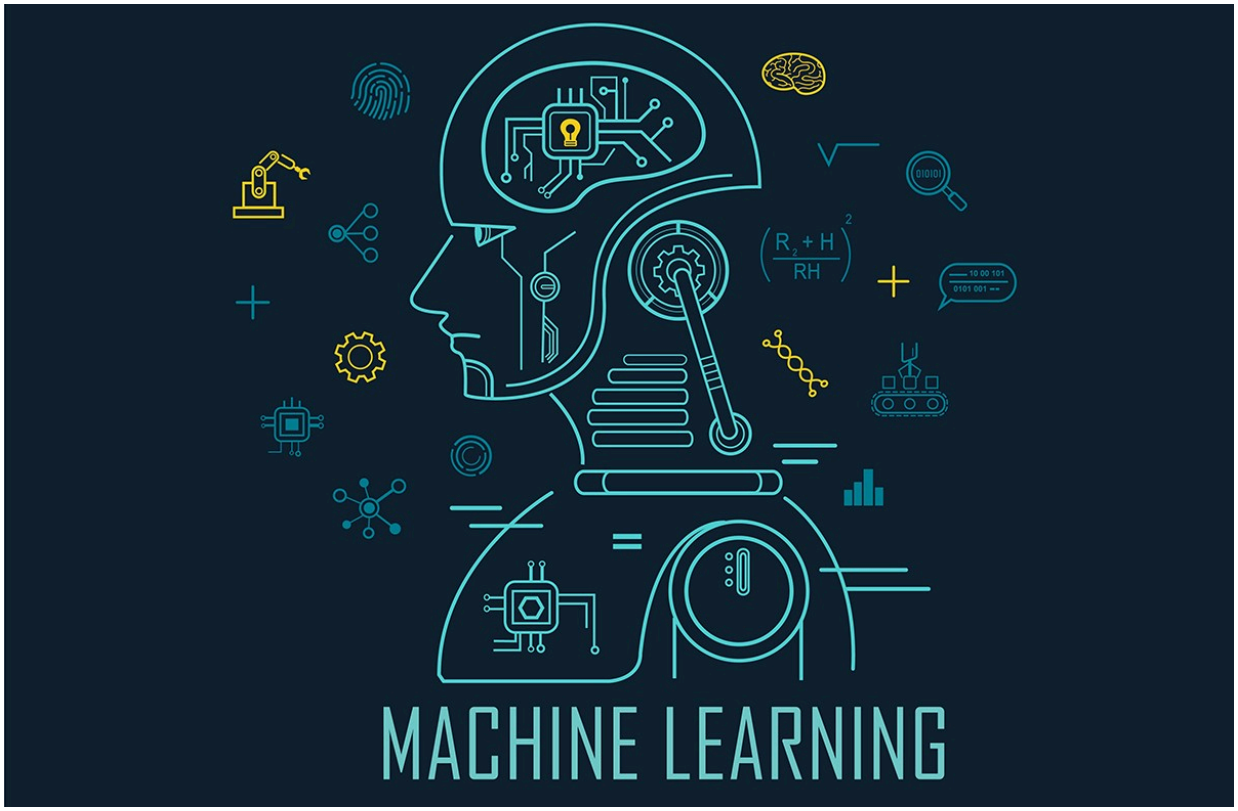
Using machine learning for super-resolution and upscaling images

Student: Roy Mahler

Instructor: Gad Lidror

Subject: Machine learning

Learning institute: Tichonet



Abstract

In this article I will be describing the methods I used to upscale images and potentially restore, inpaint, and superresolution of images in general. I used many advanced and modern techniques that are featured in the researches and articles from which I got inspiration.

Images, when saved on the computer, are made of information. This information is known as pixels, which is an array of numbers with a dimension. For example, a colored image will be using a 3-dimensional array: the array consists of the number of pixels vertically X number of pixels horizontally X the color channels like RGB (Red, Green, Blue) that contain a number. A grayscale image will be a 2-dimensional array: the number of pixels horizontally X number of pixels vertically, and in each pixel will be the darkness level as a number.

When we change the size of images, some of their data is lost. That is because we want to change the pixels horizontally and vertically so some data is lost and the image seems more "pixelated". We can only decrease the size of images unless we have the original image saved. The reason for this is that the computer cannot "invent" information he does not possess.

So what can be done to increase the image size if we don't have a bigger-sized image? With today's advanced technology and the computing powers that are available to everyone (such as Google Colab, which I used to run this very project) we can program the computer and train it to be able to process images and decide how to add and integrate the missing data accurately to eventually get an image with better resolution.

Upscaling images without the desired image already existing was practically impossible in the past. That is because computers must be able to "guess" (as I mentioned above) the missing information (in the forms of pixels) and formerly, programmers didn't have the ability to create a neural network that could do this. But as I said, today we can do this.

In this project, I will attempt to upscale images using a deep - neural network and train an AI to add and improve an image given to it.

This application and project are meant for all kinds of image repairing, upscaling, and general image improvements for any use.

Common methods for superresolution

Nowadays, deep learning scientists and other related researchers are mostly using various *Generative Adversarial Networks* (GANs) and models for superresolution, image upscaling, image repairing, and inpainting. These methods are not as new but they still remain used widely. GANs are a great way of enhancing images, but they have some major flaws such as their loss function, which is not specifically designed for the purpose of superresolution and thus might not be good for multiple usages like upscaling AND image repairing.
For this reason, many deep learning methods for superresolution are unable to be applied efficiently and widely to multiple usages and thus, are not as efficient at image enhancing.

My innovative method

The model I have trained in this project seems to be able to do various image enhancing actions including those I mentioned previously (image upscaling, repairment, inpainting, and superresolution generally). In this project, I utilize techniques and methods based upon researches made by professional and talented AI researchers and I have combined these methods to suit my own research and necessity.

Usage of data in the project

The dataset I used here is made of high-quality images with lower-quality versions of said images. The data set is called *"The Oxford IIIT Pet Dataset"* and it contains roughly 200 images per class, with 37 classes included.

Methods used in this project

I am using the following methods in my project. All of these will be explained later in this article further below:

- A U-Net architecture
- A ResNet-34 that has been modified to a U-Net - meaning an encoder and decoder based upon the ResNet-34.
- Pixel shuffle upscaling with IGNR initialization.
- Transfer learning from pre-trained ImageNet models (ResNet-34).
- A loss function based on activations from a VGG-16 model, pixel loss, and gram matrix loss.
- Discriminative learning rates.
- Progressive resizing.

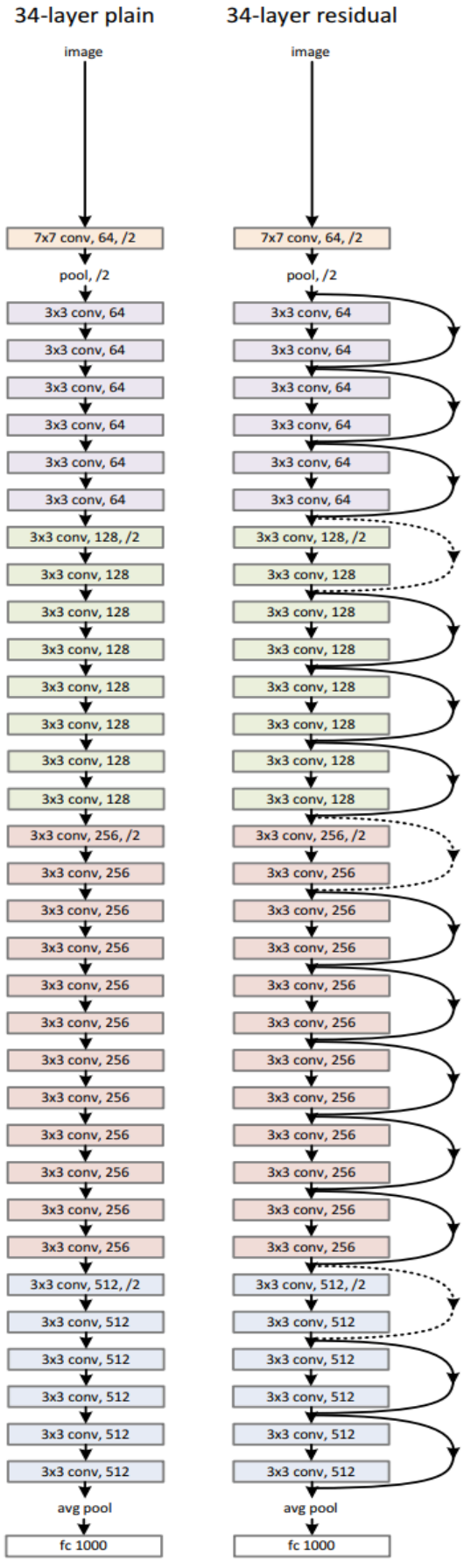
The model has approximately 41 million parameters.

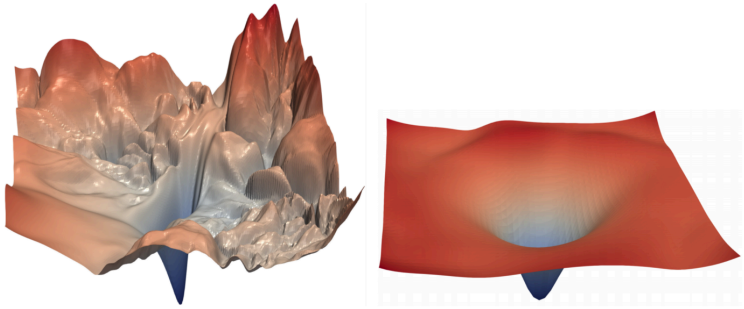
ResNet and ResBlocks

ResNets are convolutional neural networks made out of a series of *Residual Blocks* (ResBlocks). These networks have skip connections that make them unique from the traditional CNNs (Convolutional Neural Network) networks. ResNets are a great solution for the vanishing gradient problem, meaning whereas more layers are added, the training slows down significantly and the accuracy does not improve and in some cases, even decreases. The skip connections the ResNet possesses fix this problem and open up a way for much deeper CNNs.
These skip connections are shown in the diagrams to the right and will be explained further in this article.

So now we know that convolutional networks can be much deeper, more efficient, and more accurate if they contain short connections between the layers that are close to the input and to the output.
Now we can talk about ResBlocks and dense blocks.

The loss surface is very unbalanced and looks like a series of hills and sharp edges as the left diagram (a) below shows. The lowest loss is the lowest point. By adding layers to the model, the predictions might be worse.

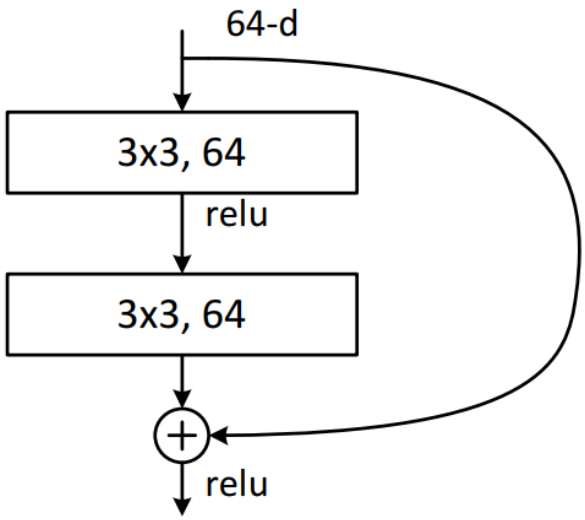




(a) without skip connections (b) with skip connections

Figure 1: The loss surfaces of ResNet-56 with/without skip connections. The proposed filter normalization scheme is used to enable comparisons of sharpness/flatness between the two figures.

An effective and successful solution is to add cross-connections between layers of the network, thus, allowing large sections that are not needed to be skipped (if needed of course). This actually creates a loss surface that seems like diagram (b) above shows. It is much easier to train the model like this, with optimal weights to reduce the loss.



A ResBlock within a ResNet. Source: Deep Residual Learning for Image Recognition:

Each block of the ResBlock type has two connections from its input, one going through a series of convolutions, batch normalizations, and linear functions and another connection skipping over that series of convolutions and other functions. These are known as identity, cross/skip connections. Then, the tensor outputs of both connections are added together.

DenseBlocks and DenseNets

A ResBlock can provide a tensor addition as an output, but by adding more skip/cross connections we get a DensBlock in which each layer takes all preceding features as input and thus, provides a combined tensor of all preceding features from each layer. Due to this "chaining", DensBlocks takes a much bigger portion of memory compared to other kinds of architectures and so, it is suited for lighter or smaller datasets.

