Integral Neural Networks
Supplementary Material

Kirill Solodskikh[*†] Azim Kurbanov[*†] Ruslan Aydarkhanov[†]
Irina Zhelavskaya Yury Parfenov Dehua Song Stamatios Lefkimmiatis

Huawei Noah's Ark Lab

{kirillceo, azimcto, ruslancto}@garch.me

{zhelavskaya.irina1, parfenov.yury, dehua.song, stamatios.lefkimmiatis}@huawei.com

Appendix A. Composite quadratures

Nowadays, a variety of numerical integration quadratures have been proposed to compute integrals for functions of a single variable. Any quadrature can be presented as a linear combination of the function values on the given partition $P = (x_0, \ldots, x_n)$, where $0 = x_0 < x_1 < \ldots < x_{m-1} < x_m = 1$:

$$\int_0^1 f(x)dx \approx \sum_{i=0}^{m} q_i f(x_i),$$

To obtain composite quadrature for the case of multivariate function $f(x_1, \ldots, x_n)$, one-dimensional quadrature should be applied iteratively to each iterated integral:

$$g(x_1) = \iiint_{[0,1]^{n-1}} f(x_1, x_2, \ldots, x_n)dx_2 \ldots dx_n,$$

$$\iiint_{[0,1]^n} f(x_1, \ldots, x_n)dx_1 \ldots dx_n = \int_0^1 g(x_1)dx_1 \approx \sum_{i=0}^{m} q_i g(x_{1i}).$$

Then we apply the same steps for each term $g(x_{1i})$, but now it is used for mutivariate functions of $n-1$ variables instead of n. This procedure defines inductive step. The correctness of application of composite quadratures is guaranteed by Fubini's theorem:

Theorem 1 (Fubini Theorem)**.** *Let $f(x_1, \ldots, x_n)$ be the Riemann integrable function on unit cube $[0,1]^n$. Then the following holds:*

$$\iiint_{[0,1]^n} f(x_1, \ldots, x_n)dx_1 \ldots dx_n = \int_0^1 \Big(\cdots \int_0^1 f(x_1, \ldots, x_n)dx_n\Big)dx_1,$$

if each iterated integral exists. Especially, if function $f(x_1, \ldots, x_n)$ is continuous, then any iterated integral exists.

Since we use only smooth and continuous functions, then from Fubini theorem it follows that we can apply composite quadratures for evaluation of the multiple integral.

To perform differentiation through integration we apply the Leibniz theorem:

Theorem 2 (Leibniz integral rule)**.** *Let $f(\lambda, x_1, \ldots, x_n)$ be a smooth function. If $\frac{\partial f(\lambda, x_1, \ldots, x_n)}{\partial \lambda}$ is continuous, then the following holds:*

$$\frac{d}{d\lambda}\iiint_{[0,1]^n} f(\lambda, x_1, \ldots, x_n)dx = \iiint_{[0,1]^n} \frac{\partial f(\lambda, x_1, \ldots, x_n)}{\partial \lambda}dx.$$

Proofs of Fubini and Leibniz theorems can be found in [12]. Combining these theorems we can prove the Neural Integral Lemma.

Proof of Neural Integral Lemma. For simplicity, we will perform calculations for fully connected integral operators. Proof for convolutions can be obtained in the same way.

Let $F_W(\lambda, x^{out}, x^{in})$ be an integral kernel and $F_I(x^{out})$ an input function. As $F_W(\lambda, x^{out}, x^{in})$ and $F_I(x^{out})$ are continuous, their product is also continuous. Therefore, according to Fubini theorem for numerical integration, we can apply composite quadrature (for fully-connected we do not

[*]The authors contributed equally to this work.
[†]Currently affiliated with Garch Lab.

need Fubini theorem because of single dimension) to the function, which is a product of the integral kernel and the input function:

$$\int_0^1 F_W(\lambda, x^{out}, x^{in}) F_I(x^{out}) dx^{in} \approx \sum_{i=0}^{m} q_i F_W(\lambda, x^{out}, x_i^{in}) F_I(x_i^{in}).$$

