Heirarchical Feature Learning in Deep Convolutional Networks

Sanchit Nevgi
snevgi@umass.edu

Abstract

Deep Convolutional Neural networks (DCNN) are able to learn abstract features in an image. However, the decisions that the network makes to reach the outcome are still not understood very well. On the other hand, studies have shown that humans, when looking at an image, break it down into the most fundamentally interpretable features. As children and as adults, we grasp a concept better when we understand the basics. In this work, we attempt to explicitly training the network on the input features and try to contrast its performance without such guidance. In a novel approach, called patch-training, we generate multiple random patches of the region-of-interest and train a DCNN as a form of pre-training. We modify the well-known architecture VGG-16 to accomodate learning of these new features. This experiment shows that learning on the patches indeed improves the recall rate for the multi-label classification task with significant reduction in training time due to reduced feature space.

1 Introduction

Deep Convolutional Neural networks (DCNNs) have state-of-the-art performance in image classification and object detection tasks. Classically, when training a neural network, we present the network with a sample of the dataset, usually in a random order. The model performs a forward pass on this sub-sampled data, subsequently a gradient is computed (on a criterion) and propagated in the backward pass, after which the parameters are updated. However, the decision-making process of Deep Neural Networks are still largely unclear and non-deterministic.

A neural network model is essentially a universal function approximator, that tries to model the intricate dependencies between the input features and output labels. In the case of image tasks, such as object detection, classification and segmentation, by plotting the saliency map, we see that the output features are influenced by contiguous regions of input pixels. That is, features at higher levels. are composed of lower level features.

This phenomena is further explored in recent research [1] [4]. By visualizing the feature representations in the hidden layers, we understand that they are indeed learning progressively higher level features. For example, in face detection, the output activations of the initial layers react to different edges, while deeper layers are able distinguish between a face and the body.

According to Bengio et al, humans learn better when examples are presented in a meaningful order. Bengio proposes a technique knows as Curriculum Learning [2], which gives better results on the speed of convergence as well as obtaining a better local optima.

2 Background/Related Work

Curriculum Learning
Imposing a curriculum is a method to guide the training of neural networks. [2] As children, we

learn from a fixed curriculum, where concepts are introduced to us from easy to difficult. Each new concept builds on prior concepts to help us grasp them. Bengio hypothesizes that choosing the examples and their order, not only leads to faster convergence but also a better quality of local minima. This strongly suggests that the choice of training strategy has a greater role to play than previously thought.

Similar training procedures have been applied to Natural Language processing tasks, where the model was initially trained on basic language syntax. Similarly, multi-stage curriculum learning have been shown to give improved generalization in vision and language tasks.

We develop on this idea, to enable a model to learn the rudimentary features of the region of interest explicitly rather than having to learn them implicity.

The choice curriculum is left up to the researcher. Other techniques have been applied to generate this curriculum automatically. A recent approach known as MentorNet [10], uses two networks in its architecture — MentorNet (Complex model) and StudentNet (Simple model). The mentor generates the curriculum, and expects the student to mimic its results.

Restricted Boltzmann machines

Boltzmann machines [8] are a class of stochastic and generative neural networks, introduced by Geoffrey Hinton. Restricted Boltzmann Machines (RBMs) impose an additional constraint that there are no connections between units of the same layer. Essentially, they are two-layered neural networks with no output layer. The lack of output layer allows for more efficient training algorithms. A stack of RBMs allows us to learn more complex relations between inputs and outputs. The training procedure is similar multi-stage hierarchical learning.