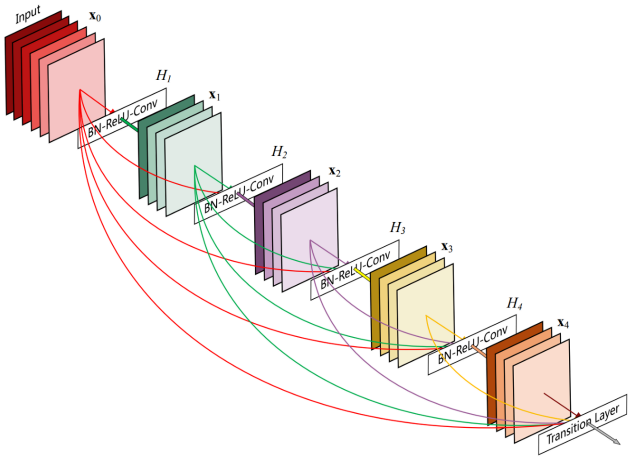
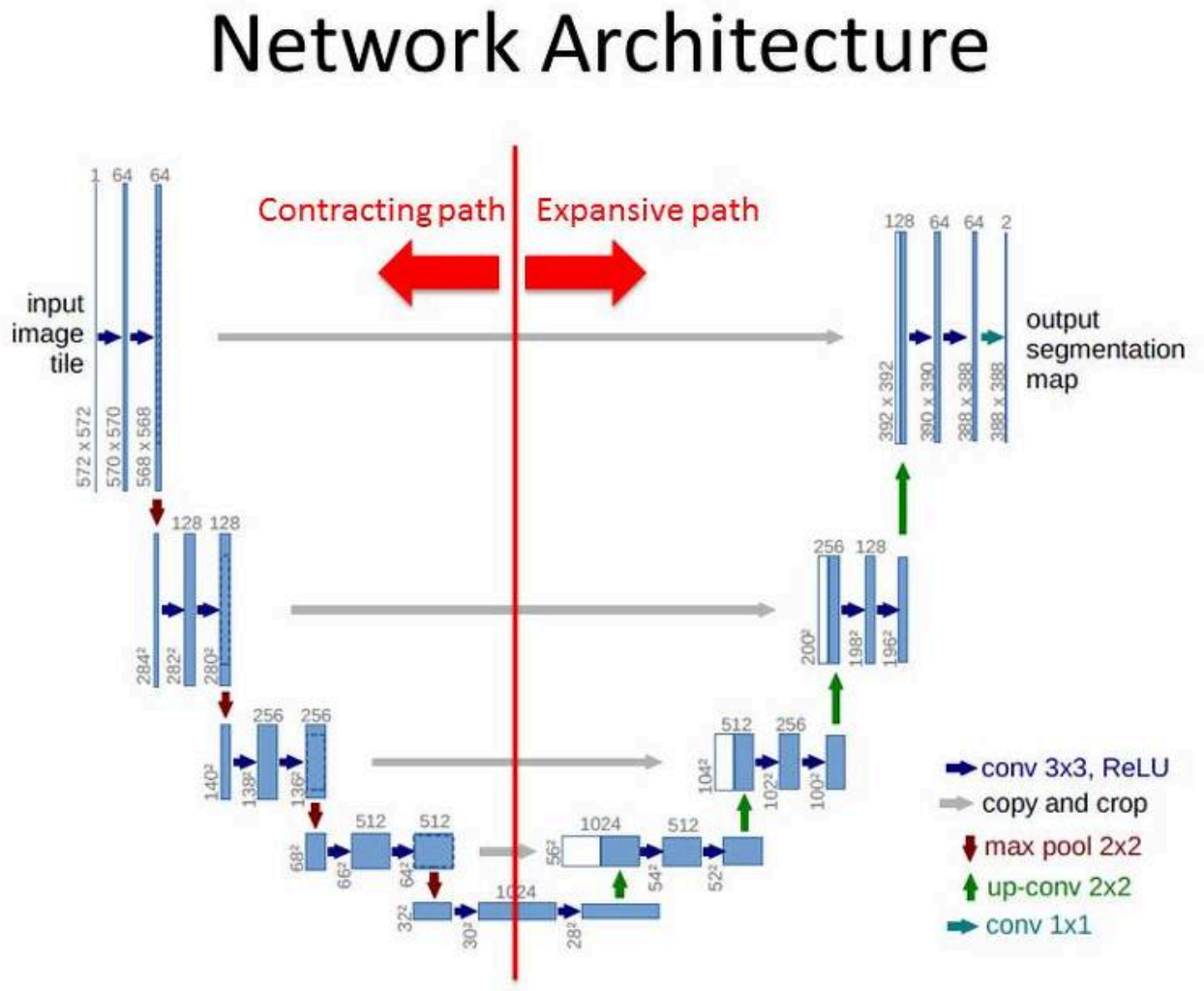


Figure 1: A 5-layer dense block with a growth rate of $k = 4$. Each layer takes all preceding feature-maps as input.

U-Nets

A U-Net is a kind of architecture for convolutional neural networks that was originally developed for biomedical image segmentation at the Computer Science Department of the University of Freiburg, Germany.

U-Nets are remarkably practical in cases where you need an output of a similar size when the input needs that amount of spatial resolution. The U-Net consists of a contracting path, also known as downsampling path or encoder (left-wing), and an expansive path, also known as upsampling path or decoder (right-wing).



The downsampling path follows the typical architecture of a convolutional network. The network is made out of repeated application of two 3x3 convolution layers (unpadded), followed by an improvement linear layer (ReLU) and 2x2 max pooling operation using stride 2 for downsampling. In each step we take in the downsampling path, we double the number of feature channels. Every step we take in the upsampling path consists of an upsampling of the feature map followed by a 2x2 convolution that is known as an "up-convolution" that halves the number of feature channels, a nexus made with the correspondingly cropped feature map from the contracting path and two 3x3 convolution layers, each followed by a ReLU layer. The cropping is essential due to the loss of border pixels in the convolution layers. At the last and final layer, a 1x1 convolution is used to map each component feature vector to the desired number of classes.

In this model, there are 150 layers in total.

"Up-Convolutions" or transposed convolutions

Each convolution located in the expensive or upsampling part of the U-Net needs to add pixels around and in-between the existing pixels to reach the desired resolution in the output.
This can be seen and visualized in the next example.

The blue pixels seen in the diagram are the original 2x2 pixels being expanded to 5x5 pixels. 2 pixels of padding are added around the outside and also a pixel in-between each pixel. In the diagram, all new pixels are zeros and are represented as white pixels.

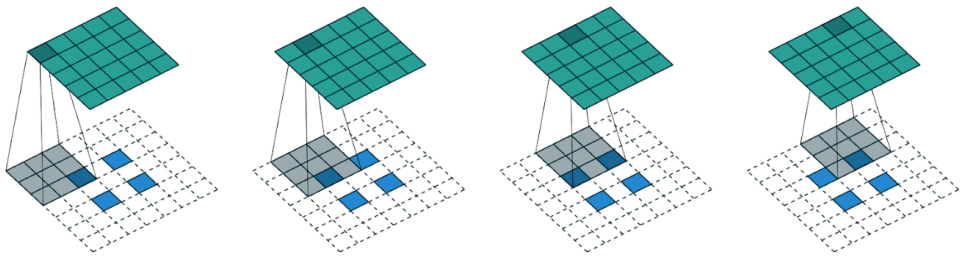


Figure 4.5: The transpose of convolving a 3×3 kernel over a 5×5 input using 2×2 strides (i.e., $i = 5$, $k = 3$, $s = 2$ and $p = 0$). It is equivalent to convolving a 3×3 kernel over a 2×2 input (with 1 zero inserted between inputs) padded with a 2×2 border of zeros using unit strides (i.e., $i' = 2$, $\tilde{i}' = 3$, $k' = k$, $s' = 1$ and $p' = 2$).

Of course, this model does not initialize the new pixels with zeros or by using the weighted average of the pixel, as otherwise, it is unnecessarily making it harder for the model to learn.

The model made here initializes the new pixels with a method known as pixel shuffle or sub-pixel convolution, as seen in the diagram below.

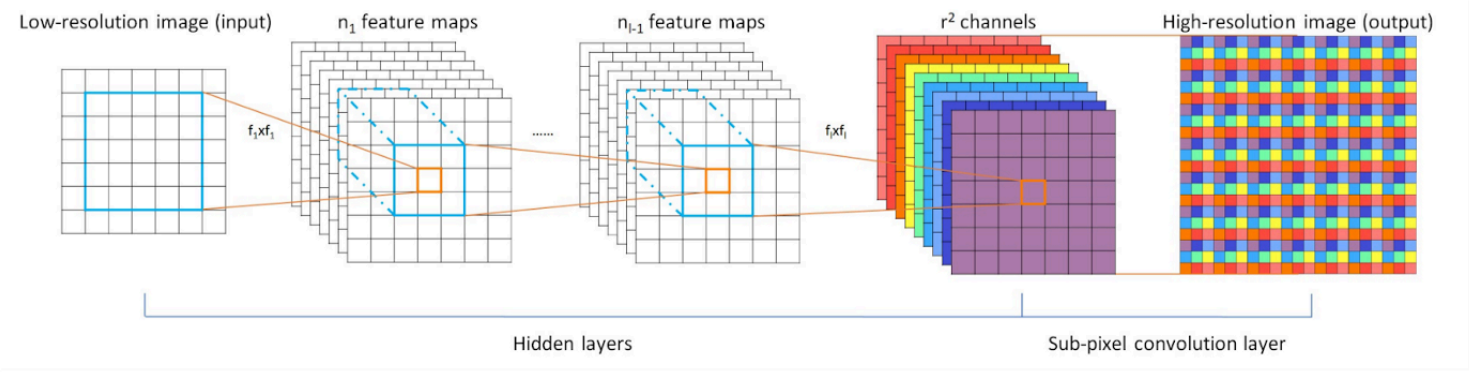


Figure 1. The proposed efficient sub-pixel convolutional neural network (ESPCN), with two convolution layers for feature maps extraction, and a sub-pixel convolution layer that aggregates the feature maps from LR space and builds the SR image in a single step.

The pixel shuffle upscales by a factor of 2, i.e, doubling the dimensions in each of the channels of the image in its current representation at that part of the network. A replication padding is then added, to provide an extra pixel around the image. Average pooling is then executed to "fish-out" features and avoid the checkerboard or "tiled" pattern which results in many super-resolution techniques.

After the rendering for these new pixels is added, the succeeding convolutions improve the details within them as the network continues along the expansive path before then upscaling another step and doubling the dimension once again.

Fine image detail using U-Net

Unfortunately, predictions made using only a U-Net architecture tend to lack fine details. To help address this and improve predictions, cross/skip connections can be added between blocks of the network.

Instead of adding cross-connections every two convolutions such as in a ResBlock, about which I've explained earlier, the skip connections cross from same-sized parts in the contracting path to the expansive part. These are the gray lines shown in the diagram presented in the "U-Net" section.

The outputs made by the U-Net blocks are concatenated, thus, making them less similar to ResBlocks and instead, more similar to DenseBlocks. But to keep

memory usage from growing too large, stride two convolutions are added to reduce the grid size back down.

ResNet-34 Encoder

In this model, I use ResNet-34, which is a 34 layer ResNet, as the encoder in the downsampling part of the U-Net.

The Fastai library introduces to us a function called "unet_learner". The function, when provided with an encoder architecture (arch), will automatically construct the decoder side of the U-Net architecture. If provided with an encoder of a ResNet-34, the function will transform it into a U-Net with cross-connections.

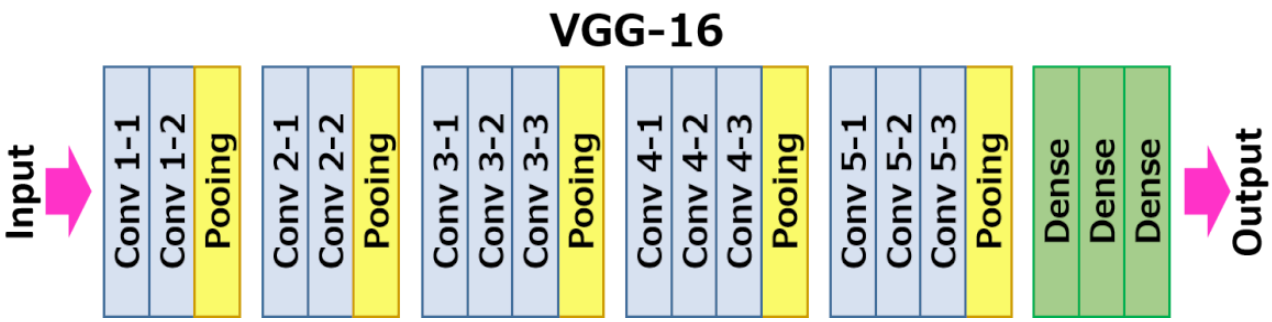
In the case of image generation models, to perform its predictions effectively, it is highly recommended to use a pre-trained model so training would speed up. Once a pre-trained model has been introduced, the U-Net possesses a starting knowledge of the kind of features that require detection and improvement. When images and photographs are used as inputs, using weights of a model pre-trained on ImageNet has been a great success so far.

Loss Function

The loss function used in this model is a feature loss, also known as perceptual loss. The loss function used here is similar and based upon the research in the paper *Losses for Real-Time Style Transfer and Super-Resolution*, and the improvements shown in the Fastai course.

As I said, this model is trained with a similar loss function featured in the paper mentioned above, i.e, using a VGG-16 pre-trained network. This model's loss function, however, can combine pixel mean squared error loss and gram matrix loss as well. This method has been found to be very effective by the Fastai researchers from whom I've taken inspiration.

