CS577 - Final Project Report - Spring 2021

Aakef Waris - A20420535

Ali Issaoui - A20474398

May 2021

1 Introduction

1.1 Paper

This project was to examine the use of perceptual loss functions in image transformation tasks and its efficacy.
The paper used in the project was: **Perceptual Losses for Real-Time Style Transfer and Super-Resolution**. [3]

1.2 Work Distribution

Aakef:

- Built the primary network structure in PyTorch
- Built the VGG16 network to be used for loss functions
- Implemented the Data Generator to be used in Training

Ali:

- Debugged the network in training, modified layers, added in reflection padding
- Wrote the training script, loss functions and optimizer.
- Tuned hyper-parameters and visualized training loss

2 Problem Statement

2.1 Perceptual Loss Function Motivations

The generalized problem to solve was to use perceptual loss functions to improve the effectiveness of image transformation tasks. Perceptual loss functions are loss functions that are computed from the intermediary layers of a pre-trained image network. The idea behind this is that traditional pixel by pixel loss functions don't capture the higher level feature representations of an image. For example, when comparing the similarity between 2 photos of a dog, its more important that the dogs have the same color, texture, shape, etc. and less important that the images pixel by pixel are close to each other. Conversely if an image has all of its pixel values shifted by 1, most loss functions would recognize them as dissimilar despite them actually being very similar to the human eye.

2.2 Loss Function and Network Details

These perceptual loss functions were: **Feature Reconstruction Loss**[3] and **Style Reconstruction Loss**[3] and were computed using the VGG16 Network. The paper used for this project demoed 2 problems to display the effectiveness of perceptual loss: style transfer, and super-resolution. For the sake of the project we chose to implement style transfer as it used both loss functions mentioned above, whereas super-resolution only used feature reconstruction.

Feature Reconstruction Loss: This loss function was used to encourage the network to have the output image $\hat{y}$ have similar feature representations to the target image y. To define this loss network we have: $\theta_j(x)$ as the activation of the jth layer of the loss network θ given an input image x. Assuming that layer j is a convolutional layer, then this should output a feature map of shape $C_j \times H_j \times W_j$ corresponding to the channels, height, and width of the activation. With this, the loss function is the squared normalized euclidean distance between activations at layer j:

$$l^{\theta,j}_{feat}(\hat{y}, y) = \frac{1}{C_j H_j W_j} \|\theta_j(\hat{y}) - \theta_j(y)\|_2^2 \tag{1}$$

Style Reconstruction Loss: This loss function was used to push the network to have the output image $\hat{y}$ have similar style to the target image y in aspects such as color, texture, and patterns. With the same definitions above for the activation of a loss network at a given layer j We can define a matrix called the $Gram\ Matrix$:

$$G^{\theta}_{jj}(x)_{c,c'} = \frac{1}{C_j H_j W_j} \sum_{h=1}^{H_j} \sum_{w=1}^{W_j} \theta_j(x)_{h,w,c} \theta_j(x)_{h,w,c'} \tag{2}$$

With this, style-reconstruction loss can be computed as the squared Frobenius norm of the difference between the Gram matrices of the output and target image:

$$l^{\theta,j}_{style}(\hat{y}, y) = \|G^{\theta}_j(\hat{y}) - G^{\theta}_j(y)\|_F^2 \tag{3}$$

The style reconstruction loss can be computed on multiple layers where the final value is the sum of style loss on the set of layers being used. Which yields $l^{\theta,J}_{style}(\hat{y}, y)$ to be sum of loss for all layers $j \in J$.

The problem here is to get these loss functions to converge when training the image transformation network.

3 Proposed Solution

The solution to solve this problem is to try to replicate the described network architecture in the paper, in style transfer, and to reference similar code repositories from other developers. The network proposed by this paper was a convolutional network without using any pooling, this is most likely because there is no classification or regression being done, instead, to down-sample and up-sample the input images, strided and fractionally strided convolutions were used. The initial part of the network is a down-sampling block which consists of convolutions with whole strided each followed by instance normalization and ending with a ReLU activation. The middle of the network consists of 5 residual blocks whose architecture will be explained in following sections. The final component of the network is an up-sampling block which consisted of an up-sampling layer, followed by reflection padding and a convolutional layer. The specific architectures of all component will be further explained in the following sections.

