A Journey Inside DeepInversion

236781 - Final Project Report

Adam Botach
botach@cs.technion.ac.il

Rafi Cohen
raficohen@cs.technion.ac.il

1 Abstract

In this project we present a full implementation for DeepInversion, a method which allows reconstructing high-quality class-conditional images from a pretrained CNN using the hidden data stored in its batch normalization layers. The method was first introduced by Yin et al. [10] in December 2019, and no public implementation was available for it during our time working on this project. After attempting to tune the method (a very challenging and time-demanding task), we demonstrate how our implementation is able to yield decent quality images. We also explore some of DeepInversion's usages, such as image synthesis for knowledge distillation purposes. Under our computational limitations, we synthesize a small dataset from a "teacher" CNN pretrained on ImageNet and then use the synthesized images to train a new "student" model from scratch. We conclude, however, that in order to achieve good results one would have to tune our implementation even further and synthesize a much larger training set, but that would require more powerful hardware than what was available to us.

Finally, we take a look at the findings of Ramanujan et al. in [7], who have recently introduced the edge-popup algorithm for finding well performing subnetworks that are hidden within randomly initialized deep networks. Specifically, we take a well performing convolutional subnetwork that was discovered using the edge-pop algorithm and attempt to see "what" it has learned by "inverting" it using DeepInversion. Unfortunately, we conclude that since these subnetworks do not contain the hidden data required for DeepInversion, "inverting" them does not lead to informative results.

2 Introduction

2.1 Deep Dream

Deep Dream [1] is a popular method that have been proposed to visualize what a deep network expects to see in an image. Given a pretrained classification CNN $\mathcal{N}$ and an input image $\hat{x}$, the method forwards $\hat{x}$ through $\mathcal{N}$ up to a certain pre-chosen layer ℓ, and then tries to maximize the activations of ℓ by using gradient ascent on the ℓ_2 norm of these activations. During this process $\mathcal{N}$ is kept fixed while the input image is gradually transformed to yield high output responses for certain classes. The features of the classes depicted in the output image depend on $\hat{x}$ and on the visual knowledge stored in $\mathcal{N}$. For example - if part of

$\hat{x}$ resembles something that $\mathcal{N}$ interprets even slightly as a dog, then Deep Dream will emphasize and enhance these features to look more and more like a dog.

Apart from the above method, which is class neutral, another method is mentioned in [1] which allows enhancing an input image in a way which would elicit a **particular interpretation**. Starting from an input image $\hat{x}$ full of random noise, a target class y and a pretrained classification network $\mathcal{N}$, we forward $\hat{x}$ through $\mathcal{N}$ and then gradually tweak the image towards what $\mathcal{N}$ considers as y. This optimization process can be expressed as solving an optimization problem of the form:

$$\min_{\hat{x}} \mathcal{L}(\hat{x}, y) + \mathcal{R}(\hat{x})$$

where $\mathcal{L}(\cdot)$ is a classification loss (e.g. standard softmax cross-entropy), and $\mathcal{R}(\cdot)$ is an image regularization term to improve $\hat{x}$'s visual quality.

In order to steer the generated images away from unrealistic images that are classified as y but posses no discernible visual information, Deep Dream proposes to add an image prior term $\mathcal{R}_{\text{prior}}$ which penalizes the total variation and ℓ_2 norm of $\hat{x}$ during the generation process. This enforces the output image to have similar statistics to natural images, for example by making neighboring pixels more correlated.

2.2 DeepInversion

Following in Deep Dreams's footsteps, DeepInversion [10] (DI) is an improved method for reconstructing class-conditional images from a pretrained classification model.

Despite the contribution of the regularization term proposed by Deep Dream, images generated by this method still lack an overlap in their distribution with natural (or original training) images, and thus lead to unsatisfactory results for uses such as knowledge distillation.

The creators of DI propose to solve this problem by introducing a new term to the above optimization objective: a feature distribution regularization term called $\mathcal{R}_{\text{feature}}$. By relying on the hidden data stored within the batch normalization [4] layers of the pretrained model, this term constrains the synthesized images to the same distribution as the model's original training images. This greatly improves the quality of the reconstructed images.

Apart from improving the visual quality of the reconstructed images, Yin et al. also manage to improve the diversity of the synthesized images by introducing Adaptive DeepInversion (ADI). This improved method adds an additional 'student' model to the original pretrained 'teacher' model, and then tries to maximize the Jensen-Shannon divergence between the logits of the two models to enforce exploring new synthesis directions. However, since the diversity of the generated images is not an important factor in our project we have decided not to include the improvement introduced by ADI and to use DI exclusively.

2.3 Hidden Networks

Traditionally, deep learning involves initializing a network with random weights, and then learning and optimizing the weights to achieve good performance for some given task. In their paper, Ramanujan et al. [7] propose an interesting take on learning. They show that large networks which are initialized randomly using a properly scaled distribution (e.g. Kaiming

[3]) often contain smaller subnetworks that perform quite well. Not only that, but they also suggest the edge-popup algorithm - an algorithm that attempts to find such subnetworks by assigning scores to the edges (weights) of the network, and choosing the edges with the best scores in each layer. The algorithm learns the scores without ever modifying the weights of the network. In their experiments, they were able to find a subnetwork of a randomly initialized Wide ResNet-50 [11] network that is only 30% of its size, and performs as well as a trained ResNet-34.

