PyTorch Implementation of DesnowNet for UCLA Visual Machines Group

Varan Mehra
University of California, Los Angeles
Los Angeles, CA 90095
nimarv@ucla.edu

Abstract

Of the few existing open-source implementations of image-based snow removal, the overwhelming majority appear to utilize the TensorFlow framework. This paper proposes an implementation of DesnowNet: Context-Aware Deep Network for Snow Removal [8] using PyTorch. We integrate open-source implementations of preexisting modules used in the model with original implementations of the novel ideas presented by the authors. We also propose changes, improvements to the methods presented in DesnowNet [8], an accommodations for potential computational infrastructure constraints.

1. Summary of work

We implement the method of snow removal proposed in DesnowNet, which conceptualizes a snowy color image x as a composition of a snow-free image y, a chromatic aberration mask a, and a snow mask z, formulated in equation (1) as:

$$x = a \odot z + y \odot (1 - z) \tag{1}$$

As such, the proposed method aims to generate a snow-free result $\hat{y}$ by estimating a snow-free image $y\prime$ that removes snow from the input image, and residual complement r which recovers the areas of the image obscured by snow particles. This relationship is expressed in equation (2):

$$\hat{y} = y\prime + r \tag{2}$$

1.1. Descriptor

Similar to the method proposed in DesnowNet, we use Inception-v4 [9] as the backbone of our model. While PyTorch's Torchvision package has an implementation of a previous version, Inception-v3, we had to look for an independent implementation of the most recent inception model. To this end, we use an open-source implementation of Inception-v4 [1]. As Liu et al. specify in section 5.1 of their paper [8], we manually changed pooling operations to be 'same' convolutions with size 3×3 and stride 1×1. Furthermore, we remove the linear layer and logit module as our use case of Inception-v4 is not classification.

The difference between the Dilation Pyramid (DP) proposed in DesnowNet [8] and Atrous Spatial Pyramid Pooling (ASPP) proposed in DeepLab [1], is that we apply a concatenation operation instead of summation. While ASPP was first shown in the original DeepLab paper, and referenced in DesnowNet, we elect to use an implementation of DeepLab v3 [2] by PyTorch [2], and modify the source code to suit our needs. More specifically, we choose not to use the pooling and projection operations in PyTorch, as all we need are 2D convolutions, batch normalization, and ReLU activations.

1.2. Recovery Submodule

1.2.1 Pyramid Maxout

For Pyramid Maxout, and Pyramid Sumout, as we will see later, the implementation by Liu et al. [8] assumes that $\beta = 4$. The algorithm itself is rather simple; we apply convolutional layers of size $1+2n$ for all $n \leq \beta$, with $1 \times n$ and $n \times 1$ convolutions instead of 5×5 and 7×7 convolutions individually.

1.2.2 Translucency Recovery

To generate the snow-free estimate $y\prime$ we generally follow the formulation from the DesnowNet paper in equation (5). However, to account for all possible values of $\hat{z}_i$ we slightly alter the equation, as shown below:

$$y_i\prime = \begin{cases} \frac{x_i - a_i \times \hat{z}_i}{1 - \hat{z}_i} & \forall \hat{z}_i < 1 \\ x_i, & \forall \hat{z}_i \geq 1 \end{cases} \tag{3}$$

With the above formulation, we explicitly handle the cases when $\hat{z}_i > 1$.

[1] We adapt this implementation by Remi Cadene

[2] The PyTorch implementation of DeepLabv3 can be found here

In order for dimensions to match up directly and assist with indexing, we concatenate SE, which represents $\hat{z}$, with itself such that $\hat{z} \in [0,1]^{p \times q \times 3}$ and $a \in \mathbb{R}^{p \times q \times 3}$.

Our first implementation utilized a tensor initialized with `NaN`s, but this does not work when using a GPU. Rather than initializing a tensor, we decide to generate $y\prime$ by applying the first case of Eq. (3) for all values of $\hat{z}_i$, and then simply replacing with x_i the values of $\hat{z}_i \geq 1$.

