"Semester" Project Report

Neural Networks (AI)

David van Wuijkhuijse (s5592968),
Marcus Harald Olof Persson (s5343798),
Richard Frank Harnisch (s5238366)
University of Groningen

November 4, 2025

Abstract

We implement a feed-forward convolutional neural network from the group up including loss, activations, optimizer, and LR scheduler. We train and evaluat the model on the German Traffic Sign Recognition Benchmark (GTSRB) dataset with five out of the 43 classes. Our model achieved a test accuracy of 20% after training on 5 classes only, equal to random guessing.

Contents

1 Introduction

With this project, we implemented the 1998 CNN introduced by Lecun et al. (1998) using our own architecture coded from scratch in Python. We then deploy this CNN on the German Traffic Sign Recognition Benchmark (GTSRB), consisting of 51,389 images in 43 classes. However, we only train on 5 of these classes due to resource constraints, as our home-made implementation is significantly slower than optimized libraries like TensorFlow or PyTorch. The dataset contains images of varying sizes and conditions, which we develop a modular suite of preprocessing transforms to handle.

Our main goal with this project is to create the core components of a CNN from scratch, only using lower-level libraries (e.g. Numpy, Pillow) in Python. The motivation behind this decision was to gain further insights into how neural networks work, by understanding and implementing the math behind the components of the CNN. We evaluated model performance using the accuracy metric and confusion matrices. We considered a model having a higher accuracy metric than 20% (random guessing baseline) to be successful. We were unable to achieve this goal due to the limited size of the original LeCun model and our computational constraints.

2 Data

The German Traffic Sign Recognition Benchmark (GTSRB) dataset consists of 51,839 images of traffic signs on German roads, categorized into 43 different classes. The images are taken from dashcam footage of cars driving on the road and thus past the signs. This means that each sign is captured multiple times as the car passes the sign. This lessens the need for data augmentation as each sign is already represented multiple times. Signs are captured in a range of different lighting and weather conditions, as well as from different angles and distances.

Figure 1: The same sign from different distances (and thus sizes) and slightly different angles.

The dataset is provided by Stallkamp et al. (2012) [2] [3] from the Institute for Neuroinformatics at the University of Bochum in Germany. By default, the dataset is split into a training set and a testing set. This was originally for a competition at the International Joint Conference on Neural Networks in 2011. Considering we are (considerably) past the deadline for submission, we have chosen to merge the training and testing sets at download time and then create our own split. This allows us to modify the split ratio as we like.

The images themselves are presented as PPM files with varying resolution and aspect ratio depending on the distance and angle of the sign to the camera. The aspect ratio is mostly close to 1:1. For more detailed analysis of the images sizes see Table 1 and Figure 3. There are three color channels (RGB) and the pixel values are in the range of 0-255. See Figure 4 for the distribution of pixel values. The labels

Figure 2: Three different signs of the same type exhibiting variation in lighting conditions and angles. For more examples of different types of variation, see Figure 9 in Appendix B.

Statistic	Width (px)	Height (px)	Aspect Ratio (W/H)
Mean	50.76	50.34	1.0056
Std	24.50	23.26	0.0734
Min	25	25	0.3681
25th Percentile	34	35	0.9706
50th Percentile (Median)	43	43	1.0000
75th Percentile	58	58	1.0385
Max	266	232	1.4375

Table 1: Descriptive statistics of image dimensions and aspect ratios in the GTSRB dataset.

for the images are provided in one master file `labels.csv` for the entire dataset. Each line contains the filename, class ID, and bounding box coordinates for the sign in the image.

We load the data as a numpy array of shape (H, W, C) for each image, where H is height, W is width, and C is the color channel (so three). The labels are loaded as integers corresponding to the class IDs. Before feeding the images to the model, we normalize the pixel values to the range [0, 1] by dividing by 255.

Data is downloaded and cleaned automatically using the `dataio/gtsrb_download.py` script. This script downloads the zipped dataset (three files: training dataset including labels, test dataset, test dataset labels), merges all images into one folder, and creates one master CSV file with all labels. It then deleted the zip files unless a parameter is set to keep them, as well as cleans the folder structure to remove unnecessary files (dataset text info files) and now empty folders. The data is freely hosted online by the University of Copenhagen Electronic Research Data Archive.

We implement a GTSRB dataset class in `dataio/gtsrb_dataset.py`. This class handles loading the images and labels from disk, applying any specified transforms, and providing access to individual samples. This is meant as a minimal implementation of the PyTorch Dataset class. The dataset provides access to a length attribute (number of samples) and a getitem method to retrieve individual samples by index (returns a tuple of image and label). The dataset class also supports caching of images in memory for faster access during training, as well as multiple worker threads for parallel loading of data. These performance optimizations were implemented later in the project after initial tests showed slow training.

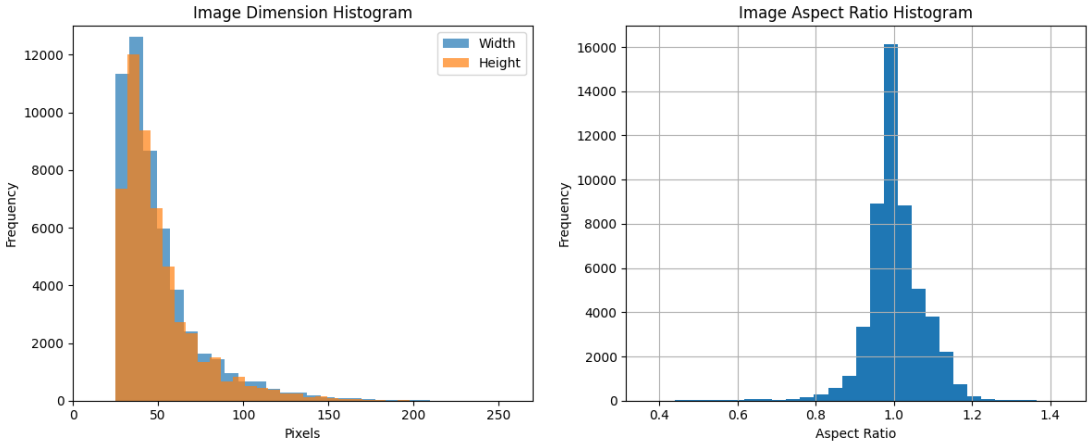

Figure 3: Distribution of image dimensions and aspect ratios in the GTSRB dataset. Most images are around 30-70 pixels in both dimensions, with a long tail towards larger sizes.

