Ain Shams University
Faculty of Computer & Information Sciences
Computer Science Department

Unsupervised Image Completion

By:

Noha Sami Mohamed [CS]
Mo’men Mostafa Kamel [CS]
Kamal Saad Kamal [CS]
Mohamed Abdullah El-Sayed [CS]
Warda Abd-Elfatah Eid [CS]

Under Supervision of:

Maryam Nabil El-Berry
Doctor,
Scientific Computing Department,
Faculty of Computer and Information Sciences,
Ain Shams University.

Doaa Mahmoud
Teacher Assistant,
Computer Science Department,
Faculty of Computer and Information Sciences,
Ain Shams University.

July 2021

Acknowledgement

We would like to thank everyone who played a role in our academic accomplishments.

First, our parents, who supported us. Without you, we could never have reached this current level of success.

Second, our supervisors: Dr. Maryam El-Berry and T.A. Doaa Mohamoud, each of whom has provided patient advice and guidance throughout the whole project.

Thank you all for your support.

Abstract

Unsupervised and self-supervised learning, or learning without human-labeled data, is a longstanding challenge of machine learning. Recently, it has seen incredible success in language, as transformer models and other variants have achieved top performance on a wide array of language tasks. However, the same broad class of models has not been successful in producing strong features for image classification.

Inspired by progress in unsupervised representation learning for natural language, we examine whether similar models can learn useful representations for images. We train a sequence Transformer to auto-regressively predict pixels, without incorporating knowledge of the 2D input structure. Despite training on low-resolution ImageNet without labels, we find that a GPT-2 scale model learns strong image representations as measured by linear probing, fine-tuning, and low-data classification. On CIFAR-10, we achieve 96.3% accuracy with a linear probe, outperforming a supervised Wide ResNet, and 99.0% accuracy with full finetuning, matching the top supervised pre-trained models. An even larger model trained on a mixture of ImageNet and web images is competitive with self-supervised benchmarks on ImageNet, achieving 72.0% top-1 accuracy on a linear probe of our features.

Table of Contents

List of Figures

List of Abbreviations

BERT:	Bidirectional Encoder Representations from Transformers
CMD:	Command Prompt
CPC v2:	Contrastive Predictive Coding v2
GELU:	Gaussian Error Linear Units
GPT-2:	Generative Pre-trained Transformer 2
GPU:	Graphic processing unit
GUI:	Graphic user interface
IGPT:	Image Generative Pre-Trained Transformer
ILSVRC 2012:	ImageNet Large Scale Visual Recognition Challenge 2012
ML:	Machine learning
MoCo:	Momentum Contrast
MLP:	multi-layer perceptron
NLP:	Natural language processing
OS:	Operating system
PNG:	Portable Graphics Format
RELU:	rectified linear unit
RoBERTa:	Robustly Optimized BERT Pretraining Approach
simCLR:	Simple Framework for Contrastive Learning Representations
TPU:	Tensor processing unit
TQDOM:	means "progress" in Arabic (taqadum, تقدّم)

List of Tables

1- Introduction

1.1 Motivation

It seems challenging to us to use very accurate unsupervised learning model in specific domain and try to apply it to other domain aiming to achieve the same accuracy. By applying our idea with sequence transformer given sufficient compute might ultimately be an effective way to learn excellent features in many domains and fields.

1.2 Problem Definition

We find that, just as a large transformer model trained on language can generate coherent text, the same exact model trained on pixel sequences can generate coherent image completions and samples. By establishing a correlation between sample quality and image classification accuracy, we show that our best generative model also contains features competitive with top convolutional nets in the unsupervised setting.

1.3 Objective

- Complete images through learning images representations and relationships between pixels.
- Curating large, labeled image datasets is both expensive and time consuming. Instead of further scaling up labeling efforts, we can instead aspire to learn general purpose representations from the much larger set of available unlabeled images and fine-tune them for classification.
- Aim to understand and bridge the gap between NLP and image classification using unsupervised and self-supervised learning models (BERT, ...etc.).

1.4 Time Plan

Time plan is a graph, schedule or chart that describes the different components of the project and how they will be implemented.

Time plan helps us to reach our goal within specific time and it is needed to track project schedule during developing process or even before that. It shows every task's start and end time and the period of each process during the entire project.

It includes the tasks performed and when these tasks will be performed.

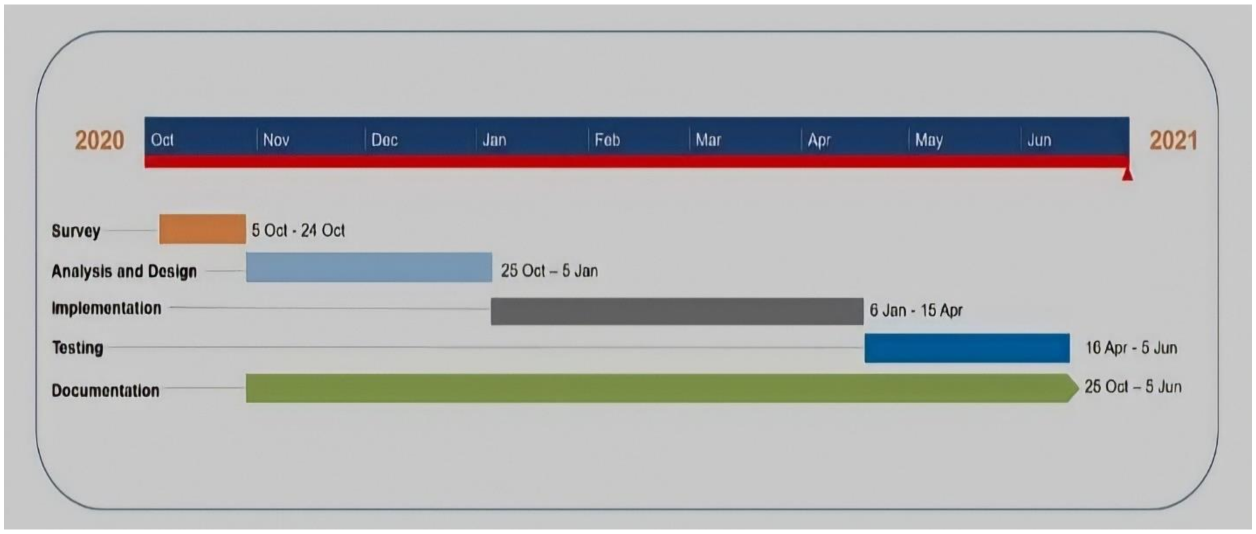


Figure 1.4.1 Time plan

1.5 Document Organization

Chapter 2: A quick look at description of the field of the project, previous work, methods used, survey results presented and comparison among them.

Chapter 3: Defines the system architecture along with the steps for the processes that happens in the system from the input to make the output.

Chapter 4: Shows the system implementation functions, algorithms and how they work together. It also shows the results of some completed images.

Chapter 5: presents a user manual with screen shots from a sample run for the project for testing purpose.

Chapter 6: Summarize our project and conclusions then show any future work and next steps that can be added to the project.

2- Background

2.1 field of the project

Just as a large transformer model trained on language can generate coherent text, the same exact model trained on pixel sequences can generate coherent image completions and samples. By establishing a correlation between sample quality and image classification accuracy, we show that our best generative model also contains features competitive with top convolutional nets in the unsupervised setting. Unsupervised and self-supervised learning, or learning without human-labeled data, is a longstanding challenge of machine learning. Recently, it has seen incredible success in language, as transformer models like **BERT**, **GPT-2**, **RoBERTa**, and other variants have achieved top performance on a wide array of language tasks. However, the same broad class of models has not been successful in producing strong features for image classification. Our work aims to understand and bridge this gap.

2.2 scientific background related to the project.

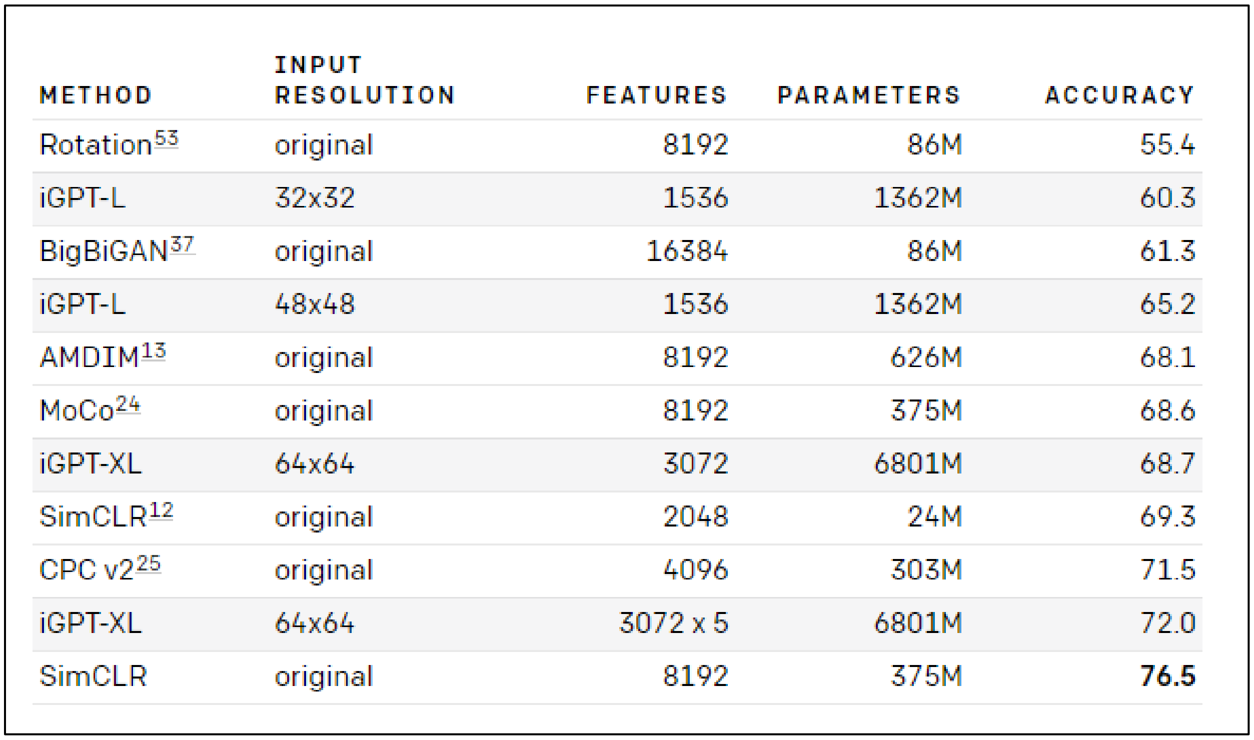
Unsupervised pre-training played a central role in the resurgence of deep learning. Starting in the mid 2000's, approaches such as the Deep Belief Network (Hinton et al., 2006)[1] and Denoising Autoencoder (Vincent et al., 2008)[2] were commonly used in neural networks for computer vision (Lee et al., 2009)[3] and speech recognition (Mohamed et al., 2009)[4]**.** It was believed that a model which learned the data distribution P(X) would also learn beneficial features for the subsequent supervised modeling of P (Y jX) (Lasserre et al., 2006[5]; Erhan et al., 2010[10]). However, advancements such as piecewise linear activation functions (Nair & Hinton, 2010)[6], improved initializations (Glorot & Bengio, 2010)[7], and normalization strategies (Ioffe & Szegedy, 2015[8] ; Ba et al., 2016[9]) removed the need for pre-training in order to achieve strong results. Other research cast doubt on the benefits of deep unsupervised representations and re-ported strong results using a single layer of learned features (Coates et al., 2011)[11] , or even random features (Huang et al., 2014[12] ; May et al., 2017[13]). The approach fell out of favor as the state of the art increasingly relied on directly encoding prior structure into the model and utilizing abundant supervised data to directly learn representations (Krizhevsky et al., 2012[14] ; Graves & Jaitly, 2014[15]). Retrospective study of unsupervised pre-training demonstrated that it could even hurt performance in modern settings (Paine et al., 2014)[16] . Instead, unsupervised pre-training flourished in a different domain. After initial strong results for word vectors (Mikolov et al., 2013)[17], it has pushed the state of the art forward in Natural Language Processing on most tasks (Dai & Le, 2015[18] ; Peters et al., 2018[19] ; Howard & Ruder, 2018[20] ; Radford et al., 2018[21] ; Devlin et al., 2018[22]). Interestingly, the training objective of a dominant approach like **BERT**, the prediction of corrupted inputs, closely resembles that of the Denoising Autoencoder, which was originally developed for images.

As a higher dimensional, noisier, and more redundant modality than text, images are believed to be difficult for generative modeling. Here, self-supervised approaches designed to encourage the modeling of more global structure (Doersch et al., 2015[23]) have shown significant promise. A combination of new training objectives (Oord et al., 2018)[24] , more recent architectures (Gomez et al., 2017)[25], and increased model capacity (Kolesnikov et al., 2019)[26] has allowed these methods to achieve state of the art performance in low data settings (H´enaff et al., 2019)[27] and sometimes even outperform supervised representations in transfer learning settings (He et al., 2019[28] ; Misra & van der Maaten, 2019[29] ; Chen et al., 2020[30]). Given that it has been a decade since the original wave of generative pre-training methods for images and considering their substantial

impact in **NLP**, this class of methods is due for a modern re-examination and comparison with the recent progress of self-supervised methods. We re-evaluate generative pre-training on images and demonstrate that when using a flexible architecture (Vaswani et al., 2017)[31] , a tractable and efficient likelihood based training objective (Larochelle & Murray, 2011[32] ; Oord et al., 2016[33]), and significant compute resources (2048 TPU cores), generative pre-training is competitive with other self-supervised approaches and learns representations that significantly improve the state of the art in low-resolution unsupervised representation learning settings.
This is especially promising as our architecture uses a dense connectivity pattern which does not encode the 2D spatial structure of images yet is able to match and even outperform approaches which do. We report a set of experiments characterizing the performance of our approach on many datasets and in several different evaluation settings (low data, linear evaluation, full fine-tuning). We also conduct several experiments designed to better understand the achieved performance of these models. We investigate how representations are computed inside our model via the performance of linear probes as a function of model depth as well as studying how scaling the resolution and parameter count of the approach affects performance.

2.3 A survey of the work done in the field.

Given the resurgence of interest in unsupervised and self-supervised learning on ImageNet, we also evaluate the performance of our models using linear probes on ImageNet. This is an especially difficult setting, as we do not train at the standard ImageNet input resolution. Nevertheless, a linear probe on the 1536 features from the best layer of **IGPT-L** trained on 48x48 images yields 65.2% top-1 accuracy, outperforming AlexNet. Contrastive methods typically report their best results on 8192 features, so we would ideally evaluate **IGPT** with an embedding dimension of 8192 for comparison. However, training such a model is prohibitively expensive, so we instead concatenate features from multiple layers as an approximation. Taking 15360 features from 5 layers in **iGPT-XL** yields 72.0% top-1 accuracy, outperforming AMDIM, **MoCo**, and **CPC v2**, but still underperforming **SimCLR** by a decent margin.



METHOD	INPUT RESOLUTION	FEATURES	PARAMETERS	ACCURACY
Rotation[53]	original	8192	86M	55.4
iGPT-L	32x32	1536	1362M	60.3
BigBiGAN[37]	original	16384	86M	61.3
iGPT-L	48x48	1536	1362M	65.2
AMDIM[13]	original	8192	626M	68.1
MoCo[24]	original	8192	375M	68.6
iGPT-XL	64x64	3072	6801M	68.7
SimCLR[12]	original	2048	24M	69.3
CPC v2[25]	original	4096	303M	71.5
iGPT-XL	64x64	3072 x 5	6801M	72.0
SimCLR	original	8192	375M	**76.5**

Table 2.3.1: A comparison of linear prob

We achieve competitive performance while training at much lower input resolutions, though our method requires more parameters and compute.

3 Analysis and Design

3.1 System Overview

3.1.1 System Architecture

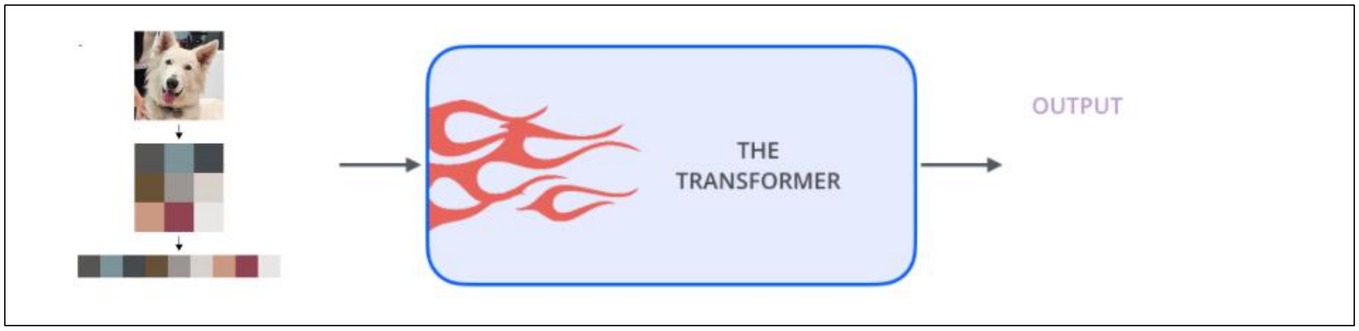


Figure 3.1.1.1: GPT-2 Architecture

GPT-2 main architecture is the transformer. The original transformer model is made up of an encoder and decoder, each is a stack of what we can call transformer blocks, but our model **GPT-2** is built using the transformer decoder blocks only, it threw away the Transformer encoder.

The **GPT-2**, and some later models like TransformerXL and XLNet are auto-regressive in nature.

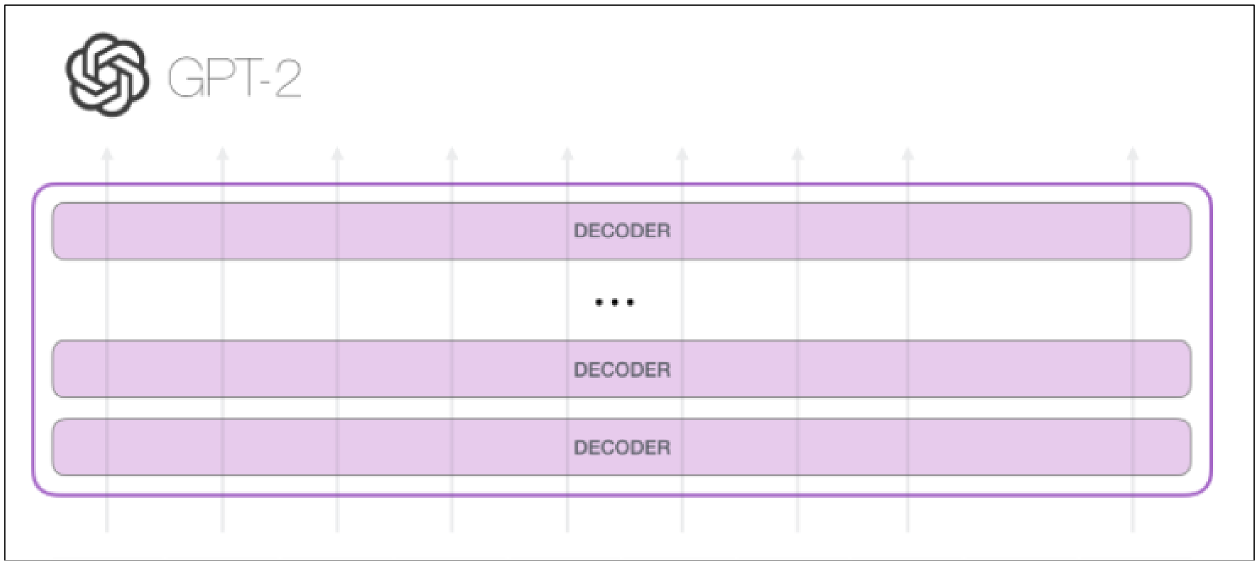


Figure 3.1.1.2: GPT-2 Decoder Blocks

The decoders are all identical in structure, each one is broken down into three sub-layers and the decoder has a small architectural variation from the encoder block- an attention layer that helps the decoder focus on relevant parts of the input pixels.