3 Methods

3.1 DeepInversion

Recall that the purpose of DeepInversion is to 'invert' images from a pretrained CNN. Given a randomly initialized input $\hat{x} \in \mathbb{R}^{H \times W \times C}$ (H, W, C being the height, width, and number of color channels), and an arbitrary target label y, the image synthesis process can be expressed as solving the following optimization problem:

$$\min_{\hat{x}} \mathcal{L}(\hat{x}, y) + \mathcal{R}(\hat{x}) \tag{1}$$

where $\mathcal{L}(\cdot)$ is a classification loss such as the standard softmax cross-entropy, and $\mathcal{R}(\cdot)$ is an image regularization term to improve $\hat{x}$'s visual quality.

3.1.1 Regularization Terms

Prior Regularization

Deep Dream [1] suggests the following regularization term:

$$\mathcal{R}_{\text{prior}}(\hat{x}) = \alpha_{\text{tv}} \mathcal{R}_{\text{TV}}(\hat{x}) + \alpha_{\ell_2} \mathcal{R}_{\ell_2}(\hat{x})$$

The first term, $\mathcal{R}_{\text{TV}}$, penalizes the total variation of $\hat{x}$, thus promoting correlation between nearby pixels. The term is defined as follows:

$$\mathcal{R}_{\text{TV}}(\hat{x}) = \|\hat{x} - \hat{x}_H\|_2 + \|\hat{x} - \hat{x}_V\|_2 + \|\hat{x} - \hat{x}_{D_1}\|_2 + \|\hat{x} - \hat{x}_{D_2}\|_2$$

where $\hat{x}_H, \hat{x}_V, \hat{x}_{D_1}, \hat{x}_{D_2}$ are horizontal, vertical and diagonal shifted variants of $\hat{x}$, all by a single pixel (note that $\hat{x}_{D_1}$ and $\hat{x}_{D_2}$ represent two different diagonal shifts).
The second term, $\mathcal{R}_{\ell_2}$, is a simpler term which penalizes the ℓ_2 norm of $\hat{x}$, and is defined as:

$$\mathcal{R}_{\ell_2}(\hat{x}) = \|\hat{x}\|_2$$

In [6] it is claimed that this term encourages the range of the values in the image to stay within a target interval instead of diverging.

Note that the magnitude of the above two regularization terms (as they are described in the paper) grows linearly with the batch size, which makes it impossible to run hyper-parameter tuning algorithms with different batch sizes. Thus, in order to make these terms batch-size independent we decided to normalize them by dividing them by the batch size.

Lastly, $\alpha_{\text{tv}}, \alpha_{\ell_2}$ are the scaling factors of the above regularization terms.

Feature Regularization

The main advancement proposed by DeepInversion (when compared with Deep Dream) is the introduction of a new feature regularization term denoted as $\mathcal{R}_{\text{feature}}$.

Yin et al. claim that the regularization terms proposed by Deep Dream provide little guidance for manipulating $\hat{x}$ to contain both low and high level features that are similar to real training images. To effectively enforce feature similarities at all levels, the authors of DI propose to minimize the distance between the feature map statistics of $\hat{x}$ and that of the original dataset $\mathcal{X}$ on which the pretrained model $\mathcal{N}$ was trained.

Assuming that the feature statistics follow a Gaussian distribution across batches ($\sim \mathcal{N}(\mu, \sigma^2)$), $\mathcal{R}_{\text{feature}}$ is defined as:

$$\mathcal{R}_{\text{feature}}(\hat{x}) = \sum_l \left\| \mu_l\left(\hat{x}\right) - \mathbb{E}(\mu_l(x)|\mathcal{X}) \right\|_2 + \sum_l \left\| \sigma_l^2\left(\hat{x}\right) - \mathbb{E}(\sigma_l^2(x)|\mathcal{X}) \right\|_2$$

where $\mu_l\left(\hat{x}\right)$ and $\sigma_l^2\left(\hat{x}\right)$ are the batch-wise mean and variance estimates of feature maps corresponding to the l-th convolutional layer of $\mathcal{N}$.

Obtaining $\mathbb{E}(\mu_l(x)|\mathcal{X})$ and $\mathbb{E}(\sigma_l^2(x)|\mathcal{X})$ might seem problematic since the original dataset $\mathcal{X}$ might not be available. Instead, the authors of DI suggest an intriguing solution: Using the running average statistics stored in the batchnorm layers [4] of the pretrained model. These layers implicitly capture the channel-wise means and variances during training, which allows the estimation of the above expectations by:

$$\mathbb{E}(\mu_l(x)|\mathcal{X}) \simeq BN_l(running\_mean)$$

$$\mathbb{E}(\sigma_l^2(x)|\mathcal{X}) \simeq BN_l(running\_variance)$$

Since the introduction of Batch Normalization in [4] BN layers have become widely adopted and can be found today in almost every well performing classification model. This makes it possible to use DeepInversion with almost any state of the art classification model which exists today.

Together with $\mathcal{R}_{\text{feature}}$, the entire regularization term $\mathcal{R}\left(\cdot\right)$ from equation (1) can be expressed as:

$$\mathcal{R}_{DI}(\hat{x}) = \mathcal{R}_{\text{prior}}(\hat{x}) + \alpha_f \mathcal{R}_{\text{feature}}(\hat{x})$$

where α_f is a scaling factor for $\mathcal{R}_{\text{feature}}$.

