# Object Detection Model Pruning with Weight Interpolation

Jacob Samar
jsamar1@vols.utk.edu
University of Tennessee
Knoxville, Tennessee, USA

Riley Tavassoli
rtavasso@vols.utk.edu
University of Tennessee
Knoxville, Tennessee, USA

## ABSTRACT

We introduce a method of pruning model weights that does not require the analysis of neuron activation, instead relying on classical interpolation algorithms to reduce the number of parameters. Pruning, in this paper, refers to the action of reducing the size and changing the values of weights from the pretrained model and applying them to a smaller model. In literature, the definition is slightly different, but the problem domain is similar. This paper discusses the multiple methods attempted, the resultant accuracy obtained, and the parameter size reduction. A variety of interpolation methods were used and compared against the pruned model with random initialization and the full-sized pretrained model given by the authors of the paper. In this paper, we test our method on the object detection model DeTr (Detection Transformer) and show a marginal improvement in validation loss with our method compared to random initialization.

## KEYWORDS

pruning, neural network, machine learning

**ACM Reference Format:**
Jacob Samar and Riley Tavassoli. 2021. Object Detection Model Pruning with Weight Interpolation. In *Proceedings of ACM Conference (Conference'17)*. ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/1122445.1122456

## 1 INTRODUCTION

Models published in papers often come with pretrained weights that the authors used. To perform experiments with their model, it is as easy as downloading the model and loading the parameters. Their results can be easily reproduced in this way, but if a reader wants to configure the original model differently and compare results, the pretrained weights cannot be used. This largely wastes computational resources by forcing a complete retraining of the model from random initialization, and with our method, we provide a solution to this problem.

The parameters of a model can be downsampled to the desired size while preserving some of the learning in the original weights. This allows for a lighter, pruned model to be trained to convergence faster than randomly initializing the weights. This leads to faster training time and computational cost savings when iterating across different model configurations. The way in which it is downsampled is crucial in maintaining coherence of pretrained weights, and so we test multiple downsampling algorithms to see which best achieve this goal. Previous work has been done on this topic, but this method downsamples with no objective function being minimized or analysis of activations which has yet to be done.

## 2 PREVIOUS WORK

There is a wealth of literature on different methods to prune a neural network in an attempt to reduce the needed computation and memory. Several approaches attempt to achieve this through generating a sparse neural network. One such approach implements a sensitivity parameter that quantifies the change in the network with the parameter to be removed and gradually adjusts low-sensitivity parameters to zero[9]. Another uses an $L_0$ regularization to generate the sparse network[8]. However, both of these methods are only targeting changes in the weights of the network, not the structure of it.

A method proposed by Li et al. reduces computation costs and maintains network accuracy by removing filters that have low relative performance in a layer[7]. Most similarly to our method, a team from Cornell devised an approach using low-rank matrix approximation[4]. Our method does not seek to approximate the full-rank weight matrix as they do, but rather use classical methods for interpolation such as nearest, linear, bi-linear, as well as conventional machine learning downsampling methods such as average and max pooling. One compression scheme developed for deep, convolutional neural networks performs global average pooling, pruning, truncated SVD, and quantization to reduce the model size. Their use of max pooling is only before the first convolutional layer, as the first layer has a greater number of parameters to reduce[6]. This is different from our approach in that respect, and it only explored the use of max pooling. Another prominent method was developed by Han et al. that assigned scores to neurons based on activation value and pruned those with low activation[5]. This method requires substantial analysis on the underlying network which our method attempts to avoid.

## 3 TECHNICAL APPROACH

The model used in this paper, Detection Transformer, works as an object detection model built atop a vision transformer[3]. It works by using a convolutional backbone, RESNET-50 by default, to extract relevant features from the input image. For this paper, the weights in the convolutional part of the network were left unchanged. Our goal was to take the pretrained weights given by the authors of the paper and fine-tune them on a video game object detection dataset separate from the training domain. The model converged sufficiently and achieved good results which first sparked the question of how important the 41 million parameters in the pretrained model were in this task. Would 30 million achieve the same? Most pruning methods require an analysis of the weights or activations, or as seen in our literature review, a lower dimensional approximation of the weights. These methods are less intuitive than simply interpolating the weights to the desired size, thereby reducing the number of parameters and hopefully maintaining the structure of the weights.

From the convolutional backbone, it then goes into an encoder-decoder vision transformer, outputting a matrix of transformed features which is then used in the two heads of the network, computing the object class and bounding box coordinates. In this paper, we halve the dimension of the transformer model, 256 by default, to 128. We also halve the transformer's fully-connected layer dimensions, 2048 by default, to 1024. This model with halved dimensions is the custom model we test our interpolation methods on. This brings the total parameters down from 41 million to 28 million. The reason the parameters do not shrink by half is because the convolutional backbone is not downsampled, and so that part of th network uses the same number of parameters in both pruned an pretrained models. The interpolation method is the independen variable across experiments.