Spatial Pyramid Pooling

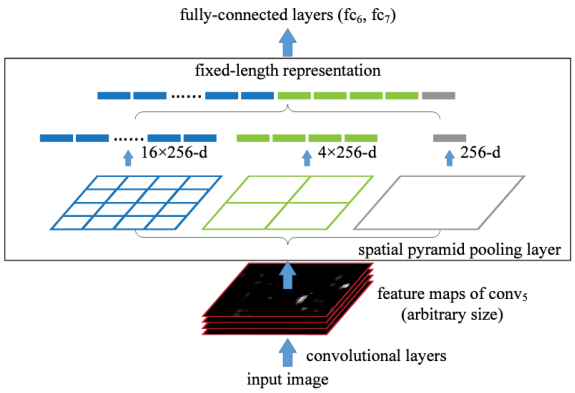


Figure 1: Spatial Pyramid Pooling

The inputs to a neural network models are usually inputs of fixed square sizes (eg 224 x 224). The square sizes of image also limits aspect ratio. Generally, this is achieved by center cropping the image or resizing the image till the desired shape is achieved (through bilinear interpolation). Doing so, we lose important spatial information. Convolutional architectures comprise of two parts, the convolutional layers and dense layers deeper in the model. Since convolutional layers behave similar to a sliding window protocol, they have no restriction on the input size. This restriction is imposed by the fully connected layers. Kaiming He et al [7] proposes a Spatial Pyramid Pooling method to generate a fixed-size representation of features, regardless of input size. The procedure is as follows, the Spatial Pyramid Pooling is inserted after the final convolutional layers. Pooling layers (MaxPool) of different kernel sizes (1, 3, 5, etc) are applied on the intermediate activations. The results of each of the pooling operations are flattened into a 1-d vector and concatenated (end-to-end), to yield a fixed-sized vector (Figure). The intuition is that, the different-sized filters capture feature activations at different scales.

(Liang-Chieh Chen et al) builds on top of this architecture by using dilated (or atrous) convolutions over pooling operations [5]. Like the SPP module, after the last convolutional layer, they use multiple filters of the same kernel sizes ($kernel\_size = 3$), but with different dilation rates $dilations \in 6, 12, 18$. The results are concatenated and a $1x1$ convoultion is applied. This effectively increases the receptive field, capturing larger spatial dependencies, and does not destroy contextual information like the pooling operations.

We use the pyramid pooling to account for the different patch sizes to obtain a fixed sized representation.

Bag of Features
The Bog-of-Features (BoF) approach relates closely to our idea of patch-training [14]. It has shown to be successful in large scale object recognition [3] [12]. In this approach, the activations of the different patches are computed. A heatmap is generated of all the patches per class. The model is trained on the generated heatmaps. This idea has is analogous to the Bag-of-words approach used in Natural Language Processing tasks.

3 Datasets

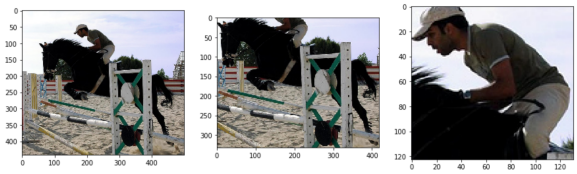


Figure 2: Pascal VOC, bounding box extraction. Left: Full image. Center: horse class. Right: person class

