MS Computer Science -RB
MS Computer Science - UJ
MS Computer Science -PK
MS Computer Science -PG

Image-Based Mathematical Equation Solver using SNN

Rishika Bhanushali – rb1182
Utkarsh Jha - uj22
Prashant Kanth - ppk31
Parth Goel - pg514

5/8/22
525 Brain Inspired Computing
Prof. Konstantinos Michmizos

Abstract

We present supervised learning in Spiking Neural Network for the problem of handwritten digit equation solving using the Gradient Descent algorithm. The entire application is divided into two phases, phase 1 being image segmentation and preprocessing. Phase 2 is the classification part. We present several insights from extensive numerical experiments regarding optimizing learning parameters and network configuration to improve its accuracy. The network uses biologically plausible neural and learning mechanisms and is applied to a subset of the MNIST dataset of handwritten digits and basic mathematical operations. The research aims to assess the classification power of a straightforward biologically motivated mechanism and develop an application to solve the handwritten equations. The network architecture is primarily a feedforward spiking neural network (SNN) composed of Leaky Integrated and Fire neurons and Voltage-based synapses.

1. Introduction

Handwriting Recognition has been a topic of boundless research and importance for more than 30 years. Digit recognition has many applications like number plate recognition, postal mail sorting, bank check processing, etc. Handwriting recognition is much more challenging than OCR, which recognizes machine-printed characters. The former is more complex due to the wide variations in the style or manner in which humans write. It is easier to do OCR because optical characters appear almost the same and are easy to isolate from one another in a word. On the other hand, handwritten characters may not be so easy to tell apart. The project's goal was to attempt to classify the handwritten mathematical operators and digits using Spiking Neural Networks.

The superior computational efficiency of biological systems has inspired the quest to reverse engineer the brain to develop intelligent computing platforms that can learn to execute various classification and inference tasks. SNN exploit event-based, data-driven updates to gain efficiency, especially if they are combined with inputs from event-based sensors, which reduce redundant information based on asynchronous event processing. However, SNNs haven't reached the same accuracy levels as their ANN counterparts in practicality. A significant reason is the lack of adequate training algorithms for deep SNNs since spike signals (i.e., discrete events produced by a spiking neuron whenever its internal state crosses a threshold condition) are not differentiable. But differentiable activation functions are fundamental for using error backpropagation, which is still the most widely used algorithm for training deep neural networks.

In this paper, we introduce a novel supervised learning method for SNNs, which closely follows the successful backpropagation algorithm and trains general forms of deep SNNs directly from spike signals. Our goal is to generate a continuous differentiable signal on which Stochastic Gradient Descent can work. For the ease of working with the Spiking Neural Networks, we have used Nengo DL. The motivation to use Nengo was that backpropagation-based methods are robust and fully supported in Nengo. It is easy to integrate the TensorFlow models as well in Nengo.

2. Background

Spiking Neural Network (SNN) is one of the leading candidates for overcoming the constraints of neural computing and to efficiently harness the machine learning algorithm in real-life applications. The concepts of SNN which is often regarded as 3rd generation of neural network, are inspired by biological neuronal mechanisms that can efficiently process discrete neuronal spike activity over time. The Leaky Integrate and Fire (LIF) neuron is the simple first-order phenomenological spiking neuron model, which can be characterized by the internal state, called membrane potential. The membrane potential governs the spiking of a neuron and integrates the input provided to the network over time. The LIF neuron generates a spike whenever the membrane potential reaches a neuronal firing threshold.

2.1 Feedforward Propagation in SNN

Jason K. Eshraghian et al. [1] explain the striking similarities between ANN and SNN and explain how SNN's can be computationally efficient in doing the same task as ANN. ANNs and SNNs can model the same types of network topologies but SNNs trade the artificial neuron model with a spiking neuron model instead. Much like the artificial neuron model, spiking neurons operate on the weighted sum of inputs, the major difference being that instead of passing the results through a continuous activation function like sigmoid and ReLU, the weighted sum contributes to the membrane potential (U) of the neuron. The rate-encoded spike input, $X[t+1]$, is converted to current through these learnable weights (W). The membrane potential is subjected to a decay rate (β) to reflect optimal sparse/dense spikes (S_{out}) and voltage increment/decrement behavior. As soon as the membrane potential reaches a threshold (U_{thr}), the neuron will emit a spike to its subsequent connections, and is reset. The equation given below drives the feedforward propagation of spikes through the LIF model.

$$U[t+1] = \beta U[t] + WX[t+1] - S_{out}[t]U_{thr}$$

$$S_{out} = 1 \; if \; U[t] > U_{thr} \; else \; 0$$

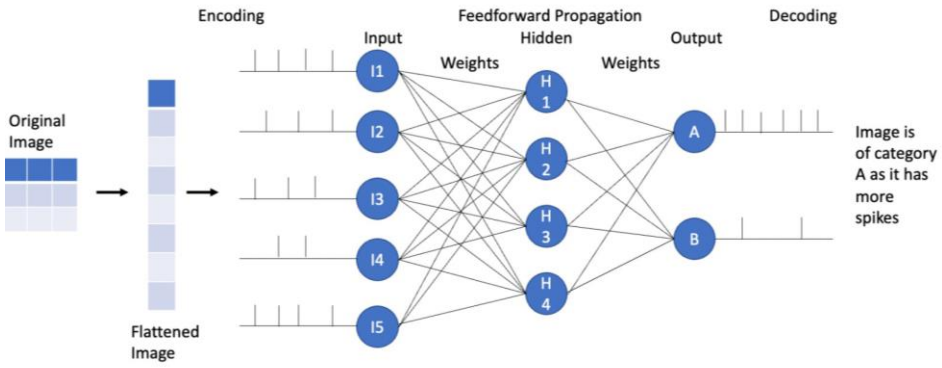


Figure 1: Spiking Neural Network depicting feed forward propagation. Input spikes are multiplied with weight of each layer to compute current, which are then converted to spikes for next layer by using voltage thresholding.

