Extracting Binary Trees from Tree Drawings

Chami Lamelas Ryan Polhemus

December 15, 2022

1 Introduction

For our project, we built a model for constructing the binary trees that correspond to drawings of trees in images. One application of such a model would be to automatically grade questions that ask the respondent to draw a binary tree. For example, one could ask the respondent to draw an example of a binary search tree. The model would read in the image the student submits and produce a unique mathematical representation of the tree. This representation could be passed to another program that would determine if the tree is indeed a binary search tree and assign the student their score. We hope that software such as this could be one day integrated into existing educational software such as Gradescope [16] to expand its autograding functionality for computer science courses.

We will not be considering the problem of recognizing all possible binary trees in order to make both the data generation and model development processes easier. Since we are unaware of any existing datasets of hand drawn binary trees, we generated various sets of binary trees that we attempted to make appear hand-drawn. Our data generation process is described in more detail in section 4.2. In particular, we will only be considering images containing only a single binary tree and nothing else. Furthermore, the tree must be representable with an *edge set* $y \in \{0,1\}^S$. The details of an edge set, including exactly how S is defined, are described in detail in section 4.1. Trees that can be represented as an edge set must meet certain requirements. A few of these requirements are not reasonable to place on respondents who would be evaluated with such a program. Despite the fact that our particular software could not be used to grade generic tree drawing problems, we believe that it serves as a potential proof of concept for the development of such software in the future.

To build this software, we use a neural network to predict the binary tree that corresponds to a particular image. The input to the network is an RGB image of a drawing of a binary tree. Figure 1 shows examples of binary tree drawings that we generated. The network produces an edge set y as the output. y is a binary vector that provides one way of representing certain binary trees. This was a primary reason as to why we restricted our problem to trees that can be represented as edge sets. By having our model predict edge sets, training our model using ground truth edge sets becomes a multi-label binary classification problem. These problems have existed for some time [18] and there exist applicable loss functions in PyTorch [15] that can be used for training our model.

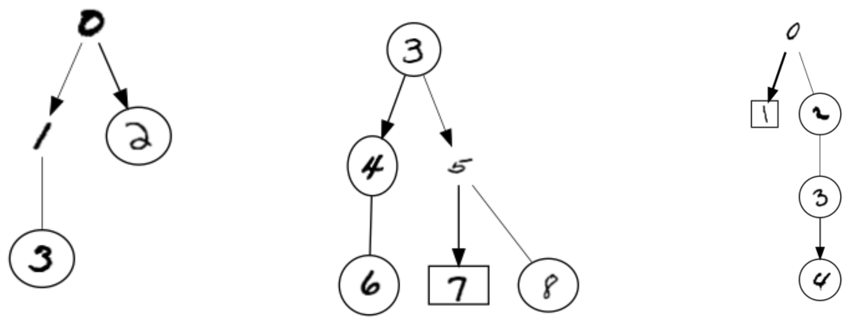

Figure 1: Sample Generated Binary Tree Drawings

We believe that trees are a form of a sequence in that nodes are linked together by edges. For this reason, we applied a transformer model [19] directly to the image to predict the output tree set. Thus, the problem was treated as purely multi-label binary classification with no explicit object detection of nodes or edges. We further discuss our motivation for selecting transformer models in section 3.

At the time of writing, we are unaware of any existing work on this problem. We evaluated the performance of our models on their accuracy of correctly predicting the edge sets corresponding to input images. We hoped to achieve good results on especially node value identification. We did not believe any particular tree structures would be more difficult to learn than others and were unsure of the accuracy of our model at predicting edge sets. However, we explain how both of these beliefs were not upheld in our experiments.

2 Related Work

As mentioned previously, we are unable to find any existing work on problems related to extracting binary trees or any similar data structures from drawings in images. Nodes in a tree are defined based on their relationship with other nodes in the tree; most importantly its parents and children. This is related to the problem of determining the layout and structure of a scene in an image. This is an active field in object detection, and we considered adapting a simplified version of the layout codebook suggested in [20] to fit the use case of this project; the set of possible trees is small enough that an entry in the codebook for each layout is not out of the question.

Based on the recent survey [13] by Khan et al. on transformers in vision, there is not a particular network that is suitable to this type of object detection task. For object detection, they discuss a variety of hybrid CNN and transformer architectures. We considered using the YOLOS [12] end-to-end transformer, but it is made for conventional object detection tasks that utilize bounding boxes in addition to class labels. As described in section 1, we will be solving this problem as a multi-label binary classification problem through the use of edge set representations of trees.

3 Background

We decided to work with the popular Vision Transformer (ViT) model [11] [13] [21]. We worked with two different implementations. The first is the PyTorch implementation [6] that follows the architecture described in "An Image is Worth 16 x 16 Words: Transformers for Image Recognition at Scale" [11] by Dosovitskiy et al. The authors make two key points that we believe make ViT a good model for the task of extracting binary trees from images. The first is that ViT, via its learned position embeddings, is able to learn distances in an image. The second observation the authors make is that starting from early layers, ViT is able to aggregate information from throughout the image via its underlying attention mechanism. We feel that both of these capabilities are crucial in identifying how nodes fit into the tree in which they are contained. In particular, we believe it can help the model learn which nodes are connected by edges and which are not. The second architecture we used was the implementation provided in Dive to Deep Learning [21]. This architecture was also based off of the work of Dosovitskiy et al., but is smaller scale. The original architecture is composed of eleven encoder blocks while the Dive to Deep Learning implementation has only two.

