Module 5

Transformers

Text Book 1: Chapter 9

Transformers - Chapter Goals

In this chapter we will:

- Learn about the **origins of GPT**, a powerful **decoder Transformer** model for text generation.
- Learn how an **attention mechanism** gives **more importance to some words** in a sentence than others.
- Understand how **queries, keys and values** are created and used inside attention.
- See the importance of **causal masking** for text generation tasks.
- Understand how attention heads are grouped into a **multihead attention layer**.
- See how a **Transformer block** combines attention, **layer normalization** and **skip connections**.
- Create **positional encodings** that capture the position of each token.
- Build a **GPT model in Keras** to generate **wine reviews** and inspect its **attention scores**.
- Learn the **types of Transformers** (encoder, decoder, encoder-decoder), **T5**, and how **ChatGPT** is trained.

Introduction

Before Transformers - Recurrent Neural Networks (RNNs)

- Text was generated using **RNNs** such as **LSTMs** and **GRUs** (Chapter 5).
- These are **autoregressive models**: they process a sequence **one token at a time**.
- They keep updating a **hidden vector** that stores the current latent representation of the input.
- To predict the next word, a **dense layer + softmax** is applied over the hidden vector.
- This was the most advanced way to generate text **until 2017**.

The Turning Point - "Attention Is All You Need" (Google Brain, 2017)

- Popularized the concept of **attention** - the mechanism behind most state-of-the-art text generators.
- Introduced **Transformers**: powerful networks for sequences that need **no recurrent or convolutional layers** - only **attention**.
- RNNs are **hard to parallelize** (one token at a time); Transformers are **highly parallelizable** and can be trained on **massive datasets**.

Introduction

RNN (LSTM/GRU) Versus Transformer

Feature	RNN / LSTM	Transformer
Processing	One token at a time (sequential)	All tokens together (parallel)
Memory of past words	Stored in a single hidden state	Looks directly at any previous word using attention
Long sentences	Old information gets weak / lost	Can attend to words far back without loss of signal
Training speed	Slow - difficult to parallelize	Fast - trained on huge datasets using GPUs
Key mechanism	Recurrence	Attention (queries, keys, values)

Example: To guess the last word of “The pink elephant tried to get into the car but it was too ___”, an RNN must remember “elephant” inside its hidden state, whereas a Transformer can **directly look back** at the word “elephant”.

GPT - Generative Pre-trained Transformer

- **GPT** was introduced by **OpenAI in June 2018** in the paper “Improving Language Understanding by Generative Pre-Training” - about one year after the original Transformer paper.
- GPT is an **autoregressive** Transformer model that powers OpenAI’s **GPT-4**.

1. Pre-training

- The model is trained on a large text corpus called **BookCorpus** (**4.5 GB** of text from **7,000 unpublished books**).
- Task: **predict the next word** in a sequence given the previous words - this is called **language modeling**.
- This teaches the model the **structure and patterns of natural language**.

2. Fine-tuning

- The pre-trained model is then trained on a **smaller, task-specific dataset**.
- Parameters are adjusted for tasks such as **classification, similarity scoring** or **question answering**.
- Later versions: **GPT-2, GPT-3, GPT-3.5 and GPT-4** - larger models trained on larger datasets, giving more complex and coherent text.

GPT - Generative Pre-trained Transformer

Easy Understanding: Pre-training and Fine-tuning

- **Pre-training** is like a student who **reads thousands of books** and learns how sentences are formed.
- **Fine-tuning** is like the same student taking a **short coaching class** for one particular exam.

Example

Input (pre-training): “The cat sat on the ___” → Model learns to predict: “mat”

Fine-tuning for classification: “The movie was wonderful” → **Positive**

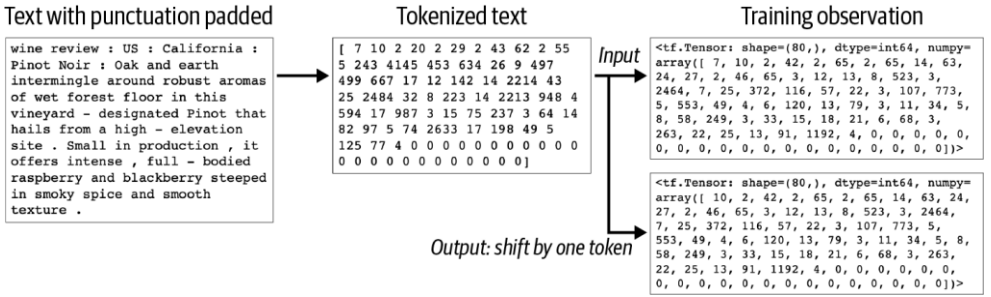
What will we build?

- Our own **variation of the original GPT model** - trained on **less data**, but using the **same components and principles**.
- Code: notebooks/09_transformer/01_gpt/gpt.ipynb (adapted from the Keras GPT tutorial by Apoorv Nandan).

The Wine Reviews Dataset

- Dataset from **Kaggle**: over **130,000 reviews of wines** with metadata such as **description and price**.
- Download command (Example 9-1): *bash scripts/download_kaggle_data.sh zynicide wine-reviews*
- Data preparation is **identical to the LSTM data preparation of Chapter 5**:

1. **Load** the data and create a list of text descriptions of each wine.
2. **Pad punctuation** with spaces, so each punctuation mark is a separate word.
3. Pass strings through a **TextVectorization** layer that **tokenizes** and **pads/clips** each string to a fixed length.
4. Create a training set: **inputs** = tokenized strings; **outputs** = the **same strings shifted by one token**.


Text with punctuation padded
wine review : US : California : Pinot Noir : Oak and earth intermingle around robust aromas of wet forest floor in this vineyard - designated Pinot that hails from a high - elevation site . Small in production , it offers intense , full - bodied raspberry and blackberry steeped in smoky spice and smooth texture .
Tokenized text
[7 10 2 20 2 29 2 43 62 2 55 5 243 4145 453 634 26 9 497 499 667 17 12 142 14 2214 43 25 2484 32 8 223 14 2213 948 4 594 17 987 3 15 75 237 3 64 14 82 97 5 74 2633 17 198 49 5 125 77 4 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
Input
Output: shift by one token
Training observation
<tf.Tensor: shape=(80,), dtype=int64, numpy= array([7, 10, 2, 42, 2, 65, 2, 65, 14, 63, 24, 27, 2, 46, 65, 3, 12, 13, 8, 523, 3, 2464, 7, 25, 372, 116, 57, 22, 3, 107, 773, 5, 553, 49, 4, 6, 120, 13, 79, 3, 11, 34, 5, 8, 58, 249, 3, 33, 15, 18, 21, 6, 68, 3, 263, 22, 25, 13, 91, 1192, 4, 0, 0, 0

The Wine Reviews Dataset

Why “shift by one token”?

- GPT learns to **predict the next word**. So, for every word in the input, the target is the **word that follows it**.

Example

Input sentence: wine review : germany : riesling ...

Input (x): [wine, review, :, germany, :]

Target (y): [review, :, germany, :, riesling]

- When the model sees “wine” it should predict “review”; when it sees “wine review :” it should predict “germany”, and so on.
- In this way, **one sentence gives many training examples** at the same time.

Attention

- Attention is what makes the Transformer **unique** and different from recurrent models.
- When we write, the **next word** we choose depends on **some of the words already written**.

Example: “The pink elephant tried to get into the car but it was too ___”

- The next word should be something like **big**. How do we know this?
  - **elephant** (not a sloth) → we prefer **big** rather than **slow**.
  - If it were a **swimming pool** instead of a **car** → we might choose **scared**.
  - **getting into the car** → size is the problem. If the elephant tried to **squash** the car, we might choose **fast** (with **it** now meaning the car).
- Other words are **not important**: the fact that the elephant is **pink** does not affect the choice; minor words (**the, but, it**) only give grammatical form.
- We **pay attention to certain words and ignore others**. Our model should do the same!

Attention

Attention Mechanism (Attention Head)

- An **attention mechanism** (also called an **attention head**) decides **where in the input** it wants to **pull information from**.
- It extracts useful information **without being clouded by irrelevant details**.
- It is **highly adaptable** - it decides where to look **at inference time**, depending on the context.

Recurrent Layer Versus Attention Head

- A **recurrent layer** builds a **generic hidden state** that summarizes the whole input at each timestep.
- Weakness: many words stored in the hidden vector are **not relevant** to the current task (e.g., predicting the next word).
- **Attention heads do not suffer from this problem** - they **pick and choose** how to combine information from nearby words, depending on the context.

Queries, Keys, and Values

How does an attention head decide where to look?

- We want to predict the word that follows **too** (pink elephant example).
- The other preceding words **“chime in” with their opinions**, but each opinion is **weighted** by how **confident** that word is about predicting what follows **too**.
  - **elephant** confidently says: the next word is likely related to **size or loudness**.
  - **was** has **not much to offer** to narrow down the possibilities.

Attention = Information Retrieval System

- A **query** (“What word follows too?”) is made into a **key/value store** (the other words in the sentence).
- The output is a **sum of the values**, **weighted by the resonance** (similarity) between the **query and each key**.

Easy analogy: Searching on YouTube - your search text is the **query**, video titles are the **keys**, and the videos are the **values**. Videos whose titles best match your query get the most importance.

Queries, Keys, and Values

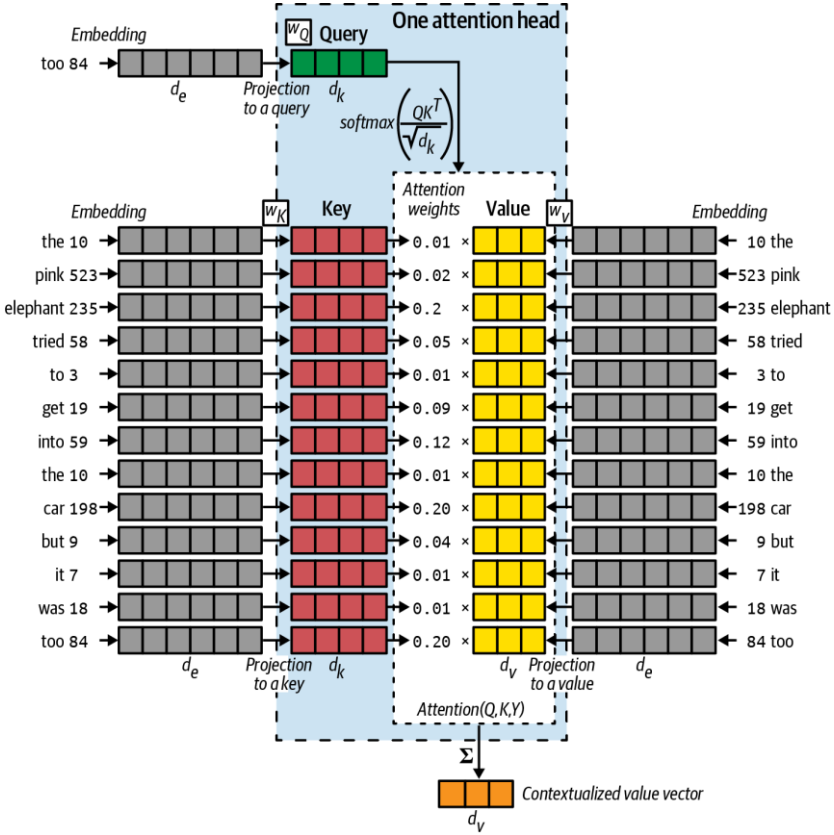


Figure 9-2. The mechanics of an attention head

Query (Q)

- Represents the **current task** - "What word follows too?"
- Derived from the embedding of the word **too** by passing it through a weights matrix $\mathbf{W_Q}$, changing dimension from $\mathbf{d_e}$ to $\mathbf{d_k}$.

Keys (K)

- Representations of **each word** in the sentence - a description of the kinds of prediction tasks each word can help with.
- Derived using weights matrix $\mathbf{W_K}$ ($d_e \rightarrow d_k$). **Keys and query have the same length ($\mathbf{d_k}$)**.

Values (V)

- The **unweighted contributions** of each word.
- Derived using weights matrix $\mathbf{W_V}$ ($d_e \rightarrow d_v$). Values need not have the same length as keys (but often do).

Queries, Keys, and Values

Step-by-step working of an attention head

1. **Compare** each key with the query using a **dot product** → **QK^T**. (This is why keys and query must be the same length.)
2. A **higher dot product** means the key **resonates more** with the query → that word contributes **more** to the output.
3. **Scale** the result by $\sqrt{\mathbf{d_k}}$ to keep the variance of the vector sum stable (approximately 1).
4. Apply **softmax** so that all contributions **sum to 1** → this is the vector of **attention weights**.
5. **Multiply** the value vectors by the attention weights and **sum** them.

Equation 9-1. Attention equation

$$Attention(Q, K, V) = softmax(QK^T / \sqrt{d_k}) V$$

- The final output is a vector of length $\mathbf{d_v}$ called the **context vector** - a **blended opinion** of the words in the sentence about what word follows **too**.

Queries, Keys, and Values

Simple Numerical Illustration (for understanding)

Query for **“too”**: $q = [1, 0]$, $d_k = 2$

Keys: elephant = [2, 0], car = [1, 1], was = [0, 1]

Values: elephant = [1, 0], car = [0, 1], was = [0, 0]

Step 1: Dot products $q \cdot k = [2, 1, 0]$

Step 2: Scale by $\sqrt{2} \rightarrow [1.414, 0.707, 0]$

Step 3: Softmax → [**0.576**, 0.284, 0.140]

Step 4: Weighted sum of values

$= 0.576 \cdot [1,0] + 0.284 \cdot [0,1] + 0.140 \cdot [0,0]$

= **[0.576, 0.284]** (context vector)

Word	**Score**	**Attention weight**
elephant	1.414	0.576
car	0.707	0.284
was	0.000	0.140

- **elephant** receives the **highest attention** (57.6%), so the prediction for the word after “too” is guided mostly by “elephant” → **big**.
- **was** gets very little attention.

Multihead Attention

- There is no reason to stop at **one attention head**! A **MultiHeadAttention** layer **concatenates** the outputs of **several attention heads**.
- Each head learns a **distinct attention mechanism**, so the layer as a whole learns **more complex relationships**.
- The concatenated output is passed through a final weights matrix $\mathbf{W_O}$ to project it to the **desired output dimension** - the same as the query input dimension ($\mathbf{d_e}$) - so that layers can be **stacked** one after another.