2.2 Backpropagation using Surrogate Gradient Descent

In supervised learning, predictive models often aim to minimize the objective function or the loss function to accurately learn the weight parameters. This is often achieved using backpropagation. The limitation of using backpropagation for training an SNN is resolved using surrogate gradient descent, Friedemann Zenke et al. [2]. As spikes are not continuous, they are not differentiable and gradient of '0' means no update to the weights, also called 'dead neuron' problem. A variety of surrogate gradient functions have been used to varying degrees of success, and the choice of function can be treated as a hyperparameter. Most common surrogate functions are, the sigmoid function: $1/(1+e^{-U+U_{thr}})$, the fast sigmoid function: $(U-U_{thr})/(1+|U-U_{thr}|)$, and a triangular function: max $(1-|U-U_{thr}|, 0)$. This is applied by substituting the Heaviside operator with a continuous function like sigmoid during backward pass of loss. The surrogate gradient thus becomes, $\partial S/\partial U \leftarrow \partial \mathbb{S}/\partial U$.

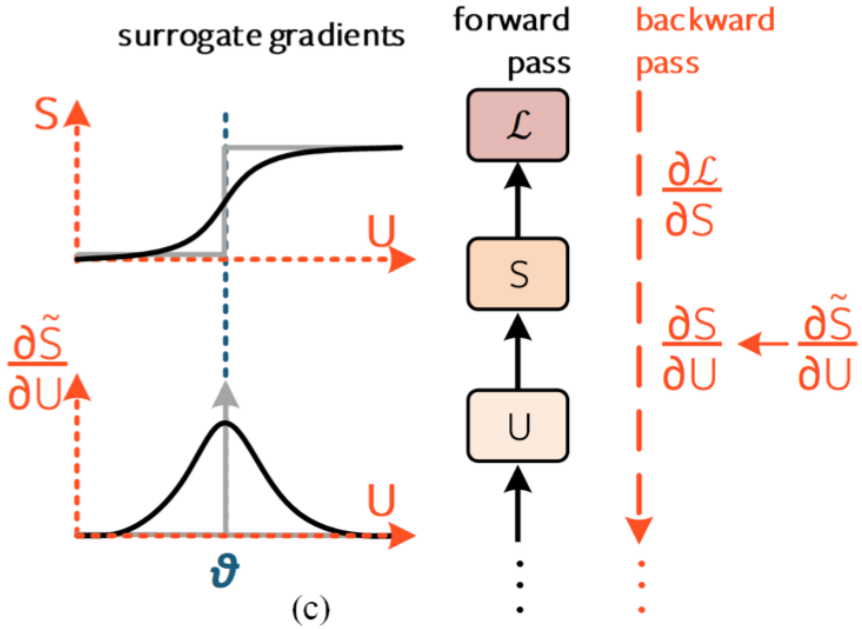


Figure 2: Source: Jason K. Eshraghian et al. [1], Surrogate gradients: the spike generation function is approximated to a continuous function during the backward pass

3. Experimental Design

As illustrated in figure 3, we designed a six-layer Spiking Convolutional Neural Network to classify handwritten digits from the Customized digits, operators' database. The challenge was to get a good dataset and various instances of each digit as well as operators. We made use of a Kaggle dataset to train our model [4]. The sizes of the images in our dataset had varying lengths (mostly 155*135), so we had to resize them to a particular size to retain the uniformity in training data. In the train-test split, all images were resized to (128*128). These were then flattened to one dimension and then sent to the network.

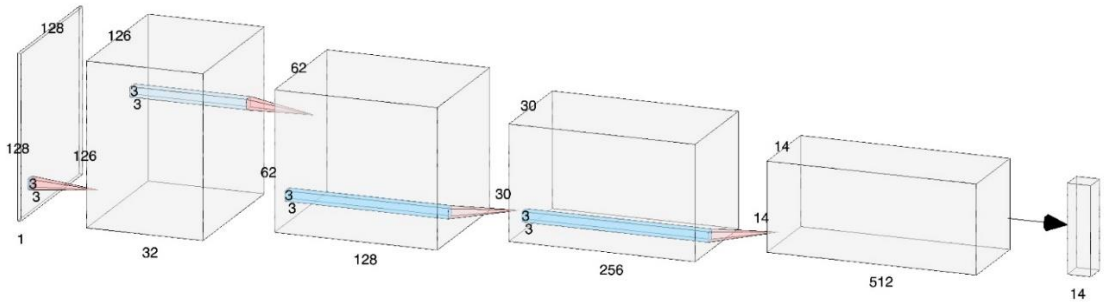


Figure 3: The proposed spiking neural network architecture for handwritten digit classification

As model parameters are not known and are mostly experimental, we experimented a lot in terms of the architecture, the learning parameters, batch sizes, epochs, etc. We also went on to experiment with various optimizers like Adam and RMSProp. We used a kernel size of 3x3 as most useful features in an image are usually local, so a smaller kernel size would mean picking up fewer pixels at a time and convolving it. We increased the number of filters for better feature extraction with every next convolutional layer. Given the input size of the image was relatively high, applying a smaller stride was a great idea as it picks up the obscured details as well in lower levels of network architecture.

We then send these to the NengoDL simulator. It makes easier to apply and explore spiking activities using special activation functions and neuron coding types, setting firing rates and applying synaptic smoothing for obtaining better plot curves using Nengo. The images are encoded by nengo in the backend in the following fashion.

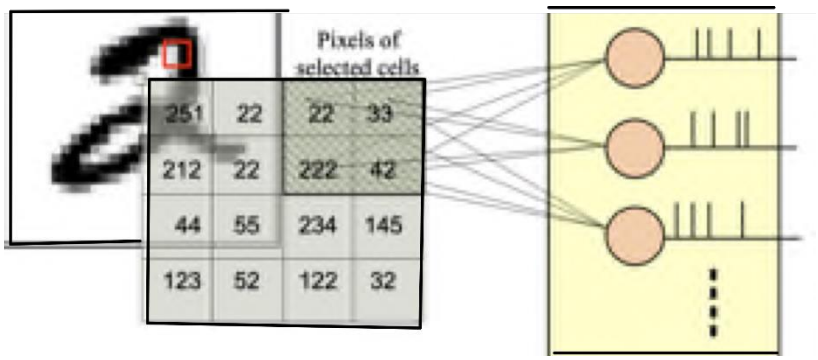


Figure 4: Rate-Based encoding of image pixels for propagation through Convolutional SNN

4. Results and Discussion

In this section we'll take a look at the results that we were able to achieve in this project. Following our aim of developing a handwritten mathematical equation solver, we trained our model to classify the different classes of the extended-MNIST (digits + mathematical symbols) dataset.