4 Method

In this section, we first describe edge sets which we use to mathematically represent binary trees. This is crucial to understand how our modelling process works. Next, we describe how we generated tree drawing datasets of varying complexity. Lastly, we describe our model architecture, training, and prediction.

4.1 Edge Sets

An edge set y can be used to uniquely represent binary trees that satisfy five properties.

1. The tree must be made up of 1 to a predetermined maximum $M > 1$ nodes.
2. The values of the tree must be unique and within $0, 1, \ldots, M-1$.
3. A node must have a lower value than its children.
4. If a node has a right child then it must have a left child.
5. Nodes on the same level of the tree must be increasing in value moving left to right.

These restrictions are imposed by how a binary tree can be uniquely built from an edge set. This particular set of restrictions can be derived from algorithm 1. Given the above restrictions, we can now more properly define y. Note we use the notation $\exists a \rightarrow b$ to mean that a node with value b is a child of node with value a.

$$\begin{array}{rrcr} y_0 = \mathbb{1}\{\exists 0 \rightarrow 1\} & y_1 = \mathbb{1}\{\exists 0 \rightarrow 2\} & \ldots & y_{M-1} = \mathbb{1}\{\exists 0 \rightarrow M-1\} \\ y_M = \mathbb{1}\{\exists 1 \rightarrow 2\} & y_{M+1} = \mathbb{1}\{\exists 1 \rightarrow 3\} & \ldots & y_{2M-3} = \mathbb{1}\{\exists 1 \rightarrow M-1\} \\ & & \ldots & \\ & & & y_S = \mathbb{1}\{\exists M-2 \rightarrow M-1\} \end{array}$$

Hence the first $M-1$ elements of y indicate the presence of edges from the node with value 0 to nodes with values $1, \ldots, M-1$. The next $M-2$ elements of y indicate the presence of edges from the node with value 1 to nodes with values $2, \ldots, M-1$. Following this pattern that means $S = M-1+M-2+\cdots+1 = ((M-1)M)/2$

Algorithm 1 Algorithm for Reconstructing Trees from Edge Sets

```
Require: An edge set $y$ and maximum number of nodes $M$ as defined in section 4.1. Below, node
    refers to a binary tree node defined in the standard manner [7]. An edge has a parent and a
    child.
Ensure: $R$ will hold the root of the constructed tree.
  $N \leftarrow$ Array of $M$ nodes with values $0 \dots M-1$
  $F \leftarrow 0$
  $R \leftarrow$ NIL
  for $i \leftarrow 1, 2, \dots, S$ do
    if $y_i = 1$ then
      $e \leftarrow$ The edge represented by index $i$ in $y$
      if $F = 0$ then
        $R \leftarrow N[\text{parent}(e)]$
        $F \leftarrow 1$
      if left($N[\text{parent}(e)]$) = NIL then
        left($N[\text{parent}(e)]$) $\leftarrow N[\text{child}(e)]$
      else
        right($N[\text{parent}(e)]$) $\leftarrow N[\text{child}(e)]$
```

4.2 Data Generation Process

We could find no existing dataset of hand-drawn binary trees, so we generated our own. This was accomplished in two phases: first, the trees were generated randomly, then they were converted to Graphviz [2] directed graphs and rendered into PNGs, including hand-drawn numbers pulled from the MNIST dataset [9]. As this is our only source of numbers, we limit the nodes to values in the range $\{0, 1, \dots, 9\}$. Due to this, for the remainder of this work we take $M = 10$ and $S = 45$.

The generation uses a traditional binary tree node which tracks its own value, its left child, and its right child. Starting with the root node, we choose a value from a folded normal distribution with $\mu = 0$, $\sigma = 5$, clamp it to the range $\{0, 1, \dots, M-1\}$, and floor it to get an integer value. This increases the chances of smaller numbers being chosen for nodes, thereby increasing the average depth of the trees as the maximum value isn't reached as quickly. We then generate up to two children, each with a chance of .95, and generate values for the ones that exist the same as the root node, clamping instead to $\{minVal, \dots, M-1\}$, where minVal is equal to one plus the current largest value in the tree. The process proceeds recursively, continuing down left children first, and tracking minVal to ensure that the values adhere to the increasing requirement; at each step, the chance of generating a child is reduced by half, to encourage more balanced trees. The edges to child nodes, if any, are added after the recursive call returns. The success of this generation method at producing a variety of differently sized trees is evidenced by figure 2, where it is apparent that the sizes follow a roughly Gaussian distribution.

The second phase requires converting the existing tree into a Graphviz graph, and inserting images for the nodes. This is also accomplished recursively, starting from the root node, and adding the edges and nodes depth-first. As each node is added, an MNIST image is drawn randomly from the set matching the node's value, and inserted into the node. To add further variety to the trees, nodes and edges have a number of randomly chosen parameters. For nodes, the border can have one of four shapes: an ellipse, a circle, a box, or no border. In addition, the height and/or the width of the node may be reduced by half. For edges, the width of the edge may be doubled or cut in half, and the arrow at the end may be present or removed. During the conversion process, the digit labels and the edge set are generated and saved to serve as labels for training and validating. After conversion, the tree is saved as a grayscale PNG, 512 × 512 pixels.

