NUCL 575 Project: American sign language recognition

Mario Utiel
mutiel@purdue.edu
Department of Computer Science, Purdue
West Lafayette, Indiana, USA

Alejandro Mayo
amayogar@purdue.edu
Department of Computer Science, Purdue
West Lafayette, Indiana, USA

Lucía Ostolaza
lostolaz@purdue.edu
Department of Computer Science, Purdue
West Lafayette, Indiana, USA

Elena Gómez
gomez233@purdue.edu
Department of Computer Science, Purdue
West Lafayette, Indiana, USA

ABSTRACT

Neural networks are powerful tools that are being applied in different fields to make our life easier. This technology is also used to increase accessibility. One possible application of neural networks is sign language recognition. Our objective was to train neural networks to recognize images, videos, or even real-time captions/signs in which any letter of the American alphabet appears in sign language. This can be used to help people to learn sign language or to translate sign language into the American alphabet. [11]
Throughout the paper, we explain how we trained some neural networks to create a model that can recognize American sign language. We used a dataset to train mainly 2 neural networks: a fully connected neural network and a convolutional neural network. After showing how the neural networks were implemented using python, we will focus on the results obtained. As well as in a comparison of the performance of the two neural networks we trained. Finally, we will see additional research we did regarding how to interpret these results and how the decisions are made.

Figure 1: American sign language alphabet

NUCL Class, Purdue, IN, USA,

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

KEYWORDS

Neural networks, images recognition, overfitting

1 INTRODUCTION

Sign language is a natural language with the same properties as spoken languages. It is a visual language in which the brain processes linguistic information through the eyes. In this type of language, the most important part is the hands' shape and movements. It is not a universal language like the many languages spoken worldwide; each country has its sign language, and regions have dialects. There are somewhere between 138 and 300 different types of sign language: French Sign Language (LSF), Brazilian Sign Language (Libras), Chinese Sign Language (CSL or ZGS), etc. In English, there are three types: Australian Sign Language (Auslan), British Sign Language (BSL), and American Sign Language (ASL) . In this paper, we will focus on the recognition of American Sign Language, since it is the most common in the United States and many parts of Canada.
The objective of our project is to find the best model to recognize ASL. To achieve it, we have trained different neural networks in a dataset of a few thousands of images. These images were represented as matrices of size 28x28 with numbers representing each pixel. We transformed them into black-and-white images to try to neutralize the color of the hand and the back of the photo.
Before starting to train the neural networks, we are explaining the two types of neural networks that we implemented:

- Fully connected neural networks (FCNN)
- Convolutional neural networks (CNN)

2 NEURAL NETWORKS

To achieve our goal, we decided to implement two different types of neural networks. Then, in this section, we will see how these NN work and their main advantages.

2.1 Fully Connected Neural Networks

A fully connected neural network is a neural network in which each neuron applies a linear transformation to the input vector through a weights matrix. In an FCNN every input vector influences every input-output because all possible connections layer-to-layer are present. The activation function is nonlinear:

$$y_j k(x) = f(\sum_{i=1}^{n_h} w_j k x_i + w_j 0)$$

. This activation function takes the dot-product of two vectors, between the input vector x and the weights matrix W [7]. An example of a fully connected neural network can be seen in figure 2.

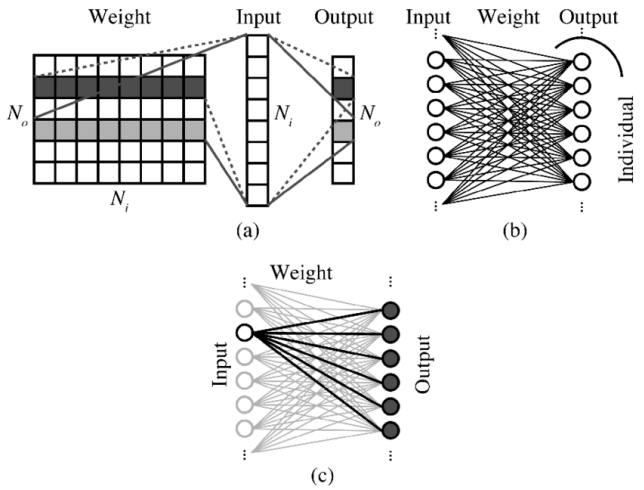

Figure 2: Fully connected neural network

One of the main advantages of this type of neural network is that no assumptions are needed to be made about the input. In addition, they are “structure agnostic". Therefore, their performance is weaker than the performance of more special neural networks.

