CaNetDa: Deep-Learning Approach to GeoGuessr on a Canada Dataset

Jia Lin Yuan, Steven Tran, Eric Pang
Department of Computer Science
University of Toronto
{jia.yuan, notsteven.tran, ericy.pang}@mail.utoronto.ca

Abstract

In this paper, we look at pinpointing an image location in Canada. Previous papers such as DeepGeo uses convolutional neural networks to predict the location of a given picture in the United States. We explore how well these methods can be extended given a dataset of Canadian locations. We compare results of previously explored methods such as convolutional neural networks [5, 13] to newer image classification techniques such as the Vision Transformer model [2]. Furthermore, we applied data-augmentations to the image to attempt to improve the performance. Due to time-constraints we were not able to fully train our networks but utilized transfer learning techniques instead. We found that our transfer-learned EfficientNet and ViT models work much better than our transfer-learned ResNet.

1 Introduction

As humans, we regularly form associations with our surroundings – subconsciously or not – and these associations helps us identify where we are. As games such as GeoGuesser demonstrates, people can be very good at giving a guess as to where a image is taken using clues such as road signs, landmarks and knowledge of local wildlife.

Our goal in this paper to pick up the intricacies of geography among a data-set of scraped scenery from Google Maps, including details like the surrounding flora, mountains, forests, and lakes. Then, using the details observed in the photo, the goal is to predict the province of where the photo was taken.

Previous works such as DeepGeo [11] utilizes CNNs to predict the location of the picture but restricted to states within the United States of America. DeepGeo trains the ResNet model from scratch and achieves an average accuracy of 38.32% across all states.

Using a similar scraping strategy, we obtain a dataset from Canada and apply transfer learning instead of training the models from scratch due to hardware limitations and time constraints. We used the pre-trained convolutional networks ResNet from `torchvision` [7] and some newer models such as EfficientNet [12] and ViT [2, 14]. We take the output of the pretrained models and train a hidden layer to give us a one-hot-vector providing us with the network's prediction.

We applied data-augmentation strategies. Taking images from our dataset, we concatenated each direction (North, East, South, West) in an attempt to give the network more to work with instead of feeding it one image at a time. We also tried resizing the image and cropping out the google logo as it may interfere with the network.

The contributions of this paper is as follows: 1. that transfer learning works well with location prediction. 2. The Canadian data set. 3. Comparison between the three different models.

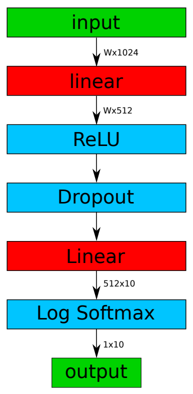


Figure 1: Replacement sequential block for classification

2 Related works

Geolocation is a complex topic that evolves as more advances are made in machine learning. One of the earlier well known implementations was IM2GPS (Hays and Efros, 2008) [4]. Using a dataset of more than 6 million GPS-tagged images from Flickr.com, they used more classical approaches such as 1 nearest neighbours on image features. An implementation using more modern methods is PlaNet [13] which uses a CNN trained on a massive dataset consisting of 126 million images scraped from the internet. Finally, the main inspiration comes from DeepGeo [11], with a ResNet model trained on 100,000 Google Street View images in the USA. IM2GPS and PlaNet focus on reducing the distance between predicted and actual coordinates, while DeepGeo turns it into a classification problem on predicting the US state the image is located in. The time required to train PlaNet (2.5 months on 200 CPU cores) made it infeasible as a foundation for our model. DeepGeo was more realistic as Street View images could be requested via Google's API and there are pretrained versions of ResNet on PyTorch.

3 Methods

3.1 Data

To acquire data necessary for our models based on Canadian datasets, the data scraping process and programs developed by Suresh et al. [11] were adapted to use latitude-longitude bounds of Canadian provinces, and various other performance enhancements were employed when scraping Google's streetview. Images were scraped from the API as 256x256 images and remained unaugmented until processed by the models so that we could consider the effect of these strategies on model performance. As mentioned in their paper, a population density-based approach was used to avoid misses when querying streetview locations.

In the end, two datasets – a large one with over 250000 training examples ($\sim$ 2.6G), and a small one with 40000 training examples ($\sim$ 400M) over 10 provinces – were gathered over the process of roughly three days. We have made the datasets available for use: small dataset, large dataset.

A custom PyTorch dataset and dataloader were also written in order to support batched and multiprocess data loading into the GPU memory for training.