The transformer decoder takes an input sequence $\mathbf{x_1}$; ….; $\mathbf{x_n}$ of discrete pixels and produces a d-dimensional embedding for each position. The decoder is realized as a stack of L blocks, the l-th of which produces an intermediate embedding $\mathbf{h^l_1}$; ……; $\mathbf{h^l_n}$ also of dimension d.

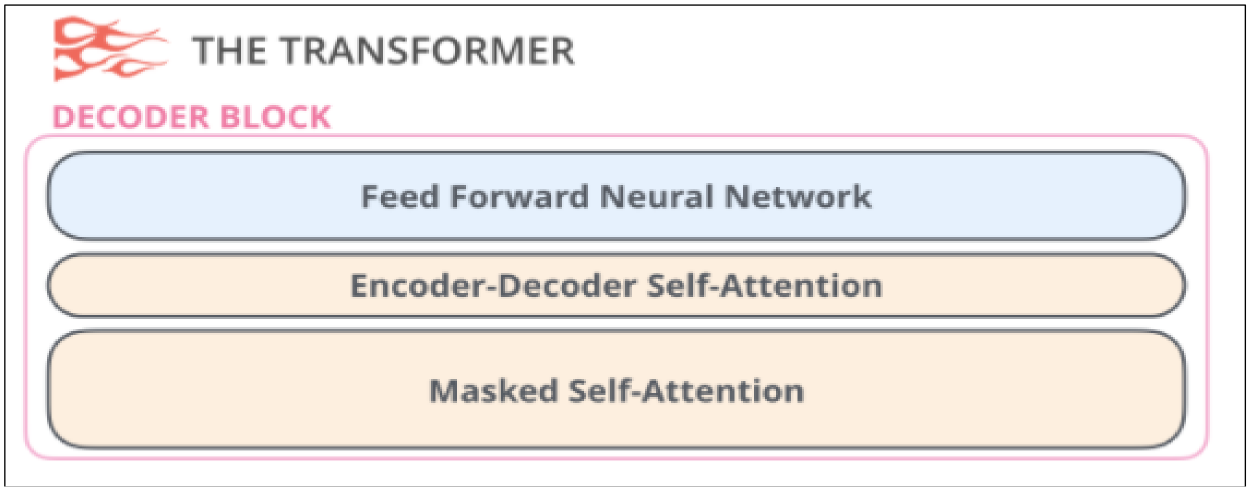


Figure 3.1.1.3: Decoder Block Architecture

We use the **GPT-2** (Radford et al., 2019)[34] formulation of the transformer decoder block, which acts on an input tensor $\mathbf{h^l}$ as follows:

- n^l = layer norm(h^l)
- $a^l = h^l$ + multihead attention(n^l)
- $h^l+1 = a^l$ + mlp(layer norm(a^l))

In particular, layer norms precede both the attention and **MLP** operations, and all operations lie strictly on residual paths. We find that such a formulation allows us to scale the transformer with ease.

As in the figure 3.1.1.3, the main layers of the decoder block and our model uses more than one decoder block**.** **GPT-2** made an update in decoder blocks that used and let us call it "*Transformer-Decoder*".

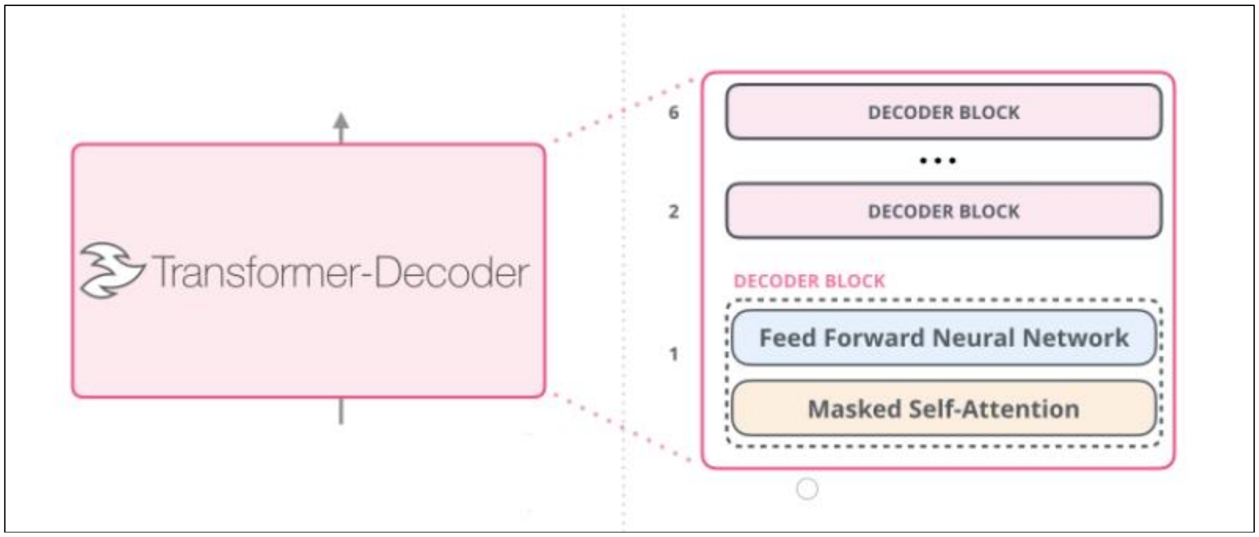


Figure 3.1.1.4: Transformer-Decoder Arch

Its decoder blocks are very similar to the original decoder blocks, except it threw away that second self-attention layer that was in figure 3.1.1.3. This early transformer-based language model was made up of a stack of six transformer decoder blocks.

3.1.2 System Users

A. Intended Users:

- System is intended to be used by any organizations or fields that interested in image generation.
- Researchers that are trying to do classification tasks and do not have the desired dataset, they can fine-tune our model to do their task instead of creating new dataset.

B. User Characteristics

- Supervisors should be familiar with Ubuntu **OS**, python.
- Supervisors may know fine-tuning and image classification.

3.2 System Analysis & Design

3.2.1 Use Case Diagram

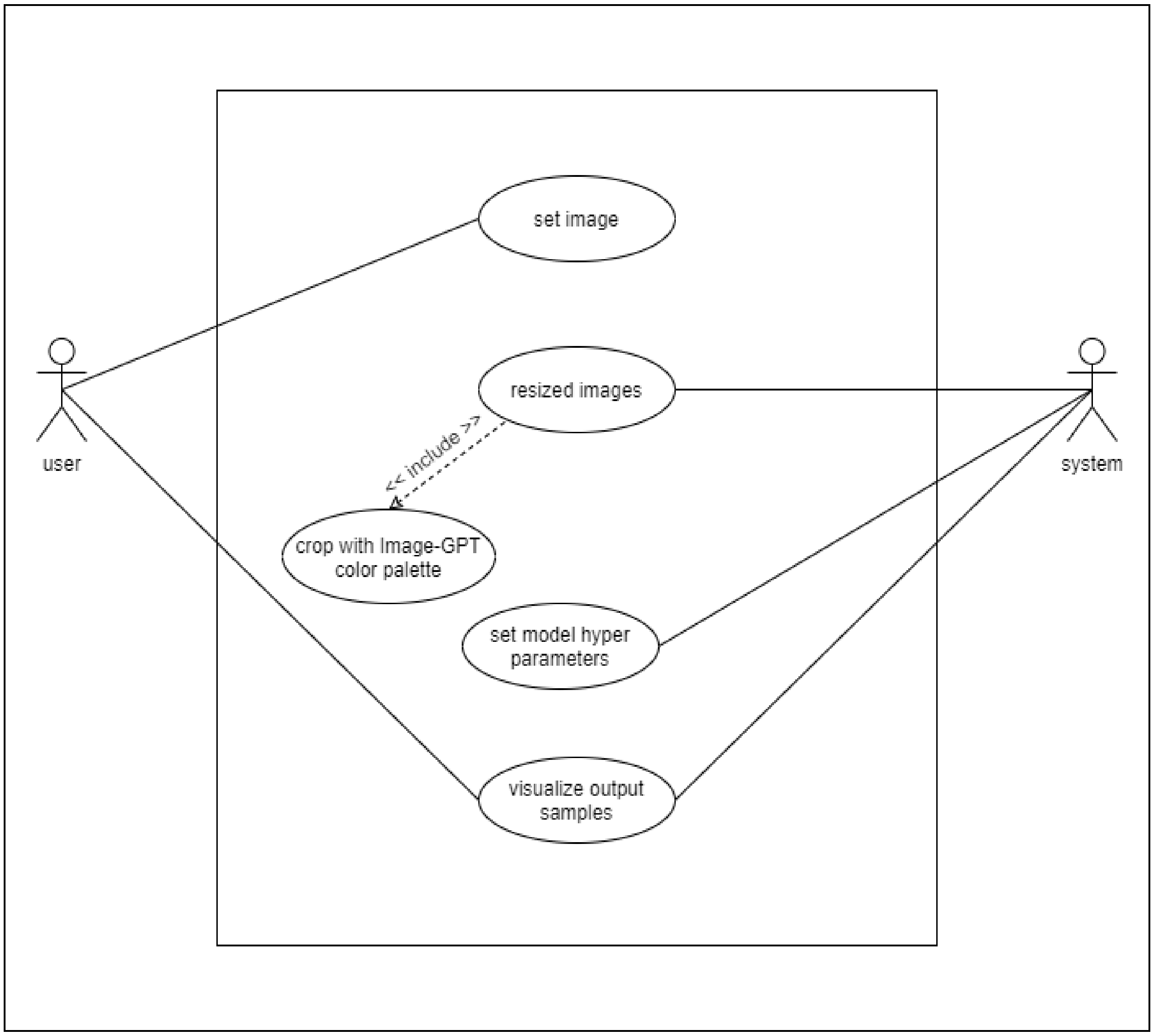


Figure 3.2.1.1: Use Case Diagram

3.2.2 Sequence Diagram

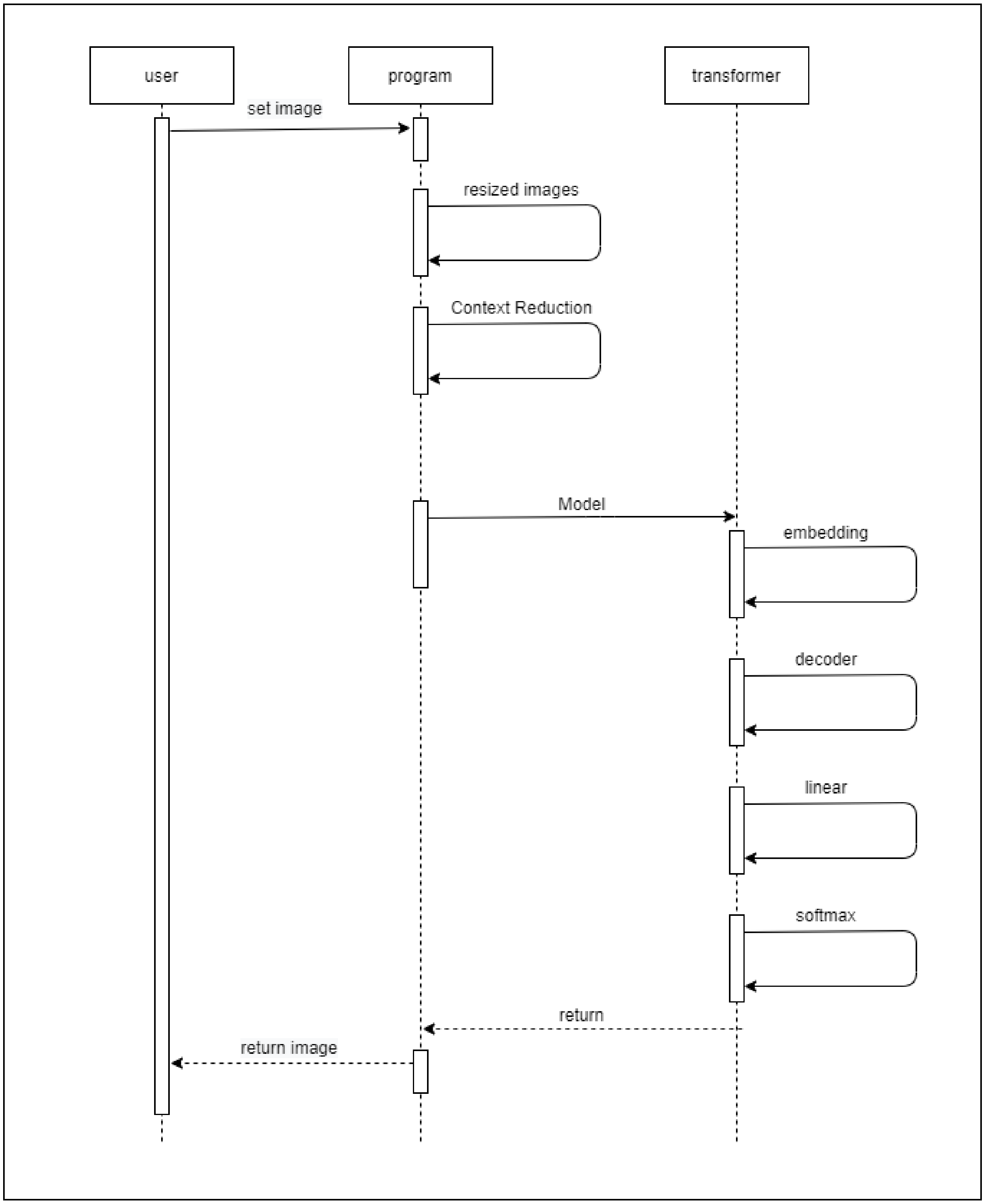


Figure 3.2.2.1: Sequence Diagram

4- Implementation and Testing

4.1 Dataset and data augmentation

4.1.1 dataset introduction.

We used the ImageNet **ILSVRC 2012**, commonly known as 'ImageNet', It is an image dataset organized according to the WordNet hierarchy. Each meaningful concept in **WordNet**, possibly described by multiple words or word phrases, is called a "synonym set" or "synset".
We split off 4% as our experimental validation set and report results on the **ILSVRC** 2012 validation set as our test set.

4.1.2 data augmentation.

We used lightweight data augmentation, specifically when employing data augmentation, we randomly resize an image such that the shorter side length is in the range [256; 384] and then take a random 224 × 224 crops. When evaluating on ImageNet, we resize the image such that the shorter side length is 224 and use the single 224×224 center crop.

4.2 description of all the functions in the system.

- default_hparams

  set the default hyperparameter in case of not initializing the hyperparameters.

- shape_list

  Deal with dynamic shape in TensorFlow cleanly.

- Softmax

  It is a function that turns a vector of K real values into a vector of K real values that sum to 1. The input values can be positive, negative, zero, or greater than one, but the softmax transforms them into values between 0 and 1, so that they can be interpreted as probabilities.

$$\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

σ	= softmax
$\vec{z}$	= input vector
e^{z_i}	= standard exponential function for input vector
K	= number of classes in the multi-class classifier
e^{z_j}	= standard exponential function for output vector
e^{z_j}	= standard exponential function for output vector

Figure 4.2.1 softmax function

- ## Gelu

  The **GELU** activation function is xΦ(x), where Φ(x) the standard Gaussian cumulative distribution function. The **GELU** nonlinearity weights inputs by their percentile, rather than gates inputs by their sign as in **ReLU.**

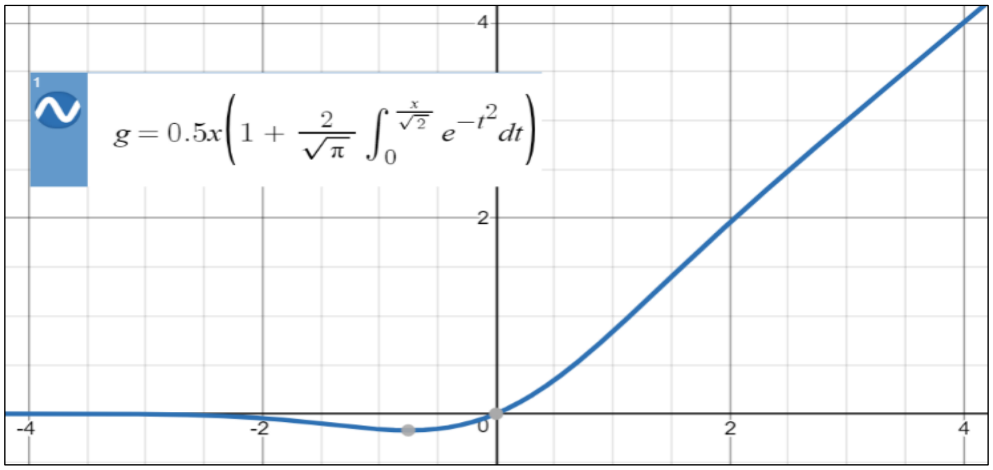


Figure 4.2.2 GELU activation function 1

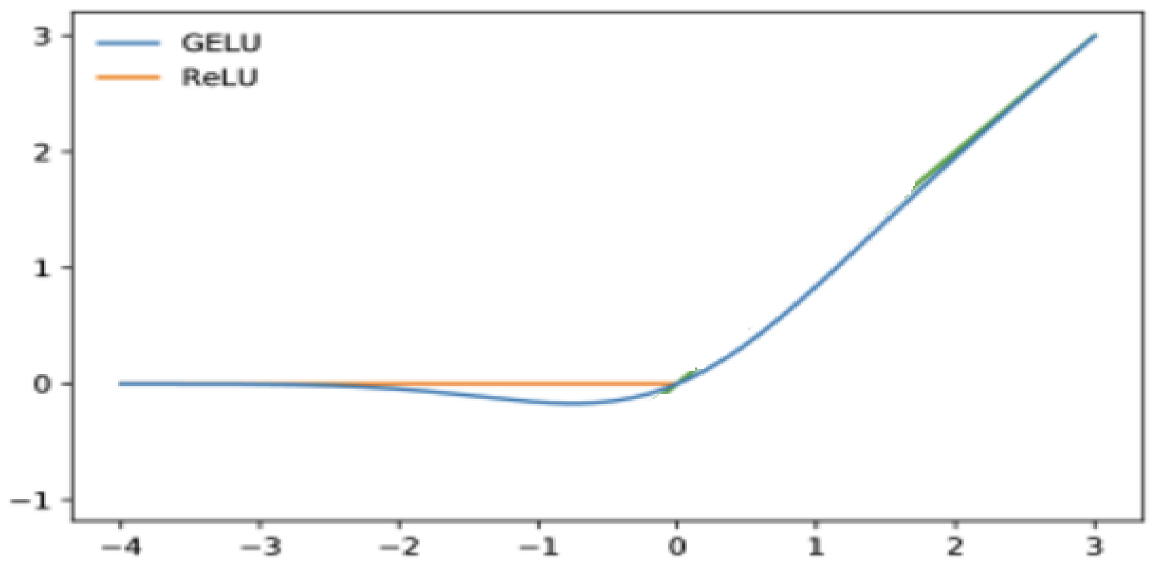


Figure 4.2.3 GELU and RELU graphs

- Norm

  Normalize input to mean = 0, std = 1, then do a diagonal affine transform.
- split_states

  Reshape the last dimension of x into [n, x.shape[-1]/n]
- merge_states

  Smash the last two dimensions of x into a single dimension.
- attention_mask

  Same as tf.matrix_band_part(tf.ones([nd, ns]), -1, ns-nd), but doesn't produce garbage on **TPUs.**
- Attn

  Just checking on some dimensions that they are correct.
- split_heads

  split the multi heads from [batch, sequence, features] to [batch, heads, sequence, features]
- merge_heads

  Reverse of split_heads
- mask_attn_weights

  creates the masked attention layer.

- multihead_attn

  multiply K-matrix and Q-matrix then rescale the result by dividing the map by the square root of the matrix dimension to measure the similarity between the key and the query matrixes then multiply it by the masked attention matrix then doing a softmax function and finally multiply the result by V-matrix.
- Mlp

  Normal multilayer perceptron function.
- block

  calculate the **mlp** operation and atten operation then add them in one matrix.
- past_shape

  return array that contains some parameters that are used in forward functions.
- expand_tile

  Add a new axis of given size.
- positions_for

  expand the array of the past length by the batch size.
- model

  the function that ties everything together and responsible for the transformer and the responsible for the classification task in the end.
- parse_arguments

  Parses the important arguments like dataset path, the checkpoint, saving directory, color clusters, …. etc.
  There are many arguments divided on data, model, parallelism, mode, and reproducibility.
- set_seed

  set the seed for the randomization.