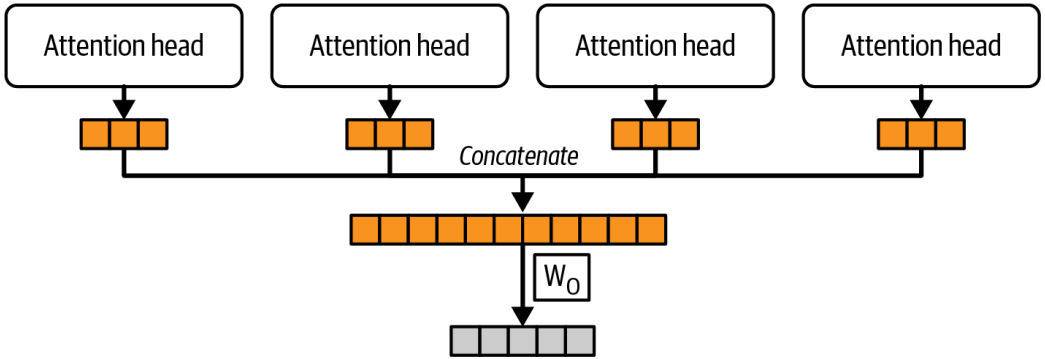


Figure 9-3. A multihead attention layer with four heads

Example: In a sentence, one head may focus on **"who"** (the subject - elephant), another on **"where"** (car), another on **grammar**. Together they give a richer understanding.

Multihead Attention

Example 9-2. Creating a MultiHeadAttention layer in Keras

```
layers.MultiHeadAttention(
    num_heads = 4,
    key_dim = 128,
    value_dim = 64,
    output_shape = 256
    )
```

- **num_heads = 4** - the layer has **four heads**.
- **key_dim = 128** - keys (and query) are vectors of length **128**.
- **value_dim = 64** - values (and output of each head) have length **64**.
- **output_shape = 256** - the output vector has length **256**.

Easy understanding: 4 heads × 64 values = 256 numbers are concatenated; W_O then projects them to the output size of 256.

Learnable parameters: For each head only three dense weights matrices (**W_Q, W_K, W_V**) and one output matrix (**W_O**). There are **no convolutions or recurrent mechanisms** in a multihead attention layer!

Causal Masking

- So far the query was a **single vector**.
- For **efficient training**, the attention layer should work on **every word at once**, predicting the **next word for each** - i.e., a **matrix of queries** processed in parallel.
- Extra step needed: apply a **mask** to the query/key dot product so that **information from future words does not leak**.
- This is called **causal masking**.
- **Without the mask**, GPT would **perfectly guess** the next word, because it would use the **key of that word itself** as a feature - i.e., it would be cheating!

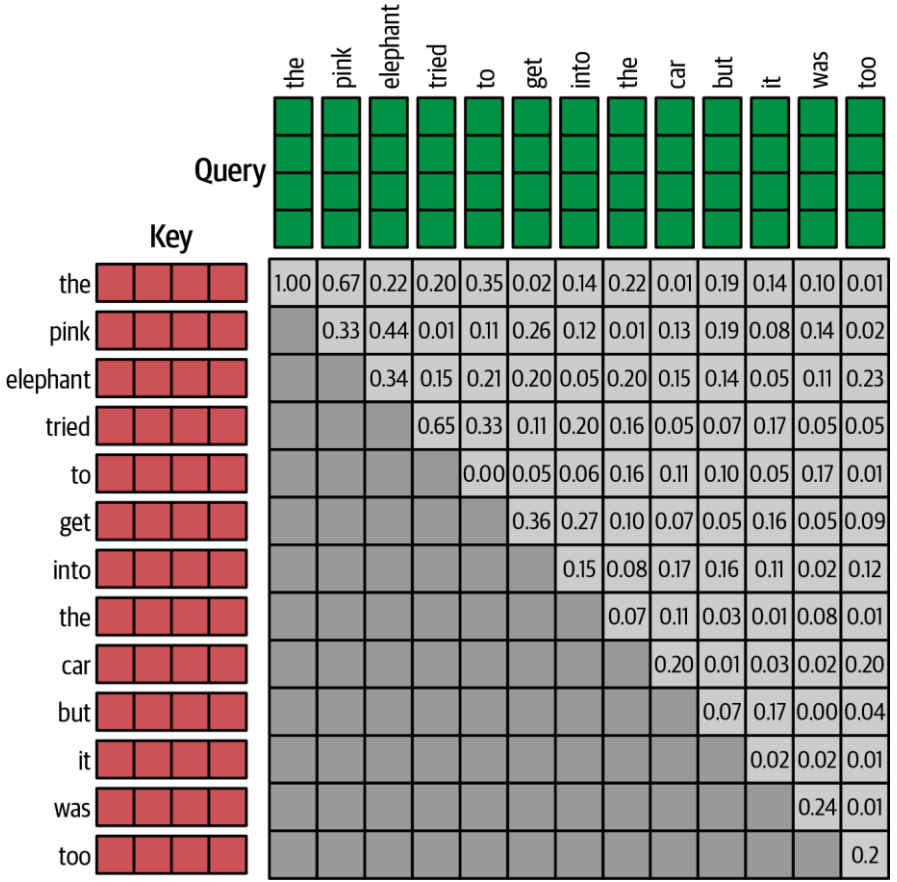


Figure 9-4. Attention scores for a batch of queries using a causal attention mask

Causal Masking

Example 9-3. The causal mask function

```
def causal_attention_mask(batch_size, n_dest,
                          n_src, dtype):
    i = tf.range(n_dest)[:, None]
    j = tf.range(n_src)
    m = i >= j - n_src + n_dest
    mask = tf.cast(m, dtype)
    mask = tf.reshape(mask, [1, n_dest, n_src])
    mult = tf.concat([tf.expand_dims(batch_size, -1),
                tf.constant([1, 1], dtype=tf.int32)], 0)
    return tf.tile(mask, mult)

np.transpose(causal_attention_mask(1, 10, 10,
                                   dtype = tf.int32)[0])
```

```
array([[1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
       [0, 1, 1, 1, 1, 1, 1, 1, 1, 1],
       [0, 0, 1, 1, 1, 1, 1, 1, 1, 1],
       [0, 0, 0, 1, 1, 1, 1, 1, 1, 1],
       [0, 0, 0, 0, 1, 1, 1, 1, 1, 1],
       [0, 0, 0, 0, 0, 1, 1, 1, 1, 1],
       [0, 0, 0, 0, 0, 0, 1, 1, 1, 1],
       [0, 0, 0, 0, 0, 0, 0, 1, 1, 1],
       [0, 0, 0, 0, 0, 0, 0, 0, 1, 1],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 1]], dtype=int32)
```

Figure 9-5. Causal mask as a numpy array - 1 means unmasked, 0 means masked

Easy example: For “the pink elephant”: when predicting after **the**, the model sees only **the**; after **pink**, it sees **the, pink**; after **elephant**, it sees **the, pink, elephant**. Future words are hidden (0).

Causal Masking

Causal mask for “the pink elephant” (rows = query word, columns = key word)

Query \ Key	the	pink	elephant
the	1	0	0
pink	1	1	0
elephant	1	1	1

- **Causal masking is only required in decoder Transformers** such as **GPT**, where the task is to **generate tokens one by one** from previous tokens.
- **Encoder Transformers** do **not** need causal masking because they are **not trained to predict the next token**.
  - Example: Google’s **BERT** predicts **masked words within a sentence**, so it can use context **before and after** the word.

The Transformer Block

- A **Transformer block** is a single component of a Transformer that applies **skip connections, feed-forward (dense) layers and normalization** around the **multihead attention layer**.

Components

1. **Multihead attention layer** (with causal mask).
2. **Add (skip connection)** + **Layer normalization**.
3. **Feed-forward layers** (dense).
4. **Add (skip connection)** + **Layer normalization**.

- Transformer blocks can be **stacked** to create **very deep networks** (GPT paper uses 12 blocks).

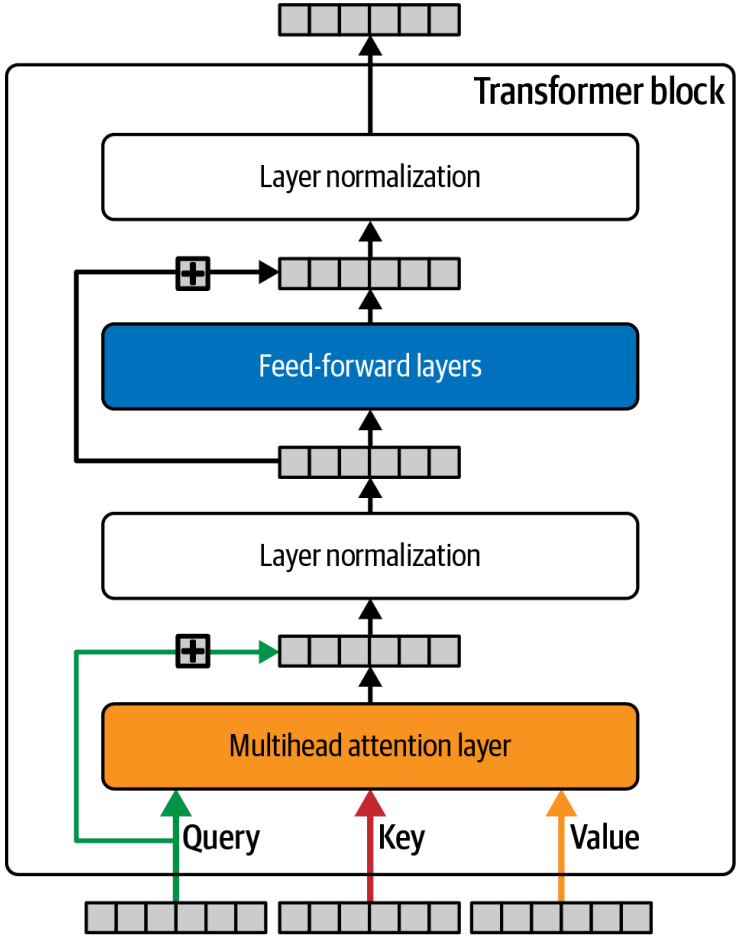


Figure 9-6. A Transformer block

The Transformer Block

1. Skip Connection

- The **query is passed around** the multihead attention layer and **added to its output**.
- Allows very **deep neural networks** that do not suffer much from the **vanishing gradient problem**.
- Provides a **gradient-free highway** that transfers information forward **uninterrupted**.

$$output = LayerNorm(x + Attention(x))$$

2. Layer Normalization

- Provides **stability** to the training process.
- **Batch normalization:** each channel is normalized to mean 0 and std 1; statistics computed **across the batch and spatial dimensions**.
- **Layer normalization:** normalizes **each position of each sequence** using statistics computed **across the channels** - the complete opposite of batch normalization.

$$LN(x) = \gamma \cdot (x - \mu) / \sqrt{(\sigma^2 + \varepsilon)} + \beta$$

The Transformer Block

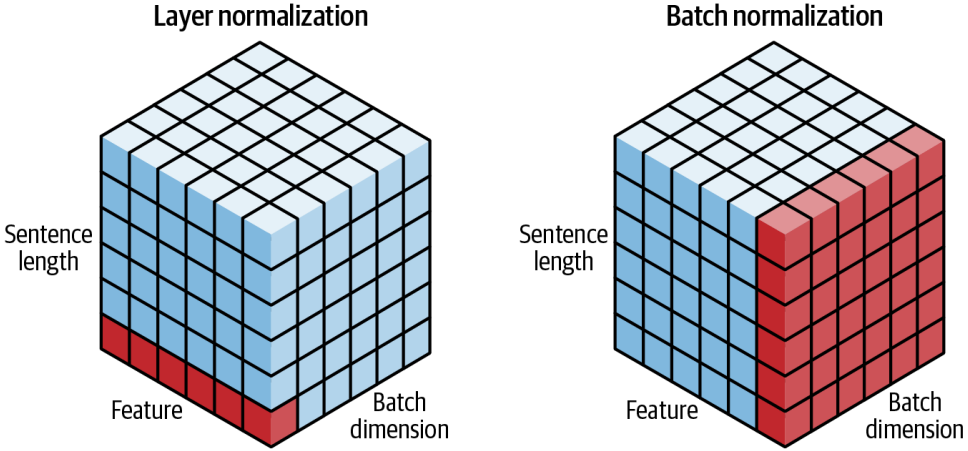


Figure 9-7. Layer normalization versus batch normalization - statistics are calculated across the blue cells (Sheng et al., 2020)

Layer Norm Versus Batch Norm

- Layer normalization was used in the **original GPT paper** and is common for **text tasks**, to avoid normalization dependencies across sequences in the batch.
- Recent work (**Shen et al.**) shows that with some tweaks, a form of **batch normalization** can be used in Transformers and can **outperform** layer normalization.

3. Feed-Forward Layers

- A set of **densely connected layers** is included in the Transformer block, to allow the component to **extract higher-level features** as we go deeper into the network.

The Transformer Block

Example 9-4. A TransformerBlock layer in Keras

```
class TransformerBlock(layers.Layer):
  def __init__(self, num_heads, key_dim, embed_dim,
               ff_dim, dropout_rate=0.1):
    super(TransformerBlock, self).__init__()
    self.attn = layers.MultiHeadAttention(
        num_heads, key_dim, output_shape = embed_dim)
    self.dropout_1 = layers.Dropout(dropout_rate)
    self.ln_1 = layers.LayerNormalization(epsilon=1e-6)
    self.ffn_1 = layers.Dense(ff_dim, activation="relu")
    self.ffn_2 = layers.Dense(embed_dim)
    self.dropout_2 = layers.Dropout(dropout_rate)
    self.ln_2 = layers.LayerNormalization(epsilon=1e-6)

  def call(self, inputs):
    input_shape = tf.shape(inputs)
    batch_size, seq_len = input_shape[0], input_shape[1]
    causal_mask = causal_attention_mask(
        batch_size, seq_len, seq_len, tf.bool)
    attention_output, attention_scores = self.attn(
        inputs, inputs, attention_mask=causal_mask,
        return_attention_scores=True)
    attention_output = self.dropout_1(attention_output)
    out1 = self.ln_1(inputs + attention_output)
    ffn_1 = self.ffn_1(out1)
    ffn_2 = self.ffn_2(ffn_1)
    ffn_output = self.dropout_2(ffn_2)
    return (self.ln_2(out1 + ffn_output), attention_scores)
```

- **__init__**: defines the sub-layers of the block.
- **causal_mask**: hides future keys from the query.
- **self.attn**: multihead attention with the causal mask.
- **ln_1**: first **add & normalize** layer.
- **ffn_1, ffn_2**: the **feed-forward** layers.
- **ln_2**: second **add & normalize** layer.
- Returns the output and the **attention scores**.

Positional Encoding

Why is it needed?