In Implementing the loss functions the specific layers they would be computed on the VGG16 network are:

Feature Reconstruction Loss: `relu3_3`

Style Reconstruction Loss: `relu1_2, relu2_2, relu3_3, relu4_3`

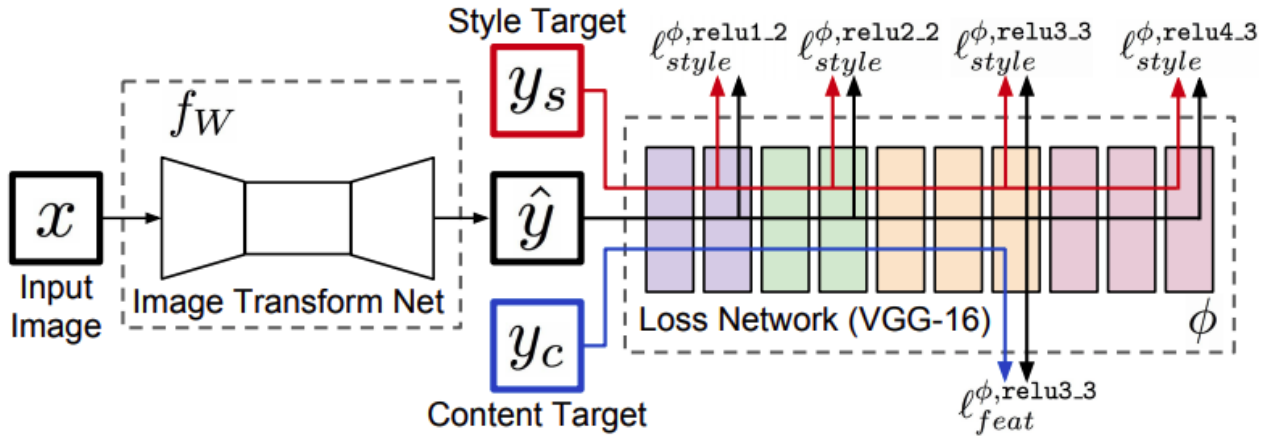

Figure 1: Training Overview: The image transformation take an image as an input and outputs another image, the output image is aimed to have styling and features as a target image based off of the perceptual loss functions computed different layers on the loss network[3]

For reference a closer look at VGG16:

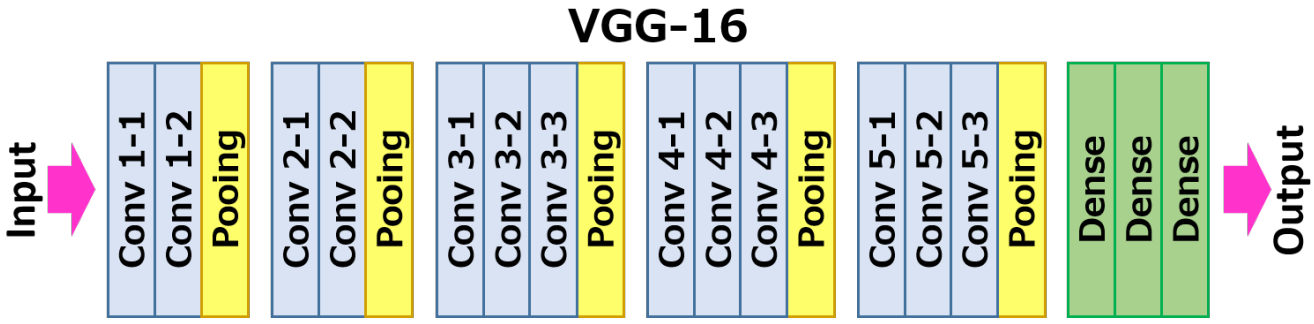

Figure 2: VGG16: The convolutuinal layers shown here are each followed by a ReLU, the indicies specified on the ReLU for the loss functions correspond to the ReLUs that follow each convolutional layer in the figure

4 Implementation Details

When implementing this network there were multiple components that were required:

- An image transformation network that would perform the style transfer on an input image
- A loss network that could compute the ReLU activations at the required layers of VGG16
- A data generator that could load images into memory
- A training script that would optimize the network weights
- A large dataset to train the network on
- Style images to create networks
- A GPU environment to efficiently train the network

4.1 Image Transformation Network (Style-Transfer)

The image transformation network, was a convolutional neural network without pooling layers. Instead for down-sampling and up-sampling, striding and fractional striding was used. The network itself consisted of 3 primary components: **down-sampling** ⇒ **residual blocks** ⇒ **up-sampling**:

Transformation Network:

```
class ImageTransformationNN(torch.nn.Module):

    def __init__(self):
        super(ImageTransformationNN, self).__init__()
        self.down_sample = DownSampleConv()
        self.res = ResidualNet()
        self.up_sample = UpSampleConv()

    def forward(self, X):

        X = self.down_sample(X)
        X = self.res(X)
        y = self.up_sample(X)
        return y
```

4.1.1 Down-Sampling

This component took in a 3 channel image, and scaled it to 32, 64, 128 channels using 3 convolutional layers with reflection padding, each followed by instance normalization. At the end ReLU activation was used.

Instance Normalization: In instance normalization the mean and variance are computed on each channel for each sample as opposed to across all samples in batch normalization[1]:

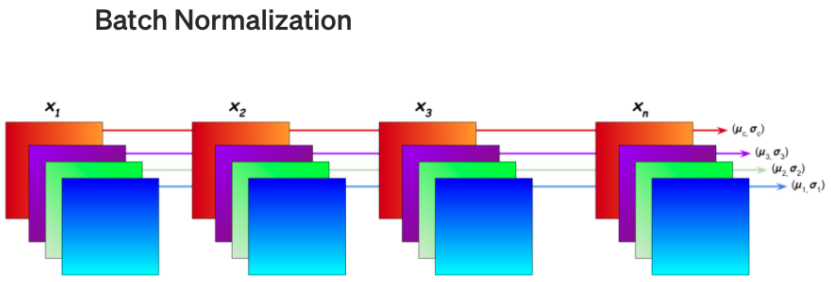

Figure 3:

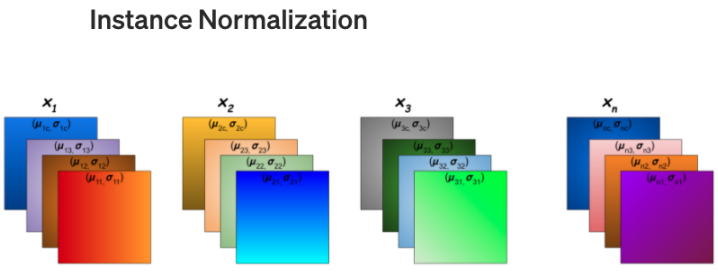

Figure 4:

Reflection Padding: In reflection padding, we pad the image with a reflection from the opposite side[5]:

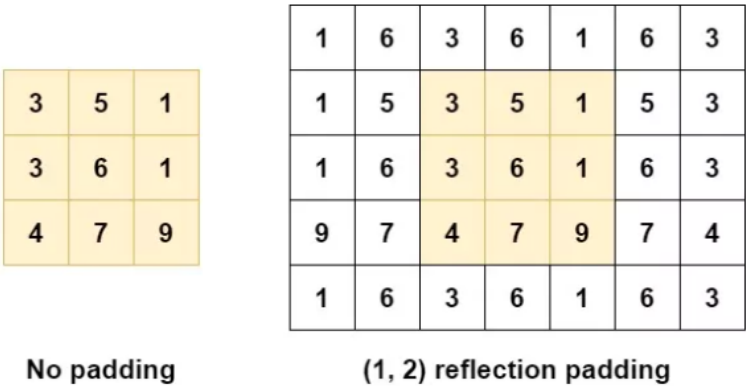

Figure 5:

Down-Sampling:

```
class ConvLayer(torch.nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size=9, stride=1):
        super(ConvLayer, self).__init__()
        reflection_padding = kernel_size // 2
        self.reflection_pad = torch.nn.ReflectionPad2d(reflection_padding)
        self.conv2d = torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride)

    def forward(self, x):
        out = self.reflection_pad(x)
        out = self.conv2d(out)
        return out

class DownSampleConv(torch.nn.Module):
    def __init__(self):
        super(DownSampleConv, self).__init__()
        self.conv2d1 = ConvLayer(3, 32, kernel_size=9, stride=1)
        self.norm1 = torch.nn.InstanceNorm2d(32, affine=True)
        self.conv2d2 = ConvLayer(32, 64, kernel_size=3, stride=2)
        self.norm2 = torch.nn.InstanceNorm2d(64, affine=True)
        self.conv2d3 = ConvLayer(64, 128, kernel_size=3, stride=2)
        self.norm3 = torch.nn.InstanceNorm2d(128, affine=True)

        self.relu = torch.nn.ReLU()

    def forward(self, X):
        ...
```