- load_data

  just load the training, validating, and testing data.

- set_hparams
  set the hyperparameters using the parsed arguments.

- create_model
  preparing the important arrays for the classification loss, total loss, accuracy. Marking the trainable parameters, distribute the training on many **GPU**s if there are more than one.
- reduce_mean
  calculate the mean for the general loss, classification loss, total loss, accuracy.

- Evaluate
  Calculate the general loss, classification loss, total loss, accuracy and them get the mean of them.
- Sample
  Creates samples, the size of each sample is [n_gpu * n_sub_batch, n_px * n_px], then calculate the logits probability, then dequantize the image and write it into **PNG**.
- squared_euclidean_distance
  it calculates the Euclidean distance but squared.
- color_quantize
  it reduces the number of distinct colors in an image.
- count_parameters
  returns the number of trainable parameters.

4.3 model description.

4.3.1 model name and description

Our model is called **IGPT** that is the same as **GPT** model (works for Languages in the **NLP** field). **IGPT** consists of a pre-training stage followed by a fine-tuning stage. In pre-training, we explore the auto-regressive objective. We also apply the sequence Transformer architecture to predict pixels instead of language tokens.

Note: **GPT** uses only Decoder's block of the Transformer architecture.

4.3.2 context reduction

As we explained in the past section after the lightweight data augmentation, we get an image with size of 224×224×3, But because the memory requirements of the transformer decoder scale quadratically with context length when using dense attention, we must employ further techniques to reduce context length. If we naively trained a transformer on a sequence of length 2242 ×3, our attention logits would be tens of thousands of times larger than those used in language models and even a single layer would not fit on a GPU. To deal with this, we first resize our image to a lower resolution, which we call the input resolution (IR). Our models have IR of $32^2 \times 3$ and it is still quite computationally intensive. While working at even lower resolutions is tempting, prior work has demonstrated human performance on image classification begins to drop rapidly below this size (Torralba et al.,2008)[39] . Instead, motivated by early color display palettes, we create our own 9-bit color palette by clustering (R, G, B) pixel values using k-means with k = 512. Using this palette yields an input sequence length 3 times shorter than the standard (R, G, B) palette, while still encoding color faithfully.

4.3.3 Token embedding and positional embedding.

First, we map each distinct pixel into the nearest pixel in our (R, G, B) pallet.

Then we get the index of the new colors and enter them into the embedding layer that has an index for every pixel in our pallet and against each of those indices a vector is attached as shown in Figure 4.3.3.1. initially these vectors are filled up with random numbers later during training phase the model updates them with values that better help them with the assigned task.

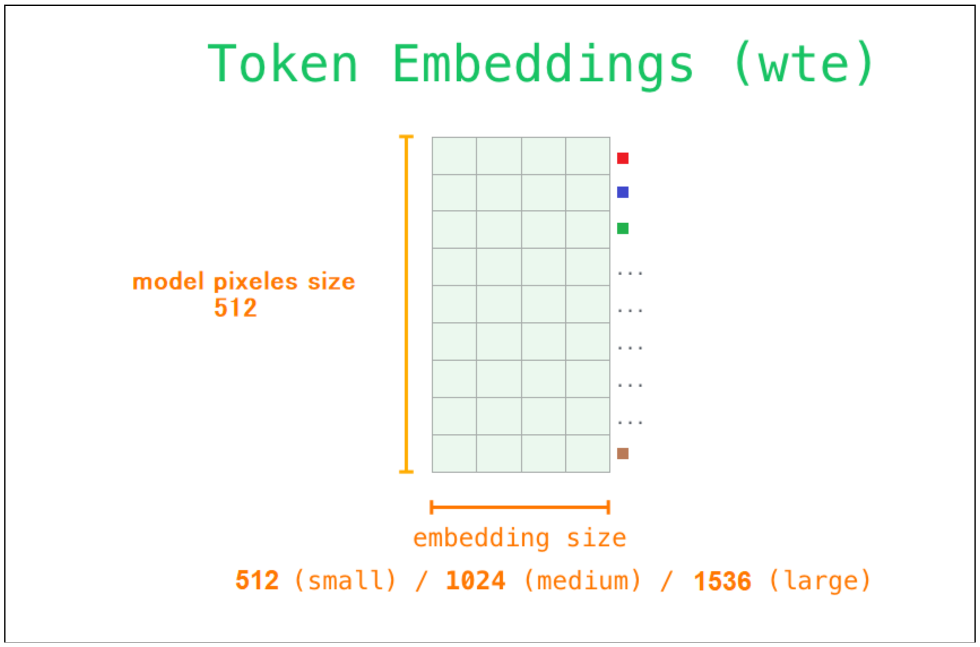


Figure 4.3.3.1 pixels embeddings

Embeddings are just vector representation of a given pixel each dimension of the pixel embedding tries to capture some representation feature about that pixel.

Then we talk about the position embeddings, since this model does not contain any recurrence or convolution, positional encoding is added to give the model some information about the relative position of the pixel in the image.

The positional encoding vector is added to the embedding vector. Embeddings represent a token in a d-dimensional space where tokens with similar meaning will be closer to each other. But the embeddings do not encode the relative position of pixels in a sentence. So, after adding the positional encoding, pixels will be closer to each other based on the similarity of their meaning and their position in the image, in the d-dimensional space.

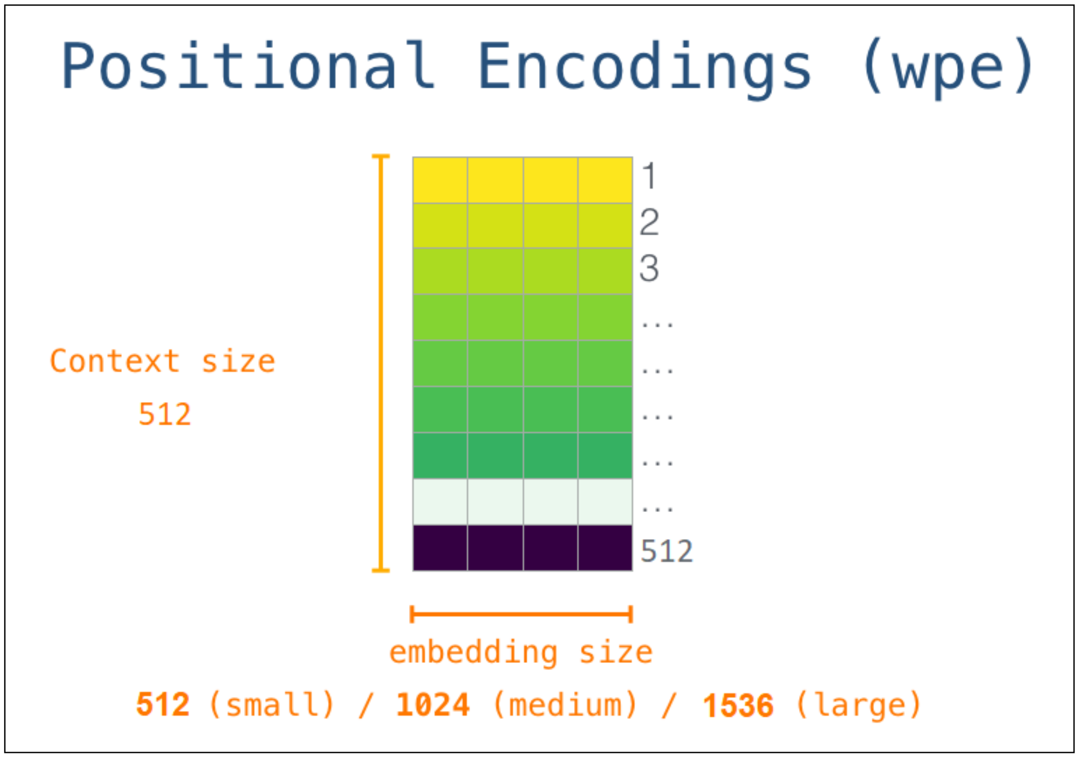

Positional Encodings (wpe)
Context size
512
1
2
3
...
512
embedding size
512 (small) / 1024 (medium) / 1536 (large)

Figure 4.3.3.2 positional embedding

The formula for calculating the positional encoding is as shown in next figure.

$$PE_{(pos,2i)} = \sin(pos/10000^{2i/d_{model}})$$

$$PE_{(pos,2i+1)} = \cos(pos/10000^{2i/d_{model}})$$

Figure 4.3.3.3 PE formulas 1

Here is what the position embedding curves can look like when plotted on a full scale in next figure.

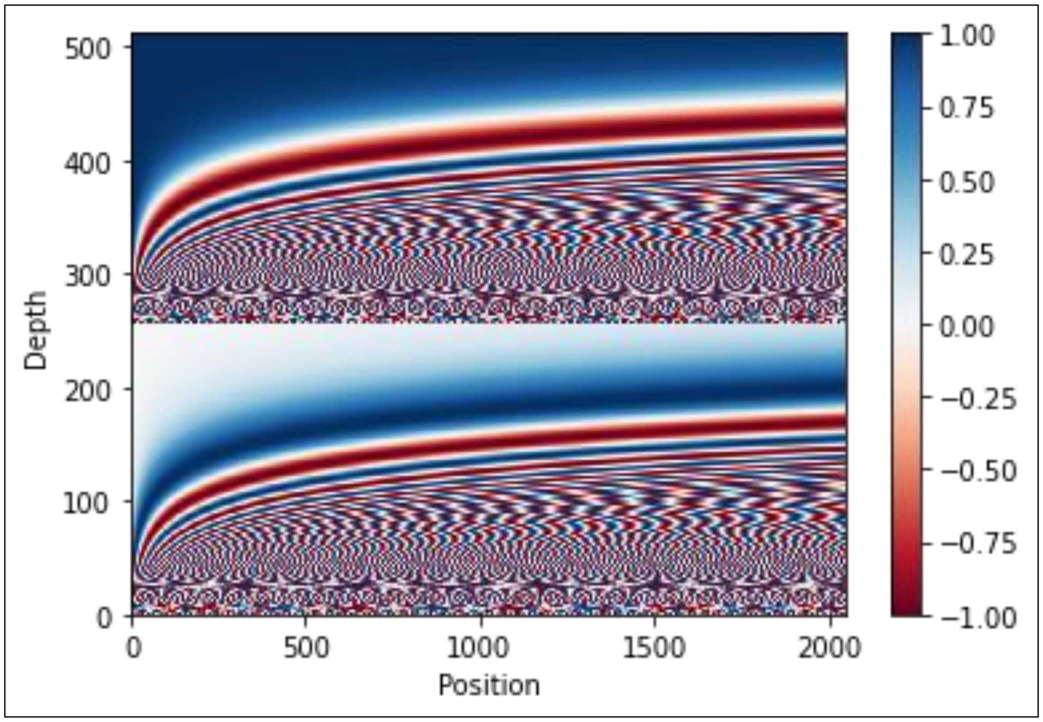


Figure 4.3.3.4 frequencies of PE

now we can just go ahead and add these position embeddings to our token embeddings and pass that to the next phase in the model.

4.3.4 Transformer-Decoder

The first block can now process the token pixel by first passing it through the masked self-attention process, then passing it through its neural network layer as shown in following figure.

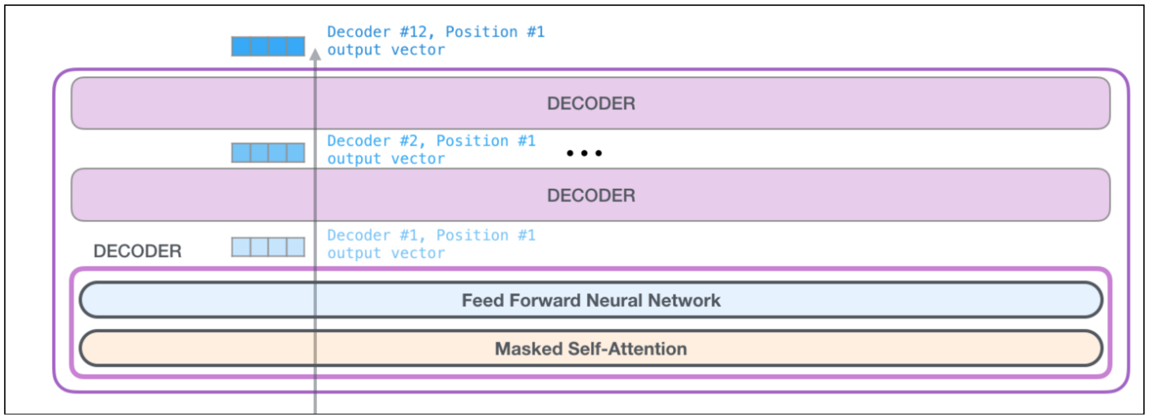


Figure 4.3.4.1 GPT as stack of decoders

4.3.4.1 Masked Self-Attention process

Masked self-attention is processed along the path of each token pixel in the segment.

The significant components are three vectors:

- **Query**: The query is a representation of the current word used to score against all the other words (using their keys). We only care about the query of the token we are currently processing.
- **Key**: Key vectors are like labels for all the words in the segment. They are what we match against in our search for relevant words.
- **Value**: Value vectors are actual word representations, once we have scored how relevant each word is, these are the values we add up to represent the current word.

Masked self-attention is applied through three main steps:

1) Creating the Query, Key, and Value vectors for each path:

Let us focus on the first path. We will take its query and compare against all the keys. That produces a score for each key. The first step in self-attention is to calculate the three vectors for each token path.

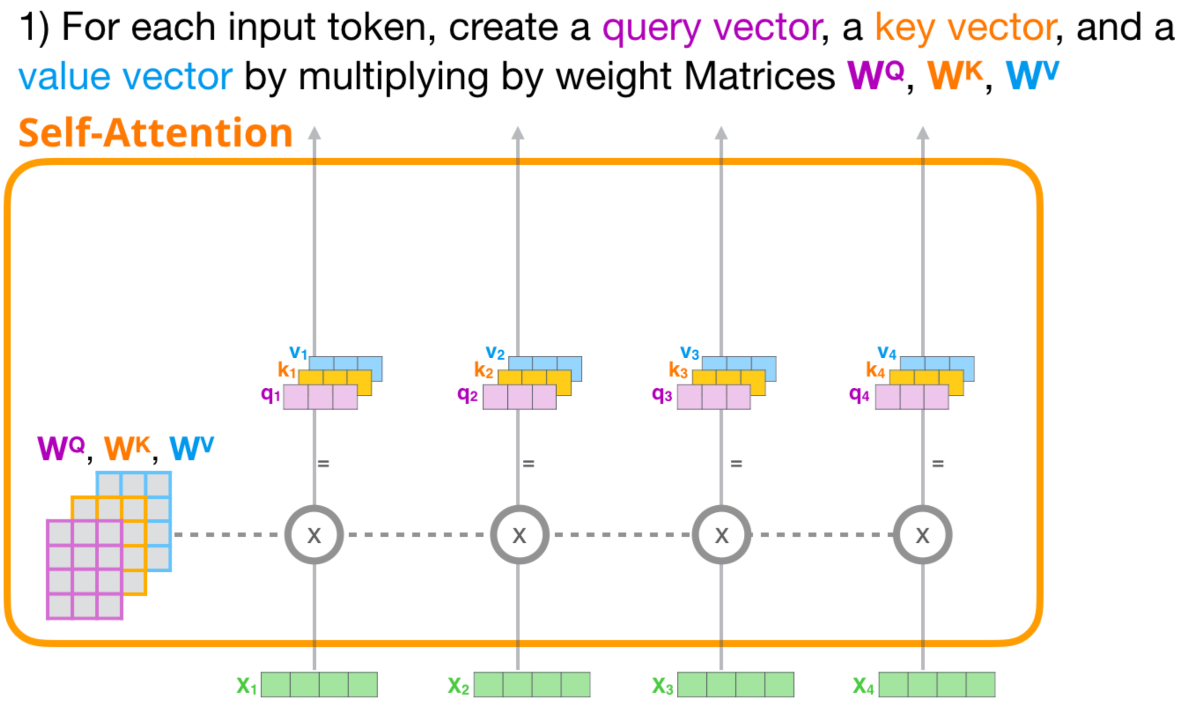

1) For each input token, create a query vector, a key vector, and a value vector by multiplying by weight Matrices WQ, WK, WV
Self-Attention
WQ, WK, WV
v1
k1
q1
v2
k2
q2
v3
k3
q3
v4
k4
q4
=
x
x1
x2
x3
x4

Figure 4.3.4.1.1: Creating Q, K, V vecs

2) Scoring and masking

First to get score, for each input token pixel, use its query vector to score against all the other key vectors.

Now that we have the vectors, we use the query and key vectors only for step #2. Since we are focused on the first token, we multiply its query by all the other key vectors resulting in a score for each of the four tokens.

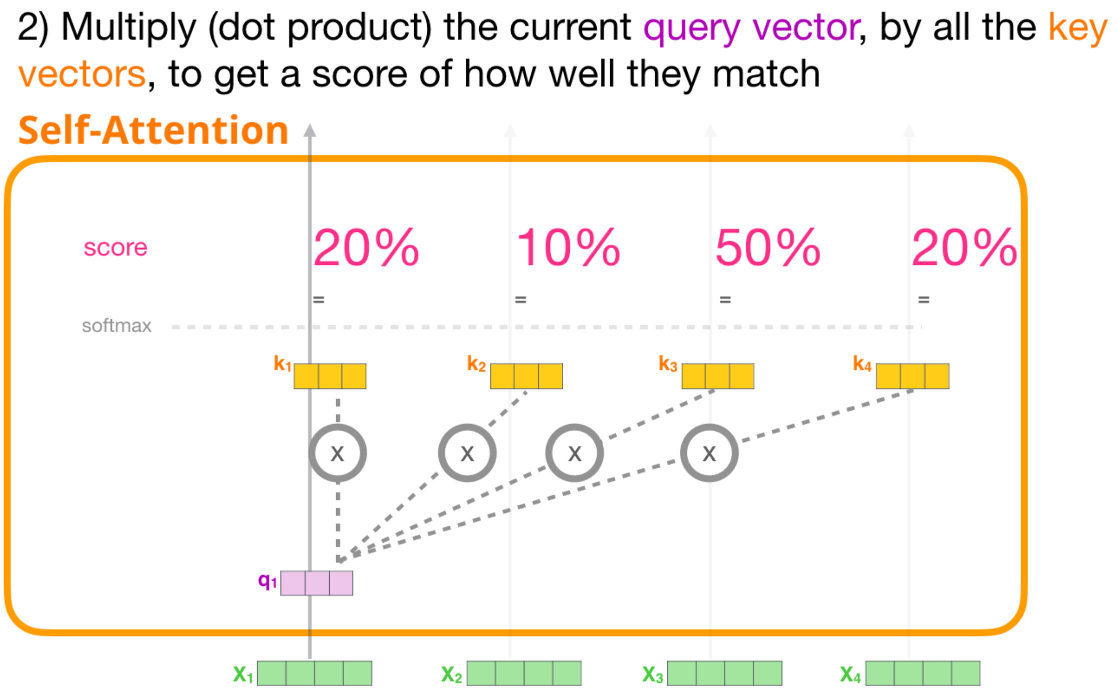

2) Multiply (dot product) the current query vector, by all the key vectors, to get a score of how well they match
Self-Attention
score
20%
10%
50%
20%
softmax
k1
k2
k3
k4
x
q1
X1
X2
X3
X4

Figure 4.3.4.1.2: Get score Process.

Second to apply masking, let us assume the model only has two tokens as input and we are observing the second token. In this case, the last two tokens are masked. So, the model interferes in the scoring step. It basically always scores the future tokens as 0 so the model cannot peak to future words.