2.2 Convolutional Neural Networks

In contrast, not all input nodes from a convolutional neural network are connected to the output. They assign importance to some characteristics of the images. Moreover, by grouping by areas, they keep some of the positional relationships between them. For example, on the next layer, the top left would be formed by the previous top left, and the three around them. For this reason, they have more flexibility in learning and they have a smaller number of weights per layer. Every layer transforms one volume of activation functions to another through a differentiable function.
Some of the different variables of the convolutions that can change the process transformations are:

- **Padding:** allows the filter to create extra layers made up of zeros to gain more insight into the edges. An interesting insight is that by fixing this, our output size could be equal to our input size. In our case, with the images more or less centered it is not extremely important. However, it should be more relevant when we introduce hands movements, and going in and outside of the camera focus.
- **Stride:** modifies the amount of movement. If it is equal to two in each movement of the filter, it will move also twice.
- **Kernel or Filter size:** the size of the "square" that will affect that output pixel. If the kernel is 3x3, each pixel on the output layer will be affected by 9 pixels of the input layer.

The formula for the final output size of a convolution comes by:

$$W_{output} = \frac{W_{input} - kernelSize + 2 * padding}{stride} \quad (1)$$

Which makes sense regarding the the attributes mentioned above. In figure 3 we can see an example of a CNN.

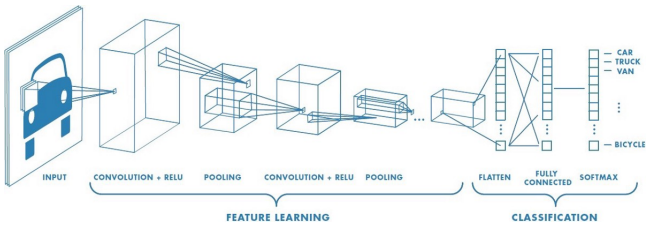

Figure 3: convolutional neural network

The main advantage of CNNs is that they can detect the most important features of images. They apply dimensionality reduction, obtaining the features minimized. This allows the network to minimize the future computational training of the fully connected layers that come after them.

3 TRANSFORMATIONS

Regarding the data, we used images of size 784 (i.e. 28x28 pixels) to train the NN. However, before feeding them, we applied some transformations. To begin with, we created batches of images for the NN to learn every batch instead of every image. We decided to make the batch size equal to 32. This means that we would feed 32 images at a time to the NN, to allow the NN to learn from each batch directly.
Moreover, regarding every single image, we applied various transformations:

- PIL Image
- Random Resized Crop
- Normalize
- To Tensor

Let’s review each of them.

3.1 PIL Image

This transformation simply converts a torch tensor (or a NumPy ndarray) with shape CxHxW (Channel x Height x Width) to a PIL image while preserving the value range. PIL is the Python Imaging Library that provides the programming language interpreter with image editing capabilities. It is used mainly for image processing. Therefore, it was very useful to manipulate images easily.

3.2 Random Resized Crop

One of the first things we noticed is how our first approaches had great results on training data. But when testing it, our models showed huge overfitting all the time. After some consideration, we decided to try and make our model more robust. But how can we make sure it happens? First, we realized the model just takes into account the very center of the image. Whenever we inputted some images where the hand was on one of the sides, or it was far from the camera, the performance dropped significantly.

For this purpose, we used a technique called data augmentation, where samples are perturbed during training to increase the model's robustness to these perturbations. That method crops a random area of the image and resized it to the given size. This transformation will take an input image and crop a random part of it before resizing that part to the initial size. We made this transformation to make the model take into account different parts of the image, not just the center or any of the corners, but all or most of the parts. This method solves the first problem.