3.2 Models

Two pretrained convolutional neural networks (CNNs) and a transformer model were chosen for use in transfer learning, fine-tuning the last layer of the models for use as province classification models. The replacement sequential block (Figure 1) consists of a linear layer with hidden dimension 512, ReLU, Dropout (p=0.2), linear layer (512x10), and finally, a log softmax over the hidden dimension to obtain log softmax probabilities over the 10 classes. The input image's width varies for each of the three models, since their architectures are different. For fair comparison, all three models described in Section 3.2.1, 3.2.2, and 3.2.3 were pretrained on the same dataset and the relevant layers replaced by the sequential block are explained in the respective sections.

For each of the three models, separate instances were trained using various image augmentation strategies on the raw images (Figure 11a):

1. No augmentation (Figure 11b): images were raw images (Google logo intact) resized to 224x224 and normalized with parameters RGB mean=$(0.485, 0.456, 0.406)$, std. dev=$(0.229, 0.224, 0.225)$
2. Center-cropped (Figure 11c): images were resized to 284x284 and then cropped to 224x224 to perform an symmetric crop removing the Google logo. The same normalization was performed.
3. Concatenated (Figure 11d): images were resized to 224 (Google logo intact) and then concatenated along their widths to obtain a single training example with a dimension of (width, height = 896, 224). The same normalization was performed.

   Note: this strategy was not used for the ViT model because its forward pass requires a 224x224 image to be passed in; this image is decomposed into 16x16 subimages and then used by the transformer as described.

3.2.1 CNN 1: 18-Layer ResNet (supporting code)

As our simplest model, we chose an 18-Layer ResNet [5] pretrained on ImageNet [10]. The model features residual connections which promotes better gradient propagation through deeper models, getting around the vanishing or exploding gradients that may be encountered without the connections. The fully connected layer seen at the end of Figure 3 is the layer replaced by the aforementioned sequential block.

3.2.2 CNN 2: EfficientNet (supporting code)

The next CNN we trained was the baseline EfficientNet `efficientnetb0` [12], also pretrained on the ImageNet dataset. This model benefits from being much smaller than the other models (Table 1) while achieving comparable performance to much larger models (Section 4). Another benefit is that if there was a need for the model to be trained from scratch, EfficientNet would train far faster than ResNet while offering comparable performance.

3.2.3 Transformer: Vision Transformer (ViT) (supporting code)

The transformer we trained was the Vision Transformer [14]. This model utilizes transformers and its attention layers to classify the image instead of convolutional neural networks by splitting the image into many parts and drawing conclusions using the related information within an image. While we used pre-trained models, fully training this model could be much faster due to "computational efficiency and scalability"([2]). The model can be seen in Figure 5.

3.3 Training

We trained the 6 CNNs (EfficientNet and ResNet) on the small dataset using Google Colab's provided GPUs over the course of a day, and the transformer was trained locally over the course of two days as we hit usage limits when training the very large model on Colab. With other parts of the model locked, the fine-tuning of the CNNs' fully connected layers and the transformer's head was done using the Adam optimizer [6] with parameter settings (learning rate=0.003, beta1=0.9, beta2=0.999, epsilon=1e-8).

With the small dataset decomposed into an 80/10/10 training/validation/test split, training ran for 1 epoch as we saw that the models would converge or reach their best performance at that point. The corresponding loss plots are shown in Figures 7-9. Examples of how the training loops vary for the different models are in the highlighted code regions in Source Code 1 and Source Code 2; for concatenated inputs, there is a single forward pass rather than 4 for the separate perspectives.

4 Experiments and Results

4.1 Experiments

To explore the capabilities of each model, we recorded the accuracy of each model overall and in specific provinces on the small dataset. The results for the tests of the models we trained is shown below in Figure 2. We allocated different amount of training time for each model because the running time of the models depends heavily on the amount of parameters of each model. While ResNet and EfficientNet took a reasonable amount of time on Colab, however the Transformer took over 10 minutes per iteration which was not feasible. We used an RTX 3070 to train locally instead. We tested the practicality of each data-augmentation strategy by re-training each model with the strategies applied to the dataset.

