Computer Vision Model for Horse Detection in a Custom Dataset

Richard Lopez, Ganesh Krishnan, Rodrigo Burberg

AAI-521-03-FA22 - Intro to Computer Vision

Abstract

This project aims to develop a system that can accurately identify horses in images with minimal false positives. In this work, we propose creating a Computer Vision model for a horse detection system. Our model will use a Vision Transformer (ViT) to detect the features of a horse. We are motivated to prove this horse detection system as a base for solving future related problems in scoring and monitoring live horse events like rodeos. Using proven Object Detection techniques and algorithms, a custom dataset was taken and used to build a model to detect horses. Object detection is done by two steps called training and inference. In the training step, we feed our deep learning architecture with training images and the corresponding coordinates for box annotation and train our model until it minimizes the validation loss. The output is a trained model. In the inference step, this learned model is used for detecting horse objects in new images. The object detector outputs the coordinates where it has detected a horse and provides confidence scores associated with those detections. This project will be beneficial for identifying horse events and could be used in other scenarios, such as surveillance or animal tracking. Additionally, this project sets a foundation for future research using Computer Vision and object detection with video for live horsing events.

Introduction

The modern AI era of computer science has fueled the development of tools that help researchers annotate and classify images and videos. In the field of Computer Vision (CV), which focuses on understanding digital images and videos, the development of object detection algorithms has been instrumental in creating powerful image analysis frameworks capable of recognizing objects from still images or video frames. While plenty of CV work has been done for self-driving cars, health care, and sports like football and basketball, the application of these object detection algorithms to live horse events has yet to be noticed. This project's scope is to

have a Computer Vision model that can identify a horse in an image. The hope is that this work can serve as the basis for future iterations that can automatically assist in scoring and monitoring live horse events such as rodeos.

The state-of-the-art object detection systems use convolutional neural networks (CNNs) or a deep-learning model. We use a Vision Transformer (ViT) for our object detection system. ViT is a relatively new approach that uses self-attention mechanisms and transformer layers instead of the convolutional layers used in CNNs. This method can provide more accurate results than traditional CNNs, and is better suited for identifying horses in digital media during live events where on-site computational resources may be limited. ViT models have been proven to outperform CNN-based models by almost 30% in terms of computational efficiency and accuracy (Sagar, 2021).

