CSE327 Final Project Report

Dowon Kim
114001605

1. Project Overview

1.1 Project Vision

This project uses a machine learning model to look at images of scenery and analyze the location of where it was taken. The model is based on the Spiking Neural Network (SNN) architecture to achieve efficient training while maintaining high accuracy. The objective of this project is to identify the country of the scenery images as accurately as possible.

1.2 Motivations

The inspiration for this project comes from an online web app called **Geoguessr**. It is a fun, interactive game that lets users guess a randomly selected location in the world. Surprisingly, people around the world have gotten unbelievably good at guessing these locations using various clues such as the orientation of the sun, types of grass, roads, mountains, buildings, and many more. I've always been intrigued by this concept and wanted to make a computer vision project based on analyzing scenery.

1.3 References

These are the references I've used for this project. For the model that I've decided to implement (Spikformer), I will describe it in more detail in the ViT Model section of the report below.

- Spikformer: When Spiking Neural Network Meets Transformer (https://arxiv.org/abs/2209.15425)
- Spike-driven Transformer (https://arxiv.org/abs/2307.01694)
- PIGEON: Predicting Image Geolocations (https://arxiv.org/pdf/2307.05845)
- Geoguessr Website: https://www.geoguessr.com/

1.4 GitHub Repository

The exact files I've used and modified to carry out the experiments can be found in this GitHub repository:

Link: https://github.com/dowonkim-sunykr/CSE327-FinalProject

2. Dataset

2.1 Selection

From the start, I knew I wanted to collect my own dataset using a Python script with the Google Maps API. It seemed like a solid choice since it's free (with limits), easy to use, well-documented, and covers pretty much anywhere Google Street View cars have been.

The first thing I did was set up an API key through Google Cloud, which gave me access to all their different services. For this project, I mainly used the Street View Static API, which lets you pull a snapshot of a location if you give it latitude and longitude coordinates.

For this project, I picked five major cities for the dataset: Seoul, Tokyo, New York, London, and Paris. For each city, I randomly generated points within about 1 km of the city center and used those to grab Street View images. In total, the dataset will contain around 50,000 images, giving me a solid amount of diverse urban scenery to train the model on.

Example of a Google Street View Image

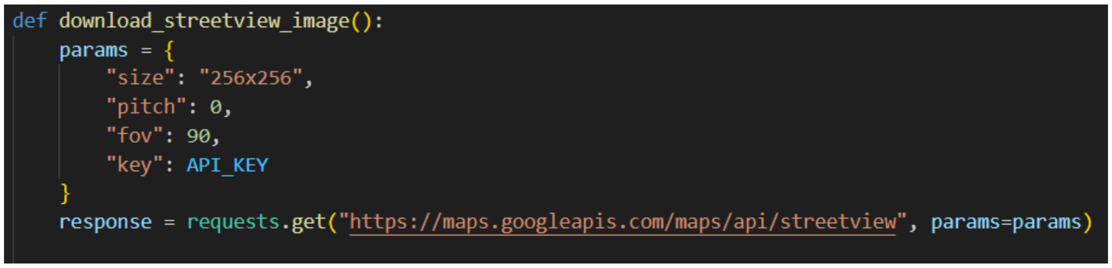
2.2 Justification

Google Street View works really well for this project because it's easy to use with a Python script and covers a lot of places around the world. The Street View Static API lets me get images just by giving it coordinates, which makes collecting a lot of data fast and straightforward. The images also have consistent quality and angles, which is helpful when training a model.

The dataset will have about 50,000 images from five major cities. That should be plenty for this kind of project, as other popular datasets such as CIFAR-10 and MNIST also have around 50,000 images, so this puts mine in a good range for training a model effectively.

2.3 Gathering

To collect the dataset, I wrote a Python script that uses the Google Maps API, specifically the Street View Static API. The first step was setting up an API key through the Google Cloud Console, which gave me access to Google's various APIs. After enabling the Street View API for my project, I was ready to start making requests.



```python
def download_streetview_image():
    params = {
        "size": "256x256",
        "pitch": 0,
        "fov": 90,
        "key": API_KEY
    }
    response = requests.get("https://maps.googleapis.com/maps/api/streetview", params=params)
```

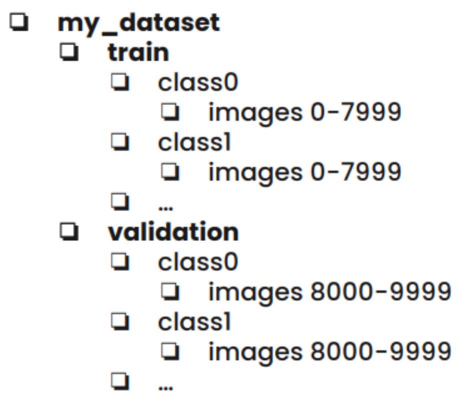
In the script, I set the exact coordinates of the five major cities: Seoul, Tokyo, New York, London, and Paris. For each city, the script requests images using coordinates around the city center. It checks if there is a valid Street View image at that location before saving it. Each image is resized to 32x32 pixels to reduce file size and speed up training, while also matching the format used in datasets like CIFAR. The script runs on multiple threads to collect data faster and makes about 20 requests per second, which stays within Google's API rate limit. Additionally, this script is included in the GitHub repository (scraper.py) but needs a valid Google Maps API key to work.

