7 WEEKS IN EINC

Neuromorphic Computing

Kalogirou Asterios – 28/04/2025

Contents

BEGINNING

First Four Weeks

- The first 4 weeks were spent on learning as much of the basics as possbile before getting some hands on experience so I dont get overwhelmed.

- A lot of reading into the team's projects and papers was performed, as well as some of the team's tutorials, which helped me to understand some of the modeling better and gave me a headstart regarding Neural Networks.

- Of course nothing beats hands-on experience so in those four weeks I also got accomodated with the main library that I was going to be using, **PyTorch**.

First Four Weeks

- Some PyTorch tutorials were completed explaining the API and also training some models from scratch with the MNIST dataset.
- This specific tutorial made me build a ANN and two CNN models to classify the MNIST and the Fashion - MNIST datasets.
- With this new found knowledge it was time to move forward !

CREATING A DATASET

Dataset (Simple)

- After the first 2-3 weeks a project idea was being discussed and first task was officialy given.
  - Starting goal:

Create a dataset with some parameterization, so it is *rather* easy to classify. After some iterations of this dataset and some consulting with my supervisor (E.A.) a final function following the sin function was chosen.

Results for the Custom Dataset

```python

def sin_distribution(size: int, numbrer_of_peaks: int, orientation: str, noise_level: float):
    """
    Creating a funtion that takes the input of x and y axis that eventually the Image is going to have
    The end result should be a distribution in grayscale that follows the sin's wave form

    Parameters:
    size --> this is the size of the x and y axis or rather the # of columns and rows
    number_of_peaks --> this is the number of peaks or the # of black lines
    orientation --> chooses between horizontal and vertical lines
    noise_level --> the noise intenisty higher = more noise

    """
    bing = np.linspace(0, 2 * numbrer_of_peaks*np.pi, size)

    if orientation == "vertical":
        # we are applying the sin function into the starting evenly spaced array that we can control its peaks
        # after we add 1 so we dont get any negative numbers
        # then we tile this new aray repeating it with the size parameter and also reshaping it
        # into a size by size array
        final = np.tile(np.sin(bing) + 1, size).reshape(size, size)

    elif orientation == "horizontal":
        # here while the first 2 lines stay the same
        # we are using indexing on the bing array so we can make this into
        bing = np.sin(bing) + 1
        final = np.tile(bing[:, np.newaxis], (1, size))
    else:
        raise ValueError(
            "This Orientations either doesnt exist or isnt implemented quite yet")

    # we apply the guassian filter the bigger the sigma the more the blur/noise we have
    final = gaussian_filter(final, sigma=0.3)

    # we also marametrize the noise semi randomly
    noise = np.random.normal(0.5, noise_level, final.shape)
    # and then clip the result
    # this is aslo customizable regarding the intensity of the pixels
    final = np.clip(final + noise, 0, 10)

    # greyscale this to 8 bits
    final = (final/10.0 * 255.0).astype(np.uint8)

    return Image.fromarray(final)
    # return Image.fromarray(final)

sin_distribution(16,4,"horizontal",0.3)
```

Figure 1: Code for the main dataset function

Yes this is the pixel accurate test image, the size limitations were discussed with Elias and Eric (mostly between them) so it can later be implemented easier on hardware.

