Intro to Neural Nets

Generative Models

Today's Agenda

Generative Language Models

- A simple auto-regressive language model
- Machine translation
- Large language models
- Working with Pre-trained LLMs

Generative Image Models

- Basic ideas
- Some Types of Models (VAEs, GANs)

Generative Language Model

Training a Generative Language Model

- We train a model to predict the next token given a sequence of observed input tokens.
- The goal is to obtain hidden layers with weights useful for decoding at inference.
- We will not use this exact model for inference (at least not directly); we will reuse its weights.

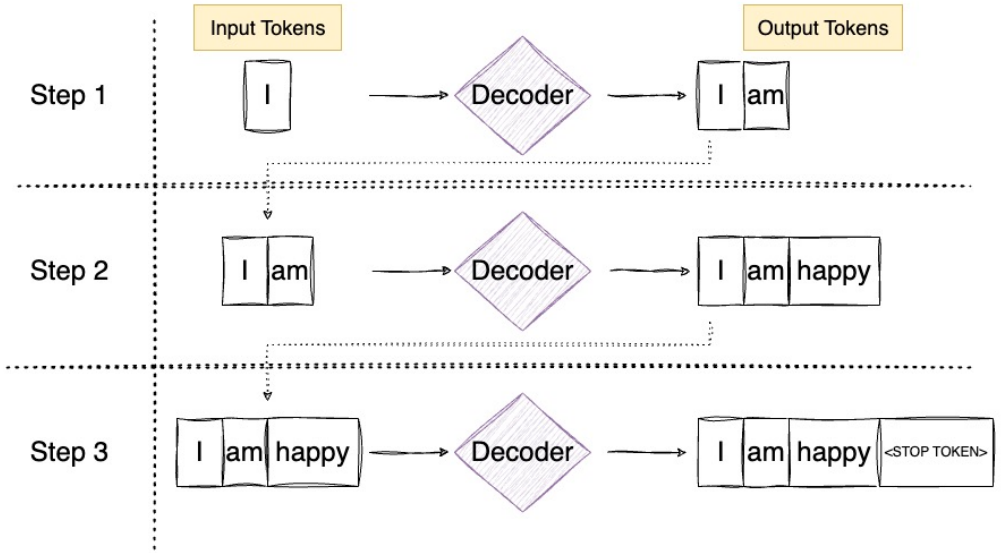
```
embedding_dim = 256
hidden_dim = 1024

inputs = layers.Input(
    shape=(sequence_length,), dtype="int", name="token_ids"
)
x = layers.Embedding(vocabulary_size, embedding_dim)(inputs)
x = layers.GRU(hidden_dim, return_sequences=True)(x)
x = layers.Dropout(0.1)(x)
outputs = layers.Dense(vocabulary_size, activation="softmax")(x)
model = keras.Model(inputs, outputs)
```

Generative Language Model

Inference with Generative Language Models

- We create a slightly different set of hidden layers and initialize them with the weights we learned in our training model.
- We wrap the new layers with a feedback loop that takes the last prediction and passes it as input to the model again along with the most recent embedding that captures ‘what we’ve done so far’.

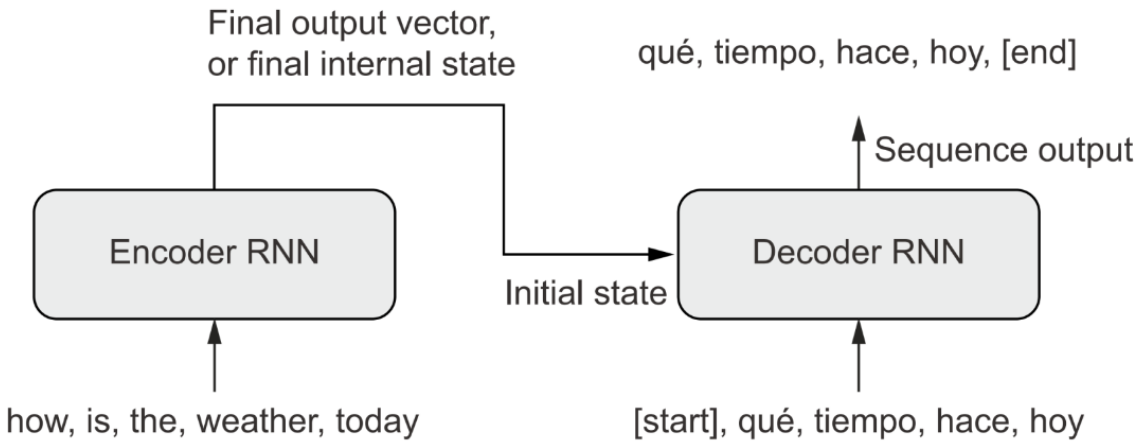
```
inputs = keras.Input(shape=(1,), dtype="int", name="token_ids")
input_state = keras.Input(shape=(hidden_dim,), name="state")

x = layers.Embedding(vocabulary_size, embedding_dim)(inputs)
x, output_state = layers.GRU(hidden_dim, return_state=True)(
    x, initial_state=input_state
)
outputs = layers.Dense(vocabulary_size, activation="softmax")(x)
generation_model = keras.Model(
    inputs=(inputs, input_state),
    outputs=(outputs, output_state),
)
generation_model.set_weights(model.get_weights())
```



