Reimplementation of SlimYOLOv3 in TensorFlow

9035

Abstract

YOLOv3 is an object detection and image classification network that is designed for speed while still providing speeds similar to state of the art one-stage networks such as RetinaNet (Redmon and Farhadi 2018). SlimYOLOv3 is a pruned version of YOLOv3 with additional pooling layers for better spatial awareness (Zhang, Zhong, and Li 2019). My experiment replicates the main result in their paper showing that channel pruning on the YOLOv3 model can provide effective compression of the model while not causing a large detriment to the performance of the algorithm. To do this, I reconstructed the YOLOv3 model framework in TensorFlow and trained YOLOv3-SPP3, specified as the unpruned model in the SlimYOLOv3 paper (Zhang, Zhong, and Li 2019). Next, I ran one pruning step and fine tuning using a sparsity of 50% to get the paper's first iteration: SlimYOLOv3-SPP-50. Pruning was able to drive down the number of parameters by 54% without a significant drop in accuracy. Unlike the paper, my fine-tuned model performed better than the original, opening a problem on how to find optimal hyperparameters in pruning.

Background Information

Deep Feature Aggregation

Deep feature aggregation is a method to aggregate "discriminative features through sub-network and sub-stage cascade" (Li et al. 2019). Discriminative features are the features that can be used to distinguish one object from another and thus are features that are learnable by a discriminator (Stuhlsatz, Lippel, and Zielke 2010). Sub-networks and sub-stages are smaller model backbones that are connected together to form the bigger backbone using spatial pyramidal pooling (SPP) and residual connections.

SPP is a pooling algorithm that improves the resilience of the model to different sized inputs and distorted images (He et al. 2014). To do this, it divides output of a convolution layer into smaller subgrids with different sizes, called octaves. For example, a 520x520 input can be divided into 40 13x13 octaves, 65 9x9 octaves, and 117 5x5 octaves. These octaves are then maximum pooled. This way, features from each of the scales (13, 9, 5, and 1) are aggregated deep in the network. The authors of the paper suggest putting it just before the last couple of layers in the network (He et al. 2014).

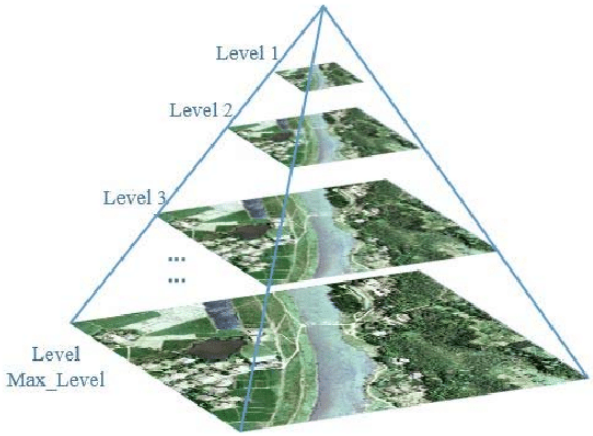

Figure 1: An image pyramid, the central structure in spatial pyramidal pooling. The original image is downsampled and distorted at various scales to produce the different octaves. Figure from (Guo et al. 2016).

The original SPP paper uses the layer and to aid with object detection. DFANet uses the SPP and residual connections to create high quality object segmentations. The authors started with the Xception network and added more residual connections between different sub-networks and sub-stage aggregations. These residual connections concatenated features from higher up in the network with deeper ones before passing it into the next layer. They allow fine-grained features from beginning of the model to aid in segmentation, as opposed to just using coarse-features deep in the network (Li et al. 2019). This allows for efficient high quality pixel-by-pixel segmentation because the features at many different scales contribute more evenly to the final segmentation result than traditional segmentation algorithms which don't have this many connections and have to trade off between detail and spatial information. In addition to allowing for more stable predictions, residual connections also aid during the training of the model. They can help the model

