Most Used Gen AI & LLM Algorithms With Use case & Code

Type	@datasciencebrain

Part 1: Large Language Models (LLMs)

1. GPT-4.5 (Orion)

Developer: OpenAI

Type: Decoder-only transformer

Capabilities: Text generation, reasoning, code generation, conversation, multimodal tasks

Description

GPT-4.5 (internally called "Orion") is a large-scale transformer model from OpenAI, succeeding GPT-4. It significantly improves contextual understanding, mathematical reasoning, code generation, and memory handling. GPT-4.5 is often accessible through OpenAI's GPT-4-turbo interface.

When to Use

- Complex reasoning and problem solving
- High-accuracy coding assistants
- Multi-turn conversational AI agents
- Processing large context windows (100k+ tokens)
- Enterprise AI workflows and intelligent document processing

When Not to Use

- On-device, real-time applications (due to high latency and size)
- Cost-sensitive, high-frequency queries where a smaller model suffices
- Tasks where absolute interpretability and transparency are required

Best Practices

- Always use system prompts to guide the behavior
- Use function-calling or tool-use interfaces for automation
- Avoid giving ambiguous or open-ended tasks without instructions
- Combine with embeddings for retrieval-augmented generation (RAG)

Sample Code (OpenAI API)

```
import openai

openai.api_key = "your_api_key"

response = openai.ChatCompletion.create(
    model="gpt-4-1106-preview",  # GPT-4-turbo alias
    messages=[
        {"role": "system", "content": "You are an expert in AI."},
        {"role": "user", "content": "Explain the advantages of transformers."}
    ]
)

print(response['choices'][0]['message']['content'])
```

---

2. GPT-3 / GPT-3.5

Developer: OpenAI

Type: Decoder-only transformer

Capabilities: Text generation, Q&A, summarization, dialogue

Description

GPT-3 and GPT-3.5 are foundational language models from OpenAI. While GPT-3 introduced large-scale language generation, GPT-3.5 improved reasoning and performance in complex tasks.

When to Use

- Conversational agents with moderate complexity
- Content summarization, rephrasing, auto-completion
- Knowledge base Q&A
- Code generation (Codex variant)

When Not to Use

- Factual or sensitive queries (hallucinations possible)
- Long-context tasks (limited to 4k or 16k tokens)
- Tasks requiring real-time reasoning or tools

Best Practices

- Specify roles and tasks clearly in prompts
- Use temperature and top_p to control creativity
- Use embeddings for semantic search

---

3. BERT (Bidirectional Encoder Representations from Transformers)

Developer: Google AI

Type: Encoder-only transformer

Pre-training Tasks: Masked Language Modeling (MLM), Next Sentence Prediction (NSP)

Description

BERT is a deep bidirectional model designed for understanding the context of words in a sentence. It is not a generative model but excels in understanding-based tasks.

When to Use

- Text classification (sentiment, spam detection)
- Named Entity Recognition (NER)
- Sentence similarity and semantic search
- Question answering (with context)

When Not to Use

- Text generation tasks
- Sequence-to-sequence tasks (translation, summarization)

Best Practices

- Use pre-trained checkpoints like `bert-base-uncased`
- Fine-tune with task-specific labeled data
- Truncate or pad sequences for batch input

Sample Code (HuggingFace Transformers)

```
from transformers import BertTokenizer, BertForSequenceClassification
import torch

tokenizer = BertTokenizer.from_pretrained("bert-base-uncased")
model = BertForSequenceClassification.from_pretrained("bert-base-uncase
d")

inputs = tokenizer("Is this a good investment?", return_tensors="pt")
outputs = model(**inputs)
```

```
prediction = torch.argmax(outputs.logits)
print(prediction)
```

---

4. T5 (Text-to-Text Transfer Transformer)

Developer: Google

Type: Encoder-Decoder transformer

Pre-training: Text-to-text tasks

Description

T5 reframes every NLP task as a text-to-text problem. For example, classification becomes “input text: ... classify sentiment” → “positive”.