Model Base	Total parameters	Size
ResNet	11310410	44 MB
EfficientNet	4338054	17 MB
Vision Transformer	304881162	1.2 GB

Table 1: Parameter counts and sizes of models

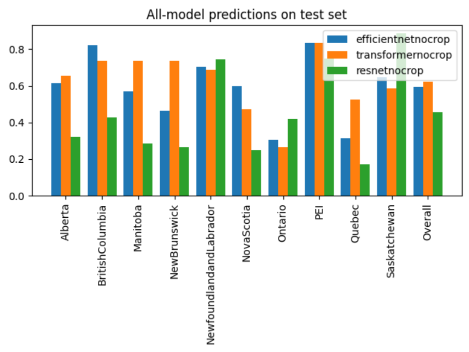


Figure 2: All-model performance without image augmentation

4.2 Comparison of the Models and Data Augmentation

From Figure 2 While we see that the transformer model beats both the convolutional neural net, it is not so obvious that it is the better model. If we look at the parameters used and the size of the model, the transformer uses 60 times the parameter size. Furthermore, the running time is vastly different for the two models, EfficientNet running 4 times faster (11-15 mins vs 2-3 mins) compared to the transformer while the transformer is only around 1-2 percent better.

The results of concatenating or cropping the images are shown in Figure 10. While we suspected that 1) the Google logo maybe interfering with predictions or 2) concatenating the images will give one network more information to work with, it does not seem to have the desired effect. We were only able to run each augmentation twice one run causing an increase in test accuracy and one caused a decrease. More runs will be required to determine the effectiveness of these strategies.

Future revisions to the model may benefit from more image preprocessing prior to the concatenation. As evident in Figure 11, the shakiness of the Google streetview vehicle causes an image that isn't perfectly stitched by merely concatenating the four perspectives. Classical visual computing strategies such as SIFT [9] & autostitch [8] may be useful to match and warp image keypoints in the images, then the impurities due to different camera exposures could be attenuated using Laplacian pyramid blending. This has computational overhead, however, and future research into convolutional layers that get around the imperfect edges may help our models.

4.3 Comparison to DeepGeo

Due to the difference in dataset, it is difficult to make a comparison. If we do a direct comparison (top 1 prediction to top 1 prediction), their best model gives a 39% accuracy whereas ours gives a 62% accuracy. However their data set is a 1 in 50 classification and ours is a 1 in 10. A more "fair" comparison would be to compare our top 1 result to their top 5 result as they are the both 10% of the total classes. This would compare our 62% to DeepGeo's 71.87% accuracy. As suspected, transfer learning cannot compare to training the entire model. There are features that you would look for in the location-guessing that may not be present in a ImageNet-network.

We can do further comparison between this model and DeepGeo by looking at the model used. While not fully trained, ResNet is by far the weakest model of the three. Further testing is needed to conclude if ViT (fully trained) and EfficientNet (fully trained) would be a better model for this task instead of ResNet.

5 Summary

We have found that in all three of the tested models, ResNet; EfficientNet; ViT; cropping and concatenation techniques did not improve the overall accuracy compared to the base model. However each model was largely restricted by computation power, as the original large dataset was not able to be trained on due to time constraints. We also saw that the ResNet performed significantly worse than the EfficientNet and the ViT models, and that the EfficientNet provided the best balance between model performance and computational power. For next steps, we would like to train on the large dataset, include finer image augmentation techniques, and use deeper models.

6 References

[1] Khaled Almezhghwi and Sertan Serte. Improved classification of white blood cells with the generative adversarial network and deep convolutional neural network. *Computational Intelligence and Neuroscience*, 2020:1–12, 07 2020. doi: 10.1155/2020/6490479.

[2] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale, 2020.

[3] Chris Fotache. How to train an image classifier in pytorch and use it to perform basic inference on single images, 2018. URL https://towardsdatascience.com/how-to-train-an-image-classifier-in-pytorch-and-use-it-to-perform-basic-inference-on-single

[4] James Hays and Alexei A. Efros. im2gps: estimating geographic information from a single image. In *Proceedings of the IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2008.

[5] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[6] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[7] György Kovács, János István Iván, Árpád Pányik, and Attila Fazekas. The openip open source image processing library. In *Proceedings of the 18th ACM International Conference on Multimedia*, MM '10, page 1489–1492, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781605589336. doi: 10.1145/1873951.1874255. URL https://doi.org/10.1145/1873951.1874255.

[8] Tianli Liao and Nan Li. Single-perspective warps in natural image stitching. *IEEE Transactions on Image Processing*, 29:724–735, 2020. ISSN 1941-0042. doi: 10.1109/tip.2019.2934344. URL http://dx.doi.org/10.1109/TIP.2019.2934344.