2.4 Dataset Details

These are the details of the 50,000 images in my dataset:

- Image size: 32x32
- Channels: 3
- File format: .jpg
- Train/Validation split: 80% / 20%
- Total file size: 45.0MB

The structure of the dataset follows closely to that of CIFAR10. Here is a diagram of my dataset's structure:



- **my_dataset**
  - **train**
    - class0
      - images 0-7999
    - class1
      - images 0-7999
    - ...
  - **validation**
    - class0
      - images 8000-9999
    - class1
      - images 8000-9999
    - ...

2.5 Preprocessing

As briefly mentioned in section 2.3, when downloading the streetview images, I had to resize them from 128x128 to 32x32. This is done for two important reasons. The first is that Google Maps images all have a decently-sized Google logo stamped onto the bottom-left corner of every streetview data. However, this is a set size and does not scale with the output width or height. My solution was to initially download it as a large image to make the logo relatively small, then downsample it to keep all the important information while making the logo irrelevant.

The second reason was that to use a dataset of 50,000 images, I knew that it would take up a lot of space, and more importantly a lot of computing power in total. I followed the CIFAR and MNIST datasets and decided on the final output size of 32x32, which meant that they were only around 950 bytes per image.

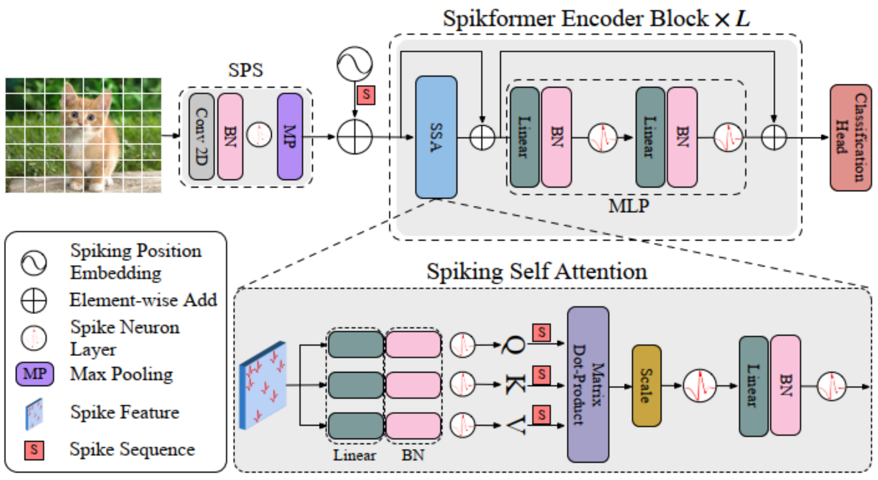
Example of a preprocessed image in my dataset (32x32)

3. Vision Transformer Model

3.1 Analysis of Spikformer

For my project, I've decided to use the **Spikformer** model. As implied by the name, it is a combination of a spiking neuron network and a transformer. It was conceptualized in 2023 and has been considered to be the foundational work for the SNN+transformer models since then.

For the inner workings of the model, I've read through and analyzed the research paper "Spikformer: When Spiking Neural Network Meets Transformer" to better understand each and every part of it, and I will now explain the most important parts in this section.