When to Use

- Multi-task NLP pipelines
- Summarization, translation, Q&A
- Generative NLP tasks using a unified framework

When Not to Use

- Real-time or low-resource environments (T5 is computationally expensive)
- When training data is not available in text-to-text format

Best Practices

- Use prompt templates to standardize input
- Train or fine-tune on your task as text-in, text-out
- Choose appropriate size: T5-small, T5-base, T5-large

Sample Code (HuggingFace)

```
from transformers import T5Tokenizer, T5ForConditionalGeneration

model = T5ForConditionalGeneration.from_pretrained("t5-base")
```

```
tokenizer = T5Tokenizer.from_pretrained("t5-base")

input_text = "summarize: Machine learning is a subset of AI..."
input_ids = tokenizer.encode(input_text, return_tensors="pt")

summary_ids = model.generate(input_ids, max_length=50)
print(tokenizer.decode(summary_ids[0], skip_special_tokens=True))
```

---

5. RoBERTa (Robustly Optimized BERT Approach)

Developer: Facebook AI

Type: Encoder-only transformer

Pre-training Improvements: Trained longer, no NSP, more data

When to Use

- Tasks similar to BERT with improved accuracy
- Sentence classification, semantic similarity, NER
- Zero-shot and few-shot learning with classification heads

When Not to Use

- Autoregressive tasks or generation
- Low-latency environments (heavier than BERT)

Best Practices

- Fine-tune on large batches
- Use `roberta-base` or `roberta-large` based on resource availability

---

6. ALBERT (A Lite BERT)

Developer: Google

Type: Encoder-only

Efficiency Improvements: Factorized embedding, parameter sharing

When to Use

- NLP tasks on resource-constrained systems
- Question answering with fast inference
- Long document classification with fewer parameters

When Not to Use

- Tasks where large-scale fine-tuning is needed
- Generation-based NLP

---

7. XLNet

Developer: Google/CMU

Type: Permutation-based autoregressive transformer

Description

XLNet improves upon BERT by capturing bidirectional context without masking. It outperforms BERT on several benchmarks.

When to Use

- Text classification, sentiment analysis
- Question answering with better accuracy than BERT
- When pre-training data order matters

When Not to Use

- Generative tasks where simpler models suffice
- Low-resource environments

---

8. PaLM (Pathways Language Model)

Developer: Google

Type: Decoder-only transformer

Scale: Up to 540B parameters

When to Use

- High-scale LLM applications (enterprise, scientific)
- Multilingual generation
- Long-context reasoning

When Not to Use

- Lightweight or edge computing use cases

---

9. Gemini (formerly Bard, by DeepMind)

Developer: Google DeepMind

Type: Multimodal large model

Description

Gemini integrates text, image, and audio input and reasoning. It’s designed for multi-modal interaction and scientific applications.

When to Use

- Multimodal assistants (image + text)
- Scientific or multi-modal document analysis
- Chatbots with image context

When Not to Use

- Pure text-based low-resource tasks

---

10. Codex

Developer: OpenAI

Type: GPT-3 fine-tuned on code

Application: Code generation, explanation

When to Use

- IDE assistants (e.g., GitHub Copilot)
- Code synthesis from natural language
- Debugging and code refactoring

When Not to Use

- Long-form NLP tasks
- When full project-level understanding is required

Sample Code

```
import openai

openai.api_key = "your_api_key"

response = openai.Completion.create(
    model="code-davinci-002",
    prompt="def fibonacci(n):",
    max_tokens=100
)

print(response["choices"][0]["text"])
```

---

11. BLOOM

Developer: BigScience

Type: Multilingual autoregressive transformer

License: Open-source

When to Use

- Translation and text generation in 40+ languages

- Open research and customization
- Multilingual chatbots

When Not to Use

- Production-scale deployment with limited compute
- Tasks needing fine-grained tuning for English-only data

Part 2: Generative AI Algorithms

1. Generative Adversarial Networks (GANs)

Invented By: Ian Goodfellow

Architecture: Generator + Discriminator (adversarial training)

Description

GANs consist of two neural networks: a **generator** that creates synthetic data, and a **discriminator** that evaluates the authenticity of that data. Training is adversarial — the generator tries to fool the discriminator, while the discriminator tries to detect fakes.

