Image Ranking - CS547 Project

Aditya Chanodia (adityac3), Nikhil Mertia (nmertia2),
Niranjan Kulkarni (nuk2), Rohan Limaye (rohansl2),
Sriram Sundararaman (srirams4), Viswarup Misra (vmisra3)

December 15, 2019

1 Introduction

Image similarity is the measurement of similarity between intensity patterns in two images. The ability to find similar images to a query image has multiple applications from visual search to duplicate product detection to domain specific image clustering. Many image similarity models use the category-level image similarity paradigm where two images belonging to the same category are considered similar. However, this cannot be used for search-by-example as the distinction of differences between within the same category, i.e., fine-grained image similarity, is important.

The performance of methods where image similarity models are built by first extracting features like Gabon filters, SIFT [1] and HOG [2] and then learn the models on top of these features is limited by the features. Usage of supervised similarity information for similarity models have proven to be of great potential for an even better fine-grained image similarity model than hand-crafted features.

Similarity ranking varies from image classification. It was observed that image classification models may not be a fit for task of distinguishing fine-grained image similarity. In this project, fine-grained image similarity is learnt using a deep ranking model which characterizes the fine-grained image similarity relationship with a set of triplets. A triplet contains a query image, a positive image and negative image. A positive image is the one close to query image than a negative image. Deep learning models employs the image similarity relationship characterized by relative similarity for better performance.

2 Paper Description and Model Architecture

Category-level image similarity has a major correlation with semantic similarity. Research by Deselaers et al. [3] studies the association between visual similarity and semantic similarity. The study has shown that visual and semantic similarities are mostly in line with one another other across different categories, but there exists a lot of visual variations within a given category. Thus, it is helpful to learn a fine-grained model that is able to characterize the fine-grained visual resemblance for images belonging to the same category.

To study the fine grained similarities the author [4] proposes a novel multi-scale network structure, which contains a convolution neural network with two low resolution paths. The paper shows that this multi-scale network structure can work effectively for similar image ranking.

The suggested Deep Ranking network consists of 3 parts:

- The triplet sampling layer
- The ConvNets
- The Image Similarity ranking layer

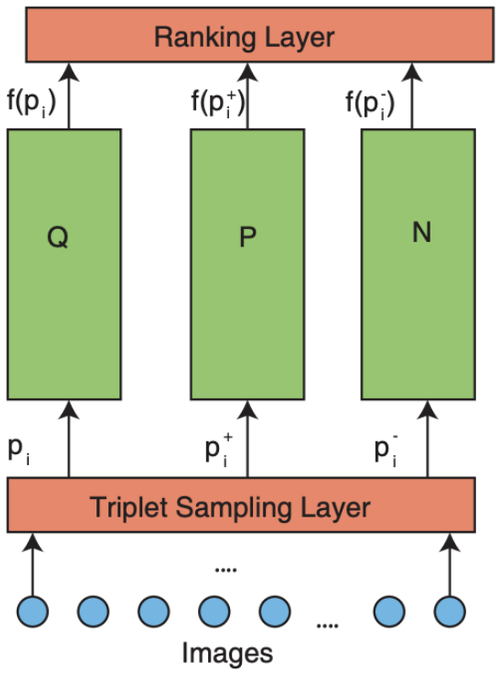


Figure 1: Model Architecture

For our study, we use ResNets inplace of ConvNets. ResNets are residual neural networks which have skip connections to provide a highway for the gradient to flow, thus preventing the vanishing gradient problem. Due to the skip connections, this speeds up learning as there are fewer layers to propagate through.

Another important aspect of the paper published by the author is that of triplet sampling. Below image is an illustrates of a triplet:

Figure 2: Example of triplets

Sampling a query image and a positive image (from the same class) are quite simple. Negative images can be sampled in two different ways either in-class or out-of-class. For this project, we will implement out-of-class samples only. Again, out-of-class samples are images sampled randomly from any class except the class of the query image.

In our case, the similarity between images is measured with the help of Squared Euclidean distance. There are other methods of measuring similarity between images like the Manhattan distance. There has been research regarding which distance measure is better for natural image cases. It has been found that Manhattan distance is better but here we use Squared Euclidean distance as the similarity metric. The smaller the distance the more similar the images are.

