	0.853365	0.848702	0.849087	0.861442	0.856346	0.862885	0.859327	0.851202	0.842163	0.85476	0.855288	0.85976	0.856538
	10	0.792115	0.839327	0.842067	0.858702	0.85899	0.849904	0.861683	0.84625	0.851779	0.863942	0.864375	0.870769	0.863846	0.865721	0.853173	0.861923	0.861779	0.856058	0.863221	0.85976	0.857019	0.866154	0.864567	0.868029	0.854471

8.2 Python code

Code used for the creation of the font dataset the evaluation of said dataset, and for the visualization of the results.

Web Scraper

```
from bs4 import BeautifulSoup
from PIL import Image, ImageOps
import requests
import wget
import os

''' This is used to download fonts from dafonts.com'''

headers = {'User-Agent': 'Mozilla/5.0'}

downloadURLs = []
errors = []

def getDownloadURLs(url):
    '''Gets all download URL's from given page and adds the to list (downloadURLs).'''

    r = requests.get(url, headers=headers)
    soup = BeautifulSoup(r.text, features='html.parser')

    for a in soup.find_all('a', href = True):

        #* Add all download URL's to list (downloadURLs)
        if ("//dl.dafont.com/dl/?f=" in a['href']):

            print("Found download URL:", a['href'])
            downloadURLs.append(a['href'])


def getAllFonts(url, nPages):
    '''Takes dafonts URL and goes through n ammount of pages.
    All download URL's are added to list (downloadURLs).'''

    for i in range(nPages):

        print("Getting download files from page " + str(i+1) + '':')
        getDownloadURLs(url + '&page=' + str(i+1))


def downloadFonts(targetDir):
    '''Downloads fonts using download URL's in list (downloadURLs) to directory (targetDir).'''

    finished = os.listdir(targetDir)

    for i in downloadURLs:
        if i.split('=')[1] + '.zip' not in finished:
            print("Downloading: " + i.split('=')[1])
            try:
                wget.download('https://' + i, targetDir)
            except Exception as e:
                print("Error occured when downloading: " + i)
                print(str(e))
                errors.append(i)

            print("")

    print("Done!")
```

Font File to PNG converter

```
import fontforge
import os

#* ASCII HTML Numbers
#* Numbers   48-57
#* Uppercase 65-90
#* Lowercase 97-122

glyphName = ('0_num', '1_num', '2_num', '3_num', '4_num', '5_num', '6_num', '7_num', '8_num', '9_num',
             'a_upper', 'b_upper', 'c_upper', 'd_upper', 'e_upper', 'f_upper', 'g_upper', 'h_upper', 'i_upper', 'j_up
per', 'k_upper', 'l_upper', 'm_upper', 'n_upper', 'o_upper', 'p_upper', 'q_upper', 'r_upper', 's_upper', 't_upper', '
u_upper', 'v_upper', 'w_upper', 'x_upper', 'y_upper', 'z_upper',
             'a_lower', 'b_lower', 'c_lower', 'd_lower', 'e_lower', 'f_lower', 'g_lower', 'h_lower', 'i_lower', 'j_lo
wer', 'k_lower', 'l_lower', 'm_lower', 'n_lower', 'o_lower', 'p_lower', 'q_lower', 'r_lower', 's_lower', 't_lower', '
u_lower', 'v_lower', 'w_lower', 'x_lower', 'y_lower', 'z_lower')

#* Font character map; 48-57 = numbers [0-9], 65-90 = uppercase letters [A-Z], 97-122 = lowercase letters [a-z]
glyphNum  = (48, 49, 50, 51, 52, 53, 54, 55, 56, 57,
             65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90,
             97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 11
8, 119, 120, 121, 122)



def export_png_from_font(file, fontDir, targetDir):

    print(fontDir + '/' + file)

    try:
        ff = fontforge.open(fontDir + '/' + file)
    except:
        return

    fontName = file.replace(' ', '_').split('.')[0]

    for i in range(len(glyphName)):

        pngName = targetDir + '/' + glyphName[i] + '_' + fontName + '.png'
        try:
            ff[glyphNum[i]].export(pngName)
        except:
            pass


def export_all_fonts(fontDir, targetDir):

    dirFiles = os.listdir(fontDir)
    fontFiles = []

    for f in dirFiles:
        if f.lower().endswith(('.ttf', 'ttc', '.otf', 'fnt', 'bdf', 'fon', 'woff')):
            fontFiles.append(f)

    for f in fontFiles:
        print("===!===" + f + "===!===")
        export_png_from_font(f, fontDir, targetDir)
```

Image Formatting Process (resizing and padding image)

```
from PIL import Image, ImageOps
import numpy as np
import os
import shutil
import cv2
import random


def get_ROI(img):
    '''Takes a Pillow (PIL) Image object and gets the region of intrest (ROI) / bounding box
    Resizes Image (Max 24px height or width) -> 2px padding will be added in different method
    Returns resized ROI as PIL Image object'''

    imgArray = np.array(img)
    height, width = imgArray.shape

    # * Get bounding box
    yMin = None
    yMax = None
    xMin = None
    xMax = None

    for y in range(height):
        for x in range(width):

            if (imgArray[y][x] != 0 and yMin is None):
                yMin = y
            if (imgArray[height-y-1][x] != 0 and yMax is None):
                yMax = height-y

    for x in range(width):
        for y in range(height):

            if (imgArray[y][x] != 0 and xMin is None):
                xMin = x
            if (imgArray[y][width-x-1] != 0 and xMax is None):
                xMax = width-x

    roiArray = imgArray[yMin:yMax, xMin:xMax]
    roi_img = Image.fromarray(roiArray)

    return roi_img


def resize_image(img, max):
    '''Resizes image to determined (max) size.
    The resized image will either have a height or width of the given max.
    The aspect ratio of the image image is preserved'''

    width, height = img.size

    if (height > width):
        ratio = max/height

        newWidth = int(ratio*width+0.5)
        if newWidth == 0:
            newWidth = 1

        resized_img = img.resize( (newWidth, max), resample = 1)
    else:
        ratio = max/width

        newHeight = int(ratio*height+0.5)
        if newHeight == 0:
            newHeight = 1

        resized_img = img.resize( (max, newHeight),  resample = 1)

    return resized_img


def pad_image(img, max):
    '''Pad image to be square with given max heights and widths'''

    width, height = img.size

    wDelta = max - width
    hDelta = max - height
```

```
    padded_img = ImageOps.expand(img, border = (int(wDelta/2), int(hDelta/2), int(wDelta/2+0.5), int(hDelta/2+0.5)),
fill=0)

    return padded_img


def format_image(fp, targetDir):
    '''Get ROI, resize image, and pad to apropriate format.'''

    img = Image.open(fp)
    img = get_ROI(img)
    img = resize_image(img, 24)
    img = pad_image(img, 28)

    fileName = fp.split("/")[-1]

    img.save(targetDir + "/" + fileName)


def format_all(imageDir, targetDir):
    '''Takes all images in "imageDir" and formats them. Puts these into "targetDir"'''

    imgFiles = os.listdir(imageDir)
    finished = os.listdir(targetDir)

    for f in imgFiles:
        if f not in finished:
            print(str(f))
            format_image(imageDir + "/" + f, targetDir)


def generate_number_dataset(imgDir, targetDir):
    '''Generates the dataset for number glyphs.
    Also randomizes order of glyphs'''

    imgFiles = os.listdir(imgDir)
    sortedFiles = []

    # * Sort out glyph types to only incude numbers
    for f in imgFiles:
        if f.split('_')[1] == 'num':
            sortedFiles.append(f)

    # * Randomize glyphs
    random.shuffle(sortedFiles)

    image_data = []
    label_data = []

    pos = 0

    for f in sortedFiles:
        print(str(pos) + ':  ' + f)
        pos += 1

        # * Generate image data
        img = Image.open(imgDir + '/' + f)
        img_arr = np.array(img)
        image_data.append(img_arr)

        # * Get label of image
        label = int((f.split('_')[0]))
        label_data.append(label)

    image_data_np = np.array(image_data)
    label_data_np = np.array(label_data)

    # * Save dataset as numpy arrays
    np.save(targetDir + r"\numbers_images.npy", image_data_np)
    np.save(targetDir + r"\numbers_labels.npy", label_data_np)


def generate_letter_dataset(imgDir, targetDir):
    '''Generates the dataset for letter glyphs.
