Aditya Kanteti
Phoenix Cloutier

Introduction

Our final project task was to build an image classifier using a dataset shared with us, one that's similar to Imagenet. In our previous HW assignments we used CNN models, but now we had the freedom to use modern architectures and test their accuracies against the dataset we were given. Here we also had the opportunity to use pre-trained weights and fine-tune models instead of training them from scratch.

ViT_L_16_Weights.IMAGENET1K_SWAG_E2E_V1:

These weights are learnt via transfer learning by end-to-end fine-tuning the original SWAG weights on ImageNet-1K data.

acc@1 (on ImageNet-1K)	88.064
acc@5 (on ImageNet-1K)	98.512
categories	tench, goldfish, great white shark, ... (997 omitted)

- *After experimenting with weights, we settled with the IMAGENET1k_SWAG_E2E_V1 weights.*

Throughout the process, we learned about how to balance training with overfitting, and how to extract as much information from a limited dataset, while keeping the model generalizable. We experimented with freezing/unfreezing the backbone and using different transformations on the dataset. We used a 100-head classifier for the final layer in our ViT model, which we fine-tuned instead of the backbone.

We also saw that leveraging the most appropriate pre-trained weights allowed us to take advantage of information learned from other's training, which improved our model. Since we had a small dataset (compared to datasets like imagenet) we wanted to use this. We fine-tuned our model and decided to freeze the backbone and only fine-tune the classifier head, which allowed us to balance efficiency with accuracy, though in the future we hope to work with unfreezing layers gradually.

In the training process our training accuracy typically reached around 90-98%, with our validation accuracy hovering around 75%. This shows that we have overfitting, something that can be addressed with regularization techniques, or changing the data augmentation. Currently our best model weights have a validation accuracy around 78% and a kaggle accuracy of 80%.

How to Reproduce Our Results

In order to reproduce our results, download the .ipynb file that was submitted along with this report. Open a Google Cloud workbench with our file, as well as the folder containing the training and testing images as originally formatted. Then, run each cell individually, in the order

that they appear in the notebook, ensuring that each finishes running before starting the next. This will generate the model weights that we submitted, which will be saved in the same directory as the file, as well as a submission csv file, similar to the one that our team submitted to Kaggle.

Model Weights:

https://drive.google.com/file/d/1EqMqNzREmTP3YkuElf6M7lZX464WRKcz/view?usp=sharing

Dataset

The dataset we got has 100 classes, each with 10 images. We have a total of 1000 training images and a test set of 1000 images. On top of that the Kaggle leaderboard testing set is around 10% of the samples from the final test set - so we can expect our training dataset to be a fraction of the true test dataset.

- *We see that the images have many different perspectives, angles, lighting conditions, and orientations:*

Because of this limited number of images, we risked both overfitting and underfitting. This is what ended up pushing us to implement data augmentation techniques, to ‘diversify’ the dataset. We decided to also set a uniform size of 512 pixels on all images, instead of the typical size of 224, because our Vision Transformer model threw an error with 224.

Beyond the images, the dataset was a challenge because many of the images are very similar to other classes. One example of this was when looking at cars, and in the cases where the model was incorrect, it was often because some of the images closely matched other classes. We used data augmentation and pre-trained weights for this reason, to ensure that we can generalize well to unseen images, and understand the differences between classes.

- *Here is an example of images from different classes that are similar to each other*

Data Preprocessing

First all images were resized to 512 pixels (224 led to issues). After that, we applied a series of transformations to ensure our model generalized well, and to augment the dataset. This increased the effective size of our dataset, and helped reduce overfitting.

```
train_transforms = transforms.Compose([
    transforms.Resize(512),
    transforms.RandomCrop(512),
    transforms.RandomHorizontalFlip(p=0.4),
    transforms.RandAugment(num_ops=2, magnitude=7),
    transforms.RandomRotation(10),
    transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.5, hue=0.2),
    transforms.RandomAffine(degrees=15, translate=(0.1, 0.1)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                         std=[0.229, 0.224, 0.225])
])
```

- *Here is a snippet of what transformations we chose to use for our model.*

- *Here is an example of what RandomAffine looks like, from Pytorch documentation*

We normalized both the training and validation datasets so our model could properly use the pre-trained weights, and for our validation set, we chose to only use center cropping instead of random cropping to maintain consistency during evaluation.

```
val_transforms = transforms.Compose([
    transforms.Resize(512),
    transforms.CenterCrop(512),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                         std=[0.229, 0.224, 0.225])
])
```

```
direct_transform = transforms.Compose([
    transforms.Resize((512, 512)),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406],
                         [0.229, 0.224, 0.225])
])
```

- *Snippets of code of the transformations on validation set and submission set*

Model Architecture

Our initial model architecture was based on Convnext Tiny, and we used their pre-trained weights. In that setup, the classifier head included dropout layers and linear transformations to avoid overfitting.

```
model.classifier[2] = nn.Sequential(
    nn.Dropout(0.5),
    nn.Linear(num_features, 512),
    nn.ReLU(inplace=True),
    nn.Dropout(0.5),
    nn.Linear(512, num_classes)
)
```

- *Snippet of code showing our classifier head with dropout and ReLU*

Still with the data we had access to and issues with fine-tuning, we couldn't balance overfitting and underfitting. This led us to find other architectures that had better pre-trained weights.