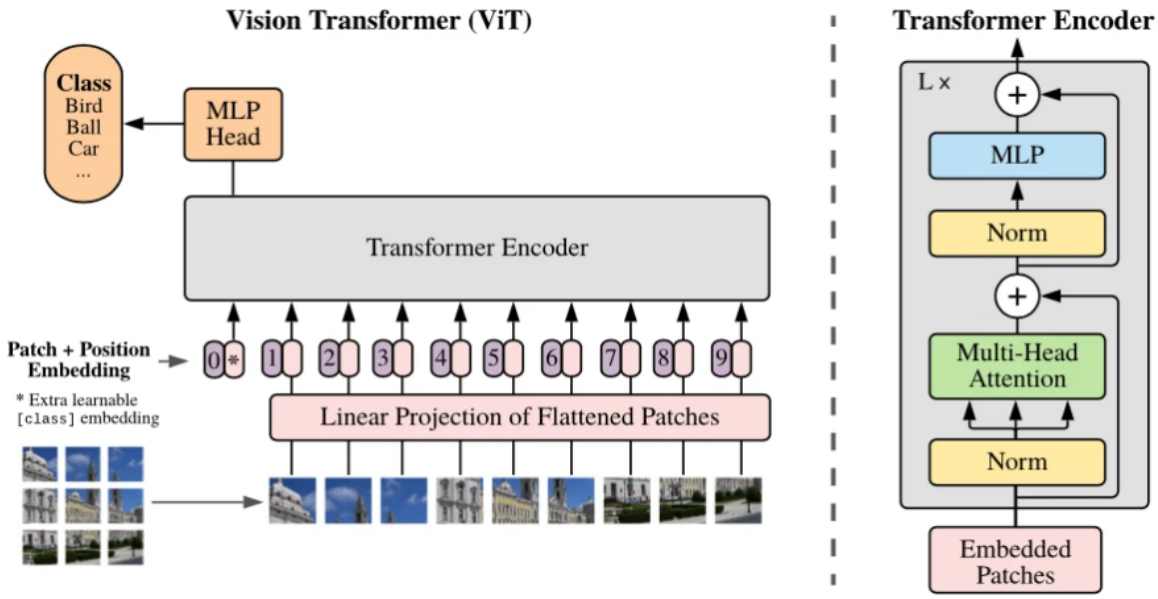


Figure 1*(Boesch, 2022)*

Problem Statement

In rodeo events, a human is often responsible for parsing a set of images and categorizing the image by event type. This process looks at the animals and humans in the image and decides the event type. We are tackling a subproblem, identifying horses in images, with object detection techniques for this project. In future work, we can prepare a more extensive dataset and identify multiple animal and object types. For example, we can detect a human, two horses, a calf, and a rope. Again, in this project, we focus on the first step of using object detection to identify horses in an image. Furthermore, we could apply object detection to video from these live rodeo events. The following section describes the custom dataset used for this work.

Dataset

Our project uses a custom dataset of over 50 horse images. We collected this dataset by downloading publicly available images from the Animal Image Dataset on Kaggle. (Banerjee, 2022). We then added nine images from American Rodeos from Google Images. The dimensions of the photos range from 1024 x 548 to 3071 x 3072 in jpeg format. One critical early step for building a ViT is loading images with corresponding annotations. Annotation labels the image or video frame with its class, bounding boxes, and other attributes. For our project, we used RoboFlow to annotate each horse in the dataset. Some alternative tools we considered include LabelImg and MatLab.

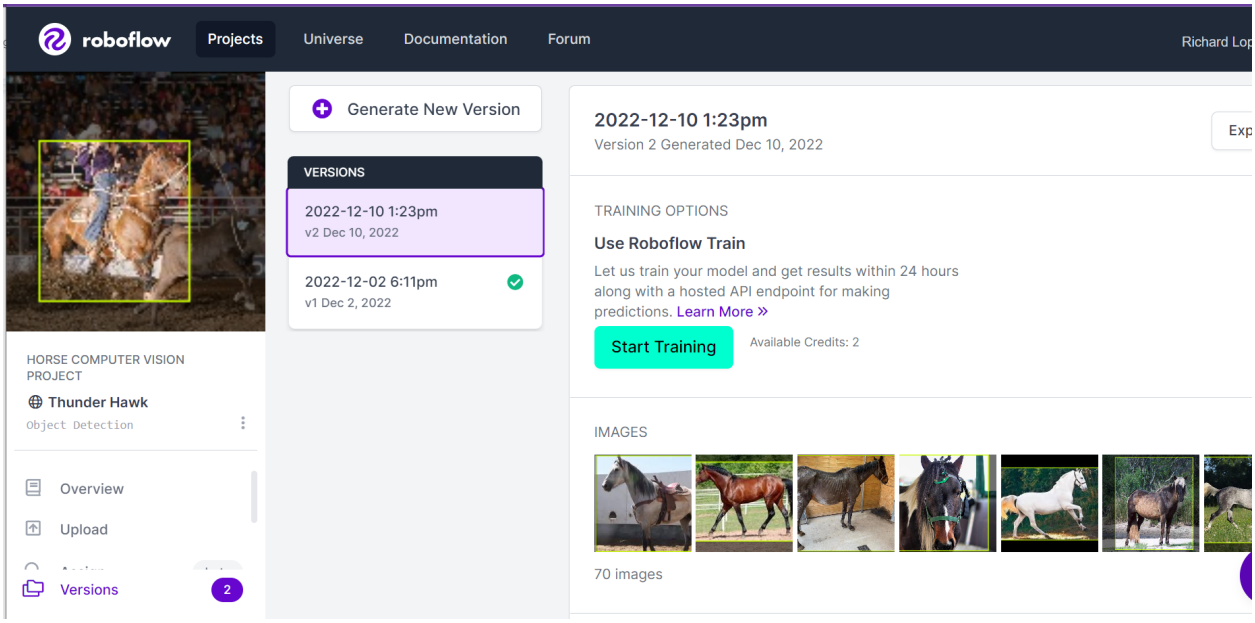


Figure 2 (Roboflow: Give Your Software the Power to See Objects in Images and Video, 2022)