- In multihead attention, **nothing cares about the order of the keys**.
- The dot product of each key with the query is computed **in parallel**, not sequentially like an RNN.
- This is a **strength** (parallel, efficient) but also a **problem**. The model must predict **different outputs** for:
  - The **dog** looked at the **boy** and … (**barked?**)
  - The **boy** looked at the **dog** and … (**smiled?**)
- Both sentences have the **same words** - only their **order** is different!

Solution - Positional Encoding

- Encode each token using a **token embedding** and also encode its **position** using a **position embedding**.
- **Token embedding:** a standard **Embedding** layer converts each **token** into a learned vector.
- **Position embedding:** a standard **Embedding** layer converts each **integer position** (0, 1, 2, …) into a learned vector.
- GPT uses an Embedding layer for position; the **original Transformer** paper used **trigonometric functions** (sine/cosine).

Positional Encoding

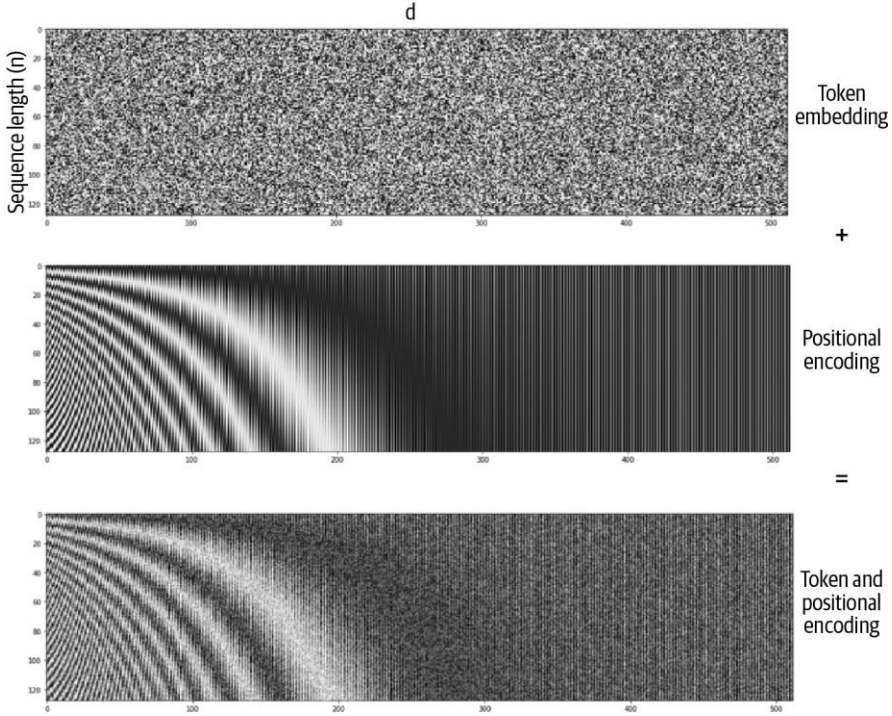


Figure 9-8. Token embeddings are added to positional embeddings to give the token position encoding

- The **token embedding is added to the positional embedding**.
- The **meaning and position** of each word are captured in a **single vector**.

$$x = TokenEmbedding(token) + PositionEmbedding(position)$$

```
class TokenAndPositionEmbedding(layers.Layer):
  def __init__(self, maxlen, vocab_size, embed_dim):
    super(TokenAndPositionEmbedding,self).__init__()
    self.token_emb = layers.Embedding(
        input_dim=vocab_size, output_dim=embed_dim)
    self.pos_emb = layers.Embedding(
        input_dim=maxlen, output_dim=embed_dim)

  def call(self, x):
    maxlen = tf.shape(x)[-1]
    positions = tf.range(start=0, limit=maxlen, delta=1)
    positions = self.pos_emb(positions)
    x = self.token_emb(x)
    return x + positions
```

Training GPT

Putting everything together

1. Input text → **Token and Position Embedding** layer.
2. → **Transformer block**.
3. → **Dense layer with softmax** over the number of words in the **vocabulary**.

- For simplicity, we use **just one Transformer block**, rather than the **12** used in the paper.

Example: Input: “the pink elephant tried to get into the car but it was too” → Output: probability for every word in the vocabulary; **“big”** gets the highest probability.

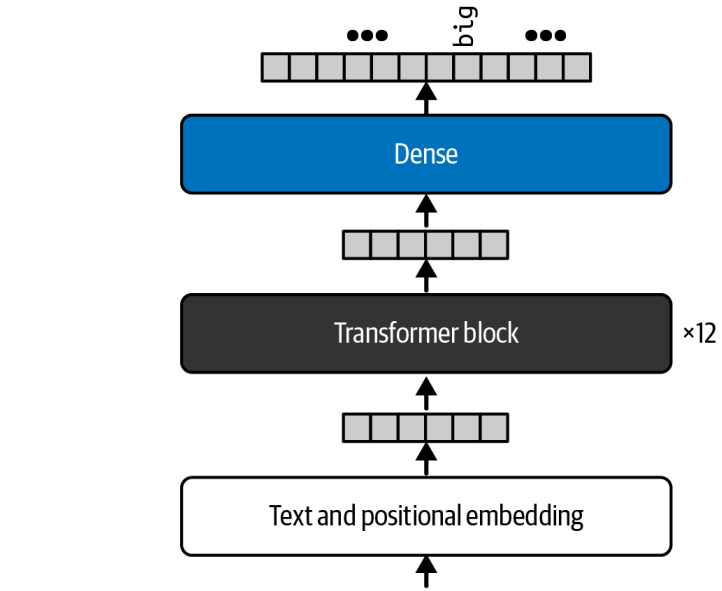


Figure 9-9. The simplified GPT model architecture

Training GPT

Example 9-6. A GPT model in Keras

```
MAX_LEN = 80
VOCAB_SIZE = 10000
EMBEDDING_DIM = 256
N_HEADS = 2
KEY_DIM = 256
FEED_FORWARD_DIM = 256

inputs = layers.Input(shape=(None,), dtype=tf.int32)
x = TokenAndPositionEmbedding(MAX_LEN, VOCAB_SIZE,
                              EMBEDDING_DIM)(inputs)
x, attention_scores = TransformerBlock(N_HEADS, KEY_DIM,
                    EMBEDDING_DIM, FEED_FORWARD_DIM)(x)
outputs = layers.Dense(VOCAB_SIZE, activation='softmax')(x)
gpt = models.Model(inputs=inputs,
                    outputs=[outputs, attention])
gpt.compile("adam",
   loss=[losses.SparseCategoricalCrossentropy(), None])
gpt.fit(train_ds, epochs=5)
```

- Input is **padded** (with zeros).
- Text encoded with **TokenAndPositionEmbedding**.
- Encoding passed through a **TransformerBlock**.
- **Dense + softmax** predicts a distribution over the **next word**.
- Model also returns the **attention scores** to inspect where it looks.
- Compiled with **SparseCategoricalCrossentropy** loss.

Settings: sequence length 80, vocabulary 10,000 words, embedding size 256, 2 attention heads.

Analysis of GPT

Generating Text

1. Feed the network an **existing sequence of words** and ask it to **predict the following word**.
2. **Append** this word to the existing sequence and **repeat**.

- The network outputs **probabilities** for each word, so we can **sample** - making generation **stochastic** (not deterministic).
- The **temperature** parameter controls how deterministic sampling is (same TextGenerator class as Chapter 5).

```
temperature = 1.0
Generated text:
wine review : us : washington : chenin blanc : a light , medium - bodied wine , this light - bodied expressi
on is not a lot of enjoyment . it ' s simple with butter and vanilla flavors that frame mixed with expressiv
e fruit . it ' s juicy and tangy with a lemon lingers on the finish .

temperature = 0.5
Generated text:
wine review : italy : piedmont : nebbiolo : this opens with aromas of french oak , menthol and a whiff of to
ast . the straightforward palate offers red cherry , black raspberry jam and a hint of star anise alongside
firm but rather fleeting tannins , drink through 2016 .
```

Figure 9-10. Generated outputs at temperature = 1.0 and temperature = 0.5

- Both outputs look like a **real wine review**: they open with the **region and type of wine**, and the wine type **stays consistent**.
- **Temperature 1.0** → more **adventurous**, more **variety**, less accurate.
- **Temperature 0.5** → more **accurate**, less variety.

Analysis of GPT

Viewing the Attention Scores

- The TransformerBlock outputs the **attention weights** - a softmax distribution over the **preceding words**.
- Darker blue = **more attention** on that word.

Example 1: "wine review : **germany** :" → model attends to **germany** to choose a German region: **pfalz 51.53%, mosel 41.21%**, rheingau 4.27%.

Example 2: "… **riesling** is a ripe, full-bodied" → it refers back to the grape: **riesling 46.56%** - no matter how far back it is (up to 80 words).

Example 3: "… with a touch of **residual sugar** ; it's a slightly" → attends to **riesling** and **sugar**: **sweet 94.23%**.

- Unlike an RNN (which keeps everything in one hidden state), attention **directly looks back** at the needed word.

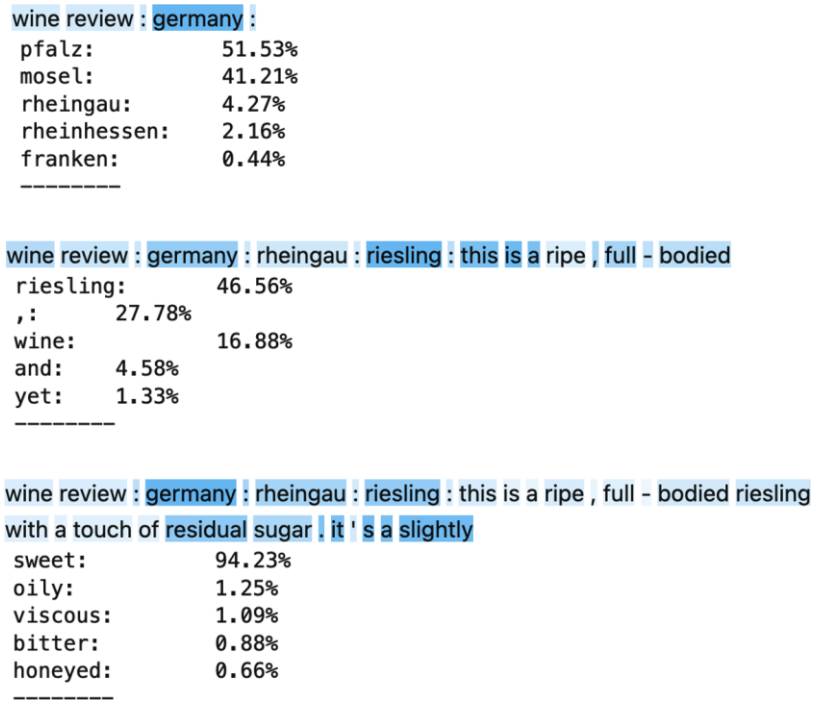


Figure 9-11. Distribution of word probabilities following various sequences

Other Transformers

- **Decoder Transformer** (our GPT): generates text **one token at a time** and uses **causal masking** to only attend to **previous words**.
- **Encoder Transformer**: **no causal masking** - attends to the **entire input** to extract a meaningful **contextual representation**.
- **Encoder-decoder Transformer**: used for tasks such as **language translation** - contains **both** encoder and decoder Transformer blocks.

Table 9-1. The three Transformer architectures

Type	Examples	Use cases
Encoder	BERT (Google)	Sentence classification, named entity recognition, extractive question answering
Encoder-decoder	T5 (Google)	Summarization, translation, question answering
Decoder	GPT-3 (OpenAI)	Text generation

Other Transformers

Encoder Transformers - BERT

- **BERT** = **Bidirectional Encoder Representations from Transformers**, developed by **Google (Devlin et al., 2018)**.
- It **predicts missing words** from a sentence using context from **both before and after** the missing word, in all layers.
- **Example:** “The elephant was too [MASK] to fit in the car” → BERT uses words on **both sides** to predict **big**.
- Encoder Transformers are used for tasks that need an **understanding of the input as a whole**: **sentence classification, named entity recognition, extractive question answering**.
- They are **not used for text generation**.

Coming next

- How **encoder-decoder** Transformers work (**T5**).
- Extensions of GPT by OpenAI - **GPT-3, GPT-4** and **ChatGPT** (designed for conversation).

T5

- **T5** from **Google** is a modern Transformer that uses the **encoder-decoder** structure.
- It **reframes many tasks into a text-to-text framework** - translation, linguistic acceptability, sentence similarity and document summarization.
- Input is text with a **task prefix**; output is also **text**.

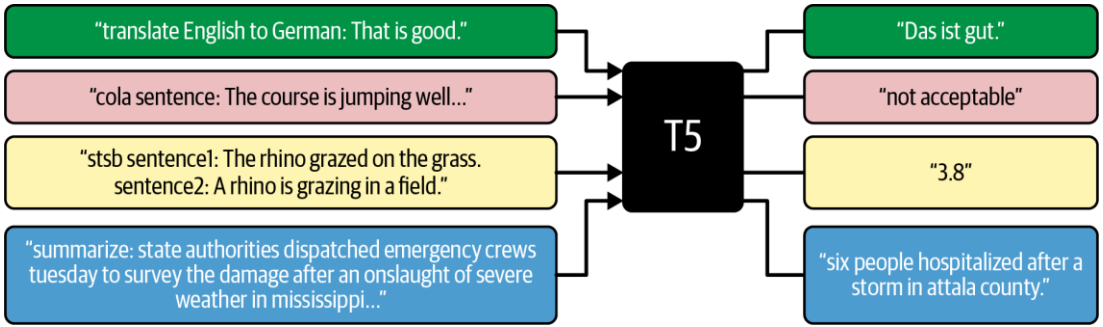


Figure 9-12. How T5 reframes a range of tasks into a text-to-text framework (Raffel et al., 2019)

Examples (from figure):

- "translate English to German: That is good." → **"Das ist gut."**
- "cola sentence: The course is jumping well." → **"not acceptable"**
- "stsb sentence1: The rhino grazed on the grass. sentence2: A rhino is grazing in a field." → **"3.8"**
- "summarize: …" → **short summary**

- T5 is trained on **750 GB** (C4 - Colossal Clean Crawled Corpus) vs **1.4 GB** English-German pairs in the original Transformer.