1.2.3 Residual Generation

To generate the residual complement r that recovers obscured parts of the image, we follow the DesnowNet implementation and use Inception-v4 [9] as the backbone, applying it to one of the output f_c of the translucency recovery module.

r is a result of applying pyramid sumout to the resulting output of the Inception-v4 model. To this end we follow the same general structure as pyramid maxout; however, instead of taking the element-wise maximum of the convolutions, we take the element-wise sum to arrive at our residual estimate r.

1.3. Pyramid Loss Function

The overall structure of our implementation of the loss function is similar to the PyTorch implementation of the ASPP operation in DeepLab v3 [2]; we add `nn.MaxPool2d()` modules with kernel size and stride determined by the parameter τ. We then iterate through the list of modules, calculating and returning the total sum of the mean squared errors between the pooling operations on the inputs. A mathematical representation of the this operation, originally formulated by Johnson et al. [6], can be seen below:

$$\mathcal{L}(x, y) = \sum_{i=0}^{\tau} ||P_{2^i}(x) - P_{2^i}(y)||_2^2 \tag{4}$$

Where P_n represents the max-pooling operation of kernel size $n \times n$ and stride $n \times n$. x and y represent images of equal dimensions (in our case, $x, y \in \mathbb{R}^{64 \times 64 \times 3}$).

2. Implementation Details

While Liu et al. [8] specified parameters and other implementation details, they did not provide an exhaustive list of factors to consider. As such, we made our own decisions about areas left unspecified, and tweak other implementation details based on our computational circumstances.

2.1. Parameters

The parameters not clearly discussed in DesnowNet [8] or bounded by the inputs and/or outputs of other modules are parameters of the Inception-v4 module [9], Dilation Pyramid [8], and the pyramid loss function [6].

2.1.1 Inception-v4 Module

While we initially planned to implement the Inception-v4 [9] module with minimal changes to the structure of the implementation we adapted, when we went to train the model in Google Colab (a cloud-based Jupyter notebook environment) , our model used up too much memory of the graphics processing unit (GPU). With local access to a GPU this would not have proven to be a major issue, but given our circumstances we were forced to reduce the size of the model.

With little room to reduce the sizes of other modules, we elected to scale down the size of all convolutional layers by a conservative factor of 4. We choose to do this because it not only allows us to train in the first place, but also speeds up training. A leaner model architecture also has the added benefit of preventing overfitting.

Later on, we also experiment with a larger model that only scales down the size of all convolutional layers by a factor of 2. Results and a comparison between the two models is discussed in Sec. 3.

2.1.2 Dilation Pyramid

While the number of input layers of the Dilation Pyramid submodule are constrained to 384 by the size of the Inception-v4 model discussed above, the number of output layers was not specified. We choose to reduce the number of output layers per pooling operation to 192 as opposed to leaving it at 384, or possibly increasing it. We do this because the Dilation Pyramid concatenates features from each level of the pyramid (unlike Atrous Spatial Pyramid Pooling [1]), and we aim to avoid making the module unnecessarily large in order to maintain important spatial information.

Liu et al. [8] specify a parameter γ, set at 4, to govern the number of levels of the dilation pyramid. Given our chosen output of 192 channels per level of the dilation pyramid, the entire operation results in a total of 960 channels.

2.1.3 Pyramid Loss Function

DesnowNet [8] also does not specify the parameter τ that denotes the level of the loss pyramid. However, we observe that β, which specifies the number of layers of the pyramid maxout and pyramid sumout operations is set to 4. γ as we see in the section above, which defines the levels of the dilation pyramid, is also set to 4.

As such, we select $\tau = 4$ in our implementation for the sake of consistency with the number of levels of other pyramid operations.

2.2. Initialization

Liu et al. [8] note that we initialize the weightings of the model by Xavier Initialization [3], but do not spec-

ify whether to use a uniform or normal distribution. The original paper by Glorot and Bengio suggests initializing weights using a uniform distribution between $-r$ and r where

$$r = \sqrt{\frac{6}{n_{in} + n_{out}}}$$

which ensures that variance is equal to

$$\sigma^2 = \frac{2}{n_{in} + n_{out}}$$

which helps ensure that the variance of the outputs is roughly equal to the variance of the inputs, to avoid the problem of vanishing or exploding gradients.