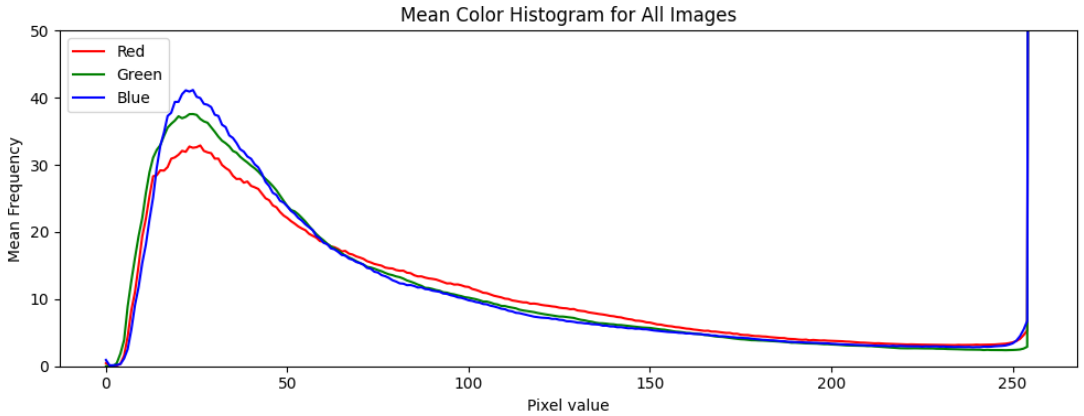

Figure 4: Histogram of pixel values across all images and color channels in the GTSRB dataset. The graph is cut off at frequency=50, but the frequency of flat white is very high due to the reflective nature of signs as well as the common use of the color white, causing (parts of) signs to max out the camera's sensor.

3 Methods and Experiments

3.1 Pipeline Overview

To begin, we preprocess the data. For this purpose, we have developed a suite of modular transform classes implemented in `transforms.py`. Each class implements a `__call__` method, allowing instances of these classes to be used as callable functions that apply specific transformations to input images. This design enables easy composition of multiple transformations into a single pipeline using the `ToCompose` class, which sequentially applies a list of transform instances to an image. On top of this, the following classes are implemented:

- `ToResize`: Resizes images to a specified square size using bicubic interpolation. This is so the images can be fed into the model which requires a fixed input size.
- `ToCenterCrop`: Crops a square out of the center of an image to a specified pixel size.
- `ToGrayscale`: Converts RGB images to grayscale using the ITU-R 601-2 luma transform using the formula
$$L = \frac{1}{1000}(299R + 587G + 114B), \tag{1}$$
where R, G, and B are the red, green, and blue pixel values, respectively. This is implemented by the PIL library. We then convert back to RGB by duplicating the grayscale channel three times, as the network expects three channels.
- `ToTensor`: Converts images from numpy arrays to PyTorch tensors and permutes the dimensions from (H, W, C) to (C, H, W). Also converts pixel values from integers (0-255) to floats (0.0-1.0).
- `ToNormalize`: Centers values around zero by subtracting the mean and sets the standard deviation to one by dividing by the standard deviation for each channel. Per-channel mean and standard deviation are computed on the training set and passed as a parameter to the transform.
- `ToRotate`: Applies random rotations within a specified angle range. Rotate function implemented by the PIL library.
- `ToRandomNoise`: Adds Gaussian noise to images with a specified mean and standard deviation. Normal distribution is sampled using numpy.

Data is loaded using the Dataloader class, implemented in `dataio/dataloader.py`. This class handles batching, shuffling, and parallel loading of data using multiple worker threads. When constructing a DataLoader, we specify the dataset to load from, batch size, whether to shuffle the data at the start of each epoch, whether to drop the last batch if it is imcomplete, the seed for shuffling, an optional custom function to collate samples into batches, how many batches to prefetch, and the number of worker threads for parallel loading. The DataLoader provides an iterator interface to loop over batches of data during training. The class also provides a default collating function that stacks images and labels into tensors. The DataLoader uses a thread pool executor to load batches in parallel, improving performance when using multiple workers. This was also implemented after slow initial training. During training we iterate through the dataloader to get new batches of images and labels.

For a diagram of the full data pipeline, see Figure 10 in Appendix B.

3.2 Model Description

The architecture of the CNN is modeled after the original 1998 CNN ([1]) designed for handwritten digit recognition. With layers:

Input Layer. The input layer takes 32×32 pixel images with 3 color channels (RGB). In the original model, characters were centered in a 28×28 field within a 32×32 input. This helps capture the essential features of the characters in a standard size. The original model also had only one color channel, but this was due to a limitation in the dataset used. We believe this was not a design choice as more channels are used later in the architecture.

Convolutional Layers (C1, C3, C5). These layers extract features from the input image using convolutional operations with different kernel sizes and numbers of feature maps. The first convolutional layer (C1) detects simple features like edges, while deeper layers (C3, C5) detect more complex features.