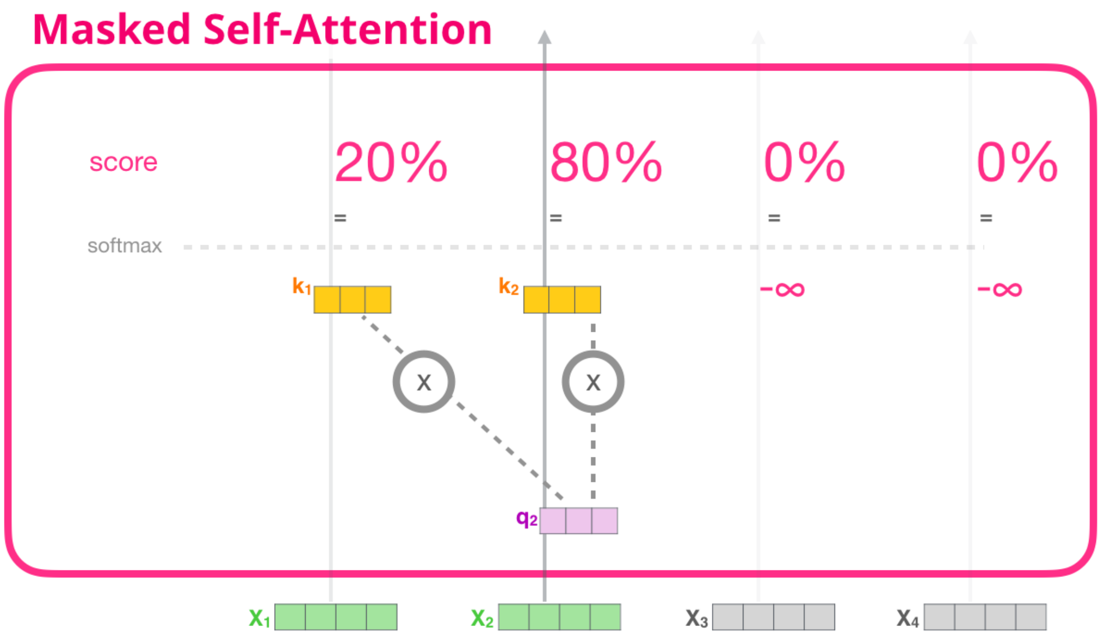
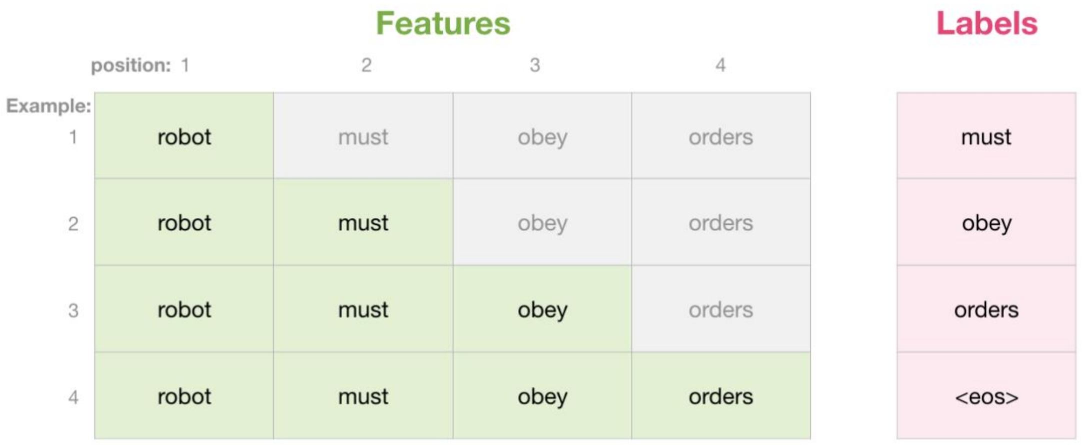


Figure 4.3.4.1.3: Masking Process

This masking process is often implemented as a matrix called an attention mask. To make the explanation easier, we will take example in a language modeling scenario. Think of a sequence of four words (“robot must obey orders”, for example), this sequence is absorbed in four steps – one per word (assuming for now that every word is a token). As these models work in batches, we can assume a batch size of 4 for this toy model that will process the entire sequence (with its four steps) as one batch.



Example:	position: 1	2	3	4	Labels
1	robot	must	obey	orders	must
2	robot	must	obey	orders	obey
3	robot	must	obey	orders	orders
4	robot	must	obey	orders	<eos>

Features
Labels

Figure 4.3.4.1.4: Batch

In matrix form, we calculate the scores by multiplying a queries matrix by a keys matrix. Let us visualize it as follows, except instead of the word, there would be the query (or key) vector associated with that word in that cell.

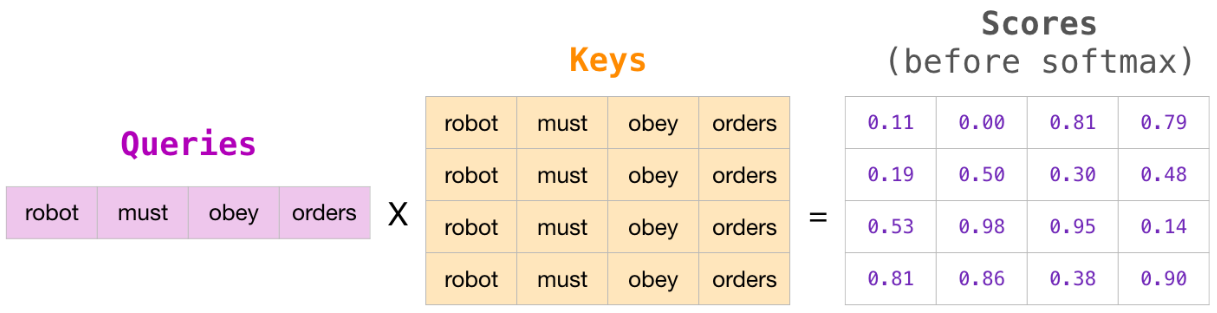
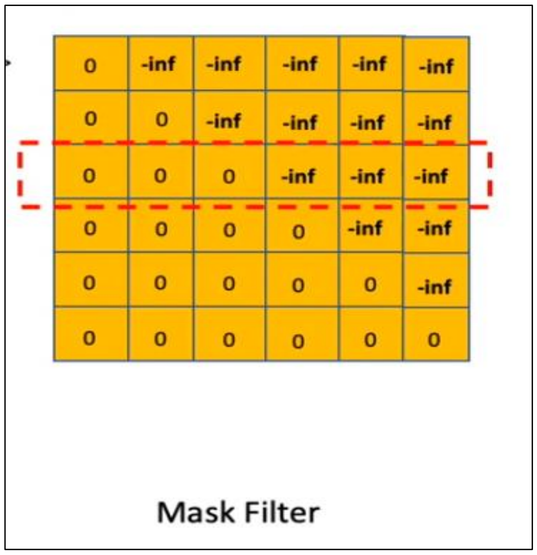


Figure 4.3.4.1.5: Scoring in detail

After the multiplication, we slap on our attention mask triangle. It set the cells we want to mask to -infinity or a very large negative number (e.g., -1 billion in **GPT2**).

We create our attention mask the same size as scores matrix.

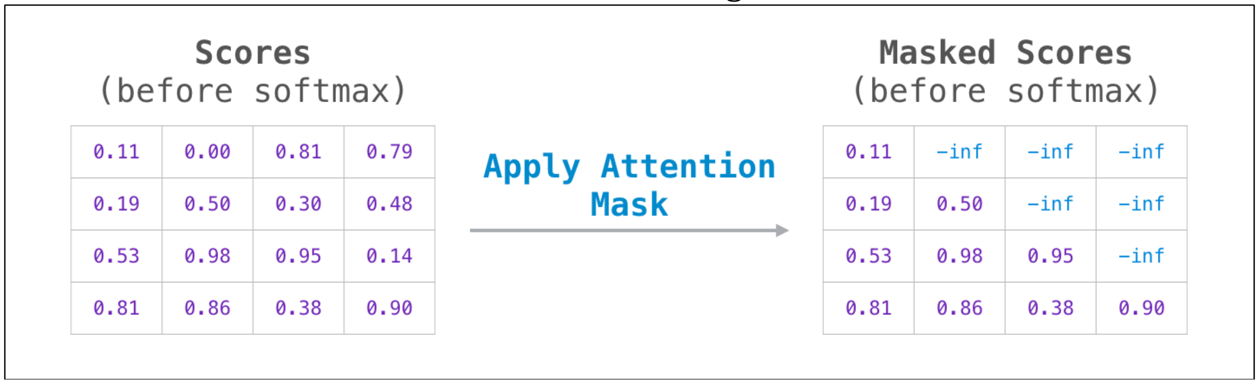


0	-inf	-inf	-inf	-inf	-inf
0	0	-inf	-inf	-inf	-inf
0	0	0	-inf	-inf	-inf
0	0	0	0	-inf	-inf
0	0	0	0	0	-inf
0	0	0	0	0	0

Mask Filter

Figure 4.3.4.1.6: Attention Mask Matrix

Then we add this mask to the score matrix to get masked score matrix.



Scores
(before softmax)

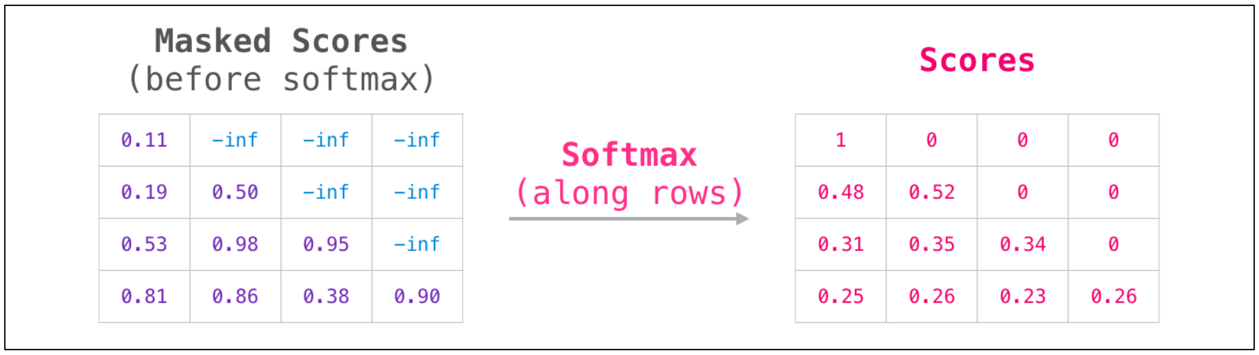
0.11	0.00	0.81	0.79
0.19	0.50	0.30	0.48
0.53	0.98	0.95	0.14
0.81	0.86	0.38	0.90

Masked Scores
(before softmax)

0.11	−inf	−inf	−inf
0.19	0.50	−inf	−inf
0.53	0.98	0.95	−inf
0.81	0.86	0.38	0.90

Figure 4.3.4.1.7: Apply Attention Mask

Then, applying softmax activation function on each row produces the actual scores.



0.11	-inf	-inf	-inf
0.19	0.50	-inf	-inf
0.53	0.98	0.95	-inf
0.81	0.86	0.38	0.90

1	0	0	0
0.48	0.52	0	0
0.31	0.35	0.34	0
0.25	0.26	0.23	0.26

Figure 4.3.4.1.7: Get final Score matrix

What this score matrix means is the following:

- When the model processes the first example in the dataset (row #1), which contains only one word ("robot"), 100% of its attention will be on that word.
- When the model processes the second example in the dataset (row #2), which contains the words ("robot must"), when it processes the word "must", 48% of its attention will be on "robot", and 52% of its attention will be on "must".
- And so on

3) Sum up the value vectors:

After getting masked scored matrix, multiply each value vectors of them by their associated masked scores. A value with a high score will constitute a large portion of the resulting vector after we sum them up.

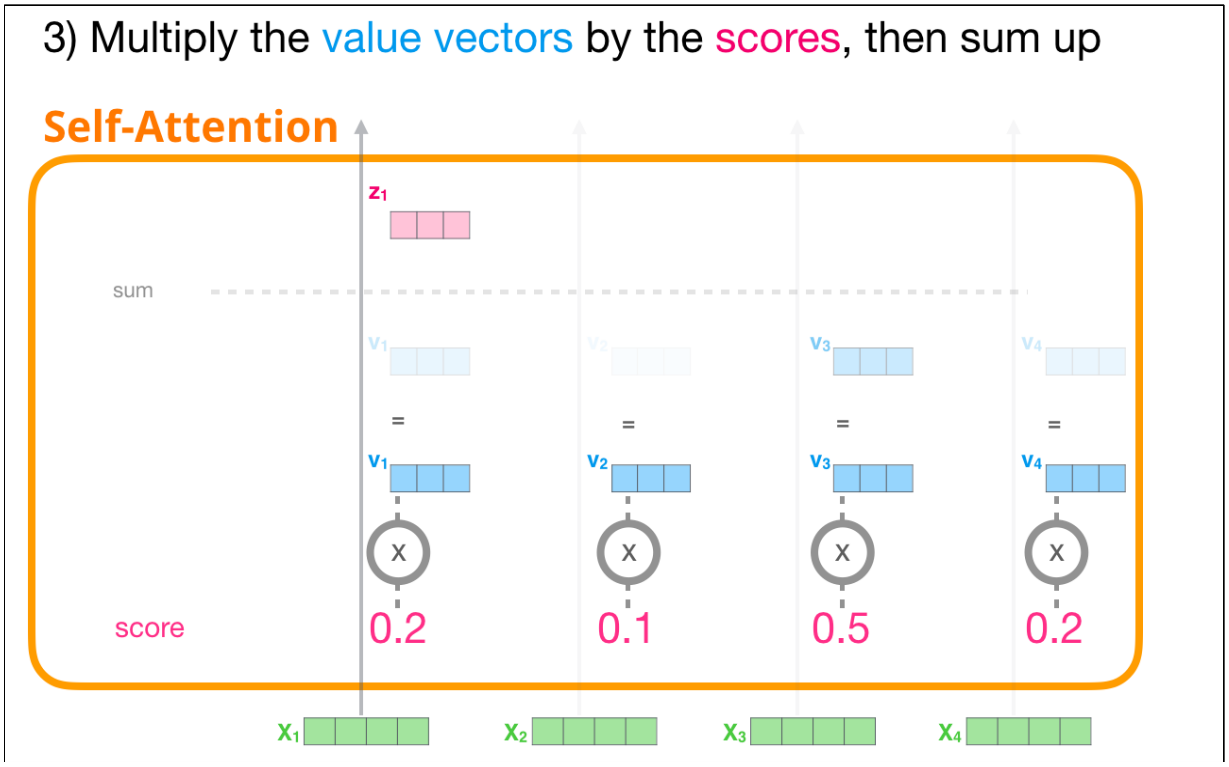


Figure 4.3.4.1.8: Multiply and Sum vecs

The lower the score, the more transparent we are showing the value vector. That is to indicate how multiplying by a small number dilutes the values of the vector.

If we do the same operation for each path, we end up with a vector representing each token containing the appropriate context of that token and that achieved the multi-head part.

Those are then presented to the next sublayer in the transformer block (the feed-forward neural network)

4.3.4.2 Fully Connected Feedforward Neural Network:

Consist of two layers:

- **Layer 1:**

  The fully connected neural network is where the block processes its input token after self-attention has included the appropriate context in its representation. It is made up of two layers. The first layer is four times the size of the model (Since **GPT2** small is 768, this network would have 768*4 = 3072 units). Why four times? That is just the size the original transformer rolled with (model dimension was 512 and layer #1 in that model was 2048). This seems to give transformer models enough representational capacity to handle the tasks that have been thrown at them so far.

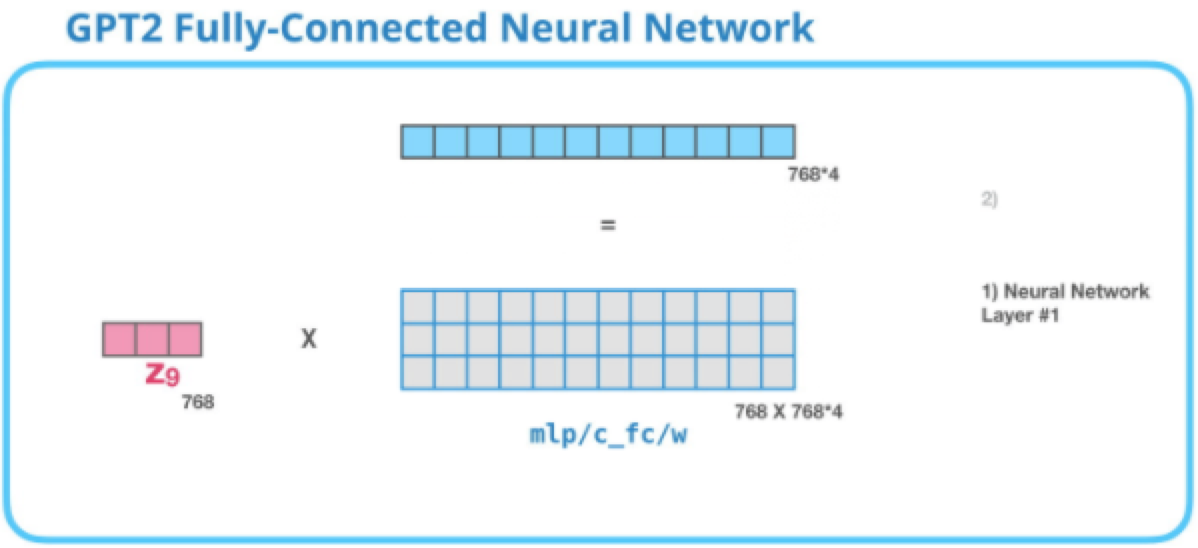

GPT2 Fully-Connected Neural Network
768*4
2)
=
1) Neural Network Layer #1
X
Z9
768
768 X 768*4
mlp/c_fc/w

Figure 4.3.4.2.1: Fully Connected layer

- **Layer 2: Projecting to model dimension**

The second layer projects the result from the first layer back into model dimension (768 for the small **GPT2**). The result of this multiplication is the result of the transformer block for this token.

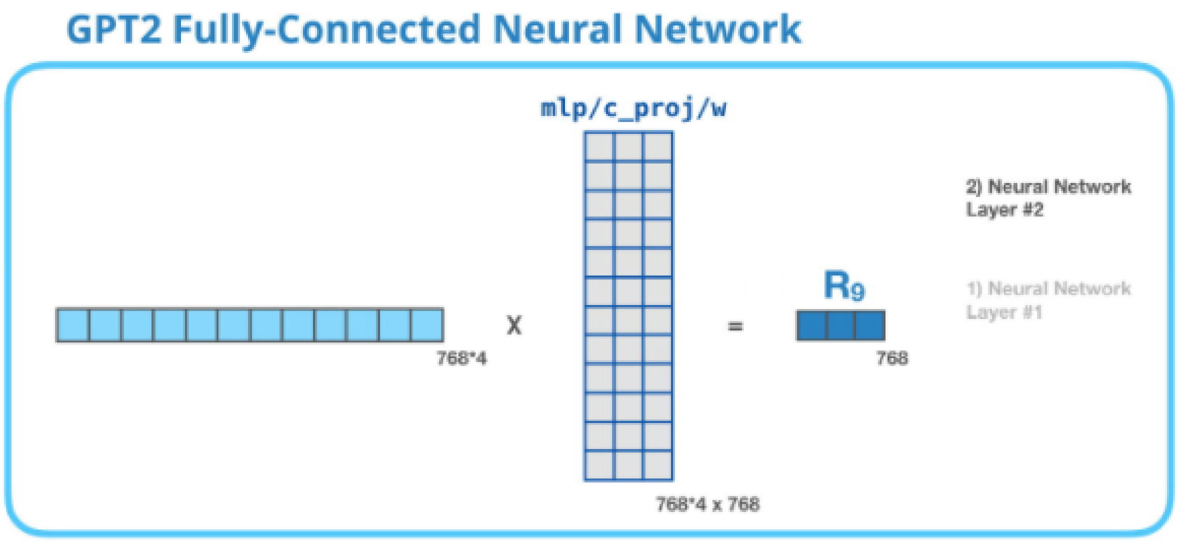

GPT2 Fully-Connected Neural Network
mlp/c_proj/w
2) Neural Network Layer #2
1) Neural Network Layer #1
R9
X
=
768*4
768
768*4 x 768

Figure 4.3.4.2.2: Fully Connected layer2

4.3.4.3 Apply softmax:

It is combined with the feedforward part, take its output and apply softmax on it to get probabilities and choose the next predicted pixel by choose the highest probability.