To prepare the dataset for our model, we split it into a training set of 46 images and a test set of 12 images. Here we also reshape and optimize the training images by reducing the image size to 224 x 224 using the *tf.image.resize* method. When resizing training images, it is crucial to consider resized images will be distorted if their original aspect ratio is not the same as the size. Using the annotations created in RoboFlow, we apply relative scaling to bounding boxes as per the given image and append to a list of images and the corresponding bounding box coordinates.

Methodology

Clearly, object detection and image recognition have a proven history with a CNN. Popular Image recognition algorithms include VGG, YOLOv3, and YOLOv7. These algorithms depend heavily on convolution layers and require high computing power. It is apparent to us that there has been an increase in interest in Vision Transformers (ViT) and Multilayer Perceptrons

(MLPs). MLPs use traditional neural networks and are capable of recognizing objects without the need for convolutional layers. ViT combines self-attention mechanisms with transformer layers instead of relying on ConvNets (*Vision Transformers (ViT) Explained | Pinecone*, 2017). Given the proposed efficiencies of a ViT architecture, we chose to implement a Vision Transformer (ViT) architecture for this project.

Develop and Train the Model

With our training and test dataset prepared, we developed our model. We begin this process by building a network using the MLP network and with a layer separating our images into patches. Specifically, our MLP creator method adds a dense dropout layer to our MLP objects. With our MLP method created, we build a patch encoder that will perform the linear transformation of the image patches. Here we can see a sample image and the corresponding patched image. With our patching method confirmed, we build a patch encoder class. The patched image is created using the Patches class we created.

```
# Implement multilayer-perceptron (MLP)
def mlp(x, hidden_units, dropout_rate):
    for units in hidden_units:
        x = layers.Dense(units, activation=tf.nn.gelu)(x)
        x = layers.Dropout(dropout_rate)(x)
    return x
```