[9] David G. Lowe. Distinctive image features from scale-invariant keypoints. *Int. J. Comput. Vision*, 60(2):91–110, November 2004. ISSN 0920-5691. doi: 10.1023/B:VISI.0000029664.99615.94. URL http://dx.doi.org/10.1023/B:VISI.0000029664.99615.94.

[10] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015. doi: 10.1007/s11263-015-0816-y.

[11] Sudharshan Suresh, Nathaniel Chodosh, and Montiel Abello. Deepgeo: Photo localization with deep neural network. *CoRR*, abs/1810.03077, 2018. URL http://arxiv.org/abs/1810.03077.

[12] Mingxing Tan and Quoc V. Le. Efficientnet: Rethinking model scaling for convolutional neural networks, 2020.

[13] Tobias Weyand, Ilya Kostrikov, and James Philbin. Planet - photo geolocation with convolutional neural networks. *Lecture Notes in Computer Science*, page 37–55, 2016. ISSN 1611-3349. doi: 10.1007/978-3-319-46484-8_3. URL http://dx.doi.org/10.1007/978-3-319-46484-8_3.

[14] Ross Wightman. Pytorch image models. https://github.com/rwightman/pytorch-image-models, 2019.

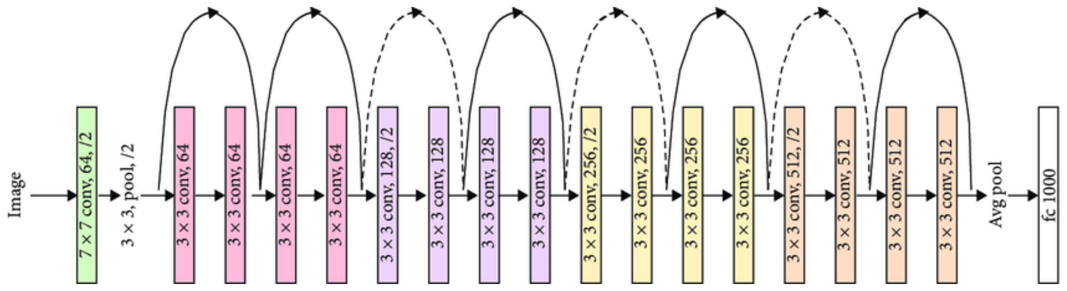


Figure 3: ResNet-18 Model Diagram [1]

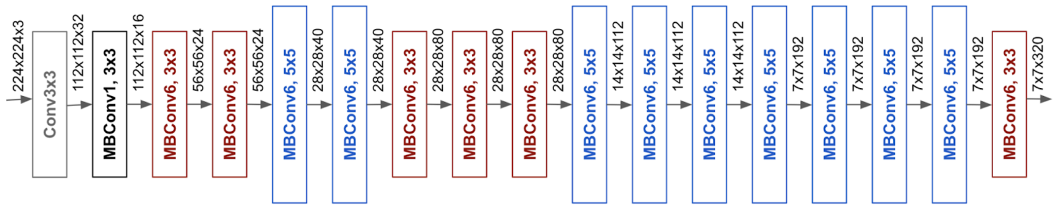


Figure 4: EfficientNet Model Diagram [12]

7 Contributions

We each contributed to the project in the following ways:

- Data scraping: Steven
- Data loading: Steven
- Design of models: Steven, Eric, Alan
- Training the models: Steven, Eric, Alan
- Data visualizations: Steven
- Report writing: Steven, Eric, Alan
  - Abstract and introduction: Alan
  - Related works: Eric
  - Method/algorithm: Steven, Alan, Eric
  - Experiment and results: Alan
  - Summary: Eric
  - References: Steven
  - Appendix: Steven