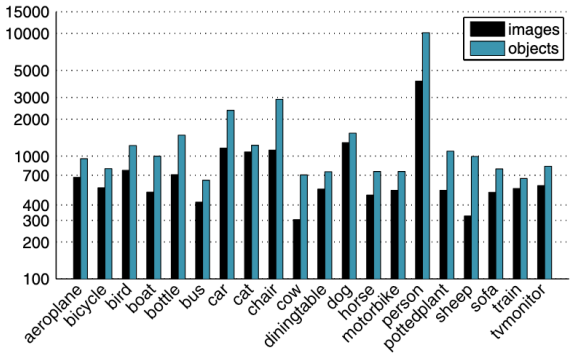


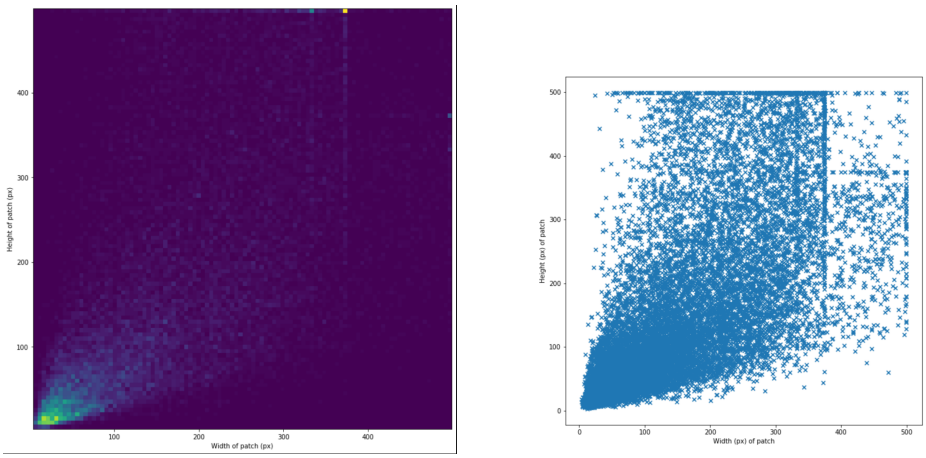
Figure 3: Pascal VOC class distribution

A high-quality dataset is neccesary to learn the intricate features of input images. We explore the Microsoft COCO [13] and Pascal VOC [6] datasets. The COCO dataset has over 100 classes and 160k images. To train effectively on this dataset, would require a model with a large capacity and substantial compute resources. The Pascal VOC has 11k examples, split into roughly 5.5k train set and 5.5k test set. This allows us to experiment rapidly on different architectures and hyperparameters.

Pascal VOC comprises of 20 classes:

```
aeroplane, bicycle, bird, boat, bottle, bus, car, cat, chair, cow,
diningtable, dog, horse, motorbike, person, pottedplant, sheep,
sofa, train, tv
```

There can be multiple objects in each image, making this a multi-label classification task.



(a) Heatmap of the patch regions. We see that smaller patches are higher density

(b) Scatter plot of patch sizes

Figure 4: Confusion matrix visualization

For model training we use two datasets — `FullSet` and `PatchSet`. The `FullSet` comprises of the full resolution train images.

The `PatchSet` is derived as follows — Using the bounding boxes of the images, we derive a new dataset comprising of the extracted objects. From 5k train images, we obtain 15k patch dataset.

To ensure that each patch has adequately representable features, we filter objects of smaller patch sizes. In our initial expreriments, we only consider patch sizes of more than $64px$ resolutions.

For both the datasets, we convert the images to Tensors and normalize the distribution using the `ImageNet` mean and standard deviation. This zero-centers the images and helps the model converge faster.

The `PatchSet` contains data with only a single label (single-label classification task).

4 Methodology

The models were built using `PyTorch 1.3`, and trained on Google Cloud AI Platform on an NVIDIA Tesla K80 GPU (1GB). Input images are represented in PIL (Python Imaging Library) format , using the Pillow Library [15]. The `torchvision` package consists of popular datasets, model architectures, and common image transformations for computer vision tasks. We use the `Normalize`, `RandomResizedCrop`, `ToTensor` transforms from the torchvision package. Additionally, torchvision provides a wrapper over the Pascal VOC dataset. We choose the dataset corresponding to the 2012 (latest) detection challenge.

We implement the VGG-16 architecture as a baseline model. VGG is chosen for its simplicity, since it doesn't have skip connections, making it amenable to patch-training.

VGG-16 comprises of 6 blocks. In each of the first 5 blocks, there are 2-3 convolutional layers of $kernel\_size = 3$, $stride = 1$ and $padding = 1$. The padding ensures that the spatial dimensions are intact through the Convolutional layers. Each Convolutional layer is followed by a ReLU non-linearity, and finally a MaxPool layer with $kernel\_size = 2$ and $stride = 2$. After the convolutional layers, VGG adds a `AdaptiveAvgPool` layer, to ensure that the output size will adhere to 7x7 resolution. The last block comprises of 3 Fully Connected (FC) layers (4096, 4096, 20, respectively), with ReLU and Dropout inserted between.