Firstly, we'll look at accuracy graphs per timesteps for the images in all the classes. This graph manages to give an idea of how the accuracy to predict different classes increases with timesteps. Since all the classes in the dataset have been converted to a number, we have provided a dictionary of key value pairs to help the readers with the graphs.

```
Dictionary = {'0': 0, '1': 1, '2': 2, '3': 3, '4': 4, '5': 5, '6': 6, '7': 7,
              '8': 8, '9': 9, '10': '+', '11': '-', '12': '*', '13': "/"}
✓ 0.2s
```

Figure 5: True Labels as key-value pairs

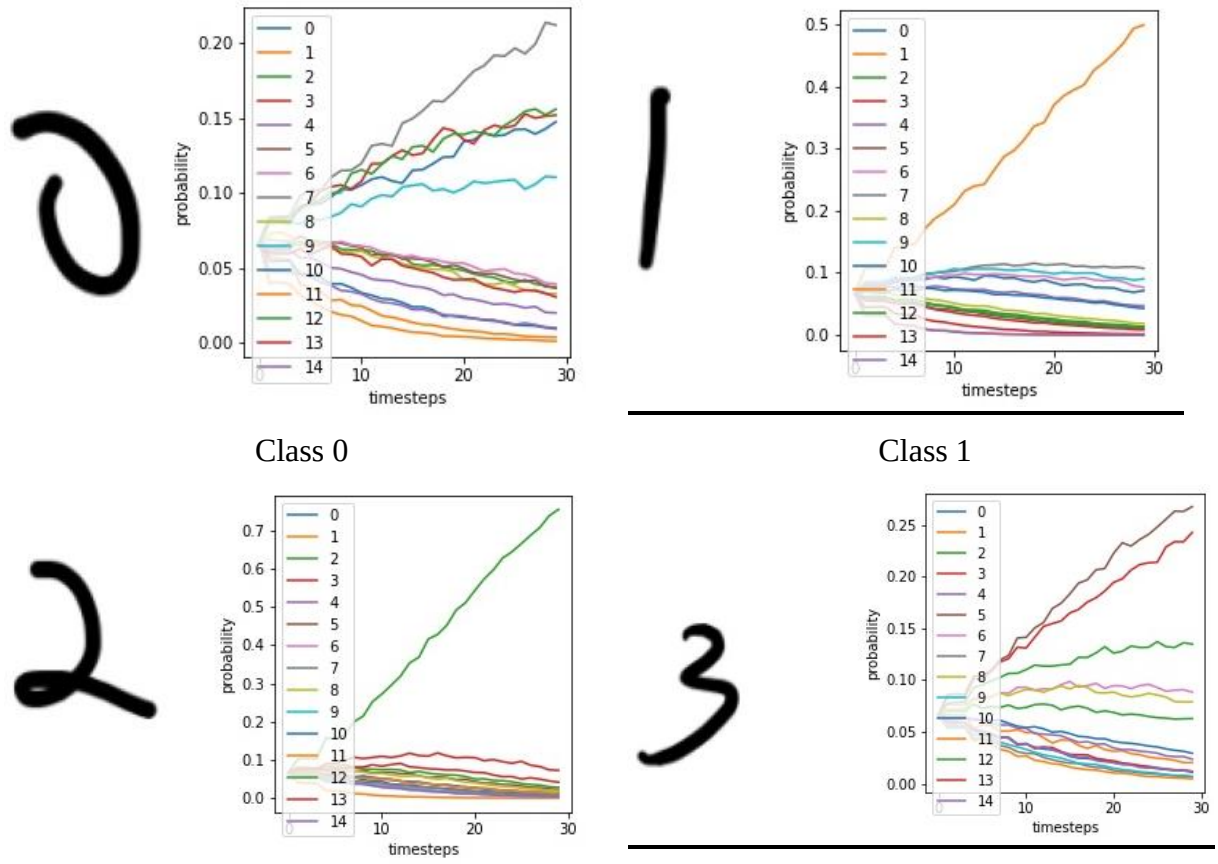
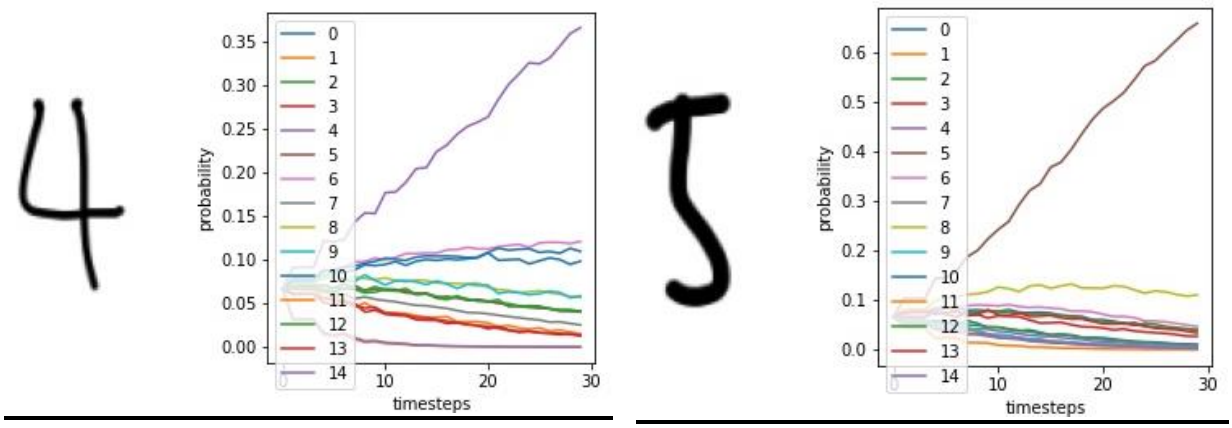
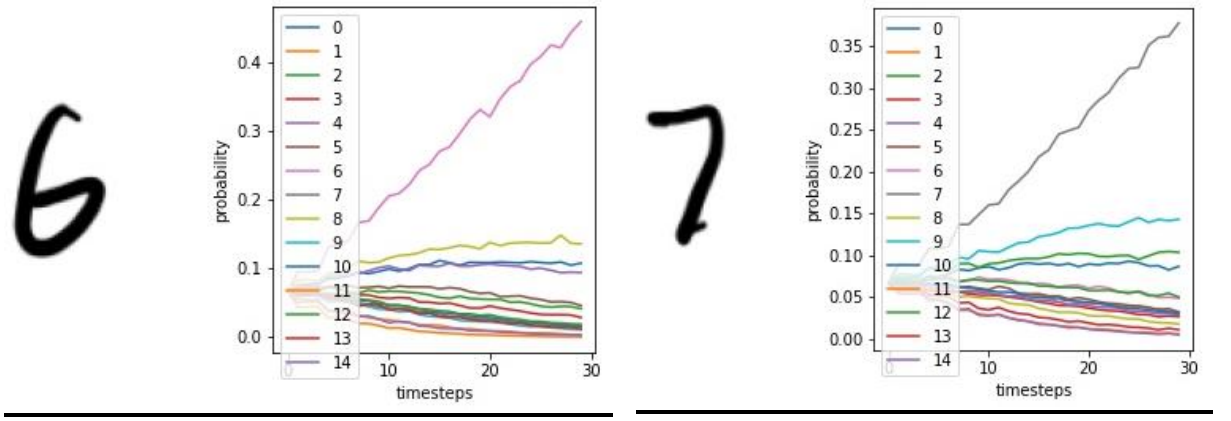