However, more recently He et al. [4] assert the following:

> Recent deep CNNs are mostly initialized by random weights drawn from Gaussian distributions

As such, we elect to use Xavier-normal initialization. To this end we adapt a preexisting implementation of Xavier-normal weight initialization [3] and apply it to our model.

2.3. Preprocessing

2.3.1 Random Cropping

As in DesnowNet [8], we randomly crop a 64 pixel square from the input image. Simply applying the PyTorch function `RandomCrop()` (from the torchvision.transforms package) takes random patches of the snowy, ground truth, and mask images, but in different locations. Since all 3 images are of the same size, we use a modified version of PyTorch's function. We randomly select the top-left corner of the 64 $\times$ 64 pixel box we wish to crop, and pass it as an argument to our custom function, which symmetrically pads the image if it is smaller than 64 $\times$ 64 pixels, and then returns the cropped patch as specified.

2.3.2 File Indexing

In order to reduce the time it takes to index the names of all of the files in the data set we instead pass a list of the file names as an argument to the dataset function. While our initial implementation simply utilized the `os.listdir()` function, indexing into Google Drive folders from Colab was a slow process that also bottle-necked the training process; so we only had to list all of the files names once, before storing the names in a separate file and loading them when needed with the `dill` package.

However, we came to realize that accessing files from Drive at train and test time was extremely inefficient and led to other errors, so we move the desired dataset into the local runtime environment. This mitigates the need for a premade list of file names, as calling `os.listdir()` at runtime is no longer an expensive operation, but we leave our original implementation as a marginal optimization (while still supporting indexing at runtime).

2.4. Training and Validation

2.4.1 Training

We follow most of the conventions used by Liu et al. [8]; we use a training batch size of 5, use a random 64 $\times$ 64 pixel cropping as the training sample, as well as the Adam Optimizer [7]. We set the parameters of our Adam Optimizer to their defaults, except for the learning rate α, which we set to 3×10^{-5}, and weight decay to $\lambda_w = 5 \times 10^{-4}$ as specified in DesnowNet implementation details [8].

It is at train time that we calculate the overall loss function, using our definition of pyramid loss as discussed previously. The loss function originally defined by Liu et al. [8] includes a term $\lambda_w ||w||_2^2$, which is already accounted for in the weight decay term of our optimizer. Thus, the loss function we work with in the training loop is as follows:

$$\mathcal{L}_{overall} = \mathcal{L}_{y\prime} + \mathcal{L}_{\hat{y}} + \lambda_{\hat{z}} \mathcal{L}_{\hat{z}} \tag{5}$$

where $\lambda_{\hat{z}}$ is specified to be 3

In order to continue training over extended periods of time, we store the dictionaries representing the states of both the model and the optimizer at regular intervals, as well as the training loss and epoch numbers to track our progress.

2.4.2 Validation

When running validation we similarly use batch sizes of 5 and randomly crop corresponding 64 $\times$ 64 patches from the images. The metric of interest for validation is peak signal-to-noise ratio (PSNR), where higher values are more desirable. PSNR can be calculated using mean-squared error (MSE) with the following equation:

$$PSNR = 10 log_{10}(\frac{R^2}{MSE}) \tag{6}$$

R represents the maximum fluctuation in the input image data type, which for our purposes is equal to 1.

Liu et al. [8] note that because of how we calculate $\hat{y}$ in equation (2), there is a possibility that y_i is not in the range [0,1] for some $\hat{y}_i \in \hat{y}$. In order to fix this, we clip the snow-free result when $\hat{y}_i \notin [0, 1]$.