We modify the VGG-16 architecture by adding an additional branch after the last convolutional layer, by adding a `AdaptiveAvgPool` layer of output size (4x4). We also add a new classifier head, consisting of 3 Fully Connected (FC layers) (Figure).

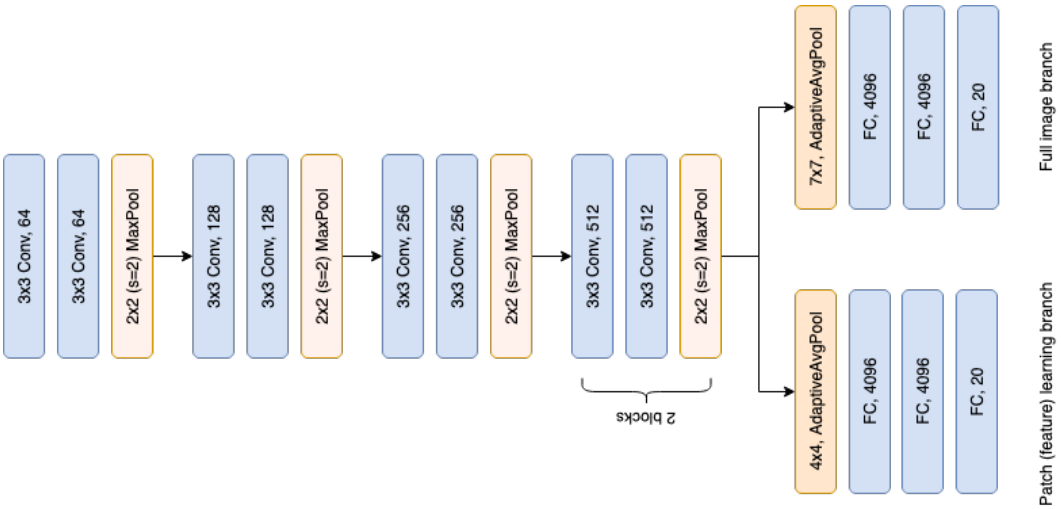


Figure 5: Our modified VGG-16 Architecture

We are essentially merging two different tasks — single-label and multi-label classification into one [9]. Therefore, we define two loss functions for each of them. It is helpful to note that, we only forward pass through one branch at a given epoch, and compute the gradient and backpropagate for one branch only. To simplify experiments, we specify a $switch\_epoch$. Training when the $epoch < switch\_epoch$ is done on the `PatchSet`, otherwise training is done on the `FullSet`.

Algorithm 1: Training procedure

```
switch_epoch = 5 ;
for epoch in epochs do
    trainPatch = Bool(epoch ≤ switch_epoch) ;
    if trainPatch then
        dataLoader = PatchLoader ;
        criterion = CrossEntropyLoss() ;
    else
        dataLoader = FullLoader ;
        criterion = BCEWithLogitsLoss() ;
    end
    for input, target in dataLoader do
        preds = model.forward(inputs, trainPatch) ;
        loss = criterion(preds, target) ;
        Backpropagate ;
    end
end
```

The patch-training uses cross entropy loss, `nn.CrossEntropyLoss`, defined as

$$loss(x, class) = -\log\left(\frac{\exp(x[class])}{\sum_j \exp(x[j])}\right) \tag{1}$$

The full-training uses binary cross entropy (`nn.BCEWithLogitsLoss`), which posits the existence of a given class.

For class c, the loss is given by —

$$loss_c(x, y) = -y_n \log(\sigma(x_n)) - (1 - y_n) \log(1 - \sigma(x_n)) \tag{2}$$

The output from a forward pass of the model is a 20-d Tensor giving logits for each class. For the `PatchSet` data, we one-hot encode the labels and use cross entropy loss. For the `FullSet` data, we multi-hot encode the labels and sum over the binary cross entropy loss over each class. $nn.BCEWithLogitsLoss$ generalizes to the multi-label case, by keeping the output and input dimensions the same.