3.3 Normalize

The second problem was that as the image has few pixels, each one of them had a bigger impact on the result. That means that the NN could just look at 5-6 pixels and made its decision based on that information, while the real result may be hidden in many more pixels that the NN does not take into account.

This time, we normalized the inputs so that the images values are rescaled to a [0,1] range. This will make information in each pixel be less important, as the range is smaller and several pixels may share similar values. Therefore, with these last 2 transformations, we try to make the models more robust to both perturbed data and images with different hand positioning.

3.4 To Tensor

Finally, we developed a less important transformation. The image is once again transformed to a tensor object. This will allow us to work with it easily.

4 IMPLEMENTATION

Let us follow with our implementation of the project. We have divided this part in a logical sequence:

- Environment
- Files
- NN layers

4.1 Environment

We have developed our project in Python, using libraries as NumPy, PyTorch or Lime. The latest will be deeply review lately. Moreover, in order to ease the availability of the files, we created a repository in GitHub. In this way, everyone could get the latest version of the files if any modifications were made by someone else. We believe it is important to note that we run every file locally, despite lacking efficient resources, as our implementation is efficient and does not need more than 5-7 minutes to completely train with 12 epochs. We will talk about that later in the training phase.

4.2 Files

We divided the code implementation in 3 main files plus an implementation of the algorithm. We believe that this implementation can be a practical case of this project: a live, real-time, webcam. Here, the model has been inserted to give live asses in sign language.

4.2.1 Data File: In this first file we implemented all data preprocessing and transformation. We loaded raw data from a CSV file where each row has a label and 784 pixels ranging from 0 to 255. Then, as already mentioned in the Transformation section, we applied some functions to the images to ensure the model will be more robust. As a summary, the random crop selected a portion of the original image and inputted it as the image to make the model robust to perturbation in data. The normalization will change the pixel values from [0,255] to [0,1].
Moreover, we created Data Loaders that will be used to feed the model with batches instead of single images. As explained before, this will make the model learn every batch that is fed into it, instead of every image. For the training data, the label is appended next to the image tensor.

4.2.2 Model Train File: The next file contains all the development and structure of our NN and the training functions. As mentioned before, we used the PyTorch module which provided us with some very useful utilities and methods to ease the implementation. The layers of each NN will be explained in more depth later on. The focus will be on the train and main functions of this file.
The main function has as inputs two parameters:

- **conv**: boolean. Determines which NN to train. 0 for the Fully Connected, 1 for the Convolutional.
- **n_epochs**: integer. Decides the number of epochs used to train the selected model.

The idea behind these parameters is to help us when running the code to train and test each model faster, and without the need to change the main code. When running the code, we printed the time spent by the selected model so that we can fairly compare them.
Now, the train function has as inputs the following parameters:

- **net**: model to train
- **criterion**: selected criterion in which the loss is calculated.
- **optimizer**: optimizer chosen to apply to the model.
- **trainloader**: Data Loader, which will be iterated to feed the model.
- **epoch**: The current epoch of the model.

The criterion used is the Cross-Entropy Loss, while the optimizer is the SGD (Stochastic Gradient Descent). In addition, as the training epochs we implemented were big enough, we decided to implement and scheduler for the learning rate. This simply varies the learning rate parameter in the optimizer after a certain amount of epochs. It is very useful because the more the model has learned, the less impact new training data has on the weight changes. We were already familiar with this kind of technique and we found it useful. At every epoch, the train function is performed on the model and the running loss is printed out and stored in a variable. This allows us to analyze the performance and convergence of the model later. The results obtained will be exposed after.

4.2.3 Evaluate File: In this file, we evaluated the models. We did it by evaluating the neural network in batches, just as the training. After the training, we saved and stored the model and the obtained weights. Then, we could load them later in the evaluation. Moreover, we used an extension module name onnx. It will help us to implement the algorithm in the practical approach that we took to test the performance of our model in real life. That is why, when running and looking and the performances, two results of what seems different models are shown, one for the actual network and

the other from the onnx module that takes the first model as reference. This file takes as input conv, the parameter that determined which NN to train.