Mathematically, a convolution is an operation that combines two functions to produce a third function expressing how the shape of one is modified by the other. For us, this means sliding a small matrix called a kernel or filter over the input image and computing a weighted sum at each position. The result is a feature map that highlights certain patterns in the input, such as edges, textures, or more complex structures.

Formally, for a 2D input image I and a kernel K of size $(m \times n)$, the convolution operation at position (i, j) is defined as:

$$S(i,j) = (I * K)(i,j) = \sum_{u=0}^{m-1} \sum_{v=0}^{n-1} I(i+u, j+v) \cdot K(u,v) \tag{2}$$

where $S(i, j)$ is the output feature map at position (i, j), $I(i + u, j + v)$ is the pixel value from the input image, and $K(u, v)$ is the kernel value at position (u, v). In practice, the kernel is moved across the image, and at each location, the sum of the element-wise products is computed. For multiple input channels (RGB), the convolution is performed separately for each channel, and the results are summed to produce a single output feature map.

Convolutional layers learn multiple kernels (filters) during training, each specializing in detecting different features. The output of each kernel is called a feature map, and stacking these maps along the channel dimension allows the network to represent increasingly complex patterns as the depth of the network increases.

Convolutions have several important properties:

- **Local connectivity:** Each neuron in the feature map is connected only to a small region of the input, allowing the network to focus on local patterns.
- **Parameter sharing:** The same kernel is applied across the entire input, greatly reducing the number of parameters compared to fully connected layers.
- **Translation invariance:** Because the same kernel is used everywhere, the network can detect features regardless of their position in the input.

In our architecture, the first convolutional layer (C1) takes the raw image input and applies several 5×5 kernels to produce feature maps that highlight basic visual patterns. Subsequent convolutional layers (C3, C5) operate on the outputs of previous layers, enabling the network to build up a hierarchy of features from simple to complex.

Subsampling Layers (S2, S4). Subsampling layers reduce the spatial resolution of the feature maps through subsampling, also known as pooling. Basically, subsampling makes the image smaller by combining multiple pixels into one. Subsampling reduces the dimensionality of the data, which decreases the number of parameters and computations in the network, and helps make the network invariant to small translations and distortions in the input image. In the original LeNet architecture, average pooling (subsampling) is used, where each unit in the subsampled feature map computes the average value over a

small local region (e.g., 2×2 window) of the input feature map, possibly followed by a trainable coefficient and bias.

Mathematically, for an input feature map X and a pooling window of size $k \times k$, the subsampled output Y at position (i, j) is given by:

$$Y(i,j) = \frac{1}{k^2} \sum_{m=0}^{k-1} \sum_{n=0}^{k-1} X(si + m, sj + n) \tag{3}$$

where s is the stride (step size) of the pooling operation. This operation reduces the spatial dimensions while retaining the most salient information, making the network more robust to small shifts and distortions in the input.

Fully Connected Layer (F6). This layer is fully connected to the previous layer and has 84 units. It integrates the features extracted by the convolutional and subsampling layers to make a final classification.

Mathematically, the forward pass of a fully connected (dense) layer can be expressed as:

$$\mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{b} \tag{4}$$

where $\mathbf{x}$ is the input vector (flattened output from the previous layer), $\mathbf{W}$ is the weight matrix, $\mathbf{b}$ is the bias vector, and $\mathbf{y}$ is the output vector of the fully connected layer. Each output unit computes a weighted sum of all input units plus a bias, enabling the network to learn complex, non-linear combinations of features.

Other key components include:

- **Loss Function (MSE):** The loss function used was Mean Squared Error (MSE), which measures the difference between the desired output and the output produced by the system. MSE is defined as:

  $$\text{MSE} = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2 \tag{5}$$

  where N is the number of samples, y_i is the observed value, and $\hat{y}_i$ is the predicted value. MSE is a common choice for regression tasks and was chosen for its simplicity and effectiveness in minimizing the error between predicted and actual values.

- **Optimizer (Momentum-based SGD):** The optimizer used was a momentum-based Stochastic Gradient Descent (SGD) algorithm. The momentum term helps accelerate the gradient vectors in the right directions, leading to faster convergence. The update rule for momentum-based SGD is given by:

  $$v_t = \gamma v_{t-1} + \eta \nabla_W E(W) \tag{6}$$

  $$W = W - v_t \tag{7}$$

  where γ is the momentum term, η is the learning rate, v_t is the update vector at time step t, and $\nabla_W E(W)$ is the gradient of the loss function with respect to the weights. The momentum term helps smooth out the updates and avoid oscillations, useful in deep networks.

- **Activation Function (Sigmoid):** The activation function used was the sigmoid function, defined as:

  $$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{8}$$

  where x is the input to the activation function. The sigmoid function introduces non-linearity into

the model, allowing it to learn complex patterns. It maps any input value to a value between 0 and 1, which can be interpreted as a probability.

- **Weight Initialization:** The weights were initialised using the method "LeCun initialisation." This method was designed to preserve the variance of neural activations during the forward pass and is particularly suited for activations like the hyperbolic tangent (tanh). The weights are initialized from a normal distribution with mean 0 and variance $\frac{1}{n}$, where n is the number of inputs to the layer:

$$W \sim \mathcal{N}\left(0, \frac{1}{n}\right) \tag{9}$$

In our modern CNN recreation:

- **Input Layer:** The input layer takes images with three channels, which are first converted to grayscale and resized to 28×28 pixels. The images are then normalized with a mean of 0.5 and a standard deviation of 0.5 for each channel. This is similar to the original model, which also focused on capturing essential features in a standardized size.

- **Convolutional Layers (C1, C2):** These layers extract features from the input image using convolutional operations with different kernel sizes and numbers of feature maps. The first convolutional layer (C1) takes input with 3 channels and produces 6 output channels using a kernel size of 5×5 with a stride of 1 and no padding. The second convolutional layer (C2) takes input with 6 channels and produces 16 output channels using a kernel size of 5×5 with a stride of 1 and no padding.

