Ancient Site Detection

1. Introduction

The goal of the project is to detect ancient sites from the given data. We tackle this problem by building a python model. The two given data sets are the training set and test set. The training set has 50,220 images. This is a binary classification problem with 41,850 negatively labeled images and 8,370 positively labeled images. The test set has 12,588 images. All the images are single-channel square images with 129 pixels on each side. The single-channel data of each pixel of the image reflects the elevation of the geographical location.

Initially inspired by the sample code provided by the professor, our model adopts ResNet152 alongside with other modifications such as adaptive learning rate and modified loss function with weight to train on the training set and the final test accuracy reachs 92%.

2. Methodology

Going deeper

After receiving the sample code for this project, we realized the code for the basic residual block is provided. Our group first modified the model in the sample code to have a deeper residual network model since we were introduced to ResNet152 and understand the essence of deep residual networks.

Before the introduction of residual networks, deep convolutional neural networks were very hard to train because of performance degradation. He et al. (2015) mentioned in their paper there is a significant increase in both training error and test error after increasing a convolutional neural network from 20 layers to 56 layers. This result is caused by having a very small gradient after the backpropagation process. To tackle this problem, researchers introduced residual learning.

Residual refers to the difference between the original input value x and the output feature H(x) for some layers in the convolutional neural network; we denote the residual as F(x). To avoid performance degradation, we want an identity mapping of the original input x in the output feature H(x) then even if the residual F(x) is zero, we do not lose any performance in this part of the network. In order to achieve this result, we simply add the original input x to the output of the layers after multiplying the x to the weights and activation; this addition of the original input is also referred to as "shortcut connection" in the paper from He et al. Finally, to make the shortcut connection compatible to regular convolutional layers, we need to make sure the input does not change dimension after multiplying with the weights. This fact also ensures that we can stack residual blocks on top of each other. In the end, ResNet achieves a significantly better result than a "plain" neural network and is not prone to gradient vanishing problems.

In our case, we ditched the residual block in the sample code and utilized ResNet152 in PyTorch, and also modified the network to take single-channel input compared to RGB input in the original problem.

Bottleneck block

Besides residual blocks, bottleneck layers are also an essential part of the very deep convolutional neural network. A bottleneck layer is a layer that contains significantly fewer parameters compared to previous layers. The reduced number of parameters directly results in reduced dimensionality of the input.

In this case, as the number of residual blocks increases, the number of weights also increases proportionally to the number of layers. To reduce the number of total weights, we add a bottleneck unit before the residual block. Therefore, all the following layers have fewer weights. For engineering purposes, ResNet152 has a built-in bottleneck unit; without them, our high-performance computer instantly runs out of memory. The potentially lower accuracy caused by fewer parameters is compensated by the added layers in the network.

Adaptive learning rate

Another engineering technicality is the adaptive learning rate. For gradient descent, we need to have a reasonable learning rate to achieve optimal learning speed. Too low of a learning rate can result in a very slow gradient descent process and too high of a learning rate can cause overshoot. Overshoot happens when there is the gradient descent that goes past the local minimum and can potentially cause the gradient to diverge.

The adaptive learning rate strategy we use starts at a relatively high learning rate and halves itself every 30 epochs. By looking at the loss over time, we realize for the first 30 epochs it frequently overshoots because the loss does not go down; after the learning rate is halved, the loss starts to go down consistently.

Max Pooling

With hundreds of layers and epochs, the training process for the model becomes very slow. In order to do more training in limited time, we added max pooling in the model. Since the size of the image is 129*129, we choose 2 as the kernel size. Then set stride as 1 and padding as 1 to retain precision. With the kernel size of 2, the number of parameters per residual block is reduced by 75%. The code is as below:

```
self.maxpool = nn.MaxPool2d(kernel_size=2, stride=1, padding=1)
```

With max pooling implemented in the network, the speed for training increased by around 33%. Before using the max pooling method, it took around 4.5 minutes to train each epoch. With the max pooling method, the train time was shortened to around 3 minutes.