4.2.4 Camera Implementation File: Lastly, we wanted to make our project practical and test its performance. After debating several options, we decided to try to insert somehow our model into a webcam. Then, it could do live predictions on what can be seen in the camera. This is where the onnx model comes in hand, as after doing some research, we found it was the easiest way to do it. We started by loading the model and open the webcam, using the onnx and the cv2 modules. Once the camera was active and sending images, we cropped it to get the center of the image and get better results. This cropped image was then reshaped to 28x28 pixel that can be feed to the model.
Lastly, when the live image is ready, it is inputted to the model and we get the predicted value. This is a number corresponding to the index of the letter in the alphabet. It is important to note that, as the model was trained with images, neither the 'J' nor the 'Z' are learned. In the American Sign Language both of them require movement. In future works, this could be an approach to take, using videos instead of images to include those letters. Continuing with the prediction, once it has been made, it is showed in the webcam in one of the corners. With this, we have been able to create a live camera which is able to represent the American Sign Language.
The results were not bad. However, there is still a lot to improve. The predictions are very sensible to the hand position in the image and also, what we have as background.

4.3 NN layers

In this last subsection of the implementation, we will talk about our NNs and the structure we have decided to use.

4.3.1 Convolutional: The Convolutional NN structure consists of 6 layers:

- 3 Convolution Layers
- 3 Linear (Fully Connected) Layers

Additionally, we implemented pooling and ReLu functions in the layers. While the pooling function was used just in the second and third layers (i.e. the second and third convolutions), the ReLu function was implemented in all the layers except for the last one. In this one, the raw output of the layer was indeed the actual output of the model. This is a typical approach that has been proved several times to be effective.

4.3.2 Fully Connected: The second model was a simpler one. It was our first approach due to its simplicity. Nonetheless, after we were not totally satisfied by the results of the model, we proceeded to experiment until we reached to the model shown in the previous section. To make a fair comparison of both models, we decided to use 6 layers too. In this case, all of them were linear layers, where every input neuron was connected to every output neuron. This huge amount of neurons and weights are what make this model to be time consuming, and inefficient. In addition, as in the other model, we used ReLu functions at every layer except the output one.

5 RESULTS

Now that we have covered the implementation and the structure of the models, we will see the results of evaluating the neural networks:

- Neural networks comparison
- Accuracies + losses
- Problems

5.1 Neural networks comparison

After training these two neural networks, we saw differences between them. The first one is that the fully connected has more overfitting than the convolutional. This occurs because in the fully connected neural network every neuron is connected to every feature of the images. When the model has an excessive number of parameters, it becomes prone to overfitting. As a result, these excessive parameters give the model the ability to memorize all the training example and then performs badly on evaluation examples. In contrast, the convolutional neural network does not require learning as many parameters in the fully connected one. Therefore, it has less overfitting. Another important difference is related to computational time. We can also see that the computational time of each one is different. Fully connected neural networks have a higher number of weights to train. This results in a high training time. On the other hand, CNN is trained to identify and extract the best features from the images for the problem, and hand with relatively fewer parameters to train. In the table 1, we can see the comparison between the FCNN and the CNN.

FCNN	CNN
+ overfitting	- overfitting
high training time	faster

Table 1: Neural networks comparison

5.2 Accuracies + losses

In table 2, we can see a comparison of the accuracy of the FCNN and the CNN in the train set and in the validation set. In Figures 4 and 5, we can see the outputs of the accuracies. As we discussed before, the FCNN has more overfitting than the convolutional one. It is proven with the accuracies. In the fully connected the training accuracy is 99.8% and in the convolutional 99.7%. The accuracy is slightly better in the fully connected. Nevertheless, when we test the NNs with the validation dataset, the accuracy of the fully connected is 93.3% while the accuracy of the convolutional is 96.8%. This means that the fully connected is too much trained. It learns the details and noise in the training data, and this impacts negatively the performance with new data. Then, it is less accurate when it has to predict data that is different from the one used to train the neural network. On the other hand, the performance of the convolutional is better than the performance of the fully connected when new data is predicted, because when it is trained it focuses on fewer features.