T5

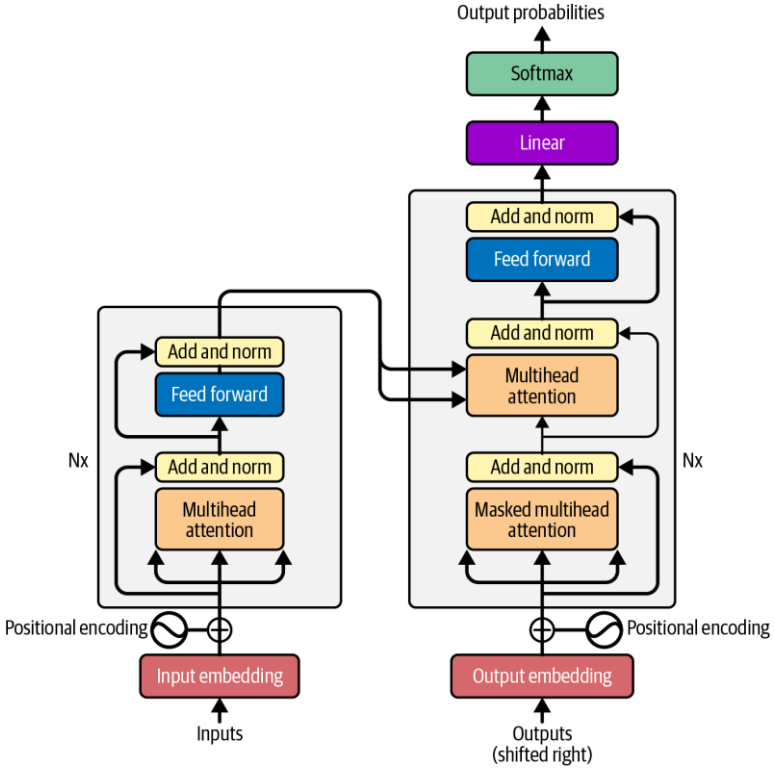


Figure 9-13. An encoder-decoder Transformer model (Vaswani et al., 2017)

- Transformer blocks are **repeated** and **positional embedding** captures the order of the input.

Two key differences from GPT

1. Encoder (left side):

- Encodes the sequence to be translated. **No causal masking** - we only want a good **representation of the whole sequence**, capturing all cross-dependencies between words, in any order.

2. Decoder (right side):

- Generates the translated text. First attention layer is **self-referential** (key, value, query from same input) with **causal masking**.
- Next attention layer takes **key and value from the encoder** and **query from the decoder → cross-referential attention**. This is how the decoder knows **what meaning** the translation must convey.

T5

Example: Cross-Referential Attention

- Translate: “The animal didn’t cross **the street** because it was too tired” → German.
- German has **three definite articles**: **der, die, das**, depending on the **gender of the noun**.
- **One attention head** attends to the word **“the”** (the word to translate).
- **Another head** attends to **“street”** (Straße - a **feminine** word in German).
- Working together, the Transformer correctly chooses **“die”**.
- Source: Tensor2Tensor GitHub repository (Colab notebook).

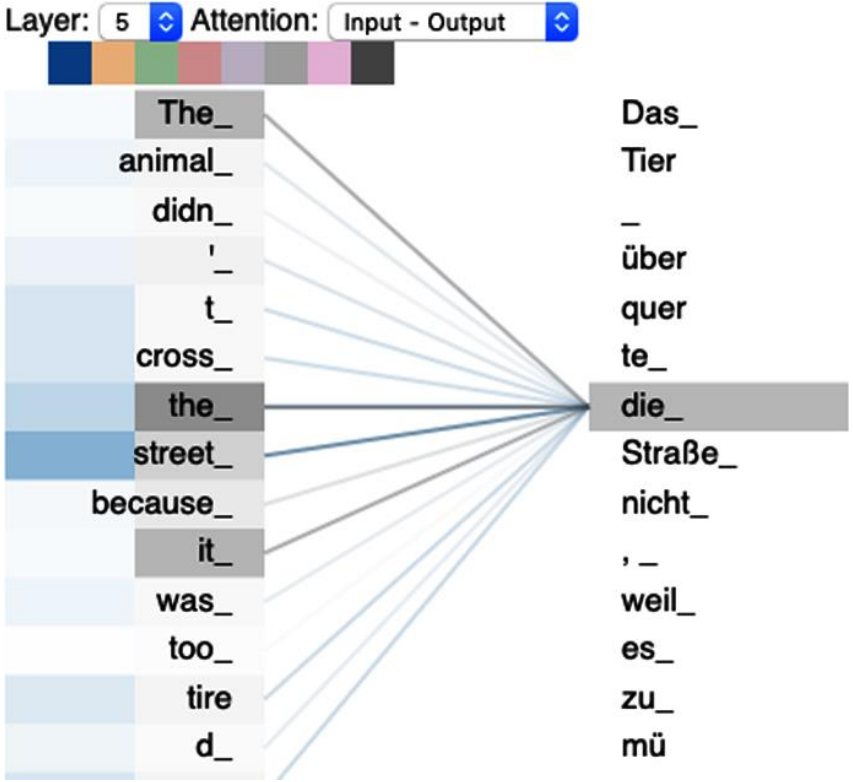


Figure 9-14. One head attends to “the” and another to “street” to translate “the” as “die”

GPT-3 and GPT-4

Table 9-2. The evolution of OpenAI's GPT collection of models

Model	Date	Layers	Attention heads	Word embedding size	Context window	# parameters	Training data
GPT	Jun 2018	12	12	768	512	120,000,000	BookCorpus: 4.5 GB of text from unpublished books
GPT-2	Feb 2019	48	48	1,600	1,024	1,500,000,000	WebText: 40 GB of text from outbound Reddit links
GPT-3	May 2020	96	96	12,888	2,048	175,000,000,000	CommonCrawl, WebText, English Wikipedia, book corpora and others: 570 GB
GPT-4	Mar 2023	-	-	-	-	-	-

- GPT-3 architecture is **similar to the original GPT** but **much larger** and trained on **much more data**.
- GPT-4 details are **not public**, but it can **accept images as input** - a **multimodal** model. GPT-3/GPT-4 weights are **not open source** (available through a commercial tool and API).

GPT-3 and GPT-4

Fine-tuning versus Few-shot Learning

- **Fine-tuning:** GPT-3 can be fine-tuned on your own data by **updating the weights** of the network.
- **Few-shot learning:** GPT-3 can be told how to react simply by giving **a few examples in the prompt** itself.
- Benefit of fine-tuning: no need to give the examples **in every prompt → saves cost** in the long run.

Few-shot example:

"happy → positive; sad → negative; excited → ?" Model answers: **positive**

Scaling

- Language models benefit hugely from **scaling** - both **number of weights** and **dataset size**. The ceiling has **not yet been reached**.

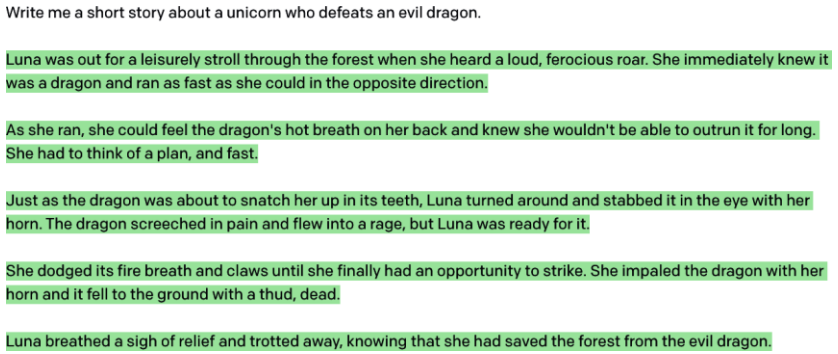


Figure 9-15. An example of how GPT-3 can extend a given system prompt

ChatGPT

- Announced by OpenAI a few months **before GPT-4** - a tool to interact with large language models through a **conversational interface**.
- Original release (**November 2022**) was powered by **GPT-3.5** - more powerful than GPT-3 and **fine-tuned for conversation**.
- It **maintains state between inputs**: in Figure 9-16 it understands that "attention" in the second question refers to **attention in Transformers**, not a person's ability to focus.
- Uses **Reinforcement Learning from Human Feedback (RLHF)** to fine-tune GPT-3.5.
- RLHF was also used in the earlier **InstructGPT** model - a fine-tuned GPT-3 designed to **follow written instructions** accurately.

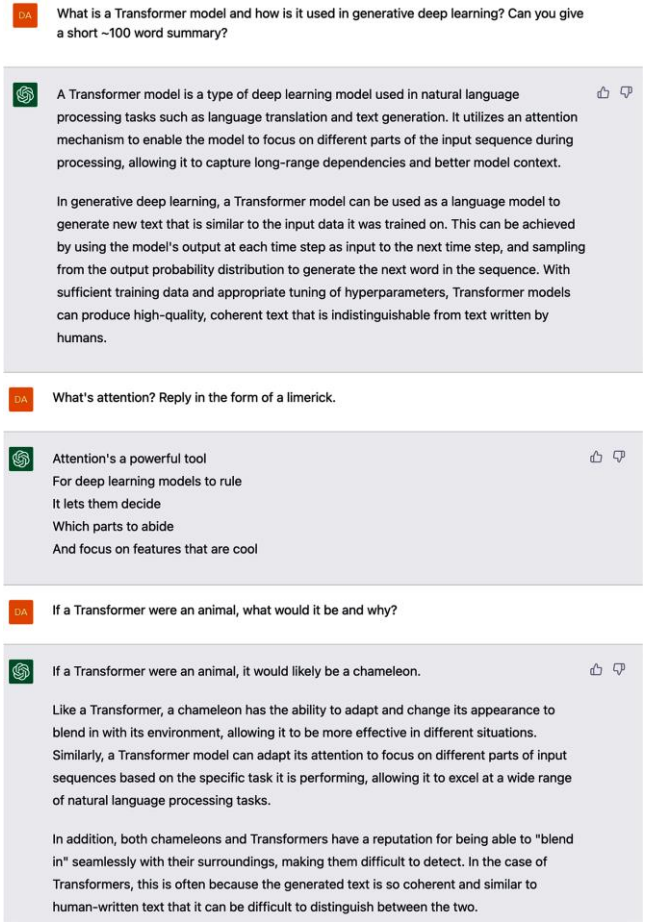


Figure 9-16. ChatGPT answering questions about Transformers

ChatGPT

Training process of ChatGPT (RLHF)

1. Supervised fine-tuning: Collect a **demonstration dataset** of prompts and **human-written desired outputs**; fine-tune GPT-3.5 using **supervised learning**.

2. Reward modeling: A **human labeler ranks** several sampled outputs from **best to worst**. Train a **reward model** that predicts the score of each output, given the conversation history.

3. Reinforcement learning: The conversation is an RL **environment**; the **policy** is the language model (from step 1). Given the **state** (conversation history), the policy outputs an **action** (sequence of tokens) which is scored by the **reward model**. **PPO** (proximal policy optimization) adjusts the weights to **maximize the reward**.

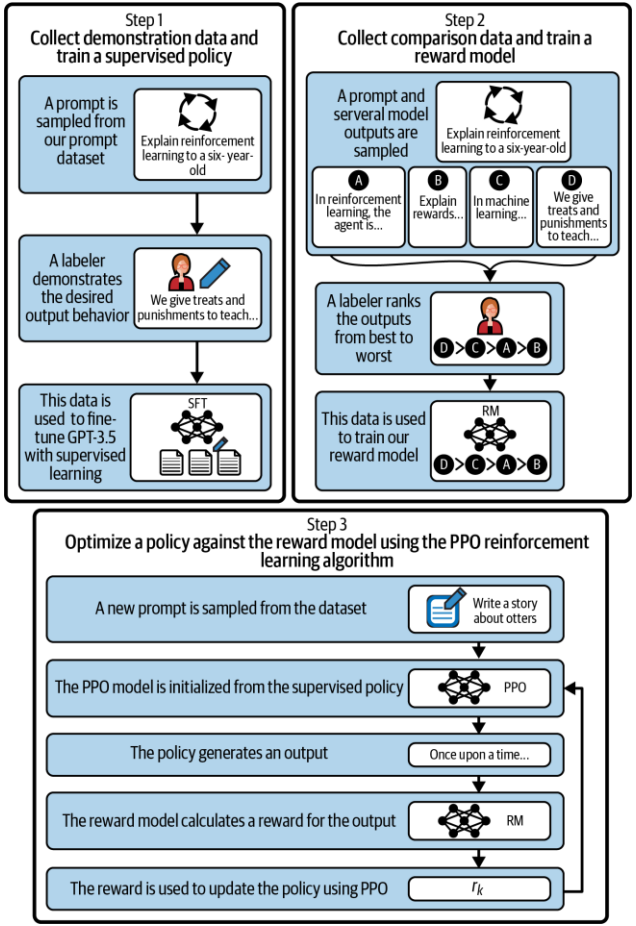


Figure 9-17. RLHF fine-tuning process used in ChatGPT (OpenAI)

ChatGPT

Easy Understanding of RLHF

- **Step 1** - Teacher shows the student **model answers** (supervised fine-tuning).
- **Step 2** - Teacher **ranks** the student's answers; a **reward model** learns how the teacher grades.
- **Step 3** - The student **practises** and improves by trying to get **higher marks** from the reward model (PPO).

Limitations and Future

- ChatGPT sometimes **"hallucinates"** - produces **factually incorrect** information.
- Still, it shows how Transformers can produce **complex, long-ranging and novel output** often **indistinguishable from human text**.
- **Visual ChatGPT** combines ChatGPT with visual foundation models such as **Stable Diffusion** - interaction through **text and images**.
- **Visual ChatGPT** and **GPT-4** may begin a **new era of human-computer interaction**.

Summary - Transformers

- **GPT** uses **attention**, which removes the need for recurrent layers (LSTMs). Attention works like an **information retrieval system** using **queries, keys and values**.
- Attention heads are grouped into a **multihead attention layer**, wrapped inside a **Transformer block** with **layer normalization** and **skip connections**. Blocks are **stacked** for deep networks.
- **Causal masking** ensures GPT cannot leak information from **future tokens**.
- **Positional encoding** keeps the **ordering** of the input sequence.
- We can **generate text** and also **inspect attention scores** to see where the model looks. GPT accesses information **at a distance without loss of signal**.
- Three families: **encoder** (BERT), **decoder** (GPT) and **encoder-decoder** (T5).
- Large language models: **T5, GPT-3, GPT-4** and **ChatGPT** (trained with **RLHF**).

Advanced GANs

Text Book 1: Chapter 10

Advanced GANs - Chapter Goals

In this chapter we will:

- See how **ProGAN** progressively trains a GAN to generate **high-resolution images**.
- Understand how ProGAN was adapted to build **StyleGAN**, a high-performing GAN for image synthesis.
- Explore how StyleGAN was adjusted to create **StyleGAN2**.
- Learn key contributions: **progressive training, adaptive instance normalization, weight modulation and demodulation, path length regularization**.
- Walk through the **Self-Attention GAN (SAGAN)**, which adds attention to the GAN framework.
- See how **BigGAN** expands on SAGAN to produce high-quality images.
- Learn how **VQ-GAN** uses a **codebook** to encode images into a **discrete sequence of tokens** modelled by a Transformer.
- See how **ViT VQ-GAN** uses **Transformers** instead of convolutional layers in the encoder and decoder.