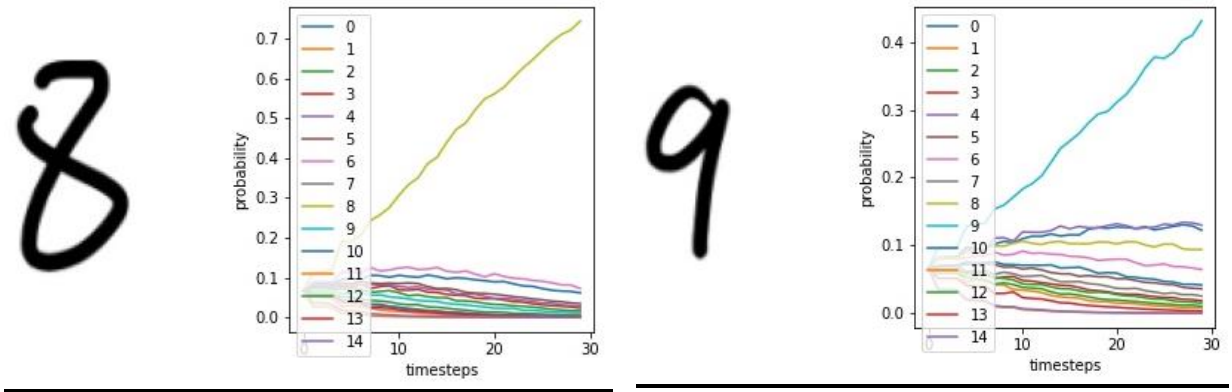
Figure 6: Probabilities as a function of timestep for different image classes