Figure 2: Pixel Accurate Image Result

Results for the Custom Dataset

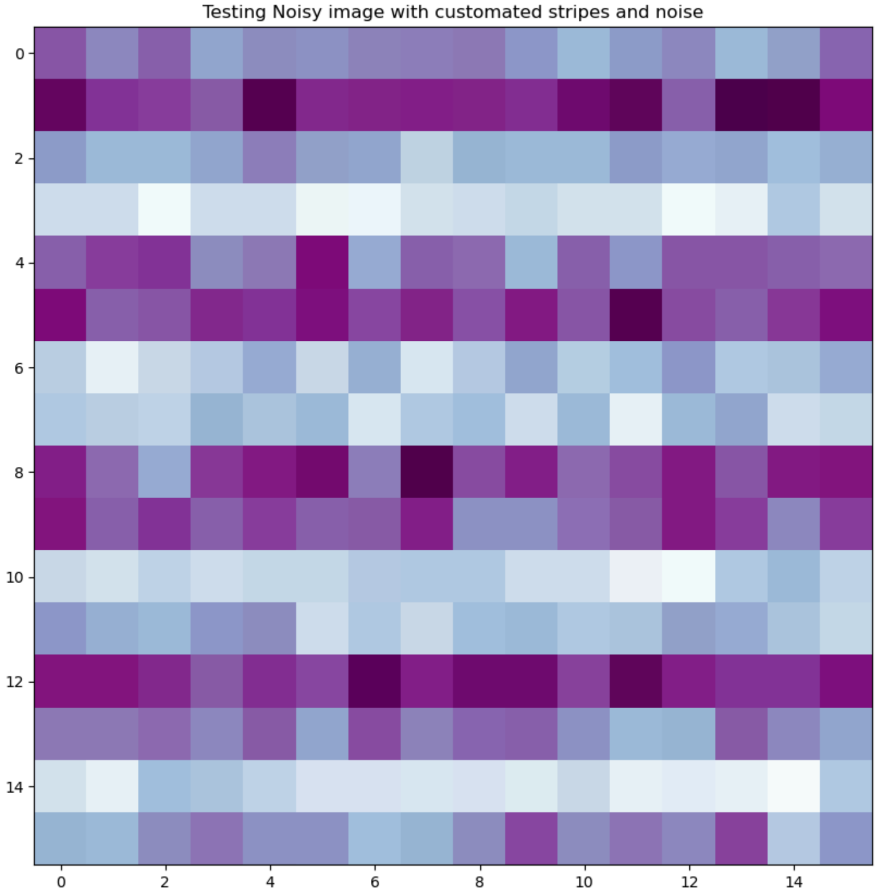


Figure 3: Scaled up plotted test Image

This is the plotted image that was randomly generated before.
A second python script was created to generate a **num_samples** images with randomized *orientation*, *stripes* and *noise*.
Here is the part of that code that customizes that:

```python
for i in range(num_samples):
        num_peaks = random.randint(1,5) #random amount of peaks
        orientation = random.choice(["horizontal", "vertical"]) #random choice between orientations
        noise_lvl = random.randrange(1, 4)/10 #random noise

        # use the function defined above t
        img = sin_distribution(size=size, numbrer_of_peaks=num_peaks, orientation=orientation, noise_level=noise_lvl)

        #label the pictures with 0,1 (hot one encoding)
        label = 0 if orientation == "horizontal" else 1
```

Dataset (notes)

- The customizable features have a certain limitations that makes sense both *spatial-wise* (e.g. the stripes cant be more that half the image size) and *logically-wise* (e.g. after a certain noise threshold the images become *smooth*)

- Here the labels are encoded to **0** and **1** for horizontal and vertical lines respectively, also the images are saved in such a way : *image.index_encoded.label.png* .
- Meaning we can extract the label directly from the loaded image without the need of a *.csv* file.

Dataset initialization

- By importing the Dataset class from the module torch.utils.data we can create a custom class that inherits from the Dataset class of PyTorch, so we can create it to our liking.
- The vital methods are the transform and the root which enable the dataset to be tracked (located) and also transformed (from png files to Tensors for PyTorch usage)

```python
from torch.utils.data import Dataset
```

Figure 5: Import of Dataset

```python
class CustomSinDataset(Dataset):
    def __init__(self, root: str, transform: Optional[Callable] = None):
        self.root = Path(root)
        self.transform = transform

        # Load all image paths
        all_images = list(self.root.glob("*.png"))

        # Extract the index and label from filenames (assuming format: "index_label.png")
        self.images_with_labels = []
        for img_path in all_images:
            # Parse the filename to get index and label
            parts = img_path.stem.split("_")
            if len(parts) == 2:
                index = int(parts[0])
                label = int(parts[1])
                self.images_with_labels.append((img_path, label, index))

        # Sort by index to ensure consistent ordering
        self.images_with_labels.sort(key=lambda x: x[2])

    def __len__(self) -> int:
        return len(self.images_with_labels)

    def __getitem__(self, index: int):
        img_path, label, _ = self.images_with_labels[index]

        # Load and process the image
        image = Image.open(img_path).convert("L")  # GRAYSCALE

        if self.transform:
            image = self.transform(image)

        return image, label
```

Figure 6: Custom class Dataset

MODELS

Models

Many models were created until a desired model was finalized, the procedure that was recommended by my supervisor was to start broad with the limitations and slowly to start adding new important features.
This method was very effective regarding the learning and the results.

1. The first model was a simple SNN model that was tested using a version of the custom dataset that required a *.csv* file. While the model did learn to differentiate vertical and horizontal lines, 30 epochs were neaded and it never fully converged for this simple task (reaching about 90% accuracy).

Models

2. In the second model we took the existing SNN model that was working and also added a convolution layer mainly using the norse library.

```python

#! DEFINING THE CLASS SNN WITH CONVOLUTIONS
class SNN_MODEL(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, num_steps):
        super(SNN_MODEL, self).__init__()

        self.num_steps = num_steps
        self.hidden_size = hidden_size

        #!DEFINE THE LAYERS WOOHOO
        # Convolution Layer
        self.conv1 = nn.Conv2d(1, 13, 4, 1)
        # LIF Layer
        self.lif1 = norse.LIFCell(p=norse.LIFParameters(alpha=0.5))
        # Pooling layer
        self.pool1 = nn.MaxPool2d(2)
        # Convolution layer 2
        self.conv2 = nn.Conv2d(13, hidden_size, 4, 1)
        # LIF Layer 2
        self.lif2 = norse.LIFCell(p=norse.LIFParameters(alpha=0.5))
        # Befroe we move on we need to calculate the the flattened size and put that in the Linear Fc1 layer
        # after conv 1 apo 16x16 se 13x13, meta to pool1 exoume 6x6 kai meta to allo conv2 exoume 3x3
        # So we flatten the image : hidden_size * 3 * 3
        self.flattened_size = hidden_size * 3 * 3
        # Linear layer
        self.fc1 = nn.Linear(self.flattened_size, output_size)
        # LIF Layer 3
        self.lif3 = norse.LIFCell(p=norse.LIFParameters(alpha=0.5))
```

Figure 7: 1st SCNN model

Models

This is a very inefficient model with many layers that are very unneccesary, to be exact, the model consists of two convolution layers, a pooling layer, 3 LIF layers and a fully connected (linear) layer.
The convergance happens extremely fast at aroun epoch 3 which means we can simplify this model quite a lot !
Since it works though, the next step was to add some kind of encoding or rather add the temporal dimension.