Figures 6 and 7 show the sum of errors made for each example in sets, which help us to understand how well the model is. Loss

	FCNN	CNN
Train accuracy	99.8	99.7
Validation accuracy	93.3	96.4

Table 2: Accuracies comparison

```
========== Fully Connected NN ==========
Training accuracy: 99.8
Validation accuracy: 93.3
========== ONNX ==========
Training accuracy: 99.8
Validation accuracy: 93.3
```

Figure 4: Accuracy FCNN

```
========== Convolutional NN ==========
Training accuracy: 99.7
Validation accuracy: 96.5
========== ONNX ==========
Training accuracy: 99.8
Validation accuracy: 96.4
```

Figure 5: Accuracy CNN

values imply how poorly or well the model behaves after each iteration of optimization. As we can see, in both plots, as the number of iterations increases the loss decrease. This means that the performance of the neural network improves in each iteration. The graph of the loss during the training of the convoluted neural network converges faster than the one for the fully connected.

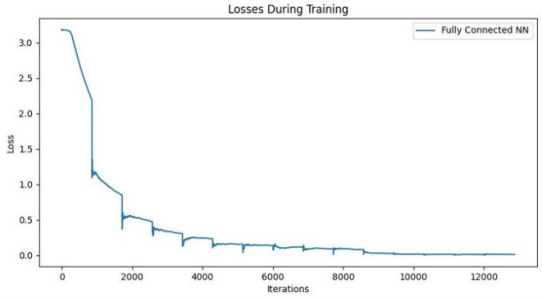

Figure 6: Loss CNN

5.3 Problems

We had to face up to some problems during the performing of our project. The first problem was that when we built the first neural net had too much overfitting. Its performance was really good in the training dataset, but not real-world and the validation dataset. In order to solve this problem we decide to add noise (white gaussian noise) to the training dataset. In this way, we get to reduce the

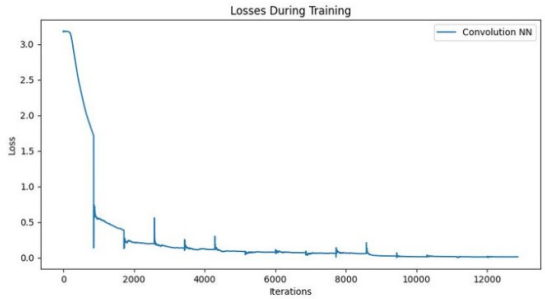

Figure 7: Loss FCNN

overfitting.

Another problem that we found is when we test the behavior of the neural networks using the camera. Instead of inserting images, we are gesturing the letters in front of the camera of laptop. When the background was not just a surface, the performance of the model was not good. So when we do this we have to do it in a clear background like a wall.

6 EXPLANATION MODELS

When performing a algorithm to make a prediction both effectiveness and accuracy are important. One of the main points but it is important that our "theory" understood by our method can be applied to real world problems.

Some simple systems such as linear regression provide also a partial explanation as it could be the own values of the weights. But these neural networks are black box algorithms. This means that the only output we receive from them is the output. And that the logic followed to obtain these come from complex mathematical operations.

This can bring several problems, as when applied in practice we need to make sure that a car detection algorithm does not confuse with other objects such as boxes. As well that the predictions make sense and not only they are correct in most cases. As the model accuracy can come from pure mathematical relationships that do not translate into logical causality.

To address these issues there have appeared several methods:

- **Local surrogate models:** The one we will focus on the most. It is based on calculating how much impact each attribute has on specific predictions also can calculate the total global importance of each variable. These can be applied in multiple fields, from classification the of tabular data to images and going through text pattern recognition. [2]
- **Rule modeling:** Can bring some rules that whenever they are accomplished the model predicts something specific, not as useful on images but could be seen with FOIL, FOLD... One example would be: if you have asthma you have bronchitis.[8]
- **State of ART MUSE:** Upon further research we found one type that was able to divide into two decision boundaries transforming the predictions done by a complex black box algorithm into decision trees. These method also allows to

specialize into specific attributes if we want to gain more insight of some variable. These would be the best one when explaining to a crowd with little to no technical knowledge. [5]

6.1 Local surrogate models

The aim on these type is to know why our algorithm made an specific prediction for an object or input. So these models work by creating specific interpretable, or white box outputs such as linear regressions for these predictions of their other parts the black box models.