We generated two datasets, Extra Variety and Extra Variety+. For Extra Variety, the node and edge diversity described above is present, but the tree values were far more limited. The trees in this set all have a root node with value 0, were complete trees, and the values increased in increments of 1 with no skipping digits. Extra Variety+ adheres to the generation procedure outlined above in its entirety, allowing root nodes to have any valid value, incomplete trees, and any gap between digits. As an example, the first tree in Figure 1 could belong to either dataset, but the latter two could only belong to Extra Variety+.

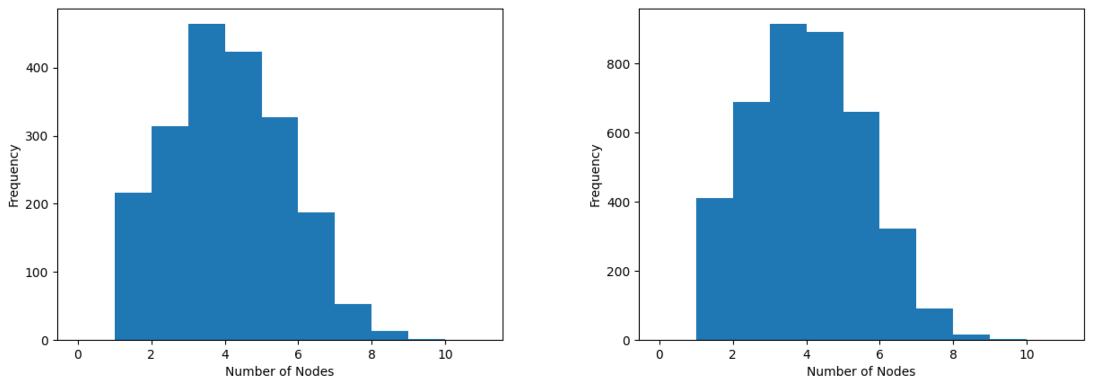

Figure 2: Distribution of Tree Sizes for 2K and 4K Extra Variety+ Datasets

4.3 Modelling Procedure

In order to make edge set predictions for an input image $X \in \mathbb{R}^{H \times W}$ with a transformer T we perform a two stage training process. In the first stage, we train T to predict which digits are present in an image. This is done by having T predict a vector $d \in \{0,1\}^M$. The vector d is defined elementwise as follows.

$$d_i = \mathbb{1}\{i-1 \text{ is present in } X\}$$

This also is a multi-label binary classification problem. After training T to predict d from X, we then train T to predict edge set y from X. Each step of this procedure is described in more detail below.

4.3.1 Preprocessing

We had two components of our preprocessing strategy. One component transformed input images to work well with our ViT model T. Recall from section 4.2 that the images X we generate are grayscale and 512 × 512. Before passing these images into T, we do the following:

1. We convert the generated images to have red, green, and blue channels.
2. We resize the images to 256 × 256.

3. We center crop the images to 224×224.
4. Lastly, we normalize each image to have channel means $[0.485, 0.456, 0.406]$ and channel standard deviations $[0.229, 0.224, 0.225]$.

This set of transformations has been recommended for image data used with both ViT [6] and ResNet architectures [5]. The second component of the preprocessing strategy serves as data augmentation. We noticed that for data sets with more variable binary trees, ViT tends to severely overfit on the training set. This has been noticed previously and Steiner et al. describe various ways to mitigate this issue [17]. One strategy they support is to apply heavy data augmentation to make up for the dataset(s) of interest being small relative to the datasets vision transformers are typically pretrained on. Thus, during the training of T to predict both d and y, we additionally apply the RandAugment transform [8] that Steiner et al. recommend to the training set. The data augmentation component of our preprocessing strategy is not applied to validation and test datasets.

4.3.2 Modifications to Model Architecture

To use the PyTorch ViT architecture [6] to predict digit outputs d we replaced the last dense layer to produce M element outputs. The existing architecture was constructed to make predictions for ImageNet [10] and produced 1000 element outputs. The Dive to Deep Learning ViT architecture allowed you to specify a particular number of output classes when instantiating the architecture. As mentioned previously, we continue training the same model after training it on digits when having it learn to predict edge sets y. For this reason, both trained ViT and Dive to Deep Learning models had to have their architectures modified for the second stage of our training process. In particular, we had to replace the last dense layer in each to produce S element outputs.

As mentioned in the previous section, we found that for more complex data sets the torch ViT overfit drastically. Steiner et al. are in favor of using dropout as a form of regularization to combat this issue [17]. The PyTorch ViT architecture has a dropout layer after patch encoding and then two dropout layers within each encoder block [6] [1]. By default, these layers have a zero percent chance of zeroing weights [11]. Thus, we modified the architecture to increase this chance to ten percent as recommended by Steiner et al.

4.3.3 Training Strategy

Since both types of targets digits d and edge sets y are binary vectors, we used binary cross entropy loss as our loss function for both of these tasks. Denote the loss functions for the two tasks as $L_d(\hat{d}, d)$ and $L_y(\hat{y}, y)$ respectively. Note here $\hat{d}$ and $\hat{y}$ are the raw outputs of our transformer T. As we know that our desired outputs will be either zeros or ones, before computing the loss, the first step is to clamp the outputs of T into $[0, 1]$ using the popular sigmoid function. Fortunately, this is already implemented nicely in a single PyTorch layer [4]. This layer works as follows to compute $L_d(\hat{d}, d)$.