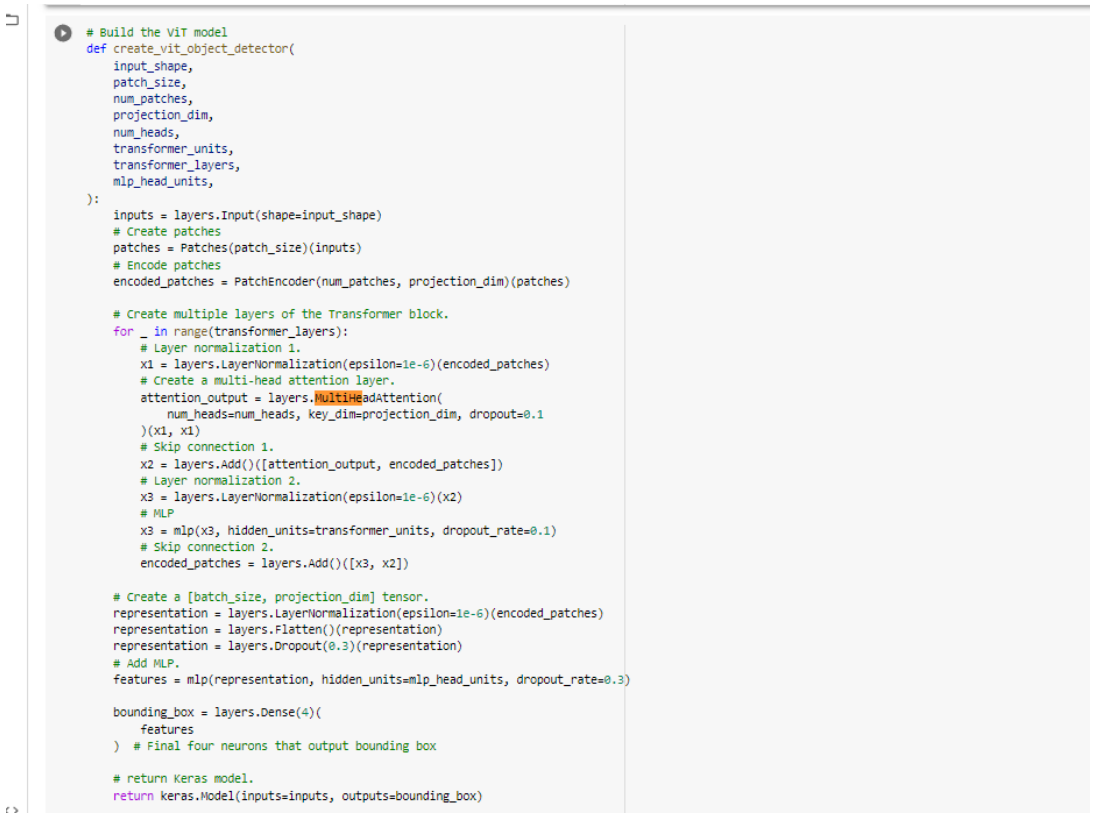
```
# Implement the patch creation layer
class Patches(layers.Layer):
    def __init__(self, patch_size):
        super(Patches, self).__init__()
        self.patch_size = patch_size

    #    Override function to avoid error while saving model
    def get_config(self):
        config = super().get_config().copy()
        config.update(
            {
                "input_shape": input_shape,
                "patch_size": patch_size,
                "num_patches": num_patches,
                "projection_dim": projection_dim,
                "num_heads": num_heads,
                "transformer_units": transformer_units,
                "transformer_layers": transformer_layers,
                "mlp_head_units": mlp_head_units,
            }
        )
        return config

    def call(self, images):
        batch_size = tf.shape(images)[0]
        patches = tf.image.extract_patches(
            images=images,
            sizes=[1, self.patch_size, self.patch_size, 1],
            strides=[1, self.patch_size, self.patch_size, 1],
            rates=[1, 1, 1, 1],
            padding="VALID",
        )
        # return patches
        return tf.reshape(patches, [batch_size, -1, patches.shape[-1]])
```

Figure 3

The patch encoder is a transformer layer responsible for encoding our image patches with a linear transformation (Editor, 2021). This helps to process the data faster while incorporating the self-attention mechanism of ViTs. This network will power our Vision Transformer object detection model. The input data will pass through our patch maker block, and the data will go through the patch encoder. The MultiHeadAttention layer, shown below, is used for the self-attention we mentioned above. It's applied to the image patches (*Tf.keras.layers.MultiHeadAttention | TensorFlow V2.11.0*, 2022). These now encoded patches, along with the self-attention layer output, are normalized and passed into an instance of our MLP creator method. At this point, the model outputs four dimensions representing the bounding box data of a detected object.



```
# Build the ViT model
def create_vit_object_detector(
    input_shape,
    patch_size,
    num_patches,
    projection_dim,
    num_heads,
    transformer_units,
    transformer_layers,
    mlp_head_units,
):
    inputs = layers.Input(shape=input_shape)
    # Create patches
    patches = Patches(patch_size)(inputs)
    # Encode patches
    encoded_patches = PatchEncoder(num_patches, projection_dim)(patches)

    # Create multiple layers of the Transformer block.
    for _ in range(transformer_layers):
        # Layer normalization 1.
        x1 = layers.LayerNormalization(epsilon=1e-6)(encoded_patches)
        # Create a multi-head attention layer.
        attention_output = layers.MultiHeadAttention(
            num_heads=num_heads, key_dim=projection_dim, dropout=0.1
        )(x1, x1)
        # Skip connection 1.
        x2 = layers.Add()([attention_output, encoded_patches])
        # Layer normalization 2.
        x3 = layers.LayerNormalization(epsilon=1e-6)(x2)
        # MLP
        x3 = mlp(x3, hidden_units=transformer_units, dropout_rate=0.1)
        # Skip connection 2.
        encoded_patches = layers.Add()([x3, x2])

    # Create a [batch_size, projection_dim] tensor.
    representation = layers.LayerNormalization(epsilon=1e-6)(encoded_patches)
    representation = layers.Flatten()(representation)
    representation = layers.Dropout(0.3)(representation)
    # Add MLP.
    features = mlp(representation, hidden_units=mlp_head_units, dropout_rate=0.3)

    bounding_box = layers.Dense(4)(
        features
    )  # Final four neurons that output bounding box

    # return Keras model.
    return keras.Model(inputs=inputs, outputs=bounding_box)
```

