Object Classification and Localization on ImageNet

Matthew Howard
Computer Science
UC Santa Cruz
Santa Cruz, CA, USA
matthoward@ucsc.edu

Alex Williamson
Computer Science
UC Santa Cruz
Santa Cruz, CA, USA
alswilli@ucsc.edu

Arindam Sarma
Computer Science
UC Santa Cruz
Santa Cruz, CA, USA
asarma@ucsc.edu

ABSTRACT

In recent years, accurate image classification has proven vital for many modern vision applications such as facial recognition and autonomous driving. More importantly, the ability to classify and localize a specific object within an image frame remains crucial for effective performance of these applications. In this paper, we propose a prototype deep learning architecture based on Convolutional Neural Networks (CNNs) that combines characteristics of several state-of-the-art image classification models. We evaluate our prototype model on ImageNet for both classification and localization tasks, and we compare our performance against several baselines. For classification, our model achieves 66% top-1 accuracy, 3% higher than the best tested baseline (ResNet). For localization, our model achieves a marginal 1% improvement on prediction loss than the default Single Shot Detector (SSD) localization framework.

1 Motivation and Objective

1.1 Project Goals

The main goal of our project is to classify and localize objects from a subsample of the object classes in the 1000 class ImageNet dataset. Ideally, we would train on all 1000 classes, but due to time constraints and limited processing power we limit our sample size. An additional goal for our project was analyzing which types of data augmentations provide better benefits to accuracy, but the results from this were inconclusive and unreportable. The final output should be high top 5 accuracy for both image classification and localization between our model and several baselines.

1.2 Related Work

A large amount of related work has been conducted on our task, so we will limit this section to a few of the more recent advances on the problem.

A general trend since the first few iterations of Convolutional Neural Network (CNN) architectures (AlexNet, VGG, etc.) for object detection has been to increase the depth of architectures to promote better learning of important features. However, researchers found that as they increased the amount of layers, the performance would actually begin to degrade after some number of epochs. This problem is known as the vanishing gradient problem, and it is caused by early layers in substantially large networks not being able to learn relevant weights since their weight updates are too miniscule. The derivatives of early weights with respect to the total loss are a product of the other derivatives for weights in deeper layers, thus as the network grows larger the smaller these derivatives get and the less impact they have on early layer weight updates. Since early layers in networks are the more important layers, with much of their representation of simpler features like edges and shapes being the catalyst for learning the larger features like eyes and other complex structures, the larger network outputs generally worse performance overall. One important architecture that fixes this problem in larger architectures which we took inspiration from when building our model is ResNet. In order to solve the vanishing gradient problem, ResNet uses deep residual networks, which are networks that feed the results of earlier layers in the feedforward stage to deeper layers. Therefore, during backward propagation, more of the gradient can be preserved layer to layer, allowing the early layers to learn more valuable features (thus minimizing the vanishing gradient problem).

Another model is Single Shot Detection (SSD), which skips the region proposal step found in models like R-CNN and Faster R-CNN, providing greater efficiency at the cost of somewhat lower accuracy. The process is as follows. As in other models (the original SSD paper uses VGG16), using convolutional steps, a number of features are extracted giving rise to a feature layer that is *nxm* in size, with *c* channels (for example, 38x38 in Conv4_3 layer in the original). This is the number of locations, or cells, that are examined in the next step. At each location, k (originally 4) bounding boxes with predefined aspect ratios are used to produce class scores and offsets to a default bounding box shape, with the highest class score being passed on as the prediction for that box. This ignores the depth of the feature maps, and in the original design, produces 38x38x4 predictions. This technique is called multibox detection, and uses small (3x3) convolutional filters that return 4+b channels; 4 channels for the bounding box (two to locate it, two to scale it), and b for each of the classes (including one for a prediction of no class).

SSD learns the best set of bounding boxes to use by using a collection of predefined starting boxes (similar in concept to the anchors in Faster R-CNN). These were hand selected after examining the most common aspect ratios of boxes that real life objects tend to be bound by. Keeping the number of boxes low helps avoid extra, and potentially unnecessary, complexity. Each feature layer has its own set of default boxes, which helps detect objects at different resolutions. The cost of a mismatched box is calculated with the ratio of the intersection of the default box with the ground truth over the union of the two, or IoU. Positive matches have a ratio of 0.5 or higher and are used to calculate the localization loss, while everything less is a negative match, and is ignored.