3.1.2 Image Refinement Techniques

DeepInversion incorporates two techniques that were first used in Deep Dream to improve the quality of the reconstructed images:

Image Clipping

Before each forward pass the synthesized images are clipped so that they would conform to the mean and variance of the original training set $\mathcal{X}$. Maintaining the correct pixel range during training encourages the synthesis process to produce valid images (whose pixels must all be in the legal range).

For a given synthesized image $\hat{x}$ during training, the clipping process is defined as:

$$\hat{x} = \min(\max(\hat{x}, -m/s), (1-m)/s)$$

Where m and s are the channel-wise RGB mean and standard deviation of $\mathcal{X}$.

Our empirical testing shows that neglecting to clip the images before each forward pass during the training process results in many dark/bright zones in the produced images. This happens since without repeated clipping many of the pixels diverge during training outside of the legal range, and thus must be clipped at the end of the reconstruction process before the resulting images can be converted to PNG format. Our hypothesis is that repeated clipping during the training process encourages these pixels over and over again to converge to legal values within the legal range.

Random Jitter

Before each forward pass each synthesized image is temporarily offset by a random jitter of up to j pixels (where j is a hyper-parameter). Our hypothesis is that these shifts introduce randomness into the gradient descent algorithm, which slows down convergence but also helps produce "smoother" images.

Random Flipping

Before each forward pass each synthesized image is bound for a temporary horizontal flip at a probability of p_f (where p_f is a hyper-parameter). The logic behind these flips is that the pretrained network $\mathcal{N}$ should be able to detect objects in real images regardless of whether they are flipped or not. Thus, flipping the synthesized images between iterations helps instill this property in the synthesized images as well with the intention of improving their quality.

3.1.3 Accelerated Reconstruction Techniques

The authors of DI also recommend the following two methods to speed up the image reconstruction process:

Multi-Resolution Synthesis

The authors of DI found out that they can speed up the synthesis process by employing a multi-resolution optimization scheme.

For 224×224 ImageNet-sized images for example, the scheme begins by first optimizing an input of 112×112 images for a few thousand iterations, then the images are upsampled to 224×224 via nearest-neighbor interpolation, and then they are optimized for a few extra thousand iterations. Overall this whole process requires significantly less iterations compared

with the original method, and most of these iterations are also faster since they optimize much smaller images.

Automatic Mixed Precision (AMP)

To speed up the training process and reduce GPU memory consumption the authors of DI recommend training using half-precision floating point (FP16) via the NVIDIA Apex library[1].

3.2 Hidden Networks

The method for finding well performing subnetworks which are hidden in randomly initialized subnetworks is based on the work of Zhou et al. [12]. In their algorithm, they essentially replace each weight w of the network with $\tilde{w} = wX$, where X is a Bernoulli (p) random variable, i.e. $X = \begin{cases} 1 & \text{with probability } p \\ 0 & \text{with probability } 1-p \end{cases}$. The value of p for each weight is then learned using SGD. These random variables can be thought of as binary masks for the weights, and they effectively define the subnetwork. Masks corresponding to subnetworks that perform better than the original untrained network are referred to as "Supermasks".

Ramanujan et al. [7] build on this idea, but claim that the stochasticity of the method proposed by Zhou et al. limits the performance of the final subnetworks. Instead, Ramanujan et al. suggest assigning a popup score s to each weight w in the network, and then choosing the mask such that only the weights with the top-k% scores of each layer are selected (k is some predefined value, and can be different for each layer).

But how would the score be chosen? First, the scores are initialized randomly using a Kaiming uniform distribution [3]. Then, suppose we are looking at weight w_{uv}, which connects node u to node v in the network. The idea is to perform backpropagation on the loss function, look at the gradients of v's input and the weighted output of node u, and increase s_{uv} if the negative gradient of v's input is aligned with u's weighted output. Essentially, this means we check how the loss "wants" v's input to change, and increase the score of w_{uv} if u's weighted output takes it there. This translates into the following update rule, which can be performed using SGD

$$s_{uv} \leftarrow s_{uv} - \alpha \frac{\partial \mathcal{L}}{\partial \mathcal{I}_v} \mathcal{Z}_u w_{uv}$$

where:

- α is the learning rate
- $\mathcal{L}$ is the loss function
- $\mathcal{I}_v$ is the input of node v
- $\mathcal{Z}_u$ is the output of node u
  - $\mathcal{Z}_u = \sigma(\mathcal{I}_u)$, where σ is some activation function (e.g. ReLU)

[1] https://github.com/NVIDIA/apex

There is one important caveat here. The loss function is calculated on the output of a subnetwork of the original network. This can be done by using a mask similar to that used by Zhou et al., except here instead of a Bernoulli (p) random variable, the mask is deterministic and denoted by $h(s_{uv})$. i.e., the input $\mathcal{I}_v$ of layer ℓ is calculated as

$$\mathcal{I}_v = \sum_{u \in \mathcal{V}^{(\ell-1)}} w_{uv} \mathcal{Z}_u h(s_{uv})$$

where $\mathcal{V}^{(\ell-1)}$ is the set of neurons on the previous layer. The masking function h takes the value of 1 if s_{uv} is in the top-$k\%$ scores of layer ℓ, and 0 otherwise.

This poses a new problem, because the gradient of h is always 0. To deal with this, the straight-through gradient estimator [2] was proposed. This estimator simply skips "straight through" h in the backwards pass, by ignoring its gradient and treating it as an identity operator.