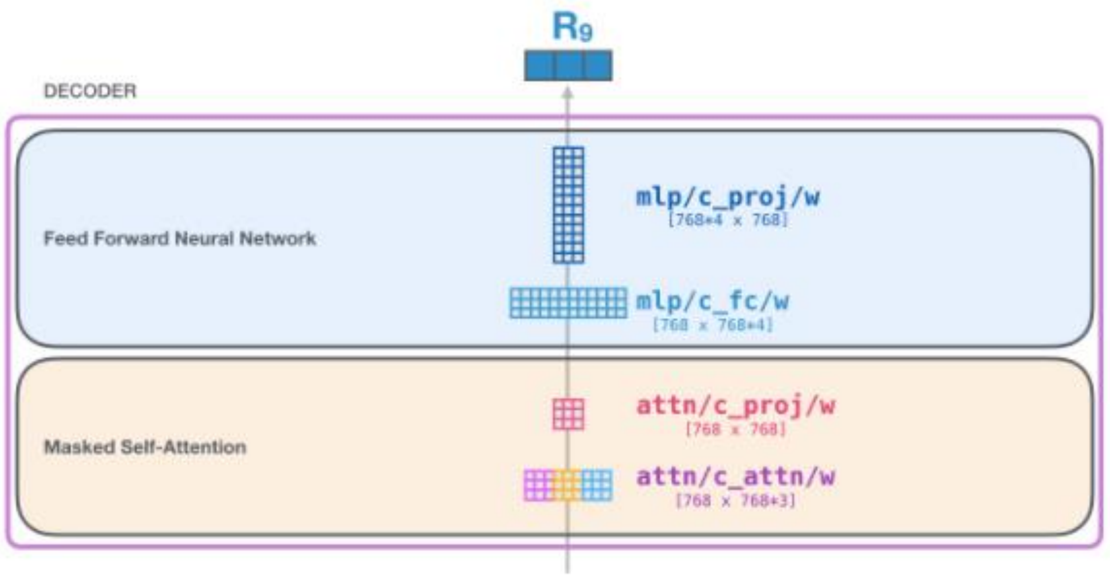


Figure 4.3.4.2.3: Decoder Block Summary

4.4 technologies used in implementation:

- **GitHub** is a Git repository hosting service, but it adds many of its own features. While Git is a command line tool, **GitHub** provides a Web-based graphical interface. It also provides access control and several collaboration features, such as a wikis and basic task management tools for every project.

Figure 4.4.1 Github logo

- **PyCharm** is an integrated development environment used in computer programming, specifically for the Python language. It is developed by the Czech company JetBrains.

Figure 4.4.2 pycharm logo

- Colaboratory, or "Colab" for short, allows you to write and execute Python in your browser, with
  - Zero configuration required.
  - Free access to **GPUs**
  - Easy sharing

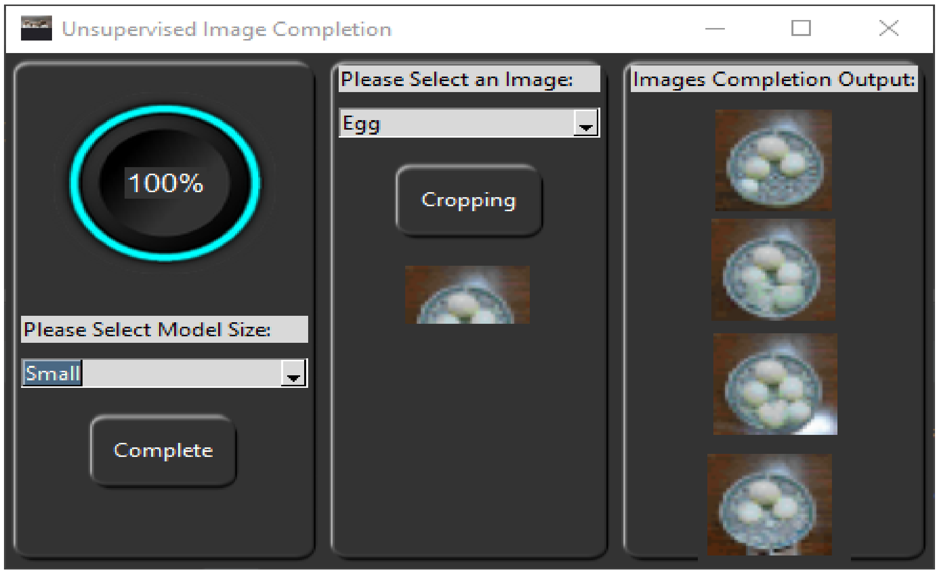
Figure 4.4.3 google colab logo

- **TensorFlow** is an end-to-end open-source platform for machine learning. It has a comprehensive, flexible ecosystem of tools, libraries and community resources that lets researchers push the state-of-the-art in **ML** and developers easily build and deploy **ML** powered applications.

Figure 4.4.4 TensorFlow logo

4.5 GUI Design:

Our model needs only single form to represent everything as shown in Figure 4.5.1, and the system will be described in detail in the next chapter.


Unsupervised Image Completion

Please Select Model Size: Small

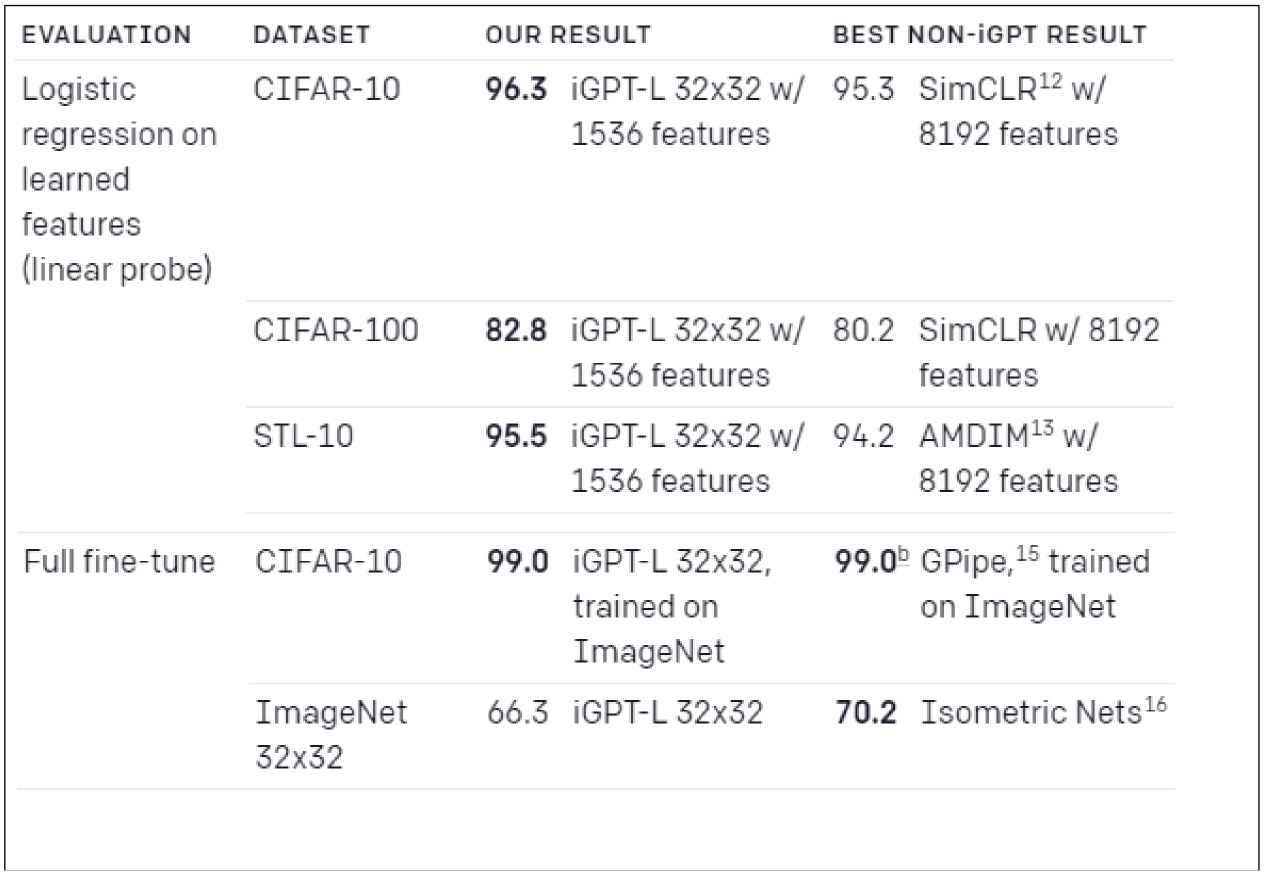
Please Select an Image: Egg

Figure 4.5.1 UI design of the model

4.6 Testing procedures and experimental results:

One way to measure representation quality is to fine-tune for image classification. Fine-tuning adds a small classification head to the model, used to optimize a classification objective and adapts all weights.
Another approach for measuring representation quality uses the pre-trained model as a feature extractor.

In the model achieves state-of-the-art performance on several classification datasets and near state-of-the-art unsupervised accuracy on ImageNet as shown in the next Table.



EVALUATION	DATASET	OUR RESULT		BEST NON-iGPT RESULT	
Logistic regression on learned features (linear probe)	CIFAR-10	**96.3**	iGPT-L 32x32 w/ 1536 features	95.3	SimCLR[12] w/ 8192 features
	CIFAR-100	**82.8**	iGPT-L 32x32 w/ 1536 features	80.2	SimCLR w/ 8192 features
	STL-10	**95.5**	iGPT-L 32x32 w/ 1536 features	94.2	AMDIM[13] w/ 8192 features
Full fine-tune	CIFAR-10	**99.0**	iGPT-L 32x32, trained on ImageNet	**99.0**[b]	GPipe,[15] trained on ImageNet
	ImageNet 32x32	66.3	iGPT-L 32x32	**70.2**	Isometric Nets[16]

Table 4.6.1 accuracies over some Dataset

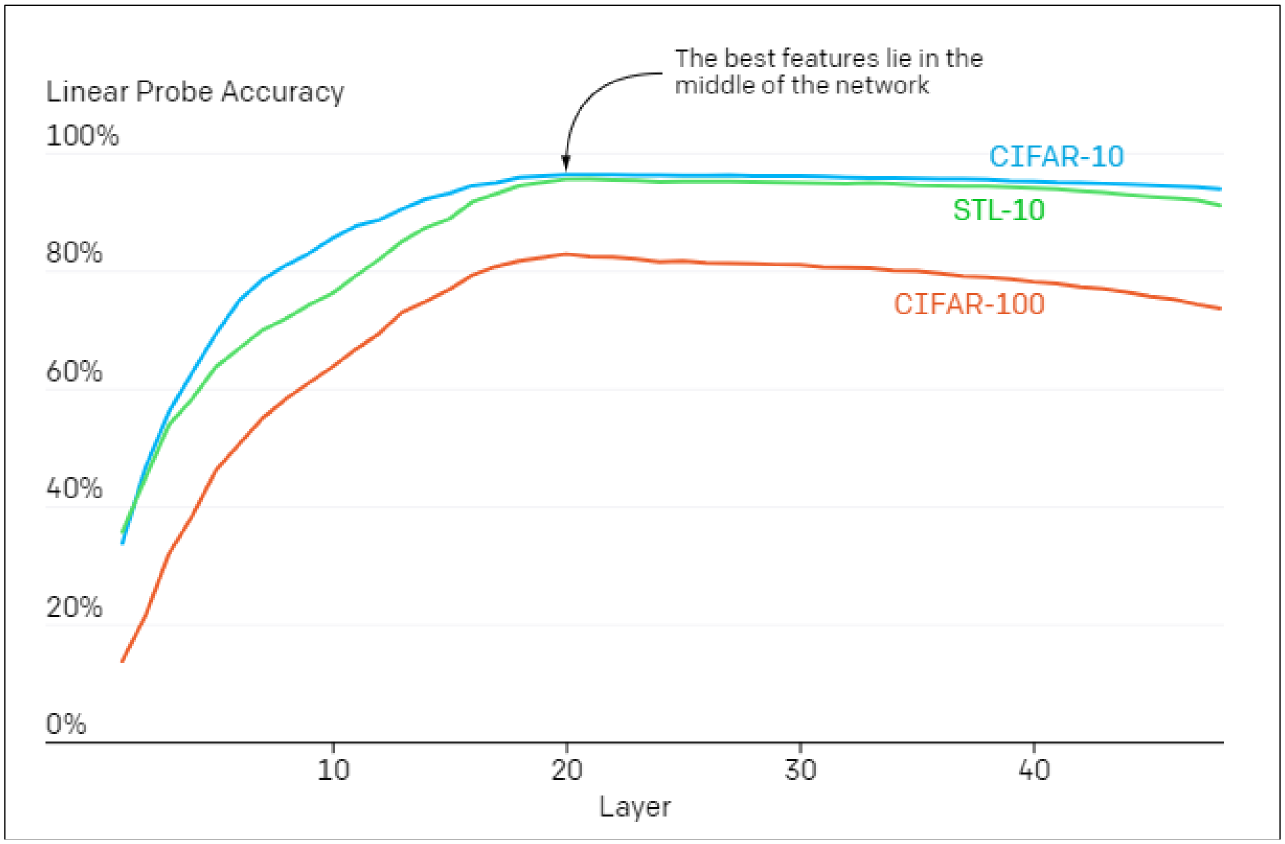


Figure 4.6.1 Linea probe Accuracy

Feature quality depends heavily on the layer we choose to evaluate. In contrast with supervised models, the best features for these generative models lie in the middle of the network.

Our next result establishes the link between generative performance and feature quality. We find that both increasing the scale of our models and training for more iterations result in better generative performance, which directly translates into better feature quality.

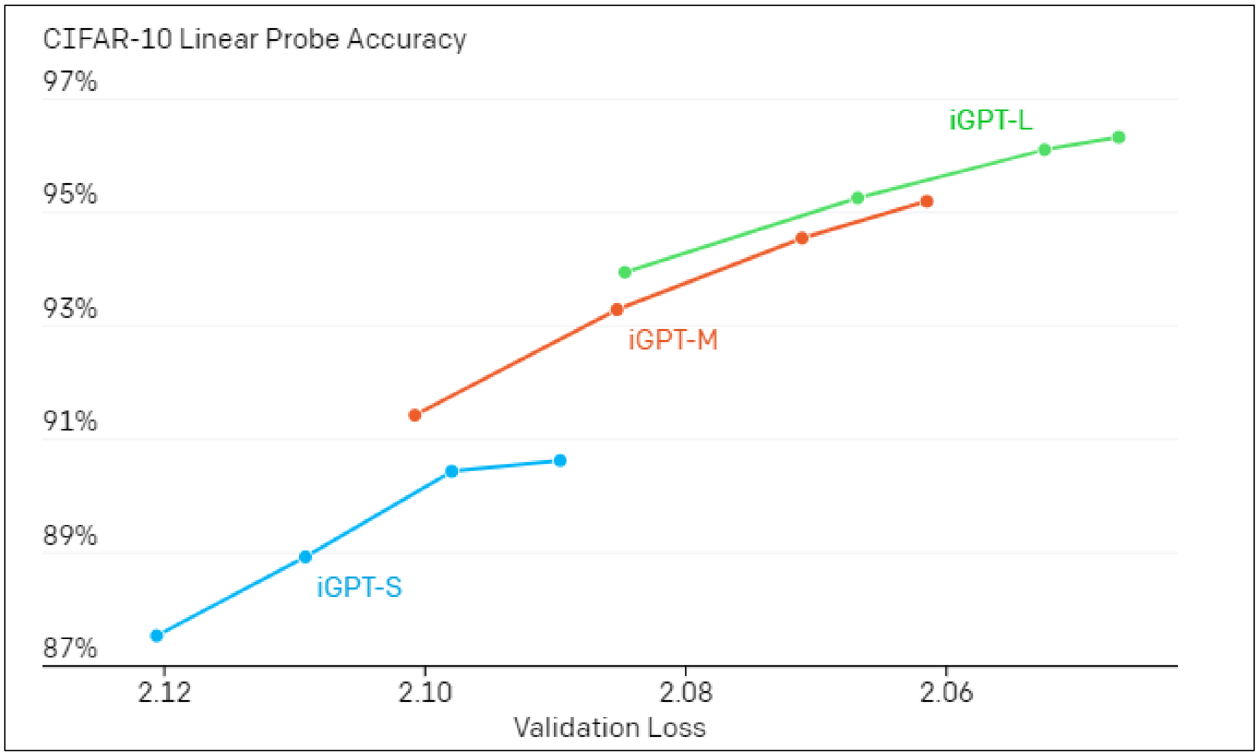


Figure 4.6.2 validation loss

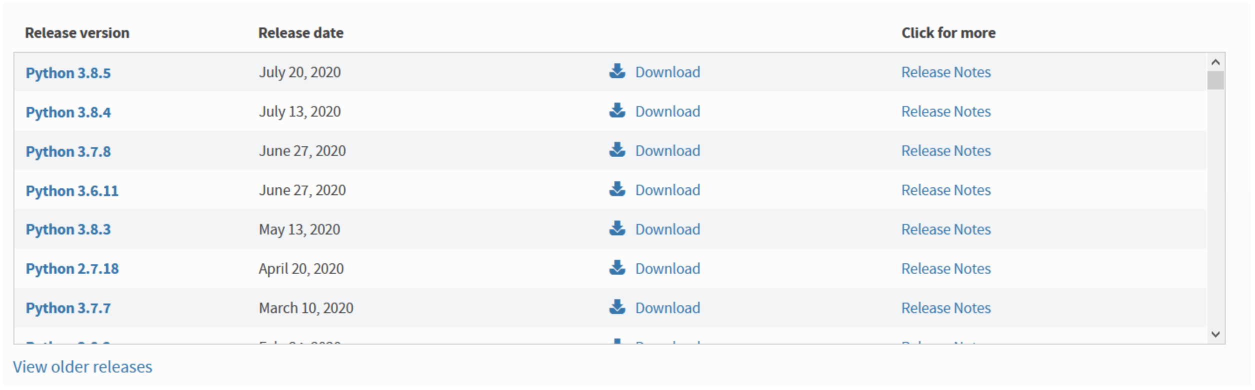
Each line tracks a model throughout generative pre-training: the dotted markers denote checkpoints at steps 131K, 262K, 524K, and 1000K. The positive slopes suggest a link between improved generative performance and improved feature quality. Larger models also produce better features than smaller models.

When we evaluate our features using linear probes on CIFAR-10, CIFAR-100, and STL-10, we outperform features from all supervised and unsupervised transfer algorithms. Our results are also compelling in the full fine-tuning setting.

