Supplement: Differentiable Patch Selection for Image Recognition

Jean-Baptiste Cordonnier[1†*] Aravindh Mahendran[2†] Alexey Dosovitskiy[2]
Dirk Weissenborn[2] Jakob Uszkoreit[2] Thomas Unterthiner[2]
[1]EPFL, Switzerland [2]Google Research, Brain Team
jean-baptiste.cordonnier@epfl.ch
{aravindhm,adosovitskiy,diwe,usz,unterthiner}@google.com

The supplementary material consists of the following: performance trade-offs associated with patch sampling versus running a CNN on the entire high resolution image (Appendix A), some theoretical discussions regarding the LP formulation of index-sorted Top-K in (Appendix B), an experiment quantifying the effect of decaying σ to 0 during training in (Appendix C), results with another differentiable top-K method we experimented with before developing our approach (Appendix D), hyper-parameter details for all our experiments (Appendix E), qualitative results (Appendix F), and a PyTorch implementation of the perturbed Top-K module (Appendix G).

A. Speed Improvements by Sampling Patches

We study the speed improvement that can be gained at inference by using our patch extraction model compared to running a model on the full image. We compare the number of samples processed per second at inference on a single V100 GPU in Figure 1. If the useful information for recognition is localized within than 10% of the pixels, which corresponds to extracting $K = 10$ patches of size 100×100 on a MegaPixel image, processing only the relevant regions with the same network (ResNet50) allows a 5 fold speed up. In this case, roughly 28 % of the inference time is spent on the feature network, while most of the remaining time is spent calculating the scores for the individual patches. The Top-K operation's influence on runtime is minimal. This should be put in contrast with the common alternative of running a significantly smaller ResNet18 on the full size image which is not faster on 1000×1000 images than our approach running a ResNet50 on the extracted patches. Using hard Top-K at inference instead of the differentiable version is consistently faster on a V100 GPU.

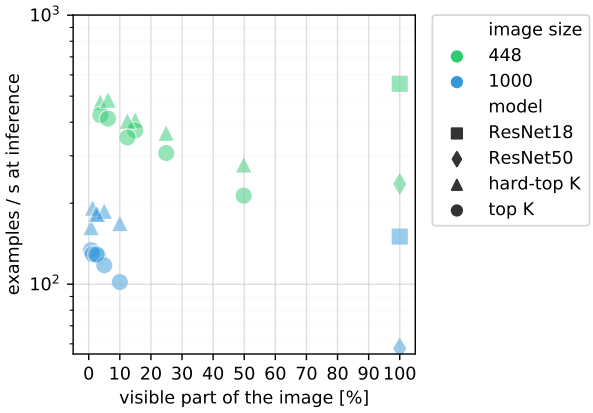


Figure 1: Comparison of inference speed on a single V100 between ResNet 50 at full resolution images vs. extracting only $K \in \{3, 5, 10\}$ patches of dimension $P \in \{50, 100\}$ with differentiable or hard Top-K. The feature network is a ResNet50 and aggregation is mean-pooling. x-axis is the percentage of extracted pixels $\left(K \cdot P^2\right)$ compared to the full image. For each point in this plot, the maximum over feasible batch-sizes was used to obtain the highest possible throughput.

B. Linear Program

To understand why the LP, in equations 4 and 5, corresponds to index sorted top-K, let us focus on the integral solutions $\boldsymbol{Y}_{ik} \in \{0, 1\}$. In this scenario, each column of $\boldsymbol{Y}$ is a one-hot indicator vector selecting exactly one of the scores in s. Furthermore, the objective function is the sum of selected scores. Thus the optimal solution is to select the K highest scores.

Lastly, the strict inequality constraint forces the selection of numbers with lower indices earlier. This can be proven by contradiction:
Consider a valid solution which has $l > m$, but s_l was selected by the k^{th} column and s_m by k'^{th} column such that $k < k'$. That is, $\boldsymbol{Y}[:, k] = \text{one hot}(l)$ and $\boldsymbol{Y}[:, k'] = \text{one hot}(m)$. Then, since this solution is valid the following

*Work done during internship at Google Research. † Equal contribution.