One issue with the SSD model is that as its first detection layer, Conv4_3, has a 38x38 spatial dimension, and smaller objects require high dimensionality layers, it has relatively poor performance on small object detection (relative to the size of the input image). In general, SSD has slightly worse classification but better localization than the models described above. Its main benefit is that due to its architecture, it remains competitively accurate even when the input image resolution is lowered to 300x300, leading to speedups of anywhere from 5-10 times over Faster-R-CNN [2].

1.3 Our Approach

While both models above have provided state-of-the-art approaches to solving the problem of object detection, they both have faults that can be further improved to learn features with even better accuracy. While ResNet has allowed for better testing accuracy to be achieved in larger networks, it is limited in that it has the tendency to overfit the training data and create long droughts or plateaus without improvement. Even though SSD did fairly well in localization, and ran quickly, its base classification architecture before branching of into separate paths for classification and localization is relatively skinny and simplistic, with lower performance for certain classes of objects. Therefore, with our architecture we improve on their limitations through a unique combination of the two networks that aims to improve on their limitations. Specifically, we propose additional dense layers and Dropout Regularization to ResNet and substitute this model in as the base classifier for SSD. We believe this fusion between the two models is a novel approach to object classification and localization, with preliminary results showing improved test data accuracy. We discuss our final model in more details later on in Section 3.

2 Dataset

2.1 ImageNet

ImageNet is a collection of quality-controlled, human-annotated images that aims to illustrate each concept ("synset", or collection of synonyms and related words) within WordNet by providing, on average, over 1000 images for each such synset. For the task of object localization, ImageNet maintains over 3000 synsets with annotated bounding boxes, with each synset containing 150 images on average. For this project, we use a collection of 12,960 images covering 25 classes complete with bounding box annotations, randomly subsampled from a 1000 synset dataset released for the Kaggle-hosted ImageNet Object Localization Challenge[1].

For each image in the dataset, a list of bounding box annotations specified in the format (label, xmin, ymin, xmax, ymax) are provided that indicate a rectangular selection (bounding box) within the image that contain the object corresponding to the specified label.

2.2 Preprocessing

Several preprocessing steps were taken to ensure consistency of our data. First, we resize all images to the target size of 224x224 to eliminate potential issues of scaling. In addition, we also normalize all image vectors from 0-255 to 0-1 and one-hot encode all associated class labels. In addition to these simple preprocessing steps, we also developed an image augmentation pipeline that allows for several image transformations to be applied at training

[1] Kaggle: ImageNet Object Localization Challenge - (https://www.kaggle.com/c/imagenet-object-localization-challenge)

time. Image transformations include: rotation, horizontal and vertical flipping, scaling, and color shifting. However, as noted in Section 1.1, results of our augmentations were unremarkable so we excluded any analysis of these tests from the paper (we included some in the progress report). We hypothesize that perhaps the augmentations we used were subpar or the data we performed the augmentations on wasn't particularly susceptible to augmentations. Additionally, it could be that benefits from augmenting could only become obvious after an extended range of epochs.

3 Models and Algorithms

3.1 Model Exploration

When we first began designing our model, we were most focused on just getting some sort of object classifier working as we had never experimented with deep learning based object classification before. This led us down a path of research for creating a model that built upon the simpler famous object classification models that were easier to understand and implement. We settled on a model that resembled VGG16 [3] but with far fewer convolution and dense layers.

The main structure of this prototype consisted of several convolutional layers packaged with a rectified linear unit (ReLU) activation function and a max pooling layer that feeds into the following convolution, with each convolutional layer varying in size, generally increasing in size by a factor of two. Each convolutional layer has a filter of size 3x3, and each pool has size 2x2 with stride length 2. We chose to keep the number of packaged convolutional layers to four for our classification task size (we started with 5 classes), considering that a large network would be overkill for the simple prototyping task on five image classes, and thus potentially negatively impact performance. With each subsequent pooling, we increased the size of the convolutional layer to (hopefully) extract more detailed features over time. By downsampling the convolutional layers with small filters and into smaller (max) pools, we reduced the dimensionality of the image data and forced the network to generalize about features of subregions of the image, which is potentially useful for detecting characteristic curves, shapes, and other elements of the classes we wish to classify. Following the convolutional layers, the output was flattened and fed into multiple connected dense layers to extract the most important features. Finally, the output was fed into a softmax activation function which provides probabilities over classes that can be used for the classification task. In addition, we applied dropout regularization to reduce the effects of overfitting, and during training we optimized categorical cross entropy as our loss function via the Adam optimizer.