Models

3. The third model’s goal was to add the time dimension to the **mix**.

The results here were much harder to “make sense” because the model was now much simpler and a little harder for me to train since we added the time dimension.
This was done by using a specific import :

```
from norse.torch.functional.encode import poisson_encode
```

Not a lot of time was spend optimizing this model because the encoding we were striving for was a ***constant current over time method.*** And the new simpler and time encoded model looks something like this :

Models

```python
class SimpleSNN(nn.Module):
    def __init__(self, input_size, output_size, num_steps):
        super(SimpleSNN, self).__init__()

        self.num_steps = num_steps

        #! DEFINE THE LAYERS (SIMPLE THOUGH)
        # Convolutional layer
        self.conv = nn.Conv2d(1, 2, 4, 1)
        # Pooling layer
        self.pool = nn.MaxPool2d(2)
        # Flatten where the new size is 13x13 cause of the kernel 4 convolution
        self.flatten_size = 2 * 6 * 6
        # Linear layer to map the flattened features to output classes
        self.linear = nn.Linear(self.flatten_size, output_size)
        # Single LIF layer for the spikes
        self.lif = norse.LIFCell(p=norse.LIFParameters())
```

Figure 8: SCNN + TIME

- So here the inputs get encoded using the *poisson_encode* function that was imported changing the shape of the tensors to [Time, Batch_Size,Channel, Height, Width]

Models

4. Fourth model is almost identical to the previous one with the only difference being the encoding.

```
from norse.torch.functional.encode import constant_current_lif_encode
```

In order to rate this model, the class was decided with highest spike count per sample. ***highest spike count per sample*** in the output layer.

Here an accuracy of ~98 % was achieved not like the 100% convergance but still very good : Here are some results taking random inputs to see if the model has a weakness to any specific pattern.

Models

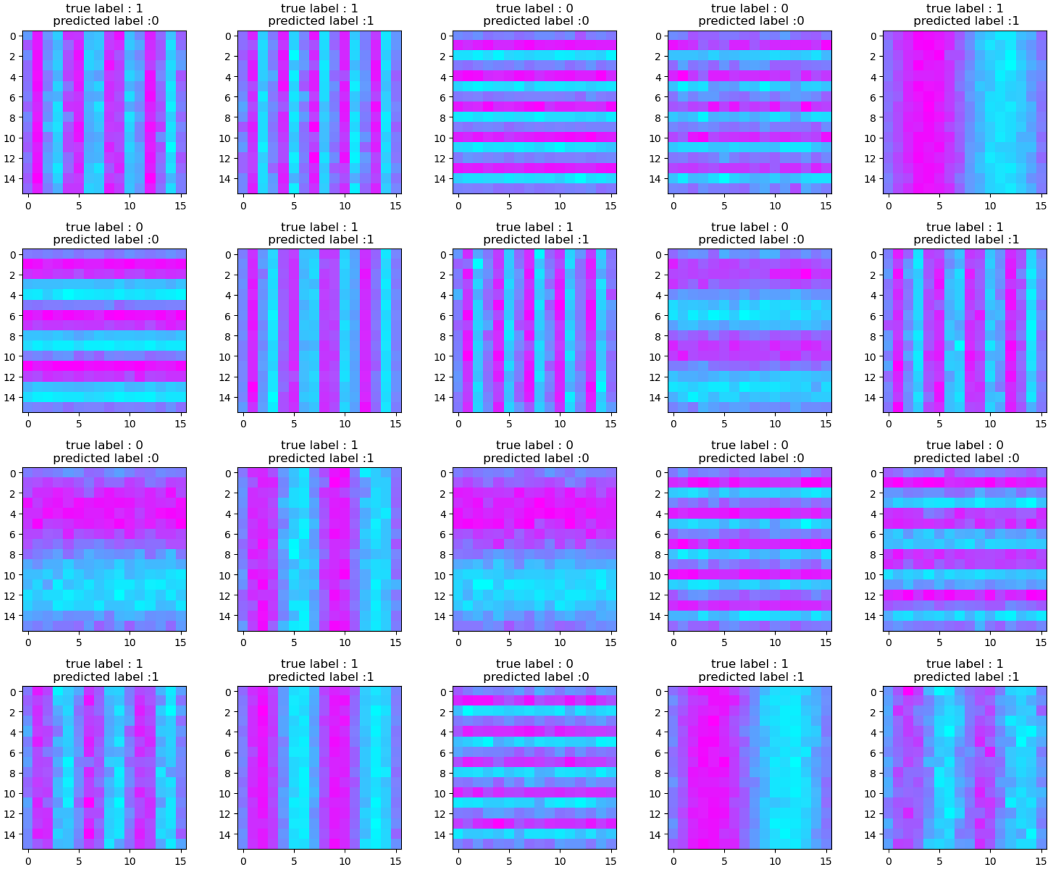


Figure 9: Random Images in the Trained Model

- Here we can see that the model struggles with a great number of stripes while also having high level of noise.
- Still this is a very good result but we can make the model even better by changing the architecture and adding a LI layer.

Models

5. The fifth model is the final model that I have made completely from scratch (so not counting any tutorials) and its the best one to date.