In this loss function, rather than encouraging the pixels of the output image to exactly match the pixels of the target image, I instead encourage them to have similar feature representations as computed by the loss network. We can, for example, call the loss function network φ, so if we say $\varphi_j(x)$ is the activations of the jth layer of the network φ when processing an image x, if j is a convolutional layer then $\varphi_j(x)$ will be a feature map.



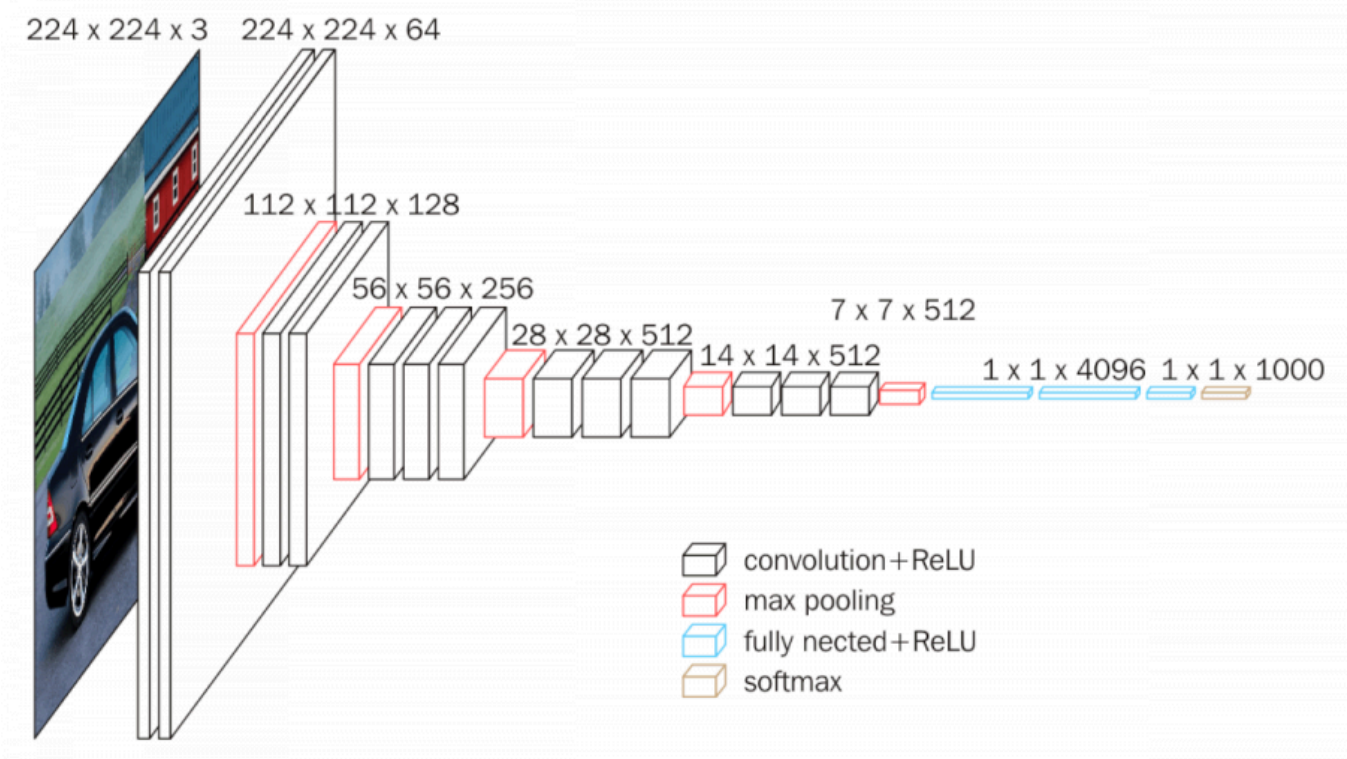
VGG-16

VGG is a convolutional neural network (CNN) that originated in 2014. Its 16 layer version makes off the loss function for the training part of the model.

The VGG model is a network pre-trained on ImageNet and is used to assess the generator model's loss. Usually, this model is used as a classifier in order to differentiate images and to determine what an image is, for instance, if it is a car, a cloud, or a person.

The head of the VGG model is ignored and the loss function uses the in-between activations in the backbone of the network, which represents the feature detections.

Further on, the backbone and the head of the network will be described.
These activations can be seen by taking a look at the VGG model and spotting all of



the max-pooling layers. In these layers, the grid size changes, and thus, features are detected.
An example visualizing the heatmaps of the activations for a variety of images is featured on the next page. We can see a variety of features detected by a network in the different layers it contains.

During the training period, this model utilizes the loss function based on the VGG model's activations. Throughout the entire training, the loss function remains, unlike other super-resolution competitor models such as the critic part of a GAN. That is another thing differentiating this model from GANs.

The feature map channels are used in order to detect features such as hair, nose, tails, wings, etc. it can also detect and identify the type of material amidst many other types of features.
The activations located at the same layer for the input image and the output image are compared using mean squared error or the least absolute error also known as L1 error for the base loss. These two are feature losses. This model uses the L1 error function instead of the squared means because unfortunately, I did not have enough time to implement the squared means and train the model to see the results.

The use of the L1 error function allows the loss function to determine what features are in the target ground truth image and to assess how well the model's prediction features match these compared to only differentiating pixel differences.

Figure 2. Visualization of features in a fully trained model. For layers 2-5 we show the top 9 activations in a random subset of feature maps across the validation data, projected down to pixel space using our deconvolutional network approach. Our reconstructions are *not* samples from the model: they are reconstructed patterns from the validation set that cause high activations in a given feature map. For each feature map we also show the corresponding image patches. Note: (i) the the strong grouping within each feature map, (ii) greater invariance at higher layers and (iii) exaggeration of

Training details

This model's training process is initialized by first creating the model as mentioned above, i.e, creating a U-Net based on ResNet-34 pre-trained on ImageNet using a loss function based on a VGG-16 network pre-trained on ImageNet as well combined with pixel loss and gram matrix.

Training data

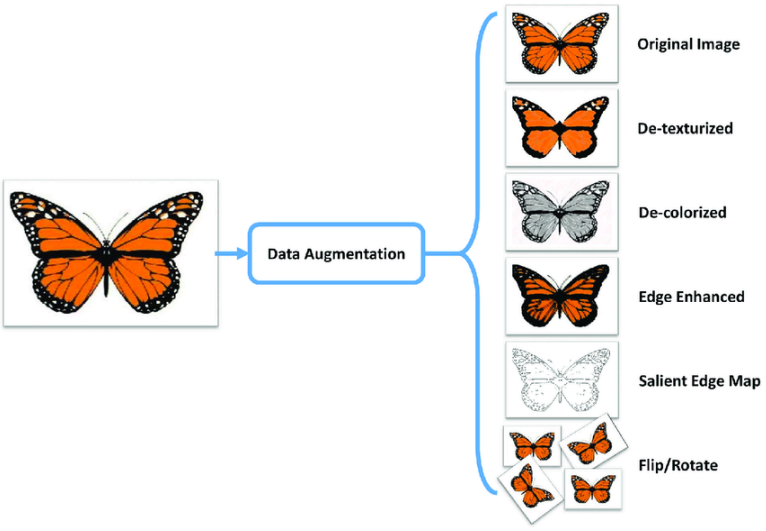
For models made for super-resolution, there is a practically infinite amount of data that can be converted to be used as training data, and numerous datasets already exist on the web. If a high-quality set of images can be obtained, it can be programmed and resized to smaller images, and thus, we have a training set with pairs of low-resolution images and high-resolution images. The predictions made by the model can then be compared with the high-resolution images.

The low-resolution image is essentially a copy of the target/ground truth image with smaller-sized dimensions. The low-resolution image is upscaled to match the dimensions of the target image so it would be an appropriate input for the U-Net-based model.

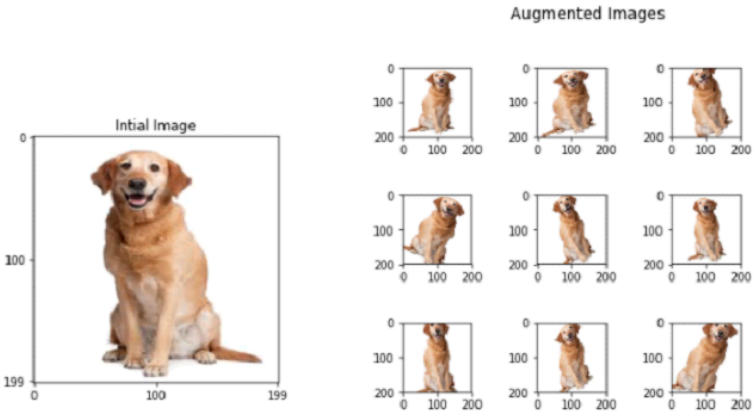
In order to create good training data that can even remove image noises, inpaint, and more, we can use image augmentation by the following actions:

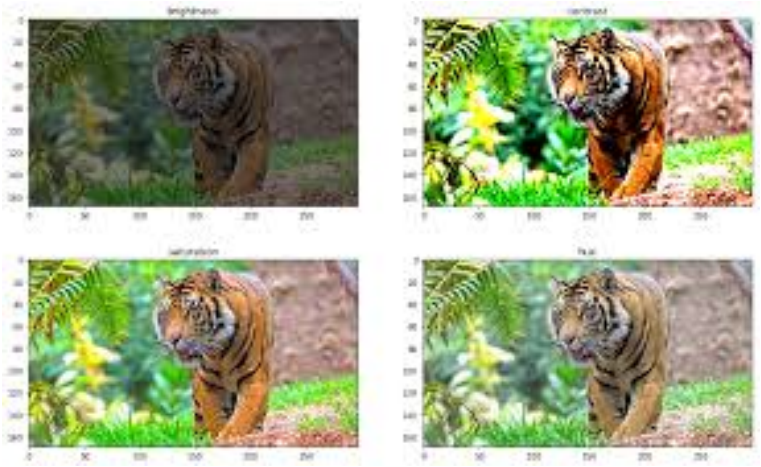
- Randomly reducing the quality of the image (within its bounds)
- Cropping random parts of the image
- Flipping vertically and horizontally
- Adjusting the image's lighting
- Stitch obstacles such as text and symbols in different sizes
- Convert the image to black and white
- Create small distortions such as holes
- Randomly adding noise
- Adding warping

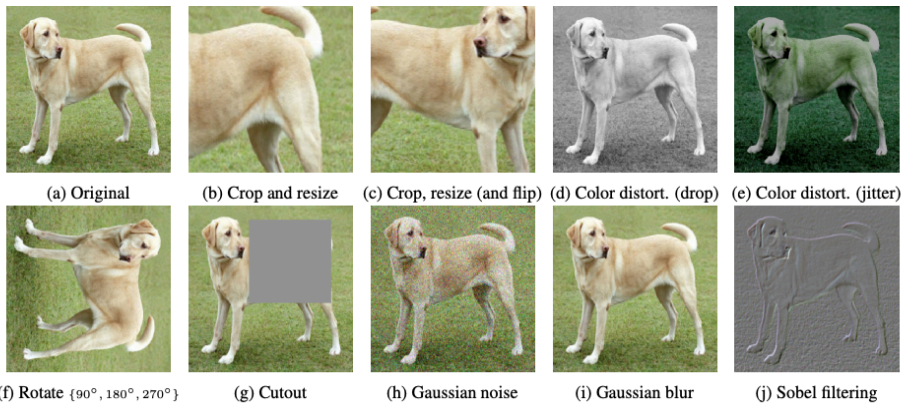
Changing the quality and adding different noises randomly for the training images improves the model's generated output and allows it to perform more actions in order to enhance a given image in the best way possible.



The images below are an example of image augmentation taken from the web:







Illustrations of the studied data augmentation operators. Each augmentation can transform data stochastically with some internal parameters (e.g. rotation degree, noise level). Note that we *only* test these operators in ablation, the *augmentation policy used to train our models* only includes *random crop (with flip and resize)*, *color distortion*, and *Gaussian blur*. (Original image cc-by: Von.grzanka)

We can understand that by augmenting the data, the training set can be expanded and will be able to include a variety of new data to help the model do the actions mentioned above.

Quality improvement

The U-Net-based model utilized here enhances the details and the different features in the upscaled output image, i.e, the model generates an improved image. The model contains roughly around 41 million parameters.

Training the head and backbone of the model

The training process for this model uses 3 methods in particular. These are progressive resizing, freezing and unfreezing the gradient descent update of the weights in the backbone and discriminative learning rates.

As mentioned above, The U-Net's architecture is split into two segments, the contracting path, and the expansive path, but here we will call them the backbone (left-wing) and the head (right-wing).

The backbone is the left side of the U-Net, i.e, the downsampling part of the network that is based on ResNet-34. The head is the right side of the U-Net, meaning the upsampling part of the network.

The backbone contains pre-trained weights based on the ResNet-34 trained on ImageNet, this is the transfer learning used in the project.

The head, however, needs to train its weights since these layers' weights are randomly generated when initialized to produce the wanted end output.

At the very start, the output from the network is practically random changes of pixels other than the Pixel Shuffle sub-convolutions used as the first step in each upscale in the upsampling path of the network.