1. For $i = 1, 2, \ldots, M$ we compute the standard cross entropy loss.

$$l_d^{(i)} = d_i \log \sigma(\hat{d}_i) + (1 - d_i) \log(1 - \sigma(\hat{d}_i))$$

2. We then average them together to produce $L_d(\hat{d}, d) \in \mathbb{R}$.

$$L(d, \hat{d}) = \frac{1}{M} \sum_{i=1}^{M} l_d^{(i)}$$

The loss between y and $\hat{y}$ is computed similarly as y is also a binary vector.

Again following the work of Steiner et al. [17], we used the Adam [14] [3] optimizer for model training. We kept our learning rate λ to be 0.0001 as we found that higher learning rates caused our models to fail to learn properly. Steiner et al., in addition to discussing dropout and data augmentation as strategies for reducing transformer overfitting, also encourage the use of weight decay [17] in Adam. However, we found that this tended to yield noticeably worse performance on training and did not resolve any overfitting issues, so we kept this at zero. We let our models train for 100 epochs as we found that this was typically adequate for determining if a model had (i) converged to a low loss and/or high validation accuracy or (ii) was unable to learn further. To further combat overfitting, we saved the model during training when it reached its maximum validation accuracy. If there were multiple epochs with the same highest validation accuracy, the model at the epoch with the lower training loss was saved.

Algorithm 2 Modification of Prim's Algorithm to Construct Edge Set

```
Require: Graph $G$ weighted with $w$ as described above. Vertices $v$ can be uniquely identified by
  their values. Let $w(u,v)$ denote the weight of the edge from vertex $u$ to vertex $v$.
Ensure: Edge set $\hat{y}$ will store the minimum spanning tree (MST) of $G$.
  $P \leftarrow$ table mapping $v$ to the weight of the edge connecting $v$ to the MST and its predecessor
  $\hat{y} \leftarrow \mathbf{0} \in \mathbb{R}^S$
  for node $v \in G$ do
    dist($P[v]$) $\leftarrow \infty$
    pred($P[v]$) $\leftarrow$ NIL
  $m \leftarrow$ minimum of vertices
  dist($P[m]$) $\leftarrow 0$
  pred($P[m]$) $\leftarrow$ NIL
  while size($P$) $> 0$ do
    $u \leftarrow$ node with minimum distance in $P$
    $u_p \leftarrow$ pred($P[u]$)
    Remove $u$ from $P$
    if $u_p \neq$ NIL then
      $i \leftarrow$ index corresponding to edge from $u_p$ to $u$
      $\hat{y}_i \leftarrow 1$
    for each neighbor $v$ of $u$ do
      if $v \in P$ and $w(u,v) <$ dist($P[v]$) then
        dist($P[v]$) $\leftarrow w(u,v)$
        pred($P[v]$) $\leftarrow u$
```

4.3.4 Prediction Strategy

Recall that T is first trained to predict digit existence labels d. Denote this trained transformer T_d. Let T_y denote the transformer trained starting from T_d to predict edge sets y. Let X' be the preprocessed version of X. By preprocessed here we refer to the non data augmentation component of preprocessing described in section 4.3.1.

We make a prediction $\hat{d} \in \{0,1\}^M$ using T_d as follows for $i = 1, \ldots, M$.

$$\hat{d}_i = \mathbb{1}\{\sigma(T_d(X'))_i \geq 0.5\}$$

That is, we first collapse the raw outputs of T_d into $[0,1]$. Then, we apply a simple thresholding strategy to classify digit $i-1$ as being present or not. We believe this is an adequate approach as the problem of identifying the presence of digits in an image is an easier problem than constructing the tree in an image.

We make a prediction $\hat{y}$ for preprocessed image X' using T_y as follows.

1. Construct a complete acyclic undirected graph G that has nodes with digit values i for which $\hat{d}_i = 1$. By constructing a complete graph, we will consider the possibility of each node being connected to all of the other nodes. Below, we will identify which nodes a node is most likely be connected to from what T_y has learned. This is taking the advantage of the fact that we know all the possible answers (via the complete) graph to this problem.
2. Compute $w \in [0,1]^S$ as follows.

$$w_i = 1 - \sigma(T_y(X'))_i$$

3. Set the weights of the edges in G using their respective entries in w. Note that from the definition of an edge set in section 4.1 that we have entries for each edge in G. For instance, the edge in G going from node with value zero to value one would get w_1.
4. Use algorithm 2, a modified version of Prim's algorithm [7], to derive the minimum spanning tree of G and then set $\hat{y}$ to be the edge set of said tree.

Hence, $\hat{y}$ is constructed as the most likely tree to span the digits identified by T_d where "likelihood" is identified from T_y.

5 Experiments

We conducted two sets of experiments. In the first set of experiments, we evaluated various modelling strategies on the easier Extra Variety dataset. In the second set, we worked with the Extra Variety+ dataset using the modelling strategy from section 4.3.