Machine Translation

Similar Idea with Some Additions

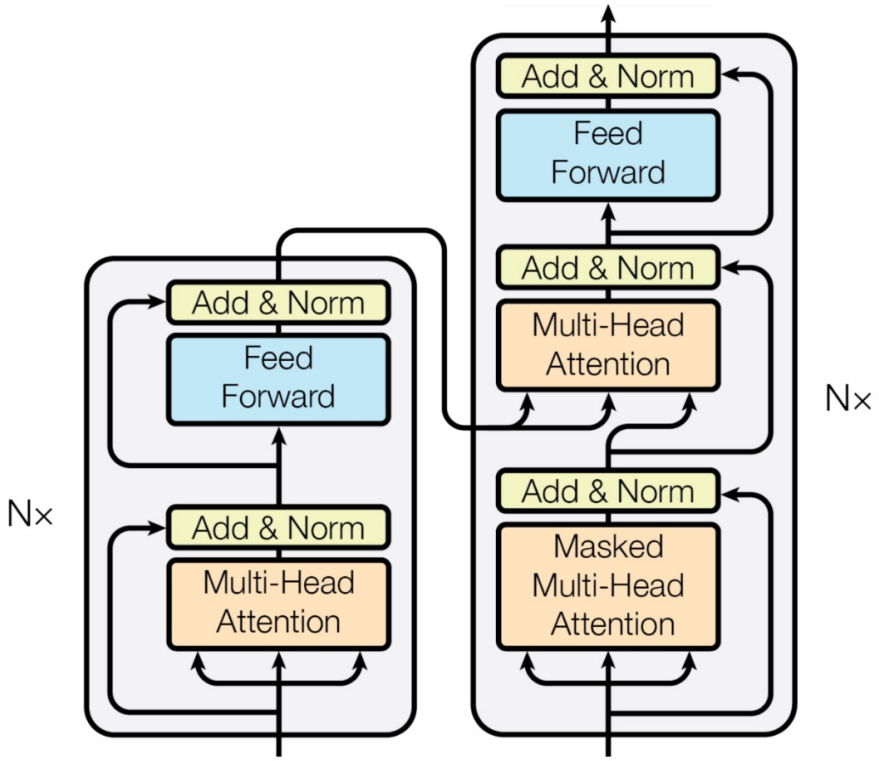
- Include an ‘encoder’ that first processes the entire input sequence to obtain an embedded representation of the input text.
- We are not just generating ‘plausible’ text now; we are generating plausible text that aligns with (i.e., conditioned upon) the input text.
- Our encoder can be bidirectional, but our decoder must be **unidirectional**. Why?



Machine Translation with Transformers

Replace RNNs with Transformers

- Transformers are an architecture that wraps MultiHeadAttention(). We have some residual connections inside and some LayerNormalization() operations, with a couple of Dense layers on the output.
- We also need to create a special masking layer to prevent the Transformer from looking ahead in the target sequence.
- Our decoder uses cross-attention. It compares the representation of the Spanish output seen thus far to the entire English sequence representation.



Brief Note: Sub-classing keras.Layer

Re-using a Block of Layers and Operations Quite a Bit? Just Declare a Custom Layer

- We achieve this by defining our new layer class, which requires an _init_() method (which runs once when the layer is declared in your network) and the call() method, which defines what happens inside the layer and what arguments it expects as input.

```python
from keras import layers

class TransformerDecoder(keras.Layer):
    def __init__(self, hidden_dim, intermediate_dim, num_heads):
        super().__init__()
        key_dim = hidden_dim // num_heads
        self.self_attention = layers.MultiHeadAttention(
            num_heads, key_dim, dropout=0.1
        )
        self.self_attention_layernorm = layers.LayerNormalization()
        self.feed_forward_1 = layers.Dense(intermediate_dim, activation="relu")
        self.feed_forward_2 = layers.Dense(hidden_dim)
        self.feed_forward_layernorm = layers.LayerNormalization()
        self.dropout = layers.Dropout(0.1)

    def call(self, inputs):
        residual = x = inputs
        x = self.self_attention(
            query=x, key=x, value=x, use_causal_mask=True
        )
        x = self.dropout(x)
        x = x + residual
        x = self.self_attention_layernorm(x)
        residual = x
        x = self.feed_forward_1(x)
        x = self.feed_forward_2(x)
        x = self.dropout(x)
        x = x + residual
        x = self.feed_forward_layernorm(x)
        return x
```

```python
from keras import layers

class TransformerDecoder(keras.Layer):
    def __init__(self, hidden_dim, intermediate_dim, num_heads):
        super().__init__()
        key_dim = hidden_dim // num_heads
        self.self_attention = layers.MultiHeadAttention(
            num_heads, key_dim, dropout=0.1
        )
        self.self_attention_layernorm = layers.LayerNormalization()
        self.feed_forward_1 = layers.Dense(intermediate_dim, activation="relu")
        self.feed_forward_2 = layers.Dense(hidden_dim)
        self.feed_forward_layernorm = layers.LayerNormalization()
        self.dropout = layers.Dropout(0.1)

    def call(self, inputs):
        residual = x = inputs
        x = self.self_attention(
            query=x, key=x, value=x, use_causal_mask=True
        )
        x = self.dropout(x)
        x = x + residual
        x = self.self_attention_layernorm(x)
        residual = x
        x = self.feed_forward_1(x)
        x = self.feed_forward_2(x)
        x = self.dropout(x)
        x = x + residual
        x = self.feed_forward_layernorm(x)
        return x
```

- The _init_() method declares what the layer contains (it creates these objects once, rather than instantiate them every time we do a forward pass). They need to be declared objects in memory so Tensorflow can work with them, calculate gradients, etc.

- The call() method is the order of operations with those objects when a forward pass occurs.

- We declare several sub-layers and their expected arguments in the _init()_ layer. We declare the sequence of operations involving those sub-layers in the call() method.

How Attention is Being Used

1. Self-Attention in Spanish Sequence (so far), i.e., with masking.

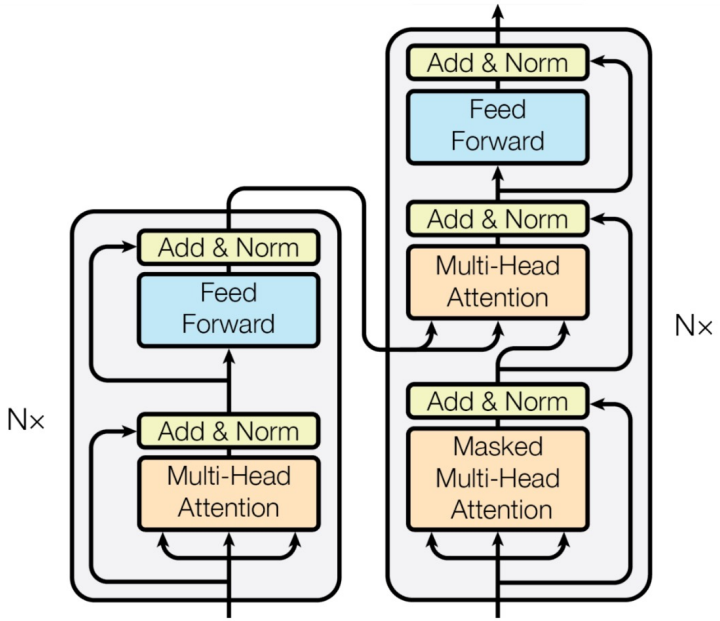
2. Self-Attention in English Sequence (entire sequence).