8 Appendix

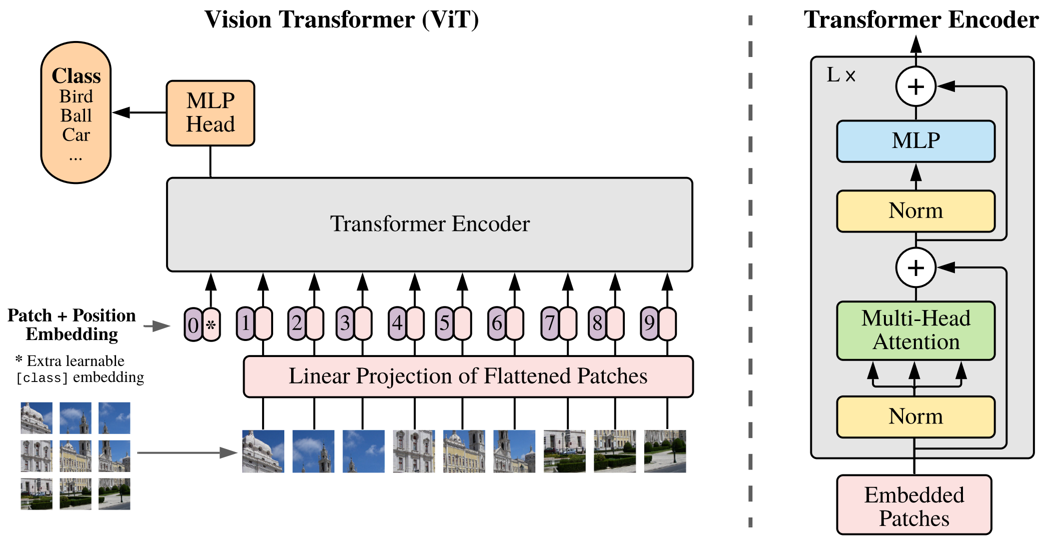


Figure 5: Transformer Model Diagram [2]

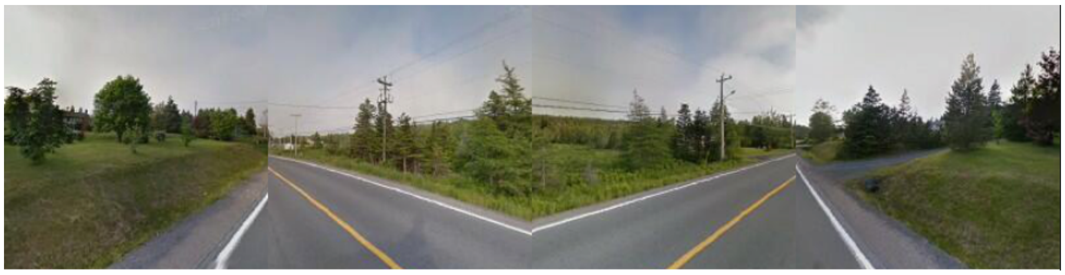
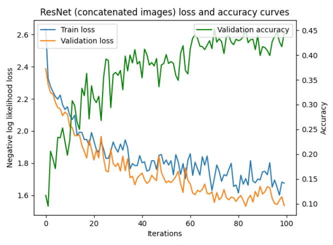
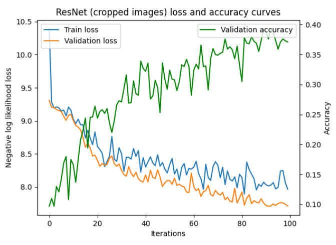


Figure 6: Example of how each model forms predictions. From ResNet with cropped images (starting from the left: north, east, south, and west perspectives). The models combine the NESW log-softmax values by taking the average across the 10 classes, and the prediction is given the argmax of the average.
Actual province: NewfoundlandandLabrador
North prediction: NovaScotia
East prediction: NewfoundlandandLabrador
South prediction: NewfoundlandandLabrador
West prediction: Quebec
Combined prediction: NewfoundlandandLabrador


(a) Training curves; concatenated images


(b) Training curves; cropped images

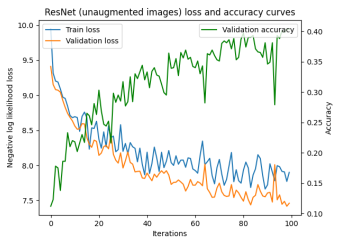
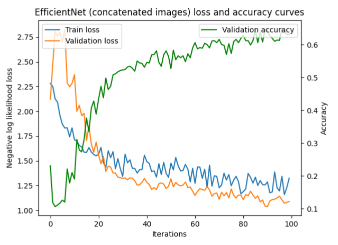
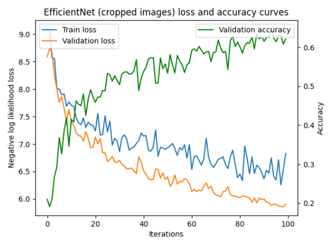

(c) Training curves; unaugmented images

Figure 7: ResNet-based model


(a) Training curves; concatenated images


(b) Training curves; cropped images

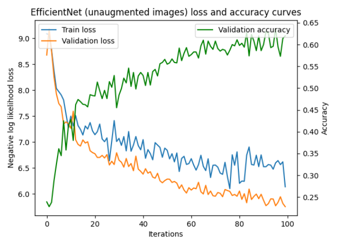
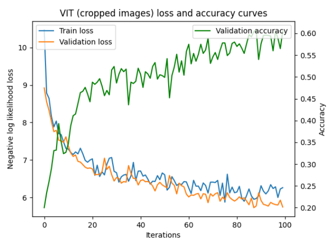