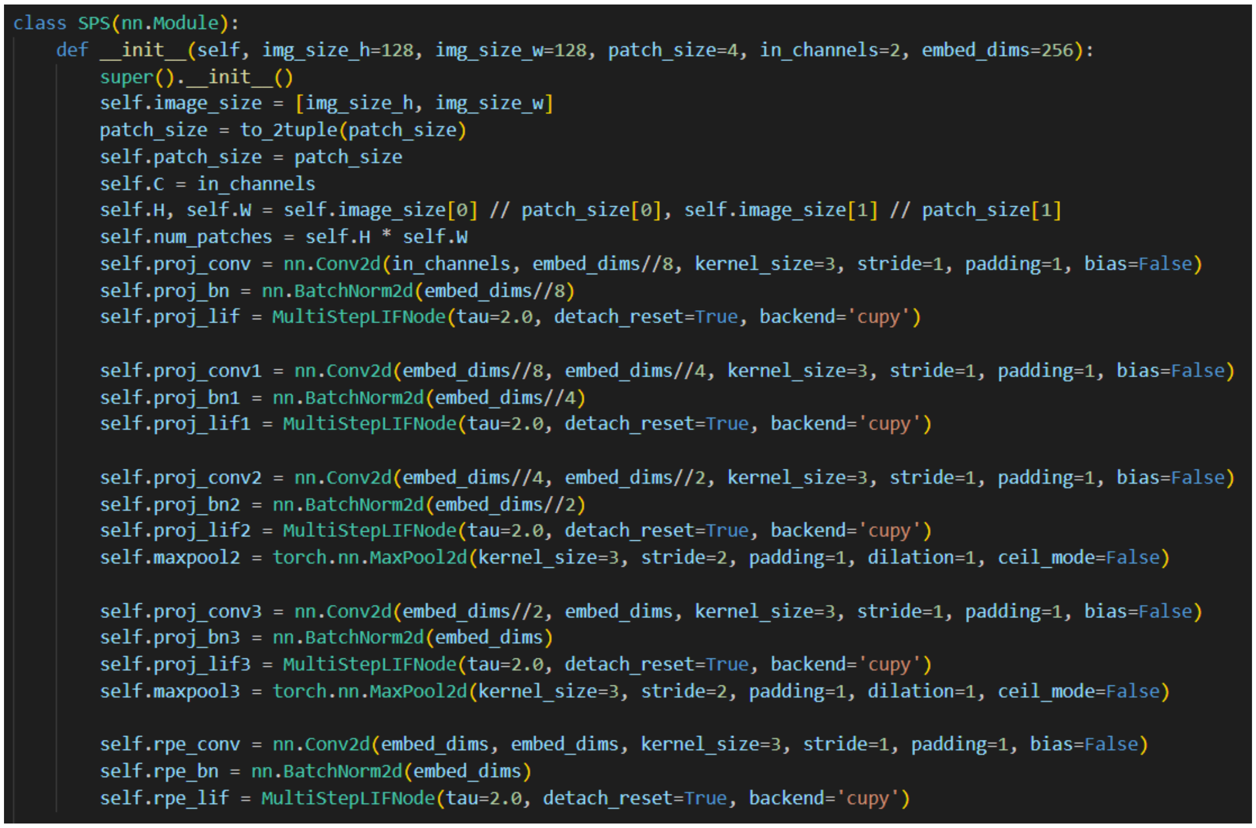
As seen in the diagram above, the Spikformer is composed of 3 main blocks: **Spike Patch Splitter** (SPS), Spikformer Encoder Block (and the **SSA** block), and the classification head.

Firstly, the main role of the SPS block is to split the input image into several patches and output a **spike-form** matrix. Spike-form data have values that are only 0s or 1s (1s corresponding to a neuron spiking) and make matrix operations significantly fast and efficient compared to classical floating point operations.

The main innovation of Spikformer is the **Spiking Self Attention** (SSA). As seen in the diagram, it takes in a query, key, and value in the form of spikes (0s and 1s) and then performs a matrix dot-product. This is significant because vanilla self-attention mechanisms notably use floating point numbers in their matrix multiplication, then use the softmax function in the final stage. This can be problematic for energy concerns since softmax uses exponential calculations in order to normalize values. The Spikformer, on the other hand, turns these floating point operations into an AND gate operation. This makes the entire model exponentially efficient, reducing energy consumption and accelerating computations.

Then finally for the classification head, it predicts the final class based on the highest value of all of its predictions.