Introduction

Recap: Generative Adversarial Network (GAN) - Chapter 4

- **Generator (G):** converts random noise **z** into a fake image.
- **Discriminator (D):** predicts whether an image is **real or fake**.
- Both are trained together in a **fight for dominance** - G tries to fool D, D tries to catch G.

Easy analogy: G is a **counterfeiter** making fake notes; D is the **police** checking them. Both keep improving.

Why "Advanced" GANs?

- The flexible architecture led to many new GAN designs - the **GAN Zoo** repository lists **over 500 GANs** (ABC-GAN to ZipNet-GAN).
- This chapter covers the most influential ones:
  - **NVIDIA:** ProGAN, StyleGAN, StyleGAN2
  - **With attention:** Self-Attention GAN (SAGAN) and BigGAN
  - **VAE + Transformer + GAN:** VQ-GAN and ViT VQ-GAN (VQ-GAN is a key part of Google's text-to-image model **Muse**)

ProGAN

- **ProGAN** (Progressive Growing of GANs) - developed by **NVIDIA Labs in 2017** to improve both the **speed and stability** of GAN training.
- Instead of training directly on **full-resolution** images, ProGAN first trains G and D on **low-resolution images** (e.g., **4 × 4 pixels**) and then **incrementally adds layers** to increase the resolution.

Problem with a normal GAN

- The generator **always outputs full-resolution images**, even in the early stages of training.
- It may be **slow to learn high-level structures**, because it immediately works on **complex, high-resolution images**.

Idea of progressive training

- First train a **lightweight GAN** to output **accurate low-resolution images**, then **gradually increase** the resolution.

Easy analogy: An artist first draws a **rough outline**, then adds **shapes**, and finally **fine details** like hair and eyes.

ProGAN - Progressive Training

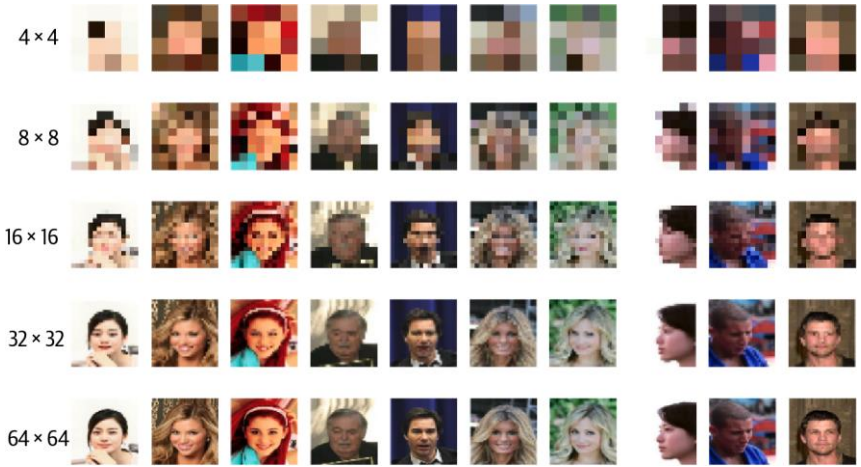


Figure 10-1. Images in the dataset can be compressed to lower resolution using interpolation

- Training starts with a training set **condensed down to 4 × 4 pixel** images using **interpolation**.
- Then 8 × 8, 16 × 16, 32 × 32, 64 × 64 …

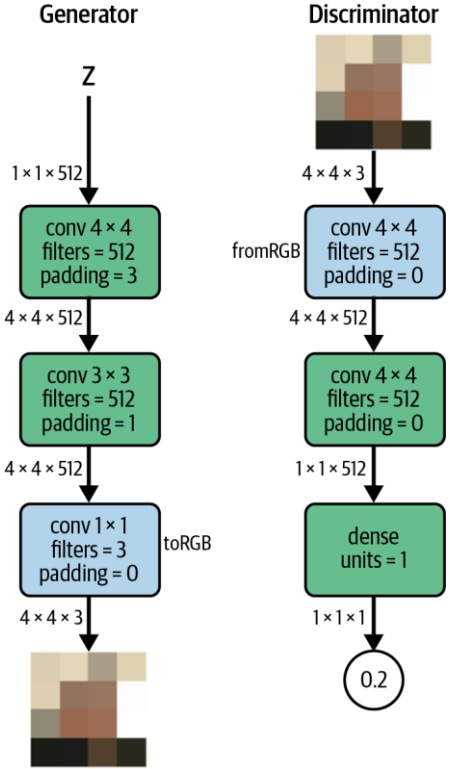


Figure 10-2. G and D for the first stage of ProGAN

First stage:

- Generator: latent noise vector **z** (length **512**) → image **4 × 4 × 3**.
- Discriminator: image **4 × 4 × 3** → **single scalar** prediction.
- **toRGB** (blue box in G): converts feature maps → **RGB image**.
- **fromRGB** (blue box in D): converts RGB image → **feature maps**.
- Trained until D has seen **800,000 real images**.

ProGAN - Progressive Training

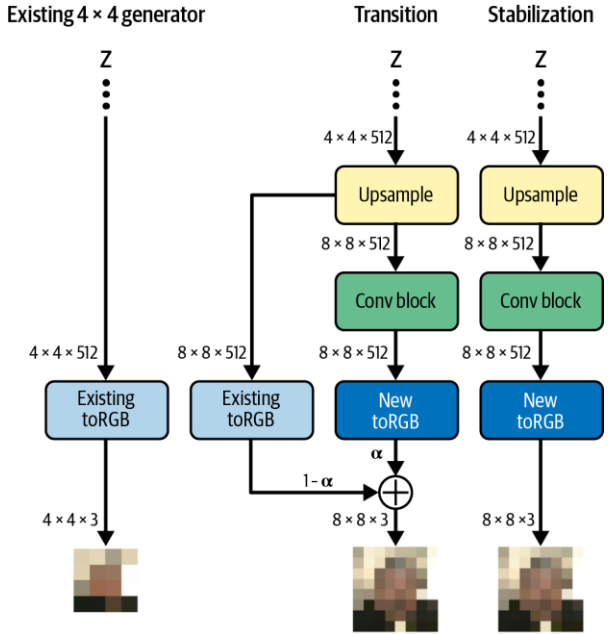


Figure 10-3. ProGAN generator training: expanding from 4 × 4 to 8 × 8

Fade-in (blending):

$$output = (1 - \alpha) \cdot old_toRGB(upsample) + \alpha \cdot new_toRGB$$

Expanding the Generator (4 × 4 → 8 × 8)

Done in **two phases**:

1. Transition phase

- New **upsampling and convolutional layers** are appended.
- A **residual connection** keeps the output of the **old trained toRGB** layer.
- New layers are **masked by a parameter α**, gradually increased **from 0 to 1**, allowing more of the **new toRGB** output and less of the old - to **avoid a shock** to the network.

2. Stabilization phase

- No flow through the old toRGB layer; the network **fine-tunes** the output.

ProGAN - Progressive Training

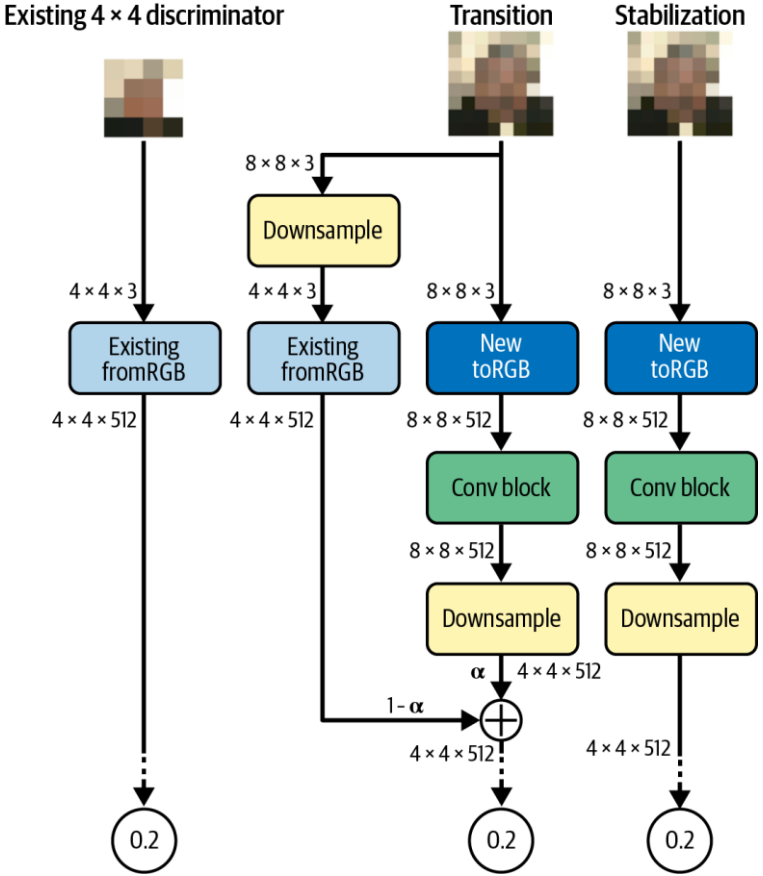


Figure 10-4. ProGAN discriminator training: expanding from 4 × 4 to 8 × 8

Expanding the Discriminator

- Additional **downscaling and convolutional layers** are blended in.
- The layers are injected at the **start of the network**, just after the **input image**.
- The existing **fromRGB** layer is connected via a **residual connection** and **gradually phased out** during the **transition phase**.
- The **stabilization phase** lets the discriminator **fine-tune** using the new layers.

Important points

- Each transition and stabilization phase lasts until D has seen **800,000 real images**.
- **No layers are frozen** - all layers remain **fully trainable** throughout.

ProGAN - Progressive Training

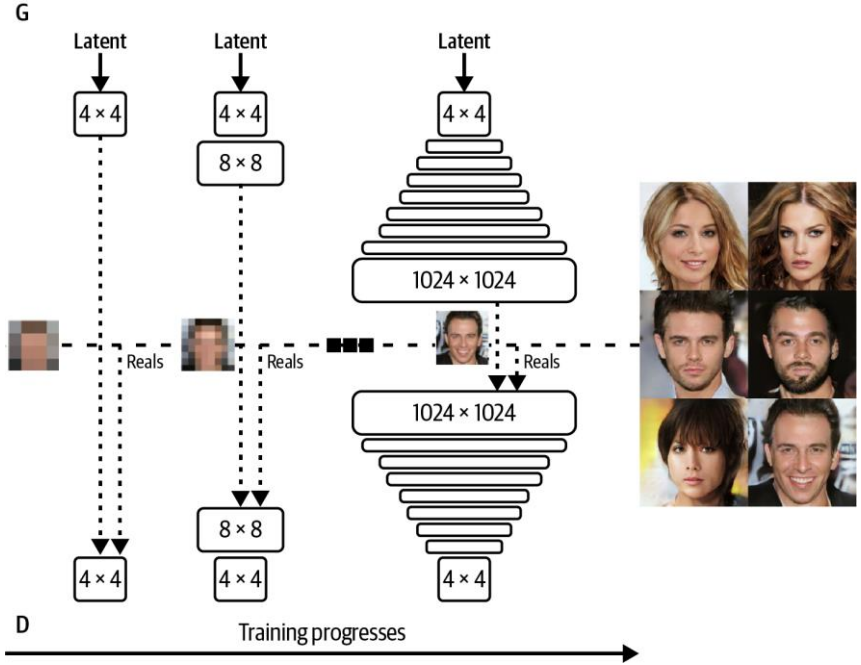


Figure 10-5. The ProGAN training mechanism and some example generated faces (Karras et al., 2017)

Example: To draw a 1024 × 1024 face, the model first learns a blurry 4 × 4 face, and finally a sharp, realistic face.

- The GAN grows from **4 × 4 → 8 × 8 → 16 × 16 → 32 × 32 → …** until it reaches **full resolution (1,024 × 1,024)**.
- **Low resolutions** learn the **overall structure** (face shape, position).
- **High resolutions** learn the **fine details** (hair, skin texture).
- Figure 10-6 shows the final G and D used to generate **1,024 × 1,024 CelebA faces** (23.1M trainable parameters each).