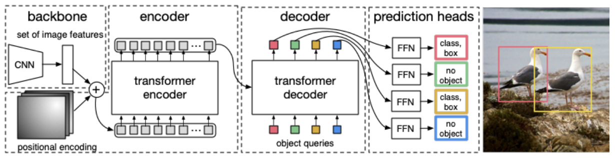

**Figure 1: DeTr Architecture**

In this paper, we show results for nearest-exact (PyTorch), area, average pooling, and max pooling. For each interpolation method, a comparison is done layer by layer for the pretrained model and the custom model. If the shape of the custom model's weights for that layer mismatch the pretrained model's, the weights from that layer in the pretrained model are downsampled using one of the methods described above and then applied to the custom model. After going through each layer and replacing the weights in the custom model with the downsampled pretrained weights, the model is trained for 3000 steps on the fine-tuning task. The validation loss is the metric of interest in this paper, showing how weight interpolation can transfer the learned parameters from the pretrained model to a smaller network and learn faster in comparison to a random initialization.

## 4 DATASET AND IMPLEMENTATION

The default model for DeTr is pretrained on the COCO 2017 detection dataset, which contains 118k training images and 5k validation images. A DeTr model has three hyperparameters: learning rate, learning rate for the backbone network, and weight decay. As our goal is to implement an effective way to down sample the network to contain less parameters, we left these at their default values (1e-4, 1e-5, and 1e-4, respectively).

The object detection domain we are targeting is virtual rocks in the online game Old School RuneScape. In this game, there are 9 classes of rocks to identify and one null class for no rocks. The dataset [2] originally had 158 labelled images. These were split 80/20 into training and validation datasets. From there, the training partition of the dataset was augmented at random with horizontal flips and zoom crops ranging from 0 to 20%. With these augmentations, 3 augmented images were generated from each training example resulting in a total dataset size of 412 images split into 381 images for training and 31 for validation. This process of random data augmentation and repeated sampling was automated using Roboflow [1].

As described in our Technical Approach, we perform 8 experiments to investigate the effect of our interpolation. All code is executed within Google Colab using GPU acceleration with the pretrained model being pulled from the Hugging Face Transformers library and the training code written using PyTorch Lightning. We load the data in batches of 4 images for training and 2 images for validation. Any larger batch size leads to running out of memory.

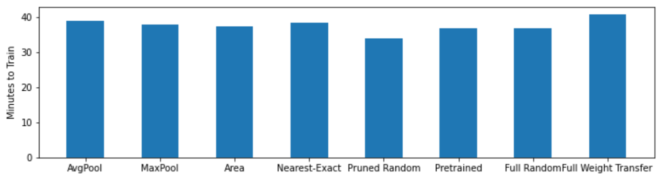

**Figure 2: Time to Train**

There was little difference in the train time of the models despite the large difference in number of parameters. This likely is a result of using Google Colab's GPU accelerator, which provides different GPUs as they are available. Still, the pruned models benefit from smaller physical size on disk and faster inference time. A breakdown of the number of parameters per model follows.

(1) Full Model: default weights loaded
**Parameters:** 41288719
(2) Full Model: random weight initialization
**Parameters:** 41288719
(3) Full Model: weight transfer
**Parameters:** 41288719
(4) Downsampled Model: random weight initialization
**Parameters:** 28128975
(5) Downsampled Model: interpolation (nearest-exact)
**Parameters:** 28128975
(6) Downsampled Model: interpolation (area)
**Parameters:** 28128975
(7) Downsampled Model: max pooling
**Parameters:** 28128975
(8) Downsampled Model: average pooling
**Parameters:** 28128975

## 5 EXPERIMENTS AND RESULTS ANALYSIS

For each model describe above, the process of training the model required an initialization of the Detr class with the correct parameters. The weights were then replaced, if necessary, before passing the model to the PyTorch Lightning Trainer and viewing the results in TensorBoard. For each experiment, training was conducted only once except for a few exception where Colab was resource-limited leading to outlier results in training time. Those models were retrained and the following results represent the one instance of training for each model. For the three full sized models, we tested random initialization and weight transfer and compared against the pretrained baseline. Weight transfer in this context refers to the transferring of only the parameters from the pretrained model into a model with the same architecture but uninitialized weights.