4 Implementation

4.1 DeepInversion

First, we would like to emphasize that while we were working on this project, no implementation of DeepInversion was publicly available, be it official or otherwise. Thus, even though our implementation tries to follow the description in the paper as closely as possible, we borrowed some details from Deep Dream and used our own judgment whenever something was vaguely described.

First, we created a class DeepInvert that holds the relevant information for the DeepInversion environment. That is:

- The model
- The loss function
- The regularization function
- Mean and variance of the dataset the model was originally trained on, used for normalizing and clipping the input before passing it through the model.

The main method of our DeepInvert class is deepInvert. This method applies the DeepInversion algorithm. It receives:

- A batch of images (usually random Gaussian noise)
- Number of iterations to apply DeepInversion
- Target labels to synthesize
- Hyper-parameters for the Adam optimizer
- Amount of jitter to use (defaults to 0)
- Probability of random flipping

deepInvert runs the following algorithm:

Algorithm 1 deepInvert

```
# Normalize the input using the mean and
# variance of the original training dataset
input = preprocess(batch)
# Create optimizer
optimizer = Adam(params=[input],
                 optimizer_hyperparameters)
for i = 1 ... iterations:

    apply_jitter_and_flip(input, jitter, flip)
    output = model.forward(input)
    optimizer.zero_grad()
    loss = loss_fn(output, target_labels) + reg_fn(input)
    loss.backward()
    optimizer.step()
    clip(input) # clips values out of range
    unjitter_and_unflip(input, jitter, flip)

# Denormalize, clip, and convert to PIL images
return postprocess(input)
```

Basically - deepInvert initializes an optimizer to optimize the batch, performs the main loop for the specified number of iterations, and lastly saves the generated images. The main loop is similar to a regular training loop - forward pass, loss calculation, backward pass, and optimization step. The main differences are:

- We apply jitter before each iteration. As described in the paper, this means rolling the images horizontally and vertically by a random amount.
- The regularization function is applied on the input, rather than on the model's output. This is because generating images requires optimizing the input itself.
- We clip the input on each iteration. As we mentioned earlier this method was originally used in Google's implementation of Deep Dream[2].

Our implementation of course supports the use of CUDA, as well as the use of Automatic Mixed Precision as described in the original DeepInversion paper.

4.1.1 Feature Regularization

Perhaps the most important part of DeepInversion, and the major contribution of the paper, is the $\mathcal{R}_{\text{feature}}$ regularization term.

At first, we had some trouble implementing it, because it requires collecting the output of the hidden layers of the network. After some research, we concluded that using *hooks* will help implementing this. Our implementation of $\mathcal{R}_{\text{feature}}$ registers a hook function to all of

[2] https://github.com/google/deepdream

the BatchNorm layers. This way, the hook function is called during the forward pass, i.e., every time before a BatchNorm layer's forward is invoked.[3]

Initially, our hook function computed the batch-wise mean and variance of feature maps corresponding to the current layer, $\mu_l(\hat{x}), \sigma_l^2(\hat{x})$, and stored them in a list. This list was later used for calculating the regularization term as it appeared in the paper [10]:

$$\begin{aligned}\mathcal{R}_{\text{feature}} = &\sum_l \|\mu_l(\hat{x}) - \text{BN}_l(\text{running\_mean})\|_2 \\ &+ \sum_l \left\|\sigma_l^2(\hat{x}) - \text{BN}_l(\text{running\_variance})\right\|_2\end{aligned}$$

However, we noticed this was causing high GPU memory usages. Due to the additive nature of $\mathcal{R}_{\text{feature}}$, we were able to change our implementation to compute $\mathcal{R}_{\text{feature}}$ on the fly. Therefore, our hook function performed the following update

$$\begin{aligned}\mathcal{R}_{\text{feature}} = &\mathcal{R}_{\text{feature}} \\ &+ \|\mu_l(\hat{x}) - \text{BN}_l(\text{running\_mean})\|_2 \\ &+ \left\|\sigma_l^2(\hat{x}) - \text{BN}_l(\text{running\_variance})\right\|_2\end{aligned}$$

And of course, $\mathcal{R}_{\text{feature}}$ was initialized to 0, and was also reset after each time the forward function of the regularization was invoked.

4.1.2 Argument Parser

To allow easy execution of experiments, we implemented an argument parser that allows running different configurations of DeepInversion both programmatically and through the command line. The parser is rich in options, as can be seen by its help message:

```
usage: main.py [-h] [--iterations I] [--early-stopping ES] [--batch-size B]
               [--no-cuda] [--amp-mode AMP] [--seed S] [--lr LR]
               [--scheduler-patience SP] [--targets T [T ...]] [--dataset DS]
               [--model-name MN] [--jitter J] [--flip FLP] [--loss-fn LF]
               [--temp TMP] [--reg-fn RF] [--a-tv ATV] [--a-l2 AL2] [--a-f AF]
               [--output-dir OD]

DeepInversion

optional arguments:
  -h, --help            show this help message and exit
  --iterations I        number of epochs to run DeepInversion for (default:
                        20000)
  --early-stopping ES   percentage of iterations with no improvement to wait
                        before early stopping (default: 15)
  --batch-size B        number of images to generate in a batch (default: 128)
  --no-cuda             disable CUDA
  --amp-mode AMP        Automatic Mixed Precision mode (default: O2)
  --seed S              random seed (default: None)
  --lr LR               learning rate (default: 0.2)
  --scheduler-patience SP
                        learning rate scheduler patience in percentage
                        relative to the number of iterations (default: 5)
```