ProGAN - Progressive Training

Generator	Act.	Output shape	Params
Latent vector	–	512 × 1 × 1	–
Conv 4 × 4	LReLU	512 × 4 × 4	4.2M
Conv 3 × 3	LReLU	512 × 4 × 4	2.4M
Upsample	–	512 × 8 × 8	–
Conv 3 × 3	LReLU	512 × 8 × 8	2.4M
Conv 3 × 3	LReLU	512 × 8 × 8	2.4M
Upsample	–	512 × 16 × 16	–
Conv 3 × 3	LReLU	512 × 16 × 16	2.4M
Conv 3 × 3	LReLU	512 × 16 × 16	2.4M
Upsample	–	512 × 32 × 32	–
Conv 3 × 3	LReLU	512 × 32 × 32	2.4M
Conv 3 × 3	LReLU	512 × 32 × 32	2.4M
Upsample	–	512 × 64 × 64	–
Conv 3 × 3	LReLU	256 × 64 × 64	1.2M
Conv 3 × 3	LReLU	256 × 64 × 64	590k
Upsample	–	256 × 128 × 128	–
Conv 3 × 3	LReLU	128 × 128 × 128	295k
Conv 3 × 3	LReLU	128 × 128 × 128	148k
Upsample	–	128 × 256 × 256	–
Conv 3 × 3	LReLU	64 × 256 × 256	74k
Conv 3 × 3	LReLU	64 × 256 × 256	37k
Upsample	–	64 × 512 × 512	–
Conv 3 × 3	LReLU	32 × 512 × 512	18k
Conv 3 × 3	LReLU	32 × 512 × 512	9.2k
Upsample	–	32 × 1024 × 1024	–
Conv 3 × 3	LReLU	16 × 1024 × 1024	4.6k
Conv 3 × 3	LReLU	16 × 1024 × 1024	2.3k
Conv 1 × 1	linear	3 × 1024 × 1024	51
Total trainable parameters			**23.1M**

Discriminator	Act.	Output shape	Params
Input image	–	3 × 1024 × 1024	–
Conv 1 × 1	LReLU	16 × 1024 × 1024	64
Conv 3 × 3	LReLU	16 × 1024 × 1024	2.3k
Conv 3 × 3	LReLU	32 × 1024 × 1024	4.6k
Downsample	–	32 × 512 × 512	–
Conv 3 × 3	LReLU	32 × 512 × 512	9.2k
Conv 3 × 3	LReLU	64 × 512 × 512	18k
Downsample	–	64 × 256 × 256	–
Conv 3 × 3	LReLU	64 × 256 × 256	37k
Conv 3 × 3	LReLU	128 × 256 × 256	74k
Downsample	–	128 × 128 × 128	–
Conv 3 × 3	LReLU	128 × 128 × 128	148k
Conv 3 × 3	LReLU	256 × 128 × 128	295k
Downsample	–	256 × 64 × 64	–
Conv 3 × 3	LReLU	256 × 64 × 64	590k
Conv 3 × 3	LReLU	512 × 64 × 64	1.2M
Downsample	–	512 × 32 × 32	–
Conv 3 × 3	LReLU	512 × 32 × 32	2.4M
Conv 3 × 3	LReLU	512 × 32 × 32	2.4M
Downsample	–	512 × 16 × 16	–
Conv 3 × 3	LReLU	512 × 16 × 16	2.4M
Conv 3 × 3	LReLU	512 × 16 × 16	2.4M
Downsample	–	512 × 8 × 8	–
Conv 3 × 3	LReLU	512 × 8 × 8	2.4M
Conv 3 × 3	LReLU	512 × 8 × 8	2.4M
Downsample	–	512 × 4 × 4	–
Minibatch stddev	–	513 × 4 × 4	–
Conv 3 × 3	LReLU	512 × 4 × 4	2.4M
Conv 4 × 4	LReLU	512 × 1 × 1	4.2M
Fully-connected	linear	1 × 1 × 1	513
Total trainable parameters			**23.1M**

Figure 10-6. The ProGAN generator and discriminator used to generate 1,024 × 1,024-pixel CelebA faces (Karras et al., 2018)

ProGAN - Other Contributions

1. Minibatch Standard Deviation

- An **extra layer in the discriminator** that appends the **standard deviation of the feature values** (averaged across all pixels and the minibatch) as an **additional constant feature**.
- If the generator produces **low variety**, the std is **small** → D can use it to **detect fake batches**.
- So the generator is **encouraged to produce variety** similar to the real training data.

Example: If G produces 16 almost identical faces in a batch, D notices “too similar = fake”.

2. Pixelwise Normalization

- Used in the **generator** instead of batch normalization.
- Normalizes the **feature vector in each pixel to unit length** → prevents the signal from **spiralling out of control**.
- Has **no trainable weights**.

$$b_{x,y} = a_{x,y} / \sqrt{((1/N) \Sigma_j (a^j_{x,y})^2 + \varepsilon)}$$

ProGAN - Other Contributions

3. Equalized Learning Rates

- Usually weights are initialized using **He initialization** - Gaussian with std **inversely proportional to √(number of inputs)**.
- Optimizers such as **Adam / RMSProp** normalize gradient updates, so the update size is **independent of the weight scale**.
- Problem: weights with a **larger dynamic range** (layers with **fewer inputs**) take **longer to adjust** → **imbalance** in training speed between layers of G and D.

Solution in ProGAN

- Weights are initialized from a **simple standard Gaussian N(0, 1)**, regardless of the number of inputs.
- The **He scaling is applied dynamically** each time the layer is called (not only at initialization):

$$\hat{w}_i = w_i \cdot c\,, \qquad c = \sqrt{(\,2 / n_{in}\,)}$$

- The optimizer sees **every weight with the same dynamic range** → applies the **same learning rate** to all.

ProGAN - Outputs

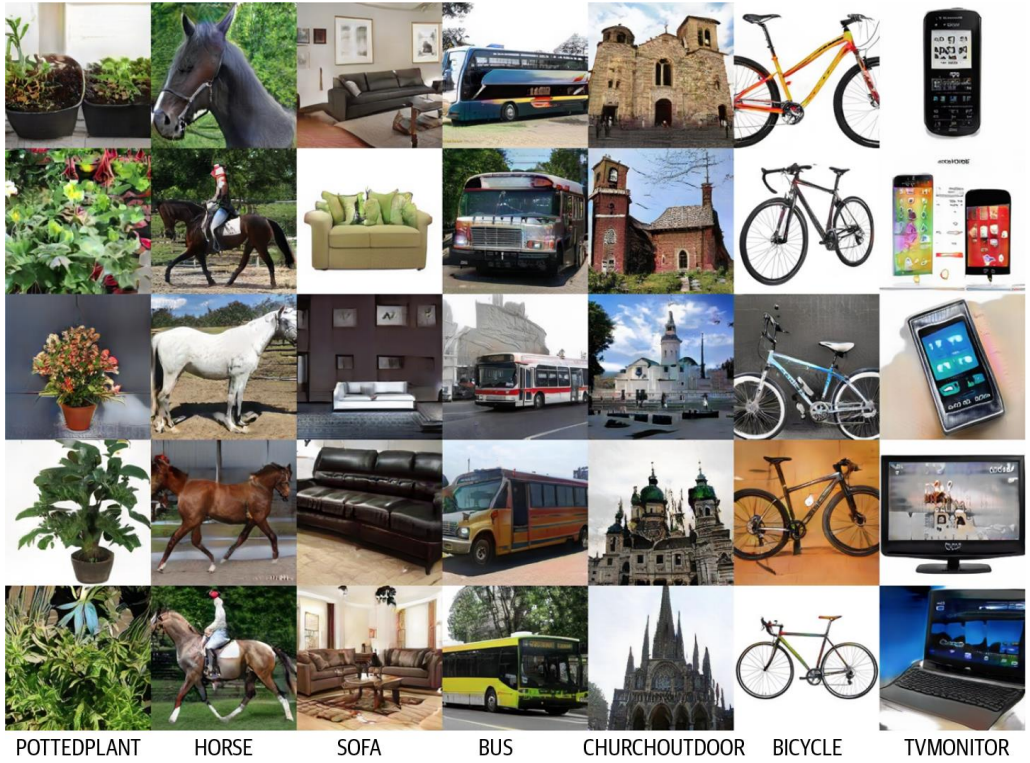


Figure 10-7. Generated examples from a ProGAN trained progressively on the LSUN dataset at 256 × 256 resolution (Karras et al., 2017)

- Besides **CelebA faces**, ProGAN was applied to the **Large-scale Scene Understanding (LSUN)** dataset with excellent results.
- Categories: **potted plant, horse, sofa, bus, church outdoor, bicycle, TV monitor**.
- Demonstrated the power of ProGAN over earlier GANs and paved the way for **StyleGAN** and **StyleGAN2**.

StyleGAN

- **StyleGAN** (2018) builds on **ProGAN**. The **discriminator is identical**; only the **generator is changed**.

Problem: Entangled latent space

- Vectors in the latent space for high-level attributes are often **entangled**.

Example: Adjusting the latent vector to give a face **more freckles** may also **change the background colour**!

- For full control of image **style**, we need **disentangled** features.

StyleGAN solution

- **Inject style vectors** at **different points** of the network:
  - Some control **high-level features** (e.g., **face orientation**).
  - Some control **low-level details** (e.g., **the way the hair falls across the forehead**).

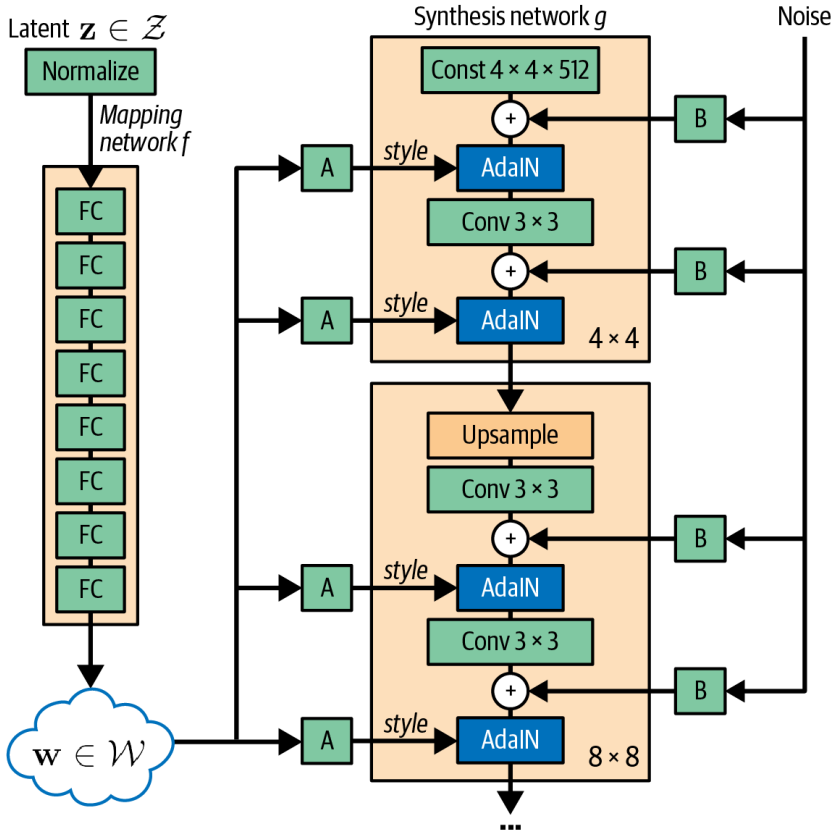


Figure 10-8. The StyleGAN generator architecture (Karras et al., 2018)

StyleGAN

1. The Mapping Network (f)

- A simple **feed-forward network (8 FC layers)** that converts input noise **z ∈ Z** into a different latent space **w ∈ W**.

$$f : z \in Z \rightarrow w \in W$$

- Gives the generator a chance to **disentangle** the noisy input into **distinct factors of variation**.
- Separates **choosing a style** (mapping network) from **generating the image** with that style (synthesis network).

2. The Synthesis Network (g)

- The **generator of the actual image** with a given style.
- Style vector **w** is injected at different points, each time via a different dense layer $\mathbf{A_i}$, which generates:
  - a **bias vector** $\mathbf{y_{b,i}}$ and a **scaling vector** $\mathbf{y_{s,i}}$
- These tell the synthesis network **how to adjust the feature maps** towards the specified style - using **AdaIN** layers.

Analogy: Mapping network = **fashion designer** choosing a style; synthesis network = **tailor** stitching the dress in that style.

StyleGAN - Adaptive Instance Normalization

- An **AdaIN layer** adjusts the **mean and variance** of each feature map $\mathbf{x_i}$ using a reference style **bias** $\mathbf{y_{b,i}}$ and **scale** $\mathbf{y_{s,i}}$.
- Both vectors have length = **number of channels** output by the preceding convolutional layer.

$$AdaIN(x_i, y) = y_{s,i} \cdot (x_i - \mu(x_i)) / \sigma(x_i) + y_{b,i}$$

Meaning of each term

- $(x_i - \mu(x_i)) / \sigma(x_i) \rightarrow$ **normalize** the feature map (mean 0, std 1) - removes the old style.
- $\times y_{s,i} \rightarrow$ apply the **new scale**; $+ y_{b,i} \rightarrow$ apply the **new bias** - adds the new style.

Why AdaIN?

- Styles injected into each layer **only affect features at that layer** - no style information leaks between layers.
- Latent vectors **w** become **much more disentangled** than the original **z** vectors.
- Earlier layers (**4 × 4, 8 × 8**) affect **coarse features**; later layers (**64 × 64 to 1,024 × 1,024**) affect **fine details**.
- We can **switch the w vector** at different points to change style at **different levels of detail**.

StyleGAN

3. Style Mixing

- Ensures the generator **cannot use correlations between adjacent styles** - keeps styles **disentangled**.
- Instead of one latent vector, **two are sampled**: $\mathbf{z_1}$, $\mathbf{z_2}$ → two style vectors $\mathbf{w_1}$, $\mathbf{w_2}$.
- At **each layer**, either $\mathbf{w_1}$ **or** $\mathbf{w_2}$ is chosen **at random**, breaking any correlation between the vectors.

4. Stochastic Variation

- The synthesis network adds **noise** (through a learned broadcasting layer **B**) after **each convolution**.
- Accounts for **random details** such as the **placement of individual hairs** or the **background behind the face**.
- The **depth** at which noise is injected decides how **coarse** its effect is.
- Therefore the **initial input** to the synthesis network can be a **learned constant** ($4 \times 4 \times 512$) instead of noise - there is enough randomness from the **styles and noise inputs**.

StyleGAN - Outputs

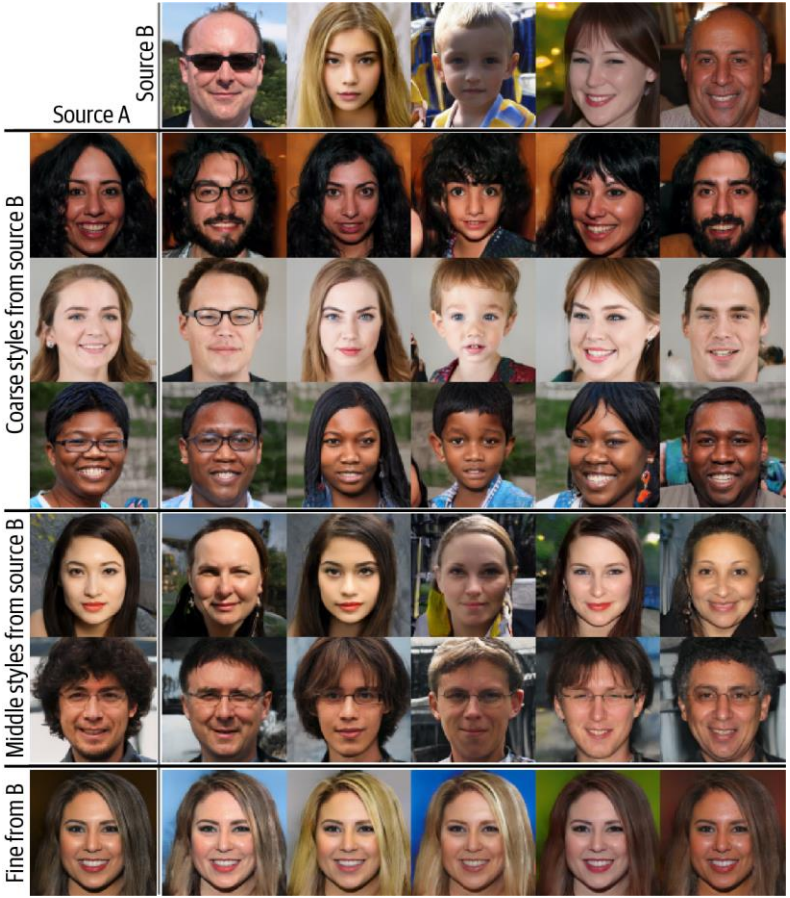


Figure 10-9. Merging styles between two generated images at different levels of detail (Karras et al., 2018)

- Two images, **source A** and **source B**, are generated from two different **w** vectors.
- To merge: the **source A w** is passed through the synthesis network but, at some point, **switched to source B w**.