It is easy to see that we can compose matrix W_{ji} for a vanilla fully-connected layer in the following way:

$$W_{ji} = q_i F_W(\lambda, x_j^{out}, x_i^{in}).$$

This means that weights of an arbitrary quadrature can be fused into the weight matrix of the vanilla fully-connected layer. Now, let us consider the backward pass of the same layer. Applying Leibniz theorem we obtain:

$$\frac{\partial}{\partial\lambda} \int_0^1 F_W(\lambda, x^{out}, x^{in}) F_I(x^{out}) dx^{in} = \int_0^1 \frac{\partial F_W(\lambda, x^{out}, x^{in})}{\partial\lambda} F_I(x^{out}) dx^{in}.$$

Now, the same partition and quadrature are applied for numerical evaluation on backward pass:

$$\int_0^1 \frac{\partial F_W(\lambda, x^{out}, x^{in})}{\partial\lambda} F_I(x^{out}) dx^{in} \approx \sum_{i=0}^{m} q_i \frac{F_W(\lambda, x^{out}, x_i^{in})}{\partial\lambda} F_I(x_i^{in}),$$

where the last sum is equal to the partial derivative with respect to λ of forward pass (A), which completes our proof. □

Appendix B. Quadrature weights

When the continuous representation with integral operators are converted into discrete weight tensors, we need to choose an integration rule and estimate the weights of the integration quadrature q_i (Eq. (A)). We provide further details on the left and right Riemann sums, as well as the trapezoidal rule. By denoting $\Delta x_i = x_i - x_{i-1}$ the weights of integration quadrature q_i are obtained as follows.

For the left Riemann sum:

$$\int_0^1 f(x)dx \approx \sum_{i=0}^{m-1} f(x_i)\Delta x_{i+1}$$

$$\implies q_i = \begin{cases} \Delta x_{i+1}, & \text{for } i \in \{0, \dots, m-1\} \\ 0, & \text{for } i = n \end{cases}$$

For the right Riemann sum:

$$\int_0^1 f(x)dx \approx \sum_{i=1}^{m} f(x_i)\Delta x_i$$

$$\implies q_i = \begin{cases} 0, & \text{for } i = 0 \\ \Delta x_i, & \text{for } i \in \{1, \dots, m\} \end{cases}$$

For the trapezoidal rule:

$$\int_0^1 f(x)dx \approx \sum_{i=1}^{m} \frac{f(x_i) + f(x_{i-1})}{2} \Delta x_i$$

$$= \sum_{i=1}^{m-1} f(x_i) \frac{\Delta x_{i+1} + \Delta x_i}{2} + f(x_0)\frac{\Delta x_1}{2} + f(x_m)\frac{\Delta x_m}{2}$$

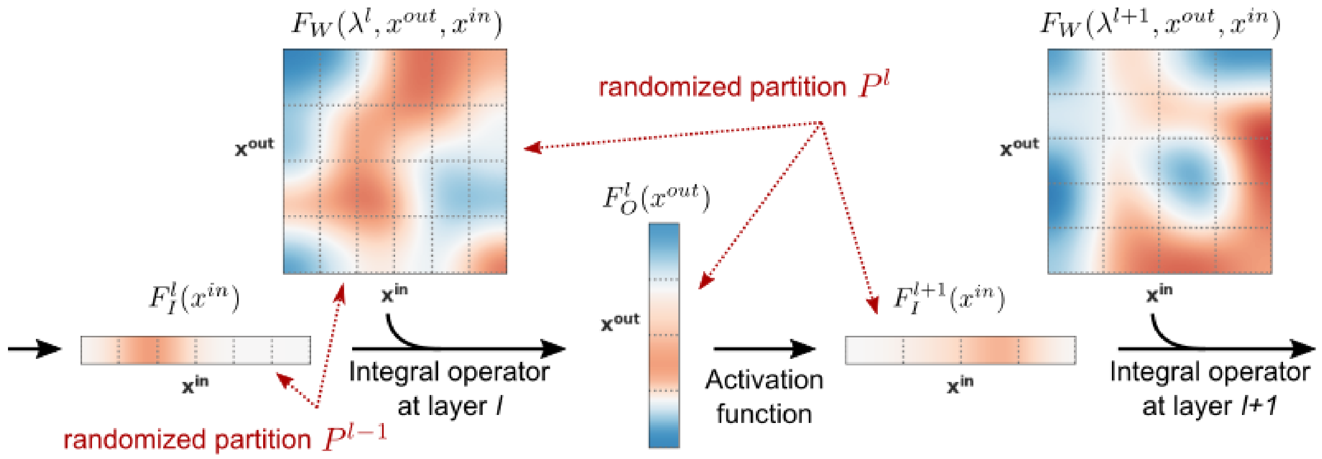
$$\implies q_i = \begin{cases} \frac{\Delta x_1}{2}, & \text{for } i = 0 \\ \frac{\Delta x_{i+1} + \Delta x_i}{2}, & \text{for } i \in \{1, \dots, m-1\} \\ \frac{\Delta x_m}{2}, & \text{for } i = m \end{cases}$$