Once the head, which is on top of the backbone, is trained, it allows the model to learn to do something different with its pre-trained knowledge located in the backbone.

Train the head, freeze the backbone

In order that only the weights in the head are initially being trained, the weights in the backbone must be frozen, and to make things easier, the Fastai library initializes the model with frozen weights in the backbone.

A learning rate finder function, introduced in the Fastai library, is then run for 100 iterations and plots a graph of loss against learning rates, a point around the steepest slope down in the direction of the minimum loss is selected as the maximum learning rate. Another option is to choose a

learning rate 10 times less than the minimum loss and see if it performs better.

The fit one cycle method is used during training to vary learning rates and momentum.

Progressive resizing

It is faster to train a model by passing through it larger numbers of smaller images at first and then scaling the network and training images up. Scaling up images and improving them from 64px by 64px to 128px by 128px is an easier task than executing that operation on a larger image and much quicker on a larger dataset. This is called Progressive Resizing, it can also help the model to generalize better as it sees many more different images and is less likely to experience overfitting.

The progressive resizing method is based on research made by Nvidia, though they used GANs.

The process is to train with small images in larger batches until the loss has decreased to an agreeable level. Following this, a new model that can accept larger images is created and then the learning was transferred from the model that trained on smaller images.

Note that Fastai enables an update of a model's initial data size, thus, keeping the weights and progress the model did to that point, using the command:
learn.data=new_data

As the image size increases, the batch size has to decrease in order to avoid running out of memory. That is because each batch will now contain larger images with four times as many pixels.

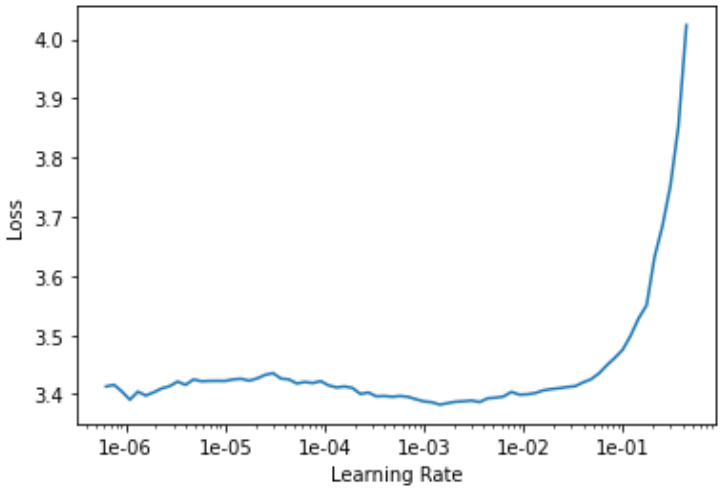
With each image size, training of one cycle of 10 epochs is executed. This is with the weights of the backbone frozen.

The image size is doubled and the model updated with the additional grid sizes for the path of larger images through the network, using the method by Fastai which I have mentioned above. It is important to

say that the number of weights does not change.

Unfreezing of the backbone

After the previous process and another fitting cycle has been done, the weights of the entire model are then unfrozen and the model will train again with discriminative learning rates. These learning rates are much smaller at the first third of the model, then increase in its second one, and increased even more in the head.



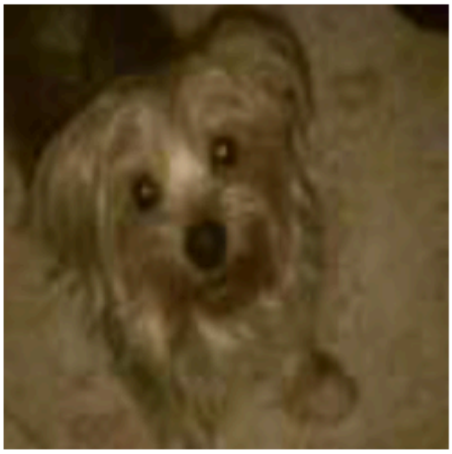
The learning rate finder will run again with the backbone and head unfrozen. Discriminative learning rates were used with values between 1e-6 and 1e-4. The learning rate in the head is still less in proportion to the previous cycle of learning, i.e, the previous *fit_one_cycle* function in the code, where only the head was unfrozen. This allows tweaking of the model without risking losing much of the accuracy already found. This method is known as learning rate annealing, a method in which the learning rate is reduced as we approach the ideal loss.

Continuing training on larger input images can highly improve the quality of the training, however, the batch size has to keep shrinking to fit within memory bounds, but nevertheless, the limit of Google Colab was reached in a way I could not continue the progressive resizing method, so I had to stay with the data I already updated.

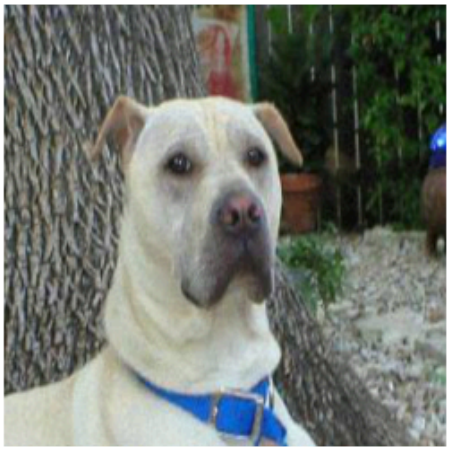
All of the training process was carried out on Google Colab, on my computer who possesses an Nvidia RTX-2060 and 16GB of RAM. All training took approximately 4-6 hours overall, but because Colab limits were reached a few times during training, my training was delayed for a long time, that is because I had to wait for at least 2 days (and sometimes 3 days) after each cycle of training before I could use Colab again.

Results - low resolution

Input / Prediction / Target



Input / Prediction / Target



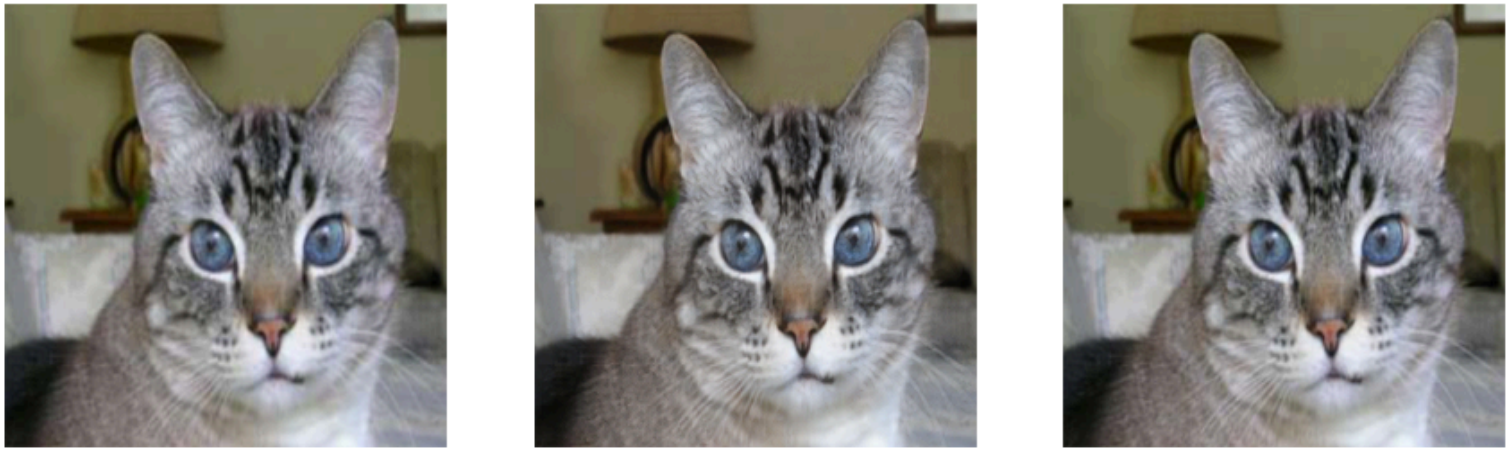
Input / Prediction / Target

Results - medium resolution

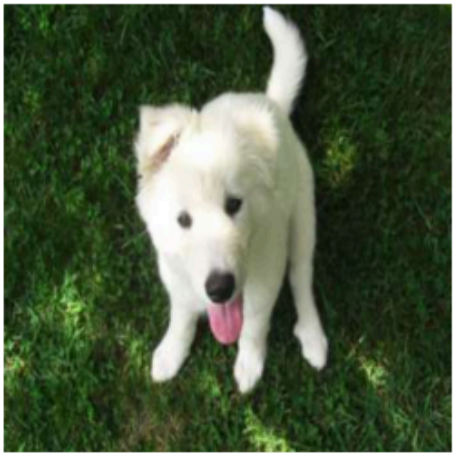
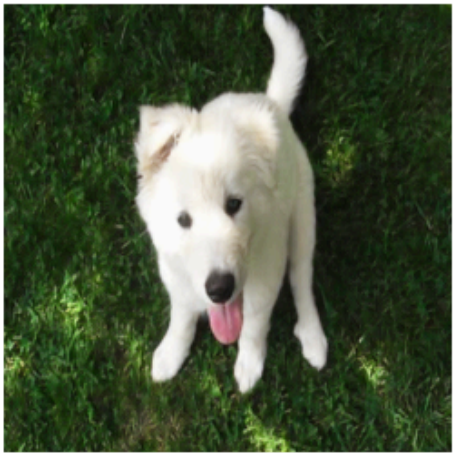
Input / Prediction / Target

Input / Prediction / Target

Input / Prediction / Target



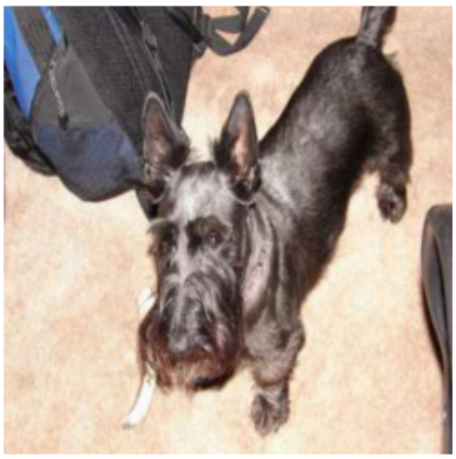
Input / Prediction / Target


Input / Prediction / Target



Input / Prediction / Target



Input / Prediction / Target

Input / Prediction / Target

Input / Prediction / Target

Results - Images not from the dataset

Top left - input of size (450, 450)
Top right - prediction of size (1280, 1280)
Bottom left - target of size (1512, 1512)

Top left - input of size (450, 450)

Top right - prediction of size (1280, 1280)

Bottom left - target of size (508,847)

Top left - input of size (450, 450)
Top right - prediction of size (1280, 1280)
Bottom left - target of size (514,550

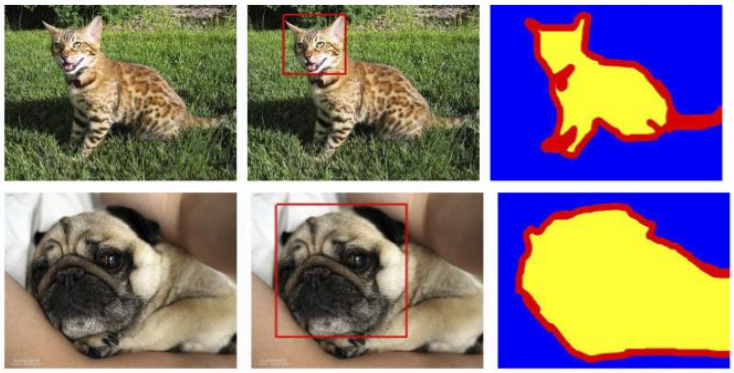
Architecture

1. Preparing and analyzing the data

The dataset I used is called "The Oxford-IIIT Pet Dataset" and it includes 37 categories of pets with roughly 200 images for each class. The images have large variations in scale, pose, and lighting. All images have an associated ground truth annotation of breed, head ROI, and pixel-level trimap segmentation.

The model created here seems to do a very good job creating fine details such as fur and eyes and other features.

Annotations examples:



Dataset statistics:

Breed	Count
American Bulldog	200
American Pit Bull Terrier	200
Basset Hound	200
Beagle	200
Boxer	199
Chihuahua	200
English Cocker Spaniel	196
English Setter	200
German Shorthaired	200
Great Pyrenees	200
Havanese	200
Japanese Chin	200
Keeshond	199
Leonberger	200
Miniature Pinscher	200
Newfoundland	196
Pomeranian	200
Pug	200
Saint Bernard	200
Samyoed	200
Scottish Terrier	199
Shiba Inu	200
Staffordshire Bull Terrier	189
Wheaten Terrier	200
Yorkshire Terrier	200
Total	4978

1.Dog Breeds

Breed	Count
Abyssinian	198
Bengal	200
Birman	200
Bombay	200
British Shorthair	184
Egyptian Mau	200
Main Coon	190
Persian	200
Ragdoll	200
Russian Blue	200
Siamese	199
Sphynx	200
Total	2371

2.Cat Breeds

Family	Count
Cat	2371
Dog	4978
Total	7349

3.Total Pets

The dataset itself is deployed using a function from the Fastai library called *untar_data()* which downloads a URL of the specified data set. The *untar_data()* is a thin wrapper for *FastDownload.get*. It downloads and extracts URL, by default to subdirectories of *~/.fastai*, and returns the path to the extracted data.