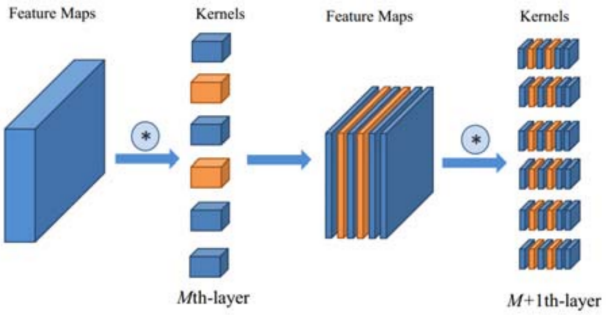

Figure 2: Removing just several channels at each convolution layer will decrease the number of weights in the model drastically because the weights that depended on the channels from higher in the model are no longer necessary either and are removed as well. This means that weights Figure from (Wang and Zhu 2019).

reach a better minimum because the loss functions of residual networks tend to be smoother (He et al. 2015).

SlimYOLOv3 uses the same principles as DFANet, such as SPP and residual connections, but the connections are not as dense, instead following the structure of ResNet and YOLO v2 following a repeating structure of residual downsampling blocks (He et al. 2015; Redmon and Farhadi 2016). The use of many more residual networks and high feature reuse in DFANet make it more similar in structure to DenseNet (Huang et al. 2018). Although high feature reuse may make the number of model parameters smaller, the overall amount of GPU memory that needs to be allocated to cache the tensors computed by residual connections until they are used deeper in the network can be large without proper memory management (Pleiss et al. 2017). Despite these improvements, residual connections should be used sparingly because they carry a high cost in dynamic memory. For the application of real time drone video object detection, high dynamic memory consumption would be undesirable, so a network with fewer residual connections (YOLOv3) was chosen to be the basis for the SlimYOLOv3 model.

Iterative Channel Pruning

Pruning is an established technique for reducing the size of the model without reducing the accuracy significantly (Cun, Denker, and Solla 1990). It utilizes an optimization that minimises the sum of the network's loss function plus the complexity of the network. After training with this new combined loss function, any component that doesn't meet a predefined threshold is cut from the model. There are two major types of pruning: weight pruning (using the number of weights as the complexity) and structural pruning (using channels or entire layers as the measure of the complexity of the model). Unfortunately, both types of pruning previously had poor performance on CNNs and would cause a large drop in accuracy when compressing the model.

More recent work devises a more stable method of channel pruning using L1 regularization to penalize large weights in the model before pruning off unnecessary channels (Liu et al. 2017). This improves the accuracy of compressed CNN models and gives them similar accuracy to the original models. Instead of one long training process with one optimization function, a training step done with L1 regularization after the normal training of the model, known as sparsity training. After the training process, channels will either contribute to the final prediction or will have negligible weights associated with them and can be safely removed from the network without a significant change in accuracy.

SlimYOLOv3 heavily relies on the Liu et al. (2017) modified process of channel pruning. The authors of the paper chose channel pruning over weight pruning in the creation of their model because channel pruning is more coarse and more efficient from a systems perspective. Weight pruning requires representing the model in sparse tensors as opposed to channel pruning which can represent a model in the same format as the original model, just with fewer weights. This eliminates the need of specialized architectures (software or hardware) for the pruned models (Zhang, Zhong, and Li 2019).

Previous DCNN channel pruning work merely demonstrates the effectiveness of the method on residual image classification models (Yang et al. 2019; He, Zhang, and Sun 2017; Liu et al. 2017). The main contribution of SlimYOLOv3 is that it demonstrates the method also equally effective on object detection models and models with spatial pyramidal pooling (Zhang, Zhong, and Li 2019).