When to Use

- Image generation (e.g., human faces, art, fashion)
- Data augmentation (e.g., rare class data synthesis)
- Super-resolution tasks
- Style transfer (e.g., pix2pix, CycleGAN)

When Not to Use

- When training stability is critical (GANs are hard to train)
- Structured text or tabular data generation
- Tasks requiring controllable outputs

Best Practices

- Use conditional GANs (cGAN) for class-specific generation
- Monitor mode collapse (where the generator produces limited variety)
- Use techniques like WGAN-GP for stable training

Sample Code (PyTorch - Simple GAN)

```
import torch
import torch.nn as nn

class Generator(nn.Module):
    def __init__(self, latent_dim):
        super().__init__()
        self.model = nn.Sequential(
            nn.Linear(latent_dim, 128),
            nn.ReLU(),
            nn.Linear(128, 784),
            nn.Tanh()
        )

    def forward(self, z):
        return self.model(z)

class Discriminator(nn.Module):
    def __init__(self):
        super().__init__()
        self.model = nn.Sequential(
            nn.Linear(784, 128),
            nn.LeakyReLU(0.2),
            nn.Linear(128, 1),
            nn.Sigmoid()
        )
```

```
    def forward(self, x):
        return self.model(x)
```

2. Variational Autoencoders (VAEs)

Architecture: Encoder + Decoder with latent space sampling

Goal: Learn latent representation and generate new samples from it

Description

VAEs are probabilistic models that learn the distribution of the data and generate samples from a latent vector space. Unlike GANs, they are based on likelihood maximization.

When to Use

- Image reconstruction with smooth latent space
- Anomaly detection (via reconstruction error)
- Synthetic data generation where distribution matters
- When you want structured, interpretable latent variables

When Not to Use

- High-fidelity image generation
- Tasks requiring fine details or realistic textures

Best Practices

- Use KL divergence loss to regularize the latent space
- Choose appropriate latent dimension size
- Normalize inputs and use batch normalization

Sample Code (PyTorch - VAE)

```
class VAE(nn.Module):
    def __init__(self):
        super().__init__()
        self.encoder = nn.Sequential(nn.Linear(784, 256), nn.ReLU())
        self.fc_mu = nn.Linear(256, 20)
        self.fc_logvar = nn.Linear(256, 20)
        self.decoder = nn.Sequential(nn.Linear(20, 256), nn.ReLU(), nn.Linear(25
6, 784), nn.Sigmoid())

    def reparameterize(self, mu, logvar):
        std = torch.exp(0.5 * logvar)
        eps = torch.randn_like(std)
        return mu + eps * std

    def forward(self, x):
        h = self.encoder(x)
        mu, logvar = self.fc_mu(h), self.fc_logvar(h)
        z = self.reparameterize(mu, logvar)
        return self.decoder(z), mu, logvar
```

3. DALL·E

Developer: OpenAI

Type: Transformer-based text-to-image model

Description

DALL·E generates realistic images and art from natural language descriptions. It uses a VQ-VAE representation and autoregressive decoding or diffusion-based decoding depending on version.

When to Use

- Text-to-image generation
- Creative artwork or illustration based on prompts

- Generating visuals for content or games

When Not to Use

- When precise object layout is needed
- Scientific or medical image generation without constraints

Best Practices

- Use prompt engineering to guide image quality
- Provide detailed and stylistic descriptions
- Use constraints like aspect ratios if supported

---

4. CLIP (Contrastive Language–Image Pretraining)

Developer: OpenAI

Purpose: Image-text alignment, zero-shot classification

Description

CLIP learns to match images and texts via contrastive learning. It maps both modalities into a shared embedding space.