5.1 Extra Variety Experiments

Our Extra Variety dataset has three subsets: extra_variety_4k which was our 4000 image training set, extra_variety_2k which was our 2000 image validation set, and extra_variety_1k which was our 1000 image test set for final evaluation. To better understand the importance of the various components of our strategy, we tested three different types of models for predicting digit labels.

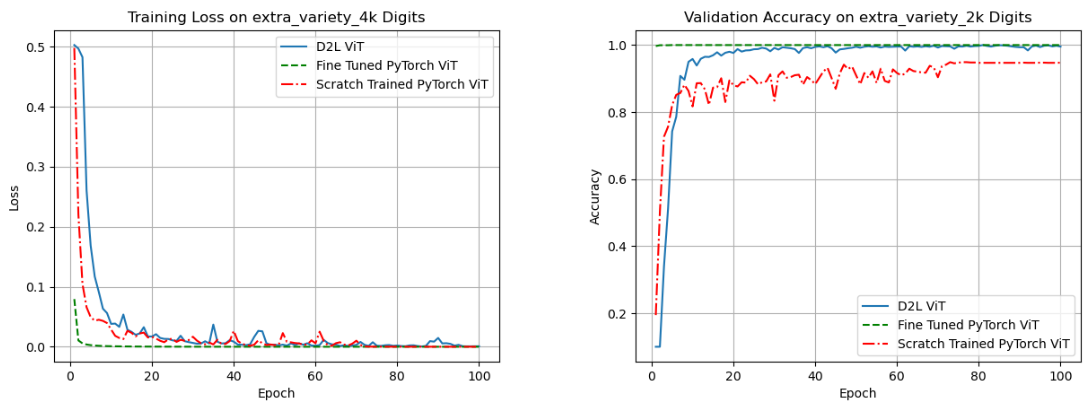

Figure 3: Extra Variety Digit Prediction Performance Comparison

D2L ViT is the ViT architecture taken from Dive to Deep Learning modified as described in section 4.3.2. Fine-Tuned PyTorch ViT is the PyTorch ViT architecture that was loaded with pretrained weights taken from training on ImageNet1K [6] and then fine-tuned on extra_variety_4k. Scratch Trained PyTorch ViT is the PyTorch ViT architecture that was loaded with untrained weights and purely trained from scratch on extra_variety_4k. Note neither of the PyTorch ViT architectures used in this set of experiments have nonzero probability dropout layers nor did we perform data augmentation on the training set.

Figure 3 shows the loss and accuracy for these three models on predicting digit labels on the training and validation sets respectively. First, we can see that all three transformers learned to predict these datasets very quickly, well within twenty epochs. It is not surprising that Fine-Tuned PyTorch ViT outperformed Scratch Trained PyTorch ViT, but it is surprising that the smaller D2L ViT architecture outperformed it as well. We believe that this could be caused by Extra Variety being an easier dataset and that D2L ViT has significantly less parameters and may be easier to train. Table 1 shows the accuracies of these three models on predicting digit labels on extra_variety_1k. Here we can see that all three models perform quite well, with Scratch Trained PyTorch ViT generalizing the worst.

Figure 4 shows the loss and accuracy of the same models on predicting edge sets on the same two datasets. All three models converge to high performance even faster: this time within ten epochs. Again, D2L ViT and Fine-Tuned PyTorch ViT outperformed Scratch Trained PyTorch ViT. While these results are very promising, we believe that this is most likely due to the Extra Variety data set being simpler and having less variability in its trees. That is, since the types of trees are more limited than Extra Variety+, it is possible these models had just memorized the set of possible trees. We believe that the other aspects of the drawings, such as the shapes of nodes and edges, have less impact.

Model	**Accuracy**
D2L ViT	0.998
Fine-Tuned PyTorch ViT	**1.000**
Scratch Trained PyTorch ViT	0.933

Table 1: Model Digit Accuracies on Extra Variety Evaluation Set

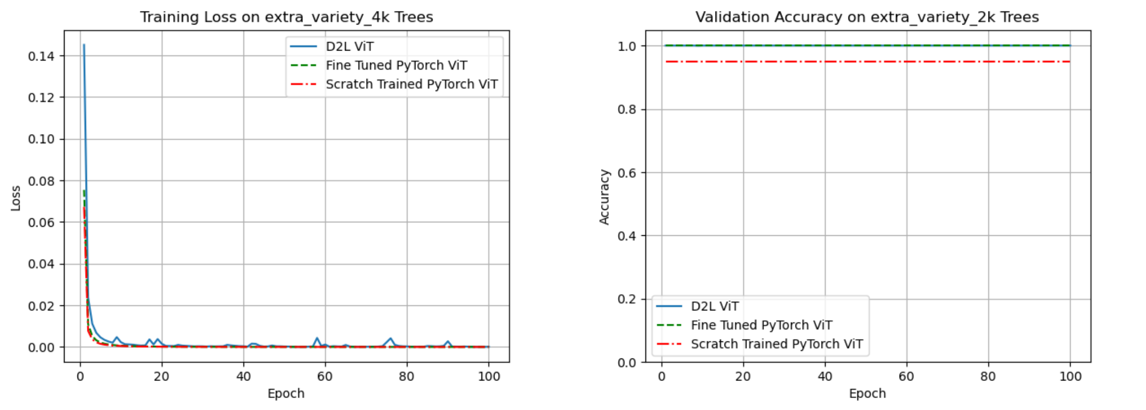

Figure 4: Extra Variety Tree Prediction Performance Comparison

Since our strategy from section 4.3 performed very well in these experiments, we wanted to better understand the importance of the various components of the strategy. The first thing we experimented with was not using the prediction strategy in section 4.3.4. That is, we had our transformers train on digits, then on trees, but make predictions using a thresholding strategy similar to the one used for digits. That is, for a transformer T_y, its predictions $\hat{y}$ for a preprocessed image X' are calculated as follows.