Appendix C. Training Integral Neural Networks

Conversion of discrete network to INN The algorithm for converting discrete networks to integral neural networks is formalized in Algorithm 1. In this algorithm, we consider layers of the network one by one. We minimize the total variation of the current layer to find the optimal permutation of filters in that layer. We permute the channels of the following layers connected to the current one as defined

Algorithm 1 Conversion of a conventional discrete NN to an integral NN.

```
Require: Discrete network Net.
 1: for each layer $l$ in Layers(Net) do
 2:     //Assume the first dimension is the filter/row
 3:     $D_{ij}$ := TotalVariation($W_l[i], W_l[j]$)
 4:     //Find the optimal permutation
 5:     path := 2opt($D_{ij}$)
 6:     //Permute filters and perform cubic interpolation
 7:     $W_l$ := SmoothInterpolate($W_l$[path])
 8:     if HasBias($l$) then
 9:         //Permute bias and perform cubic interpolation
10:         $b_l$ := SmoothInterpolate($b_l$[path])
11:     end if
12:     for each layer $\hat{l}$ in ChildrenLayers(Net) do
13:         //Permute channels of children layers
14:         $W_{\hat{l}}$[:, :] := $W_{\hat{l}}$[:, path]
15:     end for
16: end for
```



$F_W(\lambda^l, x^{out}, x^{in})$
$F_W(\lambda^{l+1}, x^{out}, x^{in})$
randomized partition P^l
$\mathbf{x^{out}}$
$F_O^l(x^{out})$
$F_I^l(x^{in})$
$\mathbf{x^{in}}$
$F_I^{l+1}(x^{in})$
Integral operator at layer l
Activation function
Integral operator at layer $l+1$
randomized partition P^{l-1}

Figure 1. Visualization of the training procedure under randomized discretization. The resulting trained network may be used with any number of channels (columns) and filters (rows) due to flexible discretization along x^{out} and x^{in}.

by its optimal permutation. Thus, each permutation is done in such a manner that there is a one-to-one correspondence between permutation of filters and channels across consecutive layers, i.e., the filters permutation in the preceding layer matches the channels permutation in the following layer. If a layer has a residual connection to another layer, then the discretizations of these two layers need to match each other: we combine these two layers in one group and compute a common "distance" (total variation) matrix for them, which is then used to find the optimal permutation. The algorithm ends once it has gone through all layers of the network.

Algorithm 1 does not guarantee total variation minimization of the layer output due to data-free nature, i.e., the algorithm utilizes information only from weights without using any calibration dataset. Anyway, such a minimization improves interpolation of the weights with a smaller partition in a sense of improving integration with the fewer number of points. Total variation can be used as an upper bound of the integration error estimation. We can formulate it as the following proposition.

Proposition 1. *Let $f(x)$ be a smooth function on the segment $[a, b]$. Then the n-th Riemann sum $R_n = \sum_i \delta x_i f(x_i)$ has integration error $E_n = \big| \int_a^b f(x)dx - R_n \big|$ for an arbitrary uniform partition is bounded by total variation $V_a^b(f)$ of function $f(x)$: $E_n \leq \frac{b-a}{n} V_a^b(f)$.*

Proof. Since function $f(x)$ is smooth, then it has minimum and maximum values: $m_f \leq f(x) \leq M_f$. Therefore, for any pair x_i, x_{i+1} we obtain:

$$m_f(x_{i+1} - x_i) \leq \int_{x_i}^{x_{i+1}} f(x)dx \leq M_f(x_{i+1} - x_i).$$

From the last inequality we derive

$$\begin{aligned} &\Big| \int_{x_i}^{x_{i+1}} f(x)dx - f(x_i)(x_{i+1} - x_i) \Big| \\ &\quad \leq (M_f - m_f)(x_{i+1} - x_i) \\ &= \frac{b-a}{n}(M_f - m_f) \leq \frac{b-a}{n} V_{x_i}^{x_{i+1}}, \end{aligned}$$

where the last inequality follows from the definition of the total variation. By summing all sub-intervals, we finalize the proof:

$$E_n \leq \sum_i \Big| \int_{x_i}^{x_{i+1}} f(x)dx - \frac{b-a}{n} f(x_i) \Big| \leq \frac{b-a}{n} V_a^b(f).$$