For our deep network architecture we use hinge loss to compute the loss for the network. For the triplet the hinge loss is defined as:

$$l(p_i, p_i^+, p_i^-) = \max\{0, g + D(f(p_i), f(p_i^+)) - D(f(p_i), f(p_i^-))\}$$

In the equation 'l' is the hinge loss for the triplet where we are trying to minimize the distance between the query image and the postive image and maximize the distance between the query image and the negative image. 'g' is a gap parameter or a regularization parameter that regularizes the distance between the image pairs: (p_i, p_i^+) and (p_i, p_i^-). 'D' is the squared euclidean distance between the two points. The ranking layer at the top evaluates the hinge loss for a given triplet. While training, it measures the model's violation of the ranking order. The gradients of the loss are back-propagated to the lower layers and the parameters of the lower layers are adjusting accordingly to minimize the hinge loss.

3 Dataset Description

For the study, we use the tiny ImageNet dataset. The dataset consists of 200 different classes. Each class has 500 training images, 50 validation images, and 50 test images. The images are 64x64 in size. We use the training set to train our model and the validation set to tune the hyperparameters. The tiny ImageNet dataset is provided in the Stanford CS231n course website. A sample of the images from the dataset is shown below.

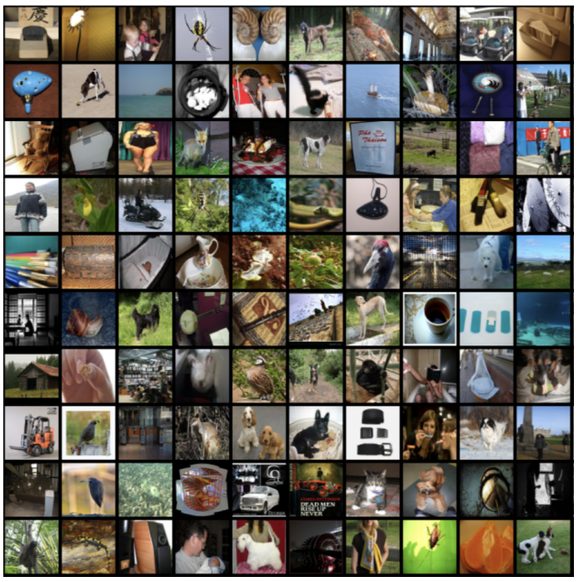


Figure 3: Sample images from tiny ImageNet

4 Code Description

The code is structured into three files: train.py, util.py and Model.py. The details on what each file does is provided below.

4.1 util.py

The util.py file has utility functions, such as a train and testloaders and most importantly the sampling of the positive and negative image given the query image, which is one of the major contributions of [4]. The positive image is from the same class as query class, hence we sample uniformly at random one of the 499 images (not including the query image). The negative image is in a different class from the query class, therefore we first sample uniformly at random one of the 199 classes (not including the query class), and then sample one of the 500 images from the sampled class. The code snippets in Figure sampling show how random sampling is done for positive and negative image given a query image.

```python
def find_pos_image(image_name, image_class, dir_path):
    img_num = int(image_name.split('_')[1].split('.')[0])
    rn = np.random.randint(499)
    if rn >= img_num:
        rn += 1
    return img_class + '_' + str(rn) + '.JPEG'
```

```python
def find_neg_image(image_name, image_class, dir_path):
    image_classes = os.listdir(dir_path)
    rn1 = np.random.randint(199)
    if rn1 >= image_class:
        rn1 += 1
    tgt_class = image_classes[rn1]
    rn2 = np.random.randint(0, 500)
    return os.path.join(tgt_class, 'images', tgt_class + '_' + r2)
```

Figure 4: Code in util.py to sample an image from the positive class (left snippet) and negative class (right snippet) uniformly at random

4.2 Models.py

The Models.py file, as the name implies, include the definition of the model architectures used in our implementation. We use a pre-trained ResNet model and performed experiments on three different ResNet architectures: resnet18, resnet34, and resnet50. To allow for easy training, we have a command line flag that specifies which model architecture should be used at run time, as seen in Figure 8. After loading the pretrained model, we pass its output to a fully connected layer of 4096 dimensions and then a dropout layer with probability 0.6 (as specified in [4]). The Tripletnet class takes in a model (for ex: resnet50), a query image, positive image, and negative image as input and passes each of the images into the model independently, as seen in the class' forward function.

```python
class Tripletnet(nn.Module):
    def __init__(self, embeddingnet):
        super(Tripletnet, self).__init__()

        self.embeddingnet = embeddingnet

    def forward(self, query, pos, neg):
        embedded_query = self.embeddingnet(query)
        embedded_pos = self.embeddingnet(pos)
        embedded_neg = self.embeddingnet(neg)
        return (embedded_query, embedded_pos, embedded_neg)

    def get_embedding(self, x):
        return self.embeddingnet(x)

def convnet_model_(model):
    convnet_model = resnet_model(model)
    convnet_model.fc = nn.Linear(convnet_model.fc.in_features, 4096)
    convnet_model.dropout = nn.Dropout2d(0.6)
    return convnet_model

def resnet_model(model_type='res18'):
    if model_type == 'res18':
        model = models.resnet18(pretrained=True)
    elif model_type == 'res34':
        model = models.resnet34(pretrained=True)
    elif model_type == 'res50':
        model = models.resnet50(pretrained=True)
    elif model_type == 'res101':
        model = models.resnet101(pretrained=True)
    else:
        model = models.resnet152(pretrained=True)
    return model
```