Switch early (4 × 4 or 8 × 8)

- **Coarse styles** from B - **pose, face shape, glasses** - are carried onto A.

Switch later

- Only **fine-grained details** from B - **colours and microstructure** of the face - are carried across.
- The **coarse features of A** are preserved.

StyleGAN2

- **StyleGAN2** builds further on StyleGAN with key changes that **improve the quality** of the generated output.

Problem in StyleGAN - Artifacts

- StyleGAN images suffered from **artifacts - water droplet-like areas** in the image.
- These were found to be **caused by the AdaIN layers**.
- Figure 10-10 shows an artifact (red circle) in a StyleGAN face.

- In StyleGAN2, **both the generator and the discriminator** are different from StyleGAN.

Key changes

1. **Weight modulation and demodulation** (replaces AdaIN)
2. **Path length regularization**
3. **No progressive growing**



Figure 10-10. An artifact in a StyleGAN-generated image of a face (Karras et al., 2019)

StyleGAN2 - Weight Modulation and Demodulation

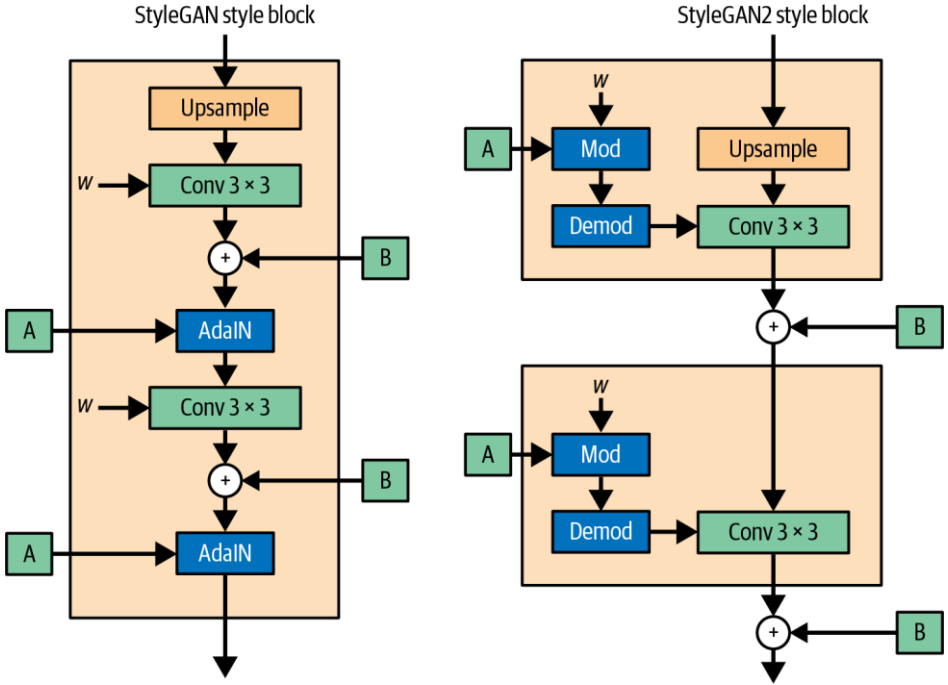


Figure 10-11. A comparison between the StyleGAN and StyleGAN2 style blocks

Remember: AdaIN = instance normalization + style modulation (scale and bias).

- Artifacts are removed by **removing AdaIN** and replacing it with **weight modulation (Mod)** and **demodulation (Demod)**.
- **w** = weights of the convolutional layer, **updated at runtime** by Mod and Demod.
- **StyleGAN:** AdaIN works on the **image tensor** (output of the conv layer).
- **StyleGAN2:** style modulation and normalization are applied **directly to the conv weights**.
- This **removes artifacts** while **keeping control** of image style.

StyleGAN2 - Weight Modulation and Demodulation

Step 1: Modulation

- Each dense layer **A** outputs a single style vector $\mathbf{s_i}$ (i indexes the **input channels** of the conv layer).

$$w'_{i,j,k} = s_i \cdot w_{i,j,k}$$

- **j** indexes the **output channels**, **k** indexes the **spatial dimensions**.

Step 2: Demodulation

- Normalize the weights so they again have **unit standard deviation → stable training**.

$$w''_{i,j,k} = w'_{i,j,k} / \sqrt{(\Sigma_{i,k} (w'_{i,j,k})^2 + \varepsilon)}$$

- **ε** is a small constant that **prevents division by zero**.

Easy understanding: Modulation = **turning up/down the volume** of each input channel according to the style; Demodulation = **resetting the overall volume** so nothing becomes too loud (no water droplets).

StyleGAN2 - Path Length Regularization

- An additional **penalty term** in the loss function.
- Goal: the latent space should be **smooth and uniform** - a **fixed-size step** in the latent space in any direction should give a **fixed-magnitude change** in the image.
- StyleGAN2 minimizes the following term (alongside the usual **Wasserstein loss with gradient penalty**):

$$E_{w,y} \left(\| J_w^T y \|_2 - a \right)^2$$

- **w** - style vectors created by the mapping network; **y** - noisy images drawn from **N(0, I)**.
- $\mathbf{J_w = \partial g / \partial w}$ - the **Jacobian** of the generator with respect to the style vectors.
- $\| \mathbf{J_w^T y} \|_2$ - magnitude of images y after transformation by the Jacobian; it should be close to a constant **a**.
- **a** = **exponential moving average** of $\| J_w^T y \|_2$ during training.
- Makes exploring the latent space **more reliable and consistent**.
- **Lazy regularization:** regularization terms are applied only **once every 16 minibatches** - no measurable drop in performance.

StyleGAN2 - No Progressive Growing

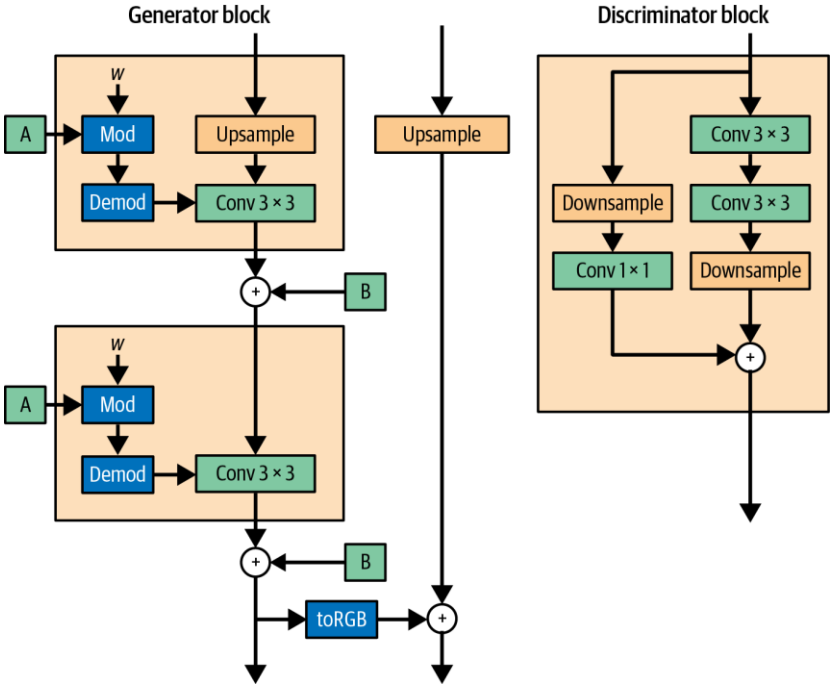


Figure 10-12. The generator and discriminator blocks in StyleGAN2

- StyleGAN2 does **not** use progressive training.
- Uses **skip connections in the generator** and **residual connections in the discriminator** to train the **entire network as one**.
- Different resolutions **need not be trained independently** and blended.
- Desired property **is still preserved**: StyleGAN2 first learns **low-resolution features** and gradually **refines** the output.

StyleGAN2 - No Progressive Growing

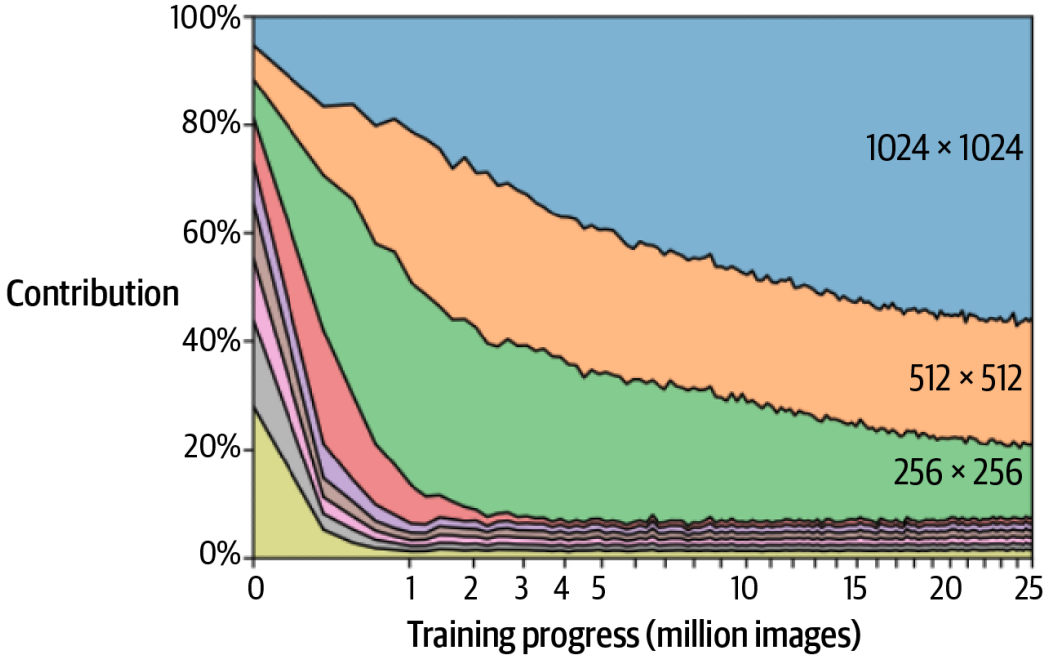


Figure 10-13. Contribution of each resolution layer to the generator output, by training time (adapted from Karras et al., 2019)

- **Early in training:** the **lower-resolution layers** are refined; skip/residual connections pass the output through the higher-resolution layers **mostly unaffected**.
- **As training progresses:** the **higher-resolution layers begin to dominate**, as the generator finds more intricate ways to make images realistic and **fool the discriminator**.
- In the figure, the **1024 × 1024** layer's contribution grows over time.

StyleGAN2 - Outputs



Figure 10-14. Uncurated StyleGAN2 output for the FFHQ face dataset and LSUN car dataset (Karras et al., 2019)

- StyleGAN2 (and scaled variants such as **StyleGAN-XL**) remain **state of the art** for image generation on datasets such as **Flickr-Faces-HQ (FFHQ)** and **CIFAR-10** (Papers with Code).
- Generated faces and cars are **highly realistic** and **free from water-droplet artifacts**.

Comparison: ProGAN, StyleGAN and StyleGAN2

Feature	ProGAN (2017)	StyleGAN (2018)	StyleGAN2 (2019)
Main idea	Progressive training from 4 × 4 to 1024 × 1024	Style vectors injected at different layers	Improves StyleGAN image quality
Generator	Grows layer by layer	Mapping network + synthesis network	Mapping + synthesis, with Mod/Demod
Discriminator	Grows layer by layer	Same as ProGAN	Residual connections
Normalization	Pixelwise normalization	AdaIN	Weight demodulation
Training	Progressive growing	Progressive growing	No progressive growing (skip/residual)
Key extras	Minibatch std, equalized learning rate	Style mixing, stochastic noise	Path length regularization, lazy regularization
Weakness	Entangled latent space	Water-droplet artifacts	—

Self-Attention GAN (SAGAN)

- **SAGAN** shows how the **attention mechanism** (that powers Transformers) can be used inside **GANs for image generation**.
- **Problem:** convolutional feature maps process information **only locally**. To connect pixels on **opposite sides** of an image, many conv layers are needed; positional information is lost → learning **long-range dependencies** is **computationally inefficient**.
- **Solution:** add **self-attention** to the GAN, so any pixel can **directly look at any other pixel**.

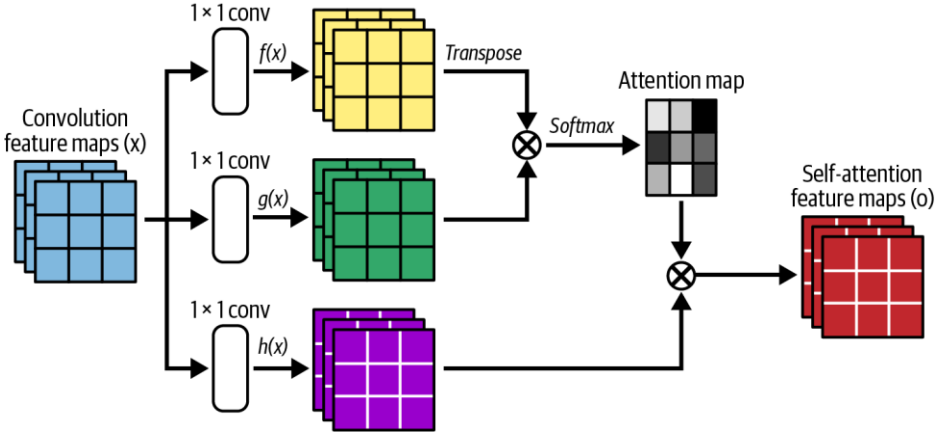


Figure 10-15. The self-attention mechanism within the SAGAN model (Zhang et al., 2018)

Working (from the SAGAN paper):

- Feature maps x pass through three **1 × 1 convs**: f(x), g(x), h(x) (like query, key, value).

$$\beta_{j,i} = softmax(\, f(x_i)^T \, g(x_j) \,)$$

$$o_j = \Sigma_i \, \beta_{j,i} \, h(x_i) \,, \quad y = \gamma \cdot o + x$$

Self-Attention GAN (SAGAN)



Figure 10-16. A SAGAN-generated bird (left) and attention maps of the final attention-based generator layer for the pixels at the three coloured dots (Zhang et al., 2018)

- **Red dot** - part of the **bird's body** → attention falls on the **surrounding body cells**.
- **Green dot** - part of the **background** → attention falls on the **other side of the bird's head**, on other background pixels.
- **Blue dot** - part of the bird's **long tail** → attention falls on **other tail pixels**, some of them **far away**.
- Such **long-range dependencies** are hard to maintain without attention, especially for **long, thin structures** like the tail.