When to Use

- Image classification using text descriptions (zero-shot)
- Image-to-text or text-to-image retrieval
- Vision-language search and filtering

When Not to Use

- Tasks requiring pixel-level understanding
- Detailed image generation (CLIP is not generative)

Best Practices

- Use it in combination with image generators for prompt filtering

- Normalize and preprocess input images as expected

Sample Code (HuggingFace CLIP)

```
from transformers import CLIPProcessor, CLIPModel
from PIL import Image

model = CLIPModel.from_pretrained("openai/clip-vit-base-patch32")
processor = CLIPProcessor.from_pretrained("openai/clip-vit-base-patch32")

image = Image.open("cat.png")
inputs = processor(text=["a cat", "a dog"], images=image, return_tensors="p
t", padding=True)
outputs = model(**inputs)
logits_per_image = outputs.logits_per_image
print(logits_per_image.softmax(dim=1))  # probability per label
```

---

5. Stable Diffusion

Developer: Stability AI

Type: Latent diffusion model (LDM)

Description

Stable Diffusion generates images by learning to denoise latent representations rather than pixels. It is efficient and open-source, enabling local deployments.

When to Use

- Text-to-image generation on personal GPUs
- Custom image synthesis
- Style-specific artistic rendering

When Not to Use

- Real-time applications with tight latency constraints

- Generation requiring 3D consistency

Best Practices

- Use prompt weighting and attention tokens for precision
- Combine with CLIP for ranking
- Fine-tune via DreamBooth or LoRA for custom styles

---

6. Text-to-Image Models (e.g., DALL·E 2, MidJourney)

Description

These models take textual prompts and generate visual content. They often use diffusion or transformer architectures.

When to Use

- Visual storytelling, conceptual design
- Game or book illustrations
- Branding or advertisement visuals

When Not to Use

- Domain-specific or scientific image generation
- When control over structure or layout is essential

---

7. Neural Style Transfer

Description: A technique to blend the content of one image with the style of another using convolutional neural networks.

When to Use

- Artistic image generation
- Visual effect pipelines

When Not to Use

- Object detection or semantic reasoning tasks
- High-resolution rendering without optimization

Sample Code (PyTorch)

```
from torchvision import models
import torch.nn.functional as F

# Load VGG19 layers and extract features
vgg = models.vgg19(pretrained=True).features.eval()
# Input: content and style images → output: stylized image
```

---

8. DeepDream

Developer: Google

Description: Enhances image features by maximizing activation of neural network layers

When to Use

- Creating surreal, psychedelic art
- Visualizing neural network activations

When Not to Use

- Structured or realistic image generation
- Professional visual design tasks

---

9. Transformers for Image Generation

Examples: ImageGPT, ViT for generative tasks

Description

Vision Transformers (ViTs) adapted to generation predict image patches or tokens. Often used with hybrid methods (e.g., GAN + ViT, Diffusion + ViT).

When to Use

- Generative image modeling
- Learning from image + text datasets

When Not to Use

- Very high-resolution image generation
- Real-time inference without optimized hardware

---

10. Diffusion Models

Examples: DDPM, Latent Diffusion (LDM), Imagen

Description: Generate images by denoising random noise through iterative steps.

When to Use

- High-quality image synthesis
- Tasks requiring photorealism or precise guidance (e.g., inpainting)

When Not to Use

- Low-latency real-time generation
- Tasks requiring conditional logic (use alongside classifiers or CLIP)

Best Practices

- Choose denoising steps and schedule carefully
- Use pretrained models like Stable Diffusion for fast experimentation
- Combine with control nets (e.g., ControlNet) for image conditioning

---