4.1.2 Residual Blocks

This component was a series of 5 Residual Blocks. Each Residual Block consisted of a convolutional layer as specified earlier with reflection padding, instance normalization, a ReLU another convolutional layer, and a final instance normalization layer. The residual was added to the output after all components were applied. The residual blocks maintained 128 channels across the residual body, the channels were not reduced until the final upsampling component[2]:

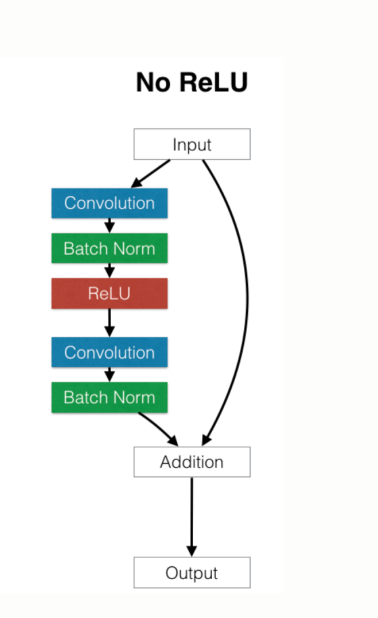

Figure 6:

Residual Blocks:

```
class RBlock(torch.nn.Module):
    # Specification:  http://torch.ch/blog/2016/02/04/resnets.html

    def __init__(self, channels:int):
        super(RBlock, self).__init__()
        self.conv2d1 = ConvLayer(channels, channels, kernel_size=3, stride=1)
        self.norm1 = torch.nn.InstanceNorm2d(channels, affine=True)
        self.relu = torch.nn.ReLU()
        self.conv2d2 = ConvLayer(channels, channels, kernel_size=3, stride=1)
        self.norm2 = torch.nn.InstanceNorm2d(channels, affine=True)

    def forward(self, X):
        residual = X
        y_hat = self.conv2d1(X)
        y_hat = self.norm1(y_hat)
        y_hat = self.relu(y_hat)
        y_hat = self.conv2d2(y_hat)
        y_hat = self.norm2(y_hat)
        y = y_hat + residual
        return y


class ResidualNet(torch.nn.Module):

    def __init__(self):

        super(ResidualNet, self).__init__()

        self.block1 = RBlock(128)
        self.block2 = RBlock(128)
        self.block3 = RBlock(128)
        self.block4 = RBlock(128)
        self.block5 = RBlock(128)

    def forward(self, X):

        y = self.block1(X)
        y = self.block2(y)
        y = self.block3(y)
        y = self.block4(y)
        y = self.block5(y)

        return y
```

4.1.3 Up-Sampling

The up-sampling component followed the same architecture of the down-sampling component except the channels were brought from 128 to 64 to 32 to 3 channels for the output image.

Up-Sampling:

```
class UpSampleConv(torch.nn.Module):
    def __init__(self):
        super(UpSampleConv, self).__init__()
        self.deconv1 = UpsampleConvLayer(128, 64, kernel_size=3, stride=1, upsample=2)
        self.in4 = InstanceNormalization(64)
        self.deconv2 = UpsampleConvLayer(64, 32, kernel_size=3, stride=1, upsample=2)
        self.in5 = InstanceNormalization(32)
        self.deconv3 = ConvLayer(32, 3, kernel_size=9, stride=1)
        self.relu = torch.nn.ReLU()

    def forward(self, X):

        y = self.relu(self.in4(self.deconv1(X)))
        y = self.relu(self.in5(self.deconv2(y)))
        y = self.deconv3(y)

        return y
```

4.2 Loss Network (VGG-16)

The loss network used was VGG16. The model weights were loaded in using PyTorch. The forward function only computed activations at the layers needed for the loss functions.

To compute the loss it makes sense to use either the first or the second block of the VGG16 (`relu1_2` or `relu2_2`) since they represent the higher level features of the image and emphasize the use of the perpetual loss on this kind tasks. After trying with both the first and the second block, we concluded that the second one (`relu1_2`) gives better result since it balances between the high level features and the low level ones.