    Both capital and lowercase letters are classified with the same label
    Also randomizes order of glyphs'''

    imgFiles = os.listdir(imgDir)
    sortedFiles = []

    #* Sort out glyph types to only incude letters
    for f in imgFiles:
        if f.split('_')[1] == 'upper' or f.split('_')[1] == 'lower':
```

```
        sortedFiles.append(f)

#* Randomize glyphs
random.shuffle(sortedFiles)

image_data = []
label_data = []

pos = 0

for f in sortedFiles:
    print(str(pos) + ':  ' + f)
    pos += 1

    #* Generate image data
    img = Image.open(imgDir + '/' + f)
    img_arr = np.array(img)
    image_data.append(img_arr)

    #* Get label of image
    #* Labels: 0 = a, 1 = b etc
    label = ord(f.split('_')[0]) - 96
    label_data.append(label)

image_data_np = np.array(image_data)
label_data_np = np.array(label_data)

#* Save dataset as numpy arrays
np.save(targetDir + r"\letters_images.npy", image_data_np)
np.save(targetDir + r"\letters_labels.npy", label_data_np)
```

Evaluation Convolutional Neural Network

```
import os
os.environ["KERAS_BACKEND"] = "plaidml.keras.backend"

from keras import layers
from keras import models

import matplotlib.pyplot as plt

class EvalNetwork:

    def __init__(self, output_type):

        #* Network Model
        self.model = models.Sequential()

        #* Convolutional Layers
        self.model.add(layers.Conv2D(64,  (3, 3), activation='relu', input_shape=(28, 28, 1)))
        self.model.add(layers.MaxPooling2D((2, 2)))
        self.model.add(layers.Conv2D(128, (3, 3), activation='relu'))
        self.model.add(layers.MaxPooling2D((2, 2)))
        self.model.add(layers.Flatten())

        #* Dense Layers
        self.model.add(layers.Dense(1600, activation='relu'))

        #* Output Layer
        if output_type == "numbers":
            self.model.add(layers.Dense(10, activation='softmax', name="output"))
        if output_type == "letters":
            self.model.add(layers.Dense(26, activation='softmax', name="output"))

        print("Loaded new evaluation model for " + output_type + ".")
        print('_________________________________________________________________')

        #* Quick model summary
        #self.model.summary()

    def train_network(self, training_data, testing_data, n_epochs, n_batch_size, visualize_training=False):

        self.model.compile( optimizer=  'adam',
                            loss=       'categorical_crossentropy',
                            metrics=    ['accuracy'])

        training_hist = self.model.fit(training_data[0], training_data[1],
                                       epochs=n_epochs, batch_size=n_batch_size,
                                       validation_data=testing_data,
                                       verbose=2)

        #* If visualize_training is True, then show the results using matplotlib
        if visualize_training:
            x = list(range(1, n_epochs + 1))
            plt.plot(x, training_hist.history.get('acc'))
            plt.plot(x, training_hist.history.get('val_acc'))
            plt.show()

        return training_hist.history


    def train_network_no_validation_data(self, training_data, validation_split, n_epochs, n_batch_size, visualize_tra
ining=False)

        self.model.compile( optimizer=  'adam',
                            loss=       'categorical_crossentropy',
                            metrics=    ['accuracy'])

        training_hist = self.model.fit(training_data[0], training_data[1],
                                       epochs=n_epochs, batch_size=n_batch_size,
                                       validation_split=validation_split,
                                       verbose=2)

        #* If visualize_training is True, then show the results using matplotlib
        if visualize_training:
            x = list(range(1, n_epochs + 1))
            plt.plot(x, training_hist.history.get('acc'))
            plt.plot(x, training_hist.history.get('val_acc'))
            plt.show()
```

Code to Test the Effectiveness of the Datasets (includes visualization of results)

```
from evalutationNetwork import EvalNetwork

import os
import tensorflow as tf
import numpy as np

from keras.utils import to_categorical
from emnist import extract_training_samples, extract_test_samples

import matplotlib.pyplot as plt
import seaborn as sn
import pandas as pd

import csv


def format_data(data_images, data_labels):
    data_images = data_images.reshape((data_images.shape[0], 28, 28, 1))
    data_images = data_images.astype('float32') / 255
    if (np.amax(data_labels) == 26):
        data_labels = data_labels - 1
    data_labels = to_categorical(data_labels)
    return data_images, data_labels


def load_data(dataset):
    ''' Loads training and testing data samples from the given dataset.