The code for the model looks something like this :

Models

```python
class LI2Model(nn.Module):
    def __init__(self, input_size, output_size, num_steps):
        super(LI2Model, self).__init__()

        self.num_steps = num_steps

        #!LAYERS:
        # Classic conv layer
        self.conv = nn.Conv2d(1, 2, 4, 1)
        # LIF Layer
        self.lif = norse.LIFCell(p=norse.LIFParameters())
        # Classical Pooling layer
        self.pool = nn.MaxPool2d(2, 1)
        # linear or fcl
        self.fcl = nn.Linear(288, output_size)
        # LI 1 layer
        self.li = norse.LICell(p=norse.LIParameters())
```

Figure 10: Code block for the final model

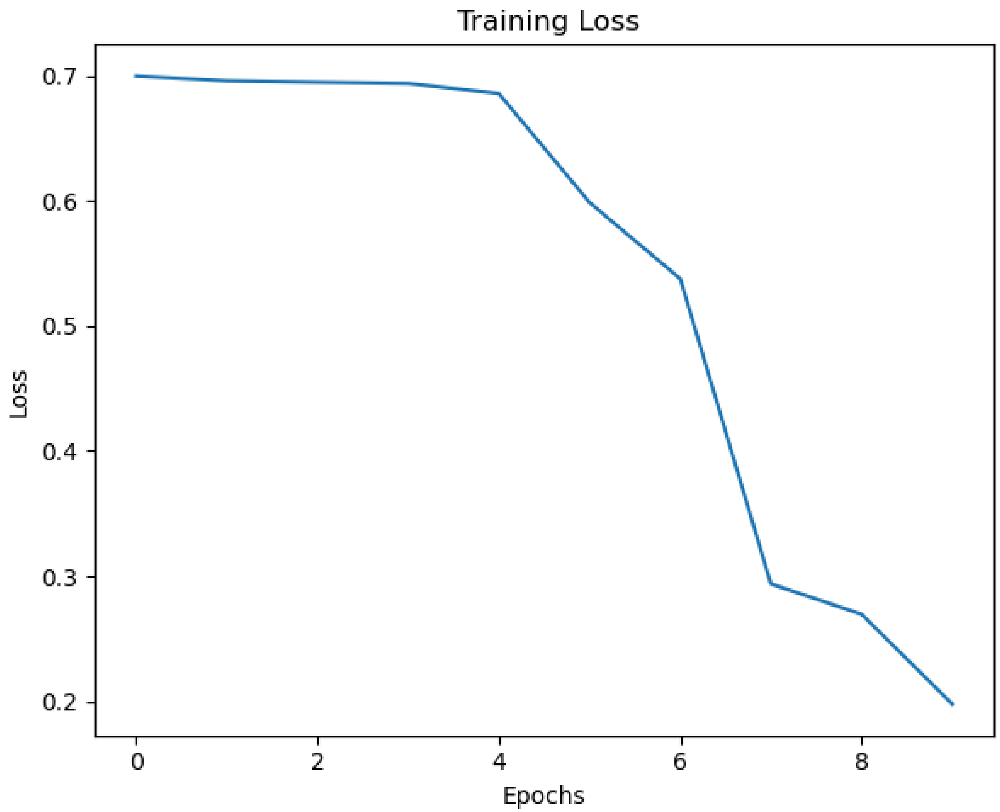
- The network architecture includes a
  - rate encoded input,
  - a convolutional layer with 2 filters size 4x4 ,
  - a LIF layer for the spikes ,
  - a max pooling layer which reduces the spatial dimensionality ,
  - a flatten + linear layer and finally
  - a LI layer which models membrance voltage traces with decay, in order to capture long-term temporal information.