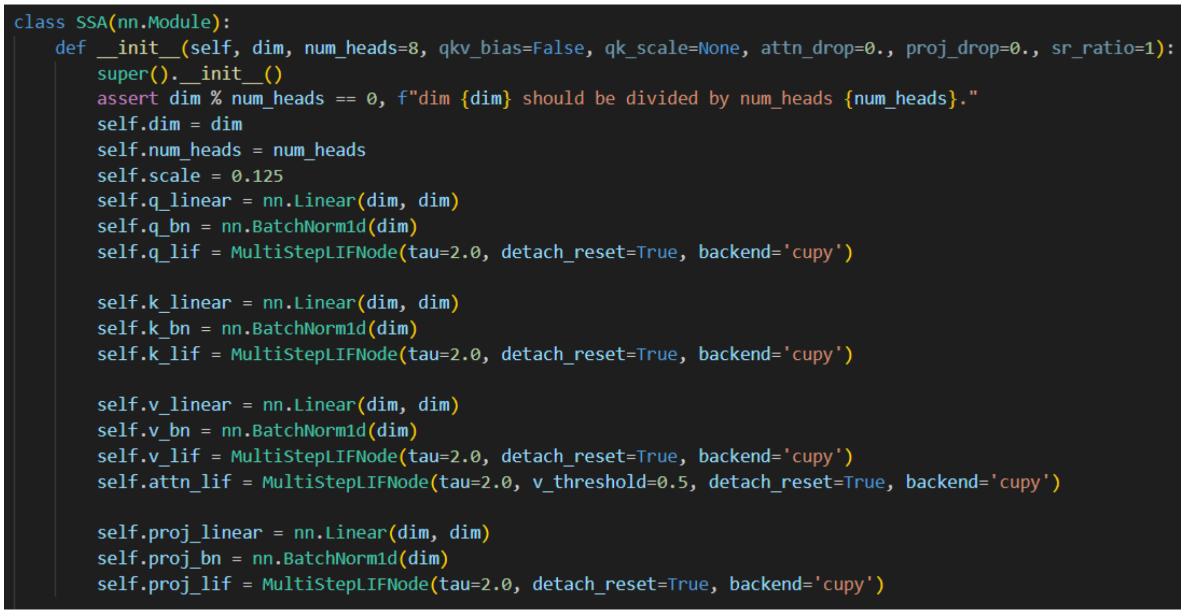
Now, I will analyze the Spikformer's actual implementation written in Python. Below is the constructor of the first part of Spikformer, the spike patch splitter (SPS):



```python
class SPS(nn.Module):
    def __init__(self, img_size_h=128, img_size_w=128, patch_size=4, in_channels=2, embed_dims=256):
        super().__init__()
        self.image_size = [img_size_h, img_size_w]
        patch_size = to_2tuple(patch_size)
        self.patch_size = patch_size
        self.C = in_channels
        self.H, self.W = self.image_size[0] // patch_size[0], self.image_size[1] // patch_size[1]
        self.num_patches = self.H * self.W
        self.proj_conv = nn.Conv2d(in_channels, embed_dims//8, kernel_size=3, stride=1, padding=1, bias=False)
        self.proj_bn = nn.BatchNorm2d(embed_dims//8)
        self.proj_lif = MultiStepLIFNode(tau=2.0, detach_reset=True, backend='cupy')

        self.proj_conv1 = nn.Conv2d(embed_dims//8, embed_dims//4, kernel_size=3, stride=1, padding=1, bias=False)
        self.proj_bn1 = nn.BatchNorm2d(embed_dims//4)
        self.proj_lif1 = MultiStepLIFNode(tau=2.0, detach_reset=True, backend='cupy')

        self.proj_conv2 = nn.Conv2d(embed_dims//4, embed_dims//2, kernel_size=3, stride=1, padding=1, bias=False)
        self.proj_bn2 = nn.BatchNorm2d(embed_dims//2)
        self.proj_lif2 = MultiStepLIFNode(tau=2.0, detach_reset=True, backend='cupy')
        self.maxpool2 = torch.nn.MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)

        self.proj_conv3 = nn.Conv2d(embed_dims//2, embed_dims, kernel_size=3, stride=1, padding=1, bias=False)
        self.proj_bn3 = nn.BatchNorm2d(embed_dims)
        self.proj_lif3 = MultiStepLIFNode(tau=2.0, detach_reset=True, backend='cupy')
        self.maxpool3 = torch.nn.MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)

        self.rpe_conv = nn.Conv2d(embed_dims, embed_dims, kernel_size=3, stride=1, padding=1, bias=False)
        self.rpe_bn = nn.BatchNorm2d(embed_dims)
        self.rpe_lif = MultiStepLIFNode(tau=2.0, detach_reset=True, backend='cupy')
```

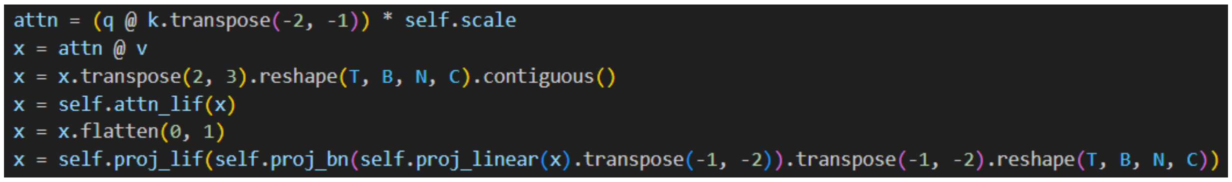
As you can see, it utilizes PyTorch to implement the various layers. More interestingly, you can see instances of MultiStepLIFNode from SpikingJelly – these are the building blocks for spiking neural networks. To simplify, it acts as a human neuron with voltage and current, and can 'spike' a signal once the input goes over the threshold. In the SPS class, the very last layer is the LIF node. This means that the forward pass will output a spike in the end.

Next is the most important part of the Spikformer, the spiking self-attention (SSA):



```
class SSA(nn.Module):
    def __init__(self, dim, num_heads=8, qkv_bias=False, qk_scale=None, attn_drop=0., proj_drop=0., sr_ratio=1):
        super().__init__()
        assert dim % num_heads == 0, f"dim {dim} should be divided by num_heads {num_heads}."
        self.dim = dim
        self.num_heads = num_heads
        self.scale = 0.125
        self.q_linear = nn.Linear(dim, dim)
        self.q_bn = nn.BatchNorm1d(dim)
        self.q_lif = MultiStepLIFNode(tau=2.0, detach_reset=True, backend='cupy')

        self.k_linear = nn.Linear(dim, dim)
        self.k_bn = nn.BatchNorm1d(dim)
        self.k_lif = MultiStepLIFNode(tau=2.0, detach_reset=True, backend='cupy')

        self.v_linear = nn.Linear(dim, dim)
        self.v_bn = nn.BatchNorm1d(dim)
        self.v_lif = MultiStepLIFNode(tau=2.0, detach_reset=True, backend='cupy')
        self.attn_lif = MultiStepLIFNode(tau=2.0, v_threshold=0.5, detach_reset=True, backend='cupy')

        self.proj_linear = nn.Linear(dim, dim)
        self.proj_bn = nn.BatchNorm1d(dim)
        self.proj_lif = MultiStepLIFNode(tau=2.0, detach_reset=True, backend='cupy')
```