(c) Training curves; unaugmented images

Figure 8: EfficientNet-based model


(a) Training curves; cropped images

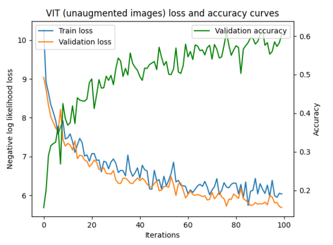
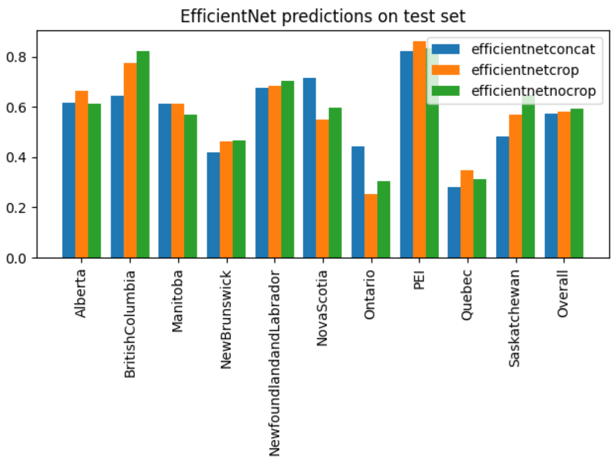
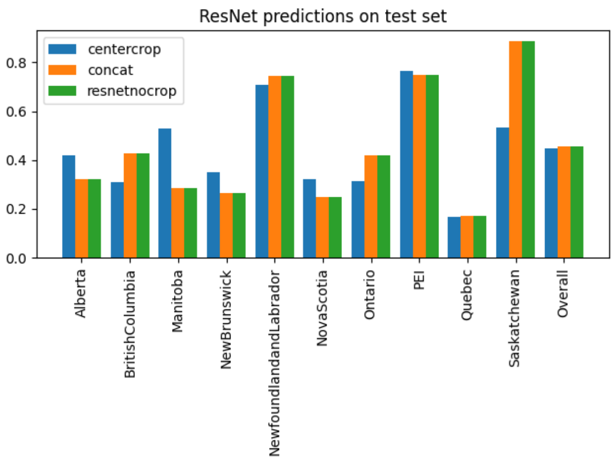