It also demonstrates that channel pruning can be done iteratively. Iterative pruning provides better results than a single prune because using very low sparsity when training can lead to over-pruning of the model. This is because the fine tuning step of the training process is important for the model to relearn information from the lost weights without training from scratch. Instead of pruning the model with 95% sparsity, the authors did three stages: two with 50% sparsity and one with 70% sparsity to prevent losing too much data too quickly. More recent work shows that this iterative pruning approach increases accuracy in models in other domains as well (Paganini and Forde 2020). It demonstrates that it has more wide scale importance outside of image recognition and can be used just as effectively in other computer vision areas as well as (Paganini and Forde 2020).

Network Architecture Search

The traditional method of designing a neural network was for a researcher to use their intuition to come up with a network based on previous building blocks that were known to have an effect on the final constructed neural network architecture. However, using intuition will miss a lot of networks that are too complex for a human to design on their own. Network architecture search (NAS) algorithms are unsupervised machine learning algorithms that are able to automatically create a neural network architecture. It does this by running an optimization algorithm on a search space (Kyriakides and Margaritis 2020).

Most network search algorithms use one of five main categories of optimization methods: evolutionary, reinforce-

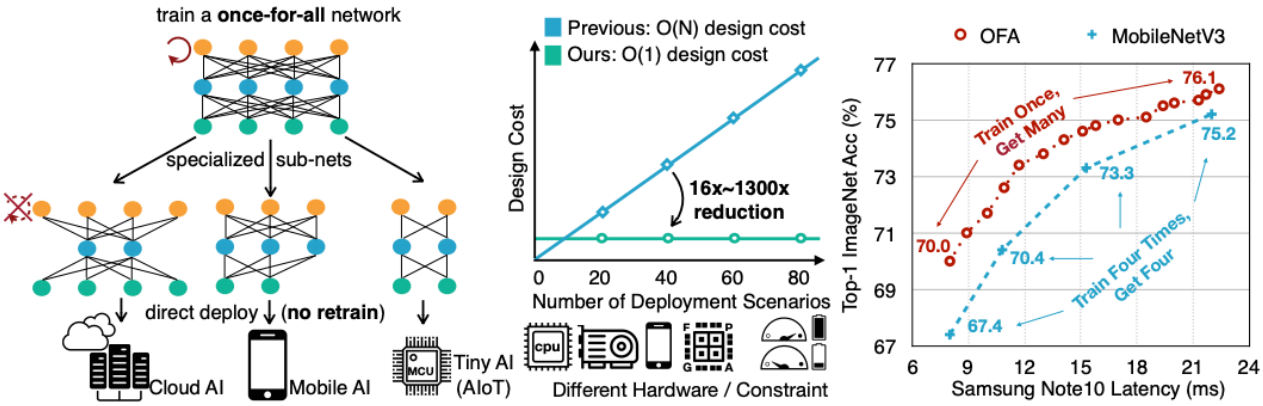

Figure 3: Simultaneous training and search of the Once-For-All network allows for efficient creation of many models with very little average cost per network. Figure from Once-For-All paper (Cai et al. 2020).

ment, Bayesian, one-shot, and meta-learning (Kyriakides and Margaritis 2020). Evolutionary algorithms mimic natural selection in biology: the "child" architectures are evaluated the most successful "parent" architectures are mutated and become the children for the next iteration. Meta-learning allows for the search algorithm to learn the weights of the algorithm while searching for it.

Once-For-All is a novel method of automatically designing models that outperforms other NAS algorithms. The main contribution of this paper, is an elegant new loss function that evenly trains many different subnetworks and optimizes them for different architecture constraints (Cai et al. 2020).

The paper uses a loss function that adds up the individual losses for each of the subnetworks and minimizes the total loss across the different subnetworks:

$$\min_{W_o} \sum_{arch_i} \mathcal{L}_{val}(C(W_o, arch_i))$$

(Cai et al. 2020). The purpose of this new loss function is to keep the accuracies of each of the subnetworks similar to each other, while training each of the subnetworks to be independent. This has a high upfront cost in terms of training time, but the overall benefit is that many subnetworks will be efficiently trained at the same time (Cai et al. 2020). In this way, it is a meta-learning NAS because the search and training are not separate steps. It is also similar to evolutionary in the step where the algorithm decides which subnetworks to choose for which constraints, because it mixes and matches layers that are used based on the loss in preparation of the pruning to the final subnetworks.