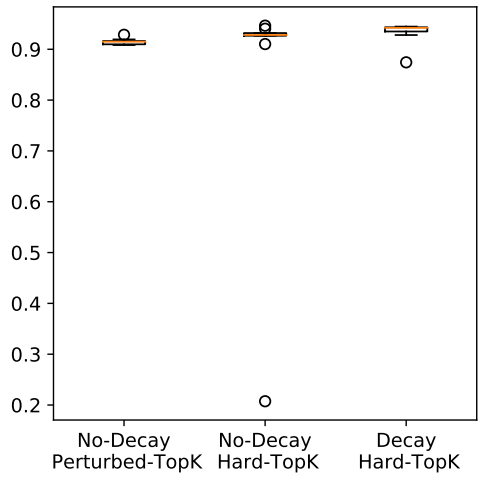


Figure 2: Ablating the effect of reducing σ (perturbation magnitude) to 0 during training. Each setting is repeated 9 times. *Left:* σ constant and perturbed top-K for both training and inference. *Middle:* σ constant, perturbed top-K for training and hard top-K for inference. *Right:* σ decays to 0 during training, perturbed top-K during training and hard top-K for inference.

holds

$$\sum_{i\in[N]} i\boldsymbol{Y}_{i,k} < \sum_{j\in[N]} j\boldsymbol{Y}_{j,k'} \tag{1}$$

$$\implies l < m \tag{2}$$

which is a contradiction to the original assumption.

C. Decaying σ to 0

We linearly decay perturbation magnitude σ to 0 in all our models. We ablate this design choice on the billiard balls dataset. These experiments use the transformer aggregation head. All other hyper-parameters are kept the same. In fig. 2 and table 1 we see that using hard top-k at inference leads to higher accuracy ($+1.3\%$). Decaying perturbation magnitude to zero further improves performance ($+1.4\%$).

	test acc. [%]
No-Decay, Perturbed-TopK	91.5 ± 0.4
No-Decay, Hard-TopK	92.8 ± 0.4
Decay, Hard-TopK	94.2 ± 0.2

Table 1: Ablating the effect of reducing σ (perturbation magnitude) to 0 during training. See caption in fig. 2 for details.

D. Differentiable Sinkhorn

Another approach to make Top-K differentiable was proposed by Xie *et. al* [4] and relies on the optimal transport formulation of Top-K proposed by Cuturi *et. al* [1]. We implemented the forward and backward pass following Algorithm 3 of [4]. We report the results for traffic sign recognition in Table 2 with similar setting as in Section 5.1. This approach gives good results but suffers when using discrete Top-K at inference.

	test acc. [%]
Sinkhorn Top $K = 5$	95.4 ± 0.7
+ discrete Top-K at inference	83.0 ± 10.5
Sinkhorn Top $K = 10$	92.6 ± 3.1
+ discrete Top-K at inference	86.3 ± 1.7

Table 2: Performance of Sinkhorn Top-K based models on the traffic signs dataset. We report the mean and standard deviation across 5 runs.

E. Experimentation Details

For all experiments except the Fine-grained bird classification ones, our scorer network consisted of a CNN with 4 convolutional layers of kernel size 3×3 with stride 1 and "valid" padding. The number of feature maps was 8, 16, 32 and 1, respectively. Every convolution except for the last was followed by a Relu activation function. The last convolution was followed by an 8×8 max pooling of stride 8.

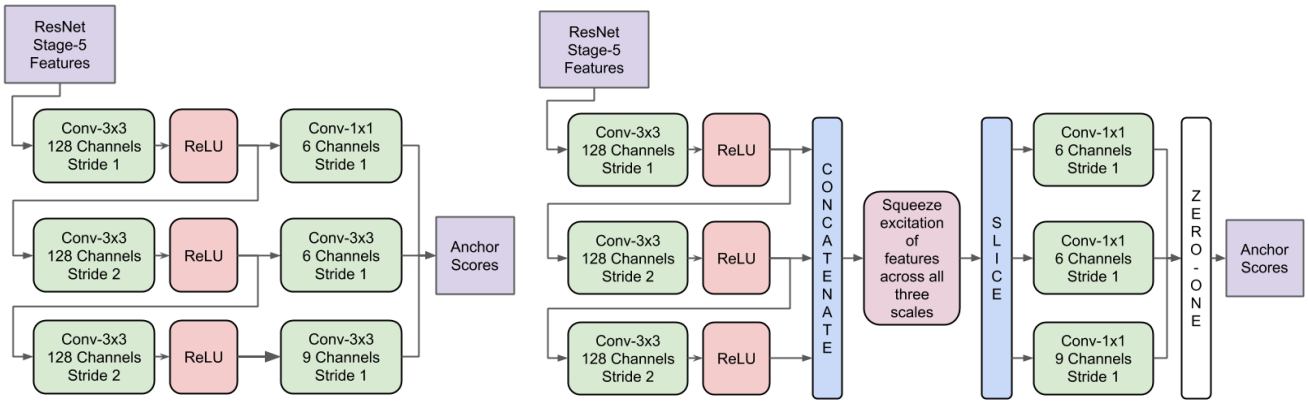
Our feature network was different depending on the dataset: On the Swedish Traffic Signs dataset, we used the same feature network as ATS [2], which is a narrow ResNet18 with 16 filters instead of the usual 64, 128, 256, 512. On the billiard balls dataset we used a standard ResNet18. On the CUB-200 dataset we use the same feature extractor as NTS-Net, that is a ResNet50.

