Detection Transformer for Hands, Guns and Phones
Project Report
CSC 671 - Spring 2024

Marco Lorenz* Undergraduate, SFSU
Joey Palanca† Undergraduate, SFSU
Ali Ghazy Alsharif‡ Undergraduate, SFSU
Robert Mateescu§ Professor CSC 671, SFSU

1 Abstract

We present an approach to detect Hands, Guns and Phones in images by training a Detection Transformer Model [5] to the Hands, Guns and Phones (HGP) dataset [11]. Thus, we show how to train an existing architecture [30] with a different dataset [12] by implementing the Torch Vision Base Class VisionDataset [43] and extending the COCO PythonAPI [35]. We further present the results of a hyperparameter study, a demo of the model during inference and sample visualizations of the transformer (self) attention mechanism. Training Code and an inference demo are available at: `https://github.com/lorenz369/hgp_detr`

2 Introduction

In 2020, Carion, Massa et al. first applied the increasingly popular transformer architecture introduced in 2017 by Vaswani et al. [45] to an objection detection problem. By viewing object detection as a direct set prediction problem on the feature map of a CNN backbone, they were able to bypass surrogate tasks like proposals [38] or anchors [25]. Since their approach forced the model to output unique predictions, post-processing steps like non-maximum suppression or the elimination of duplicate detections are not necessary either [5].

This relative simplicity combined with a well-document open-source repository without complex dependencies provides a good starting point for further exploration. Since we did not have access to the same amount of computational resources as Meta Research, with 16 V100 GPUs employed over a period of 3 days for training the model, we chose a smaller dataset to train on. The HGP dataset [12] only contains 2199 images of 3 object categories, whereas the original COCO dataset contains ¿200k labeled images, including 80 object categories [7].

Section 5 describes how to define a custom dataset for Torch Vision by extending the Torch Vision Base Class VisionDataset [43], and how to integrate it with DETR for both training and evaluation of the model [35]. Section 6 shows how hyperparameters specific to a Transformer Detection Model, like the embedding dimension or the number of queries, etc., influence the accuracy on a comparatively small dataset. We achieved a test loss of 8.97 after training for 350 epochs on a NVIDIA GeForce RTX 3080 GPU. We further validate our approach with visualizations of the predictions and the transformer (self) attention mechanism.

3 Background

3.1 Object Detection

"Object Detection is a computer vision task in which the goal is to detect and locate objects of interest in an image or video. The task involves identifying the position and boundaries of objects in an image or a video, and classifying the objects into different categories." [29] Therefore, an object detection model must predict both a class and a bounding box for each identified object, which is a more complex task than mere classification. However, the bounding box might be crucial in various contexts, for example in Autonomous Driving, Robotics, Video Surveillance or analysis of medical data.

3.2 Hands, Guns and Phones dataset (HGP)

This dataset was introduced by Duran-Vega et al. in 2021 as part of a paper called "Temporal Yolov5 Detector Based on Quasi-Recurrent Neural Networks for Real-Time Handgun Detection in Video." [11] The "Hands Guns and Phones (HGP) dataset contains 2199 images (1989 for training and 210 for testing) of people using guns or phones in real-world scenarios (people making phones reviews, shooting drills, or making calls). Every image of this dataset is labeled with the bounding boxes of Hands, Phones and Guns. All the aforementioned images were collected from Youtube videos and have different sizes." [12]

3.3 PyTorch

The original DETR implementation and our modifications use PyTorch as a highly optimized API, which itself leverages CUDA to employ GPU(s) during training, if available. CUDA® is a parallel computing platform and programming model developed by NVIDIA for general computing on graphical processing units (GPUs), particularly NVIDIA GPUs [32]. PyTorch is an open-source machine learning library developed by Facebook's AI Research lab (FAIR). It is widely used for applications such as computer vision and natural language processing, with object detection being a subset of computer vision [14].

3.4 Hungarian Algorithm

Kuhn introduced the Hungarian Algorithm in March 1955 to solve the assignment problem: "Assuming that numerical scores are available for the performance of each of n persons on each of n jobs, the 'assignment problem' is the quest for an assignment of persons to jobs so that the sum of the n scores so obtained is as large as possible. It is shown that ideas latent in the work of two Hungarian mathematicians may be exploited to yield a new method of solving this problem." [24] Carion et al. used an implementation of the Hungarian algorithm as a subroutine to construct their loss function, which is a crucial part of their contribution and described in Subsection 3.6.1.

3.5 Transformer and Parallel Decoding

Vaswani et al. introduced Transformer in 2017 [45] as a potential replacement for previous state-of-the-art approaches in sequence modeling and transduction problems, like language modeling and machine-translation [41] [4]. The dominant architecture at the were recurrent neural networks, such as long short-term memory [17] or gated recurrent [6] neural networks. The inherently sequential nature of those models constraints their input size, because they typically predict one token t at a time, utilizing their hidden state

*email: mlorenz@sfsu.edu
†email: jpalanca@sfsu.edu
‡email: aalsharif@mail.sfsu.edu
§e-mail: mateescu@sfsu.edu

h_{t-1} updated with the previous token $t-1$. The longer an input sequence, the more iterations of this hidden state update, which increases memory load and, depending on the specific architecture, leads to loss of information particularly with respect to earlier tokens of a long input sequence. Tokens refer to subparts of a text sequence ranging from single characters to entire words, usually representing a few characters. Attention has been proposed as a potential solution to the described constraint of sequential computation, which leads to more efficient aggregation of information across longer input sequences by modeling dependencies between tokens indifferent of their distance [22] [33].

In hindsight, transformer emerged not only in the form of Large Language Models (LLMs) [10] [37], but were applied to speech problems [27] [42], computer vision [34] and ultimately to multi-modal input [44] as well.

3.5.1 Encoder-Decoder Architecture

Vaswani et al. now combined the attention mechanisms with an encoder-decoder architecture to spark a new wave of research and development, including popular breakthroughs like ChatGPT [26]. The encoder-decoder approach involves two neural networks: One encodes a sequence of symbols, represented by tokens in the context of Natural Language Processing (NLP), into a fixed length vector representation, also called embedding. Accordingly, the decoder network decodes this representation back to tokens, and ultimately symbols. Embeddings are learned functions to convert input and output tokens to vectors of dimension d_model, which are then propagated through transformer networks. For example, Vaswani et al. used embedding dimension $d_model = 512$. Figure 1 shows the original transformer architecture, where both the encoder and the decoder networks consist of attention layers (see Section 3.5.3) combined with feed-forward layers, consisting of two linear transformations with a ReLU activation in between. For both networks, the input consists of embeddings: The input embedding contains the input sequence. The output embedding is masked and offset by one position, such that predictions for position i can only attend to the known outputs $< i$. Thus, the model predicts one token at a time, attending to all the previous tokens. This property is also called auto-regression and is common in LLMS. DETR, however, only leverages this general encoder-decoder approach, but without the autoregressive nature (see Subsections 3.5.4 and 3.6.2).