We see then that these predictions are local and that the importance will depend on each of the subspace that we are taking into consideration for that input. [6]

6.1.1 *Optimizing with Local surrogate.* .

Some state of art papers even use these methods to increase optimization speed. Being pretty useful on some circumstances such as physics problems calculations. In terms of accuracy it has shown to be able to compete with gradient estimator or Bayesian optimization in speed convergence. But it even outperforms when the simulation uses a lower submainfold as these insight appears thanks to these local surrogates.[12][10]

6.2 LIME

LIME is a approach to provide interpretability. Which means that as the problems got more complex human comprehension fell behind so we need to be sure that when applying these algorithm the predictions make sense.

LIME works by creating small distortion on the data input we want to create a model interpretability. Then computing the predictions of these perturbed data. Afterwards it creates a linear model over a set of neighbours that are close to these value. It is important to consider that the weights that these new noisy values have ,when introduced into the linear model, are related to the distance with the predicting item [4]. To do these we need to set the kernel which express how we manage that distance from input to other samples. With some empirical evaluation we see that these decision change the output of our result as if too large it could not approximate a good local accuracy, that's why when using LIME library, which uses this type of method it is set to $0.75 * \sqrt{Nfeatures}$. This works as a good approximate as it is a point when most cases it stabilize and the jumps from the kernel size minimizes.

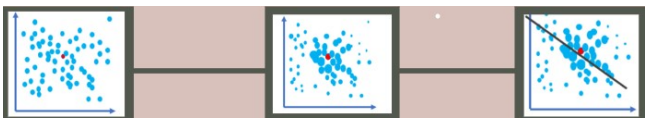

Figure 8: LIME process

We see kernel had a great impact on these approach as deciding what to take into consideration for our model. So these surrogate models although accurate they are not faithful.
This makes it not the most reliable as certain attributes comes from the weights on the linear model and the fact that with all the neighbours into consideration the total prediction would be pretty similar to the standard linear model if the accuracy of our model was high.[3]

6.3 Shap

This method came following the idea of performing a local model based on a linear regression over the specific local model. But this now tackles the issues of this previously described situation, as they tried to find other methods to accomplish better stability.

One of the solutions they thought was finding out the attribute importance independently on that specific prediction, this become the used Shapley values.

6.3.1 *Shapley values.* .

Shapley values are a concept from game theory. When we make a prediction some certain attributes come into play. For example on a regression each variable value brings a specific percentage of the final result. Shapley values try to gain this percentage that corresponds to the final result from each single variable. To do so it compares different predictions, if there is only one difference between them but the result prediction changes drastically we say then that the shapley value is high but if we swap several attributes and we see no change or a small change, we conclude that their percentage of contribution was small for that specific prediction. [9]

To calculate the Shapley values we should compare with all the possibilities, this coalition on categorical attributes comes by changing the value and on the numerical by changing between some feasible ranges for our problem. This brings us to a formula that will bring an integration for each attribute that we apply, so in this case, we have just evaluated the first two attributes:

$$S(x) = \iint f(x_1, x_2, X_3, X_4, X_5 ...) - E(f(X)) \tag{2}$$

Where S(x) is the value of the Shapley and E(f(X)) the expectancy which is the original value prediction.

We then see that this becomes a problem as we increase the number of attributes as the possibilities increase exponentially and in case we evaluate for all attributes except one we would have more than 700 integrals to compute. To do so practically when computing it we use an approximate by:

$$\phi = \frac{1}{M} \sum (f(X) - f(W)) \tag{3}$$

where W is the same X but it does not take into consideration an specific attribute or it only takes one into consideration. This way instead of trying all the combinations we look at the impact when completely taking out an attribute.