We used the ADAM-W optimizer [3] for Traffic signs and billiard balls dataset. We used SGD with momentum 0.9, similar to NTS-Net, for CUB-200. The exact details of our optimizers can be seen in Table 3. We found that weight decay coupled with momentum updates was better to reduce overfitting on CUB-200.

In our adaptation of the NTS-Net, we added Squeeze Excitation layers inside the region proposal network (RPN). This was meant to compensate for the lack of non-maximum suppression in our patch selection module. The RPN for both our method and the baseline NTS-Net are shown in figure 3. Lastly, with regards to data pre-processing, instead of normalizing pixels by a pre-computed pixel mean and pixel standard deviation, we re-scaled pixel values to lie between $[-1, 1]$ during training.

Dataset	Traffic Signs	Billiard balls	CUB-200
Batch Size	32	64	16
Learning rate (LR)	10^{-4}	10^{-4}	10^{-3}
Weight decay	10^{-4}	10^{-4}	10^{-4}
Steps	100 000	30 000	31 300
LR Schedule	Cosine decay	Cosine decay + 5% warm-up	Piece-wise constant + Decay by 0.1 at 60 and 80 epochs + 5% warm-up
Optimizer	Adam-W	Adam-W	SGD + Momentum
Entropy regularizer	0.01	0.01	0.05
Gradient clip value	0.1	1.0	-

Table 3: Summary of optimization hyper-parameter settings.



(a) NTS-Net (b) Top-K NTS-Net

Figure 3: Architecture of the region proposal head used in the baseline NTS-Net model (a), and in our adaptation (b), for fine-grained classification in the CUB-200 dataset. We added a squeeze excitation layer in the middle to allow for communication between all features just before anchor prediction. Zero-one normalization is applied across all 1614 scores.

This was to match the value range we used for pre-training our ResNet50 backbone on ImageNet ILSVRC12. Other than this minor change, we used the same data augmentation as NTS-Net.

F. Qualitative Results

We visualize patches extracted by the model on the CUB-200 dataset, on the test split, in fig. 4. This particular model is the best performing seed among the 5 random repeats we averaged to report the number for $K = 4$ in the main manuscript. It uses 'mean' aggregation and achieves 87.3% top-1 accuracy. The images visualized are not cherry picked. We see that the model is able to focus on the bird and captures the region around the eyes and torso. The lack of NMS is evident as the model often looks at patches with high overlap.