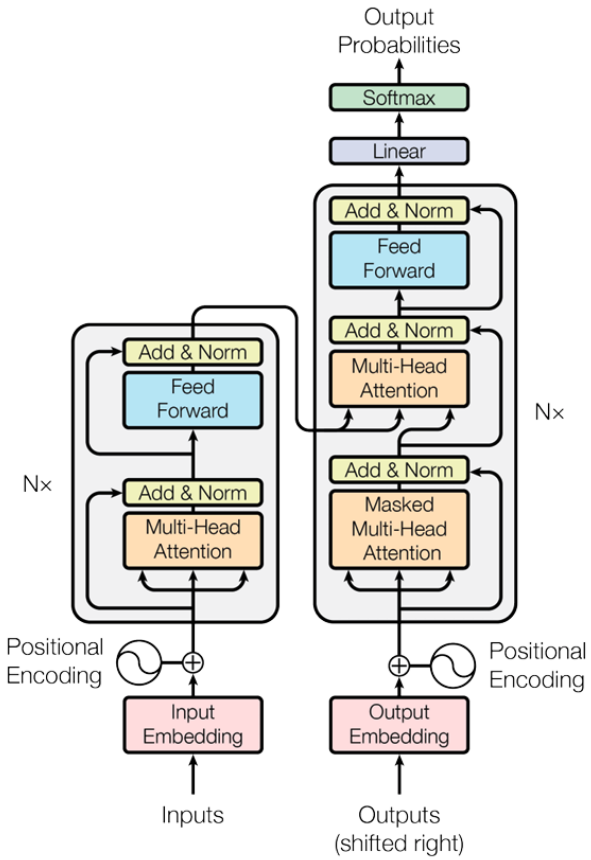

Figure 1: Transformer architecture by Vaswani et al. with 2-Sublayer-Encoder (left) and 3-Sublayer-Decoder (right), both of them stacked N times. Image Credit: [45]

3.5.2 Layer Normalization and Positional Encoding

Figure 1 includes more techniques commonly used in transformer architectures: It shows residual connections [16] around each feed-forward and attention layer to improve gradient flow and ease the training of deep neural networks. After each sublayer, the residual connection is added to the output of the sublayer, and normalized with layer normalization [3] to stabilize the model and further decrease training time. Figure 1 also shows, that positional encodings are added to the input embeddings of both encoder and decoder to provide the model with a notion of order, which it would be lacking otherwise. There are different positional encoding schemes; Vaswani et al. chose sine encodings, meaning sine and cosine functions of different frequencies depending on the position i of the token, dimension i of the positional encoding and dimension d_{model} of the input embedding:

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{model}})$$

$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{model}})$$

See [45] for details on positional encoding. DETR employs both residual connections and positional encodings in its architecture (see Subsection 3.6.2).

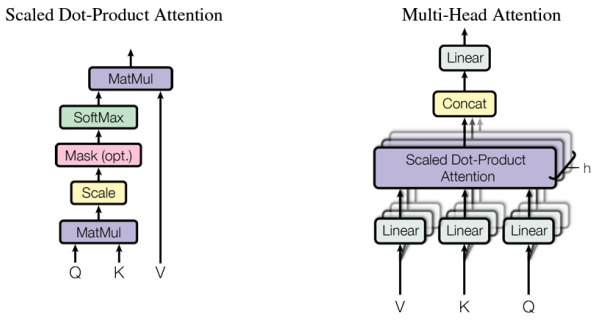

Figure 2: (left) Scaled Dot-Product Attention. (right) Multi-Head Attention consists of several attention layers running in parallel. Image Credit: [45]

3.5.3 Attention

Figure 2 shows the particular attention mechanism employed by Vaswani et al [45], which they call "Scaled Dot-Product Attention". Its input includes three vectors: A query and keys of dimension d_k, as well as values of dimension d_v. The dot product of the query with all the keys, scaled down by $\sqrt{d_k}$ and propagated through a softmax function yields the weights to determine the individual contributions of the values by computing the dot product of now the weights and the values. Optionally, masking can be applied to ensure auto-regressive decoding as described in Subsection 3.5.1.

In practice, those attention functions can be computed on a set of queries, keys and values simultaneously, represented by the matrices Q, K and V. Thus, the attention function is defined as:

$$Attention(Q,K,V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

According to Vaswani et al., the down scaling's purpose is to constraint the magnitude of the dot product QK^T, because the softmax function's gradients shrink for growing magnitudes.
The right side of Figure 2 shows multiple scaled dot-product attention heads forming a bigger unit, called the "Multi-Head Attention" block. Vaswani et al. found it beneficial to perform attention on h linear projections of the queries, keys and values, with h being the number of attention heads. This means to project the $d_(model)$-dimensional keys, values and queries to dimension d_k, d_k and d_v, respectively, with different and learned linear projections, also included in Figure 2. In practice, those dimensions d_k and d_v may be smaller than d_model, leading to performance gains, since now each attention head of with smaller dimensions can be computed in parallel, yielding h vectors of dimension d_v. For example, Vawswani et a. used $h = 8$ parallel attention layers, or heads with $d_k = d_v = d_model/h = 512/8 = 64$. Please note, that the total computational cost is similar, while multi-head attention will be faster in practice if the hardware supports parallelization, which in turn is one of the reasons why GPUs are particularly suitable for Deep Learning applications. The h output values of dim_v are concatenated again after their parallel computation to restore the original embedding dimension, following Vaswani et al.'s example: $dim_model = h*dim_v = 8*64 = 512$. The result of the concatenation then gets projected linearly once again to determine the final output of the multi-head attention block, still of dim_model. In summary, Figure 2 shows visually, what Vaswani et al. defined formally as:

$$\text{MultiHead}(Q,K,V) = \text{Concat}(\text{head}_{i=1},\ldots,\text{head}_h)W^O,$$
$$\text{where head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

and where the projections are parameter matrices $W_i^Q \in \mathbb{R}^{d_{\text{model}} \times d_k}$, W_i^K and $W^O \in \mathbb{R}^{h_d v \times d_{\text{model}}}$.
In practice, encoder attention layers exhibit a self attention property, meaning that each position i in the input sequence can attend to all other positions, because keys, queries and values all come from the previous encoder layer. Thus, they are referred to as self-attention layers. The masked multi-head attention layer (see Figure 1) is a self-attention layer as well, but it is only allowed to attend to positions in the decoder up to and including that position to preserve its auto-regressive nature. Vaswani et al. achieved this by masking out all illegal values in the input of the softmax, effectively setting every weight corresponding to a "future" position to zero. In that sense, the only non-self-attention layers are "encoder-decoder attention" or "cross-attention" layers, consuming the memory of the encoder by taking in the memory keys and values from the encoder and performing multi-head attention with the queries from the previous decoder layer. DETR employs the same multi-head attention modules, which, combined with a feed-forward layer, form an entire encoder/decoder layer and leverages all the transformer principles and techniques discussed so far, cutting off only the auto-regressive property, which is described in the following section.

3.5.4 Parallel decoding

As described in Subsection 3.5, transformers were designed and originally applied to sequential input in the form of text. Their attention mechanism (see Subsection 3.5.3) allows for global aggregation of information across an entire input sequence, which makes them particularly suitable for longer input sequences. Combined with the encoder-decoder architecture, this allows for a very good memory, because the transformer can learn to prioritize its attention. When employed to predict one token at a time to produce sequential output while always attending to the previous position, this auto-regressive approach led to impressive results in text generations, and sparked the rise of LLMs like ChatGPT [26] and BERT [10]. When transformers were applied to different types of inputs, however, the auto-regressive property became a constraint, because computation has to be repeated for each output position, which may be superfluous for mediums like images. Therefore, DETR also leverages a transformer decoder network, which predicts multiple positions, or objects, in parallel, as described in Subsection 3.6.2.

3.6 Detection Transformer (DETR)

In 2020, Facebook AI group presented a new method as an alternative to Convolutional Neural Networks in object detection. By viewing object detection as a direct set prediction problem, they streamlined the detection pipeline and removed the need for previously hand-designed components. Hereby, they were able to demonstrates accuracy and run-time performance on par with the well-established and highly-optimized Faster RCNN baseline on the challenging Microsoft COCO object detection dataset [5]. The main innovations were a custom Object Detection Set Prediction Loss function, and a transformer encoder-decoder architecture applied to Object detection. Both of them are described in the following subsections.