The way these shapley values are originally made bring up lots of strengths such as efficiency, where the feature contributions must add up to the difference of prediction for x and the average; symmetry; dummy, where a attribute with no impact has a shapley value of 0 and additivity. This attributes make sure that these predictions on the feature are fairly distributed. [1]

So when we calculate the shapley values for a general model we need to make the mean between the shapley values of all the different inputs that make it up to find the final shapley values.

This allow us to compare between the general model and specific predictions.

However they also come with some drawbacks as computing time for exact shapley value, the fact that we need to use all attributes at all times and the main drawback is that lacks of unrealistic results when the data is correlated which is the main flaw as we would need to make a previous analysis on the variables to subtract any possibility with a high correlation impact.

6.3.2 SHAP algorithm. .

Shap then uses these shapley values as its weight for performing these linear models. It grants a higher stability, as the simple weights are transformed into the shapley values and have certain attributes that the distance could not provide.

One of the main issues is that actually performing these values is more time consuming than the previous method. LIME method is exponentially faster to compute. In order to apply this method we used one of the python libraries. This library provides us a faster calculation of these shapley values, as they are optimized for some specific methods such as XgBoost. However, this is not the case in our neural networks, as we will also need to create several functions such as the predictions or transforming. As we do not use a predefined method, but our own neural network with PyTorch.

We applied these methods on our images. On the first case we use a method that is only performed with the data set itself.

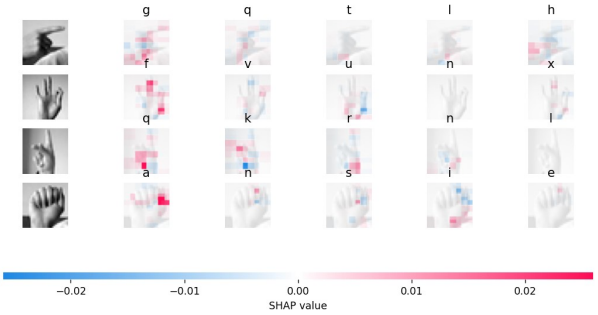

Figure 9: SHAP

On the second one we used perturbed data also, so this one holds as more robust but also takes a bit more of time although the difference on these specific case are very low as we already used noise into the images when transforming them.

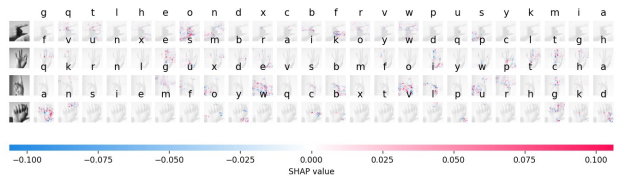

Figure 10: Perturbed SHAP

Something interesting we can bring from these images is that, because we are using convolutions networks for these methods, although the images are 28x28 we see that they tend to group together and not being individual pixels which could be a problem as a pixel itself does not provide enough information and if that was the case it would be mainly due to mathematical relationships and not logical.

This changes when we use fully connected and the final results tend to not be as useful.

Some specific interesting insights from this image are:

- On the letter A most of the percentage comes from the round shape given by the thumb
- On the F we see there is a high input given by the join of both index and thumb which is characteristic of that letter.
- Furthermore. the third image is miss labeled as Q when its K but looking at both letters we see they are quite similar although rotated, which given our transformations makes it feasible that the problem came from a specific rotation on either a Q or K.

Finally, we see how with further individual testing these methods could provide some insights into abnormalities. We could approach with specific training for example if we see that some specific label is not focusing on what is important. Also for real-world prediction we could change the type of training for that label or try some changes into the training data set to tackle this issues. These insights also would be more trustful than when using LIME due to the sturdiness of this specific method [3] .

7 CONCLUSION

Our main goal in this project was to create a model that could have a real application for deaf-mute people. Everybody else has the opportunity to generate automatic subtitles with plenty of applications, and we can use the voice assistant on our devices. Therefore, we went step by step through creating different neural network approaches.