[3] We adopt this implementation by Xiaojian Ma

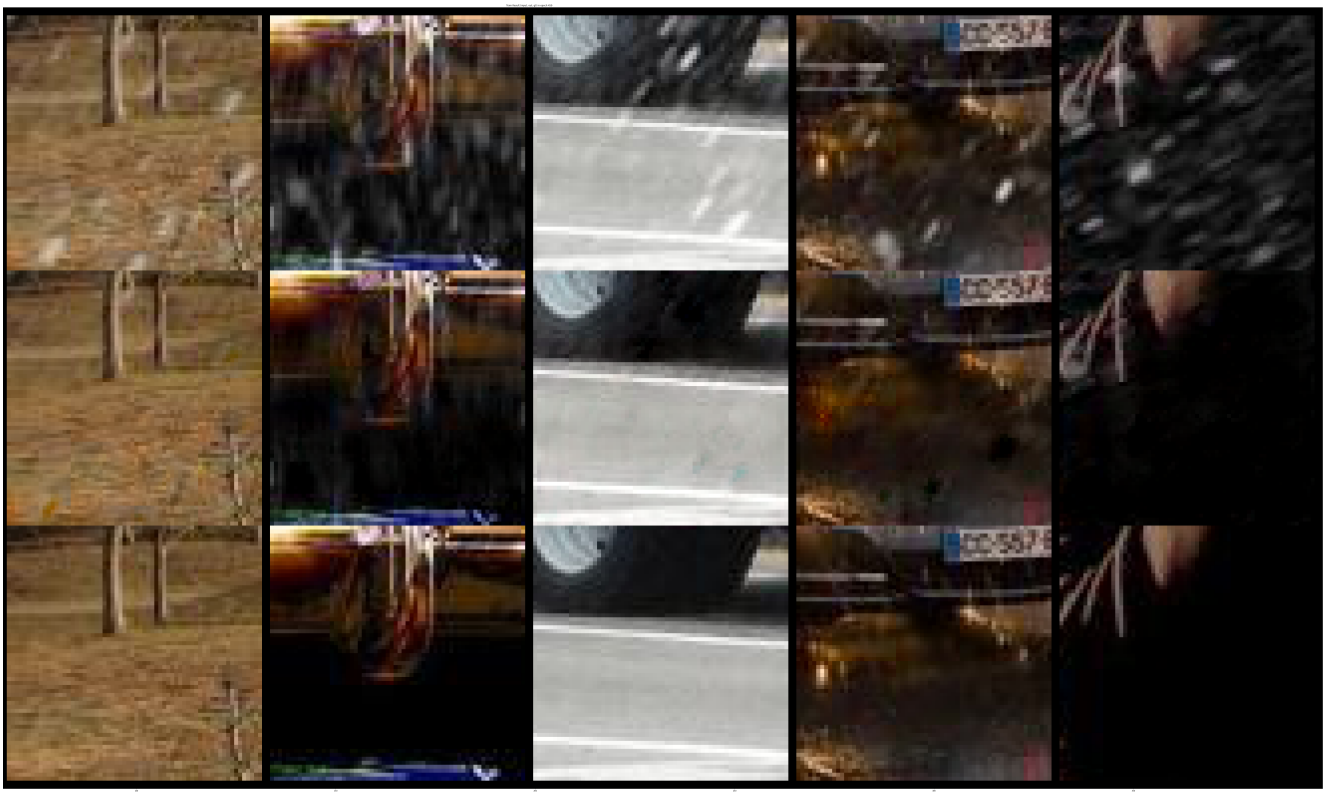

Figure 1. Qualitative results of our model. Snowy input images (top) are propagated through our model yielding $\hat{y}$ (middle), which is then compared to the ground-truth image (bottom).

Test Set	PSNR
Snow100K-S	25.5611
Snow100K-M	28.9391
Snow100K-L	29.7496
Overall	28.0892

Table 1. Validation results of our model on Snow100K test sets.

3. Results

As shown in Tab. 1, our model achieves a PSNR of 28.0892 over the entire Snow100K test set. While the original DesnowNet paper [8] achieved moderately better results than our own implementation, we expect our model to see better performance with more time spent training.