- **Max Pooling Layers (S1, S2):** TheThe architecture of the CNN is modeled after the original 1998 CNN ([1]) designed for handwritten digit recognition. With layers:

- **Flatten Layer:** This layer flattens the output from the previous layer.

- **Fully Connected Layers (F1, F2, F3):** These layers integrate the features extracted by the convolutional and pooling layers to make a final classification. The first fully connected layer (F1) takes 256 input features and produces 120 output features. The second fully connected layer (F2) takes 120 input features and produces 84 output features. The third fully connected layer (F3) takes 84 input features and produces 5 output features. The choice of 84 units in the second fully connected layer is inspired by the original model.

The loss function and optimiser remains the same, however, other key components include:

- **Activation Function (ReLU):** The ReLU activation function is defined as:

$$f(x) = \max(0, x) \tag{10}$$

where x is the input to the activation function. This is a change from the original model, which used sigmoid activation functions. Unlike the original model, which used sigmoid activation functions, ReLU helps mitigate the vanishing gradient problem. The gradient of the ReLU function is either 0 or 1, ensuring stable gradient flow during backpropagation. ReLU is also computationally efficient, as it involves simple operations without expensive exponentiation. It leads to sparse activations, meaning fewer neurons are active at any time, helping in learning more efficient representations.

- **Weight Initialization:** The weights are initialized using He initialization, which is particularly suited for networks with ReLU activation functions. For a layer with n input units, the weights are

initialized from a normal distribution with mean 0 and variance $\sqrt{\frac{2}{n}}$:

$$W \sim \mathcal{N}\left(0, \sqrt{\frac{2}{n}}\right) \tag{11}$$

This is different from the original model, which used LeCun initialization. It is designed for networks with ReLU activation functions. It maintains the variance of activations and gradients across layers, which is essential for effective training. By scaling initial weights appropriately, He initialization ensures consistent variance of activations across layers. This leads to faster convergence during training, maintaining a good balance of activations and gradients from the start.

3.3 Training Procedure

There is significant class imbalance (see Figure 5). The largest class has 3000 example images, while the smallest class only has 270 images. To address this, we implemented a class balanced cross entropy loss, defined as:

$$\mathcal{L} = \frac{1}{\sum_{i=1}^{n} w_{y_i}} \sum_{i=1}^{n} w_{y_i} \left[-\log p_{i,y_i} \right] \tag{12}$$

where n is the batch size, y_i is the true class label for sample i, p_{i,y_i} is the predicted probability for the true class, and w_{y_i} is the normalized inverse frequency of class y_i:

$$w_c = \frac{1/N_c}{\sum_{k=1}^{K} 1/N_k} \tag{13}$$

with N_c the number of samples in class c and K the total number of classes. Unfortunately our class balanced loss function never saw significant application. After realizing that we would not be able to run large-scale training runs, we made several compromises for performance sake. One of them was to train on only five classes, with only 2,500 training images per class selected randomly. Considering using the same number of samples per classe reduces class-balanced loss to standard cross-entropy loss, we did not end up using it.

The dataset is filtered to include only the chosen classes:

- 1: Speed limit (30 km/h);
- 2: Speed limit (50 km/h);
- 10: No passing for vehicles over 3.5 t;
- 13: Give right of way;
- 38: Pass this sign on the right.

The indices of the samples belonging to these classes are collected, shuffled, and clipped to a maximum of 2500 samples per class. Shuffling ensures that the data is randomized, avoiding any bias that might be present if the data was ordered in a particular way. Clipping to a maximum of 2500 samples per class ensures that the dataset is balanced, preventing the model from being biased towards classes with more samples.

The labels are mapped to new indices and one-hot encoded, and the filtered dataset is saved to a CSV file named `filtered_labels_encoded.csv`. The dataset is split into training and testing sets, with 85% of the data used for training and the remaining 15% for testing, matching the original split in the original CNN model. This allows us to compare hyperparameters by training on the training set with different