5- User Manual

5.1 Required Third Party Tools:

1. Python must be downloaded from https://www.python.org/downloads/

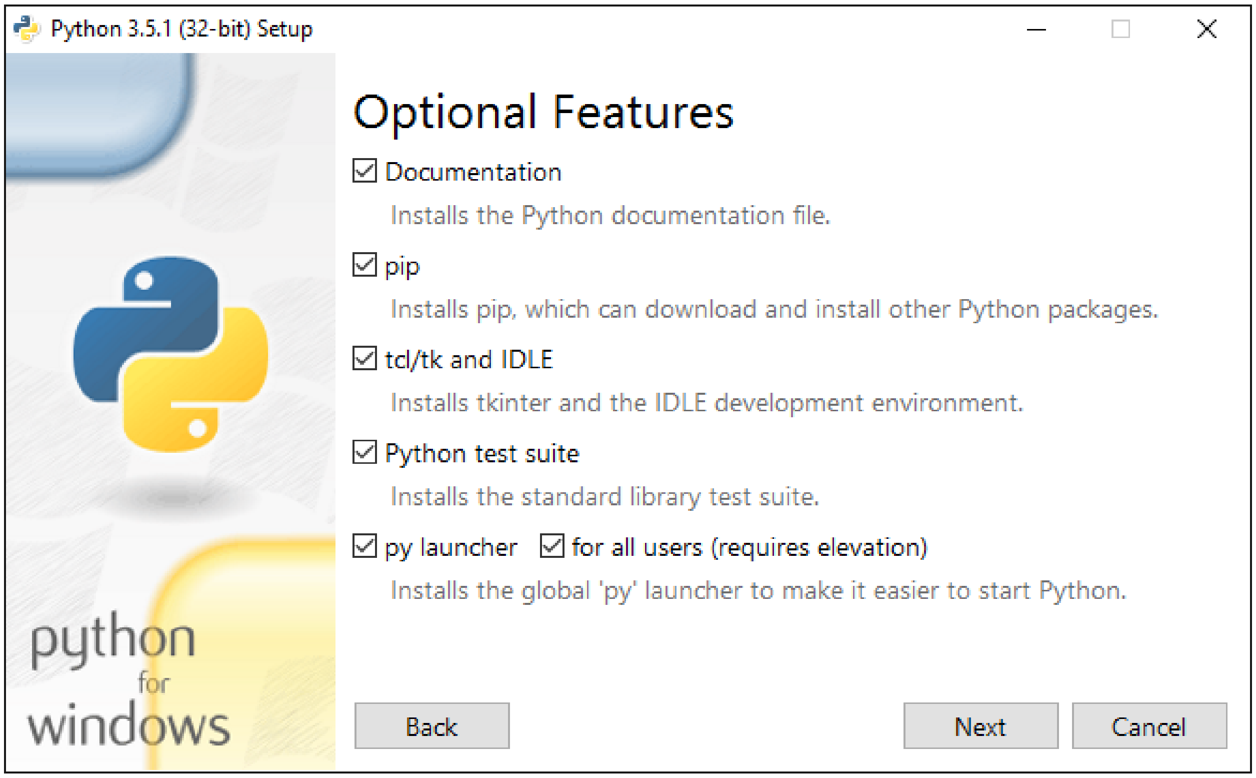


Release version	Release date		Click for more
Python 3.8.5	July 20, 2020	Download	Release Notes
Python 3.8.4	July 13, 2020	Download	Release Notes
Python 3.7.8	June 27, 2020	Download	Release Notes
Python 3.6.11	June 27, 2020	Download	Release Notes
Python 3.8.3	May 13, 2020	Download	Release Notes
Python 2.7.18	April 20, 2020	Download	Release Notes
Python 3.7.7	March 10, 2020	Download	Release Notes

View older releases

Figure 5.1.1 -- Python download.

2. Run python exe as shown.



Python 3.5.1 (32-bit) Setup

Optional Features

☑ Documentation
Installs the Python documentation file.

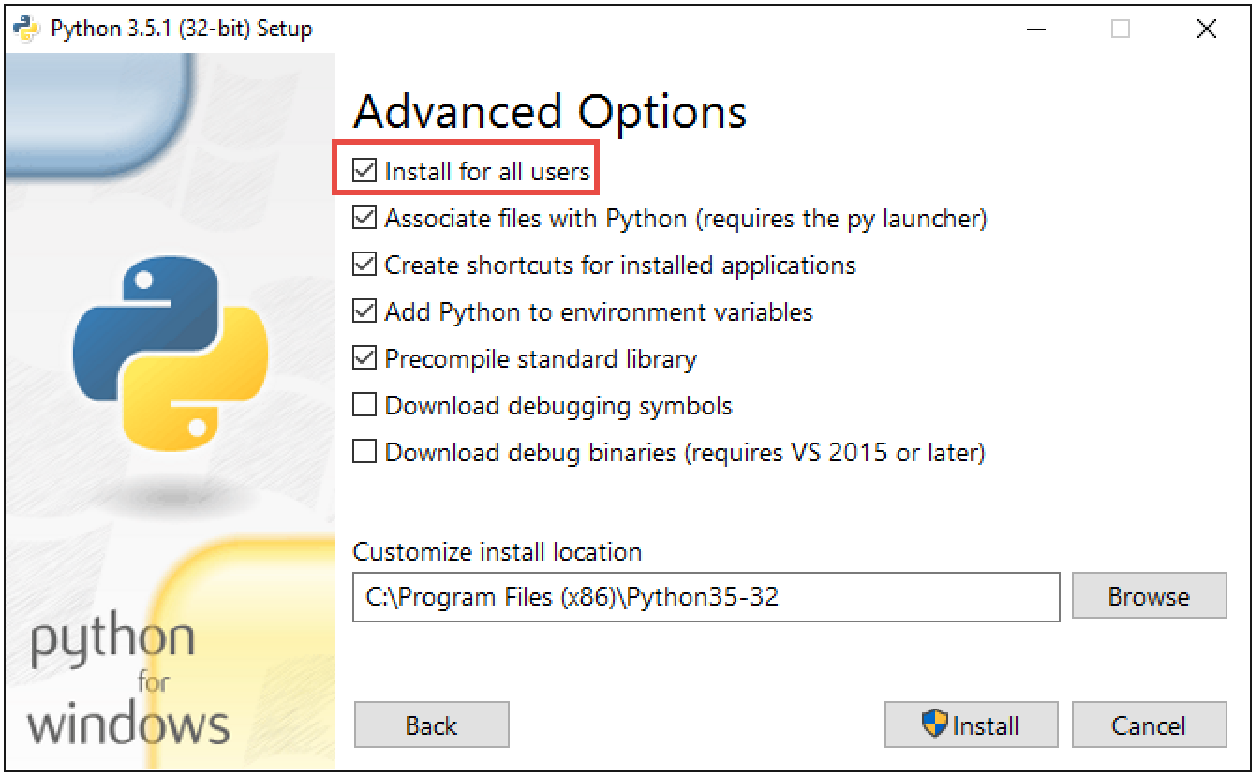
☑ pip
Installs pip, which can download and install other Python packages.

☑ tcl/tk and IDLE
Installs tkinter and the IDLE development environment.

☑ Python test suite
Installs the standard library test suite.

☑ py launcher ☑ for all users (requires elevation)
Installs the global 'py' launcher to make it easier to start Python.

Figure 5.1.1 --Python installation 1



Python 3.5.1 (32-bit) Setup

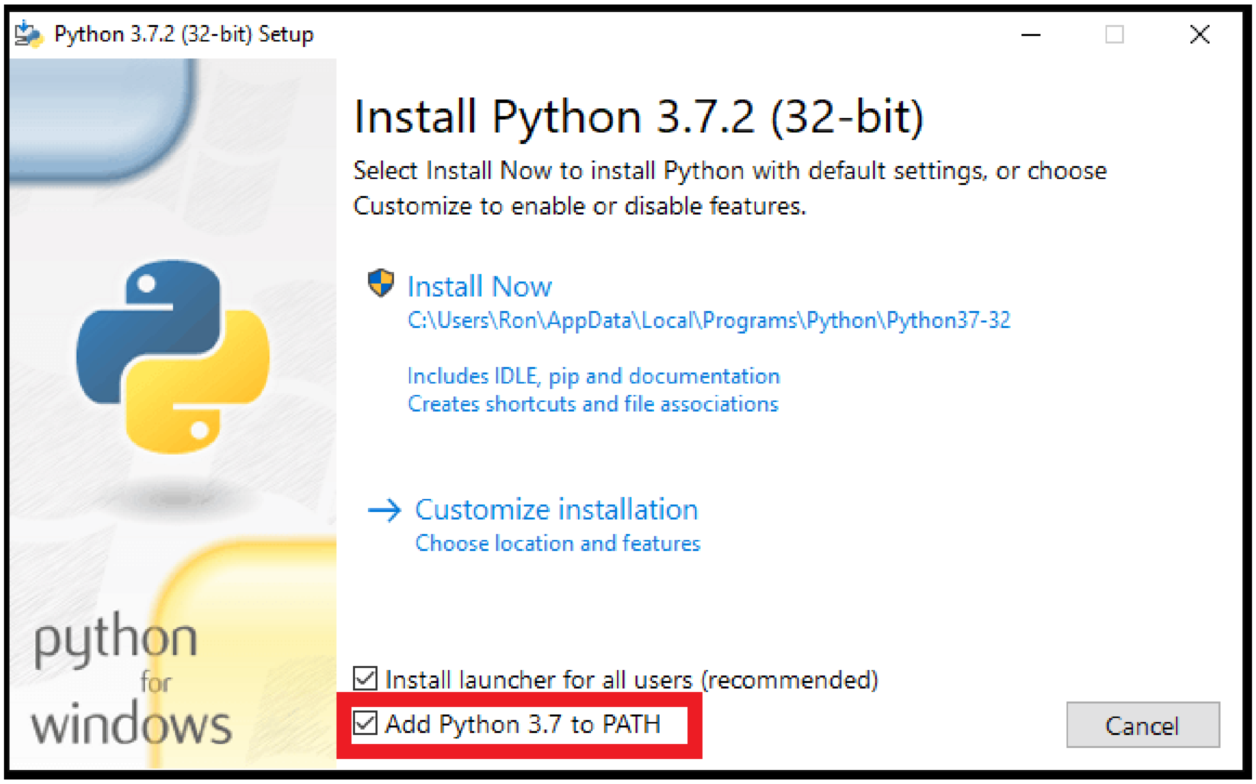
Advanced Options

- ☑ Install for all users
- ☑ Associate files with Python (requires the py launcher)
- ☑ Create shortcuts for installed applications
- ☑ Add Python to environment variables
- ☑ Precompile standard library
- ☐ Download debugging symbols
- ☐ Download debug binaries (requires VS 2015 or later)

Customize install location

C:\Program Files (x86)\Python35-32

Figure 5.1.2--Python installation 2



Python 3.7.2 (32-bit) Setup

Install Python 3.7.2 (32-bit)

Select Install Now to install Python with default settings, or choose Customize to enable or disable features.

Install Now
C:\Users\Ron\AppData\Local\Programs\Python\Python37-32

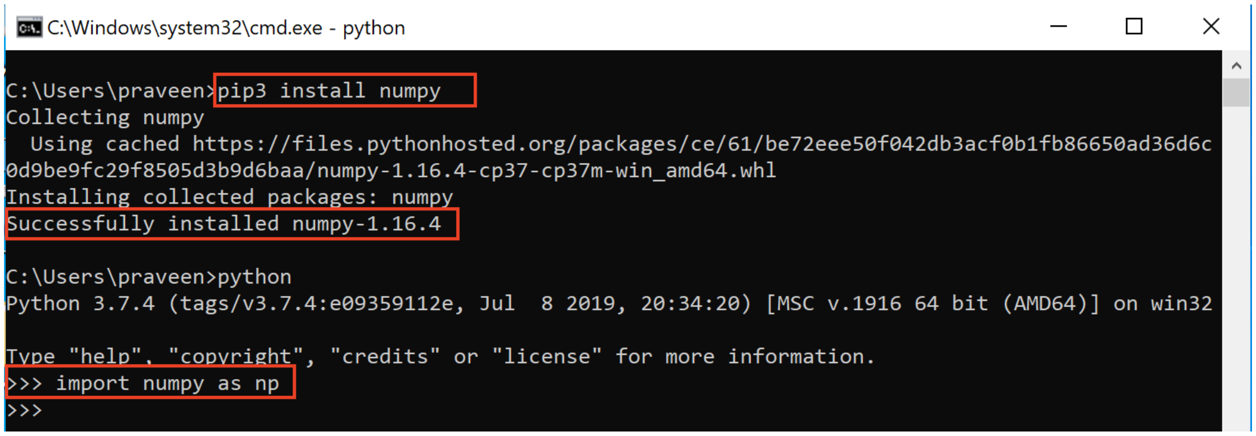
Includes IDLE, pip and documentation
Creates shortcuts and file associations

→ Customize installation
Choose location and features

- ☑ Install launcher for all users (recommended)
- ☑ Add Python 3.7 to PATH

Figure 5.1.3-Python installation 3

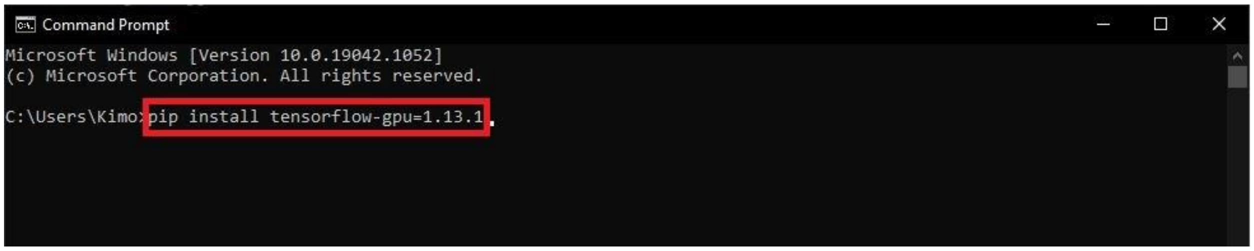
3. Install numpy package from **CMD**


C:\Windows\system32\cmd.exe - python

```
C:\Users\praveen>pip3 install numpy
Collecting numpy
  Using cached https://files.pythonhosted.org/packages/ce/61/be72eee50f042db3acf0b1fb86650ad36d6c
0d9be9fc29f8505d3b9d6baa/numpy-1.16.4-cp37-cp37m-win_amd64.whl
Installing collected packages: numpy
Successfully installed numpy-1.16.4

C:\Users\praveen>python
Python 3.7.4 (tags/v3.7.4:e09359112e, Jul  8 2019, 20:34:20) [MSC v.1916 64 bit (AMD64)] on win32

Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy as np
>>>
```

Figure 5.1.5-Numpy installation

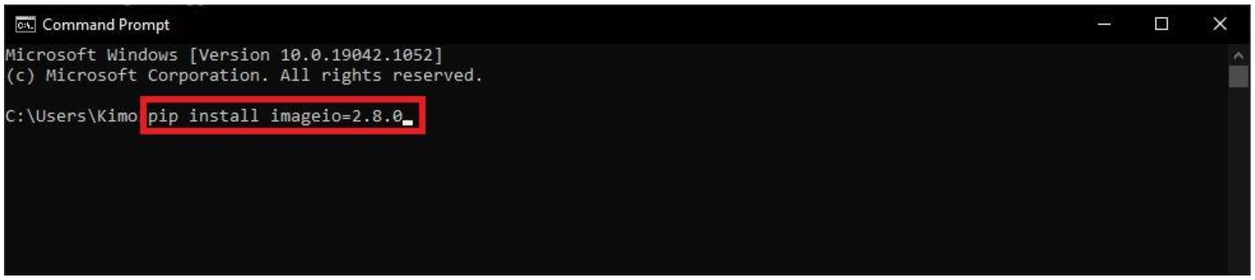
4. Install TensorFlow package from **CMD.**


Command Prompt

```
Microsoft Windows [Version 10.0.19042.1052]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Kimo>pip install tensorflow-gpu=1.13.1
```

Figure 5.1.6-Tensorflow installation

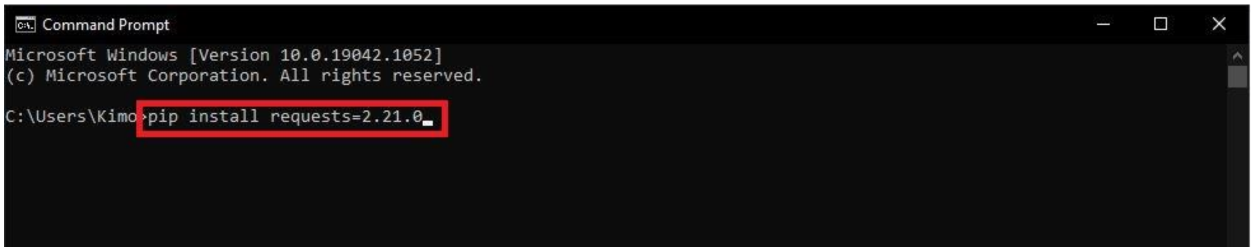
5. Install imageio package from **CMD.**


Command Prompt

```
Microsoft Windows [Version 10.0.19042.1052]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Kimo>pip install imageio=2.8.0_
```

Figure 5.1.7-Imageio installation

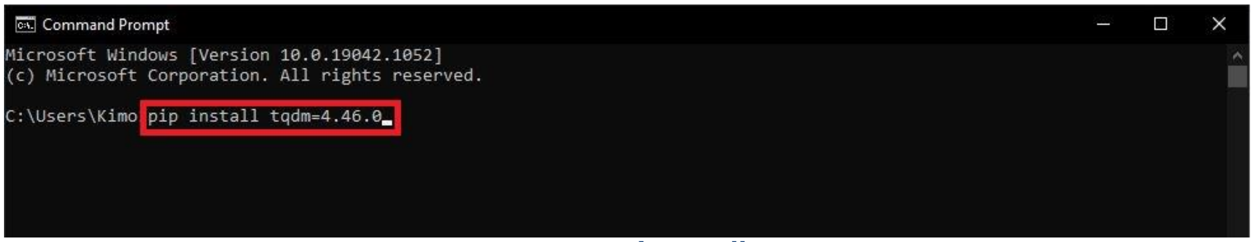
6. Install requests package from **CMD.**


Command Prompt

```
Microsoft Windows [Version 10.0.19042.1052]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Kimo>pip install requests=2.21.0_
```

Figure 5.1.8-requests installation

7. Install tqdm package from **CMD.**

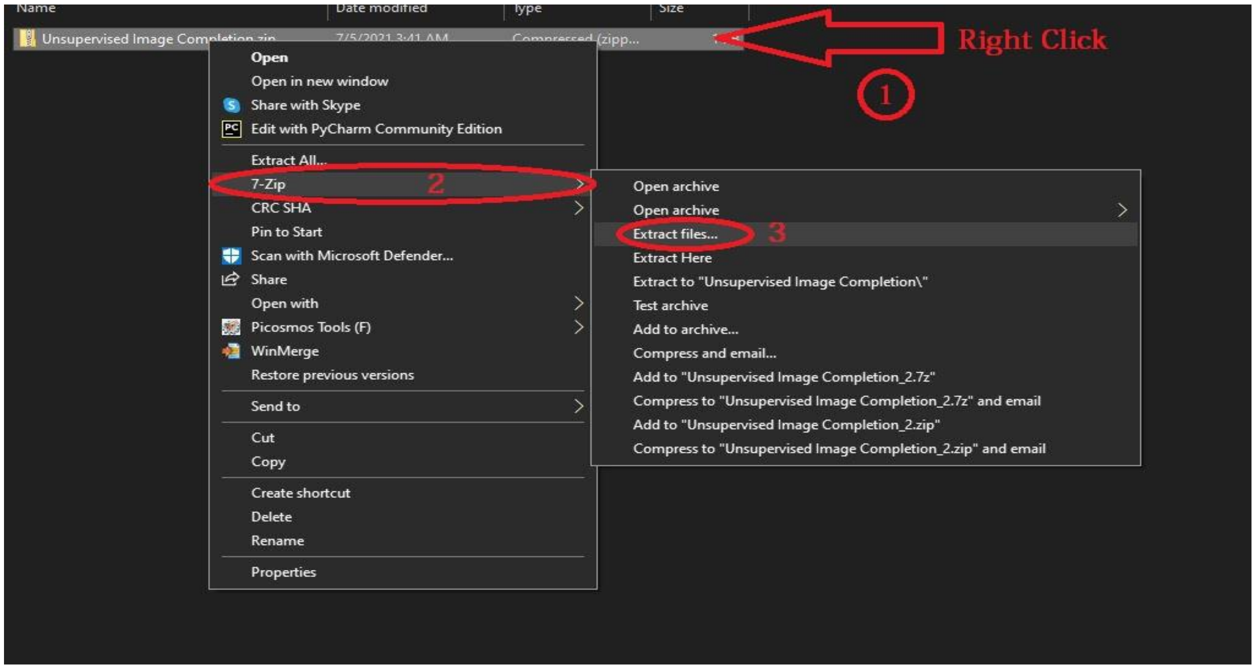


```
Command Prompt
Microsoft Windows [Version 10.0.19042.1052]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Kimo pip install tqdm=4.46.0_
```

Figure 5.1.9-tqdm installation

5.2 Running Steps:

1. Extract Project

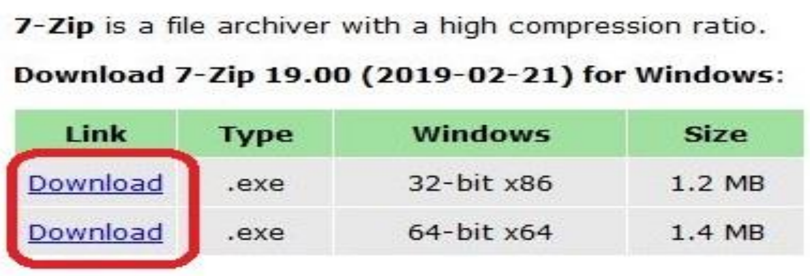


```
Name                                   Date modified        Type              Size
Unsupervised Image Completion.zip      7/5/2021 3:41 AM     Compressed (zipp...
```

Right Click
1
Open
Open in new window
Share with Skype
Edit with PyCharm Community Edition
Extract All...
7-Zip
2
CRC SHA
Pin to Start
Scan with Microsoft Defender...
Share
Open with
Picosmos Tools (F)
WinMerge
Restore previous versions
Send to
Cut
Copy
Create shortcut
Delete
Rename
Properties
Open archive
Open archive
Extract files...
3
Extract Here
Extract to "Unsupervised Image Completion\"
Test archive
Add to archive...
Compress and email...
Add to "Unsupervised Image Completion_2.7z"
Compress to "Unsupervised Image Completion_2.7z" and email
Add to "Unsupervised Image Completion_2.zip"
Compress to "Unsupervised Image Completion_2.zip" and email

Figure 5.2.1-Extract Project

You Can install 7-Zip Program from: https://www.7-zip.org/ then download the suitable version for your device



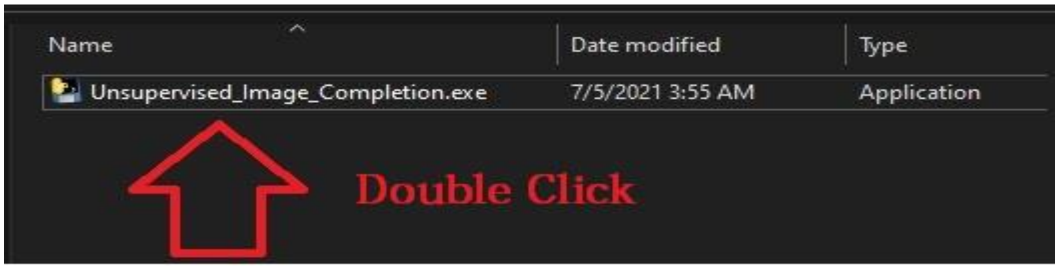
7-Zip is a file archiver with a high compression ratio.