[3] https://pytorch.org/docs/stable/nn.html#torch.nn.Module.register_forward_pre_hook

```
--targets T [T ...]    target classes for image synthesis, or -1 for
                       randomization (default: -1)
--dataset DS           dataset to perform synthesis on (default: ImageNet)
--model-name MN        name of model to use for synthesis (default: ResNet50)
--jitter J             amount of jitter to apply on each iteration (default:
                       30)
--flip FLP             horizontal flip probability (default: 0.5)
--loss-fn LF           loss function (default: CE)
--temp TMP             temperature value for CrossEntropyLoss (default: 1)
--reg-fn RF            regularization function (default: DI)
--a-tv ATV             TV regularization factor (default: 8e-3)
--a-l2 AL2             l2-norm regularization factor (default: 1e-5)
--a-f AF               feature regularization factor (default: 1e-2)
--output-dir OD        directory for storing generated images (default:
                       generated)
```

4.2 Hidden Subnetworks

For the Hidden Subnetworks, we used the official PyTorch implementation given by the authors of the paper[4]. Their implementation also includes "pretrained" subnetworks, i.e., well performing subnetworks of randomly initialized networks which were found using their method.

4.3 DeepInversion of Hidden Subnetworks

Applying DeepInversion on a Hidden Subnetwork proved to be a bit challenging. This is because the implementation of Hidden Subnetworks was designed to work solely using the command line, and heavily relied on global variables. Thus, trying to extract a pretrained subnetwork from the original implementation required us to use hacky methods, such as modifying their global variables within our code, and fooling their argument parser to receive arguments programmatically (instead of through the command line).

Once we managed to extract a pretrained subnetwork, applying DeepInversion on it was pretty straight forward. We just set the model used by DeepInversion to be the subnetwork, and then executed it as usual. The modularity of our implementation is what allowed this simplicity.

5 Experiments & Results

5.1 Hyper-Parameter Tuning for DeepInversion

After completing the implementation of DeepInversion we moved on to test the method with the hyper-parameters mentioned in [10]. Yin et al. conducted several experiments to test the method's ability to reconstruct images from models that were pretrained on the CIFAR-10 and ImageNet datasets. For ImageNet, the authors claim that they were able to synthesize quality images using the publicly available ResNet-50 model from PyTorch. They used an Adam optimizer [5] with a learning-rate of 0.05, and the following hyper-parameters:

[4] https://github.com/allenai/hidden-networks

$$\alpha_{\text{feature}} = 1 \cdot 10^{-2}, \alpha_{\text{tv}} = 1 \cdot 10^{-4}, \alpha_{\ell_2} = 1 \cdot 10^{-2}$$

Unfortunately we weren't able to reproduce the paper's results using the above hyper-parameters.

These results indicate that DeepInversion is very sensitive to the hyper-parameters being used, and that even the slightest change in the way the method is implemented might affect these parameters. This unfortunately made us realize that we had to tune our implementation by ourselves, which eventually proved to be a very difficult and time-demanding task.

To do so we implemented a grid-search algorithm which tests the method with different combinations of hyper-parameter values (for the parameters not described above we used the values that were mentioned in the paper). Then, to evaluate the synthesized images from each combination we used the Inception Score [8], a popular method to evaluate the image outputs of GANs, which was empirically found to correlate well with human evaluation.

After running the grid-search algorithm for several days and testing a decent number of combinations the best parameters found for our implementation were the following:

$$\alpha_{\text{feature}} = 1 \cdot 10^{-2}, \alpha_{\text{tv}} = 8 \cdot 10^{-3}, \alpha_{\ell_2} = 1 \cdot 10^{-5}$$

It should be noted, however, that the computational limitations imposed on us by the faculty's server (a single shared NVIDIA Titan X 12GB) did not allow us to match the batch size that was used in the experiments in the paper, or to run the grid-search algorithm as extensively as we would have liked.

While the authors of DeepInversion used a batch size of $1,216$ using 8 NVIDIA V100 GPUs, our GPU's memory constraints limited us to test the method with a batch size of only 60-80 (or slightly more using AMP). We hypothesize that using a larger batch size might lead to better results, since a bigger batch would be able to better fit to the distribution imposed by $\mathcal{R}_{\text{feature}}$ during synthesis.

Also, the results from our grid-search experiment empirically showed that the hyper-parameter to Inception Score function we were trying to optimize was highly non-convex. This, along with the fact that our model is highly sensitive to hyper-parameters, makes us believe that a significantly better set of hyper-parameters can be found for our implementation. However, that would require running the tuning algorithm on a grid with a much finer parameter resolution, and that could unfortunately take weeks using our limited hardware.