    Valid datasets incude:
    - fonts_letters
    - fonts_numbers
    - emnist_letters
    - emnist_numbers
    - fonts_letters_no_val
    - fonts_numbers_no_val
    '''

    training_images = None
    training_labels = None
    testing_images = None
    testing_labels = None

    if (dataset == 'fonts_letters'):

        train_images = np.load("./font_dataset/_Format_v2/_optimizing_dataset/_datasets/letters_images.npy")
        train_labels = np.load("./font_dataset/_Format_v2/_optimizing_dataset/_datasets/letters_labels.npy")
        test_images, test_labels = extract_test_samples('letters')
        # test_images, test_labels = extract_training_samples('letters')

        training_images, training_labels = format_data(train_images, train_labels)
        testing_images, testing_labels = format_data(test_images, test_labels)

        print("Loaded fonts training letter dataset.")
        print("Loaded EMNIST testing letter dataset")

    if (dataset == 'fonts_numbers'):

        train_images = np.load("./font_dataset/_Format_v2/_optimizing_dataset/_datasets/numbers_images.npy")
        train_labels = np.load("./font_dataset/_Format_v2/_optimizing_dataset/_datasets/numbers_labels.npy")
        test_images, test_labels = extract_test_samples('digits')
        # test_images, test_labels = extract_training_samples('digits')

        training_images, training_labels = format_data(train_images, train_labels)
        testing_images, testing_labels = format_data(test_images, test_labels)

        print("Loaded fonts training number dataset.")
        print("Loaded EMNIST testing number dataset")

    if (dataset == 'emnist_letters'):

        train_images, train_labels = extract_training_samples('letters')
        test_images, test_labels = extract_test_samples('letters')

        training_images, training_labels = format_data(train_images, train_labels)
        testing_images, testing_labels = format_data(test_images, test_labels)

        print("Loaded EMNIST training letter dataset.")
        print("Loaded EMNIST testing letter dataset")

    if (dataset == 'emnist_numbers'):

        train_images, train_labels = extract_training_samples('digits')
        test_images, test_labels = extract_test_samples('digits')
```

```
        training_images, training_labels = format_data(train_images, train_labels)
        testing_images, testing_labels = format_data(test_images, test_labels)

        print("Loaded EMNIST training number dataset.")
        print("Loaded EMNIST testing number dataset")

    if (dataset == 'fonts_letters_no_val'):

        train_images = np.load("./font_dataset/_Format_v2/_optimizing_dataset/_datasets/letters_images.npy")
        train_labels = np.load("./font_dataset/_Format_v2/_optimizing_dataset/_datasets/letters_labels.npy")

        test_images = train_images[:14000]
        test_labels = train_labels[:14000]
        train_images = train_images[14000:]
        train_labels = train_labels[14000:]

        training_images, training_labels = format_data(train_images, train_labels)
        testing_images, testing_labels = format_data(test_images, test_labels)

        print("Loaded fonts training letter dataset.")
        #print("Loaded EMNIST testing letter dataset")

    if (dataset == 'fonts_numbers_no_val'):

        train_images = np.load("./font_dataset/_Format_v2/_optimizing_dataset/_datasets/numbers_images.npy")
        train_labels = np.load("./font_dataset/_Format_v2/_optimizing_dataset/_datasets/numbers_labels.npy")

        test_images = train_images[:2700]
        test_labels = train_labels[:2700]
        train_images = train_images[2700:]
        train_labels = train_labels[2700:]

        training_images, training_labels = format_data(train_images, train_labels)
        testing_images, testing_labels = format_data(test_images, test_labels)

        print("Loaded fonts training letter dataset.")
        #print("Loaded EMNIST testing letter dataset")

    print('_________________________________________________________________')

    return ((training_images, training_labels), (testing_images, testing_labels))


def evaluate_dataset(dataset, n_epochs, n_batch_size, n_runs, csv_file=' '):

    data = load_data(dataset)

    training_images = data[0][0]
    training_labels = data[0][1]
    testing_images = data[1][0]
    testing_labels = data[1][1]

    network = None

    evaluation_data = []

    for i in range(n_runs):

        print("Run Nr " + str(i+1) + "...")