4.3 Datasets

The dataset that was used in this project is the Microsoft Common Objects in Context dataset (MSCOCO)[4]. It is a large-scale object detection, segmentation, and captioning dataset that has features, containing around 330K images for general purposes.

However, to train the model a 30,000 image subset of the MSCOCO dataset was used. Due to infrastructure limitations the full 330K sample was unable to be used.

4.4 Network Training

The training of the network was the longest part of the project. The model was trained using the Adam optimizer, with a learning rate of 1e-3.

Choosing the learning rate was not straight forward and was part of the hyper parameter tuning process. First, we started with a learning rate of 1e-2, but the network converged too fast. Then, after some research, we discovered that a high learning rate causes the model to converge fast, not achieving the optimal results. So, we tried with a learning rate of 1e-3 and 1e-4. With the learning rate of 1e-4, the losses were converging smoothly but very slowly, especially with a dataset of 30,000 images, training with this configuration and the computational power that we had was almost impossible. So, the last learning rate that was chosen for this project is 1e-3 which was a balance between the two previous values.

The network was train on 2 epochs over the entire dataset, using a batch size of 4. Since every style image requires its own network, the training of our best network using the mosaic style took around 8 hours.

4.5 Hardware and Technology

The model was trained on Google Colab, using a Tesla P100-PCIE-16GB GPU. The deep learning framework that was used was PyTorch.

5 Results and Discussions

5.1 Network Sample Outputs

The resulting network was trained on a sample style image, for any new style image, another network would have to be changed. The network we trained had the style image:

Figure 7: Mosaic

After training the network we can see it's performance on some sample images that are taken from the testing set:

Figure 8: Sample

Figure 9: Output

Figure 10: Sample

Figure 11: Output

Figure 12: Sample

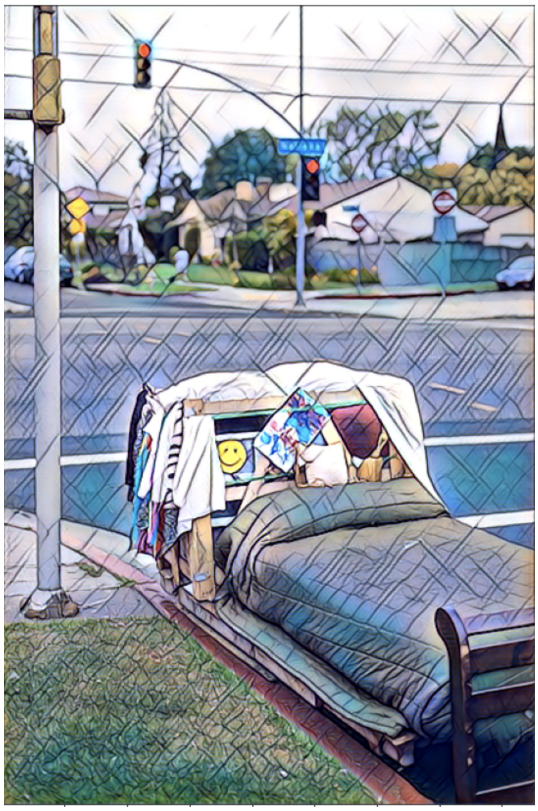

Figure 13: Output

5.2 Network Performance

Because there was no classification or regression, there is no validation loss to examine but only training loss:

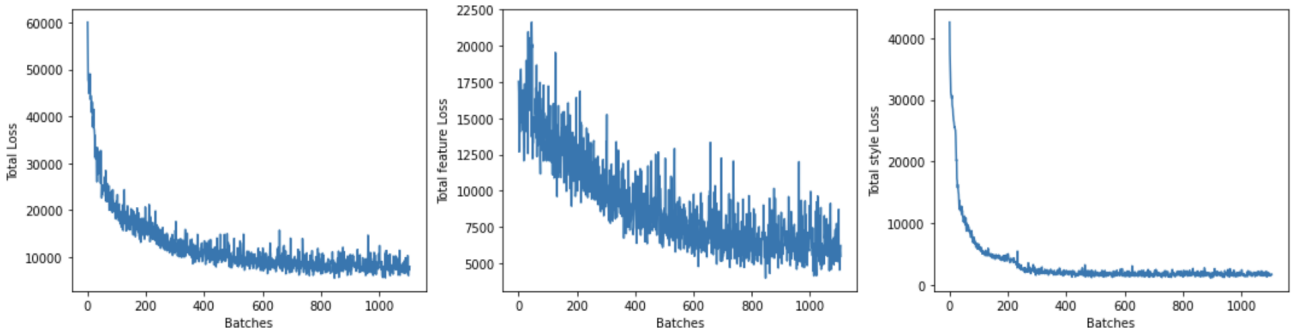

Figure 14: Training Loss: We can see here 3 training loss graphs. The style loss has a relatively smooth descent where feature loss is a lot more noisy. Regardless, it is very clear that both losses converge and this can be seen in the total loss graph