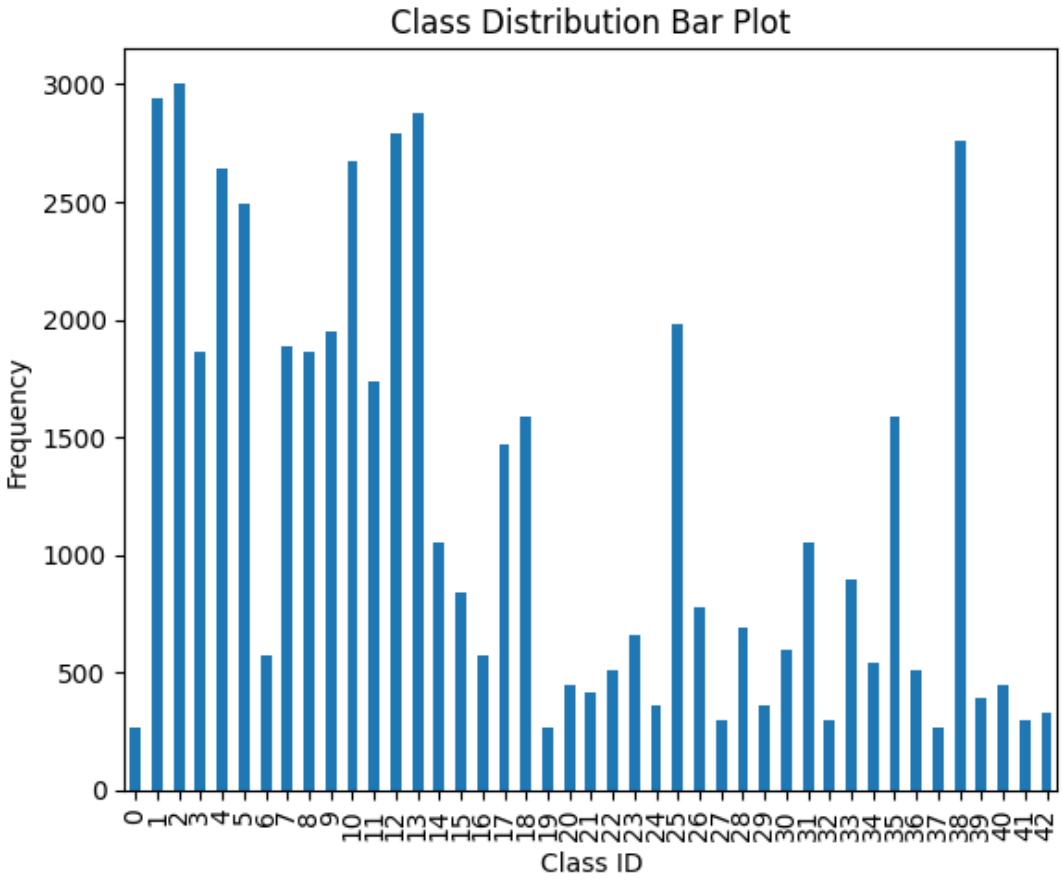

Figure 5: Class distribution of the GTSRB dataset. The dataset is imbalanced, with the smallest class having 270 images and the largest 3000. The mean is 1206 images per class.

settings and evaluating on the test set. Mixing these would cause data leakage and increase the danger of overfitting.

We introduced cross-validation to better evaluate the hyperparameters of the algorithm. In specific, k-fold cross-validation was implemented:

1. Shuffling dataset indices using a fixed seed.
2. Splitting those indices into k folds with roughly the same size.
3. For each fold, one subset of the fold was used for validation, leaving $k-1$ folds for the training set.

After the k-fold splitting was done, cross-validation would load the training and validation indices into separate data loaders, where the training indices were shuffled and the validation indices were not. Our k-fold cross validation algorithm re-initializes the model and the optimizer every fold, preventing any form of leakage. The performance of an algorithm is then simply the mean of the performance across all folds' test sets.

Hyperparameter search was also introduced along with the cross-validation algorithm. Here, we only considered the learning rate and weight decay, as other hyperparameters were picked by following the original LeNet model. The values were chosen in pairs based on some initial experiments. A simple grid search was performed, where would look for the hyperparameters used to train a model with the highest accuracy.

We note that although implemented, we did not get to use our k-fold cross-validation or hyperparameter search. We did run our algorithm, but due to the lack of learning in general changing the hyperparameters had no effects. However, implementation of cross-validation remains for reproducibility and potential future use.

4 Results

Unfortunately, we are not able to report significant results. The model achieved a test accuracy of 19.17%, which is not better than random guessing (20% for 5 classes). The test loss was measured at 1.4395, indicating poor performance overall. We believe this is because the model does not have enough parameters, so it is not able to learn the features of the dataset well enought to generalize (or even recognize the training data). However, we are unable to significanly increase the model size due to computational constraints, as our implementation is not optimized for speed like libraries such as PyTorch or TensorFlow. Training larger models takes a very long time, and we were not able to complete training runs for larger models within the project timeframe and our personal hardware constraints.

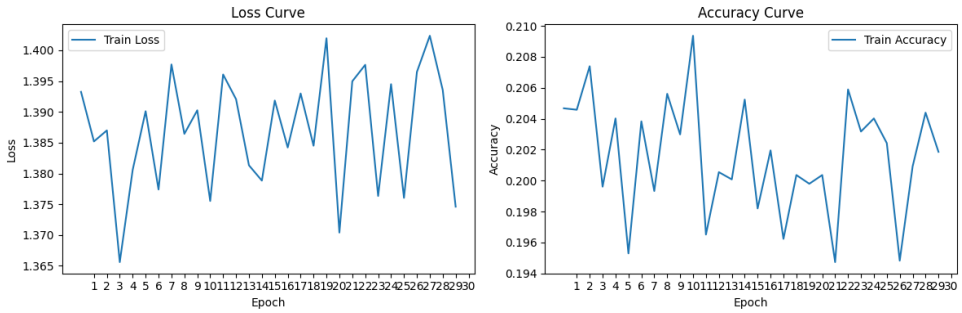

Figure 6: Learning curves for custom CNN implementation showing training loss (left) and accuracy (right) over 30 epochs. The significant fluctuations indicate unstable training dynamics.

As shown in Figure 6, the learning curves exhibit significant fluctuations in both loss and accuracy over the epochs, suggesting that the training process was unstable and the model struggled to converge effectively. This instability is likely due to the small size of the model and the complexity of the GTSRB dataset.

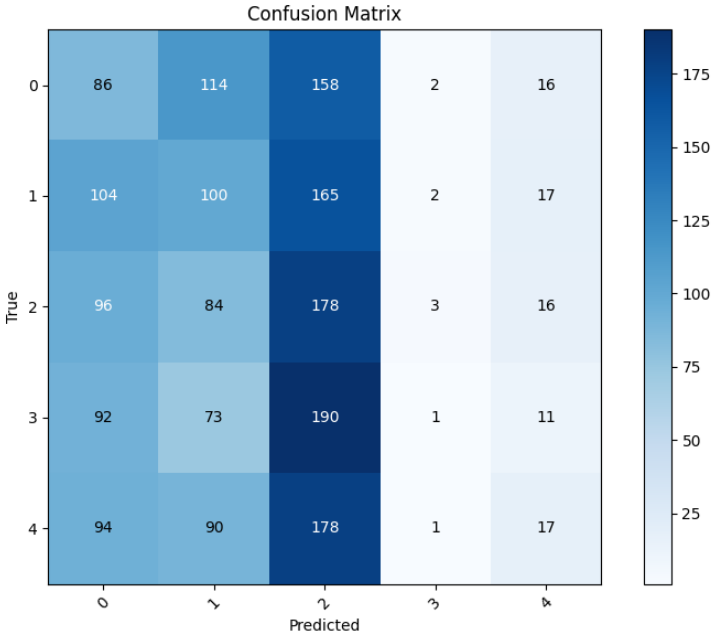

Figure 7: Confusion matrix for our custom CNN implementation. The matrix reveals substantial misclassifications across all classes, with no clear pattern of correct predictions.