Figure 5: Code in Models.py to fetch pretrained ResNet architectures

4.3 train.py

The train.py is the "main" file our implementation. Firstly, the file parses the command line flags including learning rate, batch size, number of epochs, model architecture, and type of optimizer. Then, it creates the Tripletnet model (defined in Models.py), defines the loss function (TripletMarginLoss), and calls the train and test dataloaders (defined in util.py). Lastly and most importantly, it starts training for a set number of epochs. In each epoch, we create triplets (query, positive, and negative image), calculate the triplet margin loss and apply gradient descent methods (SGD, RMSProp, ADAM depending on command line flag) to perform the parameter updates. At the end of each epoch, we also calculate the top 1, top5, top 10 and top30 accuracy as well the confusion matrix, which will be displayed in the results section of the report.

5 Training Methods and Hyperparameters

Various models with varying hyperparameters were run on the training set. We used three different model architectures namely ResNet18, ResNet34 and ResNet50. For each architecture, we used different optimizers like Schocastic Gradient Descent and Adam and also learning rates varying from 10^{-4} to 10^{-1}.

The model with ResNet50 using SGD and a learning rate of 10^{-3} was trained for 35 epochs and gave the best results, hence we decided to use those hyperparameters. As the paper suggested, we used a dropout layer (for regularization) after the fully connected layer with probability of 0.6. Total train time for the final model was 21 GPU hours. We trained our model using Google Colab.

6 Results

6.1 Test Accuracy and Training Loss

Our model achieved a test accuracy of 61%. The table below shows the 5 classes for which our model achieved the best and the worst accuracies. It is interesting to note that generic classes such as pole and lakeside achieved very low accuracies while classes that have striking and unique attributes such as ladybug and triumphal arch have very high accuracies.

Best 5		**Worst 5**	
Class	*Accuracy*	*Class*	*Accuracy*
Ladybug	0.8553	Pole	0.2493
Triumphal Arch	0.8573	Umbrella	0.2546
Espresso	0.8579	Plunger	0.2826
Bullet Train	0.8613	Lakeside	0.2906
Rugby Ball	0.8706	Wooden Spoon	0.3153

6.2 Similarity Precision

The following table shows the similarity precision (percentage of triplets being correctly ranked) for both the train and the test sets. We computed the similarity precision for both ResNet50 and ResNet34. To calculate the similarity precision, we sampled uniformly at random a positive (in-class) and a negative (out-of-class) image for every query image.

	Train Set	Test Set
ResNet50 (After 35 Epochs)	**97.52%**	**95.13%**
Resnet34 (After 15 Epochs)	**84.92%**	**85.18%**

6.3 Confusion Matrices and Accuracy/Loss Plots

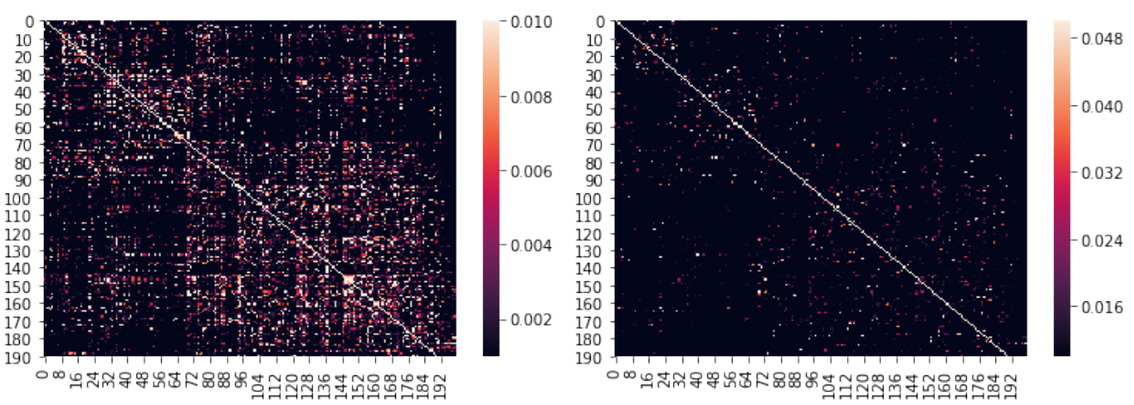


Figure 6: Heatmap plot of confusion matrices

The plot on left shows misclassification instances with probabilities between 0.1% to 1% and the plot on right shows misclassification instances with probabilities 1% to 5%. From the plot we can see that instances of misclassification are not concentrated in any particular class.