Below we present some of the images synthesized using our implementation with a ResNet-50 model pretrained on ImageNet:

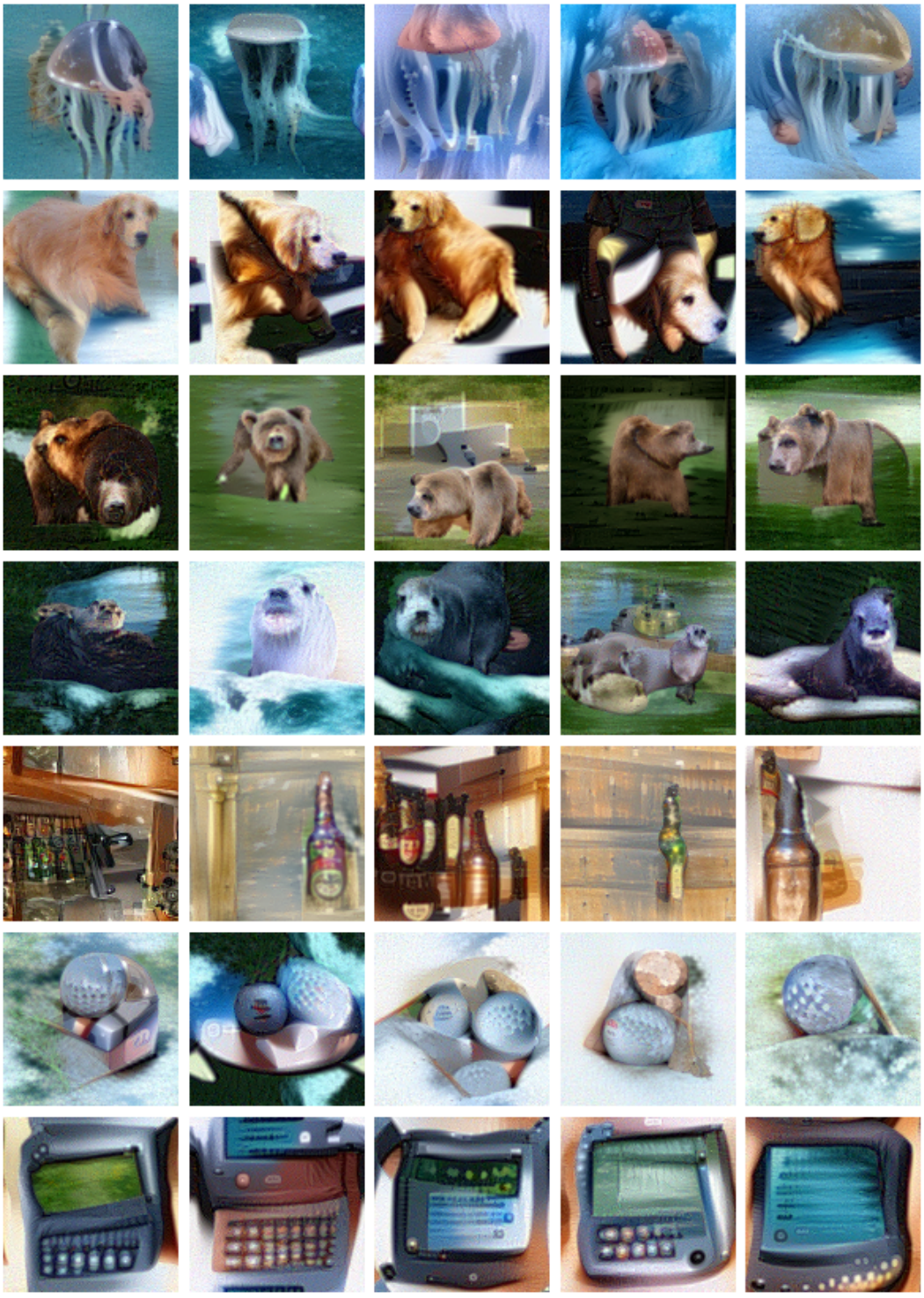

Figure 1: Samples from applying DeepInversion on ResNet-50 pretrained on ImageNet. Classes top to bottom: jellyfish, golden retriever, brown bear, otter, beer bottle, golf ball, hand-held computer.