The Pascal VOC dataset is inherently imbalanced (as seen in the figure). The 'person', 'chair' and 'car' classes are present in higher proportion as compared to other classes. Moreover, the multi-hot encoded labels results in a many more negative examples compared to positive examples.

From our expreriments with training the model as is, the model pushes all predicions to 0, (class absent in the image). Clearly, this leads to lowest average loss across all datapoints. However this generalizes poorly and the predictions made are not useful. To accomodate for this imbalance, we assign higher weights to positive samples. These weights are calculated based on the distribution of the training data (Figure).

The loss value is not an accurate indicator of the model performance, since the loss for training on patches is significantly different from the model trained on full images. Canonically, for classification we use precision and recall metric.

$precision = \frac{TP}{TP+FP} \quad recall = \frac{TP}{TP+FN}$

where `TP` is the number of True Positives, `FP` are False Positives and `FN` are False Negatives.

5 Experiments & Results

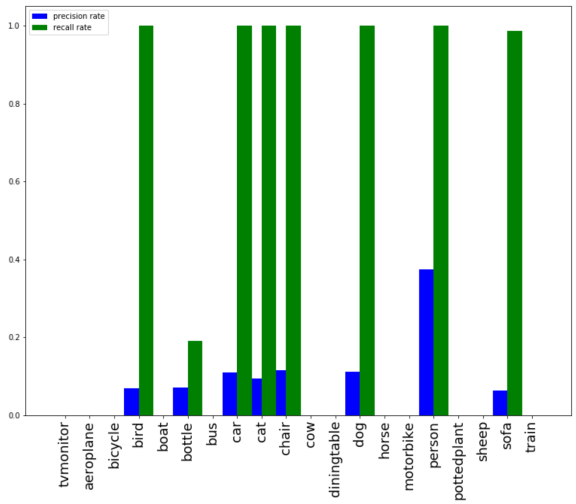


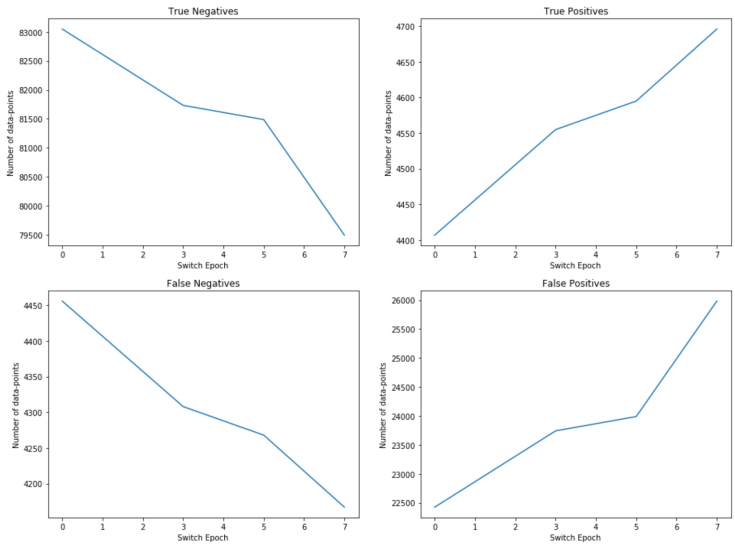
Figure 6: Precision and Recall rates for $switch\_epoch = 5$ for all classes

We compute the `PatchSet`, which involves extracting the pixels from the bounding box, converting them to tensors and normalizing them. We save them as tensors and for all further expreriments we create a dataset from the cached values, using torchvision's `TensorDataset`.

In all experiments, we train the model for 10 epochs. Further, we use a $switch\_epoch \in [0, 10]$. $switch\_epoch = 0$, corresponds to the model being trained completely on the `FullSet`. This serves as the baseline model for comparison.