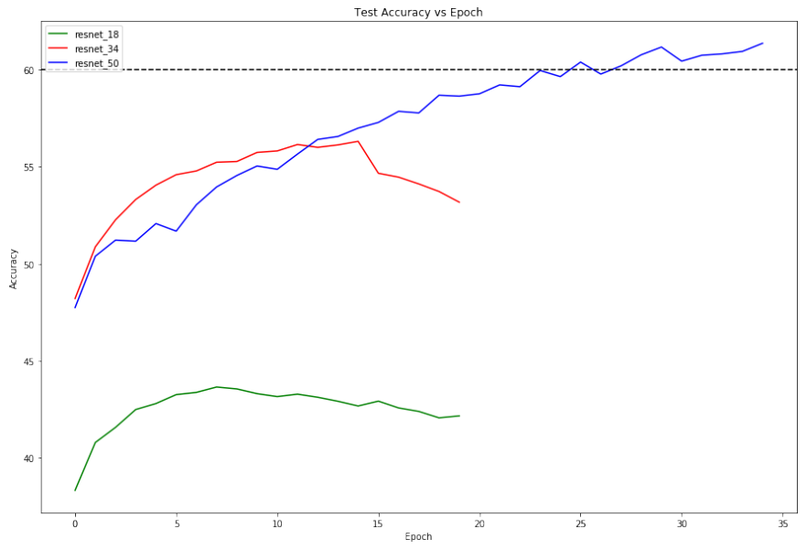


Figure 7: Accuracy Plot

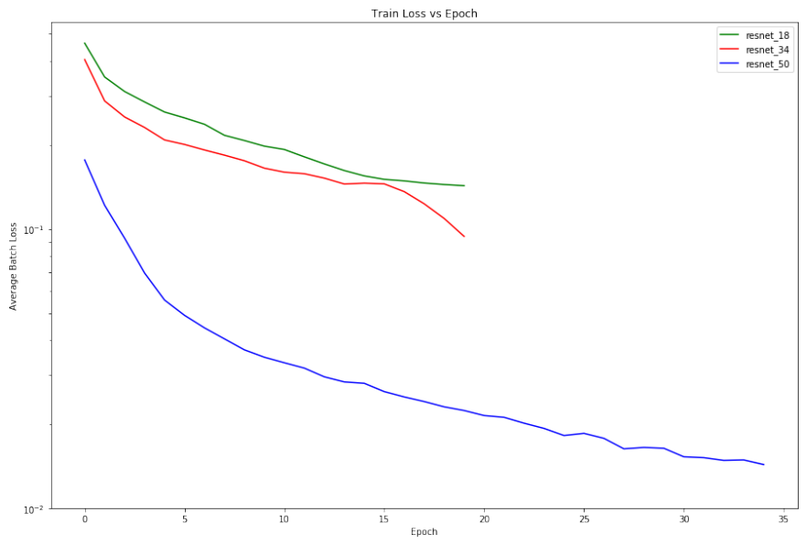


Figure 8: Average Batch Loss Plot

As we can see from the plots above, the accuracy for ResNet18 model saturated between 42% and 43% after a few epochs, whereas the ResNet34 model began to overfit after 14th epoch, leading to a decay in accuracy. Therefore, we stopped these two models after 20 epochs, but continued running the ResNet50 model as neither the accuracy saturated nor the model overfit.

6.4 Top and Bottom Ranked Results

Following tables show the top and bottom 10 ranked results for 5 different samples from the validation set. These results were obtained by the model using the ResNet50 architecture. The results for the ResNet34 model is given in the appendix.