□

Optimization of continuous weights The training procedure for training with randomized discretization is demonstrated in Fig. 1. We compute the discretization of the integral kernel of each layer l using partition P^l, which corresponds to weights of that layer, and adjust it to the necessary quadrature. We train our networks with partitions P^l of random sizes sampled from a certain range. Discretization of x^{in} of the next layer $l+1$ is defined by discretization of x^{out} of the previous layer l, as shown in the figure.

Appendix D. Experiment details

All training experiments were done using Adam optimizer [4]. For all experiments, except for partition tuning, we have used piecewise constant monotone learning rate scheduling starting from $1e-4$ with the adjustment on a plateau (10 epochs without loss function decrement)

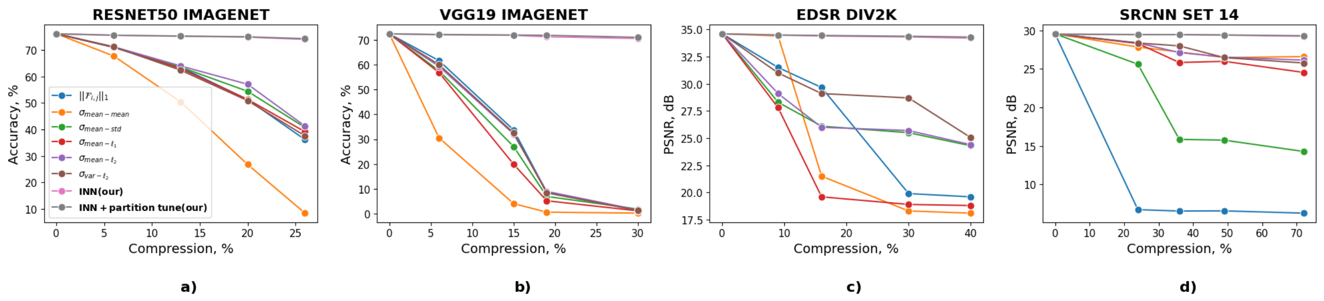


Figure 2. Visualization of different channel selection methods without fine-tuning compared with our integral networks. a) ResNet-50 on Imagenet. b) VGG-19 on Imagenet. c) 2x EDSR on Div2k [1]. d) SRCNN on Set14. [14].

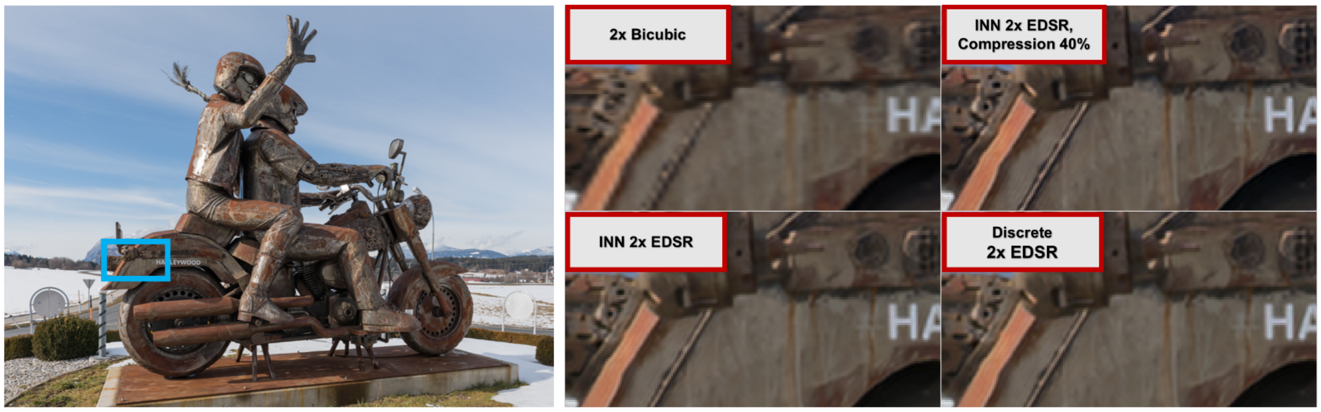


Figure 3. Results on '0793' Div2k with 2x EDSR.

by 0.33 multiplier. For the partition tuning experiments, we used only 1000 iterations for each task, with learning rate starting from $1e-4$ with adjustment by 0.33 multiplier every 300 iterations. In the case of INN initialized from a pre-trained discrete network using algorithm 1, batch normalization layers were fused into parent convolution or fully-connected layer.