3.6.1 Object Detection Set Prediction Loss

Set prediction in this context refers to the prediction of a set of objects induced by a set of queries inputted to the transformer decoder network of DETR (see Subsection 3.6.2). Set prediction is required to be permutation-invariant by definition, which renders transformers a natural choice for set prediction tasks, since they do not exhibit any notion of order without explicit positional encoding and are able to reason globally across an entire input sequence. Furthermore, set prediction requires every element of each set to be matched uniquely to one element of the other, theoretically not allowing any duplicates.

In practice, deep learning models have to be trained to output unique predictions by a loss function enforcing this property. In the following, we will describe the loss computation designed and utilized by Carion et al. in DETR [5]. It involves a two-step process:

Optimal Assignment. DETR, for each pass through the decoder, infers a fixed-size set of N object predictions. N is determined by the number of queries inputted into the decoder, and significantly larger than the average number of objects in an image in the desired context. For example, in the original DETR, N was set to 100, whereas the average object count in the COCO Dataset [7] is only about 8.

The main challenge during training is to consistently score predicted objects, each consisting of a class and a bounding box, with respect to the ground truth, such that the model will systematically learn to output unique predictions with both correct class predictions and reasonable bounding boxes. Conceptually, this involves determining "which prediction corresponds most closely to which ground truth object" to only foster these unique predictions, while penalizing any further predictions to enforce predictions of the so called null-object, denoted as $\emptyset$. To find this unique bipartite matching of

a set of predictions to a set of ground truth objects, in this context padded with $\emptyset$, can be viewed as a reformulation of the assignment problem: N Workers and jobs become prediction set and ground truth set of size N, and the n^2 performance/utility scores correspond to the n^2 distances between each prediction and each ground truth object, typically referred to as cost in the context of machine learning. Following this idea, the Hungarian algorithm computes the optimal assignment (see Subsection 3.4 of the ground truth set y, padded with $\emptyset$ to match the number of predictions, to the set of N predictions $\hat{y} = \{\hat{y}_i\}_{i=1}^{N}$, if a pair-wise *matching cost* $\mathscr{L}_{\text{match}}(y_i, \hat{y}_{\sigma(i)})$ is defined. The optimal assignment, or bipartite matching, can thus be denoted as a permutation of N elements $\sigma \in \mathscr{G}_N$ with the lowest cost:

$$\hat{\sigma} = \arg\min_{\sigma \in \mathscr{G}_N} \sum_{i}^{N} \mathscr{L}\text{match}(y_i, \hat{y}_{\sigma(i)}) \tag{1}$$

The matching cost $\mathscr{L}_{\text{match}}(y_i, \hat{y}_{\sigma(i)})$ has to score both the class prediction and the similarity of predicted and ground truth boxes. Each element i of the ground truth is thus defined as $y_i = (c_i, b_i)$, with c_i being the target class label and $b_i \in [0,1]^4$ being the target bounding box, respectively. Note, that c_i might be $\emptyset$ and b_i is a vector with ground truth box center coordinates, as well as its height and width relative to the image size. For the prediction with index $\sigma(i)$, the probability of class c_i is defined as $p_{\hat{\sigma(i)}}(c_i)$, and the predicted box $b_{\hat{\sigma(i)}}$, respectively. Thus, we define:

$$\mathscr{L}_{\text{match}}(y_i, \hat{y}_{\sigma(i)}) = -1_{\{c_i \neq \emptyset\}} \hat{p}_{\sigma(i)}(c_i) + 1_{\{c_i \neq \emptyset\}} \mathscr{L}_{\text{box}}(b_i, \hat{b}_{\sigma(i)}). \tag{2}$$

The second bounding box summand of $\mathscr{L}_{\text{match}}(y_i, \hat{y}_{\sigma(i)})$ is determined by the bounding box loss $\mathscr{L}_{\text{box}}(b_i, \hat{b}_{\sigma(i)})$. The bounding box loss $\mathscr{L}_{\text{box}}(b_i, \hat{b}_{\sigma(i)})$ itself is a linear combination of the scale-dependent l_1 loss and the generalized IoU loss [39], which is scale-invariant. In summary:

$$L_{\text{box}}(b_i, \hat{b}_{\sigma(i)}) = \lambda_{\text{iou}} L_{\text{iou}}(b_i, \hat{b}_{\sigma(i)}) + \lambda_{L1} \|b_i - \hat{b}_{\sigma(i)}\|_1 \tag{3}$$

where $\lambda_{\text{iou}}, \lambda_{L1} \in \mathbb{R}$ are hyperparameters.

Hungarian loss. Finally, after the optimal assignment or bipartite matching $\hat{\sigma}$ (1) is defined with the help of the bipartite matching cost $\mathscr{L}_{\text{match}}(y_i, \hat{y}_{\sigma(i)})$ (2), which itself scores the class prediction and the predicted bounding box with the bounding box loss $\mathscr{L}_{\text{box}}(b_i, \hat{b}_{\sigma(i)})$ (3), the actual loss function can be computed for all pairs matched in the previous step. Thus, the *Hungarian loss* is defined with the optimal assignment $\hat{\sigma}$:

$$\mathscr{L}_{\text{Hungarian}}(y, \hat{y}) + \sum_{i=1}^{N} [-\log \hat{p}_{\sigma(i)}(c_i) + 1_{\{c_i \neq \emptyset\}} \mathscr{L}_{\text{box}}(b_i, \hat{b}_{\sigma(i)})] \tag{4}$$

In practice, when $c_i = \emptyset$, the log-probability term is down-weighted by a factor 10 to account for class imbalance. Otherwise, the model could always predict ¿90% of the classes correctly by always predicting $\emptyset$, since on average only 8 ground truth objects correspond to a set of 100 object queries, resulting in 100 predictions.

3.6.2 Architecture

Figure 3 summarizes the entire DETR architecture, consisting of 3 main components, which we describe below: a CNN backbone, which serves as a feature extractor, an encoder-decoder transformer and a simple feed forward network to predict both classes and bounding boxes, or the "No-object", respectively.

Backbone. Convolutional Neural Networks, typically consisting of a combination of convolutional, max-pooling and fully connected layers (amongst others) [23], are widely adopted for image classification tasks. In the context of object detection, they are typically used as a feature extractor in a detection pipeline [46]. Carion et al. [5] experimented with different ResNet variations as their feature extractor, mainly ResNet-50 and ResNet-101 [16]. ResNet-101 adds precision gains at the expense of an additional 20M parameters. For DETR-ResNet-50, the CNN backbone accounts for approximately 64% of parameters, 74% for DETR-ResNet-101.

The initial image $x_{\text{img}} \in \mathbb{R}^{3 \times H_0 \times W_0}$ with 3 color channels, height H_0 and width W_0 is the input to the CNN backbone, which condenses it into a lower resolution activation map $f \in \mathbb{R}^{C \times H \times W}$. Typical values are $C = 2048$ and $H, W = \frac{H_0}{32}, \frac{W_0}{32}$.

Flattening and Positional Encoding. A 1×1 convolution then reduces the channel dimension of the high-level activation map f from C to a smaller dimension d_{model}, creating a new feature map $z_0 \in \mathbb{R}^{d_{\text{model}} \times H \times W}$. Since the encoder expects a one-dimensional sequence, the model collapses the spatial dimension of z_0 into one dimension, creating a $d_{\text{model}} \times HW$ feature map.

As introduced in Subsection 3.5.2 positional encodings have to be added to the input sequence of transformers to provide a notion of order. The spatial nature of images exhibits a two-dimensional order, consisting of x and y positions. Hence, DETR encodes both the x and y position of each pixel with the positional sine encoding scheme discussed earlier, resulting in two embeddings, which are concatenated and added to the feature map.