BigGAN

- **BigGAN**, developed at **DeepMind**, **extends the ideas of SAGAN**.
- Figure 10-17 shows BigGAN images trained on the **ImageNet** dataset at **128 × 128** resolution.



Figure 10-17. Examples of images generated by BigGAN (Brock et al., 2018)

Truncation Trick

- During **training**, z is sampled from **z ~ N(0, I)**. During **sampling**, a **truncated normal distribution** is used - values of z with **magnitude greater than a threshold are resampled**.

BigGAN



Figure 10-18. The truncation trick: from left to right, the threshold is set to 2, 1, 0.5 and 0.04 (Brock et al., 2018)

- **Smaller threshold → more believable** samples, but **less variety** (see the dogs become almost identical at 0.04).
- **Bigger:** batch size **2,048** (8× larger than SAGAN's **256**); channel size increased by **50%** in each layer.
- **Structural improvements over SAGAN:**
  - **Shared embedding**, **orthogonal regularization**,
  - Latent vector **z** fed into **each layer** of the generator, not just the initial layer.
- A tutorial for generating images with a **pre-trained BigGAN** is available on the **TensorFlow** website.

VQ-GAN

- **Vector Quantized GAN (VQ-GAN)** was introduced in **2020**.
- Builds on the 2017 paper “Neural Discrete Representation Learning” - the representations learned by a **VAE can be discrete** rather than continuous.

VQ-VAE (Vector Quantized VAE)

- Generates **high-quality images** while avoiding issues of continuous-latent VAEs, such as **posterior collapse** (latent space becomes **uninformative** due to an **overly powerful decoder**).
- The first version of **DALL.E** (OpenAI, 2021) used a VAE with a **discrete latent space**, similar to VQ-VAE.

Discrete latent space = Codebook

- A **learned list of vectors** (the **codebook**), each with an **index**.
- **Easy analogy:** Like a **box of crayons** - instead of any colour, each grid square must be coloured with one of the **fixed crayons** in the box.

VQ-GAN - VQ-VAE

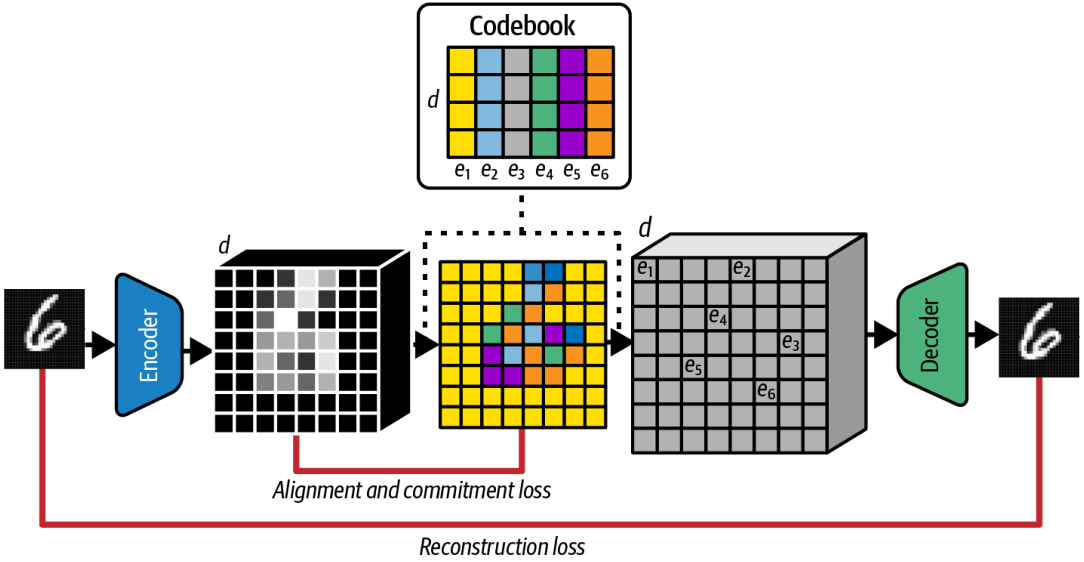


Figure 10-19. A diagram of a VQ-VAE

- **Encoder** collapses the input image into a **smaller grid of vectors**.
- Each grid vector is compared with the **codebook**; the **closest codebook vector** (Euclidean distance) is passed to the **decoder**.
- Codebook vectors have length **d** = number of channels of encoder output / decoder input.
- e.g. $\mathbf{e_1}$ may mean **background**.

$$z_q = e_k, \quad where \quad k = argmin_j \| z_e(x) - e_j \|_2$$

- The codebook is a set of **learned discrete concepts shared by encoder and decoder** to describe the contents of an image.

VQ-GAN - VQ-VAE

Loss Function of VQ-VAE

- **Reconstruction loss** + two terms - **alignment loss** and **commitment loss** - which keep encoder outputs **close to codebook vectors**.
- These terms **replace the KL divergence** term of a typical VAE.

$$L = \| x - \hat{x} \|^2 + \| sg[z_e(x)] - e \|^2 + \beta \| z_e(x) - sg[e] \|^2$$

(sg = stop-gradient; 2nd term = alignment (codebook) loss, 3rd term = commitment loss)

How to sample new images?

- A **uniform prior** (each code equally likely) will **not work**.

Example (MNIST): The **top-left grid square** is very likely to be **background**, whereas squares near the **centre** are not.

- Solution: an **autoregressive PixelCNN** (Chapter 5) **predicts the next code vector** in the grid, given the previous codes.
- The **prior is learned** by the model, rather than static as in a vanilla VAE.

VQ-GAN

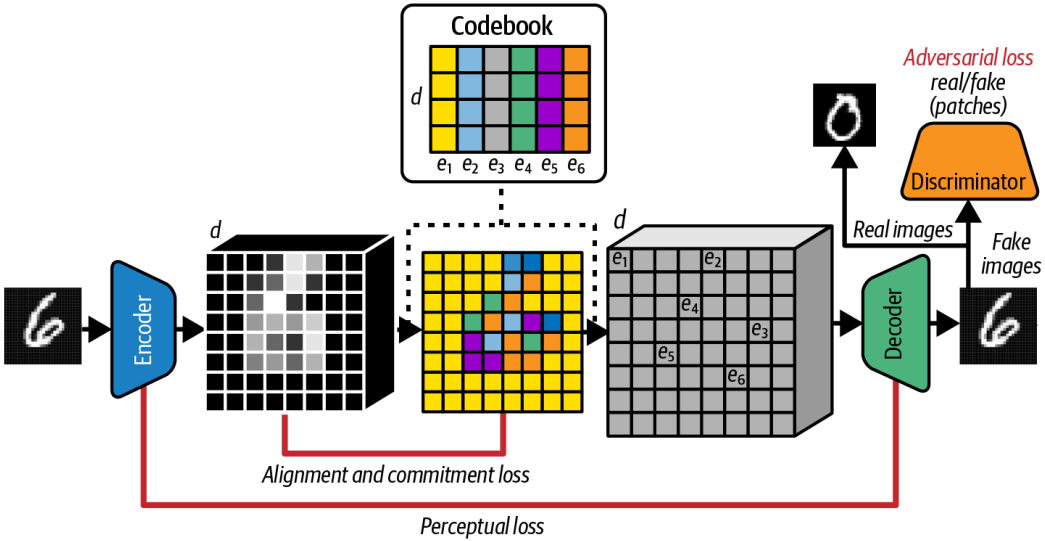


Figure 10-20. A VQ-GAN: the GAN discriminator encourages the VAE to generate less blurry images through an additional adversarial loss term

- PatchGAN was used in **pix2pix** (Isola et al., 2016) and **CycleGAN**. It judges **style** rather than content, pushing the decoder to produce **sharper** images.

Key changes to VQ-VAE

1. GAN discriminator added

- Distinguishes **VAE decoder output** from **real images** with an **adversarial loss** term.
- GANs produce **sharper** images than VAEs → better quality.
- The **VAE is still present** - the discriminator is an **additional** component (idea of **VAE-GAN**, Larsen et al., 2015).

2. PatchGAN discriminator

- Predicts if **small patches** are real or fake - outputs a **prediction vector** (one per patch).

VQ-GAN

3. Perceptual loss

- Instead of a simple **MSE reconstruction loss** on pixels, VQ-GAN uses a **perceptual loss**: the difference between **feature maps at intermediate layers** of the encoder and corresponding layers of the decoder (Hou et al., 2016).
- Produces **more realistic** images.

4. Transformer as the autoregressive model

- Instead of PixelCNN, a **Transformer** generates **sequences of codes**.
- Trained in a **separate phase**, after the VQ-GAN is fully trained.
- Uses only tokens within a **sliding window** around the token to be predicted - so the model **scales to larger images** (larger latent grids, more tokens).

$$\textit{Total loss} \approx \textit{Perceptual reconstruction} + \textit{Alignment} + \textit{Commitment} + \lambda \cdot \textit{Adversarial (PatchGAN)}$$

Easy understanding: VQ-GAN = **VQ-VAE** (compress image into codes) + **GAN** (make it sharp) + **Transformer** (write new code sequences).

ViT VQ-GAN

- Extension by **Yu et al. (2021)** - "Vector-Quantized Image Modeling with Improved VQGAN".
- The **convolutional encoder and decoder** of VQ-GAN are **replaced with Transformers**.

Vision Transformer (ViT)

- Applies the Transformer (designed for NLP) to **image data**.
- Instead of conv layers, the image is divided into a **sequence of patches**, which are **tokenized** and fed to an **encoder Transformer**.

Example: A 256 × 256 image with 8 × 8 patches → 32 × 32 = **1,024 patch tokens** - like 1,024 "words".

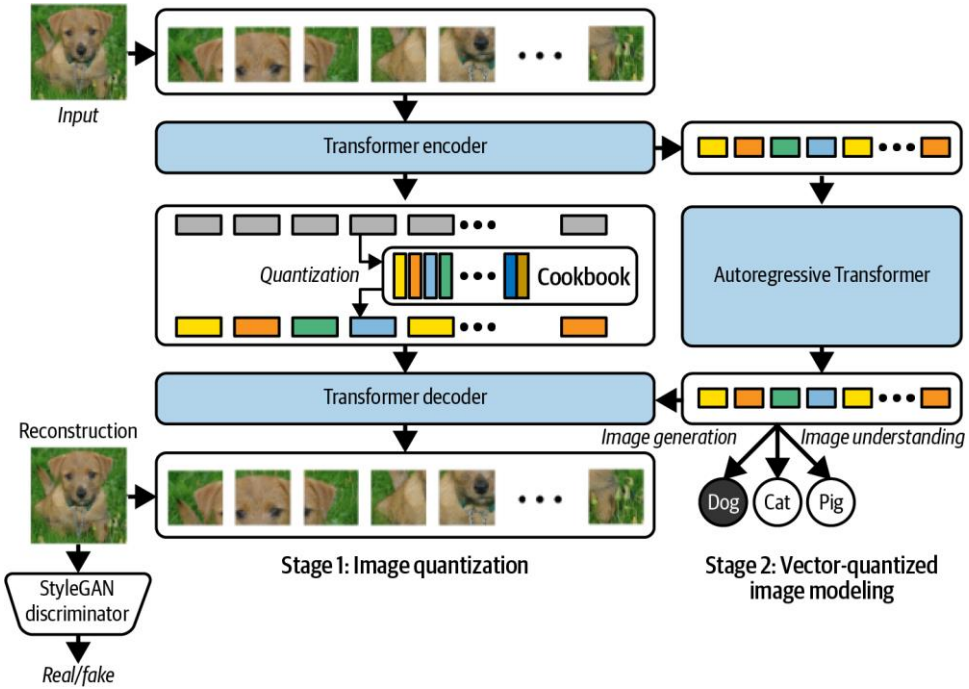


Figure 10-21. A ViT VQ-GAN (Yu and Koh, 2022)

ViT VQ-GAN

Working of ViT VQ-GAN

1. **Non-overlapping 8 × 8 patches** are **flattened** and **projected** into a low-dimensional embedding space; **positional embeddings** are added.
2. Sequence fed to a standard **encoder Transformer**; resulting embeddings are **quantized** using a **learned codebook**.
3. Integer codes are processed by a **decoder Transformer**; output patches are **stitched** together to form the image. Trained **end-to-end** as an **autoencoder**.
4. Second phase: an **autoregressive decoder Transformer** generates **sequences of codes**.

- In total: **three Transformers** + **GAN discriminator** + **learned codebook**.



Figure 10-22. Example images generated by a ViT VQ-GAN trained on ImageNet (Yu et al., 2021)

Summary - Advanced GANs

Model	Year / Lab	Key contribution
ProGAN	2017, NVIDIA	Progressive training (4 × 4 → 1024 × 1024), minibatch std, equalized learning rate, pixelwise normalization
StyleGAN	2018, NVIDIA	Mapping network + synthesis network, AdaIN, style mixing, stochastic variation
StyleGAN2	2019, NVIDIA	Weight modulation/demodulation, path length regularization, no progressive growing
SAGAN	2018	Self-attention in GANs → long-range dependencies
BigGAN	2018, DeepMind	Bigger SAGAN, truncation trick, shared embedding, orthogonal regularization
VQ-GAN	2020	VQ-VAE codebook + PatchGAN discriminator + perceptual loss + Transformer prior
ViT VQ-GAN	2021, Google	Vision Transformer encoder/decoder instead of convolutions

Summary - Advanced GANs

- **ProGAN (2017)** pioneered **progressive training**.
- **StyleGAN (2018)** gave greater **control over the image output** - the **mapping network** creates a style vector and the **synthesis network** injects the style at **different resolutions**.
- **StyleGAN2** replaced **AdaIN** with **weight modulation and demodulation**, added **path length regularization**, and kept gradual resolution refinement **without progressive training**.
- **SAGAN (2018)** built **attention** into a GAN - capturing **long-range dependencies** (e.g., similar background colours on opposite sides of an image) without deep convolutional maps.
- **BigGAN** extended this idea and trained a **larger network** for better image quality.
- **VQ-GAN** combines several generative models: a **VAE with a discrete latent space**, a **discriminator** for less blurry images, and an **autoregressive Transformer** to create new code sequences.
- **ViT VQ-GAN** replaces the convolutional encoder and decoder of VQ-GAN with **Transformers**.