Class 4 Class 5

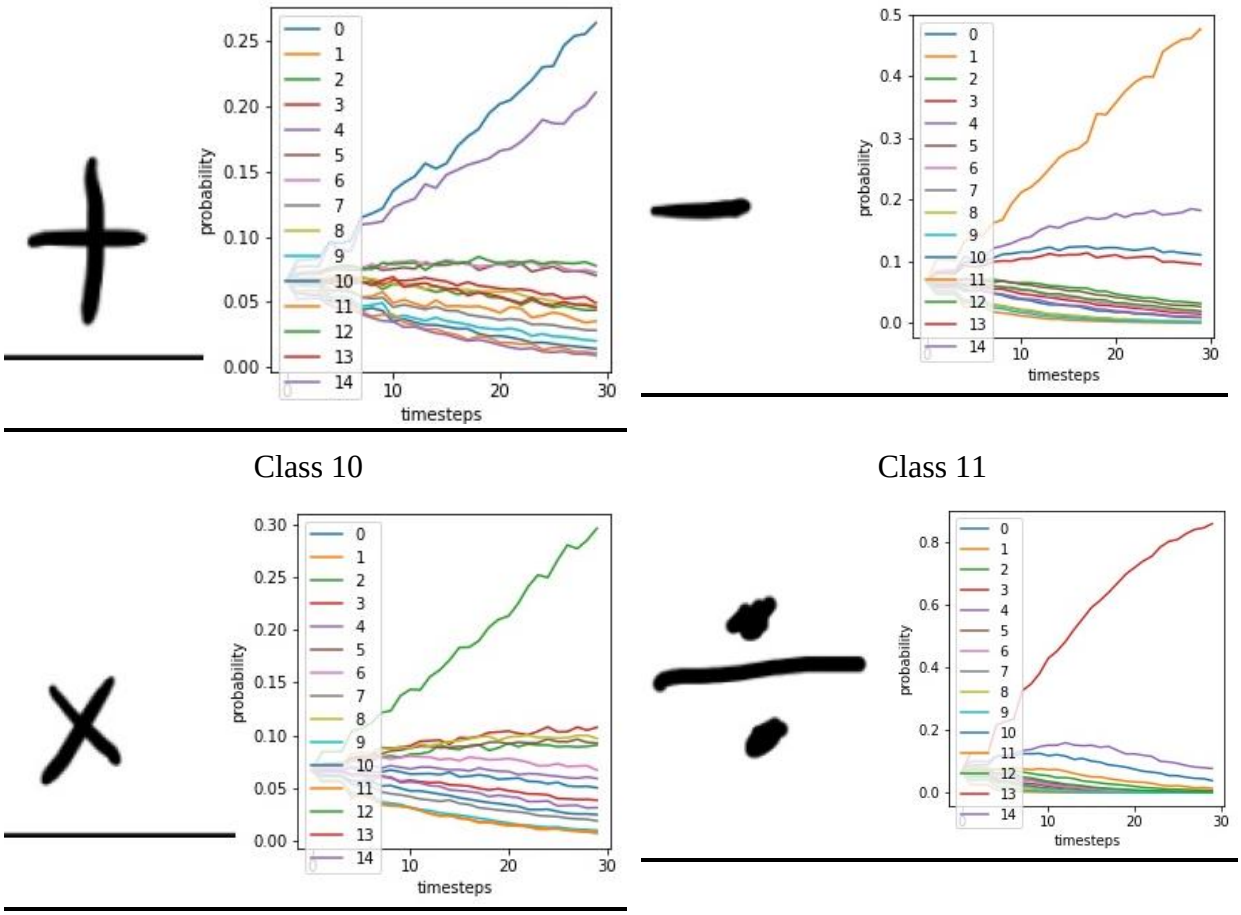


Class 6 Class 7



Class 8 Class 9

Figure 7: Probabilities as a function of timestep for different image classes



Class 10 Class 11

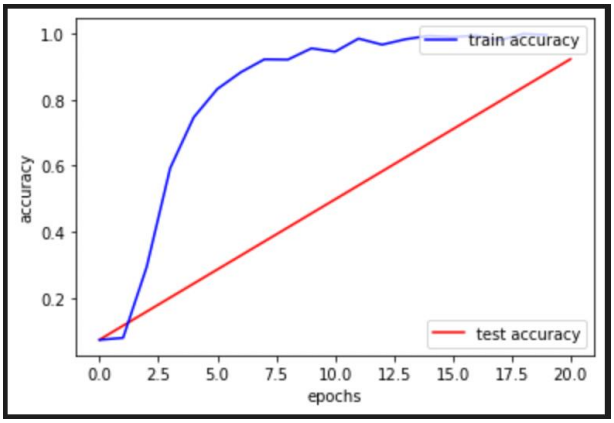
Class 12 Class 13

Figure 6: Probabilities as a function of timestep for different image classes

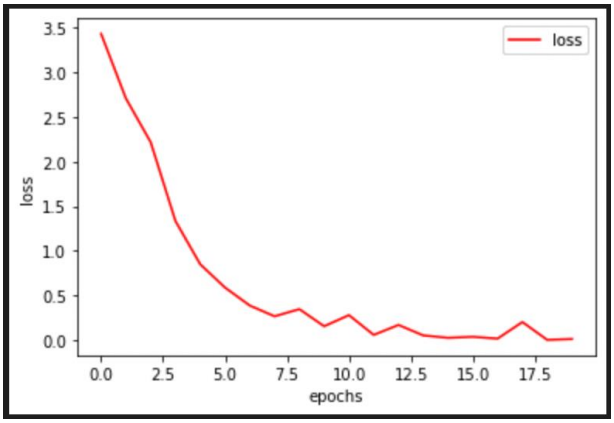
As it can be interpreted from the plots, we can comment that our model was able to easily identify categories, '1', '2', '4', '5', '6', '7', '8', '9', '-', '*', '/', as the probability to recognize these classes were much higher than probabilities to misclassify these classes from the starting timesteps itself. For categories, '0', '3', '+', the model had some problem due their resemblance with other categories (like it was confusing '0' with '2', similarly '3' with '5' and '+' with '/') but still over the period of timesteps we can see that our model was able to overcome this resemblance and give us the desired results.

Next, we'll take a look at the loss and accuracy plots. As Nengo DL doesn't provide a framework to print or store test accuracy for every epoch, we were only able to get the test accuracy before the training process and after the training process is complete. For training accuracy and loss however, we were able to get their values for every epochs which gives us a chance to analyze them more closely.

Experiment 1: (30 Timesteps, 20 Epochs)



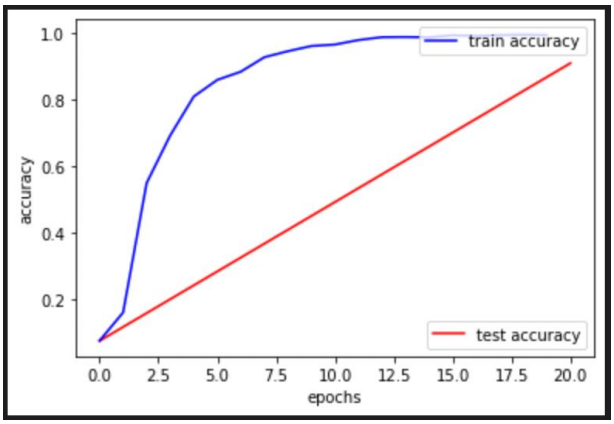
Accuracy Graph



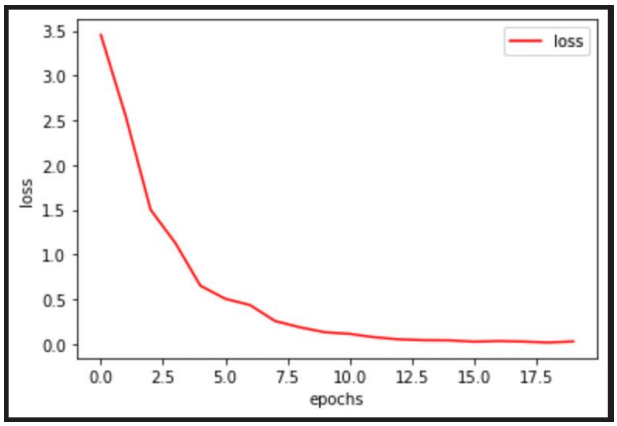
Loss Graph

Training Accuracy: 99.95% **Testing Accuracy:** 92.28% **Loss:** 0.0156%

Experiment 2: (10 Timesteps, 20 Epochs)