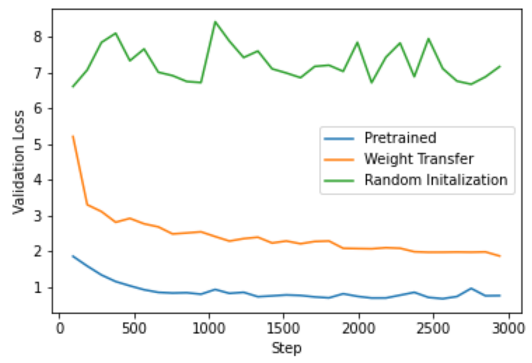

**Figure 3: Full Sized Models**

While this may sound like the pretrained model was replicated, we discovered unique interactions in PyTorch that produced different results when just transferring weights. For this reason, we include it in the comparison of validation loss for different methods of weight initialization. This of special importance because it was with this method of weight transfer that downsampled weights were applied to the smaller model. This could indicate that the loss for the pruned models is artificially inflated. The optimal solution that we could not implement in this paper would have been a complete copy of the state dictionary, but instead only the parameters of the model were able to be copied and downsampled. In **Fig. 3**, this effect can be observed in how the same model architecture differs in validation loss substantially despite having the weights from the pretrained model transferred to the uninitialized model. Still, it does far better than the randomly initialized full sized model which seems to oscillate at high loss with no stable learning occurring.

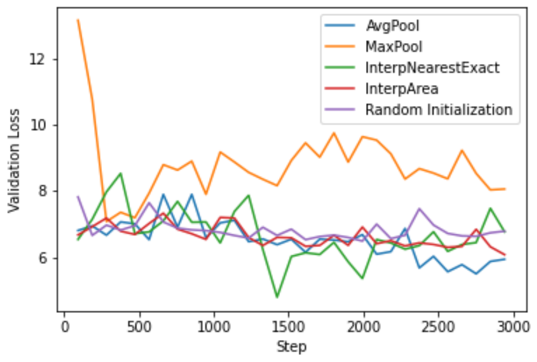

**Figure 4: Pruned Models**

As seen in **Fig. 4**, max pooling did much more poorly than the other downsampling methods. Average pooling and area interpolation both performed better than random initialization by a marginal amount. Area interpolation in this paper is the algorithm implemented in PyTorch's interpolation function that works as a type of adaptive average pooling. Because of this, it's expected that they perform similarly, which they do. Nearest-Exact interpolation finished training with almost the same validation loss as the random initialization, but it varied much more while training. This indicates that the weights from nearest-exact interpolation were poorly initialized. Notably, the random initialization of weights on the pruned model size performed on par or even slightly better than the full model when randomly initialized. This could indicate that both sizes of models were equally capable of learning the required function in the virtual domain to correctly detect and classify. Further testing would be required to validate this hypothesis by showing how successively smaller models handle random initialization. Largely, it can be said that weight interpolation is an unimpressive solution to downsizing models while preserving learning. Despite this small difference observed in **Fig. 4**, it could be that the initializations diverge and one performs far better in the later stages of training, but for this paper, there was not enough time or compute available to converge each model completely.

## 6 CONCLUSION

In this paper, we were able to show a marginal improvement in validation loss over random weight initialization for both area interpolation and average pooling of a pretrained model. The model size was reduced from 41 million parameters to 28 million parameters using our method. The small improvement in performance suggests that our downsampling methods were not ideal, but could be improved upon in the future. Furthermore, this paper has shown that pure interpolation with no analysis of the underlying activations is a poor method of preserving parameter coherence. Future iterations of this could explore how full-rank weight approximation can be achieved with purely interpolative methods.

## REFERENCES

[1] [n. d.]. Roboflow. https://app.roboflow.com/. Accessed: 2022-04-30.
[2] [n. d.]. Rocks Dataset. https://www.dropbox.com/s/ifpj2wz2wgbz4sw/rocks.tar.gz?dl=0. Accessed: 202-04-30.
[3] Nicolas Carion, Francisco Massa, Gabriel Synnaeve, Nicolas Usunier, Alexander Kirillov, and Sergey Zagoruyko. 2020. End-to-end object detection with transformers. In *European conference on computer vision*. Springer, 213–229.
[4] Jerry Chee, Megan Renz, Anil Damle, and Chris De Sa. 2021. Pruning Neural Networks with Interpolative Decompositions. *arXiv preprint arXiv:2108.00065* (2021).
[5] Song Han, Huizi Mao, and William J. Dally. 2015. DEEP COMPRESSION: COMPRESSING DEEP NEURAL NETWORKS WITH PRUNING, TRAINED QUANTIZATION AND HUFFMAN CODING. *arXiv preprint arXiv:1510.00149* (2015).
[6] Ting-Yun Hsiao, Yung-Chang Chang, Hsin-Hung Chou, and Ching-Te Chiu. 2019. Filter-based deep-compression with global average pooling for convolutional networks. *Journal of Systems Architecture* 95 (2019), 9–18.
[7] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. 2016. Pruning filters for efficient convnets. *arXiv preprint arXiv:1608.08710* (2016).
[8] Christos Louizos, Max Welling, and Diederik P Kingma. 2017. Learning sparse neural networks through $L\_0$ regularization. *arXiv preprint arXiv:1712.01312* (2017).
[9] Enzo Tartaglione, Skjalg Lepsøy, Attilio Fiandrotti, and Gianluca Francini. 2018. Learning sparse neural networks via sensitivity-driven regularization. *advances in neural information processing systems* 31 (2018).