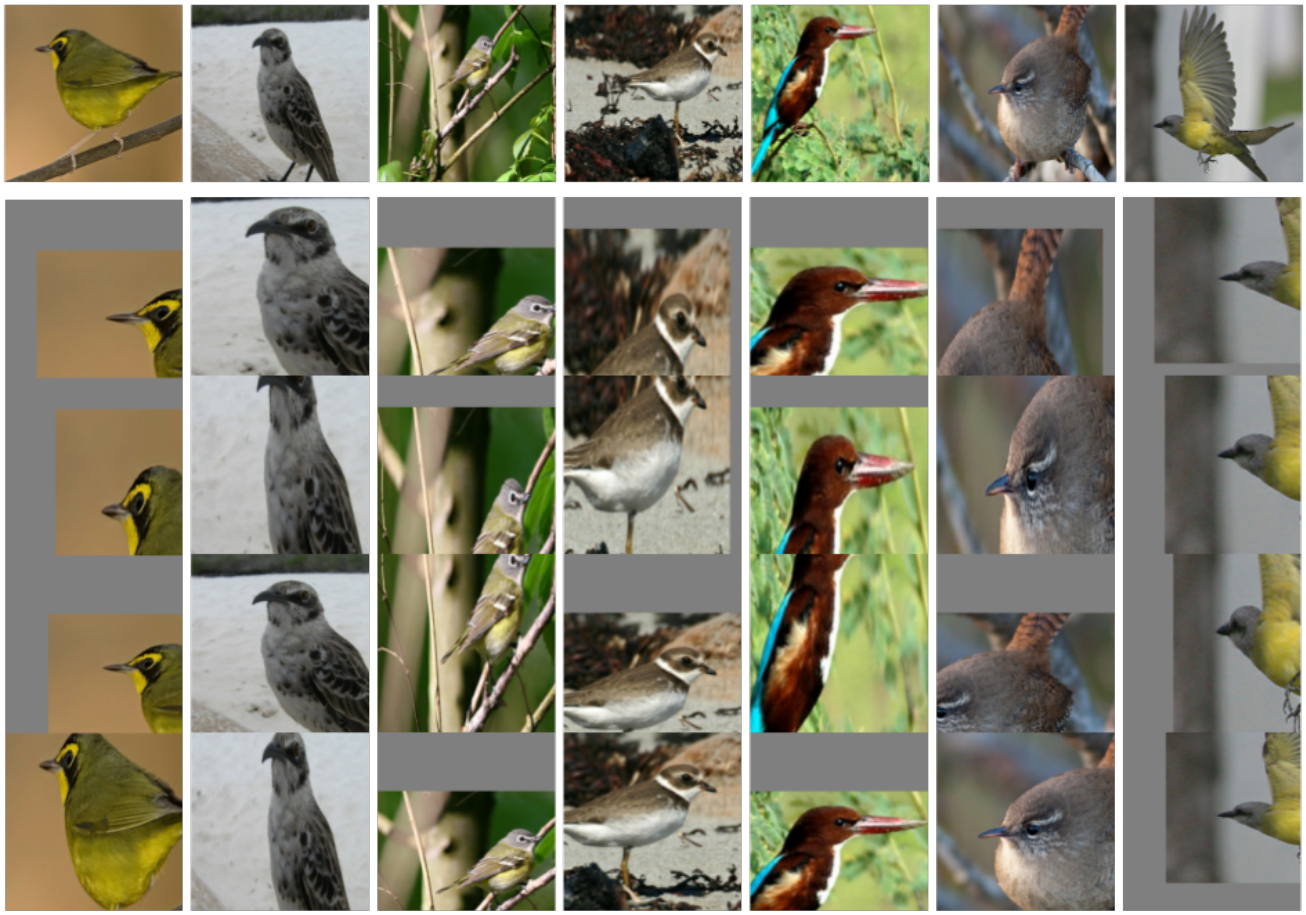


Figure 4: Patches extracted from images in the test split of the CUB-200 dataset. The first row shows the original input image after being resized to 600×600 followed by a centre crop of 480×480. Padding artifacts correspond to selecting anchors that include pixels outside the image boundary.

G. Differential Top-K PyTorch Code

```python
import torch
import torch.nn as nn


class PerturbedTopK(nn.Module):
    def __init__(self, k: int, num_samples: int = 1000, sigma: float = 0.05):
        super(PerturbedTopK, self).__init__()
        self.num_samples = num_samples
        self.sigma = sigma
        self.k = k

    def __call__(self, x):
        return PerturbedTopKFunction.apply(x, self.k, self.num_samples, self.sigma)


class PerturbedTopKFunction(torch.autograd.Function):
    @staticmethod
    def forward(ctx, x, k: int, num_samples: int = 1000, sigma: float = 0.05):
        b, d = x.shape
        # for Gaussian: noise and gradient are the same.
        noise = torch.normal(mean=0.0, std=1.0, size=(b, num_samples, d)).to(x.device)

        perturbed_x = x[:, None, :] + noise * sigma  # b, nS, d
        topk_results = torch.topk(perturbed_x, k=k, dim=-1, sorted=False)
        indices = topk_results.indices  # b, nS, k
        indices = torch.sort(indices, dim=-1).values  # b, nS, k

        # b, nS, k, d
        perturbed_output = torch.nn.functional.one_hot(indices, num_classes=d).float()
        indicators = perturbed_output.mean(dim=1)  # b, k, d

        # constants for backward
        ctx.k = k
        ctx.num_samples = num_samples
        ctx.sigma = sigma

        # tensors for backward
        ctx.perturbed_output = perturbed_output
        ctx.noise = noise

        return indicators

    @staticmethod
    def backward(ctx, grad_output):
        if grad_output is None:
            return tuple([None] * 5)

        noise_gradient = ctx.noise
        expected_gradient = (
            torch.einsum("bnkd,bnd->bkd", ctx.perturbed_output, noise_gradient)
            / ctx.num_samples
            / ctx.sigma
        )
        grad_input = torch.einsum("bkd,bkd->bd", grad_output, expected_gradient)
        return (grad_input,) + tuple([None] * 5)
```

References

[1] Marco Cuturi, Olivier Teboul, and Jean-Philippe Vert. Differentiable ranking and sorting using optimal transport. In *NeurIPS*, 2019. 2

[2] Angelos Katharopoulos and François Fleuret. Processing megapixel images with deep attention-sampling models. In *ICML*, 2019. 2

[3] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *ICLR*, 2019. 2

[4] Yujia Xie, Hanjun Dai, Minshuo Chen, Bo Dai, Tuo Zhao, Hongyuan Zha, Wei Wei, and Tomas Pfister. Differentiable top-k operator with optimal transport. *NeurIPS*, 2020. 2