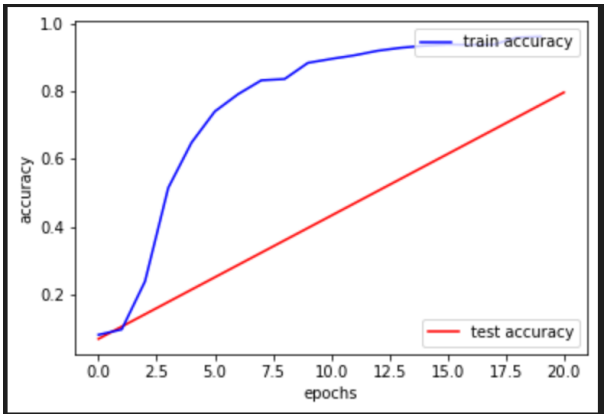
Accuracy Graph



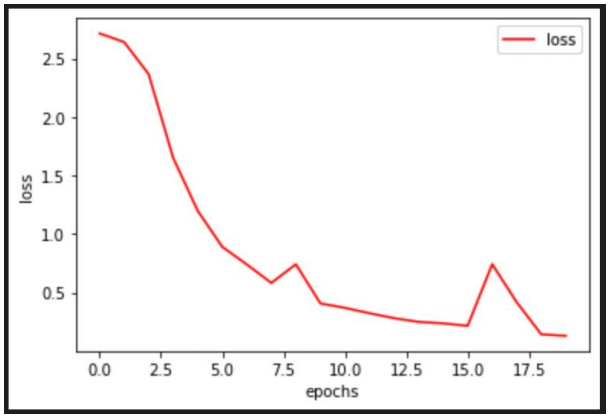
Loss Graph

Training Accuracy: 99.44% **Testing Accuracy:** 91.00% **Loss:** 0.0266%

Experiment 3: (20 Timesteps, and architecture 8_16_32_64_128)



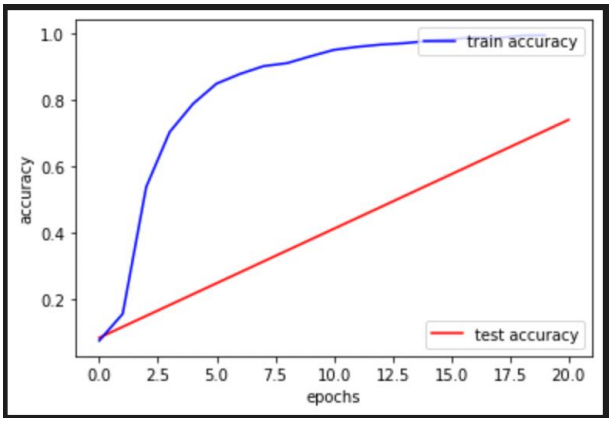
Accuracy Graph



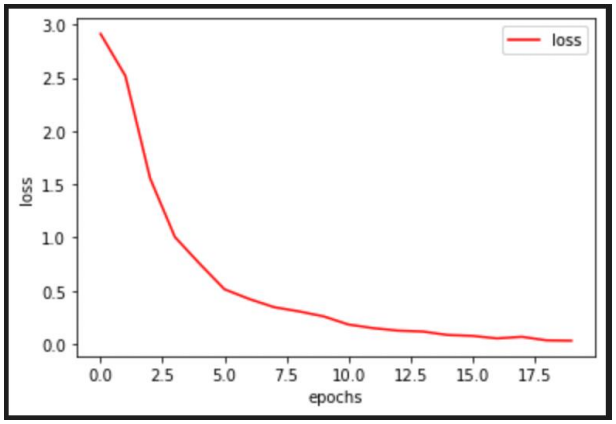
Loss Graph

Training Accuracy: 96.31% **Testing Accuracy:** 79.75% **Loss:** 0.1306

Experiment 4: (20 Timesteps, and architecture 16_32_64_128_256)



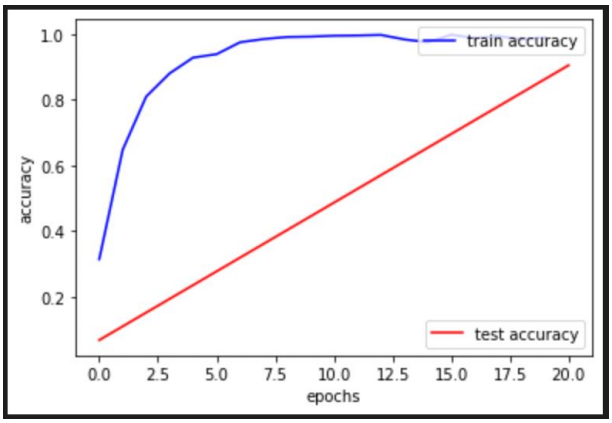
Accuracy Graph



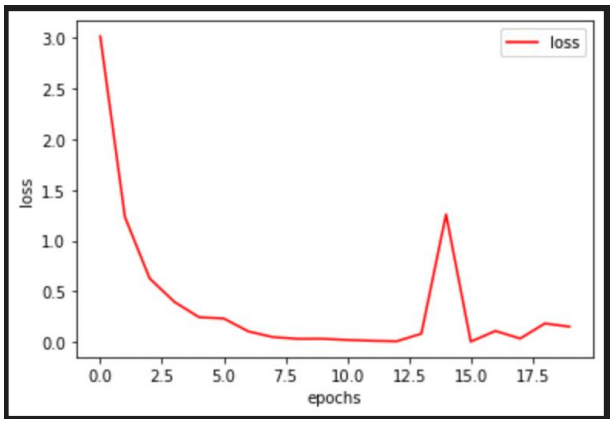
Loss Graph

Training Accuracy: 99.28% **Testing Accuracy:** 73.87% **Loss:** 0.0306

Experiment 5: (30 Timesteps, and architecture 32_64_128_256, kernel_size = 5)



Accuracy Graph



Loss Graph

Training Accuracy: 98.98% **Testing Accuracy:** 90.50% **Loss:** 0.1525

The accuracy graph consists of both training as well as testing accuracy. As we can see here that both the accuracies are steadily increasing and reaching close to 100 for most of the experiments, meaning that our model did not overfit at all and gave good results. Similarly, we can see that the loss is steadily decreasing and reaching close to 0, which again means that our model performed well.

