Set Up:

With this algorithm, we took a different approach than PointNet. Due to the success of Computer Vision models on image data, we wanted to apply this success to the given problem of 3d object detection. We took different views of each object and took a picture of the target. We used these images with different views to train a CNN classifier. We did transfer learning with many state-of-the-art Convolutional Neural Network (CNN) models as feature extractors to convert images to image embeddings. We would feed the different views into the feature extractor to get embeddings for each image view of the object. We stacked together the different views into one layer and fed that through a linear layer to get the classifications. To our knowledge, this approach has not been taken to 3d object classification. Due to the success of image classifications, we thought we could transfer this success to 3d object classification.

Our work was in some was similar to RotationNet[1] and MVCNN[2]. We found their papers after we had already started our experimentation. In RotationNet, they would make a prediction on the class of an object for each view and combined them into a single prediction. The advantage of their method is that it supported variable number of views as input. They were able to perform the task very well, holding the current state of the art for ModelNet10. Their model has the drawback that multiple views are combined using an equation similar to an ensemble. The goal of our network was to be a self-contained model.

In MVCNN they would average pool the CNN layers together of the different views of the object. This architecture most closely matched ours since the input space requires the same number of views for classification. We combined the different views through a linear layer, but still required the same number of views for each input. We thought our method of concatenating the view embeddings would perform better. They would use average pooling to combine their views. We thought this would result in some loss of information that would cause a reduction in performance.

The views we selected tried to get the most information we could about the objects. We chose to color all our objects black. We chose this color since it is the highest contrast with the white background. We wanted to use all the channels of the feature extractors to give them the best chance of extracting meaningful features from the images. We also set opacity to .5 so the image could encode more information by seeing through certain parts of the image.

Following the goal of encoding as much information as possible, we chose our views with this goal in mind. We started with 6 views. The chosen views were top, bottom, and each side of the object. We later moved on to 14 views, adding the 8 corners of a cube around the object. Our intuition was that the more views, the more information for the model to learn and reach better performance.

We also set up an encoder decoder model to try and learn image features from scratch in an unsupervised way. For most of the models we loaded the pretrained ImageNet weights. For the encoder decoder we initialized the model randomly. We wanted to see if the encoder model could learn to extract the features. We saved the lowest loss model taking the MSE between the original image and the decoder output. We then used the encoder as our feature extractor concatenating the different views and feeding them into a final linear layer.

In regard to hyperparameters, for most of the models we used SGD with lr 0.001 and momentum .9. We also tested Adam with default parameters. We used cross entropy loss. We tried different weighting of the classes due to there being a class imbalance. We did not tune the batch size as a parameter, but would change it based upon the model size for it to fit in memory. Due to having multiple images as input, we had smaller batch sizes than are typically recommended, ranging from 2 to 8 for most of our models. This could be overcome by having the optimizer update the weights on every few epochs instead of every epoch, but this was not something we implemented. We also tested different amounts of linear layers and dropout after the feature extraction.

For metrics we would keep the best validation loss and validation accuracy model during training. We typically trained for 100 epochs, but it would converge in 15-70 depending on the complexity of the model. Convergence in this case was when the validation loss or accuracy would stop decreasing. Even though the training loss and accuracy could still improve, if the validation metrics did not improve, we would consider this overfitting. We chose to keep the models with the best validation loss and highest validation accuracy since these were most likely to generalize to the test set.

We focused on ModelNet10 due to limit time and computing resources. The models we tested were large and would take even longer to train with more data that is in the ModelNet40 data set.

Our experiments were to set up the model, make any changes for the new experiment, and then feed in the data. We had the most issues related to data wrangling. We wanted to get all the data in memory, but some inefficiencies in how we set up the data caused many out of memory exceptions to occur. We found it easiest to initialize the large array first then fill it as the program runs. Caching the data up front into a single array led to much faster training times. Our 23 views would not fit into memory. We learned that images are compressed when written to the disk, so we stored this data as a list of byte arrays. This reduced the data size from 23gb as numpy array to 604mb. This allowed us to keep the data in memory and speed up training times but did have the small cost of decompressing the image. We also learned that saving a large numpy array to disk, 8gb in our case, is faster to read by compressing it using the savaez\_compressed method. It reduced the disk size to about 1gb and we were able to read it to memory twice as fast. We used google colab and google drive for storage, so your numbers may vary if reading from a local disk.

After getting the data in the right format, getting the shapes to line up, our model expected the shape [batch, views, channels, height, width] and output the 10 class probabilities. The last error we found after running some experiments and was that we were regularizing the images twice. After these issues the experiments ran smoothly.