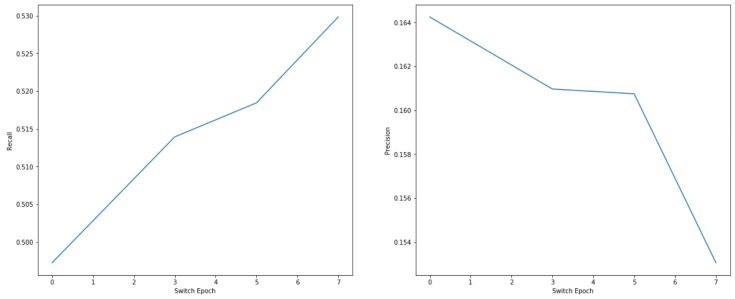
Since this is a multi-label classification, we compute the Confusion Matrix for each class (20 x 2 x 2) on the validation set after training for each switch epoch.

The complete model is trained on a single GPU, while the metrics are computed on the CPU. We use `sklearn.metrics.multilabel_confusion_matrix` for the Confusion matrix computation.



(a) Plot of TP, FP, FN, and TN over all classes vs $switch\_epoch$

Figure 7



(a) Precision and Recall rates vs $switch\_epoch$

Figure 8

To calculate a single metric, we sum the confusion matrix over all classes to yield a $2\text{x}2$ matrix. Next, we compute the precision and recall for this using the given formula. We plot the precision and recall rates as a function of $switch\_epoch$.

From the plot, we see that the recall rate improves with $switch\_epoch$. There is 3% increase in the recall rate (from 0.5 to 0.53) over all classes. However, there is a decline in the precision — 1% decrease over all classes. One seemingly obvious explaination is that, since the model is trained on the patches, which are positive examples, they have less exposure to the negative examples (one where the class is not present). This causes the False Positives and True Positives to increase proportionally. This results in the precison rate $\left(\alpha \frac{1}{FP}\right)$ to reduce.

The same explaination can be extended to the $recall$ rate. The higher training on positive examples, makes the True Negatives and False Negatives to reduce. So the $recall$ rate imporoves $\left(\alpha \frac{1}{FN}\right)$.

Notice however, that the percentage improvement and degradation in the recall and precision rates respectively is not the same. The recall rate improves significantly more. The choice of $switch\_epoch$ also plays a huge factor in the performance of the model. Our model, has a sharp decline in the $precision$ rate after $switch\_epoch = 5$. By simple cost-benefit analysis, this $switch\_epoch$ provides the best trade-off between the metrics. $+2\%$ recall rate and -0.4% precision. $switch\_epoch > 7$ perform worse on all metrics compared to the baseline.

Hyper-parameter tuning

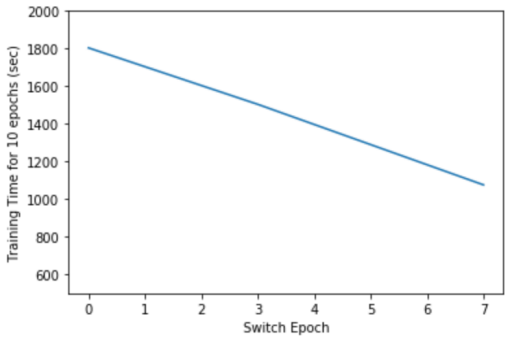


Figure 9: Training time vs $switch\_epoch$

We perform an ablation study on the model. The initial hyper-parameters were chosen from the set that gave the best performance on the baseline model (VGG-16) We use the `Stochastic Gradient Descent` optimizer [11] with an initial learning rate of $1e-3$ and $momentum = 0.9$

We train our model on different learning rates to verify if convergence is faster or a better quality of local optima is obtained. From the experiments (figure), we see that a $learningrate < 0.001$ provides roughly the same result, and higher learning rates give poor generalization performance.