We then switched to a Vision Transformer model, specifically vit_l_16 which seemed to perform best on our dataset. Because this model could handle image transformations, learn from the data, and handle pretraining (specifically on imagenet), we chose this architecture over others. We used Imagenet 1k SWAG E2E 1K weights because of how the weights are similar to our dataset, and how well it performed in testing.

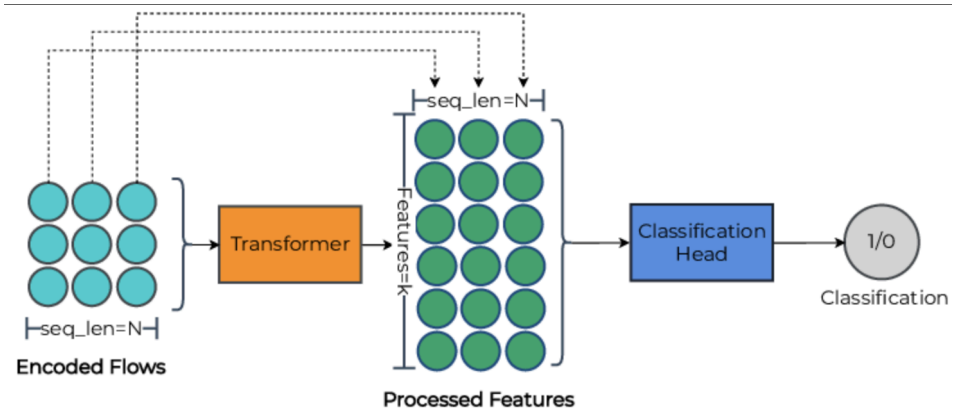
acc@1 (on ImageNet-1K)	79.662
acc@5 (on ImageNet-1K)	94.638

- *Accuracy of ViT_L_16_Weights.IMAGENET1K_V1 (not chosen) from PyTorch*

acc@1 (on ImageNet-1K)	88.064
acc@5 (on ImageNet-1K)	98.512

- Accuracy of ViT_L_16_Weights.IMAGENET1K_SWAG_E2E_V1 (chosen weights)

We replaced the final classification layer to handle our needed output of 100 classes and allowed us to train only the final layer. We decided to freeze the entire backbone and only train the classifier head because we wanted to keep the knowledge from pretraining, while fine tuning for our dataset. We got this idea from attempting to improve our convnext model, in which we tried freezing the backbone and unfreezing it as we went through the epochs.



- *Visualization of the classifier head, image from ResearchGate*

Training + Fine-tuning

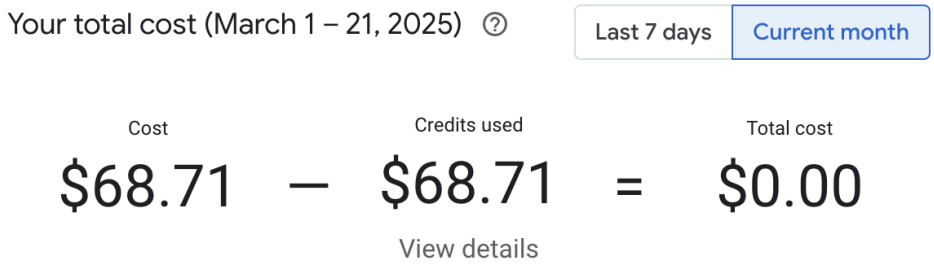
We used cross-entropy loss, along with the AdamW optimizer which handled weight decay well. We set our learning rate to 5e-4 and used a cosine annealing scheduler to adjust the learning rate based on validation accuracy. We also set the batch size to 32 to balance efficiency, since in our previous model's training we went through our Google credits incredibly quickly.

```
# Cross-entropy loss
criterion = nn.CrossEntropyLoss()

# AdamW optimizer
optimizer = optim.AdamW(model.parameters(), lr=5e-4, weight_decay=1e-4)

# Reduced learning rate based on validation accuracy
scheduler = CosineAnnealingLR(optimizer, T_max= 6)
```

- *Screenshot of code showing our (loss) criterion, optimizer, and scheduler*



- *Screenshot of GCP cost over the month from training model(s).*

We messed around with training our model for different numbers of epochs and different data augmentations and settled with nine epochs. We found that training with more epochs led to overfitting, and fewer epochs didn't let the model converge.