Transformer Encoder. The transformer enccoder of DETR is equivalent to the original architecture [45] described in Subsection 3.5, consisting of stacked encoder blocks, each with one Multi-Head Self-Attention and one feed-forward layer. They further include skip-connections and layer-normalization, following prior work [16] [3]. Carion et al. implemented two layer-normalization variations, pre- and post-layer-normalization, where either the inputs to or the output of the attention and feed-forward layers are normalized.

Transformer Decoder. The transformer decoder of DETR is almost equivalent to the original architecture [45] described in Subsection 3.5, differing in the input to the decoder. For DETR, this is a fixed size set of N, objects queries, where N is typically much larger than the average number of objects in an image instance. For example, the average number of objects in the COCO object detection dataset [7] is 8, whereas Carion et al. chose $N = 100$. Since the object-queries are purpose-fully permutation-invariant, the spatial positional encoding is not added to the input of the decoder, but to the memory keys of the encoder, when computing cross-attention in the second attention layer of each decoder block. Equivalent to the encoder, Carion et al. also implemented support of both pre- and post-layer-normalization for the decoder.

In the implementation, the default values for the transformer are $N = 6$ stacked encoder blocks, $M = 6$ decoder blocks, each with 8 attention heads per attention layer (see 3.5.3, post-layer-normalization, and a dropout of 0.1 to prevent overfitting, following Srivastava et al. [40].

Prediction feed-forward networks (FFNs). Carion et al. generate predictions of the bounding box coordinates using a 3-layer multi-layer-perceptron (MLP) with ReLU activations and a hidden dimension d_{model}. The class prediction, respectively, is the output of one linear layer with input dimension d_{model} and output dimension $O+1$, where O is the number of distinct object classes in the dataset, for example 80 in COCO [7]. The class prediction layer supports one additional output class, $\emptyset$ or the "No-Object", to account for the

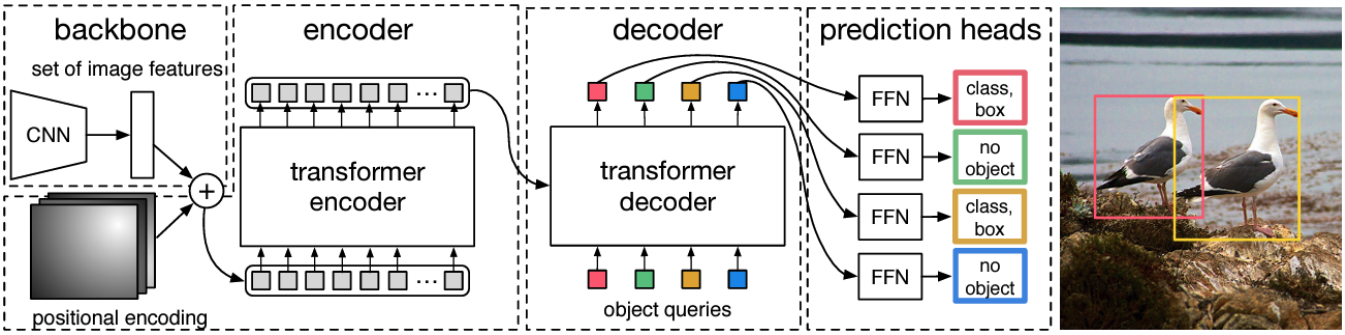

Figure 3: (left) DETR uses a conventional CNN backbone to learn a 2D representation of an input image. The model flattens it and supplements it with a positional encoding before passing it into a transformer encoder. A transformer decoder then takes as input a small fixed number of learned positional embeddings, which we call object queries, and additionally attends to the encoder output. We pass each output embedding of the decoder to a shared feed forward network (FFN) that predicts either a detection (class and bounding box) or a "no object" class. Image Credit: [5]

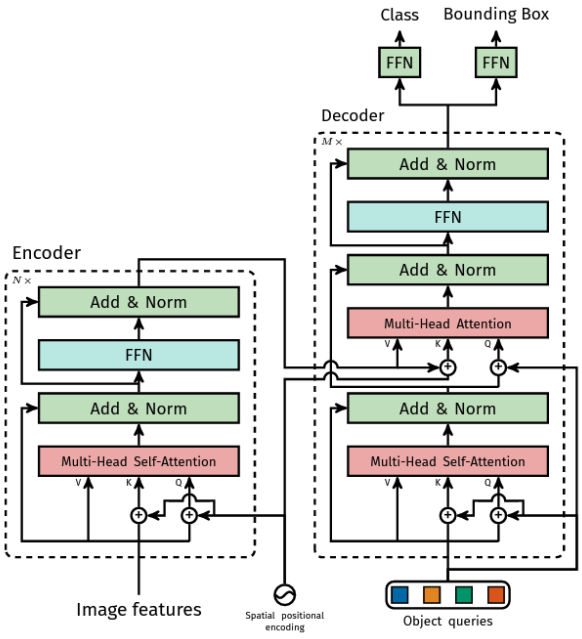

Figure 4: (left) Architecture of DETR's transformer. Image Credit: [5]

fixed size N of the object query set.

Auxiliary decoding losses. Al-Rfou et al. introduced auxiliary losses to ease the training of very deep neural networks in 2018 [1]. Carion et al. observed, that auxiliary losses during training helped the model to output the correct number of objects of each class. To support auxiliary losses, they added FFNs with shared parameters and Hungarian loss (see Subsection 3.6.1) after each decoder layer, preceded by an additional layer-normalization.

4 Approach

As indicated in Section 2, we chose to explore DETR with a smaller dataset, the HGP dataset [12] to account for our limited amount of computational resources. Hence, we followed a three-step approach:

HGP Integration. We first added support of training with the HGP dataset to the DETR implementation [30]. It natively supports a huge variety of flags, including a specification of the path to the dataset to train on. The support of the original COCO dataset [7] is implemented by the pycocotools API [35], which eases the installation, since it's a package supported by conda [2] and pip [36]. This makes the implementation of the interface to the dataset dependent on pycocotools, which is why we followed Carion et al.'s approach to implement the classes to process the object annotations and evaluate the predictions with the help of modified pycocotools code.

For the implementation of the dataset class *HGP* itself, we used a predefined Torch Vision base class [43]. We explain our implementations in more detail in the following Section 5.

Hyperparameter Search. Since DETR and its implementaion supports ¿30 hyperparameters, we chose to focus on a smaller set of hyperparameters, again to account for computational constraints. We selected a set of 5 hyperparameters, which we expected to be either impactful on the performance of the model, or are comparatively unique to detection transformers, along with the following hypotheses:

1. Batch Size: We expected the batch size to have an impact on the training time and the precision.

2. N - Number of Object Queries: We expected the number of object queries to influence the precision, and perhaps the number of predicted objects.

3. h - #attention heads: We expected the number of object queries to influence the precision.

4. d_ffn - Feed-forward dimension: We expected the number of object queries to influence the precision, and potentially the training time because of an increased parameter count.

5. d_model - Embedding/transformer dimension: We expected the number of object queries to influence the precision, and potentially the training time because of an increased parameter count.