While this model allowed our project to come alive, it exhibited poor validation accuracy on our test set once the number of classes were scaled up from more than 10. Additionally, we were loading our training data as one batch into our limited amount of system RAM (16GB), meaning we could not support more than 10 classes without crashing (images are large). Therefore, we decided to split our workload down two paths.

3.1.1 Path 1 - Expansion to Joint Classification and Localization. The first path continued to expand our simple VGG16-based model to account for object localization by adding a regression head just before the fully connected layers for bounding box coordinate predictions. The idea here was to allow the convolutional layers to learn the relevant features of the image as usual, but split the tasks of classification and localization into two separate fully connected components. The classification loss would be computed using cross entropy like before, while the localization loss utilized the L2 loss function.

However, due to the simplicity of the model and its loss function, which summed up the loss from the classification and regression heads, the backpropagation of weights did not work very well and the losses mainly interfered with learning proper weights. Each of the box predictions seemed to be generalized towards the center of the image rather that the ground truth boxes, while classification accuracy took a nosedive. In an attempt to avoid these issues, we scrapped the model for an implementation of SSD, which allowed for fast object detection without sacrificing too much accuracy. We followed a simple implementation of this architecture[2] and were able to adapt our data parsing function so that the model would be able to read in our data properly. Once coding for the model was finished, we realized the best way to modify and improve on the base implementation would be to maximize the performance of the base classifier, which largely determines the accuracy of both classification and localization. Our final model after modifying the base classifier is described later in Section 3.3.

3.1.2 Path 2 - Hyper Parameter Tuning and Mini-Batching. The second path experimented with mini-batching and the tuning of hyper parameters to see if we could improve validation accuracy for our VGG16-based

[2] SSD Keras: (https://github.com/pierluigiferrari/ssd_keras)

model when classes were scaled. Mini-batching was needed to allow us to train with more than 10 models, since it loads 1 batch (a fraction of the data) into RAM at a time when training. A data generator was used to choose an equal distribution of classes per batch that was randomly reshuffled after each epoch. While this allowed for faster convergence in our models, it slowed down the training process significantly as we became I/O bound. Initially, each batch we picked a subset of images from the training set based on the shuffle to improve generalization of weights, but these images took a long time to extract so our GPU would often be idle. To improve the efficiency, we shuffled every epoch and read from the training set sequentially to avoid searching through the array. Since this only offered moderate performance gains, we also tried to read sequentially without shuffling, but each batch would randomly augment some of the images with rotations, contrast shifts, and more. We hoped that this would allow each batch for each epoch to be different enough to generalize the learned features, but we seemed to get nonsensically worse or better results so we stuck with our second implementation.

In regards to parameter tuning, many of the parameters we experimented with initially were number of filters in convolutional layers, strides and size of the filters, number of max pooling layers, use of average pooling versus max pooling, choice of optimization and activation functions, regularization via the use of dropout, and batch normalization. Tuning these parameters was a long and tedious process though, with only marginal gains, so we moved on to increasing the number of layers (convolutional, pooling, dense) being used in the model. However, with VGG16 being very memory intensive, this path saw issues with the number of model parameters being too high for the VRAM of our gpu to handle. Therefore, before we were able to improve the accuracy of classification in a significant way, we had to focus on building a model that exhibited good accuracy without generating a lot of parameters. After some testing, we decided to implement our own version of ResNet, which allowed us to both add extra layers without getting any negative side effects of the vanishing gradient problem while also limiting the amount of parameters. We went a step further and optimized our version of ResNet to reduce overfitting as well, which ended up being our final model for classification. The details of this architecture are discussed in the following section 3.2.

3.2 **Classification Network Architecture**

We designed our prototype classification model to consist of a sequence of three stages: (1) a pooled convolutional layer, (2) residual learning blocks, and (3) a fully-connected classification network. In the following sections, we outline the details of each stage and provide justification for each design choice. Figure 1 provides an overview of the stages of our model.