In addition, the data was augmented by a function I called *resize_one()*. This function resizes the images from the dataset to a

smaller size in order to train the model with the augmented images so it can compare its output to the original image.

The *get_data()* function is what sorts the data and organizes it in a way it can be passed to the model, i.e, it gets the images from the path of the dataset and the images after augmentation. It also uses *normalize()* function from the Fastai library in order to normalize the data.

2. Build and train deep learning model

The different layers used

The model consists of a total of 150 layers structured as a U-Net, with an encoder made of a pre-trained ResNet-34.

- Conv2d - Convolution layer
- BatchNorm2d - Batch normalization layer
- ReLU - A ReLU layer
- MaxPool2d - Max pooling layer
- PixelShuffel - A pixel shuffling layer
- ReplicationPad2d - A replication connection as mentioned above
- AvgPool2d - Average pooling layer
- MergeLayer - A merge layer

Conv2d

The conv2d is a convolutional layer. Convolutional layers are the main building blocks of convolutional neural networks (CNNs).

Convolution is the application of a filter matrix to an input matrix that results in activation. The filter matrix is smaller than the input matrix. The activation is the summing of the product of the filter matrix in each of the sub-matrixes of the input one. Repeated implementation of the same filter to an input results in a map of activations called feature map, indicating the locations and strength of a detected

0	1	2	3	3
1	3	1	0	0
3	2	2	1	3
2	2	0	0	2
1	0	0	0	2

X

2	1	0
0	2	2
2	1	0

=

17	12	12
19	17	10
14	6	9

feature in an input, like an image.

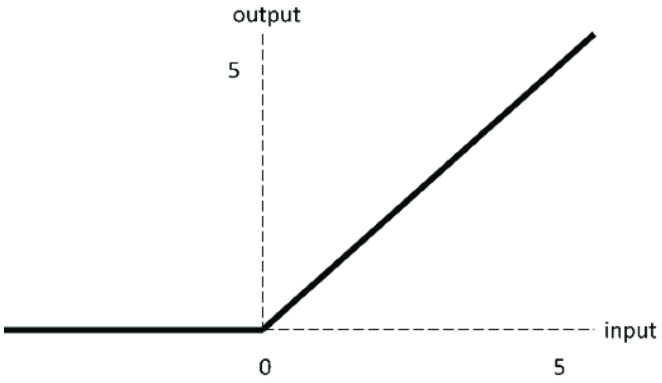
BatchNorm2d

Batch-Normalization is the process to make neural networks faster and more stable. This technique standardizes the inputs to a layer for each mini-batch. The BatchNorm2d layer performs the normalization and standardizing operations on the input of the layer, coming from a previous layer.

The issue batch-normalization can help solve is the distribution of the inputs to layers deep in the neural network may change after each mini-batch when the weights are updated. This change in the distribution of inputs to layers in the network is referred to as *"internal covariate shift"*.

ReLU

The ReLU, also known as Rectified Linear Activation Function, is a linear function that will output the input directly if it is possible, otherwise, it will output zero, i.e, the derivative for a positive input will be `1`, while the derivative for a negative input will be `0`.



The difference between ReLU and sigmoid or tanh is that these functions output `1` when given large values and for small values (less than zero) they output `0` or `-1` for sigmoid and tanh respectively. Moreover, these functions are pretty much sensitive to changes only around the mid-point of their inputs, such as `0.5` for sigmoid and `0.0` for tanh.

The simplest activation is therefore the ReLU function, that is because linear activations, where no transform to the input is applied (mostly) is very easy to train in a neural network. The problem with neural networks comprised mostly of linear functions like ReLU is that they cannot learn complex mapping functions.

In this model, the ReLUs functions are mostly there to update the weights where complex mapping isn't needed and to update the weights and produce an input

that did not change a lot and to prepare for more complex functions like the Conv2d. It also produces zero as output in areas this input isn't needed.

MaxPool2d

The max-pooling layer performs a pooling operation that calculates the maximum value (the largest value) in each sub-matrix of each feature map.

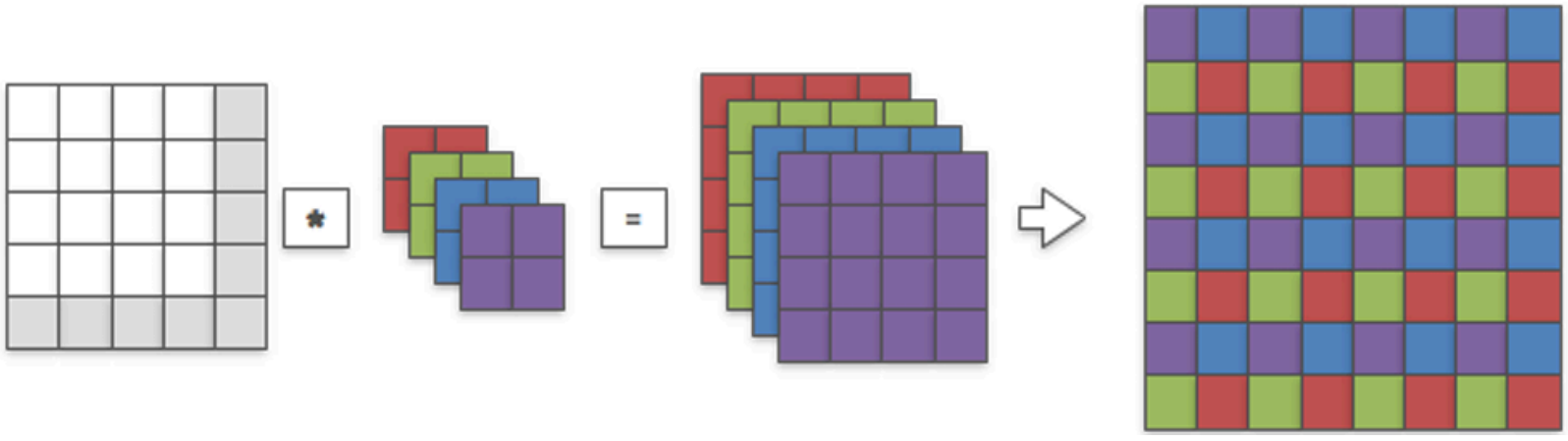
The results are essentially downsampled/pooled feature maps that pins the most present feature in the sub-matrix.

For example:

In this case, the sub-matrixes are defined with size 2 (filter 2) and they have a stride of 2. We can see that in the blue patch, the value of 9 was taken, in the cyan, it is 2, in the green is 6 and ib the red patch, the value of 3 was taken.

PixelShuffle

Pixel shuffle, also known as sub-pixel convolution, is essentially a convolution + shuffling.



This method is a process of the implementation of convolution in low-resolution space, followed by periodic shuffling operations. Sub-pixel convolution has the advantage over standard resize convolutions that, at the same computational complexity, it has more parameters and thus better modeling power.

It is important to note that pixel shuffle may suffer from tiled or checkerboard artifacts, but in order to overcome it, an average pooling layer is then executed.

ReplicationPad2d

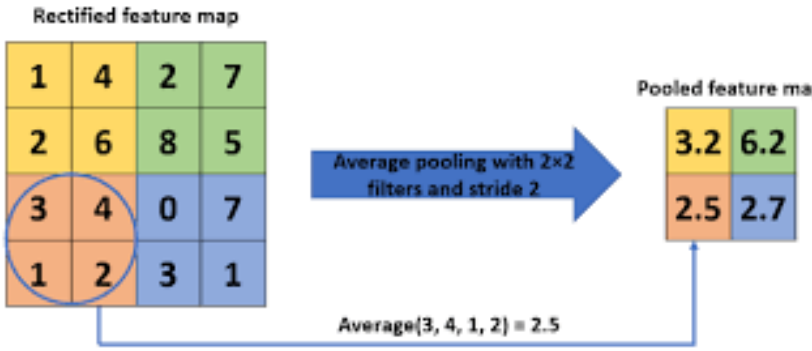
A replication padding is a padding added to an input image where values outside the boundary are set equal to the nearest image border value. It is useful when the areas near the border of the image are constant.

1	1	1	2	3	4	5	5	5
1	1	1	2	3	4	5	5	5
1	1	1	2	3	4	5	5	5
6	6	6	7	8	9	10	10	10
11	11	11	12	13	14	15	15	15
16	16	16	17	18	19	20	20	20
16	16	16	17	18	19	20	20	20
16	16	16	17	18	19	20	20	20

replicate padding

AvgPool2d

Average pooling is a pooling layer that involves calculating the average for each sub-matrix or patch of the feature map. This means that each patch of the feature map is downsampled to the average value in the sub-matrix.
That creates a feature map composed of the average value of all values in each patch.



MergeLayer

Merge layers are the layers that merge a shortcut with the result of the module by adding them or concatenating them.

Hyperparameters

During the training of the model, I used a number of functions from the Fastai library that help determine the best hyperparameters for the model to use.

A list of hyperparameters can be seen on the next page:

- wd - weight decay
- The base loss
- The learning rate
- pct_start
- Progressive resizing
- Number of epochs in each cycle
- Data augmentation
- bs - Batch Size

Weight decay

In more complex, deep, and big models, there is a risk of overfitting.

Overfitting is when a model learns the information and noise in the training to the point where it degrades the model's performance on fresh data (such as test/validation data). This is known as overfitting data. This means that the model picks up on noise or random fluctuations in the training data and learns them as ideas.

One way of dealing with overfitting is to add more data, but this can be costly, time-consuming, or even entirely out of our control, making it almost impossible in the short run.

That's where weight decay comes in. Weight decay is a regularization technique in deep learning. Weight decay works by adding a "penalty" term to the cost function of the neural network which has the effect of shrinking the weights during backpropagation (also known as backward propagation). This helps prevent the network from overfitting the training data as well as the exploding gradient problem.

Just to clarify, the exploding gradient problem describes a situation in the training of neural networks where the gradients used to update the weights grow exponentially. This prevents the backpropagation algorithm from making reasonable updates to the weights, and learning becomes unstable.

In neural networks, there are two parameters that can be regularized. Those are the weights and biases. The weights directly influence the relationship between the inputs and the outputs learned by the neural network because they are multiplied by the inputs. Mathematically, the biases only offset the relationship from the intercept. Therefore it is common that data scientists usually only regularize the weights.

In this model, I have used a parameter called wd, which represent the weight decay as seen in the following code block:

```
#wd is weight decay
wd = 1e-3
```

The *wd* parameter is set to `1e-3` (`0.001`) because the course at Fastai and a variety of articles have recommended a weight decay range of `0.1 - 0.001`.

We can see that when the pre-trained ResNet34 model is loaded to a U-Net architecture we can set a variable inside the *unet_learner* function which is named *wd* as the *wd* created by me, i.e, `1e-3` as seen in the code block below:

```
#learn is the U-Net shaped model
containing the pre-trained
ResNet34
learn = unet_learner(data, arch,
wd=wd, loss_func=feat_loss,
callback_fns=LossMetrics,
                     blur=True,
norm_type=NormType.Weight)
```

The base loss

In the base loss, I could have used either the L1 norm loss function or the Mean Squared error. The results should be almost identical.

The L1 norm loss is also known as the Least Absolute Deviations (LAD), and Least Absolute Errors (LAE).

It is basically minimizing the sum of the absolute differences (S) between the target value (y_i) and the predicted values ($\hat{y}_i$) as seen in the equation below:

$$S = \sum_{i=1}^{n} \left| y_i - \hat{y}_i \right| = L1$$

The Mean Squared error (MSE) loss function is calculated as the average of the squared differences between the ground truth (y_i) and the model's predictions ($\hat{y}_i$) and. The mean squared error is perhaps the simplest and most common loss function.

$$MSE = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

The result is always positive regardless of the sign of the predicted and actual values and the perfect value `0.0`. The loss value is minimized, although it can be used in a maximization optimization process by making the score negative.

This model uses the L1 norm loss function as its base loss function because I did not have enough time to try the mean squared error loss function as well.
The implementation of the L1 loss function in this model is using the *functional* (*F*) class from the PyTorch library that contains

a lot of different methods and functions such as pooling functions, convolutional functions, and the L1 norm loss function. This can be seen in the code block below as the variable *base_loss*:

```
#the base_loss will be used to
compare the pixels and the
features
#I'm using an L1 loss function
base_loss = F.l1_loss
```

The base_loss, i.e, the L1 norm loss is used in the class *FeatureLoss* inside the *forward()* function:

```
#feat_losses calculates the L1
loss between the pixels
    self.feat_losses =
[base_loss(input,target)]
    self.feat_losses +=
[base_loss(f_in, f_out)*w
                          for
f_in, f_out, w in zip(in_feat,
out_feat, self.wgts)]

    #calculates the L1 loss of the
features of all those layers.
Basically I'm going through every
one
    #of these end of each block
and grabbing the activations and
getting the L1 loss on each one
    self.feat_losses +=
[base_loss(gram_matrix(f_in),
gram_matrix(f_out))*w**2 * 5e3
                          for
f_in, f_out, w in zip(in_feat,
out_feat, self.wgts)]
```

The learning rate

The learning rate is the most famous hyperparameter. It controls how much to change the model in response to the estimated error each time the model's weights are updated. Choosing the learning rate may prove challenging as a value too small may result in a long training process that has the risk of getting stuck, whereas a value too big may result in inaccurate predictions, learning of a sub-optimal set of weights too fast, or an unstable training process.