3. Cross-Attention Comparison.

a. **Query:** The attended representation of the Spanish sequence (generated so far) acts as the query. "Given what I've generated in Spanish already, what information from the English sentence is most relevant to predicting the next word?"

b. **Key and Value:** The attended representation of the entire English sentence acts as both the "key" and "value." The "key" is used to compute attention weights (how much each English word should be attended to), and the "value" is the information that gets weighted and incorporated into the Spanish sequence.

4. Apply the Attention Weights to English Representation: The attention weights from comparing the query (Spanish) and key (English) are applied to the value (English). This results in a context vector representing the most relevant information from the English sentence for predicting the next Spanish word.



Multi-Head Attention for Other Tasks

BERT Embeddings

- **B**idirectional **E**ncoder **R**epresentations from **T**ransformers.

- Employs a similar (self-supervised learning) technique as Word2Vec. Mask 15% of words at random in any given sequence and then try to predict those using the words that are not masked.

- Word2vec is nice but yields static (generic) embeddings for a given word. BERT learns context-dependent representations. This can dramatically improve the predictive performance of, say, a text classifier.

Common Use Case: Sentence Embeddings

Mean Pooling of Contextual Word Embeddings

- For a given sentence, you can extract the contextual word embeddings from *after* the MultiHeadAttention layers and then average them.

Pre-Trained BERT Classifier

- Or load a pre-trained BERT model's transformer and embedding layers (backbone / base layers) to pre-process your text and obtain contextual embeddings for a given prediction task.

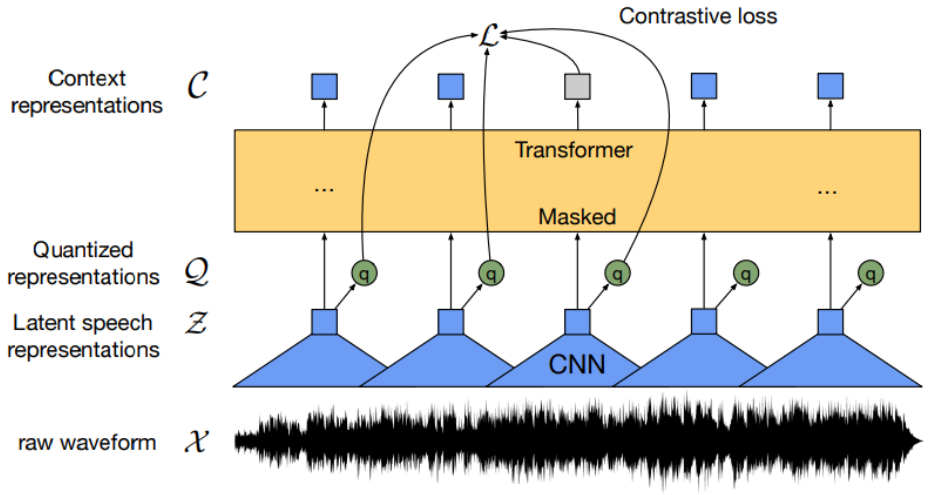
- Or, load the embedding and attention layers, freeze them, and add your Dense layers on top like we did with pre-trained vision models.

```
import keras_hub

tokenizer = keras_hub.models.Tokenizer.from_preset("roberta_base_en")
backbone = keras_hub.models.Backbone.from_preset("roberta_base_en")
```

Multi-Head Attention for Other Tasks

Wav2Vec for Audio Understanding

- wav2vec also implements a self-supervised learning architecture with random masking of sequence input.
- The model first obtains 1D convolutional feature maps per chunk/segment of raw audio signal (spoken words), yielding sequences of feature maps.
- A random subsample of feature maps is masked, and the model tries to predict them from the feature maps that remain visible. This model then yields audio embeddings highly representative of spoken semantic content in audio.

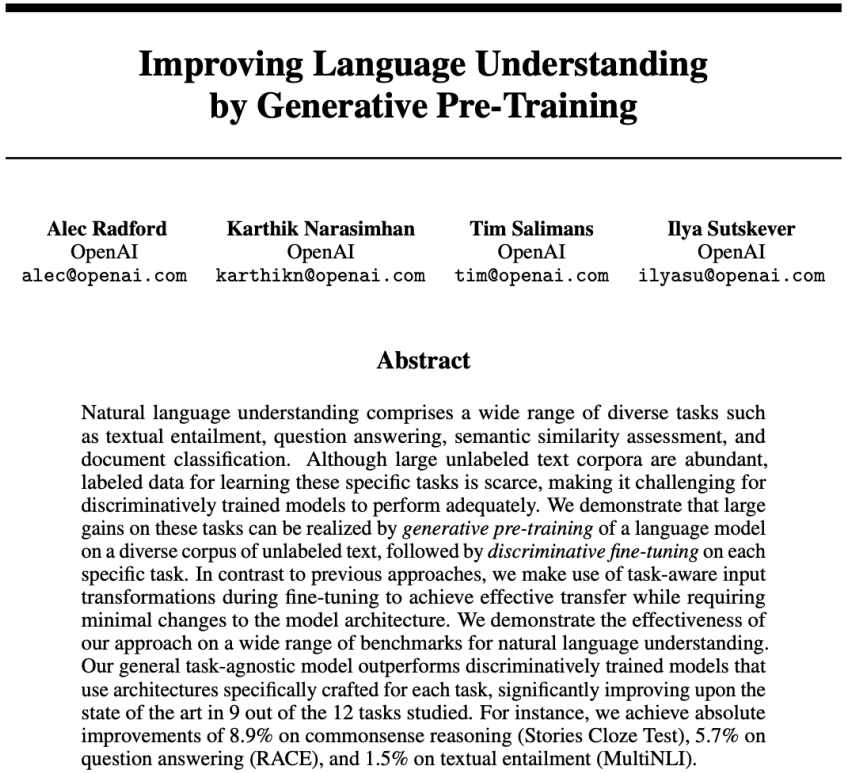


Generative Pre-trained Transformers

Generative Pre-trained Transformers (GPTs)

In 2018, not long after “Attention Is All You Need,” OpenAI put out a paper about the efficacy of scaling data and compute with a mixture of basic ingredients: i) vast amounts of highly varied text, ii) a generative language architecture, and iii) transformers (oh, and RLHF).

- GPT-1 (2017) had 117 million parameters trained on 10 million tokens.
- GPT-2 (2018) had 1.5 billion parameters and was trained on 10 billion tokens.
- GPT-3 (2020) had 175 billion parameters and was trained on 300 billion tokens.