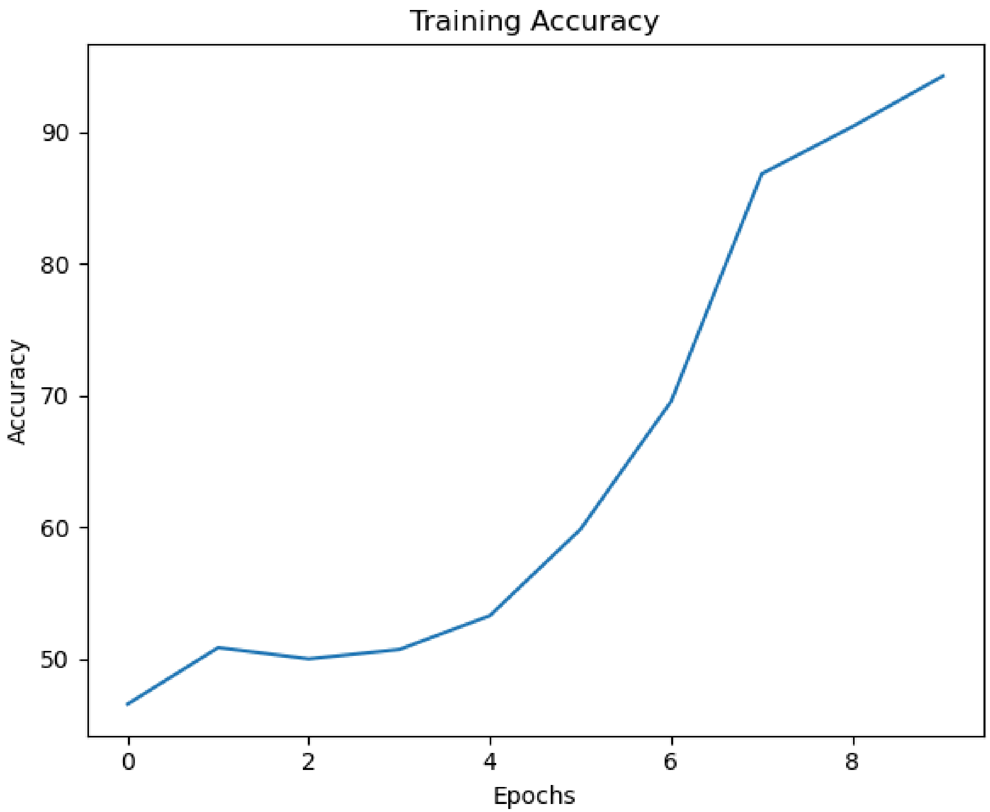
Models

This model is both simple enough to move into hardware and also gets perfect results regarding the taks it is given.

The network fully converges at around epoch 9 meaning we get $\sim 100\%$ accuracy on our custom dataset and since is the best model so far some extra plots were made using a custom python function.