To explore the influence of each selected hyperparameter, we fixed a lightweight set of training parameters, and manipulated only one parameter at a time. We only ran each configuration for 3 epochs because of the expensiveness of each epoch for a model with ~12M parameters. For reproducibility, we provide our default lightweight set of hyperparameters below: (lr=0.0001, lr_backbone=1e-05, batch_size=2, weight_decay=0.0001, epochs=3, lr_drop=200, clip_max_norm=0.1, frozen_weights=None, backbone='resnet18', dilation=False, position_embedding='sine', enc_layers=2, dec_layers=2, dim_feedforward=512, hidden_dim=64, dropout=0.1, nheads=4, num_queries=20, pre_norm=False, masks=False, aux_loss=True, set_cost_class=1, set_cost_bbox=5, set_cost_giou=2, mask_loss_coef=1, dice_loss_coef=1, bbox_loss_coef=5, giou_loss_coef=2, eos_coef=0.1, dataset_file='hgp', coco_path=None, coco_panoptic_path=None, remove_difficult=False, output_dir='', device='cuda', seed=42, resume='', start_epoch=0, eval=False, num_workers=2, world_size=1, dist_url='env://', fast_dev_run=None, section=None, rank=0, gpu=0, distributed=True, dist_backend='nccl'). For more details, please refer to our Github-Repository [28] or the DETR publication. [5]

Model Training and Visualizations. After determining a promising set of hyperparameters, we trained two different configurations for 300 and 500 epochs, respectively. We evaluated these models based on their training logs, and validated their effectiveness with some sample inference runs on selected stock photos containing Hands, Guns and Phones. Additionally, we visualized the attention in the last encoder and the last decoder layer for one of the samples. We discuss both the hyperparameter search and findings obtained during training in Section 6.

5 HGP Implementation

In the following, we describe how we integrated the HGP-Dataset [12] into the existing DETR implementation by Carion et al. [30]. Amongst smaller modifications, the main contributions were three classes, described in the following:

1. HGPDetection class: Image and target object retrieval during training

2. HGP class: Loading and parsing of the annotation file

3. HGPEvaluator class: Evaluation of predictions

5.1 HGPDetection Class

Listing 1 shows the implementation of the HGPDetection class, which itself extends the base class VisionDataset [43]. To create custom datasets, which are compatible with Torch Vision, the __getitem__() and __len()__ functions must be overwritten. The *self.prepare*() converts all associated data like image size, target bounding box, etc. into *torch.tensor* objects for the training process with PyTorch. The self._transforms performs a random resizing of the input data as well as normalization. Only during training, it provides data augmentation by a combination of random horizontal flip with random cropping before the normalization.

```
class HGPDetection(VisionDataset):
    def __init__(self, img_folder: str, lab_folder: str, ann_file: str, image_set: str, transforms)
        [...]

    def _load_image(self, id: int) -> Image.Image:
        [...]
        return Image.open(img_path).convert('RGB')

    def _load_target(self, id: int) -> List[Any]:
        return self.hgp.loadAnns(self.hgp.getAnnIds(id))

    def __getitem__(self, index: int) -> Tuple[Any, Any]:
        image_id = self.ids[index]
        image = self._load_image(index)
        target = self._load_target(index)

        target = {'image_id': image_id, 'annotations': target}
        image, target = self.prepare(image, target)
        if self._transforms is not None:
            image, target = self._transforms(image, target)

        return image, target
```

Listing 1: Source Code of the HGPDetection class

5.2 HGP Annotations class

The *HGP* class in Listing 2 serves as a custom extension of the *CocoEvaluator* class to create an API for the annotations of the HGP dataset, including simple functions for handling annotations, object categories and images. The *HGPDetection* class instantiates the *HGP* class for its annotation processing.

5.3 HGP Evaluation

Listing 3 shows the function names and signatures of the *HGPEvaluator* class, which is instantiated for certain set of *iou_types*, because the IoU [39] metric depends on predefined thresholds. IoU means Intersection over Union and is computed for bounding box and ground truth box, with different thresholds for valid predictions. In the next step, an *HGPEvaluator* object can score a set of predictions with respect to the previously defined set of IoU metrics.

6 Results

6.1 Computational Platform and Software Environment

We carried out experiments on two different platforms, the hyperparameter search on a single Nvidia A100 GPU on a GPU node of NERSC-9, Perlmutter, and the training in a coder workspace with an Nvidia GeForce RTX 3080.

Nvidia A100 - Perlmutter. I conducted all experiments on a single GPU node of Perlmutter. Perlmutter, also known as NERSC-9, is a supercomputer delivered to the National Energy Research Scientific Computing Center of the United States Department of Energy (NERSC).

Each GPU node consists of a single AMD EPYC 7763 (Milan) CPU with a base clock rate of 2.45 GHz, 32 KB L1 Cache, 512 KB L2 Cache and 256 MB of L3 Cache [31]. The NVIDIA A100

```
class HGP:
    def __init__(self, annotation_file=None):

    def getAnnIds(...):

    def getCatIds(...):

    def getImgIds(...):

    def loadAnns(...):

    def loadCats(...):

    def loadImgs(...):
}
```

Listing 2: Source Code Snippet of the HGP (annotation) class

```
class HGPEvaluator(CocoEvaluator):
    def __init__(self, hgp_gt, iou_types):

    def update(self, predictions):
```

Listing 3: Source Code Snippet of the HGP Evaluator class

GPU exhibits a peak single precision performance of 19.5 TFLOP/s for FP32 types, and a tensor core performance for TF32 types of 156 TFLOP/s, which are the two datatypes typically utilized by PyTorch. Its memory system includes 192 KB L1 Data Cache / Shared Memory per streaming multiprocessor (SM) for 128 SMs, 40 MB L2 on-chip cache and 40 GB of high-speed HBM2 memory with 1555 GB/sec of peak memory bandwidth [9].

The software environment consists of SUSE Linux Enterprise Server 15.4, CUDA 12.2, and a conda environment with Python 3.11.8, PyTorch 2.2.2, torchvision 2.2.0, pycocotools 2.0.6, scipy 1.12.0 and numpy 1.26.4.

Nvidia GeForce RTX 3080 - Coder. A Coder workspace is a public cloud software development environment [15].

Our Coder workspace is equipped with a single NVIDIA GeForce RTX 3080 GPU. The GeForce RTX 3080 Family seems to be geared towards gamers, but we could find some performance specifications like a peak 29.8 TFLOP/s for both FP32 operations and TF32 Tensor operations. Its memory system includes 8704 KB of L1 Data Cache/Shared Memory, a L2 Cache Size of 5120 KB, and an HBM memory size of 10 GB with a memory bandwidth of 760 GB/s. Interestingly, its L1 Data Cache/Shared Memory is 45x the counterpart of the flagship A100 Tensor Core GPU, whereas the A100 L2 cache is 8x the GeForce RTX 3080's equivalent.

The software environment is Ubuntu 20.04.1 with the same software packages and versions as on Perlmutter.

6.2 Hyperparameter Study

We describe our methodology in Section 4. For context with respect to the hyperparameters itself, we refer to Section 3.6 or Carion et al. [5].

6.2.1 Batch Size

Figure 5 shows how the batch size influences the average precision of our DETR modification, and also the training time. The batch size is the only hyperparameter to significantly impact the training time, which is we also plotted the training time in this chart. It suggests a speed-accuracy trade-off, where batch size 2 clearly yields the highest precision, but with the highest runtime. Both runtime and accuracy decrease, saturate, and increase again with increasing batch size, with training time hitting its minimum at batch size 8, and precision at 32, respectively. Overall, the chart suggests batch sizes 2, 4 and 8 as reasonable trade-offs, with 2 prioritizing precision, and 8 prioritizing speed. We chose a batch size of 2 for our training to prioritize precision.

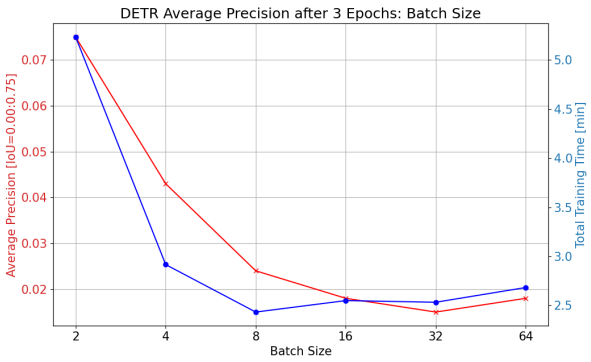

Figure 5: (left) DETR Average Precision for varying Batch Sizes.