Dropout

Dropout is a commonly used regularization method in deep neural networks in fully connected layers; it makes nodes within a layer probabilistically take on more or less responsibility for the inputs. By disabling several different neurons by some small probability, we are essentially training many different neural networks in parallel. After

applying dropout, the model becomes much stronger in generalizing therefore prevents overfitting by a great degree.

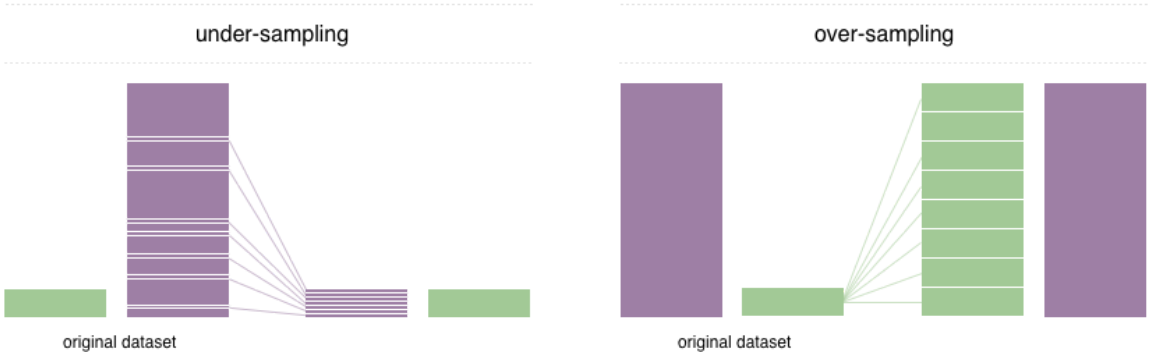
When applying the dropout method to convolutional layers, we are adding noise to the pattern and improving the robustness of the convolutional network. After testing, we found that applying batch normalization provides better results than applying dropout along to our model. In Understanding the Disharmony between Dropout and Batch Normalization by Variance Shift, Li et al. (2018) also pointed out that applying dropout before batch normalization results in unfavorable behaviors of the model. Instead of using gaussian dropout, we decided to ditch dropout all together for better performance.

Balancing the dataset

After counting our train datasets, we found it is an imbalanced dataset whose ratio of positive and negative is around 4 : 21. This situation is very concerning because we can achieve an accuracy over 0.8 by even predicting all the samples as negative. To treat this problem, we have tried two strategies respectively; one is the resampling method and the other is modifying the loss function with weights.

a) Resampling

Resampling consists of removing samples from the majority class (under-sampling) and adding more examples to the minority class (over-sampling) so that the ratio of the number of samples in each class will be balanced.



For this project we choose oversampling because of the lack of positively labeled

data. We only have around 50,000 samples and we will lose most of them if we choose under-sampling.

We can use the *WeightedRandomSampler()* method from PyTorch to implement oversampling. This method assigns each class a weight which is inverse proportional to the distribution of the classes. For example, the ratio of positively and negatively labeled training data is about 21 : 4, so the weight of the negative class is 1/42 ~ 0.0238 and the positive class is 1/8 = 0.125. It should be noticed that the sum of weights doesn't need to be 1.

One thing should be noted is that the argument "replacement" is set to *True*, which means the sample can be picked more than once. This is our exact desired outcome from this oversampling method.

```
1.     #extract labels from train set
2.      train_classes = [i['label'].item() for i in trainset]
3.      label_list = torch.tensor(train_classes)
4.      label_list = label_list[torch.randperm(len(label_list))]
5.      label_list = label_list.long()
6.
7.      #count weights for each class and assign to each sample
8.      class_ratio_count = [42, 8]
9.      class_weights = 1.0 / torch.Tensor(class_ratio_count)
10.     class_weights_all = class_weights[label_list]
11.
12.     #set up WeightedRandomSampler
13.     sampler =
    torch.utils.data.sampler.WeightedRandomSampler(class_weights_all,
    len(class_weights_all), replacement = True)
14.
15.     #set up dataloader
16.     train_loader = DataLoader(trainset, batch_size = args.batch_size,
    shuffle = False, sampler = sampler, num_workers = 5)
```

b) Loss function with weight

The idea of modifying the loss function with weights is to weigh the loss computed for different samples differently based on whether they belong to the majority or the minority of classes. We essentially want to assign a higher weight to the loss encountered by the samples associated with class with less samples.