Figure 4

"In the end, this architecture delivers "super information-rich embeddings. These embeddings are the ultimate output of the core of a transformer, including ViT." (Vision Transformers (ViT) Explained | Pinecone, 2017)

Finally, after completing our model, we proceed to the fun part – training! We create our *run_experiment* method to set up the compilation of our model. Our model is compiled using the AdamW optimizer, an improved implementation of the Adam optimizer, and the MeanSquareError loss function. (tfa.optimizers.AdamW, 2022) A couple of other notable features to highlight in our compiler setup is the use of the keras.fit function to train our deep learning models. It takes the x and y train to supply each image with its corresponding annotation. We use the *validation_split* parameter, which allows randomly splitting a subset of the training data into cross-validation. Cross-validation is utilized throughout the training process to ensure our model does not overfit. We set the cross-validation extraction to 10%.

"In computer vision, overfitting is a phenomenon that occurs when a machine learning algorithm begins to memorize the training data rather than learning the underlying patterns. This can lead to poor performance on new data, as the algorithm is not able to generalize from the training data to other datasets." (Boesch, 2022)

Finally, in our model compiler, we use the *EarlyStopping* method to monitor the loss value and stop the training early if it detects overfitting after our patience threshold. We can generate this at any time in our model creation by calling the *summary* method we defined. Here is a visual of our overall Vision Transformer Model architecture (See Appendix for full picture).