Similarly to the SPS, it uses LIF nodes to generate spikes. However, the difference is that it has independent layers q(query), k(key), and v(value) that get "combined" to recreate something similar to vanilla self-attention.



```
attn = (q @ k.transpose(-2, -1)) * self.scale
x = attn @ v
x = x.transpose(2, 3).reshape(T, B, N, C).contiguous()
x = self.attn_lif(x)
x = x.flatten(0, 1)
x = self.proj_lif(self.proj_bn(self.proj_linear(x).transpose(-1, -2)).transpose(-1, -2).reshape(T, B, N, C))
```

The result of the forward pass is a spike generated by the LIF layer after the matrix dot-product and the projection layer as seen above.

3.2 Implementation

For my implementation of the Spikformer model, I had to clone the public repository of the model from GitHub (https://github.com/ZK-Zhou/spikformer) first. After getting the code, all I had to do was install the required python libraries with the correct version numbers. These include:

- timm (0.5.4)
- cupy (10.3.1)
- pytorch (1.10.0+cu111)
- spikingjelly (0.0.0.0.12)
- pyyaml (any version)

After doing so, I just had to run the train.py Python file located in the /cifar10 subdirectory.

3.3 Changes to Model

Since the default configurations of the Spikformer model are set for training the CIFAR10 dataset, I had to make some modifications to fit my dataset well. I noticed that it had a configuration file in .yml format, so I made my own configuration file to use with streetview images. This file is available to view in the GitHub repository. I also changed the logging process for the output, so that I could get more information while running the script or debugging fatal errors. I made sure to also calculate the total time and time per epoch to gather and compare with the ResNet model later on.

4. Experiments

4.1 Training Setup & Environment

For all of the experiments conducted, I utilized Google's Colab services to gain a computational advantage over my laptop's weak GPU. I used the T4 GPU, which can provide around 4 to 5 hours of free use per session.

Additionally, as mentioned in section 3.3 above, I changed various hyperparameters with my own configuration file. These are all of the important values I've used for the Spikformer experiment:

- epochs: 300
- time_step: 4
- layer: 4
- dim: 384
- num_heads: 12
- patch_size: 4
- mlp_ratio: 4
- data_dir: ../my_dataset
- num_classes: 5
- img_size: 32
- amp: True
- batch_size: 64
- lr: 5e-4
- weight_decay: 6e-2
- cooldown_epochs: 10
- warmup_epochs: 20
- warmup_lr: 0.00001
- opt: adamw

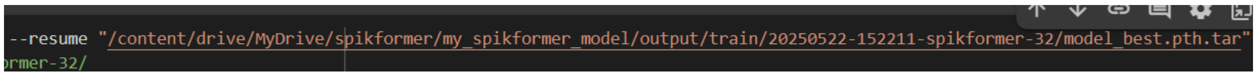
As for the experiment conducted with the comparison model, ResNet18, there was no need to do any extra setup and I was simply able to use PyTorch's default settings.

4.2 Training Procedure

The training process itself took a very long time. The average time to train and evaluate per epoch was around 3 minutes, so I expected around 15 hours in total.

I noticed the early stages of training had rapid improvements in terms of accuracy. I suspect that this is due to the adam optimizer, which uses momentum. At around 50 epochs, the improvements slowed down and I left the GPUs to do the rest of the work until all 300 epochs were finished.

On a very important note, I did not complete the training process in a single session. I knew it would take too long to do it at once, so I looked into the --resume flag in the arg parser. It allows you to carry on your model's training as if you started from that state, which is exactly what I needed. When the model finishes an epoch, if the final metric (accuracy) is within the top scoring values (in my case, top 10), then it will save the model's parameters and its state as a checkpoint using the timm library. With these checkpoints, I can resume training from the last epoch, the best-performing epoch, or any specific epoch within the top 10.