As said above, the learning rate may be the most important hyperparameter when configuring a neural network. Therefore, it is vital to know how to investigate the effects of the learning rate on model performance and to build an intuition about the dynamics of the learning rate on model behavior.

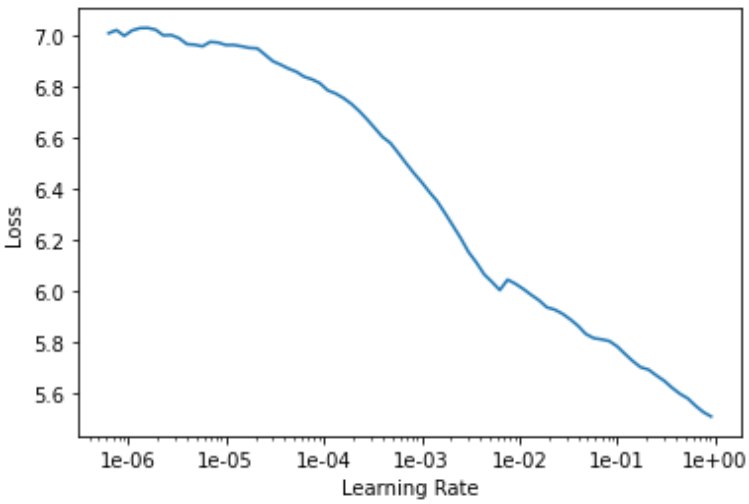
In this model, I used the function *lr_find()* from the Fastai library that iterates 100 times on default on the model, in each iteration the learning rate changes. Then, by using the function *recorder.plot()* I can plot a graph of the loss vs the learning rate

to help find which learning rate causes the steepest part of the loss. This can be seen in the below code block:

```
#.lr_find explores learning rate
from start_lr (default 1e-07) to
end_lr (default 10)
#over num_it iterations (default
100)
learn.lr_find()

#shows a graph of the loss versus
the learning rate
learn.recorder.plot()
```

This block outputs the graph mentioned above:



The steepest point is around the value of 1e-3 (0.001) so I chose this as the base learning rate and named it *lr*:

```
#the learning rate I chose by
searching the steepest loss in the
learn.recorder.plot()
lr = 1e-3
```

Now, after the optimal learning rate is found, I will use the discriminative learning rates method mentioned before to help the training process to be faster and more accurate than using a fixed learning rate.

To actually use the discriminative learning rates method, I pass the *slice()* function as an input named *lrs* to another function I created named *do_fit()* that utilizes the *fit_one_cycle()* function from the Fastai library. In the Fastai library, when passing to the *fit_one_cycle()* a *slice()* object, it will give every layer in the model a learning rate from a given number (default is 0) to an end given number which I decided to be as default the *lr* I found using the *recorder.plot()* function, as seen in the code block below:

```
#fits one cycle, saves and show
the results (because I'm using a
pre-trained model with frozen
layers)
def do_fit(save_name,
lrs=slice(lr), pct_start=0.9):
  #uses 1cycle policy with an
optimum learning rate
  learn.fit_one_cycle(10, lrs,
pct_start=pct_start)
```

In the training process, I pass the learning rates as seen below in the code blocks:

```
#first training cycle, called '1a' and passed a slice of 10 times lr
do_fit('1a', slice(lr*10))
```

```
#second training cycle, called '1b'. here I pass 2 numbers in the slice, that means it will give #every layer an lr from 1e-5 to lr (1e-3) in equal spacing
do_fit('1b', slice(1e-5,lr))
```

```
#the third training cycle called '2a'. Here I use the default learning rate (slice(1e-03))
do_fit('2a')
```

```
#the fourth training cycle called '2b'
do_fit('2b', slice(1e-6,1e-4), pct_start=0.3)
```

Now let's look at some of the results from the '2a' training cycle:

epoch	train_loss	valid_loss	pixel	feat_0	feat_1	feat_2	gram_0	gram_1	gram_2	time
0	2.215194	2.168084	0.163477	0.258612	0.291046	0.152116	0.368955	0.574025	0.359853	08:03
1	2.170885	2.139781	0.163977	0.258518	0.289441	0.150782	0.356822	0.564969	0.355271	07:59
2	2.163428	2.112683	0.164728	0.257489	0.286832	0.148965	0.347928	0.555958	0.350783	07:59
3	2.154448	2.095722	0.164792	0.256066	0.284631	0.147561	0.343588	0.551414	0.347669	07:59
4	2.121970	2.086430	0.167024	0.257047	0.284054	0.147516	0.336542	0.547835	0.346412	08:01
5	2.100517	2.071818	0.165765	0.256234	0.282684	0.146635	0.331247	0.545091	0.344162	08:00
6	2.087619	2.067739	0.166584	0.255854	0.281932	0.145929	0.330857	0.543705	0.342877	08:01
7	2.088696	2.057490	0.165154	0.255099	0.281127	0.145793	0.327521	0.541064	0.341732	08:02
8	2.075498	2.049480	0.164743	0.254181	0.279692	0.144738	0.326859	0.539355	0.339911	08:02
9	2.039007	2.042586	0.166636	0.254656	0.279737	0.144465	0.321809	0.536040	0.339243	08:01

Now let's see an example of a prediction made by the model after '2a' cycle ended:

Input / Prediction / Target

And now the results from the '2b' training cycle:

epoch	train_loss	valid_loss	pixel	feat_0	feat_1	feat_2	gram_0	gram_1	gram_2	time
0	2.051592	2.038423	0.165402	0.253982	0.279333	0.144219	0.321916	0.534866	0.338704	08:16
1	2.053264	2.036693	0.165037	0.253728	0.279132	0.143980	0.322056	0.534803	0.337957	08:16
2	2.062191	2.034981	0.165631	0.253748	0.278699	0.143813	0.321741	0.533795	0.337555	08:16
3	2.036775	2.034073	0.165713	0.254322	0.279211	0.143637	0.320134	0.534451	0.336604	08:16
4	2.054596	2.033378	0.166107	0.254611	0.279028	0.143659	0.319415	0.533858	0.336701	08:16
5	2.041799	2.029701	0.165513	0.253956	0.278439	0.143287	0.319455	0.533092	0.335958	08:16
6	2.034605	2.029416	0.165471	0.253797	0.278323	0.143280	0.319971	0.532485	0.336089	08:16
7	2.040275	2.027777	0.165377	0.253652	0.278276	0.143189	0.319177	0.532209	0.335899	08:17
8	2.037214	2.027074	0.165194	0.253525	0.278245	0.143093	0.319043	0.532189	0.335785	08:16
9	2.040682	2.027674	0.165436	0.253592	0.278355	0.143246	0.318879	0.532244	0.335922	08:17

And an example of a prediction made by the model after '2b' cycle ended:

pct_start

pct_start is a hyperparameter used in the model. It refers to the percentage of the total number of iterations where the learning rate rises during one cycle.

For example, giving the default of 0.3 means that the *lr* is going up for 30% of your iterations of the model, and then decreasing over the last 70%.

In the model, I use a *pct_start* of 0.9, so the *lr* will rise for 90% of the iterations and then decrease for the last 10% because I am using a very small-scaled *lr* (of 1e-3) and so, by raising the lr for 90%, I avoid overfitting and slow learning.

This is how I use the *pct_start* at the training section of the model:

```
#fits one cycle, saves and show the results (because I'm using a pre-trained model with frozen layers)
def do_fit(save_name, lrs=slice(lr), pct_start=0.9):
  #uses 1cycle policy with an optimum learning rate
  learn.fit_one_cycle(10, lrs, pct_start=pct_start)
```

Progressive resizing as hyperparameter

Progressive resizing has already been explained earlier but note that it can be a hyperparameter as well.

By deciding how to change the sizes and batch size, the learning process of the model can result in quite different outputs.

I chose to decrease the batch size to 12 and to upscale the size by 2 in order I won't run out of memory because of the increased size.

This can be seen in this code block:

```
#this is the progressive resizing;I doubled the size, to let the model train on better resolution
#images, so in order I won't run out of memory, I halved the batch size
data = get_data(12,size*2)

#updating the model's (learn) data to the new databunch
learn.data = data
```

Number of epochs in each cycle

The number of epochs in each cycle can be changed in the *do_fit()* function.

The number of epochs can change the results of the model because it can add or decrease learning. By having more epochs, the model can learn more but having too many epochs can cause overfitting of the training dataset whereas too few epochs can result in an underfit model.

The number of epochs can be determined as seen in the following code block:

```
#fits one cycle, saves and show the results (because I'm using a pre-trained model with frozen layers)
def do_fit(save_name, lrs=slice(lr), pct_start=0.9):
  #uses 1cycle policy with an optimum learning rate
  learn.fit_one_cycle(10, lrs, pct_start=pct_start)
```

I chose the number 10 because the model is very deep so each epoch takes a long time to complete (8 minutes) and so, if I would have chosen a higher number it would take the model a long time to complete training. Less than 10 epochs would probably cause the model to be underfitted.

Data augmentation as hyperparameter

The data augmentation is a general hyperparameter which I have explained before in the 'training data' section but now I will try to explain why it is a hyperparameter, i.e, a parameter that can affect the learning of the model.

By using different methods of image augmentation, I can not only expand my dataset, but I can also give the model new functionalities.

For example, if a function for image augmentation is created, such as the one used to resize the images in the dataset for this model as seen in the next code block:

```
#the basic augmentation function I used to "crapify" the images and resize them into
#smaller images for the training process
def resize_one(fn, i, path, size):
  #destination path
  dest = path/fn.relative_to(path_hr)
  dest.parent.mkdir(parents=True, exist_ok=True)

  #the image that I want to resize
  img = PIL.Image.open(fn)

  #target size
  targ_sz = resize_to(img, size, use_min=True)

  #the iage after the resize
  img = img.resize(targ_sz, resample=PIL.Image.BILINEAR).convert('RGB')

  #saves the image
  img.save(dest, quality=60)
```

We can make it not only able to resize images but also to add noises, graining, random text, holes, and more to images and thus, train the model to be able to fix noises and graining, remove random texts and fill holes accurately.

Thus, the model will train differently and would be able to have more functionalities and produce different outputs. It will possess an entirely different training process just by adjusting the data augmentation function according to the desired output.

bs - Batch size

Batch size is an important hyperparameter that influences the dynamics of the learning algorithm.

It is common to use large batch sizes to train neural networks as it allows computational speedups from the parallelism of GPUs. However, it is well known that too large of a batch size will lead to poor generalization (although currently, it is unknown why this is so).

For example, using a batch size equal to the entire dataset guarantees convergence to the global optimum of the objective function. However, this is at the cost of slower convergence to that optimum. On the other hand, using smaller batch sizes has been shown to have faster convergence to relatively good solutions.

That can be explained by the fact that smaller batch sizes allow the model to start learning before having to "see" all the data. The downside of using smaller batch sizes is that the model is not guaranteed to converge to the global optimum.

The way this model is using batch size can be seen in the next code block:

```
#stands for batch size and, well,
size (of image)
bs,size=32,128

#a function that collects the data
from ImageNet via its path, and
the variables I've created
beforehand
def get_data(bs,size):
  #the databunch itself (of type
ImageDataBunch)
  data =
(src.label_from_func(lambda x:
path_hr/x.name)

.transform(get_transforms(max_zoom
=2.), size=size, tfm_y=True)

.databunch(bs=bs).normalize(imagen
et_stats, do_y=True))

#the collection of all the data
after collecting and sorting it
data = get_data(bs,size)
```

In this model, I used a starting *bs* of 32 because this model is dealing with large images. Later I lower the batch size as I increase the size of the images.

3. Software deployment

The GUI that uses the model creates a new model (learn) in a shape of a U-Net with a ResNet34 inside and then loads the saved weights from the training process ('2b'). Thus, the software contains the fully-trained model and can now pass an image through it to get an upscaled version of it.