3.2.1 Stage 1 - Pooled Convolutional Layer. The first stage consists of feeding input images directly into a pooled convolutional layer. Convolutional layers are the basic building blocks used in nearly all state-of-the-art image classification networks because of their ability to learn filters that activate when detecting spatial features of the input (e.g., edges, shapes) without manually encoding the features. Additionally, pooling down-samples the convolutional representation to reduce its size to avoid overfitting and improve robustness. In specific terms, we design our first convolutional layer to contain 64 nodes with a 7x7 filter and stride length 1. We apply batch normalization, 10% dropout, and then apply a max pool with pool size 2x2.

3.2.2 Stage 2 - Residual Learning Blocks. Residual learning blocks were demonstrated by He et. al. [1] to be easier to optimize and lower in complexity, allowing for greater depth than comparable networks such as VGG. Residual blocks pass layer inputs around blocks of consecutive convolutional layers and do not add additional complexity, allowing SGD or similar optimization algorithms to be applied as they would to the same network without such shortcuts.

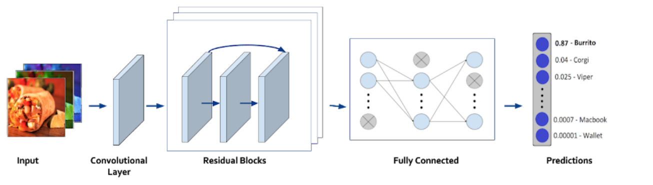


Figure 1: **Overview of proposed network architecture.**

In our model, we utilize the following blocks: a first shortcut layer around a first set of convolutional layers sized 64, 64, and 128; a second shortcut layer around a second set of convolutional layers sized 64, 64, and 256; two consecutive convolutional blocks, each with layer sizes 64, 64, 256; a third shortcut layer around a set of convolutional layers sized 128, 128, and 512; and finally, three consecutive convolutional blocks, each with layer sizes 128, 128, and 512. Our design of these residual blocks is modeled after the ResNet50 model, and includes batch normalization, filter size 1x1 and stride length 2x2 on each convolutional layer. All layers are activated via the ReLU function.

3.2.3 Stage 3 - Fully-connected Classification Network. The final stage of our model is a fully-connected network that takes the convolutional feature maps from the residual blocks as inputs and outputs predicted class probabilities. We keep the size of our fully-connected network small, utilizing three consecutive layers of size n, where n is the total number of classes. Dropout of 25% is applied to each of the first two layers, and the final layer is activated with the softmax function to enable class probability prediction. In designing our fully-connected network, we aimed to limit the impact of overfitting. By keeping the network small and adding a significant level of dropout, we are able to prevent the network from becoming too complex, while still allowing for improved learning through the non-linear combination of learned convolutional features.

3.3 Modified-SSD Object Localizer

In order to apply our proposed network to the task of object localization, we developed a modified SSD object localizer that utilizes our proposed network as the base classifier that feeds into both the classification and localization branches of the SSD model. Specifically, we apply stage 1 and stage 2 of our architecture upon the input and feed the resulting convolutional feature maps into the remainder of the SSD model, which branches into a classification network and localization network, both containing several more layers of convolutions.

4 Results and Analysis

4.1 Experimental Setup

For our final set of experiments, we decided to conduct two different rounds of testing. For the first round, we looked at the training and validation set accuracies for our prototype classification model and compared the results with the baseline ResNet50 and SSD7-classifier models. For the second round, we looked at the combined classification and localization loss computed by our modified-SSD model and compared it with the loss seen with baseline SSD7.

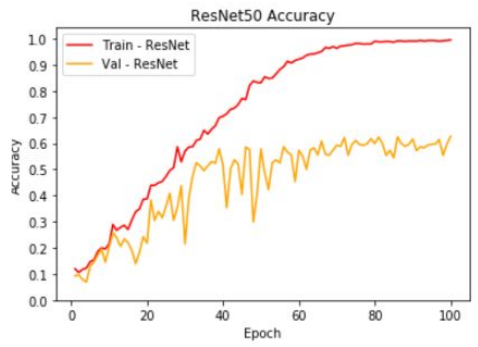
With both rounds, we split our training data up so that 80% of the total data would be the training set while the remaining 20% would be the validation set. This was a necessary step because the test data that was provided from Kaggle covered the entire 1000 classes. In the first round, we trained for 100 epochs, while in the second round we trained for 25. Both rounds trained using an Nvidia 980TI GPU with 6GB of VRAM, coupled with 16GB of system RAM and an Intel i7 6700k CPU.