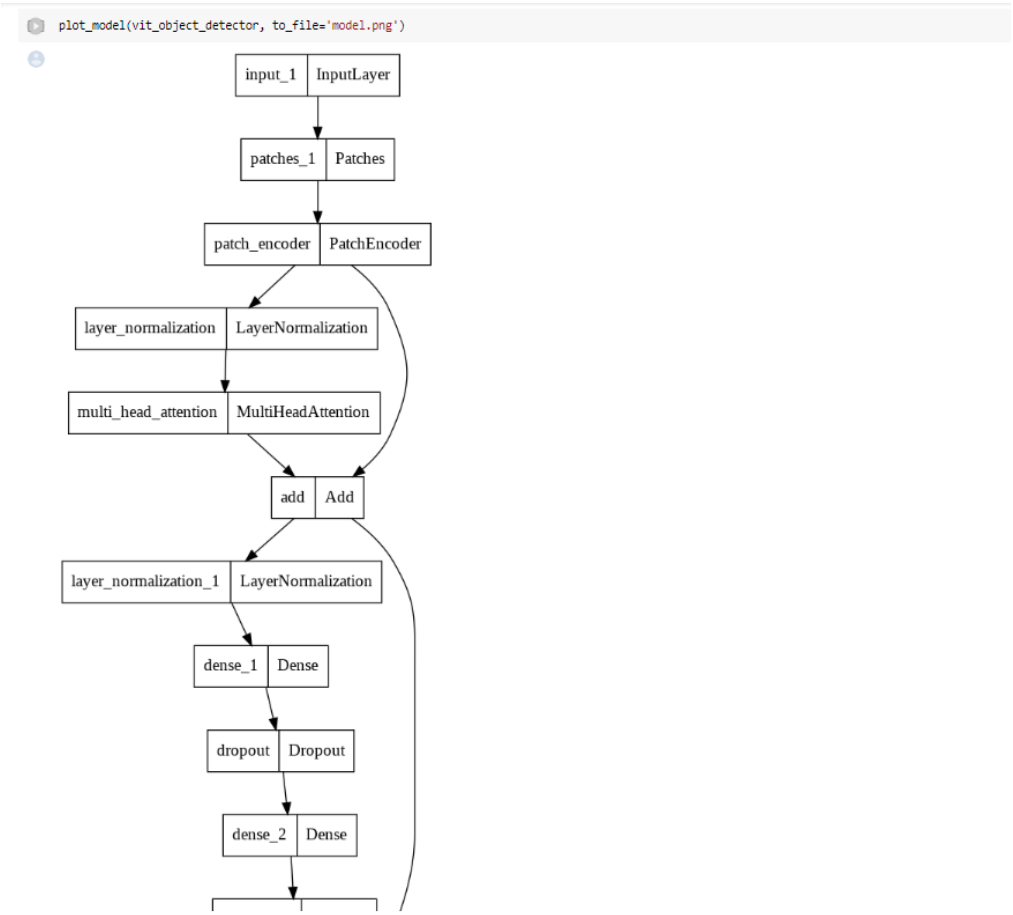


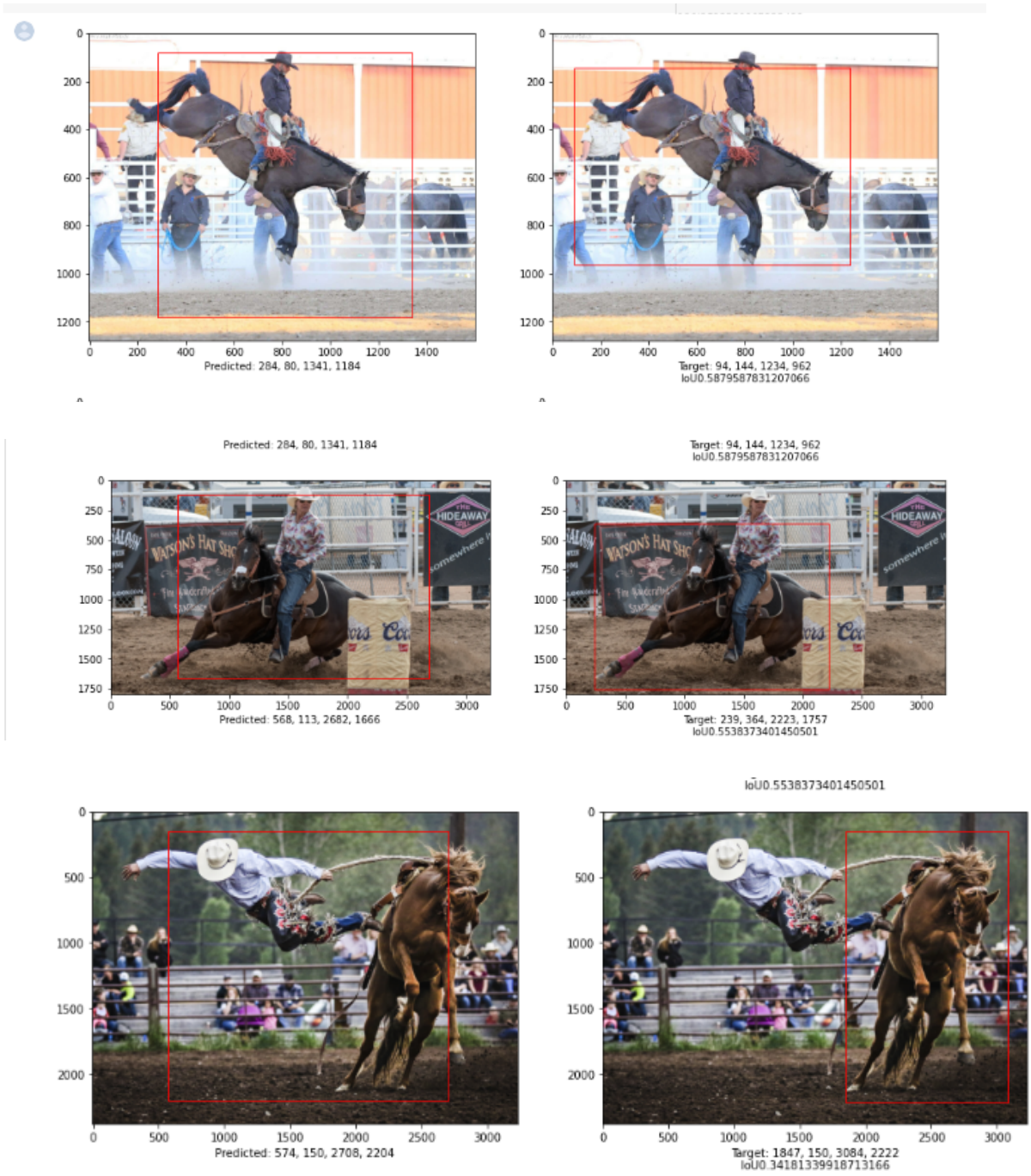
Figure 5

Our compiler was set up to experiment with different values for *num_epoch, batch_size, weight_decay, and learning_rate*. This allows us to easily experiment with different values and find the best setup for our use case. Once we ran our compiler it created an instance of our model. As expected we watched the loss decrease with every epoch. This was an important validation for future iterations where we plan to increase the number of epochs. For this project, our training data is relatively small, so the wait time was not much. We expect this process to take longer with a more extensive training dataset in future iterations. With our model trained, we began using it for object detection. We used Keras's *predict* method to feed test data into our model and get a prediction for each image in an array containing the four dimensions.

Results

With our object detector created and trained, we focused on evaluating the accuracy of our model. Intersection over union (IoU) and Average Precision (AP) are commonly used metrics to measure an object detector's performance. IoU measures the overlap between two bounding boxes, while AP measures the precision of a model's detections with respect to ground truth annotations. With our object detector created and trained, we focused on evaluating the accuracy of our model. It was understood that this project was likely to experience fluctuations in IoU since our annotations only took the horse into account and not the athlete who was present in most of them. Moreover, our dataset size was quite limited. Also, considering the fact that only one class was used (horses), AP metrics wouldn’t provide any additional insight, so we focused on IoU metrics. In later iterations, we anticipate improved results when we introduce annotation for other segments, such as athletes.

We passed images to our model using 20% of the data we reserved for validation. The Results of our ViT Model demonstrate that it can accurately detect horse objects with a mean IoU score of 0.532. These results show that the model can detect the single-horse object class from images without prior knowledge about their content. The images below show that our object detector detected the horses in most test images. A key takeaway is that, given our architecture, we do not need to keep training our model. Once we have a model tuned well, we can continue to use that model repeatedly for as many predictions as we want.



Conclusion and Future Work

In this paper, we presented a proof-of-concept project that can use Vision Transformer (ViT) object detection to identify horses in images. We prepared a custom dataset and used the annotation tool RoboFlow to annotate each image with bounding boxes around the horse

instances. The ViT model was then trained to detect horses. Our model achieved an average IoU of 0.532 on our test set, demonstrating that our model can detect single-horse objects with an acceptable IoU score.