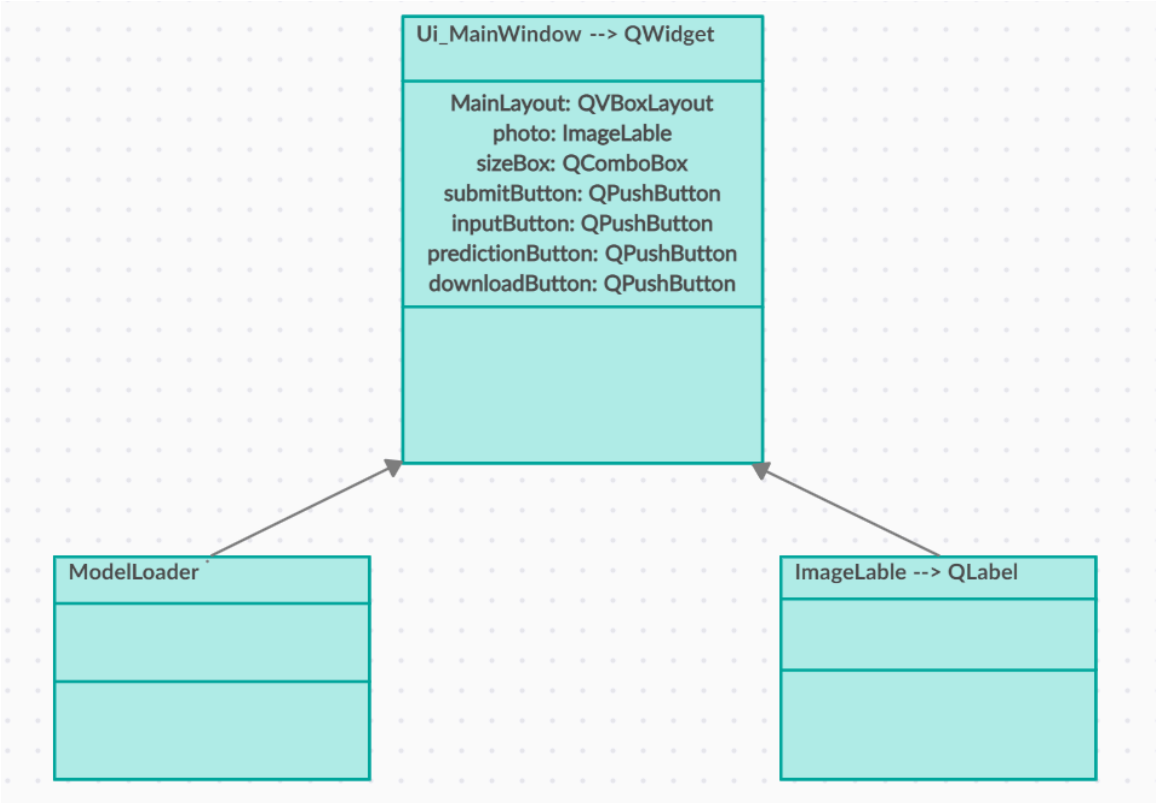
The user drags an image to the application, chooses a size for the desired prediction image through a drop-down menu, and then press the 'submit' button.

```
def dropEvent(self, event):
  if event.mimeData().hasImage:

event.setDropAction(Qt.CopyAction)
    #self.file_path is the input
image path
    self.file_path =
event.mimeData().urls()[0].toLocal
File()
    input_img =
ImageQt(self.file_path)
    pixmap =
QPixmap.fromImage(input_img)
    pixmap = pixmap.scaled(500,
500, QtCore.Qt.KeepAspectRatio)
    self.photo.setPixmap(pixmap)

    event.accept()
else:
    event.ignore()
```

UML diagram of the GUI:



Developer's guide

Imports

```
#all of the imports (I didn't even used most, but just in case)
import os
import time
from PIL import Image
import numpy as np
import tensorflow as tf
import tensorflow_hub as hub
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchvision import datasets, transforms
from torch.autograd import Variable
from os.path import exists
```

Superresolution

Core imports

```
#the core imports of the Fastai library
import fastai
from fastai.vision import *
from fastai.callbacks import *
from fastai.utils.mem import *

from torchvision.models import vgg16_bn
```

Data paths

```
#the path of my DataSet taken from ImageNet
path = untar_data(URLs.PETS)

#the path of the high resolution images
path_hr = path/'images'

#the path of the low resolution images
path_lr = path/'small-96'

#the path of the medium resolution images
path_mr = path/'small-256'
```

```
#stands for 'image list'. uses ImageList.from_folder(PATH) which gets the
the list
#of files in PATH that have an image suffix
il = ImageList.from_folder(path_hr)
```

Data augmentation

```
#the basic augmentation function I used to "crapify" the images and resize
them into
#smaller images for the training process
def resize_one(fn, i, path, size):
  #destination path
  dest = path/fn.relative_to(path_hr)
  dest.parent.mkdir(parents=True, exist_ok=True)

  #the image that I want to resize
  img = PIL.Image.open(fn)

  #target size
  targ_sz = resize_to(img, size, use_min=True)

  #the iage after the resize
  img = img.resize(targ_sz, resample=PIL.Image.BILINEAR).convert('RGB')

  #saves the image
  img.save(dest, quality=60)
```

```
#create smaller image sets the first time this nb is run, using the
function "resize_one()"

#the sets of image which I will resize for training
sets = [(path_lr, 96), (path_mr, 256)]
for p,size in sets:
  if not p.exists():
    print(f"resizing to {size} into {p}")
    parallel(partial(resize_one, path=p, size=size), il.items)
```

```
#stands for batch size and, well, size (of image)
bs,size=32,128

#stands for architecture. This is the pre-traind ResNet34 from ImageNet
arch = models.resnet34

#an ItemList suitable for image to image tasks
src = ImageImageList.from_folder(path_lr).split_by_rand_pct(0.1, seed=42)
```

Data collection and arrangement

```
#a function that collects the data from ImageNet via its path, and the
variables I've created beforehand
#the function creates a DataBunch object
def get_data(bs,size):
  #the databunch itself (of type ImageDataBunch)
  data = (src.label_from_func(lambda x: path_hr/x.name)
          .transform(get_transforms(max_zoom=2.), size=size, tfm_y=True)
          .databunch(bs=bs).normalize(imagenet_stats, do_y=True))

  data.c = 3
  return data
```

```
#the collection of all the data after collecting and sorting it
data = get_data(bs,size)
```

```
#an example to check if everyting is fine with the data
data.show_batch(ds_type=DatasetType.Valid, rows=2, figsize=(9,9))
```

Crappy images (left) and original images (right):

Feature Loss

Gram matrix

```
#creates a gram matrix for the loss function further on
def gram_matrix(x):
  n,c,h,w = x.size()
  x = x.view(n, c, -1)
  return (x @ x.transpose(1,2))/(c*h*w)
```

Base loss

```
#the base_loss will be used to compare the pixels and the features
#I'm using an L1 loss function
base_loss = F.l1_loss
```

Pre-trained VGG-16 based loss function

```
#I only want the convolutional part of the VGG model
#the VGG.features attribute gives me exactly that
vgg_m = vgg16_bn(True).features.cuda().eval()
#I'm turning off the "requires_grad" because I don't want to update the
weights of this model, I just want it for the loss
requires_grad(vgg_m, False)
```

```
#I will enumerate through all the children of the VGG model to find all of
the max pooling layers, because
#that's where the grid size changes
#the layer i-1 is the layer before the grid size changes
#so blocks is the list of layer numbers just before the max pooling layers
blocks = [i-1 for i,o in enumerate(children(vgg_m)) if
isinstance(o,nn.MaxPool2d)]
blocks, [vgg_m[i] for i in blocks]
```

```
([5, 12, 22, 32, 42],
 [ReLU(inplace=True),
  ReLU(inplace=True),
  ReLU(inplace=True),
  ReLU(inplace=True),
  ReLU(inplace=True)])
```

FeatureLoss function

```
#the class FeatureLoss is my loss function for the model. It creates a
loss function made from a combination
#of the VGG16 model pre-trained on ImageNet, gram matrix and L1 loss
function
#note that this is perceptual loss function
class FeatureLoss(nn.Module):
  def __init__(self, m_feat, layer_ids, layer_wgts):
    super().__init__()

    #m_feat is the pre-traind model (VGG16) which contains the features
which I want to generate for the feature loss
    self.m_feat = m_feat

    #loss_features are all of the layers from the pre-traind model from
which I want the features to create the losses
    self.loss_features = [self.m_feat[i] for i in layer_ids]

    #my hooked outputs (because that's how to grab intermediate layers
with pytorch)
    self.hooks = hook_outputs(self.loss_features, detach=False)
    self.wgts = layer_wgts
    self.metric_names = ['pixel',] + [f'feat_{i}' for i in
range(len(layer_ids))
          ] + [f'gram_{i}' for i in range(len(layer_ids))]

  #make_features finds the features of a given image (x) and passes it
through the VGG model which contain only
  #the convolutional part of the pre-trained VGG model
  def make_features(self, x, clone=False):
    #passing the given input (x) through the VGG model
    self.m_feat(x)

    #a copy of all the stored activations
    return [(o.clone() if clone else o) for o in self.hooks.stored]

  #the forward function of the perceptual loss, where we calculates and
get the features of the target (y)
```

```
  #in order to create the features of the output (ŷ)
  def forward(self, input, target):
    #in out_feat i'm passing the target (y) through the VGG's stored
activations and grab a copy of them,
    #thus, getting the features of the target
    out_feat = self.make_features(target, clone=True)

    #in in_feat i'm doing almost the same thing, i.e, getting the features
of the input, so that the
    #output of the generator (ŷ)
    in_feat = self.make_features(input)

    #feat_losses calculates the L1 loss between the pixels
    self.feat_losses = [base_loss(input,target)]
    self.feat_losses += [base_loss(f_in, f_out)*w
                         for f_in, f_out, w in zip(in_feat, out_feat,
self.wgts)]

    #calculates the L1 loss of the features of all those layers. Basically
I'm going through every one
    #of these end of each block and grabbing the activations and getting
the L1 loss on each one
    self.feat_losses += [base_loss(gram_matrix(f_in),
gram_matrix(f_out))*w**2 * 5e3
                         for f_in, f_out, w in zip(in_feat, out_feat,
self.wgts)]

    #.metrics prints out all of the seperate layer loss amounts
    self.metrics = dict(zip(self.metric_names, self.feat_losses))
    return sum(self.feat_losses)

  def __del__(self): self.hooks.remove()
```

```
#this is the feature loss i'll be using in the model. Created using the
#FeatureLoss class presented before
feat_loss = FeatureLoss(vgg_m, blocks[2:5], [5,15,2])
```

Train

Initialization of the training model (learn)

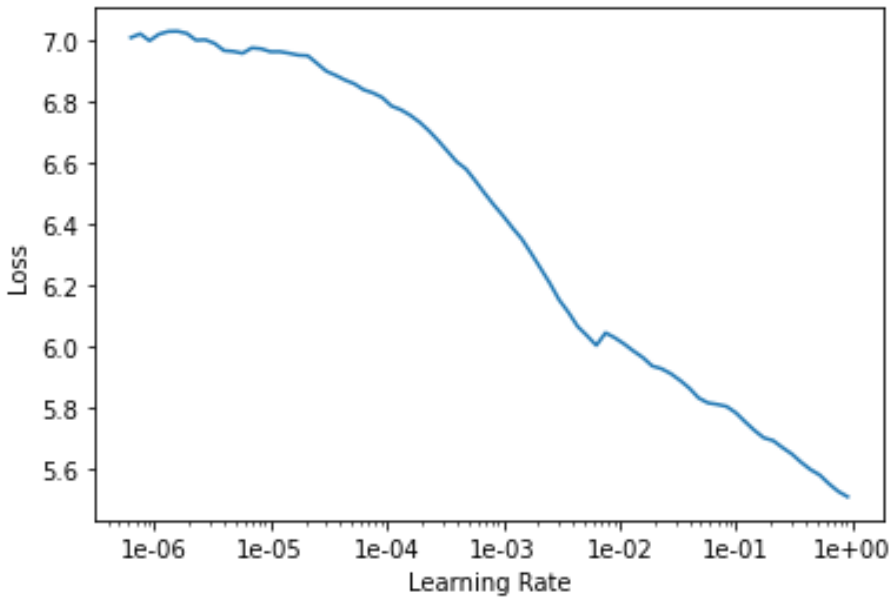
```
#wd is weight decay
wd = 1e-3

#learn is the U-Net shaped model containing the pre-trained ResNet34
learn = unet_learner(data, arch, wd=wd, loss_func=feat_loss,
callback_fns=LossMetrics,
                     blur=True, norm_type=NormType.Weight)
gc.collect();
```

```
#shows all layers in the model (learn)
learn.summary()
```

```
#.lr_find explores learning rate from start_lr (default 1e-07) to end_lr
(default 10)
#over num_it iterations (default 100)
learn.lr_find()

#shows a graph of the loss versus the learning rate
learn.recorder.plot()
```



```
#check the path of the model (learn)
learn.path
```

```
PosixPath('/root/.fastai/data/oxford-iiit-pet/small-96')
```

```
#the location of the current working directory
print(os.getcwd())
```

```
/content
```

```
#changes the path of the model if it is not saved on google drive

#IMPORTANT -----> must run so the model can be saved and/or loaded
if 'google.colab' in str(get_ipython()):
  #the google drive path
  temp_path = Path('/content/drive/MyDrive/Deep_Learning/models/')

else:
  #the current working directory
  temp_path = Path(os.gercwd())

learn.path = temp_path
```

```
#the learning rate I chose by searching the steepest loss in the
learn.recorder.plot()
lr = 1e-3
```