In our project, we use the *CrossEntropyLoss* method as our loss function which has built-in weight parameters. The original loss function can be described as:

$$\mathbf{loss}(x,class)=-\log(\sum_j \exp(x[class])/\exp(x[j]))=-x[class]+\log(\sum_j \exp(x[j]))$$

After introducing weight, it can be described as:

$$\mathbf{loss}(x,class)=weight[class](-x[class]+\log(\sum_j \exp(x[j])))$$

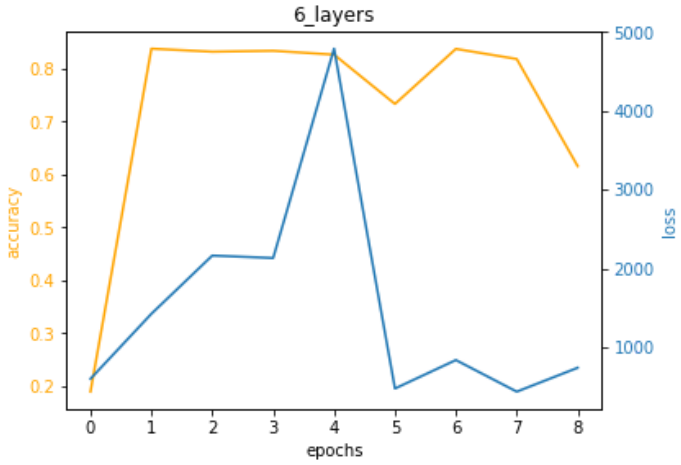
$$\mathbf{loss}=\sum_{i=1}^{N} loss(i,class[i]) \,/\, \sum_{i=1}^{N} weight[class[i]]$$

We implement the formula using the following lines of code:

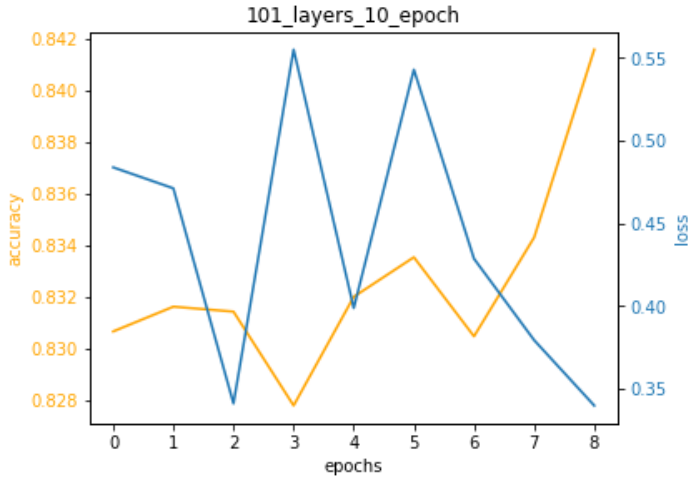
```
1.      train_classes = [i['label'].item() for i in trainset]
2.      falseCounter = Counter(train_classes)[0]
3.      trueCounter = Counter(train_classes)[1]
4.      ratio = round(falseCounter/trueCounter, 3)
5.      weights = torch.tensor([ratio, 1.], device = device0)
6.
7.      loss_function = nn.CrossEntropyLoss(weights)
```

3. Results

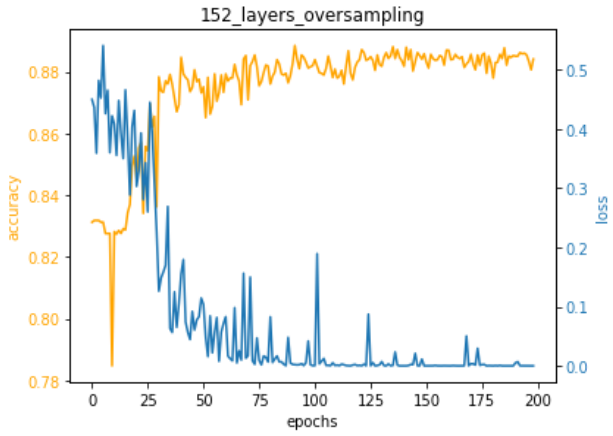
We ran the sample model provided on latte and the training graph of our result is shown below:

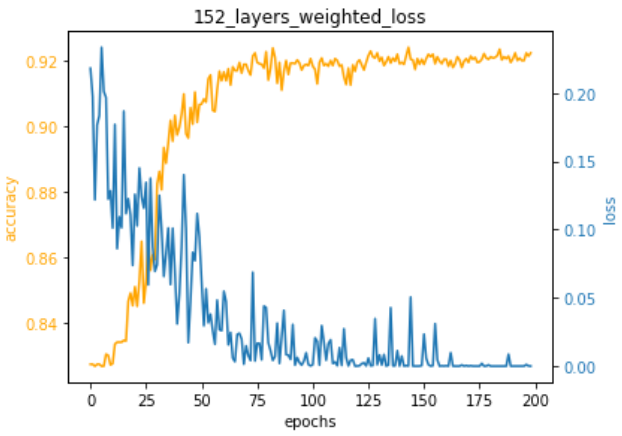


The result is not bad especially it only has 6 layers and was trained on 9 epochs. However, its accuracy and loss are not very stable and the result even gets worse in the last two epochs. We thought it happened because the amount of layers was too few. Therefore, we decided to add more layers to our model. We observed that the basic residual block has been implemented in sample code, so we adjusted it a little and increased the number of layers to 101. The new result is shown below:



The graph shows that the training accuracy becomes more unstable but the trend still increases. This result suggests we need to increase the number of training cycles to make the model converge. After increasing the training epochs (to 200 epochs), the model with 101 layers has a significant increase in accuracy which is around 86%. After comparing the training result of the 6-layer model to the 101-layer model, we also can conclude a deeper model considerably outperforms the shallow one. Therefore, we added layers to the model until it has 152 layers which resemble ResNet152. However, the large model requires more GPU memory and time than what we were provided, so we introduced “bottleneck block” and “max pooling” to reduce the cost of GPU memory and time. Beside, we also observe that the training dataset is imbalanced with a negative and positive ratio of 21 : 4, which means even if we predict all the samples as negative we still get a good accuracy over 0.8. Therefore, we have to deal with the imbalanced data using the strategies mentioned previously. The results are shown on the next page. Both of them have increased performance but the one with weighted loss function has better results. Lastly, we used the model train by weighted loss function to predict the test data set provided by the professor.





4. Discussion

In the sample code, the default CNN model contains three convolutional layers. For the first one, the kernel size is 1. For the second one, the kernel size is 3 and the padding is 1. For the third one, the kernel size becomes 1 again. The stride is 1 for all three layers. With only 6 layers and 10 epochs, the test accuracies are around 83% (Figure 1). Then we tried to add more layers to check whether the test accuracies could increase quickly.

We add more ResBlock() in the SimpleModel class.

```
1. class SimpleModel(nn.Module):
2.     def __init__(self, num_class=2):
3.         super(SimpleModel, self).__init__()
4.         self.num_class = num_class
5.         self.block1 = nn.Sequential(
6.             ResBlock(in_channel = 1, out_channels=[4, 4], s = 1),
7.             ResBlock(in_channel = 1, out_channels=[4, 4], s = 1)),
8.             ResBlock(in_channel = 1, out_channels=[4, 4], s = 1))
9.         self.fc = nn.Linear(129 * 129, self.num_class)
```

There is no significant increase in the test accuracies for the 9-layer model (Figure 2). In addition, the test accuracies for each epoch were not stable. Sometimes they would plunge to 20% for both models.