Classification experiments For classification experiments on Imagenet dataset [10], we have used pre-trained discrete
models from torchvision library [8]. For experiments on Cifar10 [5], we have used pre-trained discrete networks from pytorchcv library. INNs initialized from pre-trained discrete networks were trained for 50 epochs (batch size 128) in the case of Cifar10 dataset and 100 epochs (256 batch size) in the case of Imagenet dataset. For INN initialized from scratch, we have used the same training setup as for discrete models.

Super-resolution experiments For super-resolution experiments using the SRCNN network [2], we have firstly trained a discrete model. The discrete model was trained for 400 epochs on the 91-image dataset [13] with batch size 16. For the EDSR [7] experiments, we have used official PyTorch [9] implementation with pre-trained models. INNs initialized from the pre-trained models were trained during 80 epochs for each super-resolution model.

D.1. Pruning without fine-tuning

For completeness, we provide additional plots of accuracy depending on compression rate (see Fig. 2). It is easy to see that our networks outperform the channel selection methods. We use similar notations as in [6] for the provided channel selection methods. We include results for ResNet-50 [3] and VGG-19 [11] on Imagenet, 2x EDSR (also see Fig. 3), 3x SRCNN.

D.2. Feature maps visualization

The output of an INN continuously depends on the channel index, as channels in integral networks are discretizations of continuous signals. At the same time, there is no any specific continuous order in a discrete network (see Fig. 4)

D.3. Implementation

We provide code listings for 2D convolution integral layer implementation and trainable partition. Implementa-

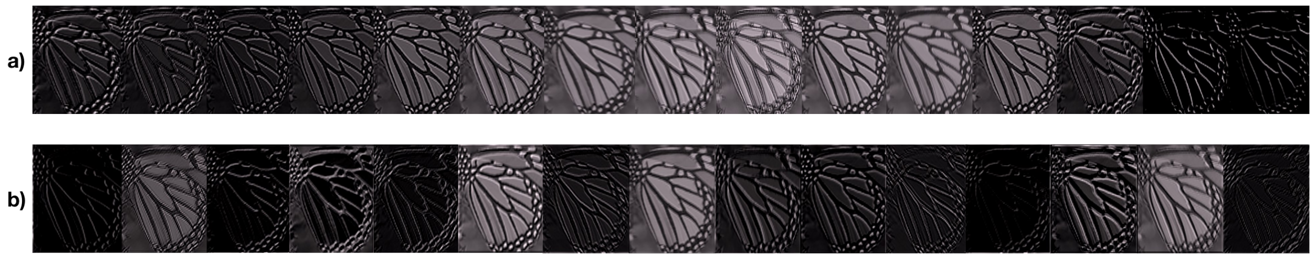


Figure 4. Visualization of feature maps after the first convolutional layer in SRCNN. a) Integral SRCNN. b) Discrete SRCNN. Feature maps of the integral network are organized in a continuous way compared to the vanilla network.

tion of the fully-connected layer can be obtained in a similar way. These listings do not include full implementation of our library but provide the sketch of integral operators development using PyTorch.

```
class Conv2dCubicWeights(torch.nn.Module):
    """
    Parameters
    ----------
    in_points: size_t.
    out_points: size_t.
    kernel_size: size_t.
    """
    def __init__(self, in_points, out_points,
    kernel_size=3):

        torch.nn.Module.__init__(self)
        self.weight = torch.nn.Parameter(
            2.0*torch.rand(
                1, kernel_size**2, out_points,
    in_points
            ) - 1.0
        )
        self.kernel_size = kernel_size


    def forward(self, in_channels, out_channels):
        """
        Parameters
        ----------
        in_channels: size_t.
        out_channels: size.t
        """
        grid = [
            torch.linspace(-1, 1, in_channels),
            torch.linspace(-1, 1, out_channels)
        ]
        grid = torch.stack(
            torch.meshgrid(grid[0], grid[1]),
            dim=-1
        ).unsqueeze(0)

        weight = torch.nn.functional.grid_sample(
            self.weight, grid, mode='bicubic',
            padding_mode='zeros',
            align_corners=True
        )
        weight = weight.permute(3, 2, 0, 1)

        return weight.reshape(
            out_channels, in_channels,
            self.kernel_size, self.kernel_size
        )
```