Results of Final Model





Results of Final Model

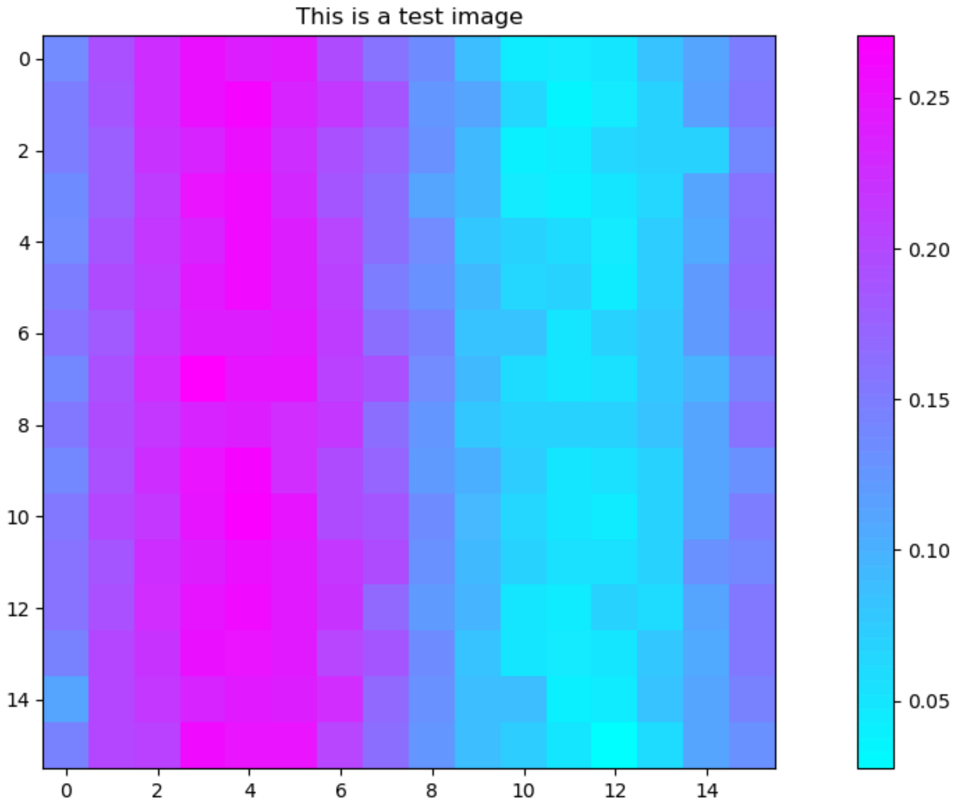


Figure 12: Test Image

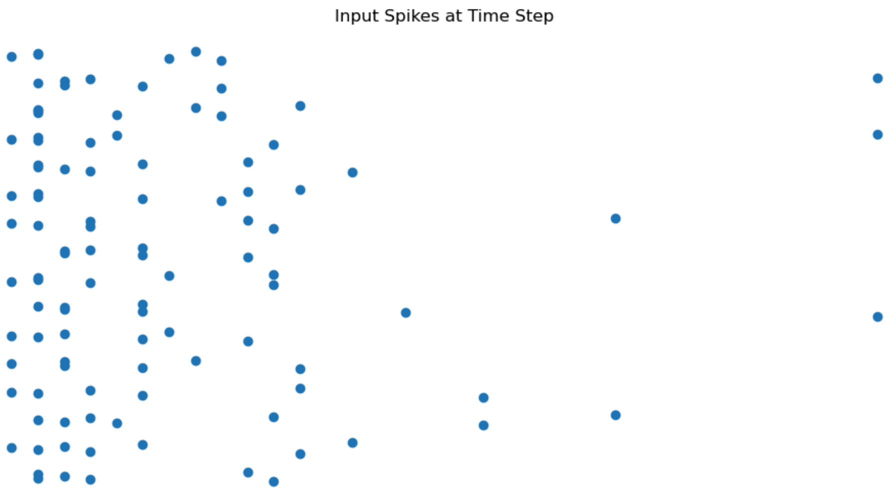


Figure 13: Input Spikes of the Test Image

Results of Final Model

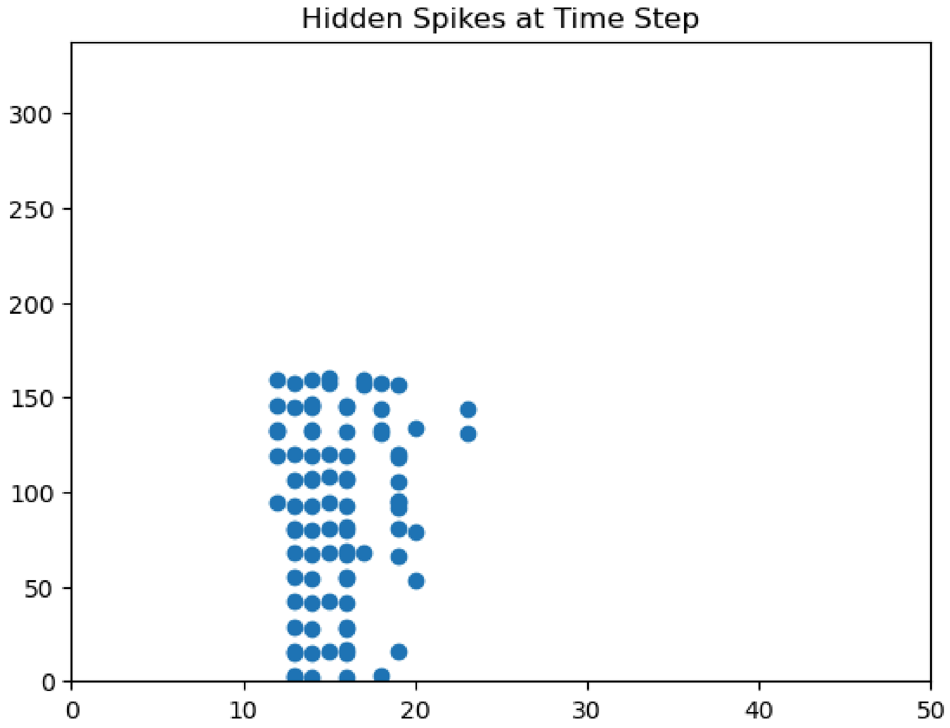


Figure 14: Hidden spikes over Time

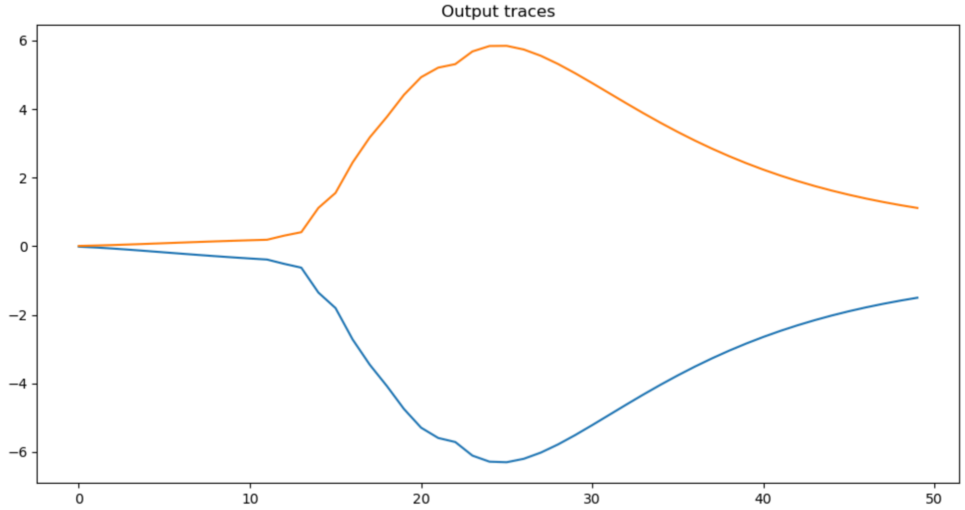


Figure 15: Membrance Voltage Traces of the two Output Neurons over Time

Comments

- Here we can see that the output neurons are symmetric over the $y = 0$ line meaning that classification is possible with only one output neuron.

- Having achieved our starting goal, an idea to change the dataset and make it a little harder was suggested.

- And so this exact model, with minor changes to the scaled encoded input values, was tested on a dataset that has the same parameterization as the previous one with the difference that now striped patches are randomly placed around a noisy grid.

Comments

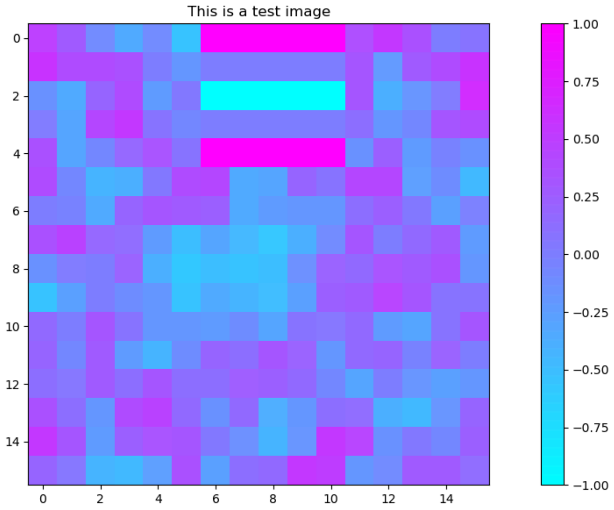


Figure 16: Test image with an easier version

- The dataset with the easier pictures got a training for $\sim$ 18 epochs and it fully converged at around 12 epochs
- While the dataset with the harder images is learning, more epochs are recquired and more changes on the model to make this model converge fully.