Our qualitative results in Fig. 1 also indicate good performance. Overall our model is particularly effective at removing smaller snow particles, as well as recovering the colors behind them. Furthermore, we observe that the model is resistant to removing snow-colored objects present in the ground truth snow-free image.

As we mentioned in Sec. 2.1.1, later into the process we began training a second model and sought to compare its performance to our original implementation. We observe in Fig. 2 that while our original model performs better, our secondary model appears to converge faster. Given more time, we would expect the larger model to outperform its smaller counterpart, as a larger architecture allows the model to learn a more complex function.

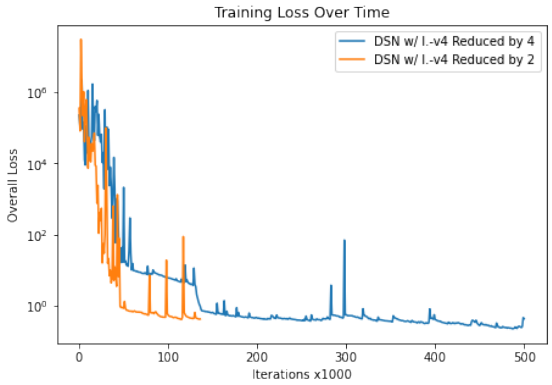

Figure 2. Losses of our original implementation (blue) and larger secondary model (orange) over 50 and 13 epochs respectively

Figure 3. Failure cases. Our model attempts to remove snow from the input (top), but the output (middle) differs significantly from the ground truth (bottom).

3.1. Failure Cases

While our model most effectively removes and recovers the areas behind smaller snow particles, the results in Fig. 3 show cases in which the model does not perform as well. When presented with objects that look particularly snow-like, such as small dots on an item of clothing, our model treats these objects as snow particles to be removed and recovered. The model also struggles to effectively recover the areas behind larger opaque snow particles; the residual complement generally appears to be the correct color, but often a darker shade than the ground truth image. This issue in particular is one that has seen improvement as training progressed, and we expect that this trend would continue given more time.

4. Suggested Modifications

While DesnowNet is an effective model for snow removal, there is certainly room for improvement.

The train-test split of the Snow100K dataset was roughly even. Having more training data is always valuable, and as long as we have a sufficient amount of testing data for validation, we should aim to use as much data as possible for training. Given the size of the dataset, we may want to use at least 80% of the data for training, perhaps even more.

Liu et al. [8] chose to use Inception-v4 [9] as the backbone of the model because of its optimized features at multi-scale receptive fields. While it has shown to be an effective choice, we recommend using a deep residual network instead [5]. Skip connections are effective in combating vanishing or exploding gradients and the accuracy saturation problem. The use of a residual network would allow us to train a deeper and more effective network.

We believe the use of residual modules, a more training-oriented split of the data, and perhaps normalization of inputs, would yield better results.

References

[1] Liang-Chieh Chen, George Papandreou, Iasonas Kokkinos, Kevin Murphy, and Alan L. Yuille. Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs. *IEEE transactions on patter analysis and machine intelligence*, 40(4):834–848, 2018. 1, 2

[2] Liang-Chieh Chen, George Papandreou, Florian Schroff, and Hartwig Adam. Rethinking atrous convolution for semantic image segmentation. *arXiv preprint arXiv:1706.05587*, 2017. 1, 2

[3] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. *Aistats*, 9:249–256, 2010. 2

[4] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015. 3

[5] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016. 5

[6] Justin Johnson, Alexandre Alahi, and Li Fei-Fei. Perceptual losses for real-time style transfer and super-resolution. *European Conference on Computer Vision*, pages 694–711, 2016. 2

[7] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. 2014. 3

[8] Yun-Fu Liu, Da-Wei Jaw, Shih-Chia Huang, and Jenq-Neng Hwang. Desnownet: Context-aware deep network for snow removal. *IEEE Transactions on Image Processing*, 1(1), 2017. 1, 2, 3, 4, 5

[9] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alex Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. *AAAI*, 4:12, 2017. 1, 2, 5