This is because Once-For-All is able to train many more subnetworks at the same time than compared to manually training each one. The experiment in the paper trains more than 10^19 different subnetworks at the same time, an unimaginable feat using traditional training methods (Cai et al. 2020). This is way too many subnetwork architectures to deal with, so instead of finding the loss for each and the paper instead compute the losses of only 16000 subnetworks and add them together to get the final loss, and they were able to show that this has a no relatively small effect on the final accuracy of the subnetworks (Cai et al. 2020).

Despite having the benefit of being able to generate and train many different models at once, network architecture search is not good for every scenario. The method's extremely high upfront computation cost requires many TPU clusters to locate any set of architectures on the same training data. Because of this, it is only viable when designing a model for many applications or many models for one application but on many different devices. It is not very useful if the target device is known during training, such as the drone example that motivates the SlimYOLOv3 paper. Since doing channel pruning alone can improve performance of a general purpose algorithm on a target device, it is a more efficient choice for creating a low power neural net unless simultaneous training of a slightly similar architecture is required.

Implementation Details

Despite the fact that there are many third party implementations of the YOLOv3 paper that the the authors based their architecture on in almost every machine learning platform, the only publicly available implementation of SlimYOLOv3 is the one on the official repository linked by the paper. To add to the confusion, SlimYOLOv3 used two separate implementations for the base YOLOv3 model: they trained the unpruned YOLOv3-SPP3 network that they designed using the original C implementation of YOLOv3 published with the paper (DarkNet), ran sparsity training in the PyTorch implementation released by Ultralytics Inc. (because it was easier to modify to add their L1 regularization term to the loss function), and fine tuned the model in DarkNet. Because of this, there was no unified codebase for SlimYOLOv3 that is able to do all of the tasks required to train the model through all three phases.

My re-implementation of SlimYOLOv3 alleviates these issues by replicating the experiment and bringing all 3 of these training phases into one unified repository. It is also more convienient for people who are trying to learn more about the model since the repository is more well documented and easier to use than the original paper's repository.

In addition to unifying the code base, creating a TensorFlow implementation opens up the ability for the model to be trained and used on Google proprietary hardware such as Cloud and Edge TPU, which can provide faster training. Although I haven't benchmarked the code base on a TPU, it is able to train and evaluate on a TPU.

Easier Dataset Loading and Data Preprocessing

The VisDrone2018-DET dataset that the original repository uses is not supported in DarkNet, PyTorch, or TensorFlow, implying the need of a conversion script in order to transform it into a format that the authors could have used for training their model, making their experiment hard to replicate. Despite this, I have not found any conversion script in their repository.

In order to ease the burden of setting up the repo, I made a dataset converter from the VisDrone dataset format to TensorFlow Datasets that only requires the Google Drive URLs of the dataset. Since the datasets format hasn't changed from year to year, the same converter has been tested on the VisDrone2019/2020-DET to ensure forward compatibility. Unfortunately, the dataset, although availible to the public, requires creating an account on the VisDrone website, identifying what organization you belong to, and agreeing to terms that do not allow for disseminating the dataset freely, so the process cannot be fully automatic.

Data preprocessing done by the YOLOv3 paper is very intensive and requires many different image transformations to be done in order to create more variety in the dataset. Because of the scale of this endeavor, I didn't use any nonessential preprocessing operations in my implementation of SlimYOLOv3. YOLO models require a ground truth label to store the expected anchor box for the prediction. This is a required preprocessing operation in order to train YOLO, so it is the only one that I included in my implementation. Despite this, the models that I trained in the experiment were reasonable, even though they did not meet the results demonstrated in the SlimYOLOv3 paper. This means that these operations only just help in the last couple of percentage points on the mean average precision of the generated bounding boxes and classes, and are not necessary for a majority of the results in the paper.