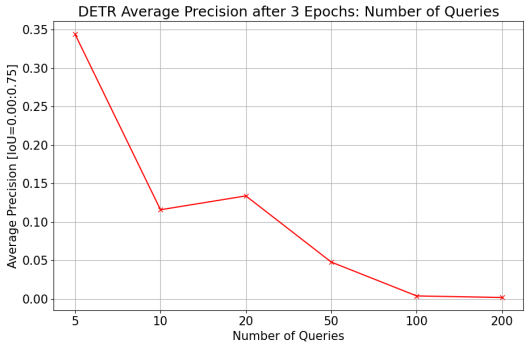

Figure 6: (left) DETR Average Precision for numbers of Queries.

6.2.2 Number of Queries

Figure 6 shows DETR's average precision for increasing numbers of queries. It suggests, that a low number of queries leads to a higher precision, with 20 queries presenting themselves as the only exception to that pattern. This result is surprising, given that 100 is DETR's default value. We hypothesize, that this discrepancy is likely due to the low number of training epochs before measurement: At this early stage, the model may still have difficulty determining the correct number of objects in an image, particularly because our pretrained ResNet-18 backbone [8] is trained the ImageNet dataset with 1000 classes [16], whereas we only score the DETR model on Hands, Guns and Phones. This likely leads to a high number of object predictions in the prediction set during the early stages of training, whereas the ground truth set mostly contains "No-objects" leading to low accuracy, particularly for higher numbers of queries.

6.2.3 Number of Attention Heads

Figure 7 shows another surprising result with respect to DETR's default value of 8 attention heads. The results suggest, that 32 yields the highest precision by a significant margin, followed by 4, whereas 8 and 16 perform poorly. Potential explanations might be the early stage of training, or the difference in the average number of objects per image for the original COCO dataset (8) and HGP (3) [7] [12].

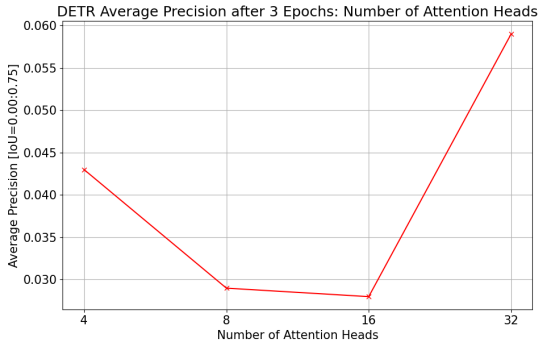

Figure 7: (left) DETR Average Precision for varying numbers of Attention heads.

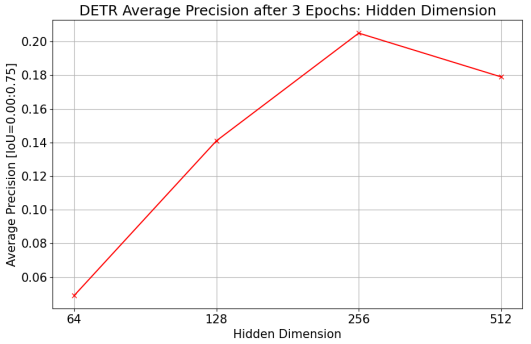

Figure 8: (left) DETR Average Precision for varying Embedding/Transformer (Hidden) dimensions.

6.2.4 Embedding/Transformer Dimension and Feed-forward Dimension

Figure 8 suggests 256 as a reasonable Hidden Dimension, followed by 512, which is aligned with DETR's default value. Figure 9 shows a similar result, with DETR's default FFN dimension being 2048, but with a zigzag pattern. We hypothesize, that 512 might be a reasonable FFN dimension as well, because it could help to prevent the complex DETR architecture from overfitting to a smaller dataset.

6.2.5 Discussion

In general, it's interesting to contrast DETR's default values with our findings. For the Embedding/Transformer/Hidden dimension, the feed-forward dimension and the batch size, our results align, even though the batch size presents an interesting trade-off between speed and precision. For the number of queries and attention heads, our results diverge from the default values. We hypothesize, that this is likely due to the early stage of training during measurement, but also because of the differences in size and classes in our datasets. Less objects per image suggest, that less object queries would be sufficient, whereas a low number of queries might inhibit generalization to varying numbers of objects during inference, because the model might overfit to the low object count per image in the HGP dataset (3). With respect to attention heads, our results suggest 32, which intuitively seems very high, considering it scales down the dimension of each attention head dramatically. Future work for increased number of epochs will help determine the validity of our hypotheseses.

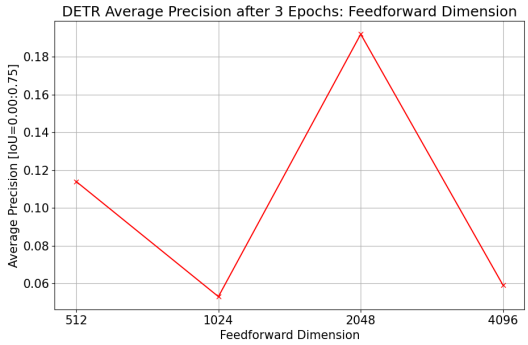

Figure 9: (left) DETR Average Precision for varying Feed-forward dimensions.

6.3 Visualizations

To create visualizations for our model predictions, and its underlying attention mechanisms, we used the techniques described in DETR's hands on Colab Notebook [13].

For the predictions visualized, we used the following set of hyperparameters to train on our Coder environment, described in Subsection 6.1: batch_size=2, epochs=500, backbone='resnet18', dim_feedforward=2048, hidden_dim=256, nheads=32, num_queries=20, dataset_file='hgp'; we set the remaining hyperparameters to their default value.

For our visualizations, we used the checkpoint obtained after 400 epochs, since the training log clearly suggested, that the model was overfitting to the training data limited in the amount of pictures.

6.3.1 Predictions

Figure 10 shows the general validity of our approach to train DETR with a custom dataset of Hands, Guns and Phones. It accurately classifies and locates both hands and the phone in a real-world scenario. Through experimentation with different images obtained from random websites, we found, that our model generalizes well to different sizes, lighting and colors and perspectives, as long as the image contains up to 2 hands and phone or gun. However, we identified to common failure cases:

Failure case - Combination of classes Gun and Phone. As Figure 11 illustrates, our model typically fails to detect objects of the class phone in an image, whenever an object of the class gun is present. We experimented with multiple instances of this class combination, and even if the phone is clearly visible, of reasonable size and central to the image, our model fails to detect it even with low probability. We hypothesize, that this is a feature of overfitting to our training dataset. The HGP dataset typically contains images of people carrying either a gun or a phone, or of only gun(s) or only phone(s). Perhaps, the spatial features of a gun are more distinct and unique than the features of phones, which exhibit shapes similar to other objects. This could explain, why our model is picking up on guns first.