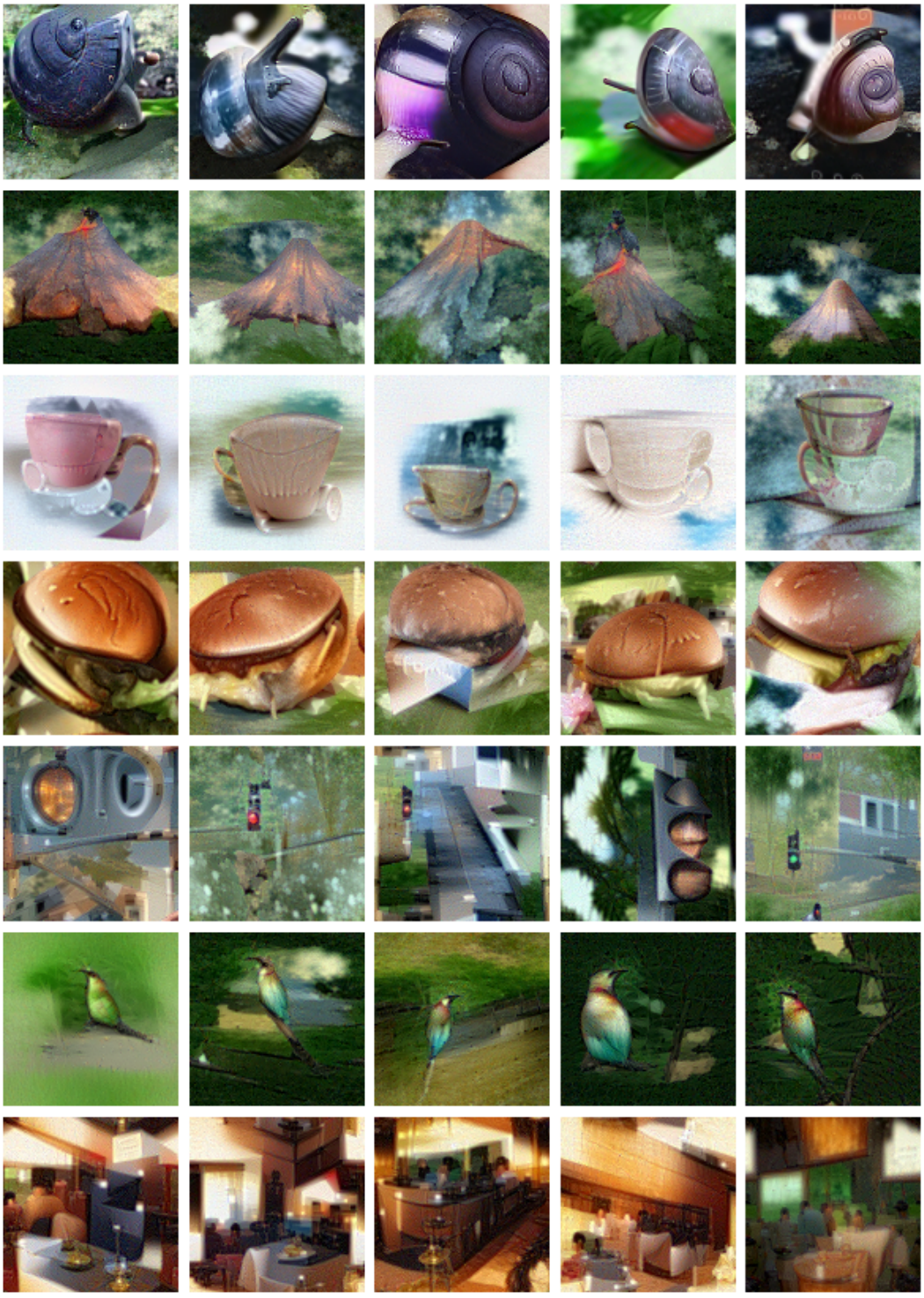

Figure 2: Samples from applying DeepInversion on ResNet-50 pretrained on ImageNet. Classes top to bottom: snail, volcano, cup, cheeseburger, traffic light, bee eater, restaurant.

We would like to emphasize, however, that not all of the images synthesized using our implementation with the above hyper-parameters turned out to be of high quality. We state again that further tuning is still required to obtain stable high-quality batches.

5.2 Evaluation of the Synthesized Images

In this section we propose a way to evaluate the quality of the images synthesized using our implementation of DeepInversion.

We hand-picked 10 categories from ImageNet, and for each category we synthesized 500 images using our implementation with a pretrained ResNet-50.

Then, we used a pretrained ResNet-152 (which has a 78.31%/94.06% top-1/top-5 accuracy respectively on ImageNet) to classify the synthesized images.

The model achieved 97.02%/99.32% top-1/top-5 accuracy respectively on our synthesized dataset.

5.3 Knowledge Distillation Using the Synthesized Images

One of the most intriguing utilizations of DeepInversion is knowledge distillation. Yin et al. claim in [10] that the method enables transferring knowledge from a pretrained teacher to a randomly initialized student CNN without needing the original dataset the teacher was trained on.

To test the knowledge distillation potential of our synthesized images, we used the dataset we synthesized in section 5.2 (5,000 images for 10 hand-picked classes) to train a randomly initialized 'student' ResNet-50 from scratch. For this experiment we used the PyTorch implementation of ResNet-50, but since the model was originally designed for ImageNet, we had to replace its fully connected layer with a layer which fits our smaller dataset.

After training we evaluated the trained model using a test set of 500 images (50 images per class) originated from the validation set of ImageNet. Note that the pretrained model we used to synthesize our dataset with was not trained on these images.

Unfortunately, as we stated in section 5.1 our hardware limitations made it very difficult for us to tune our DI model, and thus we suspected that the quality of the images in our synthesized dataset would not be high enough for knowledge distillation purposes.

Testing the above student model on the test set validated our suspicions, as it was only able to achieve 38.8%/62.4% top-1/top-3 accuracy respectively on the set.

Still, we believe that these results show the potential of our implementation, and we think that they can be much improved by tuning our DI model even further, synthesizing a larger training set and using a larger batch size during synthesis (all three would require better hardware, however).

5.4 DeepInversion of Hidden Subnetworks

After experimenting on our implementation of DeepInversion, we finally got to run it on a well performing subnetwork of a randomly initialized ResNet-50 that was extracted using the edge-popup algorithm [7]. Our results are as follows:

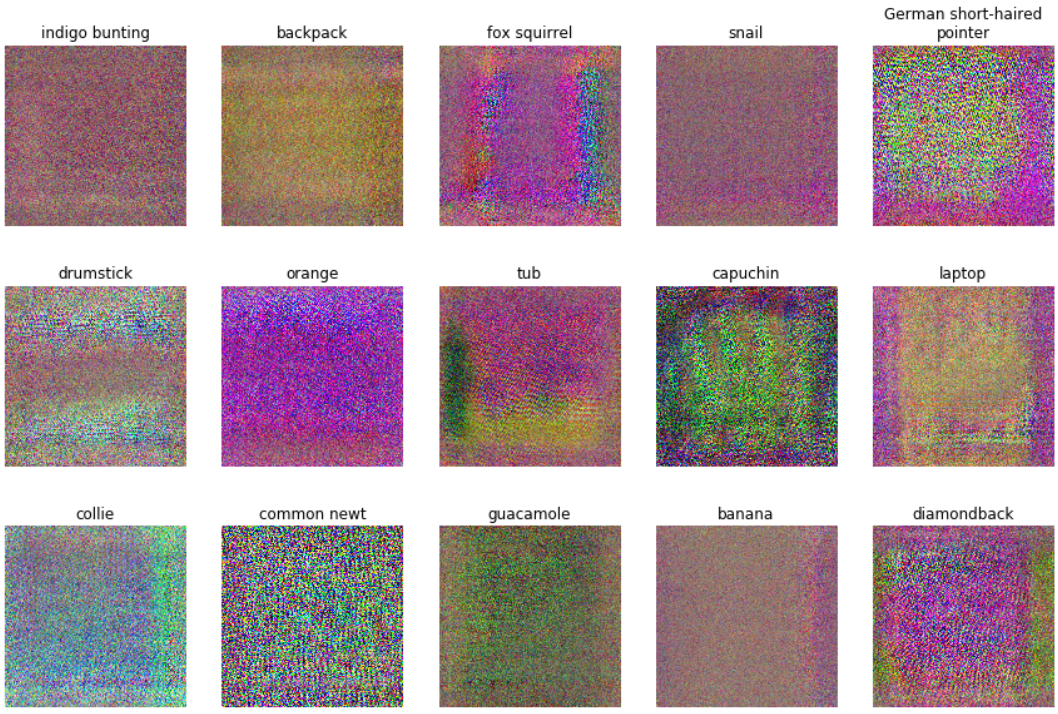

Figure 3: Labeled samples from applying DeepInversion on a Hidden Subnetwork.

As shown above, the results resemble colored noise and are not distinguishable at all. We actually expected this experiment to fail due to one major reason: The hidden subnetworks extracted using the edge-popup algorithm do not contain the BatchNorm statistics mentioned in the Feature Regularization in section 3.1.1. We can see however that the other regularization terms did have some effect on the output, for example, the Total Variation regularization caused nearby pixels to have similar color.

One could then ask why our results do not resemble those of Deep Dream, then? We believe the reason for that is that while the loss function used in Deep Dream is usually the raw output of some layer (or the class scores, in case of the last layer), In DeepInversion on the other hand the loss function is the Cross Entropy loss. In [9] it is claimed that on top of maximizing the probability that the image will belong to a specific label, using the Cross Entropy loss also tries to minimize the probability of it belonging to any of the other labels, which could potentially insert unwanted noise into the image. Furthermore, we believe that using DeepInversion without $\mathcal{R}_{\text{feature}}$ (which is effectively the case here) would require re-tuning the rest of the hyper-parameters in order to maximize the quality of the synthesized images. However, both suggestions (changing the loss function and re-tuning the hyper-parameters) are outside the scope of the project.

6 Conclusion

In this project we have presented an implementation of DeepInversion, a method which allows reconstructing high-quality class-conditional images from a pretrained CNN using the hidden data stored in its batch normalization layers.

We have shown that since Hidden Subnetworks do not contain said hidden data, applying DeepInversion on them results in unsatisfactory results.

Additionally, we believe that our results on DeepInversion demonstrate the potential of the method for purposes such as knowledge distillation, but have also concluded that in order to unlock this full potential one would first have to further tune the model - a very challenging and time demanding task.

References

[1] Mike Tyka Alexander Mordvintsev, Christopher Olah. Google AI Blog: Inceptionism: Going Deeper into Neural Networks, 2015.

[2] Yoshua Bengio, Nicholas Léonard, and Aaron Courville. Estimating or Propagating Gradients Through Stochastic Neurons for Conditional Computation. aug 2013.

[3] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE International Conference on Computer Vision*, volume 2015 Inter, pages 1026–1034, 2015.

[4] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *32nd International Conference on Machine Learning, ICML 2015*, volume 1, pages 448–456. International Machine Learning Society (IMLS), feb 2015.

[5] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. dec 2014.

[6] Aravindh Mahendran and Andrea Vedaldi. Understanding Deep Image Representations by Inverting Them. nov 2014.

[7] Vivek Ramanujan, Mitchell Wortsman, Aniruddha Kembhavi, Ali Farhadi, and Mohammad Rastegari. What's Hidden in a Randomly Weighted Neural Network? 2019.

[8] Tim Salimans, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, and Xi Chen. Improved Techniques for Training GANs. jun 2016.

[9] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Deep inside convolutional networks: Visualising image classification models and saliency maps. In *2nd International Conference on Learning Representations, ICLR 2014 - Workshop Track Proceedings*, 2014.

[10] Hongxu Yin, Pavlo Molchanov, Zhizhong Li, Jose M Alvarez, Arun Mallya, Derek Hoiem, Niraj K Jha, and Jan Kautz. Dreaming to Distill: Data-free Knowledge Transfer via DeepInversion. 2019.

[11] Sergey Zagoruyko and Nikos Komodakis. Wide Residual Networks. In *British Machine Vision Conference 2016, BMVC 2016*, volume 2016-Septe, pages 87.1–87.12. British Machine Vision Conference, BMVC, may 2016.

[12] Hattie Zhou, Janice Lan, Rosanne Liu, and Jason Yosinski. Deconstructing Lottery Tickets: Zeros, Signs, and the Supermask. may 2019.