Modular Model Code

To ease the transfer of the same code base to some other model other than YOLOv3, all YOLOv3 specific code was put directly in the modeling folder. All of the other pieces of code can be adapted to be used in other models that come out in the future that may need to use channel pruning in TensorFlow.

Where possible, the two more complex DarkNet layers (`convolution` and `yolo` [reshaping and non-max supression]) were implemented in a way that was most similar to the original YOLOv3 paper. Unfortunately, due to differences between DarkNet and TensorFlow in how they handle variable shaped tensors, non-max supression in the `yolo` layer had to be treated specially in the training loop of the model.

To allow for better readability, the structure of the model as a whole was broken down into more readable building blocks instead of directly to the primitive layers. These blocks are the repeating units in the model that have very little variation from one point in the model to another.

Details of Pruning Algorithm

Although the primary focus of the SlimYOLOv3 paper is using channel pruning to reduce the size of YOLOv3 for drone video applications, the authors did not give many details on their channel pruning algorithm. In order to choose which channels to prune off from the model, they first select two hyperparameters (the proportion of channels to remove from the batch normalized convolution layers, and a threshold of layers to pass in order to not lose too much information from the model). Then, they iterate through every batch normalized convolution layer in the model and sort the channel weights for each of these layers. The proportion of the channels that need to be removed is multiplied by the number of channels in the layer and then this number of channels is removed from the lower end of this sorted list. Some extra logic is required to handling routing of the concatenation layers in the model, but sorting the channel weights to find out which channels to remove is the crux of the algorithm.

Training Procedure

This experiment only covers the unpruned model and first pruning round of the original SlimYOLOv3. This covers all of the four stages of their experiment, which are then repeated with different sparsities. Because of this, these four stages are enough to show the validity of channel pruning on YOLOv3. My experiment won't try to replicate the iterative pruning aspect because of resource constraints on Google Colab.

In terms of required resources, YOLOv3-SPP3 (the unpruned model) took 1.5 weeks to train on 15% of the VisDrone2018-DET dataset (970 images). The original paper used a batch size of 64 for training their model, however I had to scale the batch size back to 16 to prevent out of memory errors due to the limitations on Google Colab. This means that $\frac{120200*4}{970} = 496$ epochs were required to achieve the same training as the original implementation on the smaller dataset and new batch size. The sparsity training phase was faster, only requiring 100 epochs and 2 days, as specified in the original implementation. Pruning is a graph manipulation algorithm, not a machine algorithm, so it only takes a couple of seconds to run on the entire network. Another $\frac{60200*4}{970} = 248$ epochs of fine tuning were completed to bring the model up to the optimal performance. This last phase took 3 days to finish training. In order to circumvent GPU usage limits on Google Colab, the user that the model was being trained on was swapped out periodically, so that the training process could finish in a continuous span of 2 weeks.

Figure 4: A disproportionate detection from one of the images in the VisDrone2018-DET dataset.

These values are on par with the original implementation and are calculated to adjust the paper's parameters to the resource constraints. They should not effect the overall performance of the final model. The original implementation likely trained faster than this, purely because they used the entire GPU instead of sharing the GPU with other users. In addition, the original paper used a server with 56 CPUs and 4 GPUs to train the model, so their training time is not reflective of how much training time is required on Google Colab.

I used 0.001 sparsity, 10% per-layer threshold, and the 50% weight the pruning threshold as the hyperparameters during the pruning process. These were found in the original paper's source code and example execution commands. It is unclear if these were the hyperparameters used in the paper, but very different results from my process imply that the paper used different hyperparameters.

To make the experiment easier to replicate, I created a utility script in the training folder that is able to run experiments that are specified in a YAML configuration file. Replicability of the experiment will not only help researchers validate that the model functions as it is supposed to, but also help laypeople train their own models on their own datasets if they so desire.