There is still much room for improvement with this project, namely the ability to detect other object types related to the use case. For example, in a rodeo, we would like to detect athletes, bulls, ropes, barrels, and other objects specific to event types. In addition, we would like to apply our model to video. We could develop a more robust solution ready to test in live events with these enhancements. There is promising research in using ViT for solving video-related object detection tasks. (Selva et al., n.d.) The long-term goal would be to allow users to upload videos and images for the events and automate post-event data entry around participation, scoring, and results. The next step is gathering a larger dataset with the kinds of images we would expect to encounter in the rodeo events. ViT, indeed, is a solid alternative to CNN-based object detection models. However, their performance is very dependent on large training datasets. "ViT models are generally found to perform best in settings with large amounts of training data..." (Steiner et al., n.d.) We just had over 55 images for training. With a larger dataset and increasing the number of epochs, we could improve the performance of our model. This project has given us an excellent foundation to build on. Our ViT model proved that this architecture is a viable approach for developing this kind of Computer Vision powered system.

References

- Sagar, R. (2021, August 30). *Are Visual Transformers Better Than CNNs*. Analytics India Magazine. https://analyticsindiamag.com/are-visual-transformers-better-than-convolutional-neural-networks/
- Boesch, G. (2022, March 6). *Vision Transformers (ViT) in Image Recognition - 2022 Guide - viso.ai*. Viso.ai. https://viso.ai/deep-learning/vision-transformer-vit/#:~:text=Moreover%2C%20ViT%20models%20outperform%20CNNs,to%20computational%20efficiency%20and%20accuracy.
- Sonawani, S., Alimo, R., Detry, R., Jeong, D., Hess, A., & Amor, H. (n.d.). *Assistive Relative Pose Estimation for On-orbit Assembly using Convolutional Neural Networks*. Retrieved December 10, 2022, from https://arxiv.org/pdf/2001.10673.pdf (Sonawani et al., n.d.)
- Banerjee, S. (2022). *Animal Image Dataset (90 Different Animals)*. Kaggle.com. https://www.kaggle.com/datasets/iamsouravbanerjee/animal-image-dataset-90-different-animals
- Nikolas Adaloglou. (2021, January 28). *How the Vision Transformer (ViT) works in 10 minutes: an image is worth 16x16 words | AI Summer*. AI Summer; Sergios Karagiannakos. https://theaisummer.com/vision-transformer/
- *Vision Transformers (ViT) Explained | Pinecone*. (2017). Pinecone. https://www.pinecone.io/learn/vision-transformers/
- Boesch, G. (2022, July 16). What is Overfitting in Computer Vision? How to Detect and Avoid it - viso.ai. Viso.ai. https://viso.ai/computer-vision/what-is-overfitting/

- Selva, J., Johansen, A., Escalera, S., Nasrollahi, K., Moeslund, T., & Clapés, A. (n.d.). *Video Transformers: A Survey*. Retrieved December 12, 2022, from https://arxiv.org/pdf/2201.05991.pdf
- *TestEngine*. (2021, January 12). *Vision Transformers (ViT) - TestEngine*. TestEngine. https://testengine.ai/vision-transformers-vit/
- Steiner, A., Kolesnikov, A., Zhai, X., Wightman, R., Uszkoreit, J., & Beyer, L. (n.d.). *How to train your ViT? Data, Augmentation, and Regularization in Vision Transformers*. Retrieved December 12, 2022, from https://openreview.net/pdf?id=4nPswr1KcP
- tfa.optimizers.AdamW. (2022). *tfa.optimizers.AdamW | TensorFlow Addons*. TensorFlow. https://www.tensorflow.org/addons/api_docs/python/tfa/optimizers/AdamW
- *labelImg*. (2017, July 21). PyPI. https://pypi.org/project/labelImg/1.4.0/
- Solawetz, J. (2020, July 16). *How to Train a TensorFlow 2 Object Detection Model*. Roboflow Blog; Roboflow Blog. https://blog.roboflow.com/train-a-tensorflow2-object-detection-model/
- *Roboflow: Give your software the power to see objects in images and video*. (2022). Roboflow.com. https://roboflow.com/
- *tf.keras.layers.MultiHeadAttention | TensorFlow v2.11.0*. (2022). TensorFlow. https://www.tensorflow.org/api_docs/python/tf/keras/layers/MultiHeadAttention

Appendix

1.0 Model Code

[Find Colabe Here]

2.0 Model Layers and Architecture

[View Full Image Here]