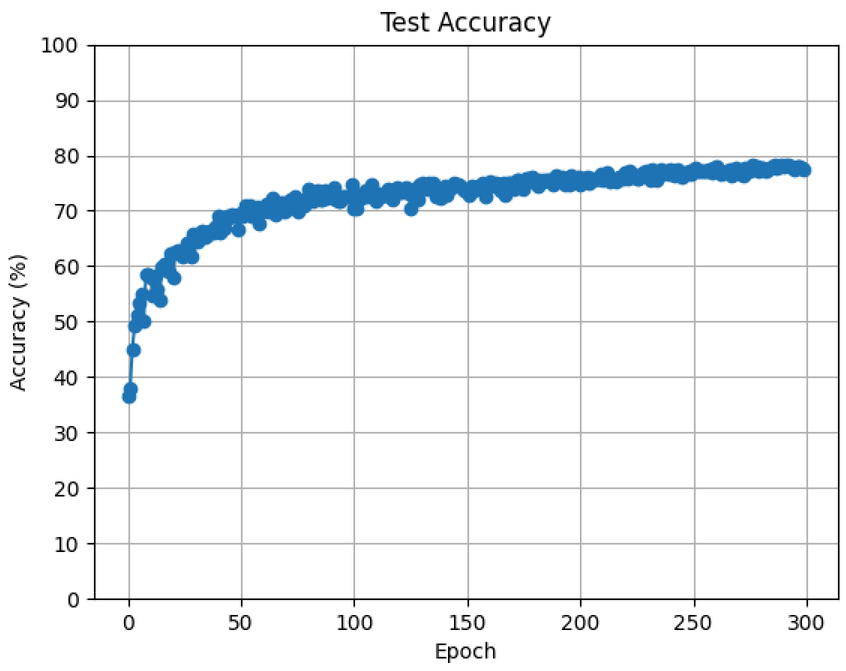
```
--resume "/content/drive/MyDrive/spikformer/my_spikformer_model/output/train/20250522-152211-spikformer-32/model_best.pth.tar"
rmer-32/
```

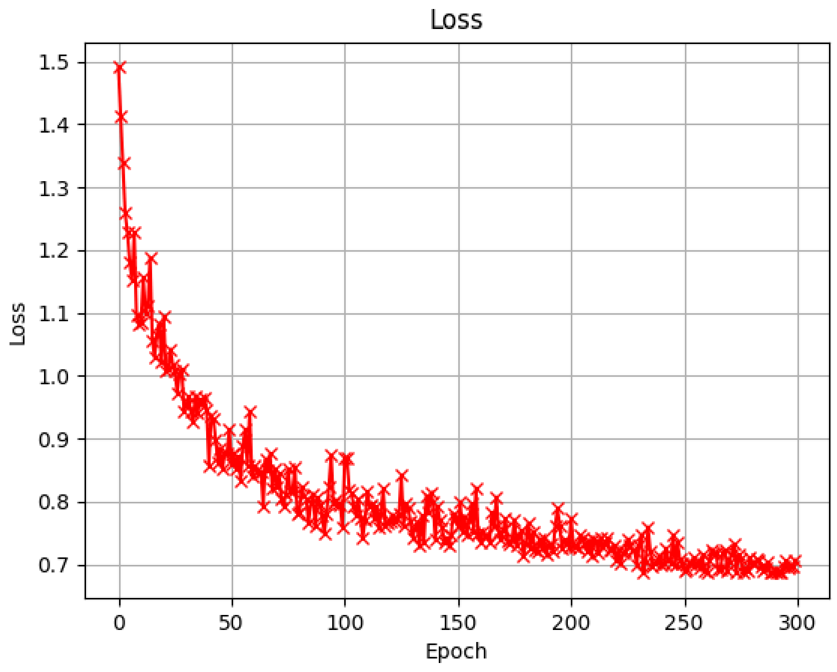
In the end, I completed training the Spikformer model in 4 separate sessions, fully utilizing the checkpoint functionality to not waste any time.

5. Results

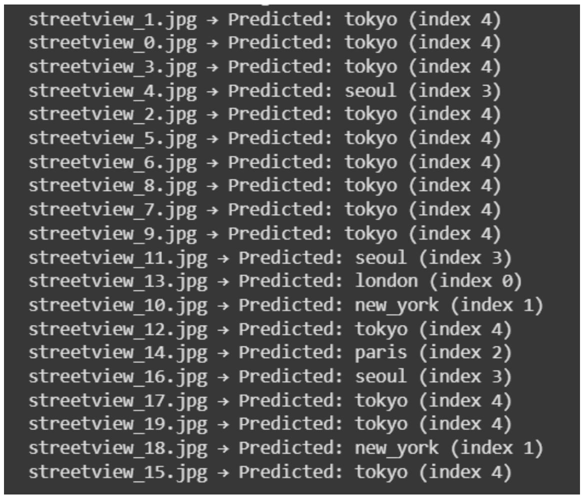
5.1 Spikformer's Performance

After 300 epochs, I managed to achieve a top 1 accuracy of **78.24%**. This is quite impressive in my opinion, considering that there are 5 classes of scenery images that look pretty similar to an untrained eye (like myself). I saved the output into a long .txt file, then parsed the results and plotted it onto a graph:





As you can observe, it slowed down massively at around the 100 epoch mark, only managing to gain around a 5% increase in accuracy until the very last epoch. For the best accuracy value, I verified this by making a simple script that checks 20 streetview images of Tokyo and makes a prediction for the 5 valid city classes. It managed to score 13/20=65%, which was good enough to believe that it was working.



```
streetview_1.jpg → Predicted: tokyo (index 4)
streetview_0.jpg → Predicted: tokyo (index 4)
streetview_3.jpg → Predicted: tokyo (index 4)
streetview_4.jpg → Predicted: seoul (index 3)
streetview_2.jpg → Predicted: tokyo (index 4)
streetview_5.jpg → Predicted: tokyo (index 4)
streetview_6.jpg → Predicted: tokyo (index 4)
streetview_8.jpg → Predicted: tokyo (index 4)
streetview_7.jpg → Predicted: tokyo (index 4)
streetview_9.jpg → Predicted: tokyo (index 4)
streetview_11.jpg → Predicted: seoul (index 3)
streetview_13.jpg → Predicted: london (index 0)
streetview_10.jpg → Predicted: new_york (index 1)
streetview_12.jpg → Predicted: tokyo (index 4)
streetview_14.jpg → Predicted: paris (index 2)
streetview_16.jpg → Predicted: seoul (index 3)
streetview_17.jpg → Predicted: tokyo (index 4)
streetview_19.jpg → Predicted: tokyo (index 4)
streetview_18.jpg → Predicted: new_york (index 1)
streetview_15.jpg → Predicted: tokyo (index 4)
```

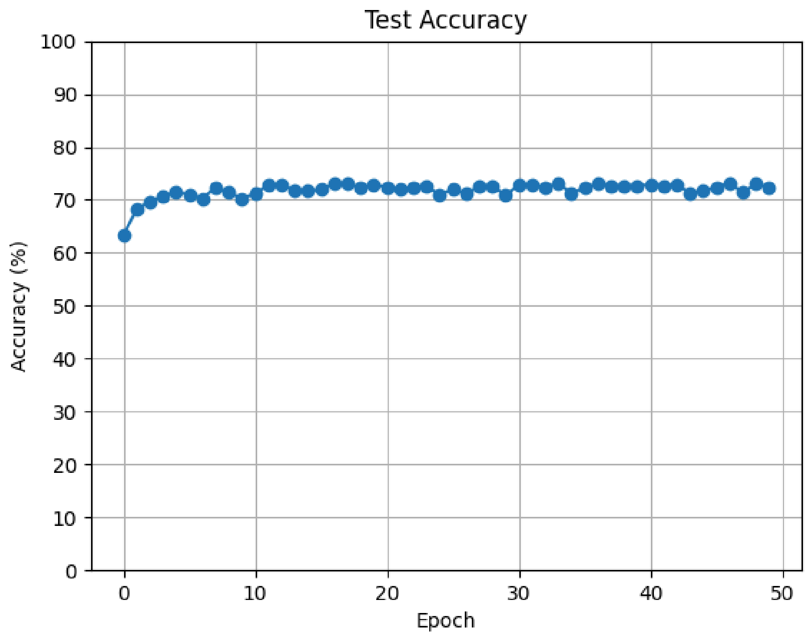
In total, the experiment took 56377.04 seconds, or **15 hours 39 minutes and 37 seconds**.

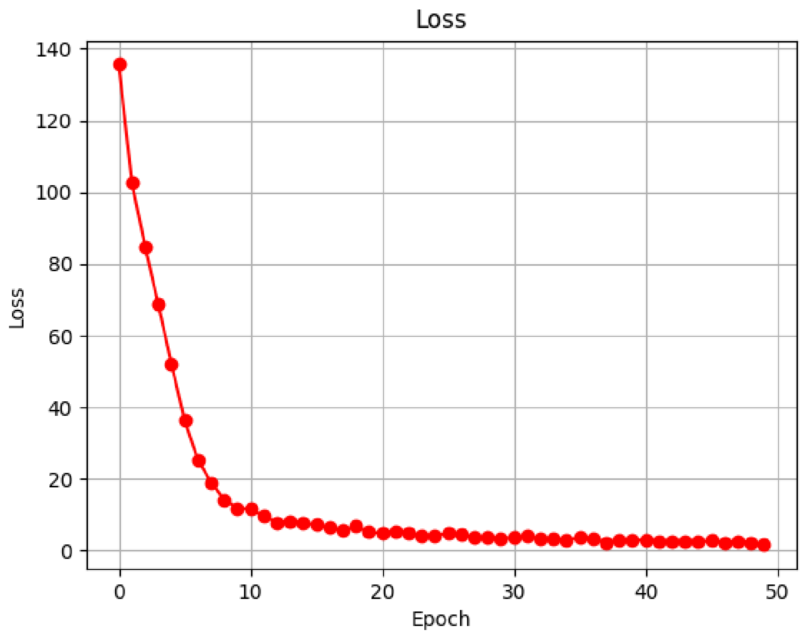
5.2 ResNet18's Performance

After concluding my experiment with the Spikformer, I used TorchVision's ResNet18 model to run an additional test for comparison. All relevant hyperparameters from the Spikformer were left the same, except for the fact that I used pre-trained weights for the ResNet. This is because I was mainly curious about the peak of the model's performance, not how quickly it learns.

```
model = models.resnet18(pretrained=True)
model.fc = nn.Linear(model.fc.in_features, num_classes)
model = model.to(device)
```

After everything was correctly set, I ran the experiment using Google Colab once again. Here are the results for ResNet18:





After around 10 epochs of training, I noticed that the performance improvements began slowing down and stabilizing. At around 50 epochs, I couldn't see any reason to continue this experiment further and decided to stop the training early to not waste any more time and prevent possible overfitting.