Improving Language Understanding by Generative Pre-Training

Alec Radford OpenAI alec@openai.com
Karthik Narasimhan OpenAI karthikn@openai.com
Tim Salimans OpenAI tim@openai.com
Ilya Sutskever OpenAI ilyasu@openai.com

Abstract

Natural language understanding comprises a wide range of diverse tasks such as textual entailment, question answering, semantic similarity assessment, and document classification. Although large unlabeled text corpora are abundant, labeled data for learning these specific tasks is scarce, making it challenging for discriminatively trained models to perform adequately. We demonstrate that large gains on these tasks can be realized by *generative pre-training* of a language model on a diverse corpus of unlabeled text, followed by *discriminative fine-tuning* on each specific task. In contrast to previous approaches, we make use of task-aware input transformations during fine-tuning to achieve effective transfer while requiring minimal changes to the model architecture. We demonstrate the effectiveness of our approach on a wide range of benchmarks for natural language understanding. Our general task-agnostic model outperforms discriminatively trained models that use architectures specifically crafted for each task, significantly improving upon the state of the art in 9 out of the 12 tasks studied. For instance, we achieve absolute improvements of 8.9% on commonsense reasoning (Stories Cloze Test), 5.7% on question answering (RACE), and 1.5% on textual entailment (MultiNLI).

GPTs (and LLMs Generally) Have Many Problems

Various Challenges with LLMs (Among Many Others)

- Hallucination: They produce plausible text as output, not necessarily truthful or correct text. Therefore, they are highly dependent on the quality of training data.

- Prompt Brittleness: Rewording a prompt even slightly can lead to considerable changes in model response. For example, if you ask the same question of an LLM in different languages, you could get very different answers (conceptually and semantically)!

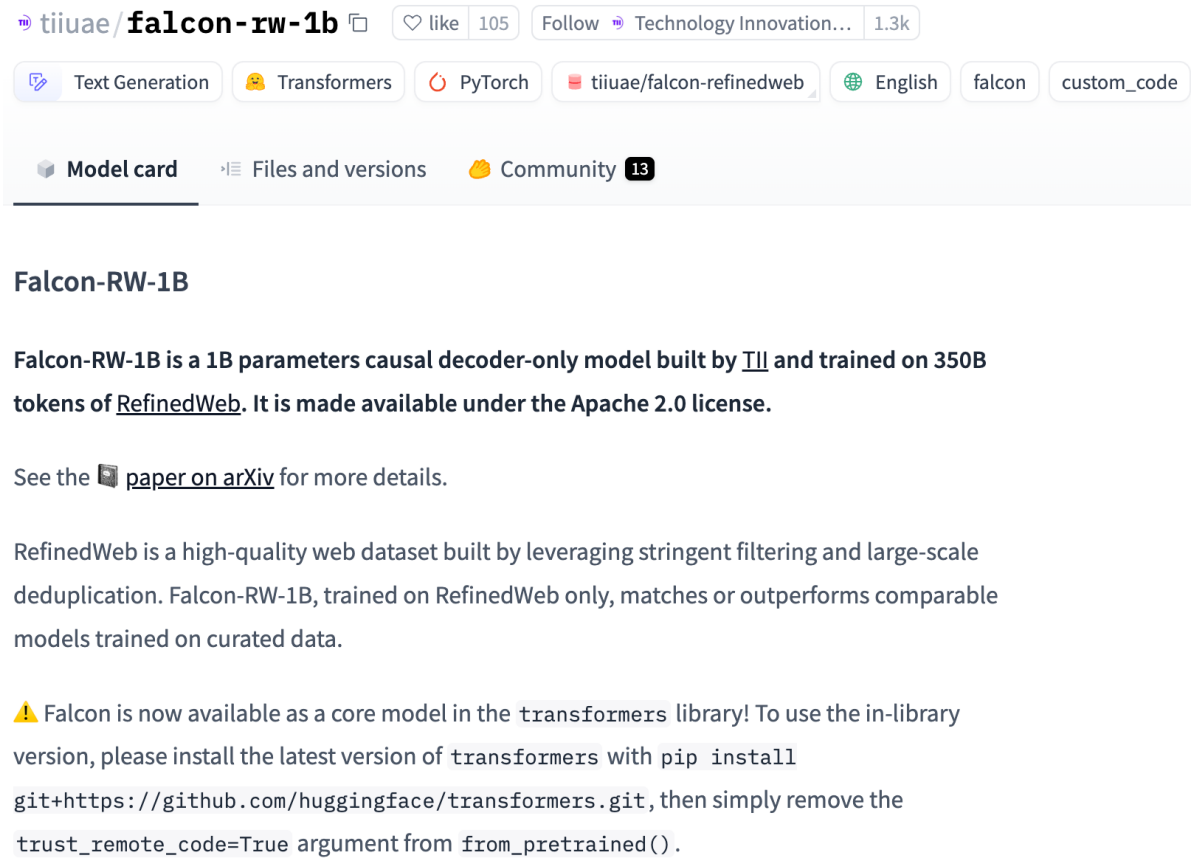
- Memorization: Assessing the performance of LLMs on various task benchmarks is quite challenging because it is often tough to assess whether the model has seen the problem previously and is merely looking up the answer in training data. This affects whether performance reflects how a model might perform on an entirely new task.

Lots of Pre-Trained LLMs Around

Loading Pre-trained Models from Keras Hub

Check out what is available on the Keras hub. Many open-source and free-to-use models are there, though many are too large for a Colab notebook.

Kaggle hosts some of the models. For a given model, you may need to visit Kaggle and create an API key, complete a request form, or agree to terms and conditions.

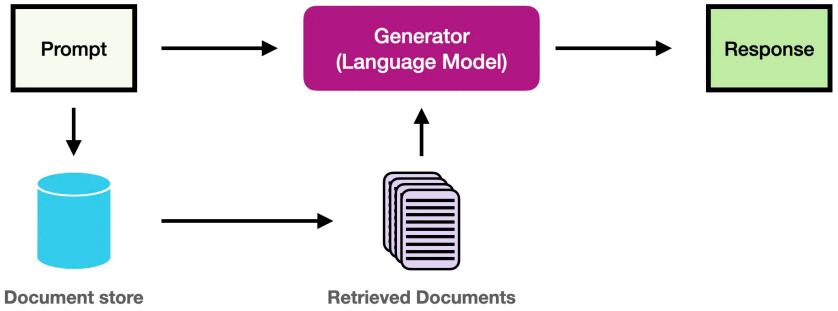


Some Innovations

Memory-Augmented Reasoning

To improve the accuracy and fidelity of model responses, we have them refer to local facts and knowledge rather than just answer based on pre-trained parameters.