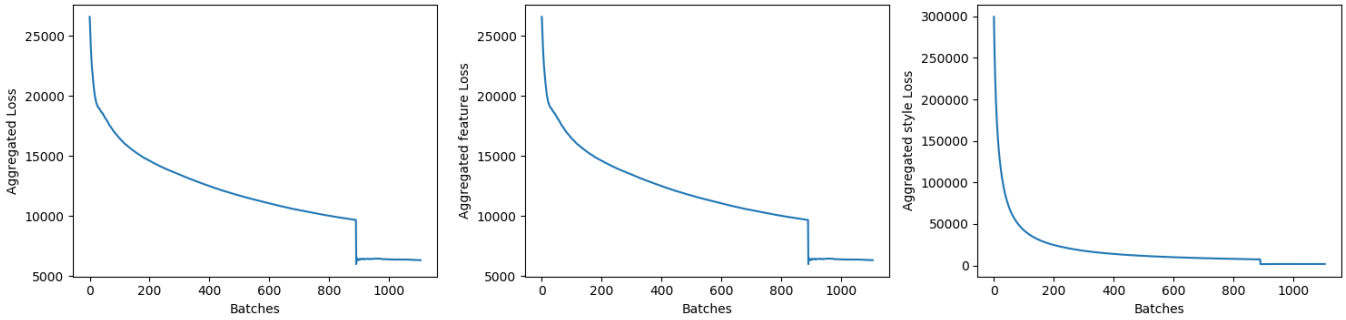

Figure 15: Aggregated Losses: These figures represent the aggregated losses (loss over the number of batches) during the training. The drop after the 800's batches is due to finishing the first epoch and starting the second one over the same dataset.

5.3 Evaluation:

We can see that the loss functions appear to converge after 1000 iterations, however, the losses are still decreasing and training more, and over the entire COCO dataset would give better results. In the original paper, the training over the entire dataset took only 4 hours for 40 000 iterations thanks to the computational capacity that they had. However, in our case, using Google Colab's resources, training over approximately 5000 iterations took 8 hours, and due to the lack of time in our project, we could not train the network more but the continually decreasing loss shows that our network would achieve better performance when trained over the entire COCO dataset.

6 Conclusion

In this project, we have demonstrated that using pre-trained networks as perceptual loss functions instead of traditional loss functions can significantly increase the reliability of image transformation tasks, especially when trying to capture more abstract features. This experience proved a great opportunity to employ techniques learned through the semester in designing neural networks, and to face the different problems that a machine learning engineer would encounter. Future contributions to this project could include performing other image transformation tasks such as super resolution.

References

[1] MaYank Agarwal. *Batch Normalization, Instance Normalization, Layer Normalization: Structural Nuances*. Aug. 2020. URL: https://becominghuman.ai/all-about-normalization-6ea79e70894b#:~:text=In%20%E2%80%9CInstance%20Normalization%E2%80%9D,%20mean,sample%20across%20both%20spatial%20dimensions..

[2] Sam Gross and Micheal Wilber. *Training and investigating Residual Nets*. URL: http://torch.ch/blog/2016/02/04/resnets.html.

[3] Li Fei-Fei Justin Johnson Alexandre Alahi. "Perceptual Losses for Real-Time Style Transfer and Super Resolution". In: (2016).

[4] Tsung-Yi Lin et al. *Microsoft COCO: Common Objects in Context*. cite arxiv:1405.0312Comment: 1) updated annotation pipeline description and figures; 2) added new section describing datasets splits; 3) updated author list. 2014. URL: http://arxiv.org/abs/1405.0312.

[5] Christian Versloot. *Using Constant Padding, Reflection Padding and Replication Padding with TensorFlow and Keras*. Feb. 2020. URL: https://www.machinecurve.com/index.php/2020/02/10/using-constant-padding-reflection-padding-and-replication-padding-with-keras/.