The confusion matrix in Figure 7 further illustrates the model's difficulties, showing a high number of misclassifications across all classes. This indicates that the model was unable to effectively distinguish between different traffic sign categories.

5 Discussion

As evident by our results, the model didn't achieve quite as well as we hoped it would. Despite these lacking results, we still learned a lot throughout this project. First, we learned that architecture matters a lot for these types of projects. The original LeNet CNN Lecun et al. [1] is quite small, and designed for small classification tasks (e.g. MNIST). We think that this architecture does not quite suit a bigger, more complex dataset, like GTSRB, which contains a lot more variation in the types of classes and imagery. We argue that a bigger model would be required, with more layers or neurons. Furthermore, we also realized the importance of proper data exploration, and in turn create a more extensive pipeline.

We think that the main reason that the model did not achieve a high accuracy is due to its small nature. We argue that a model with more layers or more neurons would have been able to capture the details better, resulting in better generalization and a higher accuracy. Additionally, different preprocessing steps could have resulted in a different performance. We note that the shape we are resizing all images to (28 x 28 pixels) is quite small, and images could have potentially lost a lot of important, differentiating features in this process. Other steps could have potentially be used more carefully, which might influence the model's robustness and ability to learn from the images.

We do not see a lot of future potential for this project. Although writing a neural network from scratch was a ton of fun, and we learned a lot from doing so, it is not very practical to do this for other machine learning projects. Using pre-built machine learning libraries, like PyTorch, Keras or TensorFlow

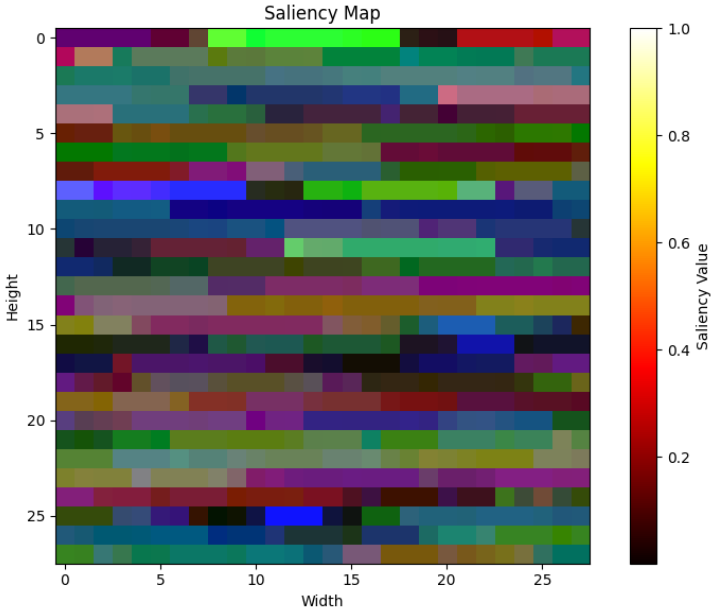

Figure 8: A sample's saliency map from our custom implementation showing which image regions most influenced the model's predictions. The diffuse activation patterns suggest the model failed to learn meaningful features.

speed up the process of building a neural network tremendously. For that reason, we argue that just using these libraries instead is a better practice, as these are used a lot in machine learning projects.

Out of curiosity, we ran the same model through PyTorch for personal fun comparison, which can be found in the appendix.

6 Conclusion

In this project, we set out to implement a convolutional neural network from scratch to classify traffic signs from the GTSRB dataset. Our implementation followed the original architecture, originally designed for handwritten digit recognition. While our model achieved only modest results (19.17% accuracy), the process provided valuable insights into the challenges of neural network implementation.

Our approach involved implementing all components of a CNN without relying on established deep learning frameworks. This implementation allowed us to develop a deeper understanding of how neural networks function at a fundamental level.

Some key findings:

- Architectural limitations: The simple LeNet design proved inadequate for the complexity of traffic sign classification, highlighting the importance of appropriate model selection for specific tasks.
- Implementation challenges: Our unstable training dynamics (Figure 6) and poor feature learning (Figure 8) demonstrated the complexity of properly implementing even seemingly simple neural networks.
- Data considerations: Our preprocessing pipeline, particularly the aggressive resizing to 28x28 pixels, likely destroyed important discriminative features in the traffic sign images.
- Performance realities: The significant gap between our implementation and practical performance levels underscored why production systems rely on optimized libraries.

Perhaps the most valuable lesson learned was the appreciation for what established frameworks like PyTorch provide. While our implementation served as an excellent educational exercise, it lacked the robustness and efficiency needed for real-world applications. This experience has given us firsthand understanding of why practitioners typically use these optimized libraries for actual machine learning projects.

Ultimately, this project successfully achieved its educational objectives by revealing both the capabilities and limitations of basic neural network architectures, while providing crucial experience in implementing machine learning systems from the ground up.

References

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

[2] Stallkamp, J., Schlipsing, M., Salmen, J., and Igel, C. (2011). The German Traffic Sign Recognition Benchmark: A multi-class classification competition. In *IEEE International Joint Conference on Neural Networks*, pages 1453–1460.