Failure case - Multiple instances of the same class. Figure 12 shows another common failure case: If our model is presented with samples, which contain more than one instance of a gun or a phone, or more than two instances of a hand, it is likely to correctly classify and locate only a group of objects, but not the individual objects. We suspect, that one potential reason might be the low number of queries (20) in our configuration, as opposed to default queries. In our first training run, we even operated with only 5 queries, because Subsection 6.2 misleadingly suggested so. After some sample predictions, we decided to increase the number to 20 queries,

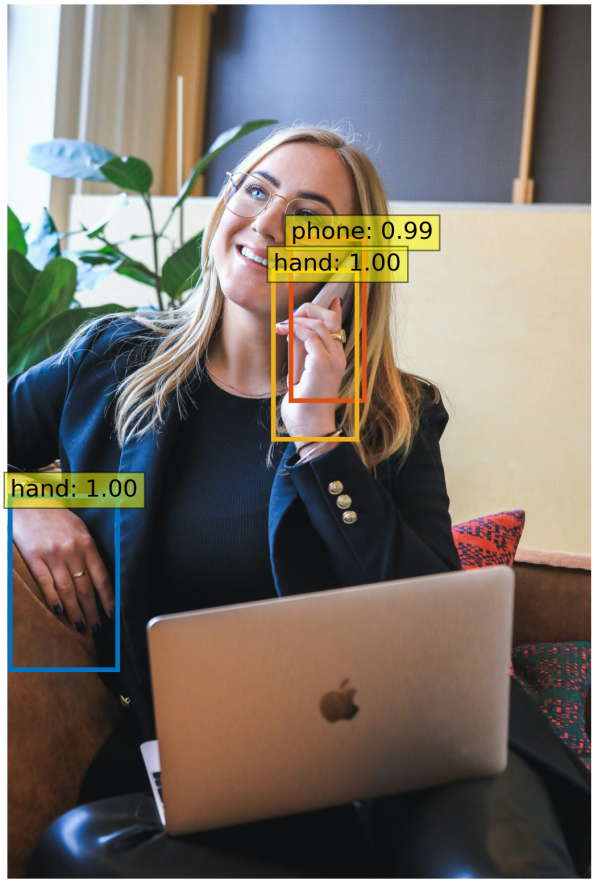

Figure 10: (left) Visualization - correct sample predictions of both hands and phone. Image Credit [19]

Figure 11: (left) Visualization - failure case: Combination of Gun and Phone. Image Credit [21]

Figure 12: (left) Visualization - failure case: Multiple instances of the same class. Image Credit [20]

which might still be suboptimal. Another potential explanation is the overfitting to our training data, which mostly contains 1 gun or phone, sometimes accompanied by 1-2 hands, which seems to be the range our model is confident with, as illustrated in Figure 10.

6.3.2 Attention

The transformer encoder-decoder architecture and the attention mechanism described in Subsection 3.5 applied to Object Detection is one of the central contributions of DETR, and one of its advantages. Figure 13 shows a visualization of the self-attention in the last encoder layer, which conceptually contains the memory handed into the cross-attention layer of the decoder. It provides intuition, how the encoder contributes to object separation. Each of the 4 pixels seems to aggregate most of its information with respect to a certain region of interest, corresponding roughly to the shapes of objects in the image. For example, the first pixel (280, 200) clearly attends to the other pixels of the gun. The other pixels seem to be attending to broader regions of interest, but all exhibit a notion of the hands in the center of the image.

After propagating the sample through the decoder, the attention becomes more focused in Figure 14, which illustrates the attention of certain queries in the last decoder layer. Each query clearly attends to the pixels in the image which are relevant for their object of interest. The visualizations for query 2 and 11 even suggest, why the bounding box for the gun extends farther downwards than necessary. The attention mechanism seems to connect the ammunition in the hands with the gun itself, therefore attending to the ammunition as well, and then extending the bounding box to include the ammunition as well. For the hands, a similar pattern occurs, where particularly query 17 seems to attend to part of the right hand as well. Similar patterns might contribute to the incorrect classification of multiple instance of the same class, particularly if they are too similar.

7 Conclusion and Future Work

We have shown how to apply the DETR model to a custom dataset by relying on interfaces defined by PyTorch, Torch Vision and pycocotools [14] [43] [35]. We further demonstrated a superficial hyperparameter search with limited computational resources, and its shortcomings: The combination of a pretrained backbone with notion of 1000 different classes with a loss function and an accuracy metric, which is focused around only Hands, Guns and Phones is difficult to evaluate after only 3 epochs. However, even with potentially suboptimal hyperparameters, we have shown how to train a custom DETR model, that generalizes well in the scope of the training dataset. A more extensive hyperparameter search, or a larger and more diverse dataset, will likely improve our results in the future.

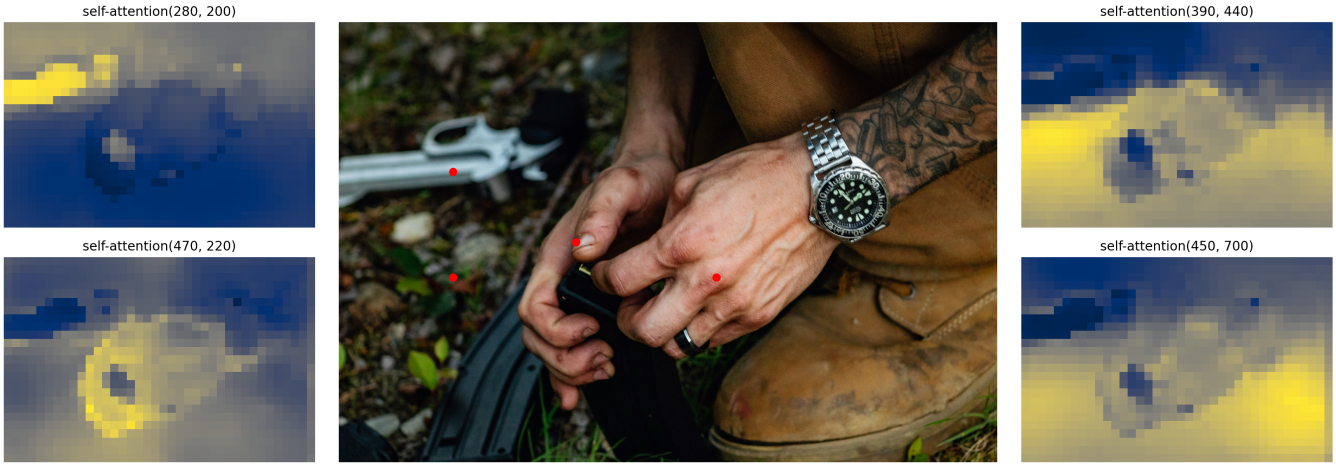

Figure 13: (left) Visualization: Self-attention layer in the last encoder layer. For 4 different sample pixels (red dots), the attention to each pixel in the image is illustrated with yellow indicating high attention, and blue indicating low attention, respectively. Image Credit: [18]

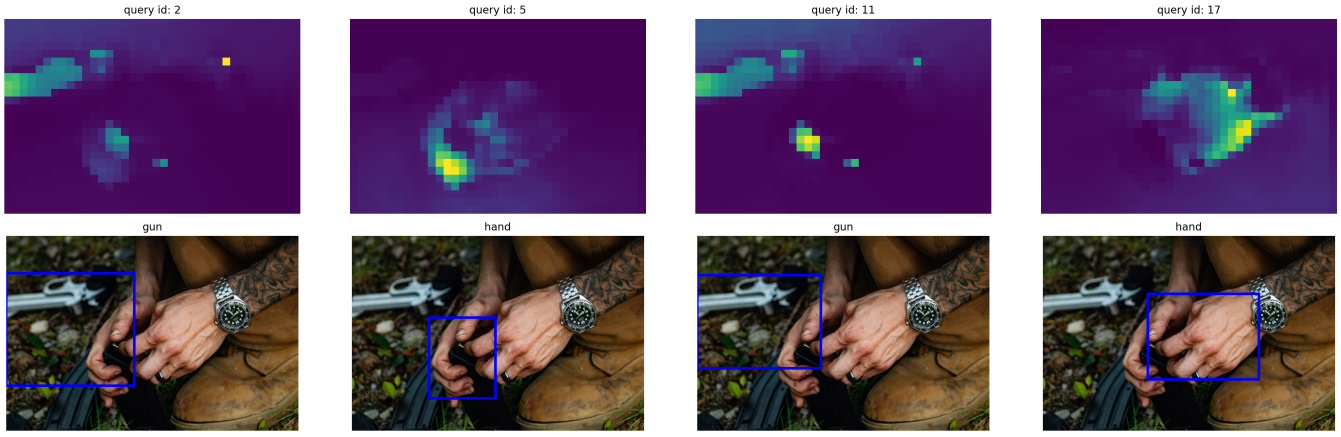

Figure 14: (left) Visualization: Attention weights of the last decoder layer. This corresponds to visualizing, for each detected object, which part of the image the model was looking at to predict this specific bounding box and class. Image Credit: [18]

REFERENCES

[1] R. Al-Rfou, D. Choe, N. Constant, M. Guo, and L. Jones. Character-level language modeling with deeper self-attention, 2018.