Download 7-Zip 19.00 (2019-02-21) for Windows:

Link	Type	Windows	Size
Download	.exe	32-bit x86	1.2 MB
Download	.exe	64-bit x64	1.4 MB

Figure 5.2.2 7-zip Download

2. Open Project exe File



Name	Date modified	Type
Unsupervised_Image_Completion.exe	7/5/2021 3:55 AM	Application

Figure 5.2.3 Open Project exe

3. Select an Image.

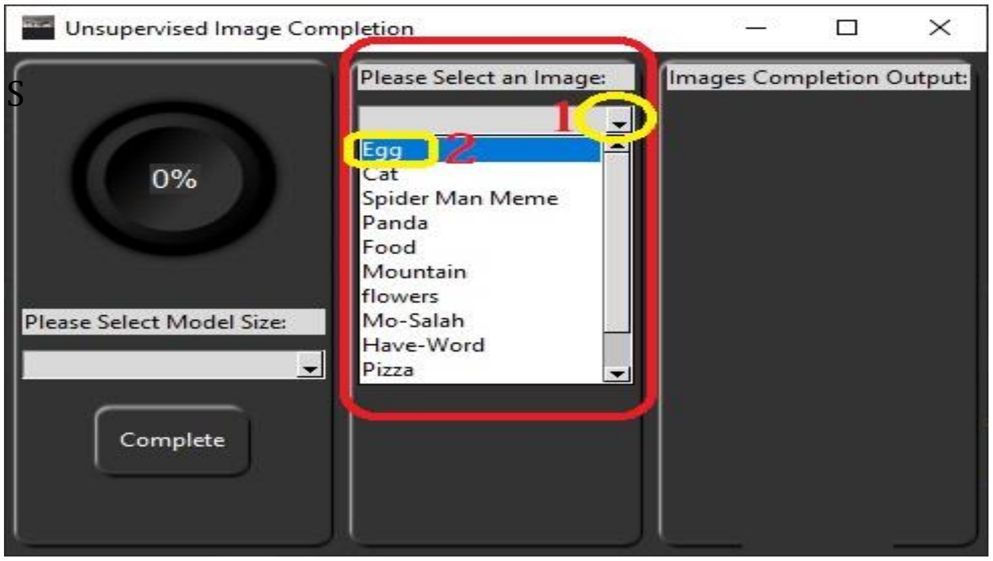


Figure 5.2.4 Select an Image.

4. Image will Appear.

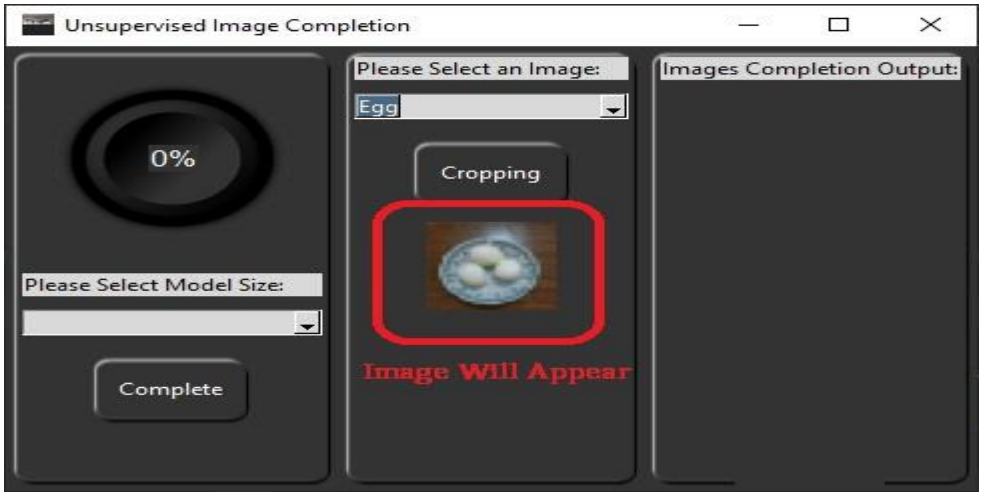

Unsupervised Image Completion
0%
Please Select Model Size:
Complete
Please Select an Image:
Egg
Cropping
Image Will Appear
Images Completion Output:

Figure 5.2.5 Image Selected.

5. Press Cropping Button to Crop Image

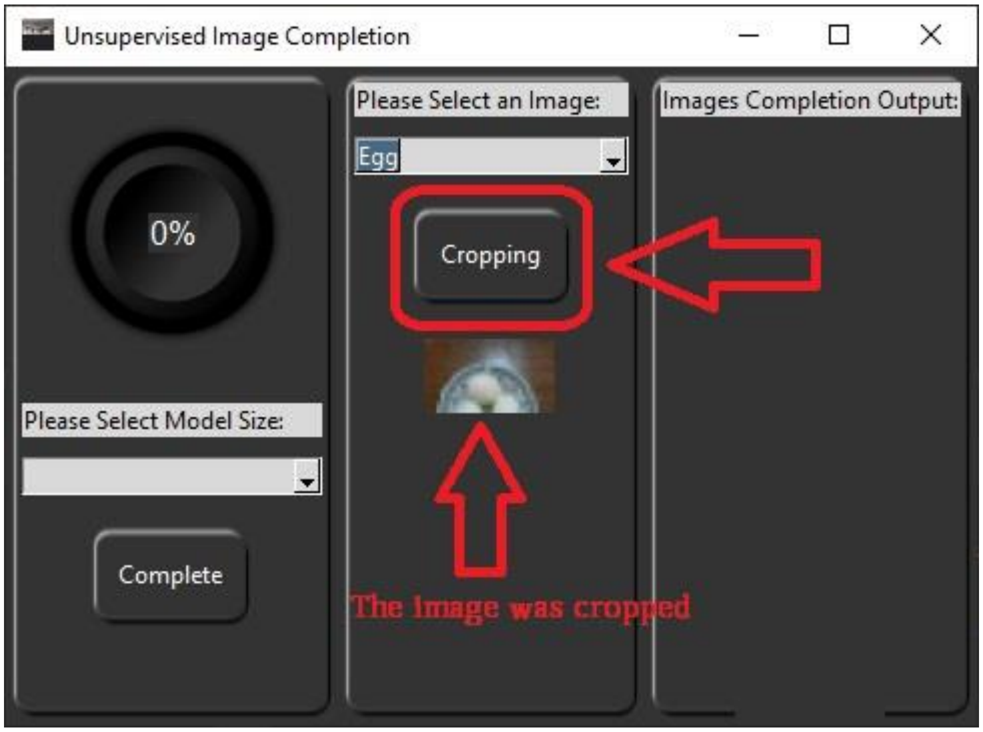

Unsupervised Image Completion
0%
Please Select Model Size:
Complete
Please Select an Image:
Egg
Cropping
Images Completion Output:
The image was cropped

Figure 5.2.6 Cropping Button

6. Select Model Size

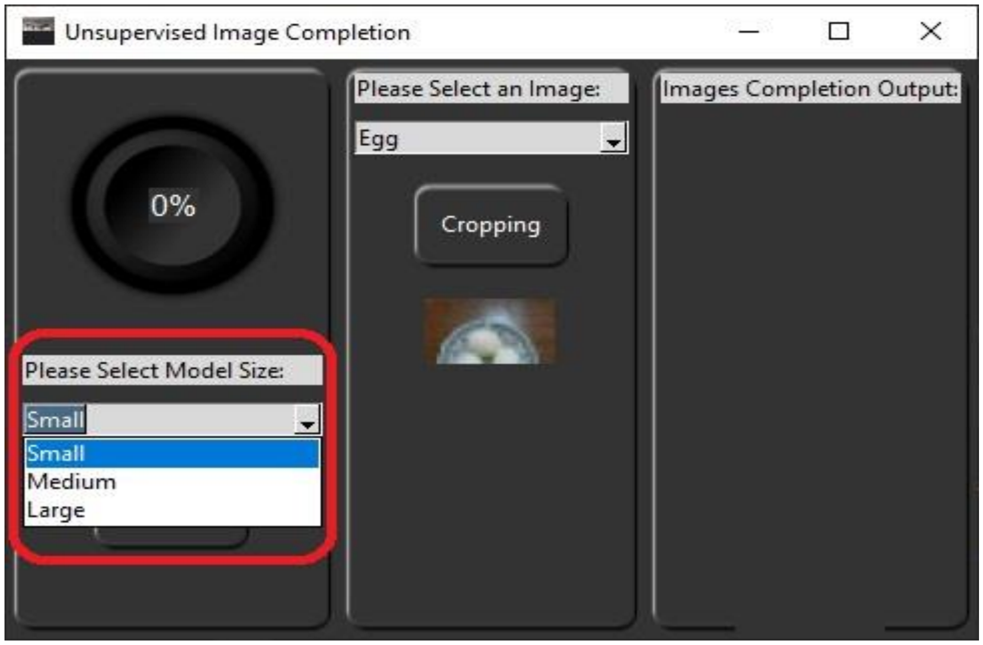

Unsupervised Image Completion
0%
Please Select Model Size:
Small
Small
Medium
Large
Please Select an Image:
Egg
Cropping
Images Completion Output:

Figure 5.2.7 Select Model Size

7. Press On Complete Button

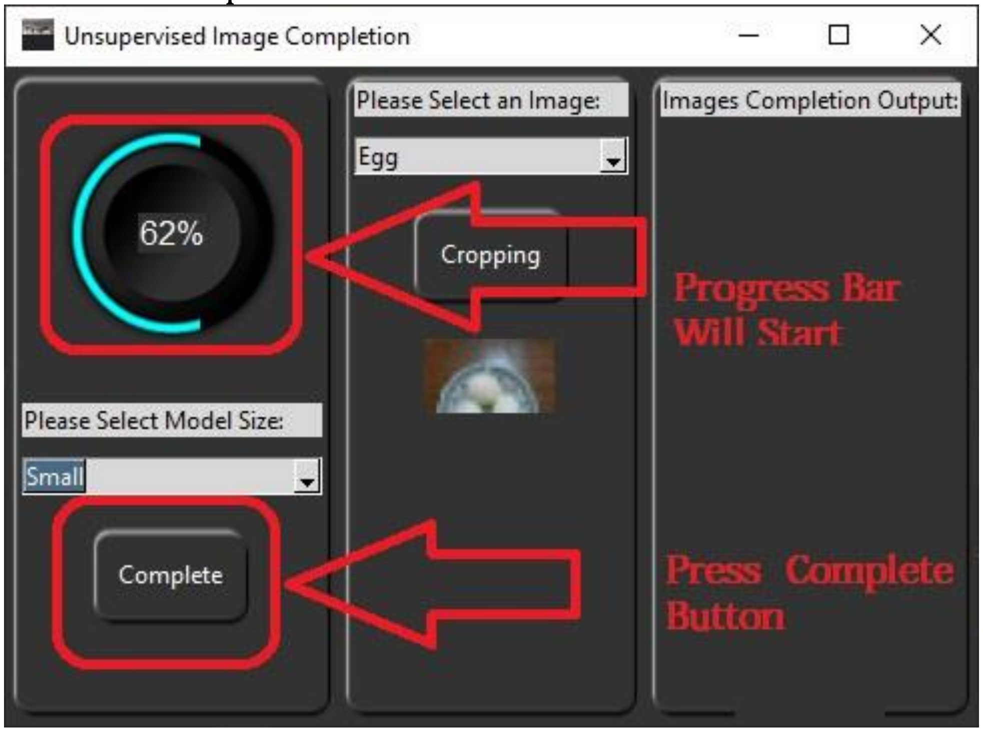


Figure 5.2.8 Press Complete

8. Output Will Appear After Process Finished

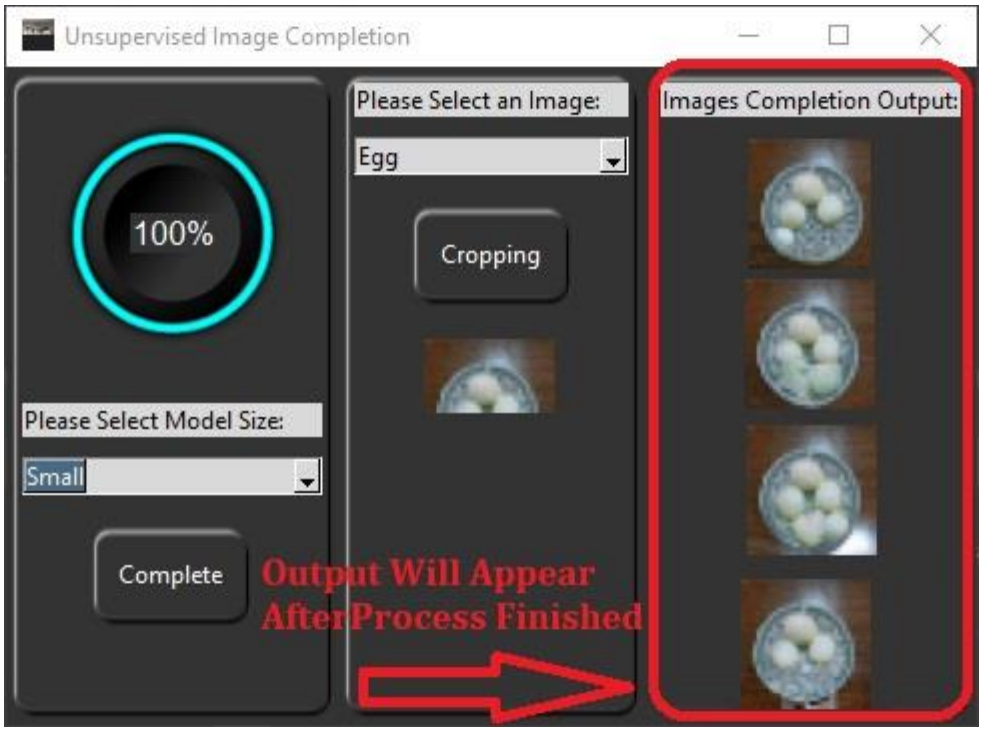

Unsupervised Image Completion
100%
Please Select Model Size:
Small
Complete
Please Select an Image:
Egg
Cropping
Images Completion Output:
Output Will Appear After Process Finished

Figure 5.2.9 Egg Small Model Output

Note: **To use Custom Image, you need to use Full Version on Google Colab Will Explain all steps at Section 5.3**

- Example For Medium Model

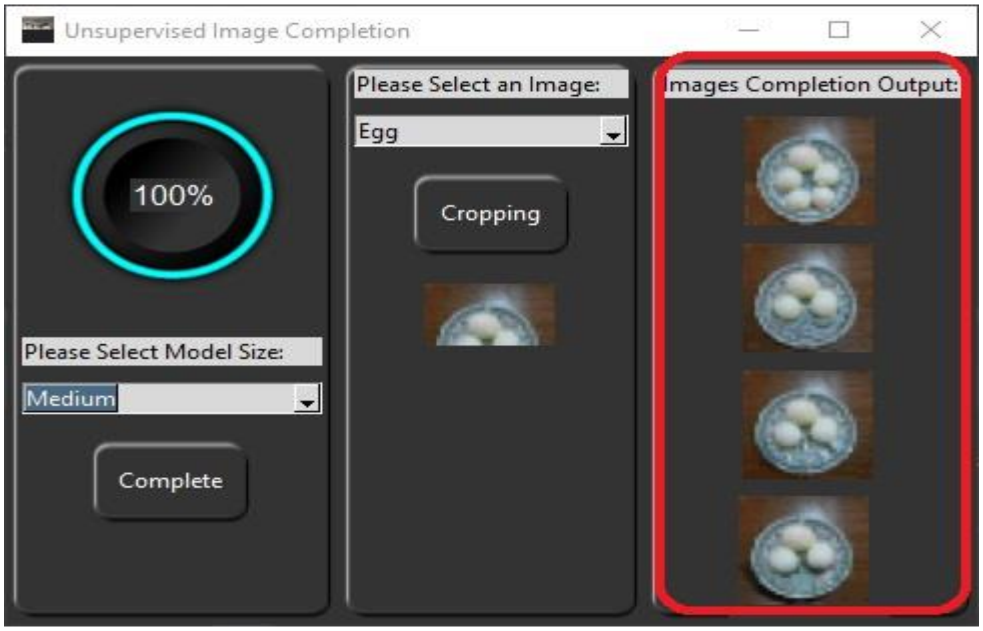
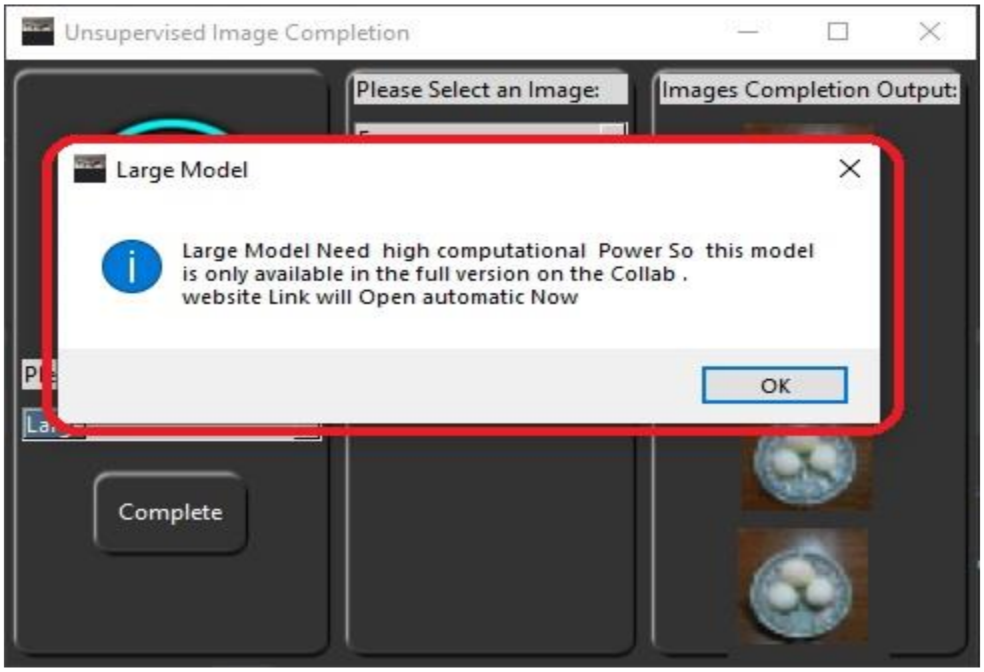