We also used dropout and L2 weight decay to reduce the risk of overfitting (a mistake we learned from convnext). Our choice of only training the classifier head was also important as it helped us train quickly while keeping the pre-trained features.

We were careful with our epochs chosen since we saw the training accuracy reach 90-98% and our validation accuracy stabilized around 75%. We plan on gradually unfreezing the layers to avoid overfitting, but for now we are keeping it completely frozen.

Evaluation and Results

We found that our model reached a training accuracy of 98% at its highest, and a validation accuracy of 40%-75% accuracy over the epochs. We used 5-fold cross-validation to ensure that we could capture its true (validation) accuracy. In the first few epochs, we see the biggest jump in accuracy, while later epochs only increase by a few percentages.

```
Train Epoch 1: Loss: 3.6857, Accuracy: 19.00%
Validation Epoch 1: Loss: 2.3950, Accuracy: 44.50%
/opt/conda/lib/python3.10/site-packages/torch/optim
ted where possible. Please use `scheduler.step()` t
he new chainable form, where available. Please open
  warnings.warn(EPOCH_DEPRECATION_WARNING, UserWarn
Model saved with accuracy: 44.50%
Train Epoch 2: Loss: 1.8234, Accuracy: 63.50%
Validation Epoch 2: Loss: 1.8882, Accuracy: 59.00%
Model saved with accuracy: 59.00%
Train Epoch 3: Loss: 1.3088, Accuracy: 72.12%
Validation Epoch 3: Loss: 1.3198, Accuracy: 67.50%
Model saved with accuracy: 67.50%
Train Epoch 4: Loss: 0.9404, Accuracy: 83.38%
Validation Epoch 4: Loss: 1.2661, Accuracy: 67.00%
Train Epoch 5: Loss: 0.8931, Accuracy: 87.12%
Validation Epoch 5: Loss: 1.2427, Accuracy: 67.50%
Train Epoch 6: Loss: 0.8676, Accuracy: 88.50%
Validation Epoch 6: Loss: 1.1958, Accuracy: 70.00%
Model saved with accuracy: 70.00%
Train Epoch 7: Loss: 0.8233, Accuracy: 84.88%
Validation Epoch 7: Loss: 1.0637, Accuracy: 72.50%
Model saved with accuracy: 72.50%
```

- *Screenshot showing how our validation accuracy jumps quickly in first epochs*

For our Convnext model, we saw the highest validation accuracy at 60%, but our submission accuracy was less than 10% on Kaggle. This turned out to be an issue with how our submission csv was structured, but by then we realized that the best way to improve accuracy was to switch to a different architecture.

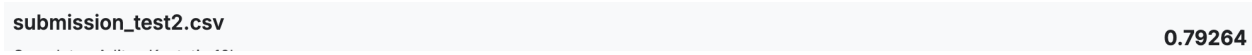
submission

ID	Label
0.jpg	22
1.jpg	56
10.jpg	55
100.jpg	55
101.jpg	91
102.jpg	91
103.jpg	91
104.jpg	79

- *What one of our (bad) submissions looked like from Convnext*

When we first tried standard weights on our (vision transformer) model, our Kaggle accuracy was around 60%, and after changing to our current weights, and tweaking the epochs/augmentations our accuracy jumped to between 75-80%.

- *Before changing our weights to SWAG*



- *After changing our weights to IMAGENET1k_SWAG_E2E_V1*

We noticed that our submission accuracy was between 75-80% on Kaggle, and these fluctuations in our accuracy were dictated by the number of epochs we used, and the data augmentations we used. Oftentimes using more augmentations or more epochs led to a lower accuracy on Kaggle, so we avoided ‘overdoing’ the augmentations and held back on too many epochs.

Conclusion

We found that using a Vision Transformer with the correct weights allowed our model to improve over a CNN or other architectures. Since we spent a lot of time using Convnext and optimizing it to perform well on validation sets/Kaggle submissions, we have a lot of ideas on how to improve our ViT model to go from 80% to being closer to 90%.

One example is to see if aggressive data preprocessing is negatively impacting our accuracy, and whether pulling back specific augmentations will make the model perform better. We already found that random erasing, something that crops out random parts of the image, made our model perform poorly, so digging deeper into the data augmentations and their impact on the accuracy is something we hope to look into. Another idea is to gradually unfreeze the backbone, something that helped our ConvNext model, but we didn’t perform it on our ViT model. Since this makes training longer, and requires more fine-tuning, we didn’t want to spend the limited amount of time and compute we had available on it.

Overall we hope to see how our model performs on the final (full sized) kaggle dataset, and what other teams did to perform well and improve their accuracy.