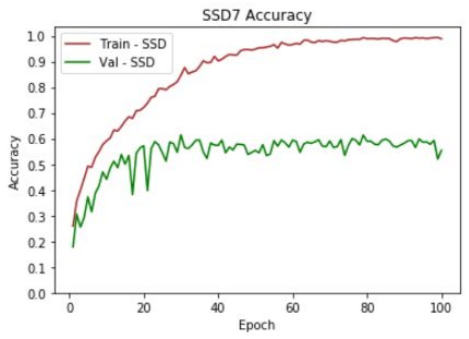
4.2 Image Classification Accuracy

In Figure 2, we display the results for the first round of testing, as described in Section 4.1. A general trend for all tested models is the tendency for overfitting. However, with our model, we notice that this trend is reduced slightly, with the training accuracy curve growing more slowly over time and the validation curve taking longer to level out on a specific value. In fact, the general trend for our validation accuracy curve seems to show a positive slope, meaning that if we trained over a larger number of epochs is it is likely that our accuracy would level out on a higher value than the other graphs. This hypothesis is strengthened by the fact that we see a large amount of variance in our model from epoch to epoch; with other models this variance leveled out as the curve's slope went to zero so this could further suggest a larger growth in accuracy for our model.

One odd feature we noticed in the graph of our final model is that there is a drop in the validation accuracy between epochs 20 - 40. We are unsure why we happened to see this sort of output, as it was not replicated on subsequent runs of the training on the same data, but it could be that for that specific run we reached a local minima on the loss function.

The final top-1 validation accuracies we plotted are as follows: 63% (ResNet50), 61% (SSD), **66% (Final Model)**. Although the change in value here is relatively small, the continuing upward trend in our plot suggests that this small change will continue to grow with more epochs. The final top-5 validation accuracies are: **93% (ResNet50)**, 90% (SSD7), 91% (Final Model). Resnet ended up being better in this respect, but our top-5 accuracy is good nonetheless.





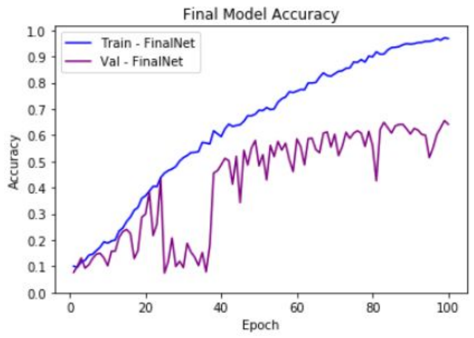


Figure 2: **Comparing the train and validation accuracy for our classification model against the ResNet50, SSD-7 classifiers.**

4.3 Object Localization Performance

For object localization, we trained the default SSD7 model and compared it with a modified version of SSD that utilizes our network architecture as the base classifier. Figure 3 displays our results, where we are able to achieve a marginal improvement (~1%) in loss minimization compared to the standard SSD model. We believe that this small performance gain could be caused because our classifier generates better confidence values for classification but provides worse performance as a bounding box predictor. As a result, the total loss for SSD, which is a summation between the cross entropy classification loss and L2 loss for localization, could receive net gain; the classification loss could be less while the localization loss could be greater.

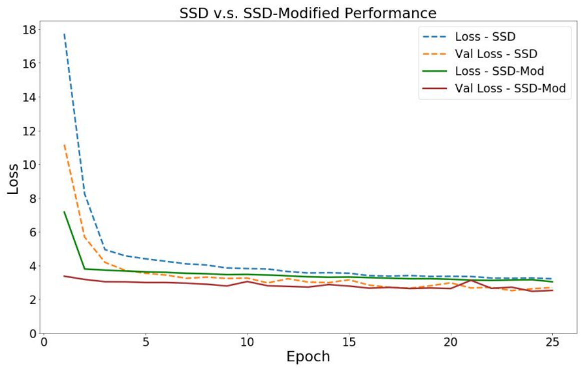


Figure 3: **SSD loss (classification + localization) for default SSD and our modified SSD.**

5 Contribution

5.1 Matt's Contribution

Matt assisted in several areas of the project, including major contributions to the preprocessing and experiment pipeline, as well as model development and optimization. Specifically, Matt contributed to the development of the pipeline that enabled data loading and conversion, data storage, as well as model fitting and interfaces between the model and the file system. In addition, Matt wrote several utilities for analyzing our experimental results, including metric calculation and dataframe manipulation. Further, Matt also assisted heavily in the problem solving process, including idea generation, debugging, and parameter tuning. In summary, Matt was considerably involved in all phases of the project.