Results and Analysis

I followed the stated training hyperparameters layed out in the SlimYOLOv3 paper and reiterated in the training procedure section and was able to reproduce their claims about the effectiveness of channel pruning.

The authors only publicly released their pretrained models for a short period of time before the link stopped working. Emailing the authors to post the pretrained weights again received no response, so the only way to compare their implementation on the same hardware (Colab), retraining from scratch would have to be done. Even then, training again would result in slightly different weights and the pruning process thresholds on the weights, so the resulting pruned model may have a different size and complexity based on these small variations, so there is no way to fully replicate their experiment. In addition, training on a significantly smaller subset of the training data made the accuracy of the initial model lower than the accuracy of their model.

Despite this and resource limitations, I was able to reach high mean average precision (mAP) on the dataset. Although this may be low for other classes of image classification algorithms, the YOLO family is a one stage image classifier and is optimized for speed over accuracy. The results of my experiment replication are summarized in Table 1. My final fine-tuned model reached comparable mAP during the training process to the fine-tuned model created by the paper implementation, despite using a much smaller training dataset (̃22.73% mAP despite using only 12% of the training images).

This is a paradox that I haven't been able to resolve in this paper and am currently looking into potential reasons that the mAP could be higher than expected. One potential solution was proposed in the YOLOv3 paper rebuttal which claims that mAP is a bad measure of the accuracy of a model due to its inability to reliably catch misidentifications (Redmon and Farhadi 2018). This also agrees with visual inspection of the bounding boxes of the slim model, showing very large bounding boxes compared to the unpruned model. Despite its problems, I have decided to use it to evaluate my model because it remains a commonly used method for evaluating machine learning models and at least is able to show progress during training.

The resulting model from my training process was 40% larger (more parameters) but 43% less complex (fewer FLOPS) then their model. At first, this may seem paradoxical, but certain channels cost more in a residual network than others and having a larger model, in some cases, can lead to less complexity (Dong et al. 2017).

In addition, the original paper noticed a disproportionate number of cars in the training dataset which caused much higher accuracy for cars compared to other classes. This issue is also appears in my implementation of the model. The example in 4 demonstrates this pattern well because the model is unable to detect the motorcyclists or pedestrians in the image, but is able to detect almost all of the cars in the image.

Conclusions

I was able to reimplement the SlimYOLOv3 model in TensorFlow (a platform in which this model does not yet have any other implmentation). Using this reimplementation, I was able to replicate the first couple of phases in their training process, able to create a 50% sparse SlimYOLOv3 model. The unpruned model has comparable performance to the paper implementation and the fine-tuned version was

Model Phase	Latency	Parameters	mAP	BFLOPS	Latency	Parameters	mAP	BFLOPS
Unpruned	1.688	63.9M	21.71	152	43.1ms	63.9M	23.3	151.72
Sparse	1.689	63.9M	7.11	152	-	-	-	-
Pruned	1.321	29.1M	0.00	69.7	-	-	-	-
Fine-tuned	1.32	29.1M	22.73	69.7	25.6ms	20.8M	22.6	122

Table 1: A comparison of the results obtained during my experiment (left) and the original paper's experiment (right) for pruning with 50% sparsity (Zhang, Zhong, and Li 2019). The paper never reported these metrics for the sparse and pruned models, only just the final working models: original YOLOv3-SPP3 and the SlimYOLOv3-SPP3-50. Latency is measured in seconds and the mean average precision (mAP) is measured in percent. The latencies of the paper implementation were calculated by the authors on different hardware, and thus can't be compared to my latencies (Zhang, Zhong, and Li 2019).

able to perform slightly better than the paper implementation. Clearing all of the weights in the network form scratch results in much poorer results than the pruning process. Additionally, I was able to demonstrate that the fine-tuning process is a critical step in pruning that allows the network to recover from the artificial damage that I did to the network.

A further unresolved problem not mentioned in the paper or my implementation is deciding the best pruning threshold hyperparameters for sparsity training to optimize size and/or complexity. The issue bleeds into the study of neural architecture search, which will have to answer the question of how to choose these hyperparameters by optimizing the model itself.