$$\hat{y}_i = \mathbb{1}\{\sigma(T_y(X'))_i \geq 0.5\}$$

For $i = 1, 2, \ldots, S$. The second aspect of the strategy we experimented with was not having the initial training phase on predicting digit labels. Instead, have an untrained (or pretrained transformer in the case of Fine-Tuned PyTorch ViT) T be fed in the images once and only learn to predict edge sets. This is opposed to the training strategy in section 4.3.3 where models see the training set twice: first to learn to predict digit labels and then to predict edge sets. Surprisingly, neither of these modifications had noticeable impact on training and prediction using Fine-Tuned PyTorch ViT. It converged to zero loss and a one hundred percent validation accuracy at a fast rate similar to before.

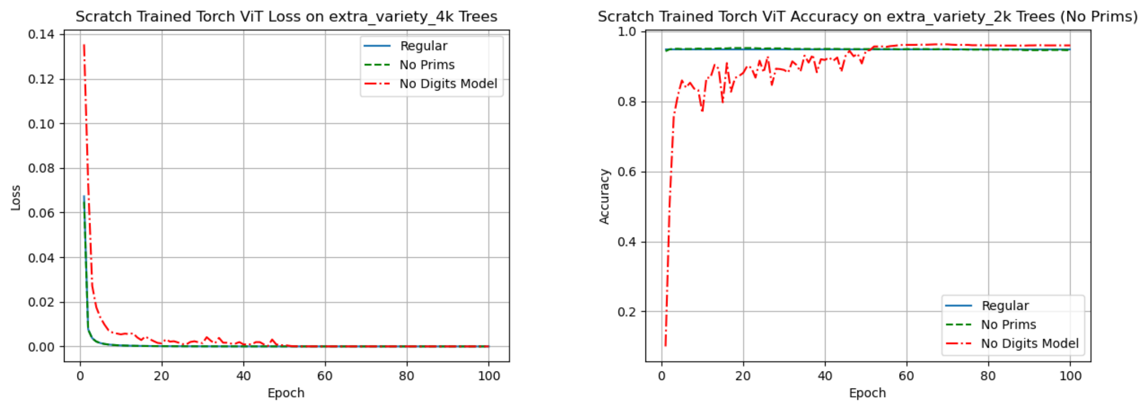

Figure 5: Scratch Trained PyTorch ViT Performance with Missing Components

However, this was not entirely the case with D2L ViT or Scratch Trained PyTorch ViT. For the Scratched Trained PyTorch ViT, figure 5 compares the training and prediction strategies from sections 4.3.3 and 4.3.4 (regular) with training from the digits model but not using the section 4.3.4 prediction strategy (no prims) and training from scratch never learning to predict digits (no digits model).

As we observed with the Fine-Tuned PyTorch ViT, we can see that the use of the section 4.3.4 prediction strategy does not cause the performance to degrade. This implies that the predicted outputs $\sigma(T_y(X'))_i$ (see section 4.3.4) are very close to zero for edges that are not present and very close to one that edges that are. However, figure 5 does tell us that without training from a model that has learned to predict digits, it takes longer to learn to predict edge sets. This does not strike us as surprising. However, it is interesting to see in the validation accuracy plot, that training to learn edge sets without digits first results in slightly higher validation accuracy. This indicated to us that it is likely a model's ability to learn digit labels is closely tied to its ability to learn edge sets if the model continues training from the digit model. We observed similar behavior for both "no prims" and "no digits model" setups for the D2L ViT.

Table 2 shows the results of using nine different modelling strategies to predict edge sets for the Extra Variety test set. The results generally follow the performance we observed on the validation set. Fine-Tuned PyTorch ViT performed the best consistently and was the architecture that was least dependent on the overall modelling strategy. D2L ViT's performance was also more consistent here. The most interesting result is that Scratch Trained PyTorch ViT performed the best when it was trained from scratch to learn edge sets, completely deviating from our strategies in sections 4.3.3 and 4.3.4. However, these results may be somewhat misleading in the sense they may not indicate that these models generalize well. This is discussed more in the following section.