## A CODE DESIGN

Our code was adapted from an example usage of fine-tuning DeTr[1]. The code for the loading a dataset, creating a DeTr feature extractor, and training the model was used unedited other than small adaptations for our dataset.

The Detr class, which is used for the creation of the models, was adapted to allow for a choice between the generation of a pretrained model or a custom model using a passed in configuration. The class is described by **Algorithm 1** with lines 6-9 being altered for our experiments.

**Algorithm 1** Detr Class

```
1:  Start Detr Class
2:      procedure INIT(lr, lrBackbone, weightDecay, configuration, pretrained)
3:          self.learningRate ← lr
4:          self.learningRateBackbone ← lrBackbone
5:          self.weightDecay ← weightDecay
                          ▷ This option for custom models was added
6:          if pretrained is False then
7:              self.model ← custom detr model
8:          else
9:              self.model ← pretrained detr model
10:     procedure FORWARD(pixelValues, pixelMask)
11:         output ← single model output of 2 parameters
12:         return output
13:     procedure COMMONSTEP(batch, batchIdx)
14:         outputs ← model outputs with of batch
15:         loss ← loss on these outputs
16:         lossDict ← lossDict on these outputs
17:         return loss, lossDict
18:     procedure TRAININGSTEP(batch, batchIdx)
19:         loss,lossDict← commonStep(batch, batchIdx)
20:         Log the training loss and values in lossDict
21:         return loss
22:     procedure VALIDATIONSTEP(batch, batchIdx)
23:         loss,lossDict← commonStep(batch, batchIdx)
24:         Log the validation loss and values in lossDict
25:         return loss
26:     procedure CONFIGUREOPTIMIZER
27:         optimizer ← adam optimizer using hyperparameters
28:         return optimizer
29:     procedure TRAINDATALOADER
30:         return trainDataloader
31:     procedure VALDATALOADER
32:         return valDataloader
33: End
```

Finally we created three functions to enable our downsampling on the weights: rgetattr, rsetattr, and customWeights. The rsetattr and rgetattr are both utility functions for accessing nested properties, specifically the weights of a layer inside a model, adapted from a stackoverflow post[2]. The customWeights function is our main contribution of code and provides the main functionality of our experiment. It is outlined in **Algorithm 2**. It takes in a pretrained model, a custom model, and the type of downsampling to perform ("Maxpool", "AvgPool", "Area Interp", "Nearest-Exact Interp", "None"). For every layer in both the pretrained and custom models, it will set the weights from the pretrained to the custom. If the shape of the layers mismatches, which is the case in our downsampled models where the encoder and decoder dimensions are changed from 2048 to 1024, the method will perform the specified downsampling prior to setting the weights in the custom model.

**Algorithm 2** customWeights

```
1:  procedure CUSTOMWEIGHTS(pretrainedModel, customModel, downsampleType)
2:      pretrainedLayers ← names + weights from pretrainedModel
3:      for layer in customModel do
4:          if layer.name in pretrainedLayers then
5:              pretrainedSize ← dims of pretrainedModel layer.
6:              customSize ← dims of layer.
7:              if pretrainedSize != customSize then
8:                  input ← weights in pretrainedModel.
9:                  if downsampleType == "Area" then
10:                     output ← Area interp on input
11:                 if downsampleType == "Nearest-Exact" then
12:                     output ← nearest interp on input
13:                 if downsampleType == "Max Pooling" then
14:                     output ← Max pooling on input
15:                 if downsampleType == "Ave Pooling" then
16:                     output ← Ave pooling on input
17:             else
18:                 output ← All weights from pretrained
19:             layer.weights ← set to output
```

## B WORKLOAD DISTRIBUTION

### B.1 Riley Tavassoli

Wrote and modified all code in Colab notebook, performed all experiments, analysis, and graph generation. Authored abstract, introduction, technical approach, and results

### B.2 Jacob Samar

Performed small code changes and general QA on the experiment software. Wrote large portions of previous work, dataset and implementation, and code design. Provided editing and smaller additions to other portions of the paper.

[1]https://github.com/NielsRogge/Transformers-Tutorials/blob/master/DETR/Fine_tuning_DetrForObjectDetection_on_custom_dataset_(balloon).ipynb
[2]https://stackoverflow.com/questions/31174295/getattr-and-setattr-on-nested-subobjects-chained-properties