Unsupervised Image Completion
100%
Please Select Model Size:
Medium
Complete
Please Select an Image:
Egg
Cropping
Images Completion Output:

Figure 5.2.10 Egg Medium Model Output

- Example for Large Model



Large Model Need high computational Power So this model is only available in the full version on the Collab . website Link will Open automatic Now

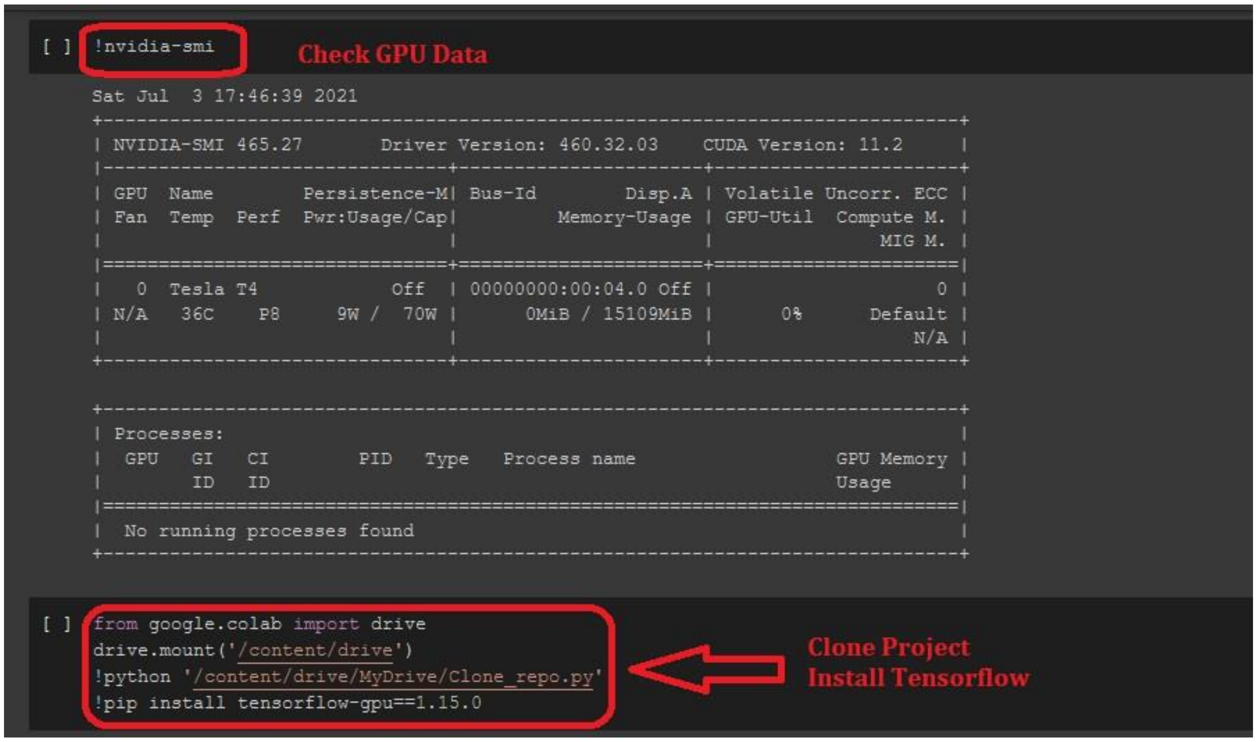
Figure 5.2.11 Large Model

Large Model Need high computational Power So this model is only available in the full version on Google Colab.

5.3 Step To use Full Version on Google Colab:

Run Cells :

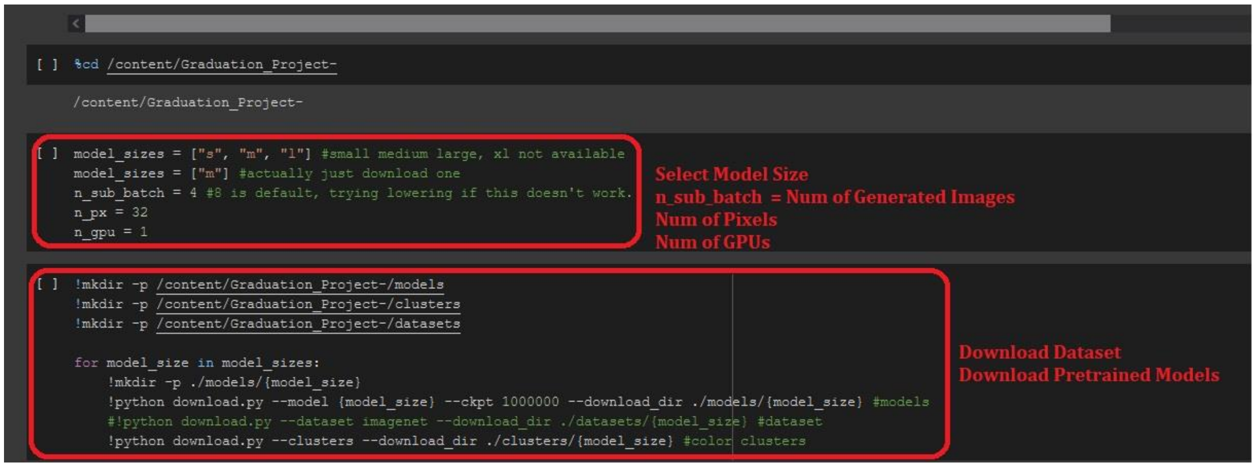
1. Clone Project from GitHub & Install TensorFlow



```
[ ] !nvidia-smi

    Sat Jul  3 17:46:39 2021
    +-----------------------------------------------------------------------------+
    | NVIDIA-SMI 465.27       Driver Version: 460.32.03    CUDA Version: 11.2     |
    |-------------------------------+----------------------+----------------------+
    | GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
    | Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
    |                               |                      |               MIG M. |
    |===============================+======================+======================|
    |   0  Tesla T4            Off  | 00000000:00:04.0 Off |                    0 |
    | N/A   36C    P8     9W /  70W |      0MiB / 15109MiB |      0%      Default |
    |                               |                      |                  N/A |
    +-------------------------------+----------------------+----------------------+

    +-----------------------------------------------------------------------------+
    | Processes:                                                                  |
    |  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
    |        ID   ID                                                   Usage      |
    |=============================================================================|
    |  No running processes found                                                 |
    +-----------------------------------------------------------------------------+

[ ] from google.colab import drive
    drive.mount('/content/drive')
    !python '/content/drive/MyDrive/Clone_repo.py'
    !pip install tensorflow-gpu==1.15.0
```

Figure 5.3.1 Colab first, second cell

2. Set Model Size, Num of Images to Generate, Num of pixels, **GPUs**, then download Dataset, Pretrained Models

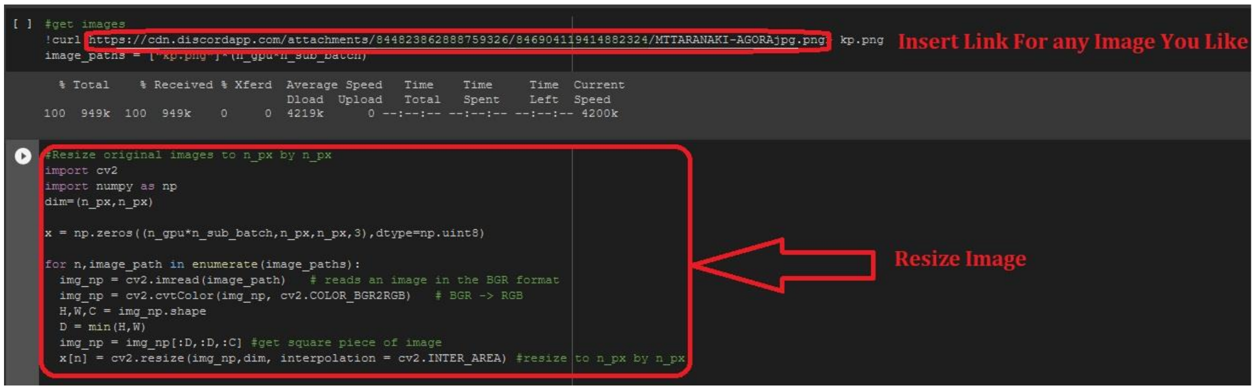


```
[ ] %cd /content/Graduation_Project-

    /content/Graduation_Project-

[ ] model_sizes = ["s", "m", "l"] #small medium large, xl not available
    model_sizes = ["m"] #actually just download one
    n_sub_batch = 4 #8 is default, trying lowering if this doesn't work.
    n_px = 32
    n_gpu = 1

[ ] !mkdir -p /content/Graduation_Project-/models
    !mkdir -p /content/Graduation_Project-/clusters
    !mkdir -p /content/Graduation_Project-/datasets

    for model_size in model_sizes:
        !mkdir -p ./models/{model_size}
        !python download.py --model {model_size} --ckpt 1000000 --download_dir ./models/{model_size} #models
        #!python download.py --dataset imagenet --download_dir ./datasets/{model_size} #dataset
        !python download.py --clusters --download_dir ./clusters/{model_size} #color clusters
```

Select Model Size
n_sub_batch = Num of Generated Images
Num of Pixels
Num of GPUs
Download Dataset
Download Pretrained Models

Figure 5.3.2 Colab Set Variables, Down

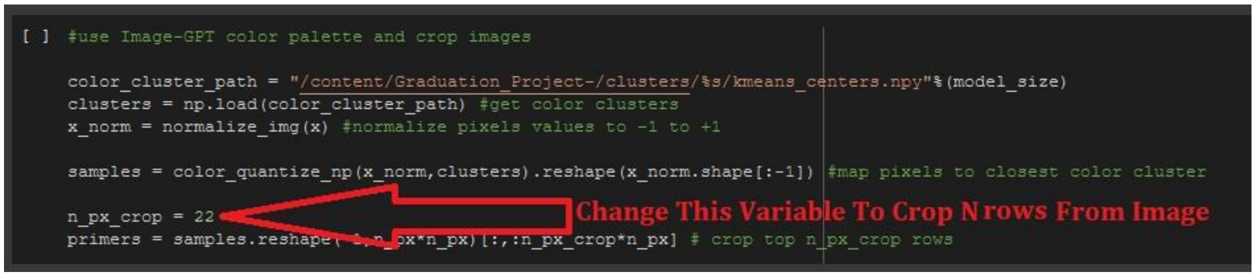
3. Choose Custom image using



```
[ ] #get images
    !curl https://cdn.discordapp.com/attachments/844823862888759326/846904119414882324/MTTARANAKI-AGORAjpg.png > kp.png
    image_paths = ["kp.png"]*(n_gpu*n_sub_batch)

      % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                     Dload  Upload   Total   Spent    Left  Speed
    100  949k  100  949k    0     0  4219k      0 --:--:-- --:--:-- --:--:-- 4200k

#Resize original images to n_px by n_px
import cv2
import numpy as np
dim=(n_px,n_px)

x = np.zeros((n_gpu*n_sub_batch,n_px,n_px,3),dtype=np.uint8)

for n,image_path in enumerate(image_paths):
  img_np = cv2.imread(image_path)   # reads an image in the BGR format
  img_np = cv2.cvtColor(img_np, cv2.COLOR_BGR2RGB)   # BGR -> RGB
  H,W,C = img_np.shape
  D = min(H,W)
  img_np = img_np[:D,:D,:C] #get square piece of image
  x[n] = cv2.resize(img_np,dim, interpolation = cv2.INTER_AREA) #resize to n_px by n_px
```

Figure 5.3.3 Colab Insert Image and Res

4. Set Number of rows Cropped from image



```
[ ] #use Image-GPT color palette and crop images

    color_cluster_path = "/content/Graduation_Project-/clusters/%s/kmeans_centers.npy"%(model_size)
    clusters = np.load(color_cluster_path) #get color clusters
    x_norm = normalize_img(x) #normalize pixels values to -1 to +1

    samples = color_quantize_np(x_norm,clusters).reshape(x_norm.shape[:-1]) #map pixels to closest color cluster

    n_px_crop = 22
    primers = samples.reshape(-1,n_px*n_px)[:,:n_px_crop*n_px] # crop top n_px_crop rows
```

Figure 5.3.4 Colab Crop image

5. Run all Cell in order and images Result will appear on last cell.

6- Conclusion and Future Work

6.1 Conclusion

We have shown that by trading off 2-D knowledge for scale[35] and by choosing predictive features from the middle of the network, a sequence transformer can be competitive with top convolutional nets for unsupervised image classification. Notably, we achieved our results by directly applying the **GPT-2** language model to image generation. Our results suggest that due to its simplicity and generality, a sequence transformer given sufficient compute might ultimately be an effective way to learn excellent features in many domains. Our results suggest that generative image modeling continues to be a promising route to learn high-quality unsupervised image representations. Simply predicting pixels learns state of the art representations for low resolution datasets. In high resolution settings, our approach is also competitive with other self-supervised results on ImageNet.

6.2 Future Work

- ✓ Additionally, we observed that our approach requires large models order to learn high quality representations and give more accurate outputs. Expand **IGPT** to **IGPT-XL**, by training the model on **100 million** web Images and train on **ImageNet**.
- ✓ generate higher resolution images, using **IGPT-XL** allows to produce images with size of 64×64.
- ✓ Trying to enhance the output of the model to make the images clearer.

References

[1] Hinton, G. E., Osindero, S., and Teh, Y.-W. A fast learning algorithm for deep belief nets. Neural computation, 18 (7):1527–1554, 2006.

[2] Vincent, P., Larochelle, H., Bengio, Y., and Manzagol, P.-A. Extracting and composing robust features with denoising autoencoders. In Proceedings of the 25th international conference on Machine learning, pp. 1096–1103, 2008.

[3] Lee, H., Grosse, R., Ranganath, R., and Ng, A. Y. Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations. In Proceedings of the 26th annual international conference on machine learning, pp. 609–616, 2009.

[4] Mohamed, A.-r., Dahl, G., and Hinton, G. Deep belief networks for phone recognition. 2009.

[5] Lasserre, J. A., Bishop, C. M., and Minka, T. P. Principled hybrids of generative and discriminative models. In 2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06), volume 1, pp. 87–94. IEEE, 2006.

[6] Nair, V. and Hinton, G. E. Rectified linear units improve restricted boltzmann machines. In Proceedings of the 27th international conference on machine learning (ICML-10), pp. 807–814, 2010.

[7] Glorot, X. and Bengio, Y. Understanding the difficulty of training deep feedforward neural networks. In Proceedings of the thirteenth international conference on artificial intelligence and statistics, pp. 249–256, 2010.

[8] Ioffe, S. and Szegedy, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. arXiv preprint arXiv:1502.03167, 2015.

[9] Ba, J. L., Kiros, J. R., and Hinton, G. E. Layer normalization. arXiv preprint arXiv:1607.06450, 2016.

[10] Erhan, D., Bengio, Y., Courville, A., Manzagol, P.-A., Vincent, P., and Bengio, S. Why does unsupervised pretraining help deep learning? Journal of Machine Learning Research, 11(Feb):625–660, 2010.

[11] Coates, A., Ng, A., and Lee, H. An analysis of single layer networks in unsupervised feature learning. In Proceedings of the fourteenth international conference on artificial intelligence and statistics, pp. 215–223, 2011.

[12] Huang, P.-S., Avron, H., Sainath, T. N., Sindhwani, V., and Ramabhadran, B. Kernel methods match deep neural networks on timit. In 2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), pp. 205–209. IEEE, 2014.

[13] May, A., Garakani, A. B., Lu, Z., Guo, D., Liu, K., Bellet, A., Fan, L., Collins, M., Hsu, D., Kingsbury, B., et al. Kernel approximation methods for speech recognition. arXiv preprint arXiv:1701.03577, 2017.

[14] Krizhevsky, A., Sutskever, I., and Hinton, G. E. Imagenet classification with deep convolutional neural networks. In Advances in neural information processing systems, pp. 1097–1105, 2012.

[15] Graves, A. and Jaitly, N. Towards end-to-end speech recognition with recurrent neural networks. In International conference on machine learning, pp. 1764–1772, 2014.

[16] Paine, T. L., Khorrami, P., Han, W., and Huang, T. S. An analysis of unsupervised pre-training in light of recent advances. arXiv preprint arXiv:1412.6597, 2014.

[17] Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., and Dean, J. Distributed representations of words and phrases and their compositionality. In Advances in neural information processing systems, pp. 3111–3119, 2013.

[18] Dai, A. M. and Le, Q. V. Semi-supervised sequence learning. In Advances in neural information processing systems, pp. 3079–3087, 2015.

[19] Peters, M. E., Neumann, M., Iyyer, M., Gardner, M., Clark, C., Lee, K., and Zettlemoyer, L. Deep contextualized word representations. arXiv preprint arXiv:1802.05365, 2018.

[20] Howard, J. and Ruder, S. Universal language model fine-tuning for text classification. arXiv preprint arXiv:1801.06146, 2018.

[21] Radford, A., Narasimhan, K., Salimans, T., and Sutskever, I. Improving language understanding by generative pretraining. 2018.

[22] Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805, 2018.

[23] Doersch, C., Gupta, A., and Efros, A. A. Unsupervised visual representation learning by context prediction. In Proceedings of the IEEE International Conference on Computer Vision, pp. 1422–1430, 2015.

[24] Oord, A. v. d., Li, Y., and Vinyals, O. Representation learning with contrastive predictive coding. arXiv preprint arXiv:1807.03748, 2018.

[25] Gomez, A. N., Ren, M., Urtasun, R., and Grosse, R. B. The reversible residual network: Backpropagation without storing activations. In Advances in neural information processing systems, pp. 2214–2224, 2017.

[26] Kolesnikov, A., Zhai, X., and Beyer, L. Revisiting selfsupervised visual representation learning. In Proceedings of the IEEE conference on Computer Vision and Pattern Recognition, pp. 1920–1929, 2019.

[27] H´enaff, O. J., Razavi, A., Doersch, C., Eslami, S., and Oord, A. v. d. Data-efficient image recognition with contrastive predictive coding. arXiv preprint arXiv:1905.09272, 2019.
[28] He, K., Fan, H., Wu, Y., Xie, S., and Girshick, R. Momentum contrast for unsupervised visual representation learning. arXiv preprint arXiv:1911.05722, 2019.
[29] Misra, I. and van der Maaten, L. Self-supervised learning of pretext-invariant representations. arXiv preprint arXiv:1912.01991, 2019.
[30] Chen, T., Kornblith, S., Norouzi, M., and Hinton, G. A simple framework for contrastive learning of visual representations. arXiv preprint arXiv:2002.05709, 2020.
[31] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is all you need. In Advances in neural information processing systems, pp. 5998–6008, 2017.
[32] Larochelle, H. and Murray, I. The neural autoregressive distribution estimator. In Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics, pp. 29–37, 2011.
[33] Oord, A. v. d., Kalchbrenner, N., and Kavukcuoglu, K. Pixel recurrent neural networks. arXiv preprint arXiv:1601.06759, 2016.

[34] Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., and Sutskever, I. Language models are unsupervised multitask learners. 2019.

[35] Sutton, R. “The Bitter Lesson.”. 2019.

[36] Jay Alammar. The Illustrated GPT-2 (Visualizing Transformer Language Models). August 12, 2019. https://jalammar.github.io/illustrated-gpt2/

[37] Jay Alammar. The Illustrated Transformer. June 27, 2018. https://jalammar.github.io/illustrated-transformer/

[38] Mark Chen., Alec Radford., & Rewon Child. (2020) “Generative Pretraining from Pixels”, OpenAI.

[39] Torralba, A., Fergus, R., and Freeman, W. T. 80 million tiny images: A large data set for nonparametric object and scene recognition. IEEE transactions on pattern analysis and machine intelligence, 30(11):1958–1970, 2008.