5.2 Extra Variety+ Experiments

For our next set of experiments, we worked with the far more complex Extra Variety+ dataset. Our Extra Variety+ dataset has three subsets: extra_variety+_4k which was our 4000 image training set, extra_variety+_1k which was our 1000 image validation set, and extra_variety+_2k which was our 2000 image test set for final evaluation. First, we were curious to see if the models we evaluated in the previous section generalized well to this new set of tree drawings. We had the

Model	**Accuracy**
D2L ViT	0.998
D2L ViT (no prims)	1.000
D2L ViT (no digits model)	0.990
Fine-Tuned PyTorch ViT	**1.000**
Fine-Tuned PyTorch ViT (no prims)	**1.000**
Fine-Tuned PyTorch ViT (no digits model)	**1.000**
Scratch Trained PyTorch ViT	0.933
Scratch Trained PyTorch ViT (no prims)	0.933
Scratch Trained PyTorch ViT (no digits model)	0.958

Table 2: Model Tree Accuracies on Extra Variety Evaluation Set

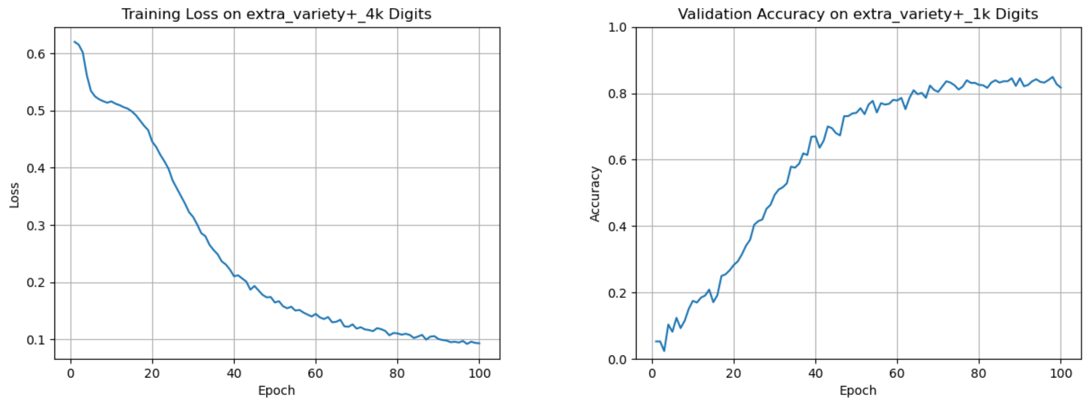

Figure 6: Extra Variety+ Digit Prediction Performance Comparison

nine trained models listed in table 2, as well as their corresponding digit models where appropriate, make edge set predictions for extra_variety+_1k. All of the models performed extremely poorly, getting at most ten to twelve percent accuracy. The trees that they tended to predict properly were trees that could have existed in the Extra Variety dataset. This indicated to us that these models were performing so well previously just because they had "memorized" the tree drawings due to there being too low tree variability in Extra Variety.

In our initial experiments, we used our training and prediction strategies from sections 4.3.3 and 4.3.4 to learn digits and edge sets for the Extra Variety+ dataset. However, we found that it performed very poorly in both tasks. Therefore, we decided that the task of identifying digits and edge sets in Extra Variety+ was a substantially more difficult problem. We decided to work with only Fine-Tuned PyTorch ViT for the remainder of modelling experiments on Extra Variety+. We did this for two reasons: (i) it performed consistently the best on the easier Extra Variety task and (ii) starting with a pretrained transformer is more typical in computer vision [21].

We initially observed that the Fine-Tuned PyTorch ViT seemed to overfit on digit labelling predictions. We believe that it may not have overfit in the Extra Variety dataset because it may have memorized where certain digits could appear in the tree images. Note, as mentioned in section 3, learning locations and distances is something the developers of ViT believe it does well. Thus, it only appeared as if the model was generalizing well. For future experiments we used the preprocessing strategy that includes data augmentation described in section 4.3.1 as well as dropout layers as described in section 4.3.2.

Figure 6 shows the loss and accuracy on the Extra Variety+ training and validation sets of using Fine-Tuned PyTorch ViT to predict digit labels. The slower convergence of training loss demonstrates that this is indeed a harder task for the model to learn. We analyzed the images for which the transformer failed to identify the digits properly, but we did not find a particular pattern worth reporting. Figure 7 shows the model's loss and accuracy of predicting edge sets on the same pair of datasets. Here, we used exactly the modelling procedure detailed in section 4.3. Note that the convergence of both training loss and validation accuracy is faster than on digits.

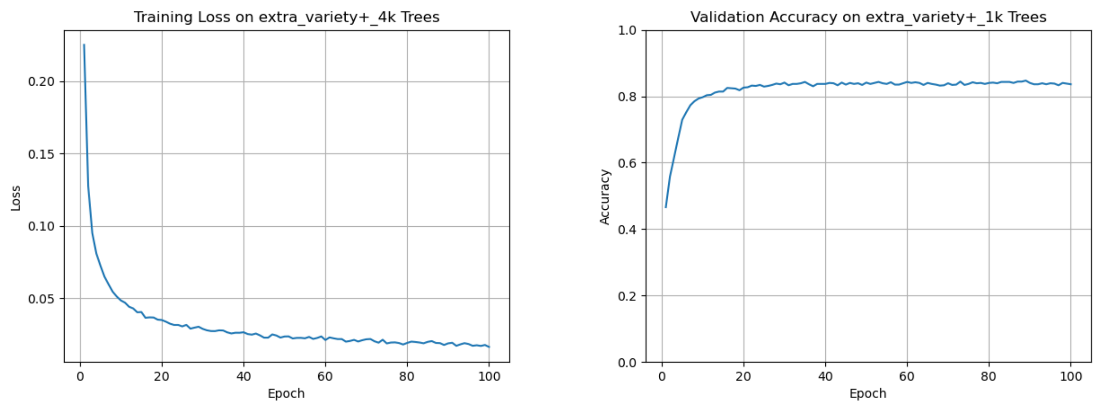

Figure 7: Extra Variety+ Tree Prediction Performance Comparison

Table 3 shows the accuracy on predicting digit labels and edge sets on the Extra Variety+ train, validation, and test sets. While the model performs the best on the training set, the performance on the validation and test sets is not far behind which indicates that the various strategies we used from Steiner et al. [17] were beneficial in preventing overfitting. It is also interesting to observe that the edge set accuracy is closely tied to the digit accuracy. This, in addition to the training loss for edge set learning converging faster, makes us wonder how much the model is learning from the second stage of our procedure.