Learning rate	Precision	Recall
0.0001	0.159	0.52
0.001	0.161	0.518
0.01	0.166	0.494

From relevant work, we learn that a larger batch size gives a marginal improvement over performance metrics. We choose an intial $batchsize = 16$. We compare this to the performance on $batch\_size = 8, 32$

Batch Size	8	16	32
Precision	0.128	0.134	0.134
Recall	0.632	0.604	0.605
Train time (s)	2231	1266	1136

6 Conclusion & Future

From the experiments we conclude that the training procedure has more impact on the quality of local minima obtained as well as the speed of convergence than previously thought. Training on the positive examples, by patch-training, in the object classification task has some merit. We are able to control the trade-off between precision and recall by modifying the $switch\_epoch$. More training on the patches improves the $recall$ rate, while degrading the $precision$ rate. Inspite of this caveat, the improvement in $recall$ is significantly more than the reduction in $precision$ rate. This could potentially be exploited by increasing the network capacity to improve overall classification accuracy. The training time also reduces significantly (over 30% reduction over 10 epochs). For more complex and deeper networks, this reduction will become even more evident.

In literature, guided training in neural networks has shown a lot of promise. Inspite of this, it is not applied in practice much. One possible explanation is that, the choice of training procedure is not widely researched yet and no specific guidelines exist to implement them. In the past two years, this area has gathered a lot of interest, particulary using neural networks to guide the training of other networks.

The next logical step would be to explore more neural architectures and make them amenable to guided training and ensuring that the network understands which features are important without compromising the overall performance.

References

[1] Artem Babenko and Victor S. Lempitsky. Aggregating deep convolutional features for image retrieval. *ArXiv*, abs/1510.07493, 2015.

[2] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, pages 41–48, New York, NY, USA, 2009. ACM.

[3] Wieland Brendel and Matthias Bethge. Approximating cnns with bag-of-local-features models works surprisingly well on imagenet. *ArXiv*, abs/1904.00760, 2019.

[4] Chaofan Chen, Oscar Li, Alina Barnett, Jonathan Su, and Cynthia Rudin. This looks like that: deep learning for interpretable image recognition. *CoRR*, abs/1806.10574, 2018.

[5] Liang-Chieh Chen, George Papandreou, Florian Schroff, and Hartwig Adam. Rethinking atrous convolution for semantic image segmentation, 2017.

[6] M. Everingham, S. M. A. Eslami, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. The pascal visual object classes challenge: A retrospective. *International Journal of Computer Vision*, 111(1):98–136, January 2015.

[7] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Spatial pyramid pooling in deep convolutional networks for visual recognition. *CoRR*, abs/1406.4729, 2014.

[8] G. E. Hinton. Boltzmann machine. *Scholarpedia*, 2(5):1668, 2007. revision #91076.

[9] Geoffrey E. Hinton, Oriol Vinyals, and Jeffrey Dean. Distilling the knowledge in a neural network. *ArXiv*, abs/1503.02531, 2015.

[10] Lu Jiang, Zhengyuan Zhou, Thomas Leung, Li-Jia Li, and Li Fei-Fei. Mentornet: Learning data-driven curriculum for very deep neural networks on corrupted labels, 2017.

[11] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014.

[12] Svetlana Lazebnik, Cordelia Schmid, and Jean Ponce. Beyond bags of features: Spatial pyramid matching for recognizing natural scene categories. *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*, 2:2169–2178, 2006.

[13] Tsung-Yi Lin, Michael Maire, Serge J. Belongie, Lubomir D. Bourdev, Ross B. Girshick, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. Microsoft COCO: common objects in context. *CoRR*, abs/1405.0312, 2014.

[14] Eva Mohedano, Amaia Salvador, Kevin McGuinness, Ferran Marqués, Noel E. O'Connor, and Xavier Giró. Bags of local convolutional features for scalable instance search. In *ICMR '16*, 2016.

[15] Stéfan van der Walt, Johannes L. Schonberger, Juan Nunez-Iglesias, François Boulogne, Joshua D. Warner, Neil Yager, Emmanuelle Gouillart, Tony Yu, and the scikit-image contributors. scikit-image: image processing in Python. *PeerJ*, 2:e453, 6 2014.