Listing 1. Continuous weights parametrization for 2D convolution.

```
def integral_uniform_conv2d(
    input, weight, out_channels,
    quadrature, bias=None, stride=1
):
    """
    Parameters
    ----------
    input: torch.Tensor.
    weight: torch_integral.parametrizations.
    CubicWeight2d.
    bias: torch_integral.parametrizations.
    CubicBias1d.
    quadrature: torch_integral.quadratures.
    BaseIntegrationQuadrature.
    stride: size_t.
    """
    in_channels = input.shape[1]
    discrete_weights = weight(in_channels,
    out_channels)
    quadrature_weights = quadrature(
        discrete_weights.shape,
        integration_dims=(1, 2, 3)
    )
    # Adjust discrete weights using quadrature
    weights
    discrete_weights = discrete_weights*
    quadrature_weights
    padding = (discrete_weights.shape[-1] - 1)//2

    # Sample bias function: Bias(x_out)
    if bias is not None:
        bias = bias(out_channels)

    return torch.nn.functional.conv2d(
        input, discrete_weights, bias=bias,
        stride=stride, padding=padding
    )
```

Listing 2. Integral convolution layer.

```
class TrainablePartition(torch.nn.Module):
    """
    Parameters
    ----------
    n_points: size_t. Number of points in
    partition including 0 and 1.
    """
    def __init__(self, n_points):

        torch.nn.Module.__init__(self)

        delta = torch.linspace(0, 1, n_points)
        delta = delta[1:] - delta[:-1]
        self.delta = torch.nn.Parameter(delta)


    def forward(self):
        """
        Returns partition coordinates of the
    segment [0, 1].
        """
        # prevent zero division by adding 1e-8
        # abs also could be used instead of
    square
        grid = self.delta**2 + 1e-8
        grid = grid / grid.sum()
        # fix left point of segement as 0
        grid = torch.cat(
            [torch.Tensor([0.0]), grid], dim=0
        )

        return grid
```

Listing 3. Trainable partition implementation in Pytorch.

References

[1] M. Bevilacqua, A. Roumy, C. Guillemot, and M. L. Alberi-Morel. Low-complexity single-image super-resolution based on nonnegative neighbor embedding. *Proceedings of the British Machine Vision Conference*, pages 135.1–135.10, 2012. 4

[2] Chao Dong, Chen Change Loy, Kaiming He, and Xiaoou Tang. Image super-resolution using deep convolutional networks. *CoRR*, abs/1501.00092, 2015. 4

[3] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016. 4

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

[5] Alex Krizhevsky and Geoffrey E Hinton. Learning multiple layers of features from tiny images. *Technical report*, 2009. 4

[6] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. *arXiv preprint arXiv:1608.08710*, 2016. 4

[7] B. Lim, S. Son, H. Kim, S. Nah, and K. Mu Lee. Enhanced deep residual networks for single image super-resolution. In *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, pages 136–144, 2017. 4

[8] Sébastien Marcel and Yann Rodriguez. Torchvision the machine-vision package of torch. In *Proceedings of the 18th ACM International Conference on Multimedia*, MM '10, page 1485–1488, New York, NY, USA, 2010. Association for Computing Machinery. 4

[9] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in Neural Information Processing Systems*, 32:8026–8037, 2019. 4

[10] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael S. Bernstein, Alexander C. Berg, and Li Fei-Fei. Imagenet large scale visual recognition challenge. *CoRR*, abs/1409.0575, 2014. 4

[11] Zisserman A. Simonyan, K. Very deep convolutional networks for large-scale image recognition. *arXiv preprint*, arXiv:1409.1556, 2014. 4

[12] Michael Spivak. *Calculus*. Publish or Perish, fourth edition, 2008. 1

[13] J. Yang, J. Wright, T. Huang, and Y. Ma. Image super-resolution as sparse representation of raw image patches. In *2008 IEEE conference on computer vision and pattern recognition*, pages 1–8, June 2008. 4

[14] R. Zeyde, M. Elad, and M. Protter. On single image scale-up using sparse-representations. *International conference on curves and surfaces*, pages 711–730, 2010. 4