Results:

Our first results were with ResNet50 were promising. We achieved the highest validation accuracy of all our experiments with 98%. Since we started out with 6 views, the models would train fairly quickly. We let the training job run for 300 epochs. This could partly explain why it is the lowest validation accuracy. It was able to search for the parameters longer than models with fewer views since it trained for more epochs. Its’ accuracy of 91.3 was 34th out of the 43 existing models on the dataset. The mean average precision score(mAP) of 95.93 is a new state of the art for the dataset, beating the other 13 models who had this metric recorded. With this promising start, we expected our model to continue to improve and achieve much better results with time.

Adding mish between the linear layers improved accuracy but lowered the mAP score by just a little. All of the data augmentation tried was not able to best the score with no augmentation. After we ran a few more experiments with ResNet50 we decided to add more views to see how that would improve the metrics. We assumed a large increase in the metrics by giving the model more information.

We ran the same first 2 experiments with ResNet50 comparing with and without mish with the 8 additional views. To our surprise, the model performed better without mish this time. It was within 1% accuracy and close to 1 mAP, so this result would need to be verified over multiple runs. This led to our best result at the time with 92.84% accuracy and 96.27 mAP, both being the best runs so far.

With the new views, we tested WideResNet 101 2, Inceptionv3, EfficientNet B6, and ShuffleNet v2 x2. These were all available in the Pytorch library as models. We used them as feature extractors and compared them to Resnet50. We ran all of them without mish due to the better performance on the earlier experiment. Inception v3 performed the best, setting a best for experiments in both accuracy with 93.61% and mAP of 97.18. The rest of our experiments were unable to outperform the metrics from Inception v3. This result is 24th out of 43 models and the top in mAP for ModelNet10.

We used Keras to do some experiments with EfficientNet. We had a different model for each view and averaged their scores to get the final prediction similar to RotationNet. In these experiments, we tried the Adam optimizer, and some weighting of the loss. We found that our models were performing the worst on the classes with the fewest examples, which is why our accuracy was not improving. Our few experiments with this did not improve our results. Our encoder performed poorly compared to other feature extractors. Trying the 23 views with Inception v3 led us to tie our previous best accuracy. Both models had the same number of test answers correct.

Analysis:

We were surprised by many of our results. The first item that surprised us was that our first attempt was able to get state of the art on mAP. Not all models recorded their mAP score, so we were likely helped by that fact. In our opinion, mAP is likely more important in this dataset due to class imbalance in the training and test set.

We also expected to be able to improve our accuracy by more than we did. In all our experimentation, it went up by 2.31% from our first experiment. It was a reduction in error of 27% so it is still good progress.

We were also surprised by the fact that mish did not help performance more in our experimentation. This is a sample size of one in a specific application, but we expected it to hold in all scenarios, that an activation function is better than not having one, but it did not in our case with 14 views.

We also expected the 23 views to make a bigger impact on performance. The fact that the 14 view Inception v3 performed the same on accuracy and a little better on mAP than the 23-view version came as a surprise. We thought giving the model more information would help it learn more. This is again a small sample size. We would like to run more experiments along these lines in the future, testing if a model is able to perform better with 23 views.

The color black may have been a poor choice for colors. The reason being that it made it difficult to see shadows of the object to get depth and some features of the object.

We also had some questions about the imbalanced classes. These were the ones that hurt our accuracy the most. The PyTorch and Keras built in functions for dealing with class imbalance did not improve performance on these classes. More work should be done to investigate if the smaller classes had too little data for the smaller classes to generalize well so more data of the smaller class is needed, or if the larger classes dominated the gradients too much, and so it can be solved with better loss weighting.

Conclusion and Future work

If we had more time and computing available, there are a few more ideas to try to improve performance.

The first one would be to use some type of Recurrent Neural Network (RNN) instead of concatenating the outputs of the CNN, we could use the RNN to combine the features from the different views. Additionally, the same could be done for LSTM and for Transformers. Take the CNN feature extractor output and feed them into each of these types of architecture. In addition, view positional embeddings could be added as an additional input to the RNN similar to position ending in NLP. We could also remove the CNN and have a transformer be the classifier as has been done in some recent work [3].

Another variable to test would be trying more augmentations. We did not get to experiment with this very much. Translations and random erasing are specific augmentations that has successfully been applied to image data in some previous experimentation and we expect to be successful here, and they might be applied successfully if implemented differently.

We would also want to try more dropout techniques. One would be view dropout, by either using all white pixels as the input or to zero out the dense output for one of the views. Vanilla dropout would also likely be useful between linear layers. We only had one experiment covering this, but would like to test it more, since our models typically overfit our training set, as shown in the tables that training accuracy would typically be at 99+%

Overall we succeeded in getting good results on the dataset with 24thout of 44 total models on accuracy and a new state of the art for mAP. Going into the experiments, we would be okay with below average to moderate results. The method of using CNNs as feature extractors on images proved to be successful.

Sources

[1] https://arxiv.org/abs/1603.06208

[2] <http://vis-www.cs.umass.edu/mvcnn/>

[3] https://arxiv.org/abs/2010.11929