[3] Stallkamp, J., Schlipsing, M., Salmen, J., and Igel, C. (2012). Man vs. computer: Benchmarking machine learning algorithms for traffic sign recognition. *Neural Networks*.

A Use of AI Tools

We used LLM tools during both writing the report and coding. The helper of choice was GitHub Copilot using GPT-4.1. During coding, it was used in EDA for plotting as well as to write non-ML boilerplate code, such as converting .ppm images to a numpy array.

During writing, the autocomplete feature was used to speed up formulation of sentences as well as format figures and charts. The ideas and structure of the report are ours.

B Additional Materials

Figure 9: 25 random samples from the dataset.

C PyTorch Implementation Comparison

For reference, we include results from a PyTorch implementation of the same architecture to highlight performance differences that may stem from implementation details rather than architectural choices.

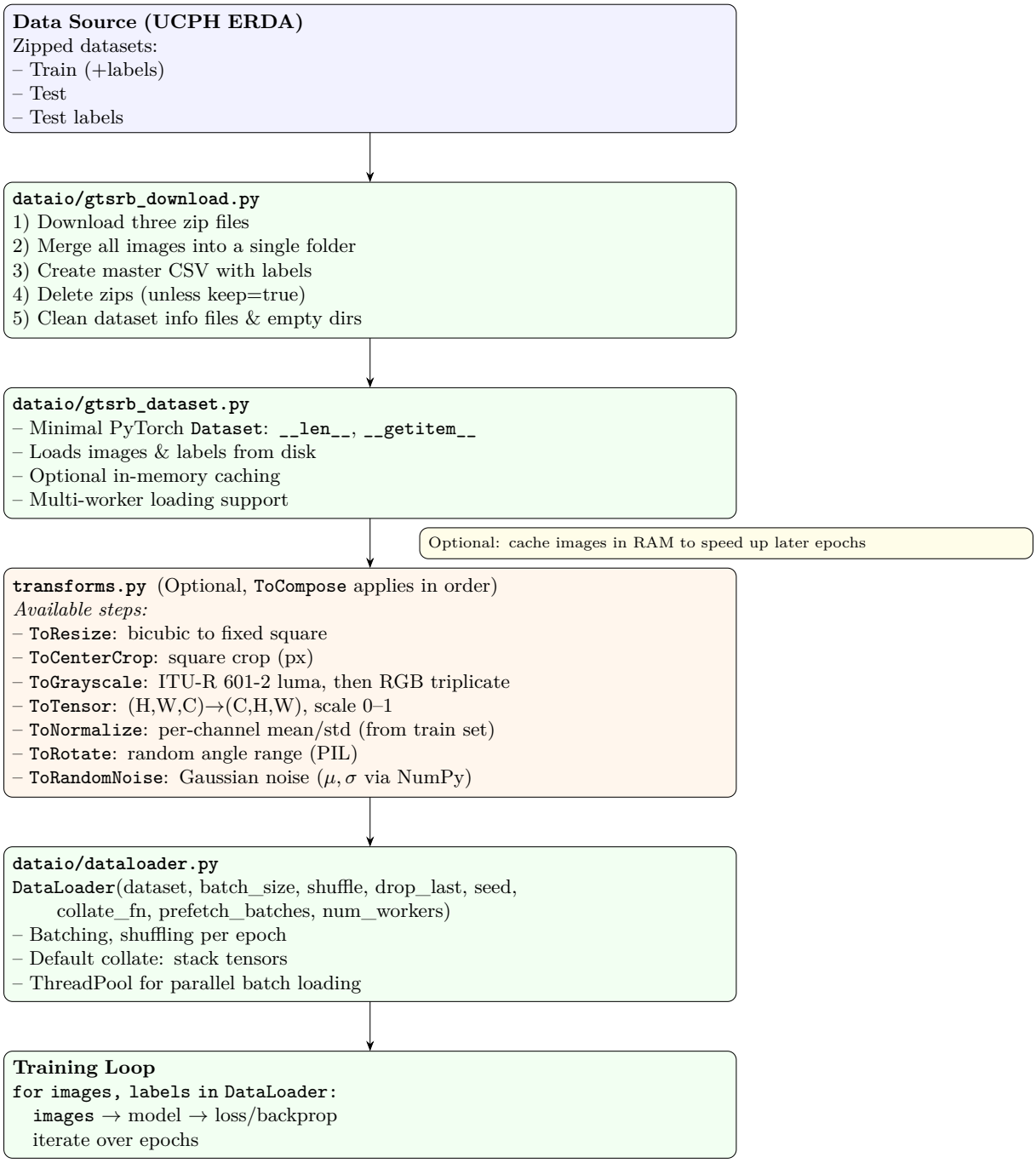

Figure 10: GTSRB data pipeline from download → dataset → transforms → dataloader → training.