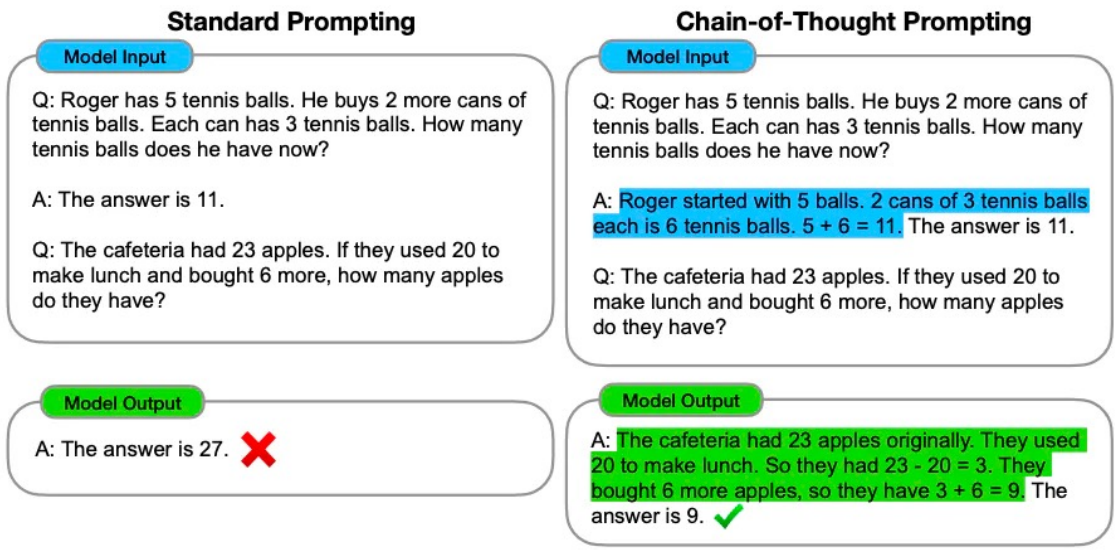
- RAG (retrieval augmented generation) implements this idea—we obtain embedded representations of all relevant documents ahead of time and store them. When it comes time to prompt the LLM, we first convert the prompt to an embedding. We then perform a document lookup by comparing the prompt embedding to document embeddings. We then retrieve the top N documents and include them alongside the original prompt.



Some Innovations

Guiding Examples (Answers / Reasoning)

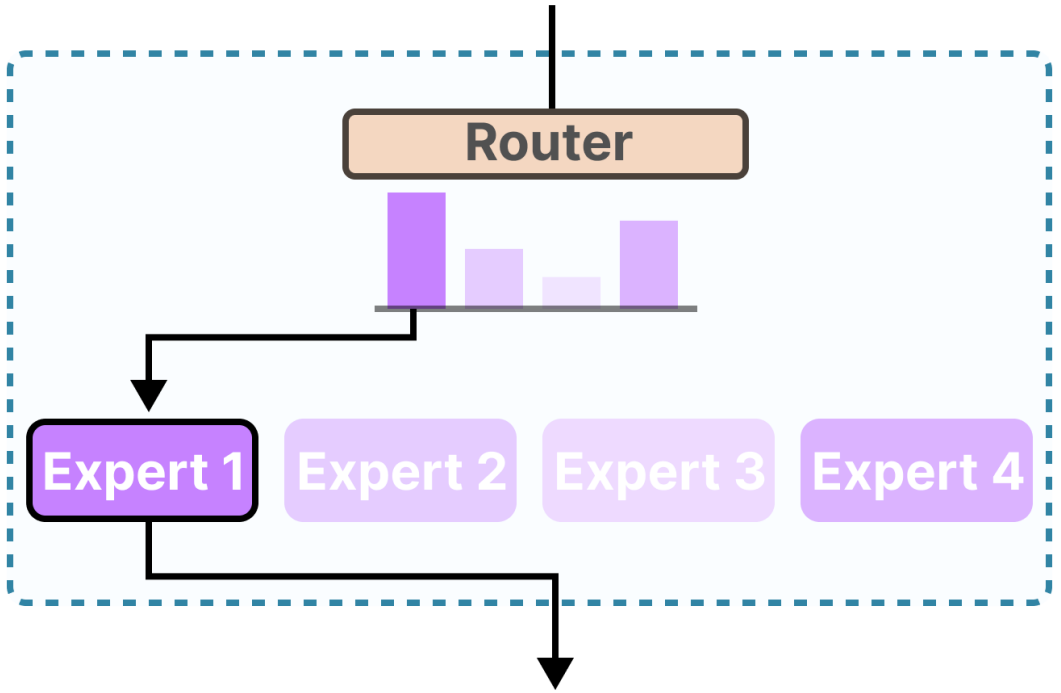
- Zero-Shot Prompt: Ask the question and instruct the model to think through its answer step by step.
- One/Few-Shot Prompt: Provide the model with examples of questions and correct example answers.
- One/Few-Shot Chain of Thought: Provide examples of question → reasoning process → answer.



Recent Architectural Innovations

Mixture of Experts

- We have a multiplex design, where the input determines which network component is activated to process and generate a prediction.
- Originally, this was ‘hard routing’ (softmax argmax), but more recently, it has become soft routing (just softmax), and we take weighted averages of expert output.
- This adds a lot of overhead to the model, so inference is expensive/slow. Deepseek innovated on this and ‘learned’ the routing rule (or its approximation) after training and made a new network that fixes the routing rule.