As you can see in the two graphs, ResNet18 managed to achieve an accuracy of **73.16%**. This is around 5% less than the highest accuracy of the Spikformer. The model also stabilized very quickly and saw almost no improvements near the end of my experiment. It is also important to mention that the total time was 1843.5 seconds, or **30 minutes and 43 seconds** to train 50 epochs. I will be comparing this time to the Spikformer in the next section.

5.3 Comparison & Analysis

In almost every aspect, the Spikformer outperformed ResNet18 by a considerable margin. Firstly, the validation accuracy was 5.08% higher for the Spikformer. In my opinion, this is a clear indication that the transformer (the SSA block) played a significant role in boosting the performance by utilizing the self-attention mechanism. While trying to visualize the internal layers of the Spikformer was outside of the scope of this project, I believe that the SSA block was able to analyze key features in the input data such as the roads, cars, buildings, trees, and more in a way that a normal CNN architecture like ResNet could not.

Additionally, the training time between the two models had the most noticeable difference by far. The Spikformer averaged 187.92 seconds per epoch, but ResNet18 averaged nearly double that time at 364.18 seconds per epoch. In my opinion, this is due to the Spikformer being an overall simpler model, but also due to a lot of operations being simple spike-form calculations using 0s and 1s whereas ResNet18 has complicated floating point MAC operations and softmax.

5.4 Me vs Spikformer

After I was done with training the models, I wanted to put it to the test to see if it was actually good. I set up a simple test to give me 25 new, random images from any of the 5 cities to try and guess to the best of my ability.

25 Images used for this minigame

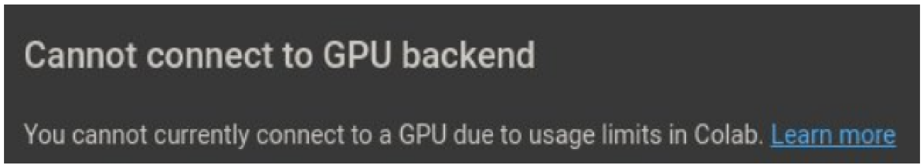
I finished labeling all 25 images and tallied my score, which ended up being **14 out of 25 (56%)**. Then, I fed those same images to the Spikformer using the best performance checkpoint (78.24%) to guess the locations. Sadly for me, it scored **19 out of 25 (76%)** correct guesses and beat me in this minigame. I was surprised and satisfied with this result, however, because it meant that my training worked well and my project was overall a success.

6. Unresolved Concepts and Methods

When implementing and modifying the Spikformer model, I wondered how I could measure its energy consumption. However, due to limited time constraints, I could not figure it out and ended up leaving that idea behind. I am very disappointed about this, as evaluating energy efficiency is a crucial part of testing any spiking neural network model over other types of neural network models.

7. Challenges

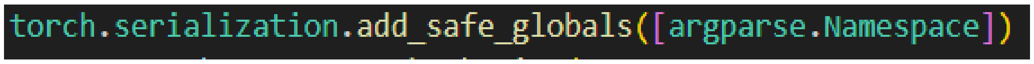
One of my main challenges for this project was working around the limitations of Google Colab's free services. Since Google restricts the free tier's sessions to only 4-5 hours per day, my training would often stop even when I had lots of free time for that day. This led me to some frustrating issues of wasting lots of time that I otherwise would not have if I had my own powerful GPU and thinking about spending money to bypass this limitation.



Cannot connect to GPU backend

You cannot currently connect to a GPU due to usage limits in Colab. Learn more

Additionally, there was also a checkpoint loading issue that resulted in a "Weights only load failed" error during training. Fortunately, after about an hour of searching the web, I was able to identify the error as a compatibility issue related to a PyTorch 2.6 update. I added a line of code to prevent the error from happening, and it worked fine after that.



```
torch.serialization.add_safe_globals([argparse.Namespace])
```

8. Key Insights

Overall, this project was a fun and interactive way to learn about vision models. More specifically, I enjoyed learning about SNNs and the details of the Spikformer model. I also developed a deeper understanding of *how* I should train vision models. This includes factors such as setting up the environment correctly, preprocessing my dataset, analyzing the output data, and much more. After having finished the project and gaining these useful insights, I now feel confident that I will be able to do it again in the future with less trouble. Maybe I'll even be able to design my own model to experiment with. It was a joyful experience to be able to explore these vision models through a topic of my own choosing.