References

Cai, H.; Gan, C.; Wang, T.; Zhang, Z.; and Han, S. 2020. Once-for-All: Train One Network and Specialize it for Efficient Deployment.

Cun, Y. L.; Denker, J. S.; and Solla, S. A. 1990. Optimal Brain Damage. URL http://yann.lecun.com/exdb/publis/pdf/lecun-90b.pdf.

Dong, X.; Huang, J.; Yang, Y.; and Yan, S. 2017. More is Less: A More Complicated Network with Less Inference Complexity.

Guo, N.; Xiong, W.; WU, Q.; and Jing, N. 2016. An Efficient Tile-Pyramids Building Method for Fast Visualization of Massive Geospatial Raster Datasets. *Advances in Electrical and Computer Engineering* 16: 3–8. doi:10.4316/AECE.2016.04001.

He, K.; Zhang, X.; Ren, S.; and Sun, J. 2014. Spatial Pyramid Pooling in Deep Convolutional Networks for Visual Recognition. *Lecture Notes in Computer Science* 346–361. ISSN 1611-3349. doi:10.1007/978-3-319-10578-9_23. URL http://dx.doi.org/10.1007/978-3-319-10578-9_23.

He, K.; Zhang, X.; Ren, S.; and Sun, J. 2015. Deep Residual Learning for Image Recognition.

He, Y.; Zhang, X.; and Sun, J. 2017. Channel Pruning for Accelerating Very Deep Neural Networks. *CoRR* abs/1707.06168. URL http://arxiv.org/abs/1707.06168.

Huang, G.; Liu, Z.; van der Maaten, L.; and Weinberger, K. Q. 2018. Densely Connected Convolutional Networks.

Kyriakides, G.; and Margaritis, K. 2020. An Introduction to Neural Architecture Search for Convolutional Networks.

Li, H.; Xiong, P.; Fan, H.; and Sun, J. 2019. DFANet: Deep Feature Aggregation for Real-Time Semantic Segmentation.

Liu, Z.; Li, J.; Shen, Z.; Huang, G.; Yan, S.; and Zhang, C. 2017. Learning Efficient Convolutional Networks through Network Slimming.

Paganini, M.; and Forde, J. 2020. On Iterative Neural Network Pruning, Reinitialization, and the Similarity of Masks.

Pleiss, G.; Chen, D.; Huang, G.; Li, T.; van der Maaten, L.; and Weinberger, K. Q. 2017. Memory-Efficient Implementation of DenseNets.

Redmon, J.; and Farhadi, A. 2016. YOLO9000: Better, Faster, Stronger.

Redmon, J.; and Farhadi, A. 2018. YOLOv3: An Incremental Improvement.

Stuhlsatz, A.; Lippel, J.; and Zielke, T. 2010. Discriminative feature extraction with Deep Neural Networks. In *The 2010 International Joint Conference on Neural Networks (IJCNN)*, 1–8. doi:10.1109/IJCNN.2010.5596329.

Wang, W.; and Zhu, L. 2019. Channel Pruning for Efficient Convolution Neural Networks. *Journal of Physics: Conference Series* 1302: 022073. doi:10.1088/1742-6596/1302/2/022073. URL https://doi.org/10.1088%2F1742-6596%2F1302%2F2%2F022073.

Yang, C.; Yang, Z.; Khattak, A. M.; Yang, L.; Zhang, W.; Gao, W.; and Wang, M. 2019. Structured Pruning of Convolutional Neural Networks via L1 Regularization. *IEEE Access* 7: 106385–106394. doi:10.1109/ACCESS.2019.2933032.

Zhang, P.; Zhong, Y.; and Li, X. 2019. SlimYOLOv3: Narrower, Faster and Better for Real-Time UAV Applications. *CoRR* abs/1907.11093. URL http://arxiv.org/abs/1907.11093.