Training process (Cycle fitting)

```
#fits one cycle, saves and show the results (because I'm using a
pre-trained model with frozen layers)
#I pass a slice object because in Fastai, it will give every layer in the
model an lr from 0 to the
#number specified in equal spacing or if given 2 numbers inside the slice,
it will give every layer
#an lr from the first number to the second number also in equal spacing
def do_fit(save_name, lrs=slice(lr), pct_start=0.9):
  #uses 1cycle policy with an optimum learning rate
  learn.fit_one_cycle(10, lrs, pct_start=pct_start)

  #saves the weights of the current training cycle in a .pth file with the
given name
  learn.save(save_name)

  #.show_results() shows some predictions
  learn.show_results(rows=1, imgsize=5)
```

```
#first training cycle, called '1a' and passed a slice of 10 times lr
do_fit('1a', slice(lr*10))
```

```
#unfreezes the weights of the arch (the pre-trained ResNet)
learn.unfreeze()
```

```
#second training cycle, called '1b'. here I pass 2 numbers in the slice,
that means it will give
#every layer an lr from 1e-5 to lr (1e-3) in equal spacing
do_fit('1b', slice(1e-5,lr))
```

```
#this is the progressive resizing;I doubled the size, to let the model
train on better resolution
#images, so in order I won't ran out of memory, I halved the batch size
data = get_data(12,size*2)
```

```
#updating the model's (learn) data to the new databunch
learn.data = data

#freezes the weights of the arch again
learn.freeze()
gc.collect()
```

```
#the third training cycle called '2a'. Here I use the default learning
rate (slice(1e-03))
do_fit('2a')
```

epoch	train_loss	valid_loss	pixel	feat_0	feat_1	feat_2	gram_0	gram_1	gram_2	time
0	2.215194	2.168084	0.163477	0.258612	0.291046	0.152116	0.368955	0.574025	0.359853	08:03
1	2.170885	2.139781	0.163977	0.258518	0.289441	0.150782	0.356822	0.564969	0.355271	07:59
2	2.163428	2.112683	0.164728	0.257489	0.286832	0.148965	0.347928	0.555958	0.350783	07:59
3	2.154448	2.095722	0.164792	0.256066	0.284631	0.147561	0.343588	0.551414	0.347669	07:59
4	2.121970	2.086430	0.167024	0.257047	0.284054	0.147516	0.336542	0.547835	0.346412	08:01
5	2.100517	2.071818	0.165765	0.256234	0.282684	0.146635	0.331247	0.545091	0.344162	08:00
6	2.087619	2.067739	0.166584	0.255854	0.281932	0.145929	0.330857	0.543705	0.342877	08:01
7	2.088696	2.057490	0.165154	0.255099	0.281127	0.145793	0.327521	0.541064	0.341732	08:02
8	2.075498	2.049480	0.164743	0.254181	0.279692	0.144738	0.326859	0.539355	0.339911	08:02
9	2.039007	2.042586	0.166636	0.254656	0.279737	0.144465	0.321809	0.536040	0.339243	08:01

```
#unfreezes the weights in the pre-trained arch
learn.unfreeze()
```

```
#the fourth training cycle called '2b'
do_fit('2b', slice(1e-6,1e-4), pct_start=0.3)
```

epoch	train_loss	valid_loss	pixel	feat_0	feat_1	feat_2	gram_0	gram_1	gram_2	time
0	2.051592	2.038423	0.165402	0.253982	0.279333	0.144219	0.321916	0.534866	0.338704	08:16
1	2.053264	2.036693	0.165037	0.253728	0.279132	0.143980	0.322056	0.534803	0.337957	08:16
2	2.062191	2.034981	0.165631	0.253748	0.278699	0.143813	0.321741	0.533795	0.337555	08:16
3	2.036775	2.034073	0.165713	0.254322	0.279211	0.143637	0.320134	0.534451	0.336604	08:16
4	2.054596	2.033378	0.166107	0.254611	0.279028	0.143659	0.319415	0.533858	0.336701	08:16
5	2.041799	2.029701	0.165513	0.253956	0.278439	0.143287	0.319455	0.533092	0.335958	08:16
6	2.034605	2.029416	0.165471	0.253797	0.278323	0.143280	0.319971	0.532485	0.336089	08:16
7	2.040275	2.027777	0.165377	0.253652	0.278276	0.143189	0.319177	0.532209	0.335899	08:17
8	2.037214	2.027074	0.165194	0.253525	0.278245	0.143093	0.319043	0.532189	0.335785	08:16
9	2.040682	2.027674	0.165436	0.253592	0.278355	0.143246	0.318879	0.532244	0.335922	08:17

```
#export the full model

#NOTE --> the .export() is not very effective and thus, i'm not loading
the model using it
learn.export('weights.pkl')
```

Test

Initialization of the test model (learn)

```
#create an initial learner that is of None type
learn = None
gc.collect();
```

```
#the size of the output image if the memory is less than 8GB (820, 820)
--> changed to 1024
256/320*1024
```

```
819.2
```

```
#the size of the output image if the memory is greater than 8GB (1280,
1280)
256/320*1600
```

```
1280.0
```

```
#the free memory
free = gpu_mem_get_free_no_cache()

# the max size of the test image depends on the available GPU RAM
if free > 8000:
  size=(1280, 1280) # >  8GB RAM --> size=(1280, 1600)

else:
  size=(1024, 1024) # <= 8GB RAM --> size=(820, 1024)

print(f"using size={size}, have {free}MB of GPU RAM free")
```

```
using size=(1280, 1280), have 13725MB of GPU RAM free
```

```
#the initialization of the model (learn) for the sake of validatin/testing
learn = unet_learner(data, arch, loss_func=F.l1_loss, blur=True,
norm_type=NormType.Weight)
```

```
#instead of having a low-resolution data for the initialization of the
model, I want to see what happens
#when initialized with the medium-resolution images. This part is not
essential for the sake of the predictions
data_mr = (ImageImageList.from_folder(path_mr).split_by_rand_pct(0.1,
seed=42)
          .label_from_func(lambda x: path_hr/x.name)
          .transform(get_transforms(), size=size, tfm_y=True)
          .databunch(bs=1).normalize(imagenet_stats, do_y=True))
data_mr.c = 3
```

```
#again, changing the path of the model (learn) to a google drive path

#IMPORTANT -----> must run so the model can be saved and/or loaded
if 'google.colab' in str(get_ipython()):
  #the google drive path
  temp_path = Path('/content/drive/MyDrive/Deep_Learning/models/')

else:
  #the current working directory
  temp_path = Path(os.gercwd())

learn.path = temp_path
```

```
#loading the latest training cycle weights to the model
learn.load('2b');
```

```
#updating the model's data
learn.data = data_mr
```

Predictions

```
#the path of a certain image from the dataset if I want to check from
there
fn = data_mr.valid_ds.x.items[0]; fn
```

```
#the input image (y). This image was resized from the original size of
(1512, 1512) to a size of (450, 450)
img =
open_image('/content/drive/MyDrive/Deep_Learning/my_data/img-billy2-450x45
0jpg-450x450'); img.shape
```

```
torch.Size([3, 450, 450])
```

```
#the prediction

#NOTE --> the only important variable here is the img_hr which is the
tensor of the prediction image (ŷ)
p,img_hr,b = learn.predict(img)
```

```
#shows the input (x) image (of size (450, 450))
show_image(img, figsize=(18,15), interpolation='nearest');
```

```
#shows the output (ŷ) image (size of (1280, 1280))
Image(img_hr).show(figsize=(18,15))
```

```
#checking the shape of the output image
print(img_hr.shape)
```

```
torch.Size([3, 1280, 1280])
```

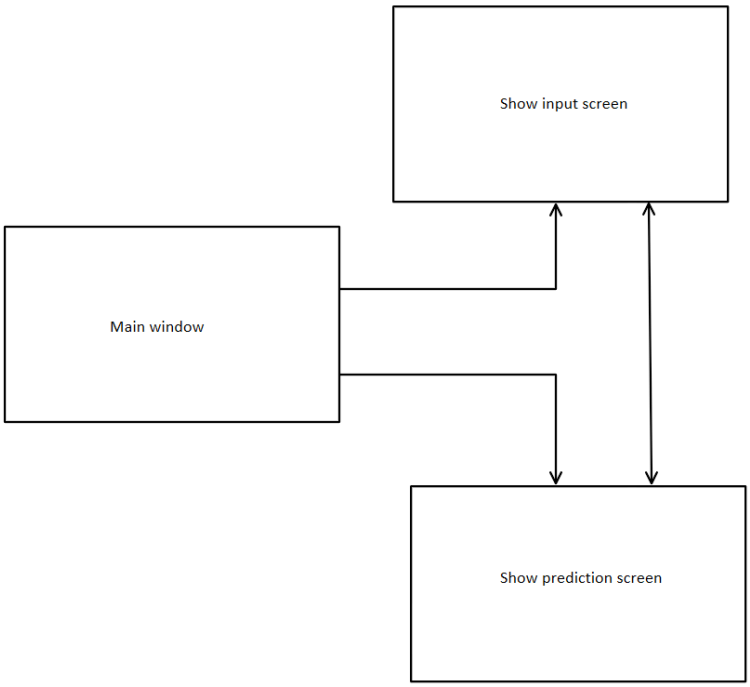
```
#the original image of size (1512, 1512), taken from my phone
```

```
img_origin =
open_image('/content/drive/MyDrive/Deep_Learning/my_data/billy2.jpeg');
img_origin.shape
```

```
torch.Size([3, 1512, 1512])
```

```
#shows the original image of size (1512, 1512)
show_image(img_origin, figsize=(18,15), interpolation='nearest');
```

User's guide



The main window is the starting window where the user is asked to drag an image to a specific place and to choose the size of the input.

The prediction window is the same as the main one, but you can see the upscaled image inside the dashed border.

The input window does the same but shows the input image.

The main window contains a QLabel where the different images can be seen, a drop-down menu to choose a desired size, a submit button to submit the size for evaluation, an input button to see the input, a prediction button to see the output, and a download button to download the upscaled image.

Reflection

In the beginning, when I just started the project, I had a hard time deciding on a topic of my interest. I started with different ideas and decided on trying to find the possible presence of black holes in globular clusters.

I even created an image dataset containing images of globular star clusters with block holes in them and clusters without black holes.

Because the images were all too similar and because we cannot know for sure if there aren't some black holes in the clusters I labeled as 'without black hole' and because for good accuracy I needed more parameters I couldn't achieve, I decided to switch an idea.

I had more ideas but eventually, after a spontaneous talk with family members, I got the idea of upscaling images as a project. I started researching this topic and found articles about the method I used with this model and chose to follow them.

Therefore, it was hard at first to decide about the topic of the project, but once I've decided to upscale images it got very interesting and I gained a lot of knowledge I couldn't learn in the regular machine learning class.

I learned how to use the Fastai library and PyTorch but most important, in my opinion, is that I gained the skill to learn an entirely different library, which works very differently from what I've learned in class, all by myself.

It was very hard though, to get used to this new library and most of my errors were because of very banal mistakes such as forgetting a parameter or referring to the wrong directory.

I also did not have enough time to train the model exactly as I wanted and research the hyperparameters more in depth. I wanted, for example, to augment the data in a way so that the model could not only upscale images but also be able to remove text, fix holes and reduce noises.

My conclusions are that we don't necessarily need GANs to improve image quality and that there are other successful methods to do it even better.

If I would have started the project today I would have tried to augment the data, use a more varied dataset (because I used only a

pet dataset) and try to explore the hyperparameters further.

If it would have happened, The working process would surely be more efficient and significant in its results.

To conclude, I had a lot of fun working on this project and it was an experience I couldn't get any other way in high school.

Bibliography

- Ronneberger, O., Fischer, P., & Brox, T. (2015). U-Net: Convolutional Networks for Biomedical Image Segmentation. https://arxiv.org/abs/1505.04597.
- Johnson, J., Alahi, A., & Fei-Fei, L. (2016). Perceptual Losses for Real-Time Style Transfer and Super-Resolution. https://arxiv.org/abs/1603.08155.
- Thomas, C. (2019). Deep learning-based super resolution, without using a GAN. https://towardsdatascience.com/deep-learning-based-super-resolution-without-using-a-gan-11c9bb5b6cd5.
- Zeiler, M. D., & Fergus, R. (2013). Visualizing and Understanding Convolutional Networks. https://arxiv.org/abs/1311.2901.
- Smith, L. N. (2018). A disciplined approach to neural network hyper-parameters: Part 1 -- learning rate, batch size, momentum, and weight decay. https://arxiv.org/abs/1803.09820.
- Karras, T., Aila, T., Laine, S., & Lehtinen, J. (2018). Progressive Growing of GANs for Improved Quality, Stability, and Variation. https://arxiv.org/abs/1710.10196.
- Shi, W., Caballero, J., Theis, L., Huszar, S., Aitken, A., Ledig, C., & Wang, Z. (2016). Is the deconvolution layer the same as a convolutional layer?. https://arxiv.org/abs/1609.07009.
- Aitken, A., Ledig, C., Theis, L., Caballero, J., Wang, Z., & Shi, W. (2017). Checkerboard artifact-free sub-pixel convolution. https://arxiv.org/abs/1707.02937.
- Ruiz, P. (2018). Understanding and visualizing ResNets. https://towardsdatascience.com/understanding-and-visualizing-resnets-442284831be8.
- Howard, J. (2018). Deep Learning Part 1 v3 Lesson 7. [YouTube video]. https://youtu.be/nWpdkZE2_cc.
- Fastai course v3. [Online course]. https://course.fast.ai/videos/?lesson=1.