        print('_________________________________________________________________')

        network = EvalNetwork(dataset.split('_')[1])
        evaluation_data.append(network.train_network(

                (training_images, training_labels),
                (testing_images, testing_labels),
                n_epochs, n_batch_size
        ))
        #* If data is already located in csv file, they are opened from here
        with open(csv_file, mode='a') as eval_data:
            data_writer = csv.writer(eval_data, delimiter=',', quotechar='"', quoting=csv.QUOTE_MINIMAL)
            data_writer.writerow(evaluation_data[-1].get('acc') + ["|"] + evaluation_data[-1].get('val_acc'))
        print('_________________________________________________________________')

    avr_acc     = [0] * n_epochs
    avr_val_acc = [0] * n_epochs

    #* Calculate average accuracys
    for i in range(n_runs):
        acc =     evaluation_data[i].get('acc')
        val_acc = evaluation_data[i].get('val_acc')

        for j in range(n_epochs):
```

```
            avr_acc[j]     += acc[j]     / n_runs
            avr_val_acc[j] += val_acc[j] / n_runs

    print(avr_acc)
    print(avr_val_acc)

    plt.figure(figsize=(8, 6))

    x = list(range(1, n_epochs + 1))
    plt.plot(x, avr_acc, '-b.', markerfacecolor='none', label="Font network accuracy")
    plt.plot(x, avr_val_acc, '-gx', label="EMNIST validation accuracy")

    #plt.xticks([0, 5, 10, 15, 20, 25, 30])
    plt.title("Font Dataset Performance validated with EMNIST Dataset \n- " + dataset.split('_')[1].upper() + " -
", fontsize=15, fontweight="bold")
    plt.xlabel("Epoch", fontsize=12)
    plt.ylabel("Accuracy", fontsize=12)

    plt.grid()
    plt.legend(loc='lower right')
    plt.show()


def generate_confusion_matrix(dataset, n_epochs, n_batch_size):

    #* Generate confusion matrix
    #* 1st of validation set
    #* 2nd of training set

    data = load_data(dataset)

    training_images = data[0][0]
    training_labels = data[0][1]
    validation_images = data[1][0]
    validation_labels = data[1][1]

    network = EvalNetwork(dataset.split('_')[1])



    network.train_network((training_images, training_labels),
                          (validation_images, validation_labels),
                          n_epochs, n_batch_size)

    val_predictions = network.model.predict(validation_images)
    train_predictions = network.model.predict(training_images)

    val_true_labels      = [i.index(max(i)) for i in validation_labels.tolist()]
    val_predicted_labels = [i.index(max(i)) for i in val_predictions.tolist()]

    train_true_labels      = [i.index(max(i)) for i in training_labels.tolist()]
    train_predicted_labels = [i.index(max(i)) for i in train_predictions.tolist()]

    if (dataset == 'fonts_letters' or dataset == 'emnist_letters'):
        val_confusion_matrix   = np.zeros((26,26), dtype=int).tolist()
        train_confusion_matrix = np.zeros((26,26), dtype=int).tolist()
    else:
        val_confusion_matrix   = np.zeros((10,10), dtype=int).tolist()
        train_confusion_matrix = np.zeros((10,10), dtype=int).tolist()

    for i in range(len(val_true_labels)):
        val_confusion_matrix[val_true_labels[i]][val_predicted_labels[i]] += 1

    for i in range(len(train_true_labels)):
        train_confusion_matrix[train_true_labels[i]][train_predicted_labels[i]] += 1

    val_df_cm   = pd.DataFrame(val_confusion_matrix)
    train_df_cm = pd.DataFrame(train_confusion_matrix)

    if (dataset == 'fonts_letters' or dataset == 'emnist_letters'):
        tick_label = [i for i in "ABCDEFGHIJKLMNOPQRSTUVWXYZ"]
    else:
        tick_label = [i for i in "0123456789"]

    plt.figure(figsize=(24, 16))
    sn.set(font_scale=1.4)
    sn.heatmap(val_df_cm, annot=True, annot_kws={"size": 16}, fmt='d', linewidths=1, cmap=sn.cm.rocket_r,
               xticklabels=tick_label,
               yticklabels=tick_label)
    plt.xlabel("Predicted Label")
    plt.ylabel("True Label")
    plt.title("Confusion matrix \n- " + "NUMBERS" + " -", fontsize=20, fontweight="bold")
    plt.show()

    plt.figure(figsize=(24, 16))
```

```
sn.set(font_scale=1.4)
sn.heatmap(train_df_cm, annot=True, annot_kws={"size": 16}, fmt='d', linewidths=1, cmap=sn.cm.rocket_r,
           xticklabels=tick_label,
           yticklabels=tick_label)
plt.xlabel("Predicted Label")
plt.ylabel("True Label")
plt.title("Confusion matrix \n- " + "NUMBERS" + " -", fontsize=20, fontweight="bold")
plt.show()
```