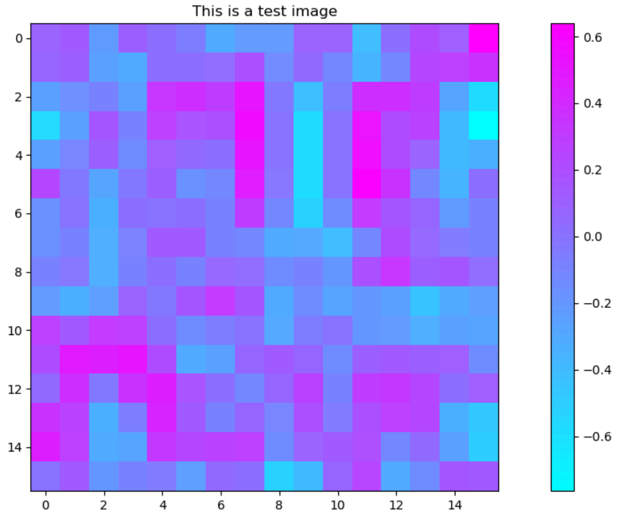


Figure 17: Test image with a harder version

Comments

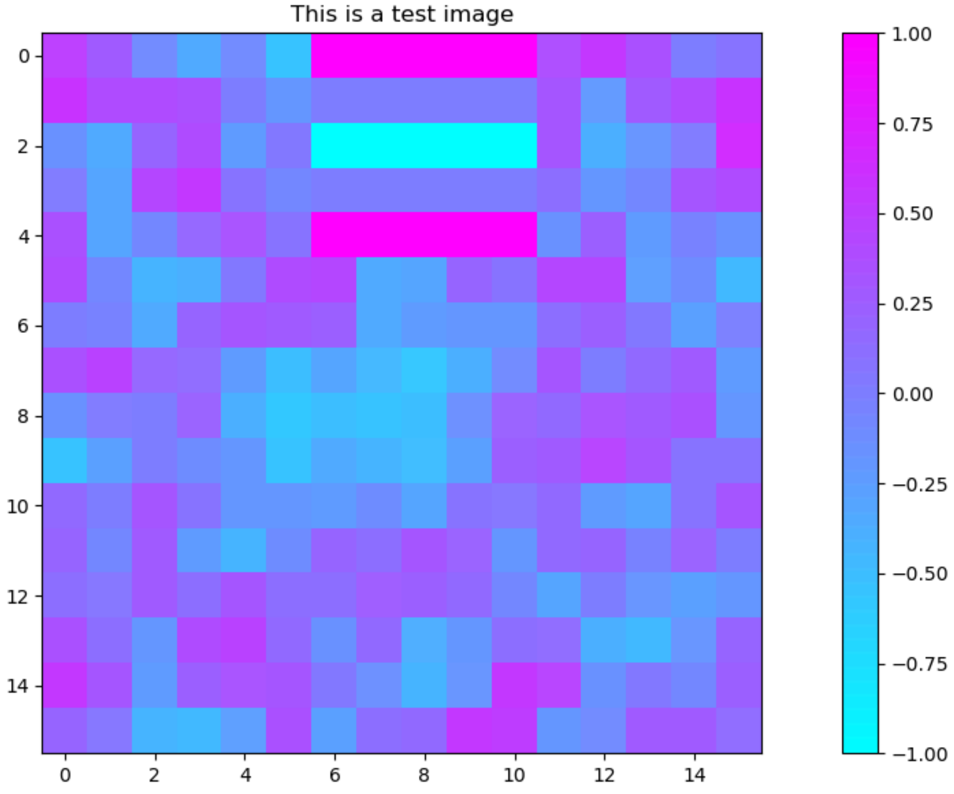


Figure 18: Test image of the easier version

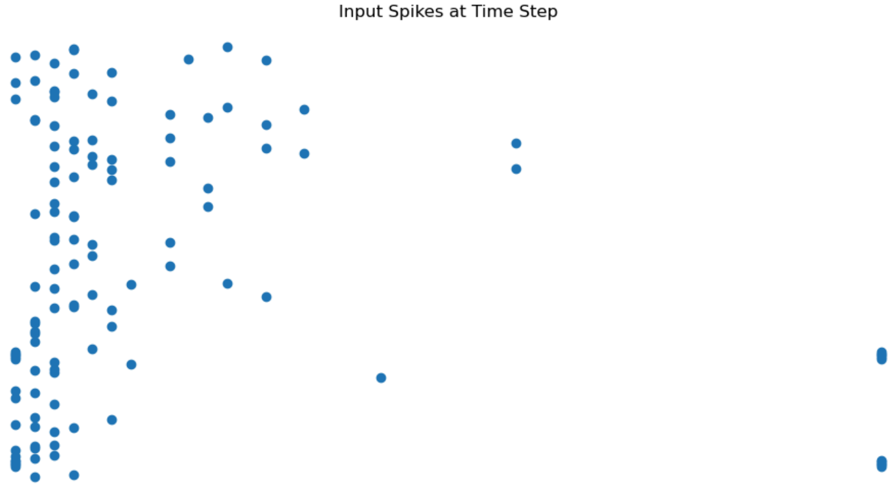


Figure 19: Input spikes over time

Comments

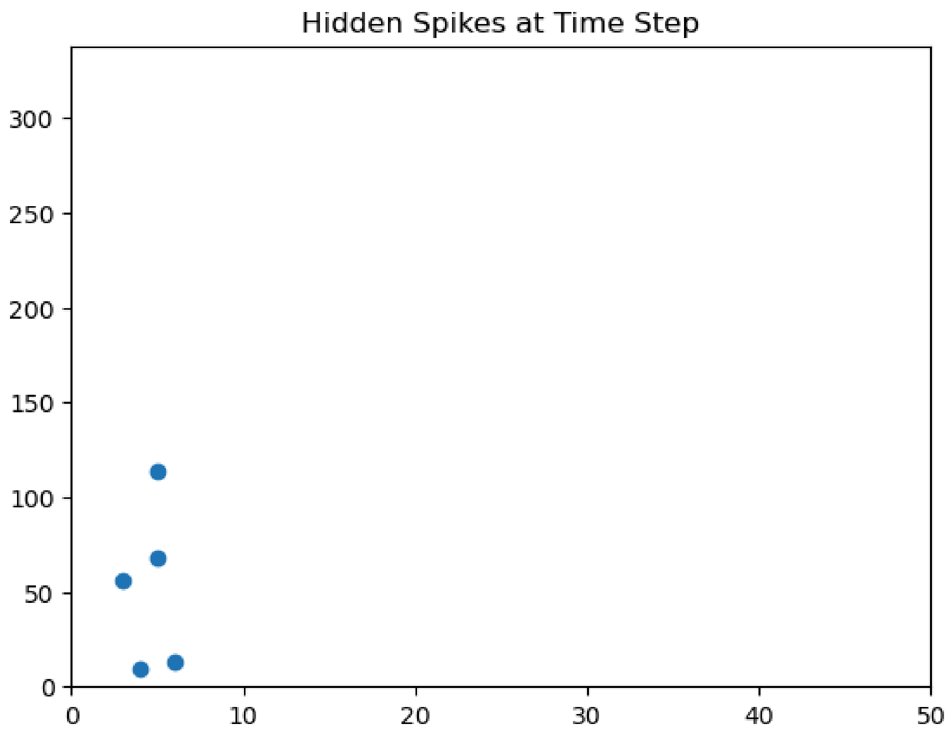


Figure 20: Hidden Spikes

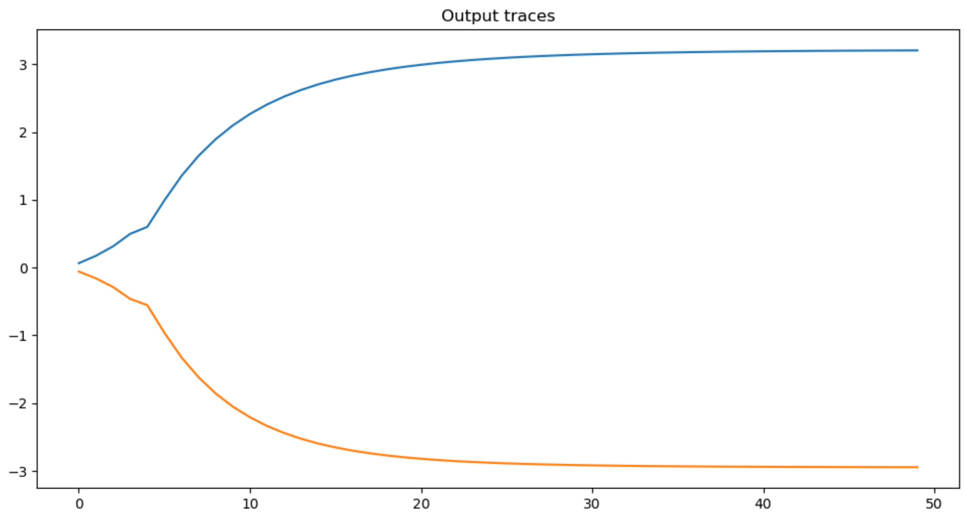


Figure 21: Membrance Voltage traces in the 2 neurons

LOOKING FORWARD

Next Steps

- As of now the next direct step is going to be transfering this model and the (easier) dataset to hardware and to try and make it work there.(it has already begun)
- Also since the membrance voltage trace of the output neurons are symmetric the second existing output neuron could be changed, along with the way that the image is encoded through time to classify the rotation of it (clockwise - counterclockwise).
- Finally some other ideas might pop up which are always fun to talk about and implement.

CONCLUSION

Conclusion

- Over the course of these 7 weeks, I've had the opportunity to immerse myself deeply into the field of programming and neuromorphic computing . I have gained a solid foundation in PyTorch, built and trained spiking neural network models, and developed custom datasets—all of which helped me better understand both the theory and practical application of neural-inspired computation.
- Looking ahead, I'm enthusiastic about continuing this work—particularly as we begin transitioning models to hardware—and I'm eager to keep learning, improving, and contributing to the team's goals.