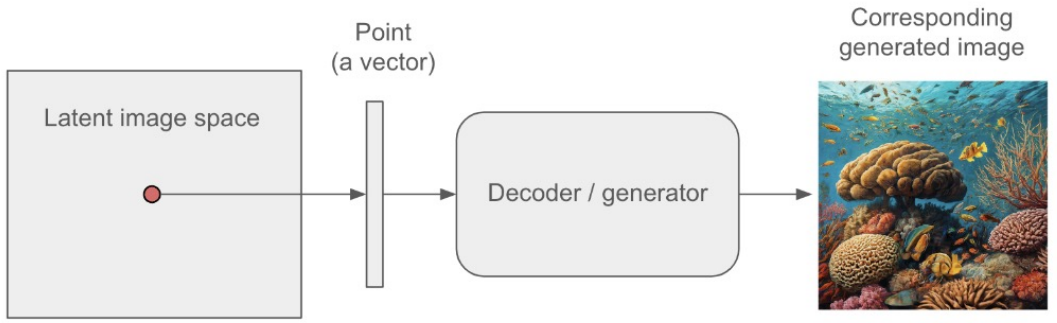
Generative Models

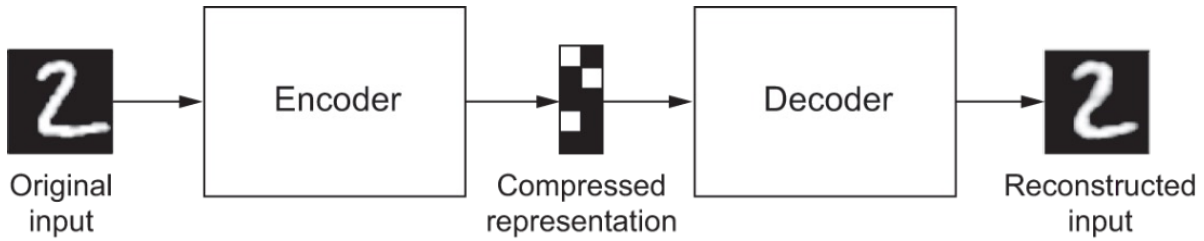
Generative Models Have Taken Off

- **Text-to-Image:** Midjourney, Stable Diffusion, DALL-E, etc.
- **Audio + Photo to Video:** D-ID
- **Text to Voice:** ElevenLabs

Generating Images with VAEs

Variational Auto-encoders employ an auto-encoder architecture (similar to our translation case, or the image-sharpening example) to compress and recover the input image. The trick is in the architecture we use to capture the compression.



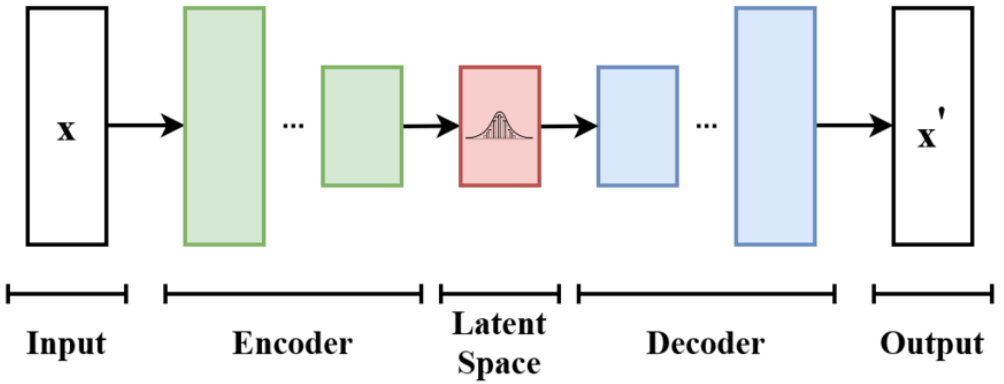


Variational Auto-encoder (VAE)

Mapping Images to a Vector of Latent Distributions

- We learn to map each image in our training data to a distribution over latent variables — typically a Gaussian with a learned mean and variance.

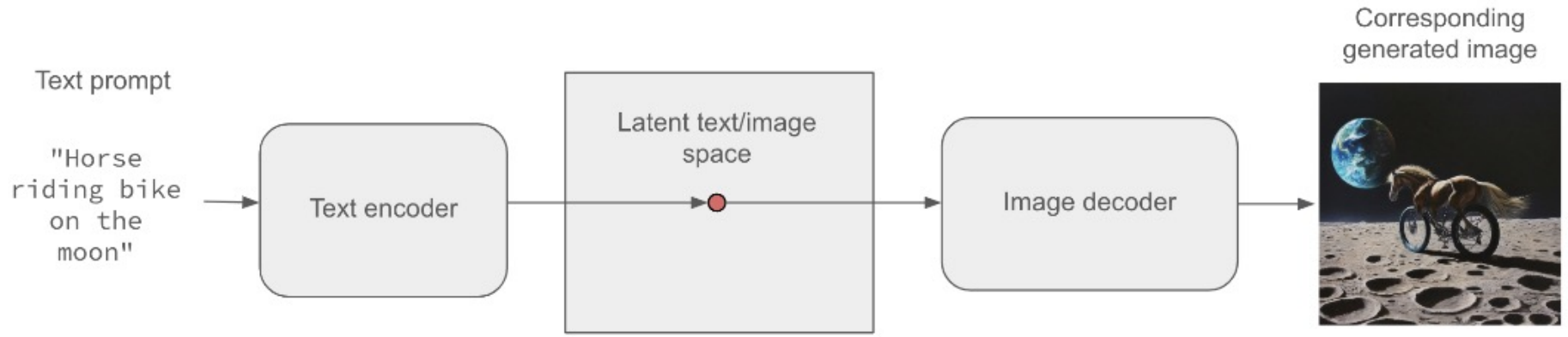
- We do this in a self-supervised way, using an auto-encoder architecture. The model architecture compresses the input image down into a vector that represents mean and variance values in several embedding dimensions and then converts that latent representation back to an image.



Conditional VAE (Guiding with Text)

We train the model by passing it an image along with textual captions. The image is projected into the latent space as a set of means and variances in a way that considers the text (conditional on the text). The decoder then samples from the normal distributions specified at that point in the latent space and attempts to reconstruct the input image, again taking the caption as input.

At inference, we drop the encoder and randomly sample from the latent distributional space (synthesize a random vector of values). We use that, along with a textual prompt, to synthesize a new image.



Encoder Portion

The encoder portion takes in the image and compresses it with convolutions (like our image sharpening task). We then have two ‘stand-in’ hidden layers intended to capture a multivariate normal distribution (means and variances associated with each embedding dimension).

```
import keras
from keras import layers

latent_dim = 2

encoder_inputs = keras.Input(shape=(28, 28, 1))
x = layers.Conv2D(32, 3, activation="relu", strides=2, padding="same")(encoder_inputs)
x = layers.Conv2D(64, 3, activation="relu", strides=2, padding="same")(x)
x = layers.Flatten()(x)
x = layers.Dense(16, activation="relu")(x)
z_mean = layers.Dense(latent_dim, name="z_mean")(x)
z_log_var = layers.Dense(latent_dim, name="z_log_var")(x)
encoder = keras.Model(encoder_inputs, [z_mean, z_log_var], name="encoder")
```