(b) Training curves; unaugmented images

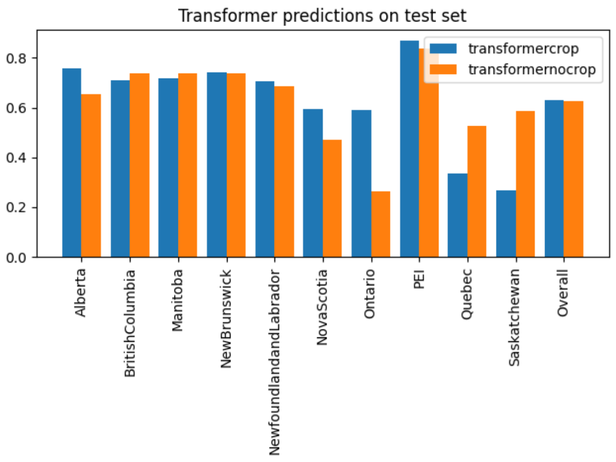
Figure 9: ViT-based model



(a) EfficientNet prediction performance

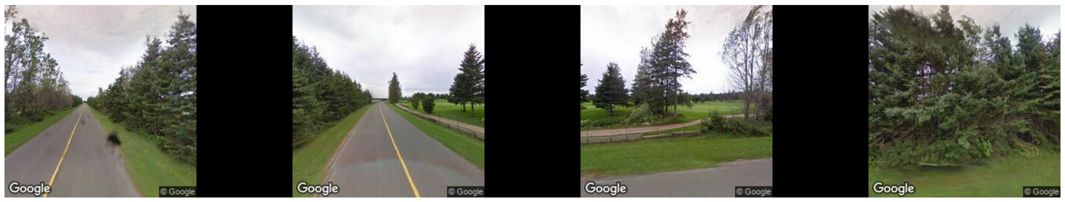


(b) ResNet prediction performance

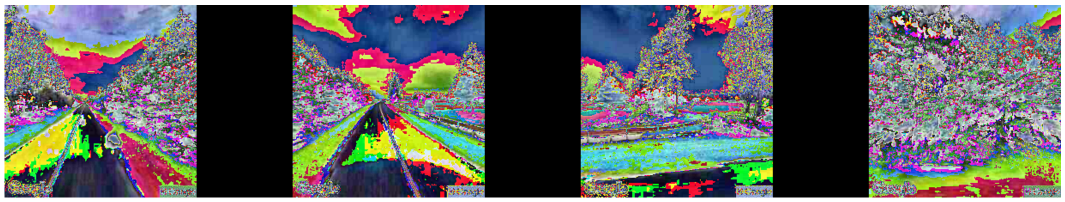


(c) Transformer prediction performance

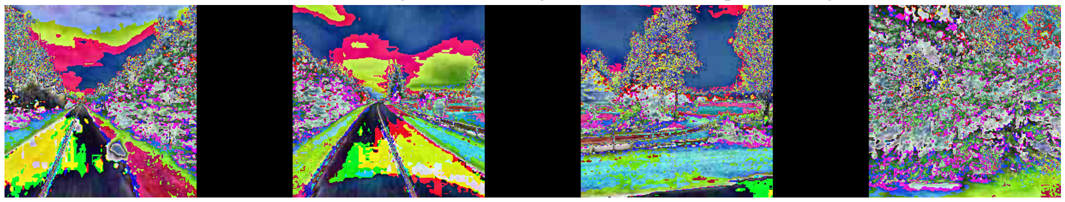
Figure 10: Models' performances on test set



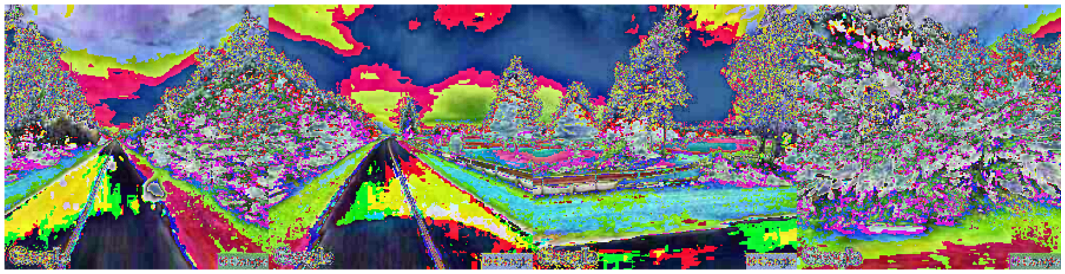
(a) Raw images



(b) Transformed images without augmentation (four separate images)



(c) Transformed images with cropping (four separate images)



(d) Transformed and concatenated images (one image)