Subset	**Digit Accuracy**	**Tree Accuracy**
extra_variety+_4k (train)	0.8723	0.8725
extra_variety+_1k (validation)	0.8490	0.8470
extra_variety+_2k (test)	0.8545	0.8485

Table 3: Fine-tuned Torch ViT Extra Variety+ Performance

6 Conclusion

For an easier tree dataset such as Extra Variety we believe that existing transformer architectures should have no trouble identifying the corresponding tree edge sets. However, we believe this may be more due to transformers memorizing a relatively small amount of trees as opposed to learning the structure of trees. For more realistic datasets, such Extra Variety+, our approach shows decent performance. However, we believe it is possible that with better hyperparameter tuning and access to better hardware, our approach could yield even higher accuracy than the 0.84 to 0.87 range listed in table 3. For instance, it is possible, based on the training loss in figure 6, that the model could learn more. Nonetheless, we believe that our approach has been successful in providing a proof of concept of using a transformer architecture to construct mathematical representations of binary trees from images of tree drawings.

Our code for data generation, modelling procedure, and training is available at `https://github.com/ChamiLamelas/Tufts-CS137-FinalProject`.

7 Contributions

- Writing the proposal - Chami 90% Ryan 10%
- Coding - Chami 60% Ryan 40%
- Running and coding experiments - Chami 90% Ryan 10%
- Collecting data - Chami 30% Ryan 70%
- Discussions (of project ideas and experiment results) - Chami 50% Ryan 50%
- Writing the final report - Chami 70% Ryan 30%

References

[1] Dive to deep learning vit. `https://d2l.ai/chapter_attention-mechanisms-and-transformers/vision-transformer.html`. Accessed: 2022-12-13.

[2] Graphviz documentation. `https://graphviz.org/`. Accessed: 2022-12-13.

[3] Pytorch adam optimizer. `https://pytorch.org/docs/stable/generated/torch.optim.Adam.html`. Accessed: 2022-12-13.

[4] Pytorch binary cross entropy loss. `https://pytorch.org/docs/stable/generated/torch.nn.BCEWithLogitsLoss.html`. Accessed: 2022-12-13.

[5] Pytorch resnet model. `https://pytorch.org/hub/pytorch_vision_resnet/`. Accessed: 2022-12-13.

[6] Pytorch vit_b_16 model. `https://pytorch.org/vision/main/models/generated/torchvision.models.vit_b_16.html`. Accessed: 2022-12-13.

[7] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2022.

[8] Ekin D Cubuk, Barret Zoph, Jonathon Shlens, and Quoc V Le. Randaugment: Practical automated data augmentation with a reduced search space. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition workshops*, pages 702–703, 2020.

[9] Li Deng. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

[10] Jia Deng et. al. Imagenet: a large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[11] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

[12] Yuxin Fang, Bencheng Liao, Xinggang Wang, Jiemin Fang, Jiyang Qi, Rui Wu, Jianwei Niu, and Wenyu Liu. You only look at one sequence: Rethinking transformer in vision through object detection. *Advances in Neural Information Processing Systems*, 34:26183–26197, 2021.

[13] Salman Khan, Muzammal Naseer, Munawar Hayat, Syed Waqas Zamir, Fahad Shahbaz Khan, and Mubarak Shah. Transformers in vision: A survey. *ACM computing surveys (CSUR)*, 54(10s):1–41, 2022.

[14] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[15] Adam Paszke et. al. Pytorch: an imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[16] Arjun Singh, Sergey Karayev, Kevin Gutowski, and Pieter Abbeel. Gradescope: a fast, flexible, and fair system for scalable assessment of handwritten work. In *Proceedings of the fourth (2017) acm conference on learning@ scale*, pages 81–88, 2017.

[17] Andreas Steiner, Alexander Kolesnikov, Xiaohua Zhai, Ross Wightman, Jakob Uszkoreit, and Lucas Beyer. How to train your vit? data, augmentation, and regularization in vision transformers. *arXiv preprint arXiv:2106.10270*, 2021.

[18] Grigorios Tsoumakas, Eleftherios Spyromitros-Xioufis, Jozef Vilcek, and Ioannis Vlahavas. Mulan: A java library for multi-label learning. *The Journal of Machine Learning Research*, 12:2411–2414, 2011.

[19] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[20] Tao Wang, Xuming He, Yuanzheng Cai, and Guobao Xiao. Learning a layout transfer network for context aware object detection. *IEEE Transactions on Intelligent Transportation Systems*, 21(10):4209–4224, 2020.

[21] Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola. Dive into deep learning. *arXiv preprint arXiv:2106.11342*, 2021.