Epoch	Test Accuracy
1	18.95%
2	83.83%

3	83.26%
4	83.43%
5	82.70%
6	73.37%
7	83.79%
8	81.88%
9	61.61%
10	74.75%

Figure 1: 6 Convolutional Layers

Epoch	Test Accuracy
1	82.70%
2	19.96%
3	44.02%
4	83.14%
5	49.94%
6	83.01%
7	83.24%
8	82.95%
9	83.01%
10	84.12%

Figure 2: 9 Convolutional layers

Judging from the test accuracies from models with 6 and 9 convolutional layers, we suspected that the volatile fluctuation between epochs was due to insufficient number of convolutional layers. Then we increased the number of convolutional layers to 101. The result was better (Figure 3) but took significantly more time to run each epoch. The accuracy increased steadily and at 10 epochs the accuracy became 84.5% and more importantly accuracy between each epoch was much more stable than before.

Epoch	Test Accuracy
1	83.07%
2	83.16%
3	83.14%

4	82.78%
5	83.20%
6	83.35%
7	83.05%
8	83.43%
9	84.16%
10	84.48%

Figure 3: 101 Convolutional Layers

Now we know that going deeper in convolutional layers is very likely to be the correct approach to increase the model accuracy. But on the way of increasing convolutional layers, the training time between each epoch increased significantly and we had to increase the stride of each convolutional layer to up to 4, because otherwise even the V100 GPU with 32GB of VRAM would run out of memory.

Now that the main problem shifted from model accuracy to model efficiency. After discussion we decided to utilize ideas from the existing ResNet152 model. First thing we did was to use a bottleneck structure, which basically means a 1x1 convolution is performed before our original residual block to reduce the channels of input before the 3x3 convolutional layers to save VRAM. Later another 1x1 convolutional layer will project it back to the original shape. Second thing we did was to implement a max pooling layer, with the kernel size of 2, the number of parameters per residual block was reduced by 75% and thus our model can use VRAM more efficiently.

After implementing the two approaches stated above, the VRAM was no longer a constraint and we can successfully train the ResNet152 model within reasonable time (about 15 hours). The resulting model can now achieve accuracy of around 90%. Then we discussed several methods which could potentially increase the model accuracy. The first method we tried was using an adaptive learning rate. The adaptive learning rate strategy we utilized starts at a relatively high learning rate and half the learning rate every 30 epochs. In this case we start at 0.0004 and decrease it by half every 30 epochs. The resulting model converged at a much faster rate while maintaining a high accuracy, the epochs needed to reach accuracy of around 90% effectively reduced by around 30%. After this we noticed that the dataset is highly imbalanced so the second method we tried is using oversampling or adding the weight to the loss function. We tried both ways and adding the weight to the loss function turned out to have better accuracy, we

think that oversampling may have caused overfitting. The last approach we tried is the dropout method, there are a lot of papers proving dropout will lead to slightly better results. But here the dropout method proves to be a failed attempt. After doing some research we found out that the dropout method doesn't work well with batch normalization. Thus we later deleted this method.

5. Future Work

To improve upon our current model, we could update our accuracy measurement to F-1 score or AUROC for this binary classification problem. This can give us a better understanding of the model's performance while dealing with an imbalanced dataset. Furthermore, we could train our model on the original dataset with multiple class labels instead of binary class labels in this case. The multi-class label may provide better understanding and fine tuning possibility for the model. Another idea for improvement is to assign less weights to the dominant classes.

6. Contribution

Hao Hu:

Code: Implemented the main structure of ResNet152 in the model, added weights to the loss function and adjusted code.

Paper: Edited the Introduction, wrote part of Methodology and Discussion.Adjusted the format of the paper.

Kexu Qian:

Code: Implement the max pooling method in the model and adjust code.

Paper: Write the Introduction, part of Methodology and part of Discussion. Adjust the format of the paper.

Zian Yang:

Code: Implemented training results visualization and modified code for prediction to make it runnable on local machines.

Paper: Edited Introduction and revised results. Wrote "Going deeper", "Bottleneck Block", "Adaptive learning rate", "Dropout" part of Methodology.

Wei Liu:

Code:Implement the oversampling method and adjust model

Paper: Write the Result and "Balancing the dataset" part of Methodology