Decoder Portion

The decoder portion takes in a point in the latent distribution and up-samples that into an image.

```
latent_inputs = keras.Input(shape=(latent_dim,))
x = layers.Dense(7 * 7 * 64, activation="relu")(latent_inputs)
x = layers.Reshape((7, 7, 64))(x)
x = layers.Conv2DTranspose(64, 3, activation="relu", strides=2, padding="same")(x)
x = layers.Conv2DTranspose(32, 3, activation="relu", strides=2, padding="same")(x)
decoder_outputs = layers.Conv2D(1, 3, activation="sigmoid", padding="same")(x)
decoder = keras.Model(latent_inputs, decoder_outputs, name="decoder")
```

Generative Adversarial Networks (GANs)

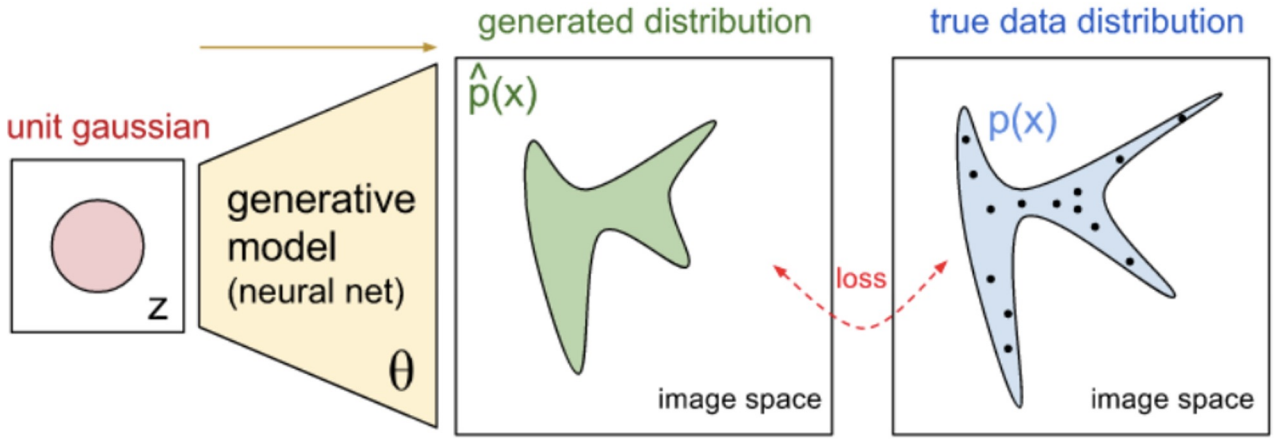
GANs are a Powerful Flexible Tool for Generative Modeling

- What is a GAN? How do GANs work?
- What problems can we address with GANs?
- How do we implement a GAN?

The Goal

Synthesize Samples That Conform to the Real Data's Distribution

- We train a network to produce synthetic samples that are indistinguishable from true samples.
- More concrete, this means we train a network to learn the probability distribution of real data.
- For example, learn the joint probability distribution of pixel values in a set of images.
- So, the resulting network could take a random vector of noise as input, and map it to a synthetic output that looks very similar to real data.



How Might We Do This?

A Neural Net That Produces Image Output

- Take in a random vector as input, and have it produce image predictions.
- Next, compare those predictions with authentic images. But how?
- We don't want it to try to predict a specific image, because then it won't be able to produce new synthetic examples.
- The problem? **The loss function is highly complicated.**
- A solution…

Generative Adversarial Nets

Ian J. Goodfellow, Jean Pouget-Abadie[*], Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair[†], Aaron Courville, Yoshua Bengio[‡]
Département d'informatique et de recherche opérationnelle
Université de Montréal
Montréal, QC H3C 3J7

Adversarial Architecture

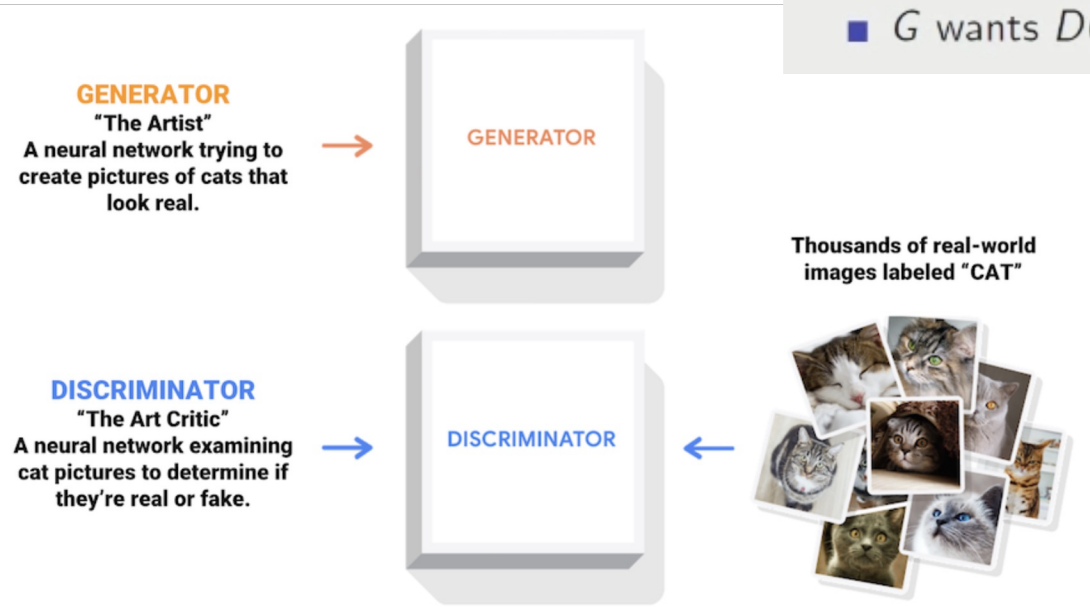
Discriminator

- Serves as an adaptive loss function for the generator.
- It's a throw-away network that is just there to help train the generator.