Best accuracies and loss were given by experiment 1 and are:

Training Accuracy: 99.95%
Testing Accuracy: 92.28%
Loss: 0.0156

5. Conclusions and Future Scope

Handwritten digit equation classification was a challenging task which involved Image segmentation, preprocessing, and image classification after that. Our ML model was developed using Spike Neural Net on Kaggle dataset consisting of handwritten digits and symbols. We used Nengo DL library in Python to implement SNN which integrates TensorFlow into Nengo. Output was calculated with 14 units, where 0 – 9 represented digits and 10 -13 as symbols of "+, -, *, %" respectively.

We used 20 epochs to test the model for the dataset and we achieved 92 percent accuracy. Accuracy increased from 7.37 initially to 92. Each epoch had 30 timesteps and we plotted probability against timestep for correct classification. We got results for both digits and symbols. We found that probability of classification of image to their correct label increased with each timestep rate and after 30 timesteps probability almost became 1 from 0.1 initially.

Main aim of our project was to develop a handwritten mathematical equation solver using Spiking Neural Networks. With our model's higher accuracy, we were able to classify digits and symbol correctly, which is stepping stone for mathematical equation solving. Broadly speaking this project

can be divided into three main components, first is the image processing part, second is building a spiking neural network for classification purposes and third is the equation solving part. The work that we did in this paper so far corresponds to the second component, i.e., building the SNN. Our future work can correspond to the first component, i.e., the image processing part. We can also classify advanced level mathematical symbols like Log, Sin, Summation, Integration etc.

Image processing is an essential part of our algorithm. We already have a model ready to classify digits and mathematical symbols, but the big question is that how our model will get those images to classify, this is where the image processing part comes in. Our input images will consist of a complete mathematical equation, we can't simply give the whole image as input to the model as it is only trained to recognize one class per image. So, to tackle that we will apply image processing to make bounding boxes around all the different components of our image. Then, we can simply crop these bounding boxes to get different images containing single components, resize these images to the input size that our model has been trained on and convert these images into greyscale. Once this is done, we can have a loop and send these images one by one in our model for prediction. Once all the images have been predicted to their most likely category, this is where our third component comes in, i.e., the mathematical equation solving part. We can include an API to enter the equation from our code and get the result back to display for the users.

Despite such high accuracy, biggest limitation was hardware consumption, and our program would crash because memory resources would exhaust. As we know SNN, which is based on neuromorphic computing, requires heavy hardware support. that is drawback for running various algorithms. Despite being such time efficient, its dependency on hardware should be reduced, which will make its implementation for various machine learning project easier.

Acknowledgments

"These guidelines have grown out of an outline prepared by Prof. Konstantinos Michmizos for 425/525. The authors gratefully acknowledge helpful discussions with TA Neelesh Kumar of the Department of Computer Science.

References

[1] Jason K. Eshraghian et al. Training Spiking Neural Networks Using Lessons From Deep Learning. arxiv:2109.12894, 2022.

[2] Friedemann Zenke and Tim P Vogels. The remarkable robustness of surrogate gradient learning for instilling complex function in spiking neural networks. Neural Computation, 33(4):899-925, 2021.

[3] Chankyu Lee et al. Enabling Spike-Based Backpropagation for Training Deep Neural Network Architectures. arxiv:1903.06379v4, 2020.

[4] Xai Nano. HandWritten Maths Symbols, https://www.kaggle.com/datasets/xainano/handwrittenmathsymbols

[5] Lee Jun Haeng et al. Training Deep Spiking Neural Networks Using Backpropagation https://www.frontiersin.org/articles/10.3389/fnins.2016.00508/full