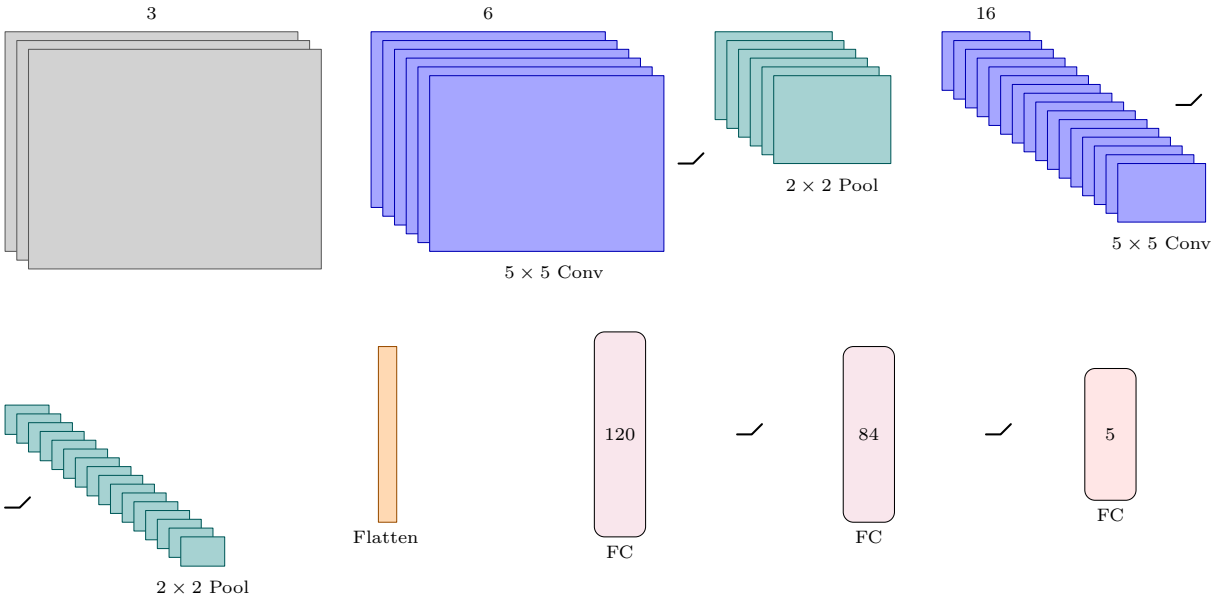

Figure 11: Visual representation of the original CNN architecture we replicated. The presence of a small ReLU marker indicates a ReLU activation after that layer.

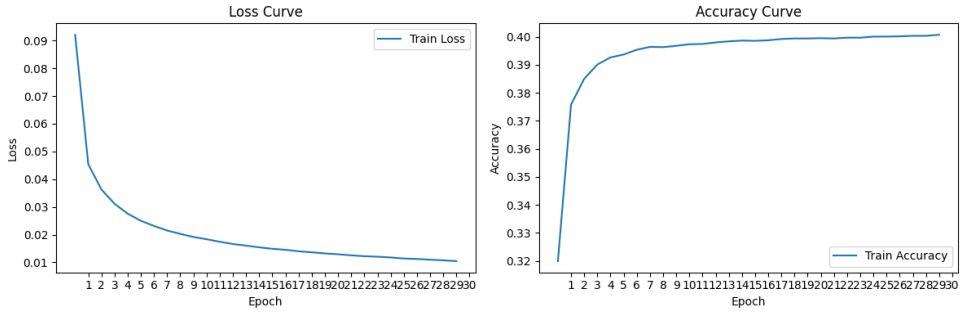

Figure 12: Learning curves for PyTorch implementation of the same architecture, showing more stable training dynamics compared to our custom implementation.

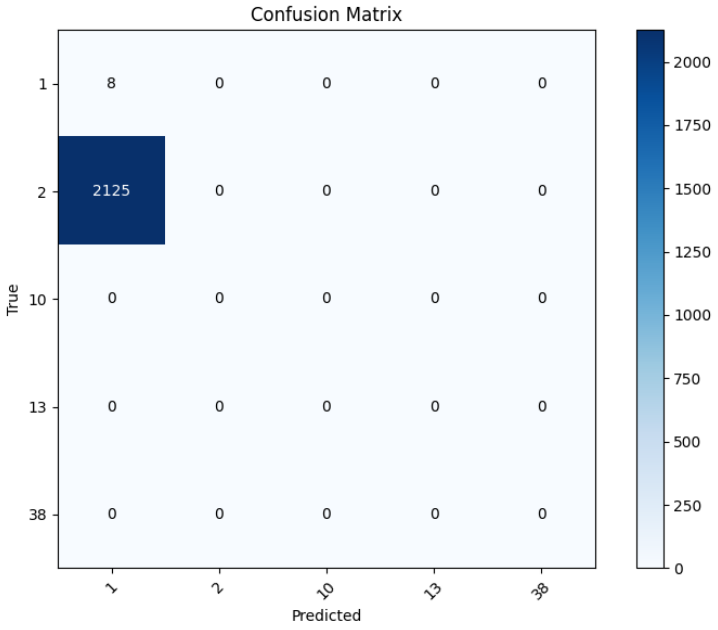

Figure 13: Confusion matrix for PyTorch implementation.

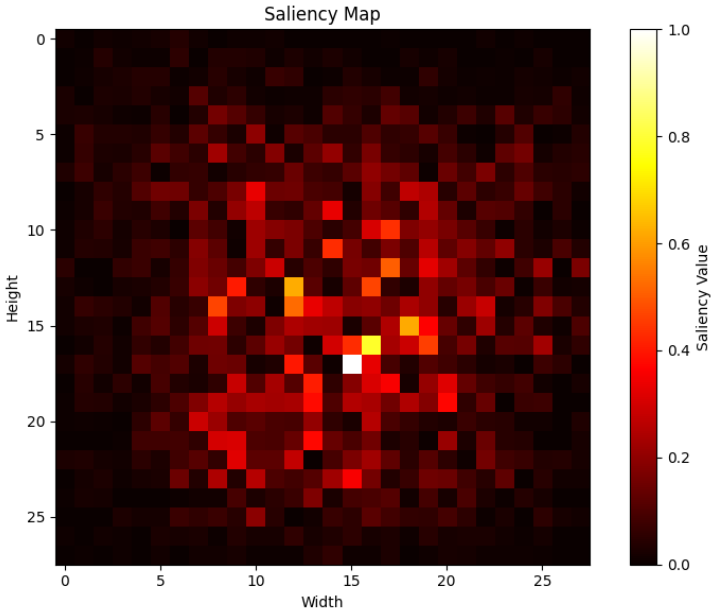

Figure 14: A sample's saliency map from PyTorch implementation showing more focused activation patterns compared to our custom version (Figure 8).