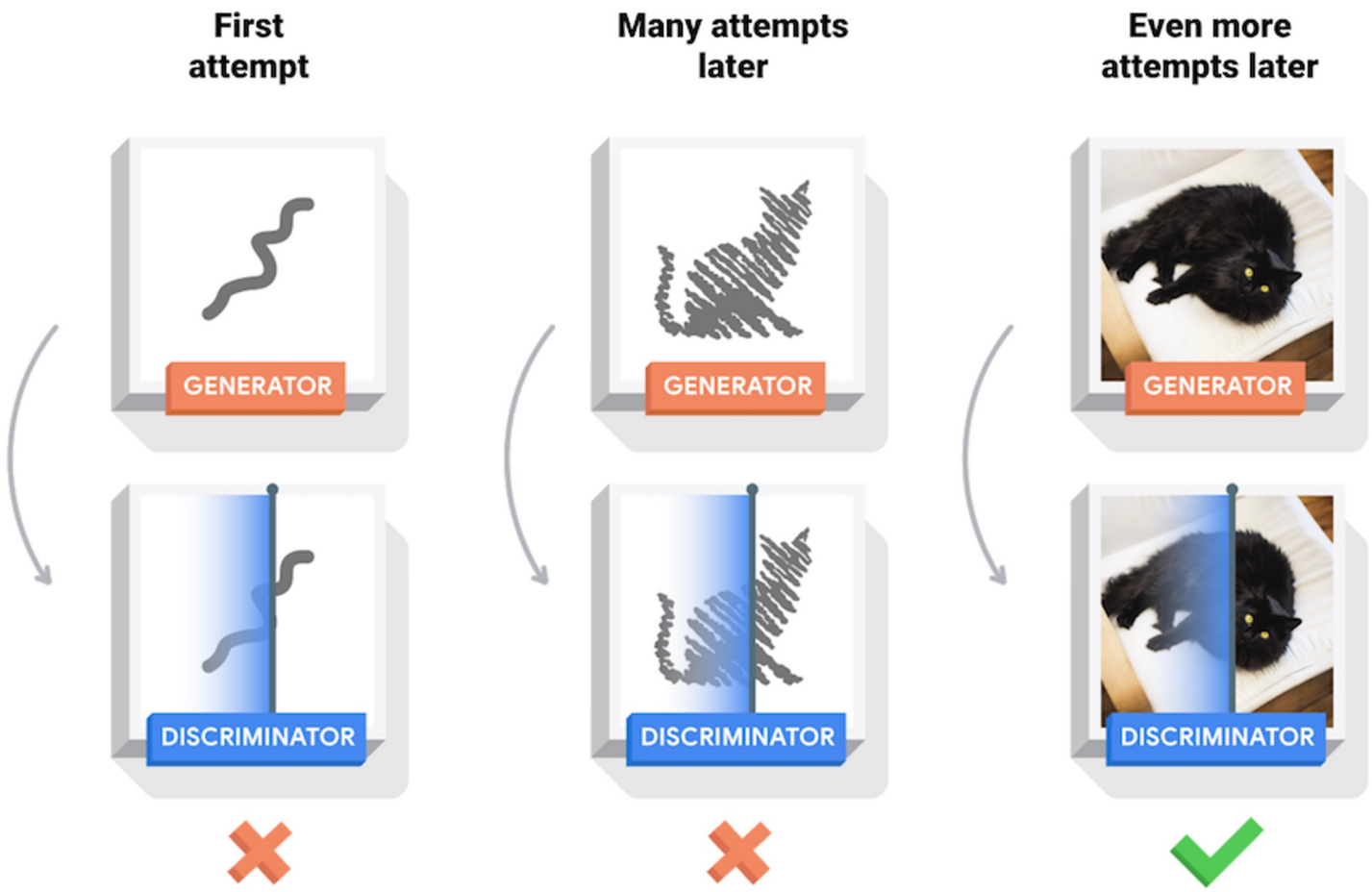
A GAN is defined by the following min-max game

$$\min_G \max_D V(D, G) = \mathbb{E}_X \log D(X) + \mathbb{E}_Z \log(1 - D(G(Z)))$$

- D wants $D(X) = 1$ and $D(G(Z)) = 0$
- G wants $D(G(Z)) = 1$



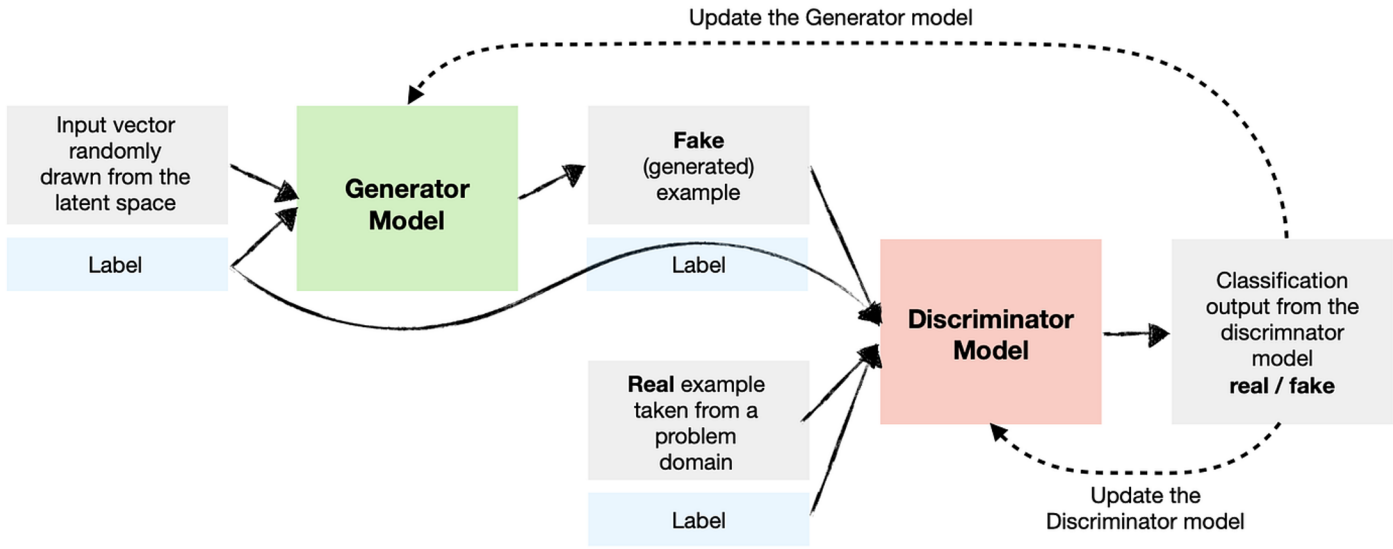
Adversarial Architecture



Conditional GANs

GAN /w Labels as Input Too

- We don't want our GAN to just learn P(X); it needs to learn P(X|Y).
- This is a simple modification; we give two inputs to the generator, a label and a noise vector. So, it tries to produces images that match real images given a particular label, rather than images in general.
- End result is a generator that you can pass a noise vector and a label, and it spits out an image of that label.



Questions?