Appendix

```
#Importing libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
from skimage.io import imread, imshow
import os
import cv2
from urllib.request import urlretrieve
import nengo
import nengo_dl
import tensorflow as tf
from PIL import Image
from resizeimage import resizeimage

#for kernel optimization
os.environ['TF_GPU_ALLOCATOR']='cuda_malloc_async'
os.environ['TF_FORCE_GPU_ALLOW_GROWTH']='True'

#loading training images from the dataset

def load_images_from_folder(mod,folder):
    for filename in os.listdir(mod):
        if any([filename.endswith(x) for x in ['.jpeg', '.jpg']]):
            img = cv2.imread(os.path.join(mod, filename), 0)
            #img = cv2.imread(os.path.join(mod, filename), as_gray=True)
            res = mod.rsplit('/', 1)[1]
            #appending the training labels
            train_labels.append(int(res))
            resized = cv2.resize(img,(128,128), interpolation = cv2.INTER_
AREA)
            #appending training images
            train_images.append(np.array(resized))

#digit + operator folders
folders = ['0','1','2','3','4','5','6','7','8','9','10','11','12','13']
folder_dir="./Dataset modified/"

train_labels=[]
train_images=[]
i=0
for folder in os.listdir(folder_dir):
    mod=folder_dir+folders[i]
```

```
    print(mod)
    i+=1
    load_images_from_folder(mod,folder)
#loading testing images from the dataset

def load_images_from_folder_testing(mod,folder):
    images = []
    for filename in os.listdir(mod):
        if any([filename.endswith(x) for x in ['.jpeg', '.jpg']]):
            img = cv2.imread(os.path.join(mod, filename), 0)
            #img = cv2.imread(os.path.join(mod, filename), as_gray=True)
            res = mod.rsplit('/', 1)[1]
            #appending the testing labels
            test_labels.append(int(res))
            resized = cv2.resize(img,(128,128), interpolation = cv2.INTER_
AREA)
            #appending the training labels
            test_images.append(np.array(resized))
    return images

#digit + operator folders
folders = ['0','1','2','3','4','5','6','7','8','9','10','11','12','13']
folder_dir="./Dataset modified test/"

test_labels=[]
test_images=[]
i=0
for folder in os.listdir(folder_dir):
    mod=folder_dir+folders[i]
    print(mod)
    i+=1
    images = load_images_from_folder_testing(mod,folder)
#changing the pixel intensity value to uint8
train_labels=np.asarray(train_labels).astype('uint8')
test_labels=np.asarray(test_labels).astype('uint8')
train_images = np.array(train_images)
import numpy as np
test_images = np.array(test_images)


print("train images before flattening",train_images.shape)
print("test images before flattening",test_images.shape)

# flatten images
train_images = train_images.reshape((train_images.shape[0], -1))
```

```
test_images = test_images.reshape((test_images.shape[0], -1))

print("train images",train_images.shape)
print("train labels",train_labels.shape)

print("test images",test_images.shape)
print("test labels",test_labels.shape)

#MODEL
with nengo.Network(seed=0) as net:
    # set some default parameters for the neurons that will make
    # the training progress more smoothly
    net.config[nengo.Ensemble].max_rates = nengo.dists.Choice([100])
    net.config[nengo.Ensemble].intercepts = nengo.dists.Choice([0])
    net.config[nengo.Connection].synapse = None
    neuron_type = nengo.LIF(amplitude=0.01)

    # this is an optimization to improve the training speed,
    # since we won't require stateful behaviour in this example
    nengo_dl.configure_settings(stateful=False)

    # the input node that will be used to feed in input images
    inp = nengo.Node(np.zeros(128 * 128))

    # add the first convolutional layer
    x = nengo_dl.Layer(tf.keras.layers.Conv2D(
        filters=32, kernel_size=3))(inp, shape_in=(128,128, 1))
    x = nengo_dl.Layer(neuron_type)(x)

    # add the second convolutional layer
    x = nengo_dl.Layer(tf.keras.layers.Conv2D(
        filters=64, strides=2, kernel_size=3))(x, shape_in=(126, 126, 32))
    x = nengo_dl.Layer(neuron_type)(x)

    # add the third convolutional layer
    x = nengo_dl.Layer(tf.keras.layers.Conv2D(
        filters=128, strides=2, kernel_size=3))(x, shape_in=(62,62, 64))
    x = nengo_dl.Layer(neuron_type)(x)

    # add the fourth convolutional layer
    x = nengo_dl.Layer(tf.keras.layers.Conv2D(
        filters=256, strides=2, kernel_size=3))(x, shape_in=(30,30, 128))
    x = nengo_dl.Layer(neuron_type)(x)
```

```
    # add the fifth convolutional layer
    x = nengo_dl.Layer(tf.keras.layers.Conv2D(
        filters=512, strides=2, kernel_size=3))(x, shape_in=(14,14, 256))
    x = nengo_dl.Layer(neuron_type)(x)

    # linear readout
    out = nengo_dl.Layer(tf.keras.layers.Dense(units=14))(x)

    # we'll create two different output probes, one with a filter
    # (for when we're simulating the network over time and
    # accumulating spikes), and one without (for when we're
    # training the network using a rate-based approximation)
    # probes essentially collect data over time
    out_p = nengo.Probe(out, label="out_p")
    out_p_filt = nengo.Probe(out, synapse=0.1, label="out_p_filt")

# create the simulator that will run the CSNN
minibatch_size = 100
sim = nengo_dl.Simulator(net, minibatch_size=minibatch_size, device="/gpu:
0")

# add single timestep to training data
train_images = train_images[:, None, :]
train_labels = train_labels[:, None, None]

# when testing our network with spiking neurons we will need to run it
# over time, so we repeat the input/target data for a number of
# timesteps.


n_steps = 30
test_images = np.tile(test_images[:, None, :], (1, n_steps, 1))
test_labels = np.tile(test_labels[:, None, None], (1, n_steps, 1))

# we create a custom accuracy classification problem because we are only c
oncerned with the accuracy of the actual SNN
def classification_accuracy(y_true, y_pred):
    return tf.metrics.sparse_categorical_accuracy(
        y_true[:, -1], y_pred[:, -1])

# note that we use `out_p_filt` when testing (to reduce the spike noise)
sim.compile(loss={out_p_filt: classification_accuracy})
print("accuracy before training:",
      sim.evaluate(test_images, {out_p_filt: test_labels}, verbose=0)["los
s"])
```

```
# train the rate-based approximation of the CSNN
do_training = True
if do_training:
    # run training
    sim.compile(
        optimizer=tf.optimizers.RMSprop(0.001),
        #optimizer=tf.optimizers.Adam(0.001, 0.9, 0.999),
        loss={out_p: tf.losses.SparseCategoricalCrossentropy(from_logits=T
rue)},
        metrics={out_p: classification_accuracy}
    )
    sim.fit(train_images, {out_p: train_labels}, epochs=20, verbose=1)

    # save the parameters to file
    sim.save_params("./mnist_params")
else:
    # download pretrained weights
    # load parameters
    sim.load_params("./mnist_params")

#print the accuracy of the CSNN
sim.compile(loss={out_p_filt: classification_accuracy})
print("accuracy after training:",
      sim.evaluate(test_images, {out_p_filt: test_labels}, verbose=0)["los
s"])

# code to plot example outputs from the CSNN
data = sim.predict(test_images[:854])
sim.close()


for i in range(5):
    plt.figure(figsize=(8, 4))
    plt.subplot(1, 2, 1)
    plt.imshow(test_images[i, 0].reshape((128, 128)), cmap="gray")
    plt.axis("off")

    plt.subplot(1, 2, 2)
    plt.plot(tf.nn.softmax(data[out_p_filt][i]))
    plt.legend([str(i) for i in range(15)], loc="upper left")
    plt.xlabel("timesteps")
    plt.ylabel("probability")
    plt.tight_layout()
```