5.2 Alex's Contribution

Alex contributed in a variety of ways to the project. A large part of his efforts went into researching which models to experiment with throughout the process of the project and writing the code for their implementations. Additionally, many of the tests involving hyper parameter tuning were tested using ideas he came up with related to Deep Learning theory. Alex also worked with Matt on many of the features used throughout the pipeline of the project, including the function for parsing the data from the image files and the main pipeline functionality for loading in training data, fitting the models, and printing the results of the models via graphs. When problems came up surrounding with implementation, Alex was very involved with the process of finding ways to address them and help out with coding portions of the fixes. Additionally, Alex helped design the poster and put a lot of effort into forming the project report(s).

5.3 Arindam's Contribution

Arindam contributed to the project in the following ways. Proposals on how to tackle the data augmentation portion of the project, as well as testing the resulting work were his responsibility (unfortunately, the problems faced in other portions of the project combined with the lack of time to fully explore this area led to inconclusive, unreportable results). He identified the problems with gpu performance, determining that their source lay not in the raw computational powers of the hardware, but in the I/O capabilities of the architecture the hardware was in. Arindam also helped resolve and mitigate these issues as much as possible considering the limited resources the team had. Additionally, he conducted research into state of the art models to examine which were fit for the team's purposes, and helped write and test scale versions of these models to determine if there were parts that could be salvaged for future use. Finally, as with the rest of the team, he helped create the poster and final report.

6 Future Work

6.1 Extending Scope

One limitation of our project is that we only run our training sessions on 25 classes and for 100 epochs. We limited ourselves to these choices because of time and memory constraints. Our original goal was training for 100 classes at minimum, but training took far too long on 1 local GPU and the number of epochs had to be scaled much higher if we wanted to see good accuracy even in the top 5 values for any model we tested on. Additionally, tuning our proposed models would have been a lot more tedious because of how long it would take to see any clarifying results from running different versions of our model. Even with 25 classes, this was an issue and meant that we didn't get enough time to tune our hyperparameters in the final model as much as we would have liked. One way we could fix this issue in the future is by running on a cluster of gpus that are built to handle large tasks like the ones presented in our project, perhaps the Hummingbird Cluster at UC Santa Cruz. Given that our memory issues were only known about towards the end of the quarter, as we were running most tests with 5 classes for a large majority of the time, we decided not to switch to using the cluster because of the time it could have taken to set up with our environment. We also thought about using Google Cloud with multiple GPUs, but the issue here came down to costs of using them. For future work, we will most definitely be planning our project around using these types of resources from the start, which could hopefully allow for more time to gain access to the resources we need to run our tests in an ideal scenario.

6.2 Hyper Parameter and Preprocess Tuning

Another limitation of our project, as briefly mentioned in the previous paragraph, is that we did not allow ourselves enough time to finely tune our model. With any type of image processing at a large scale, deep learning architectures take a long time to train and we did not prepare our testing environment well enough to account for this. To prevent this issues, we should have trained on multiple gpus simultaneously, perhaps via multiple google Colab instances. Again, with most of our issues related to training time popping up at the end of the quarter, we decided to stick with running our local GPU during testing rather than spend time switching to Colab (data transfer and I/O issues were causing massive time constraints). If we had properly accounted for these types of issues towards the beginning of the quarter, one way we would have extended our work would be to tune how the base SSD model was running its classification and localization regression heads (after splitting off from the base classifiers). Currently, we only tuned the base classifier to resemble our main classification model described in section 3.2, which does provide some performance gain but with more time we are confident edits to the additional convolutional layers used would have produced even better loss results.

7 Conclusion

To conclude, we achieved a model(s) for classification and localization that performed better than leading object detection models being used today. However, there is a lot of future work to be done in order to extend our improvements further.

REFERENCES

[1] Kaiming He, Xiangyu Zhang, Shaoqing Ren: "Deep Residual Learning for Image Recognition", 2015; arXiv:1512.03385.

[2] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu: "SSD: Single Shot MultiBox Detector", 2015; arXiv:1512.02325. DOI: 10.1007/978-3-319-46448-0_2.

[3] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. In ICLR, 2015.