Figure 11: Examples of different image augmentation strategies

Source Code 1: Training loop for models trained without concatenated inputs [3]

```
device = torch.device("cuda" if torch.cuda.is_available()
                                     else "cpu")
# Different for other models
model = EfficientNet.from_pretrained('efficientnet-b0')
for param in model.parameters():
    param.requires_grad = False

model._fc = nn.Sequential(nn.Linear(1280, 256),
                                 nn.ReLU(),
                                 nn.Dropout(0.2),
                                 nn.Linear(256, 10),
                                 nn.LogSoftmax(dim=1))
criterion = nn.NLLLoss()
optimizer = torch.optim.Adam(model._fc.parameters(), lr=0.003)
model.to(device)

epochs = 1
steps = 0
running_loss = 0
print_every = 10
train_losses, test_losses, accs = [], [], []

model.eval()
for epoch in range(epochs):
      for i, data in enumerate(train_loader):
          inputs, labels = data
          steps += 1
          inputs, labels = inputs.to(device), labels.to(device)
          optimizer.zero_grad()

          # Different for concatenated inputs
          n_logps = model.forward(inputs[:,0,...])
          n_loss = criterion(n_logps, labels)
          n_loss.backward()

          e_logps = model.forward(inputs[:,1,...])
          e_loss = criterion(e_logps, labels)
          e_loss.backward()

          s_logps = model.forward(inputs[:,2,...])
          s_loss = criterion(s_logps, labels)
          s_loss.backward()

          w_logps = model.forward(inputs[:,3,...])
          w_loss = criterion(w_logps, labels)
          w_loss.backward()

          loss = n_loss + e_loss + s_loss + w_loss

          optimizer.step()
          running_loss += loss.item()

          if steps % print_every == 0:
              test_loss = 0
              accuracy = 0
              model.eval()
              with torch.no_grad():
                  for inputs, labels in test_loader:
                      inputs, labels = inputs.to(device), labels.to(device)

                      # Different for concatenated inputs
                      n_logps = model.forward(inputs[:,0,...])
                      n_batch_loss = criterion(n_logps, labels)
                      test_loss += n_batch_loss.item()

                      e_logps = model.forward(inputs[:,1,...])
                      e_batch_loss = criterion(e_logps, labels)
                      test_loss += e_batch_loss.item()

                      s_logps = model.forward(inputs[:,2,...])
                      s_batch_loss = criterion(s_logps, labels)
                      test_loss += s_batch_loss.item()

                      w_logps = model.forward(inputs[:,3,...])
                      w_batch_loss = criterion(w_logps, labels)
                      test_loss += w_batch_loss.item()

                      logps = (n_logps + e_logps + s_logps + w_logps)/4

                      ps = torch.exp(logps)
                      top_p, top_class = ps.topk(1, dim=1)
                      equals = top_class == labels.view(*top_class.shape)
                      accuracy += torch.mean(equals.type(torch.FloatTensor)).item()
              train_losses.append(running_loss/len(train_loader))
              test_losses.append(test_loss/len(test_loader))
              accs.append(accuracy/len(test_loader))
              print(f"Epoch {epoch+1}/{epochs}.. "
                    f"Train loss: {running_loss/print_every:.3f}.. "
                    f"Test loss: {test_loss/len(test_loader):.3f}.. "
                    f"Test accuracy: {accuracy/len(test_loader):.3f}")

              running_loss = 0
              model.train()
```

Source Code 2: Training loop for models trained with concatenated inputs [3]

```
device = torch.device("cuda" if torch.cuda.is_available()
                                  else "cpu")
# Different for other models
model = EfficientNet.from_pretrained('efficientnet-b0')
for param in model.parameters():
    param.requires_grad = False

model._fc = nn.Sequential(nn.Linear(1280, 256),
                                 nn.ReLU(),
                                 nn.Dropout(0.2),
                                 nn.Linear(256, 10),
                                 nn.LogSoftmax(dim=1))
criterion = nn.NLLLoss()
optimizer = torch.optim.Adam(model._fc.parameters(), lr=0.003)
model.to(device)

epochs = 1
steps = 0
running_loss = 0
print_every = 10
train_losses, test_losses, accs = [], [], []

model.eval()
for epoch in range(epochs):
      for i, data in enumerate(train_loader):
          inputs, labels = data
          steps += 1
          inputs, labels = inputs.to(device), labels.to(device)
          optimizer.zero_grad()

          # Different for non-concatenated inputs
```

```
            new_in = torch.cat(
              [inputs[:,0,...], inputs[:,1,...], inputs[:,2,...], inputs[:,3,...]],
            dim = 3)
            n_logps = model.forward(new_in)
            n_loss = criterion(n_logps, labels)
            n_loss.backward()

            loss = n_loss

            optimizer.step()
            running_loss += loss.item()

            if steps % print_every == 0:
                test_loss = 0
                accuracy = 0
                model.eval()
                with torch.no_grad():
                    for inputs, labels in test_loader:
                        inputs, labels = inputs.to(device), labels.to(device)

                        # Different for non-concatenated inputs
                        new_in = torch.cat(
                          [inputs[:,0,...], inputs[:,1,...], inputs[:,2,...], inputs[:,3,...]],
                        dim = 3)
                        n_logps = model.forward(new_in)
                        n_batch_loss = criterion(n_logps, labels)
                        test_loss += n_batch_loss.item()

                        logps = n_logps

                        ps = torch.exp(logps)
                        top_p, top_class = ps.topk(1, dim=1)
                        equals = top_class == labels.view(*top_class.shape)
                        accuracy += torch.mean(equals.type(torch.FloatTensor)).item()
                train_losses.append(running_loss/len(train_loader))
                test_losses.append(test_loss/len(test_loader))
                accs.append(accuracy/len(test_loader))
                print(f"Epoch {epoch+1}/{epochs}.. "
                      f"Train loss: {running_loss/print_every:.3f}.. "
                      f"Test loss: {test_loss/len(test_loader):.3f}.. "
                      f"Test accuracy: {accuracy/len(test_loader):.3f}")

                running_loss = 0
                model.train()
```