[2] Ananconda, Inc. conda-forge:pycocotools. https://anaconda.org/conda-forge/pycocotools. [Online; accessed 2024-05-17].

[3] J. L. Ba, J. R. Kiros, and G. E. Hinton. Layer normalization, 2016.

[4] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate, 2016.

[5] N. Carion, F. Massa, G. Synnaeve, N. Usunier, and A. K. anc Sergey Zagoruyko. End-to-end object detection with transformers. https://doi.org/10.48550/arXiv.2005.12872, May 2020.

[6] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling, 2014.

[7] COCO Consortium. Coco - common objects in context. https://cocodataset.org/. [Online; accessed 2024-05-15].

[8] T. Contributors. Pretrained resnets on pytorch. https://pytorch.org/vision/stable/models/resnet.html. [Online; accessed 2024-05-18].

[9] N. Corporation. System specifications nvidia a100. https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/. [Online; accessed 2024-05-17].

[10] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

[11] M. A. Duran-Vega, M. Gonzalez-Mendoza, L. Chang, and C. D. Suarez-Ramirez. Tyolov5: A temporal yolov5 detector based on quasi-recurrent neural networks for real-time handgun detection in video. https://paperswithcode.com/paper/tyolov5-a-temporal-yolov5-detector-based-on, November 2021.

[12] D.-V. et al. Hgp (hands guns and phones dataset). https://paperswithcode.com/dataset/hgp, November 2021. [Online; accessed 2024-05-15].

[13] I. Facebook. Detr's hands-on colab notebook. https://colab.research.google.com/github/facebookresearch/detr/blob/colab/notebooks/detr_attention.ipynb. [Online; accessed 2024-05-18].

[14] F. A. R. L. (FAIR). Tensors and dynamic neural networks in python with strong gpu acceleration. https://github.com/pytorch/pytorch. [Online; accessed 2024-05-15].

[15] I. GitHub. Coder. https://github.com/coder. [Online; accessed 2024-05-17].

[16] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, 2016. doi: 10.1109/CVPR.2016.90

[17] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9, 1997.

[18] Jay Heike. Silver colored analog watch. Unsplash. [Online; accessed 2024-05-18].

[19] Magnet.me. Woman in blue long sleeve shirt sitting on brown couch. Unsplash. [Online; accessed 2024-05-18].

[20] maxx. Black semi-automatic airsoft pistol on white textile. Unsplash. [Online; accessed 2024-05-18].

[21] Kadyn Pierce. A toy gun and a cell phone sitting on a table. Unsplash. [Online; accessed 2024-05-18].

[22] Y. Kim, C. Denton, L. Hoang, and A. M. Rush. Structured attention networks, 2017.

[23] M. Krichen. Convolutional neural networks: A survey. *SIGGRAPH Computer Graphics*, 12(151), July 2023. doi: 10.3390/computers12080151

[24] H. W. Kuhn. The hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2, 1955. doi: doi/10.1002/nav.3800020109

[25] T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollár. Focal loss for dense object detection, 2018.

[26] Y. Liu, T. Han, S. Ma, J. Zhang, Y. Yang, J. Tian, H. He, A. Li, M. He, Z. Liu, Z. Wu, L. Zhao, D. Zhu, X. Li, N. Qiang, D. Shen, T. Liu, and B. Ge. Summary of chatgpt-related research and perspective towards the future of large language models. *Meta-Radiology*, 1(2):100017, Sept. 2023. doi: 10.1016/j.metrad.2023.100017

[27] C. Lüscher, E. Beck, K. Irie, M. Kitza, W. Michel, A. Zeyer, R. Schlüter, and H. Ney. Rwth asr systems for librispeech: Hybrid vs attention. In *Interspeech 2019*, interspeech_2019. ISCA, Sept. 2019. doi: 10.21437/interspeech.2019-1780

[28] Marco Lorenz. Hgp detr. https://github.com/lorenz369/hgp_detr. [Online; accessed 2024-05-17].

[29] Meta AI. Object detection. https://paperswithcode.com/task/object-detection. [Online; accessed 2024-05-15].

[30] Meta Research. End-to-end object detection with transformers. https://github.com/facebookresearch/detr. [Online; accessed 2024-05-15].

[31] NERSC. Nersc documentation - perlmutter architecture. https://docs.nersc.gov/systems/perlmutter/architecture. [Online; accessed 2024-05-15].

[32] NVIDIA Corporation. Cuda zone. https://developer.nvidia.com/cuda-zone. [Online; accessed 2024-05-15].

[33] A. P. Parikh, O. Täckström, D. Das, and J. Uszkoreit. A decomposable attention model for natural language inference, 2016.

[34] N. Parmar, A. Vaswani, J. Uszkoreit, Łukasz Kaiser, N. Shazeer, A. Ku, and D. Tran. Image transformer, 2018.

[35] Piotr Dollar and Tsung-Yi Lin. Coco api - dataset @ http://cocodataset.org/. https://github.com/cocodataset/cocoapi/tree/master/PythonAPI/pycocotools. [Online; accessed 2024-05-15].

[36] Python Software Foundation. pip:pycocotools. https://pypi.org/project/pycocotools/. [Online; accessed 2024-05-17].

[37] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever. Language models are unsupervised multitask learners. 2019.

[38] S. Ren, K. He, R. Girshick, and J. Sun. Faster r-cnn: Towards real-time object detection with region proposal networks, 2016.

[39] H. Rezatofighi, N. Tsoi, J. Gwak, A. Sadeghian, I. Reid, and S. Savarese. Generalized intersection over union. June 2019.

[40] N. Srivastava. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958, June 2014. doi: 10.3390/computers12080151

[41] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks, 2014.

[42] G. Synnaeve, Q. Xu, J. Kahn, T. Likhomanenko, E. Grave, V. Pratap, A. Sriram, V. Liptchinsky, and R. Collobert. End-to-end asr: from supervised to semi-supervised learning with modern architectures, 2020.

[43] Torch Contributors. Torchvision 0.18 documentation. https://pytorch.org/vision/main/generated/torchvision.datasets.VisionDataset.html. [Online; accessed 2024-05-15].

[44] Y.-H. H. Tsai, S. Bai, P. P. Liang, J. Z. Kolter, L.-P. Morency, and R. Salakhutdinov. Multimodal transformer for unaligned multimodal language sequences. In A. Korhonen, D. Traum, and L. Màrquez, eds., *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pp. 6558–6569. Association for Computational Linguistics, Florence, Italy, July 2019. doi: 10.18653/v1/P19-1656

[45] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need, 2023.

[46] Z. Zou, K. Chen, Z. Shi, Y. Guo, and J. Ye. Object detection in 20 years: A survey, 2023.