Query	Top 10 Ranked	Euclidean Distance				
		9.8561	9.9934	10.0172	10.1965	10.2252
		10.2421	10.3051	10.3763	10.4290	10.5815
		9.02181	9.18032	9.38993	9.50974	9.62696
		9.63387	9.64303	9.72425	9.73617	9.78329
		7.71128	7.73687	7.92264	7.98977	8.00093
		8.03256	8.10433	8.10453	8.25943	8.33238
		10.0167	10.1825	10.2986	10.3118	10.3296
		10.4066	10.4611	10.4956	10.5187	10.5239
		8.88397	9.06652	9.12363	9.22323	9.26768
		9.26865	9.30893	9.40012	9.40907	9.43236

Figure 9: Top 10 ranked results for ResNet 50

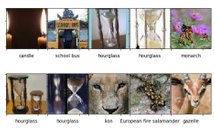
Query	Bottom 10 Ranked	Euclidean Distance				
		21.3060	21.0150	20.9023	20.8851	20.7732
		20.3617	20.3082	20.2847	20.0994	20.0160
		22.1114	21.7215	21.0026	20.4733	20.2800
		20.1995	20.1576	20.1207	20.0927	20.0788
		21.5370	21.0972	20.9260	20.6635	20.4909
		20.2977	20.2846	20.1362	20.1142	20.1078
		21.2769	19.8251	19.1613	19.0386	19.0053
		18.8572	18.8245	18.7803	18.7718	18.7555
		22.1282	21.3195	21.1995	20.9014	20.8644
		20.6676	20.6560	20.6287	20.6013	20.5727

Figure 10: Bottom 10 ranked results foe ResNet 50

6.5 Future Improvements

One possible tweak that could improve the performance would be use a deeper model architecture like ResNet101. We observed that there has been a steady increase in test performance when the model size gets large (from ResNet18 to ResNet34 to ResNet50). When we trained on ResNet50, the model didn't overfit since testing accuracy was steadily increasing. Therefore, our deduction is that using a deeper model shouldn't overfit on the validation/test set, and should yield an increase in test performance.

In addition, our negative image sampling method didn't include in-class samples that are very distant from the query image, as the paper performed. Accounting for in-class samples will increase the quality of training triplets and thus should result in better training and test performance.

7 Appendix

We also trained the ResNet34 model, which achieved a test accuracy of 55%. The top and bottom 10 ranked results obtained using the ResNet34 model are listed below.

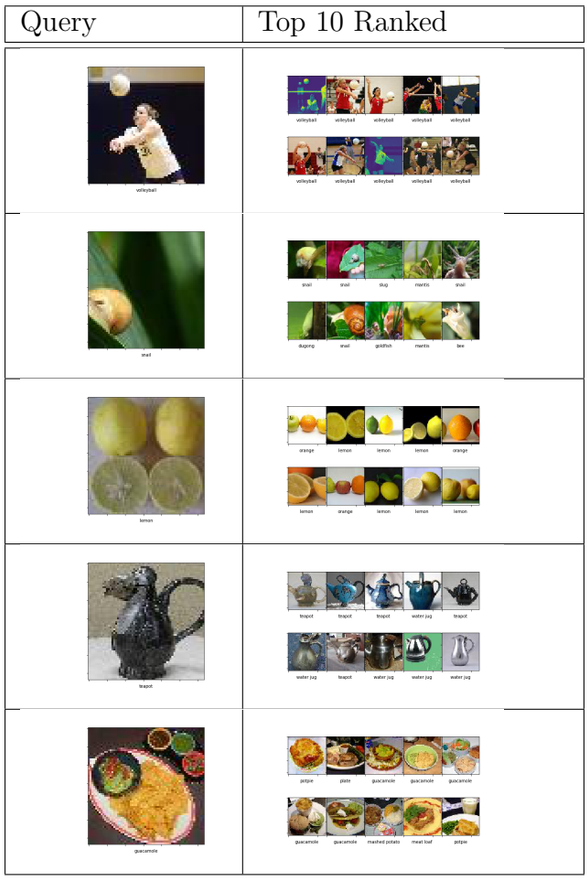


Figure 11: Top 10 ranked results for ResNet 34

Figure 12: Bottom 10 ranked results for ResNet 34

References

[1] D.G. Lowe, Object Recognition from Local Scale-Invariant Features. Proceedings of the Seventh IEEE International Conference on Computer Vision. DOI: 10.1109/ICCV.1999.790410

[2] Navneet Dalal, Bill Triggs. Histograms of Oriented Gradients for Human Detection. International Conference on Computer Vision Pattern Recognition (CVPR '05), Jun 2005, San Diego, United States. pp.886–893, ff10.1109/CVPR.2005.177ff. ffinria-00548512f

[3] T. Deselaers and V. Ferrari. Visual and semantic similarity in imagenet. In CVPR, pages 1777–1784. IEEE, 2011.

[4] J. Wang, Y. Song, T. Leung, C. Rosenberg, J. Wang, J. Philbin, B. Chen, and Y. Wu. Learning fine-grained image similarity with deep ranking. CoRR, abs/1404.4661, 2014.