After understanding how the neural networks worked, we developed the FCNN and the CNN in python. As the second one worked better, we kept working with that one. It was crucial to add some layers from the convolutional neural network because the important part of the images is a group of pixels instead of each pixel individually. They can express finger intersection, positions, shape, etc. We improved our model robustness by adding noise and applying some transformations.

We have faced overfitting problems, background problems, etc. At the end, when our model faces a situation with a different light, hand, or background, it struggles. Ideally, we would like to overcome the misclassification by taking into account more pixels on each image. In this way, we could have more defined gestures. Similar letters would be easier to differentiate. Moreover, we would need a more diverse dataset with some noise in the background, with hands of people with different color tones, and for example with gloves.

For future works, using short videos instead of images could help solve the problem of some letters not being learned as movement is needed to represent them. For example, the letters 'J' and 'Z' could be also taken into account.

REFERENCES

[1] Kjersti Aas, Martin Jullum, and Anders Løland. 2019. Explaining individual predictions when features are dependent: More accurate approximations to Shapley values. https://doi.org/10.48550/ARXIV.1903.10464

[2] Anonymous. 2020. The Reverse Turing Test for Evaluating Interpretability Methods on Unknown Tasks. In *NeurIPS 2020 Workshop on Human And Model in the Loop Evaluation and Training Strategies*. https://openreview.net/forum?id=y190Uu1z5Zk

[3] Liat Antwarg, Ronnie Mindlin Miller, Bracha Shapira, and Lior Rokach. 2019. Explaining Anomalies Detected by Autoencoders Using SHAP. https://doi.org/10.48550/ARXIV.1903.02407

[4] Damien Garreau and Ulrike von Luxburg. 2020. Explaining the Explainer: A First Theoretical Analysis of LIME. In *Proceedings of the Twenty Third International Conference on Artificial Intelligence and Statistics (Proceedings of Machine Learning Research, Vol. 108)*, Silvia Chiappa and Roberto Calandra (Eds.). PMLR, 1287–1296. https://proceedings.mlr.press/v108/garreau20a.html

[5] Himabindu Lakkaraju, Ece Kamar, Rich Caruana, and Jure Leskovec. 2019. Faithful and Customizable Explanations of Black Box Models. In *Proceedings of the 2019 AAAI/ACM Conference on AI, Ethics, and Society* (Honolulu, HI, USA) *(AIES '19)*. Association for Computing Machinery, New York, NY, USA, 131–138. https://doi.org/10.1145/3306618.3314229

[6] Caio Nóbrega and Leandro Marinho. 2019. Towards Explaining Recommendations through Local Surrogate Models. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing* (Limassol, Cyprus) *(SAC '19)*. Association for Computing Machinery, New York, NY, USA, 1671–1678. https://doi.org/10.1145/3297280.3297443

[7] Alexander G. Schwing and Raquel Urtasun. 2015. Fully Connected Deep Structured Networks. *CoRR* abs/1503.02351 (2015). arXiv:1503.02351 http://arxiv.org/abs/1503.02351

[8] Farhad Shakerin and Gopal Gupta. 2019. Induction of non-monotonic logic programs to explain boosted tree models using lime. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. 3052–3059.

[9] Lloyd S. Shapley. 1952. *A Value for N-Person Games*. RAND Corporation, Santa Monica, CA. https://doi.org/10.7249/P0295

[10] Sergey Shirobokov, Vladislav Belavin, Michael Kagan, Andrei Ustyuzhanin, and Atilim Gunes Baydin. 2020. Black-Box Optimization with Local Generative Surrogates. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 14650–14662. https://proceedings.neurips.cc/paper/2020/file/a878dbebc902328b41dbf02aa87abb58-Paper.pdf

[11] Yuancheng Ye, Yingli Tian, Matt Huenerfauth, and Jingya Liu. 2018. Recognizing American Sign Language Gestures From Within Continuous Videos. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*.

[12] Zongzhao Zhou, Yew Soon Ong, Prasanth B. Nair, Andy J. Keane, and Kai Yew Lum. 2007. Combining Global and Local Surrogate Models to Accelerate Evolutionary Optimization. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 37, 1 (2007), 66–76. https://doi.org/10.1109/TSMCC.2005.855506