APIs & Frameworks d'Intégration

Outils Modernes pour Applications IA

Cours d'Introduction à l'IA

Partie 1

Écosystème des Outils

Écosystème d'Intégration IA

Modèles

LLMs, Vision, Audio

Orchestration

LangChain, LangGraph

Interfaces

Gradio, Streamlit

Modèles Locaux vs APIs Cloud

Comprendre les deux approches d'intégration de l'intelligence artificielle : héberger ses modèles en local ou exploiter des APIs cloud. Chaque approche offre des avantages spécifiques selon les contraintes de coût, de performance, de confidentialité et de personnalisation.

🧠 Modèles Locaux

- ✔ **Contrôle total** sur les données et le modèle
- ✔ **Personnalisation** complète (fine-tuning, quantization)
- ⚠ **Infrastructure requise** : GPU, maintenance, électricité
- 💡 **Idéal pour** : données sensibles, projets R&D, IA embarquée

☁ APIs Cloud

- ✔ **Déploiement instantané** sans infrastructure locale
- ✔ **Scalabilité automatique** gérée par le fournisseur
- ⚠ **Coût à l'usage** (par requête ou par token)
- 💡 **Idéal pour** : prototypes rapides, start-ups, applications grand public

🔍 Critères de Décision Clés

💰 Coût

Local : Investissement initial (GPU)
Cloud : Paiement à l'usage

⚡ Performance

Local : Latence minimale locale
Cloud : Puissance Cloud, dépend réseau

🔒 Confidentialité

Local : Aucune donnée sort du réseau
Cloud : Données transitent par API

🎯 Personnalisation

Local : Fine-tuning complet
Cloud : Prompt engineering uniquement

Partie 2

Hugging Face Transformers

Qu'est-ce que Hugging Face?

Plateforme leader pour le Machine Learning

- Hub de 500,000+ modèles pré-entraînés
- Bibliothèque transformers pour NLP
- Support multi-modalités (texte, vision, audio)
- Communauté open-source active

Hugging Face

Hugging Face Hub

Découverte de modèles

Recherche par tâche, langue, popularité

Model Cards

Documentation complète

Métriques

Downloads, likes, performances

Datasets

Jeux de données associés

Concept de Pipeline

Gère automatiquement preprocessing, model et postprocessing

Pipelines Disponibles

📝 Texte

sentiment-analysis, text-generation, summarization

🖼️ Vision

image-classification, object-detection

🎵 Audio

speech-recognition, text-to-speech

🔀 Multimodal

visual-question-answering

Exemple: Pipeline Simple

```
from transformers import pipeline

classifier = pipeline("sentiment-analysis")

result = classifier("Ce cours est excellent!")

print(result)
# [{'label': 'POSITIVE', 'score': 0.9998}]
```

Spécifier un Modèle

```
classifier = pipeline(
"sentiment-analysis",
model="nlptown/bert-base-multilingual"
)

generator = pipeline(
"text-generation",
model="gpt2"
)
```

Contrôler l'Inférence

```
generator = pipeline(
"text-generation",
model="gpt2"
)

result = generator(
"L'IA va",
max_length=50,
temperature=0.7,
num_return_sequences=1
)
```

max_length

Longueur maximale

temperature

Créativité (0.1-1.5)

top_k / top_p

Échantillonnage

Chargement Manuel

```
from transformers import AutoTokenizer, AutoModel

tokenizer = AutoTokenizer.from_pretrained("bert-base")
model = AutoModel.from_pretrained("bert-base")

inputs = tokenizer("Hello", return_tensors="pt")
outputs = model(**inputs)
```

Optimisation & Production

Quantization

Réduire la mémoire (8-bit, 4-bit)

Batching

Traiter plusieurs inputs

Caching

Réutiliser les calculs

ONNX

Export cross-platform

Cas d'Usage Hugging Face

Analyse de texte

Sentiment, classification, NER

Génération

Texte, résumés, traduction

Vision

Classification, détection, segmentation

Audio

Transcription, synthèse vocale

🎯 **Pour vos projets**

Prototypage rapide avec modèles pré-entraînés

Points Clés Hugging Face

1. Hub avec 500k+ modèles open-source
2. Pipelines abstraient la complexité
3. Support multi-modalités (texte, vision, audio)
4. Contrôle fin avec chargement manuel
5. Optimisations pour la production

Partie 3

LangChain

Orchestration d'Applications LLM

Qu'est-ce que LangChain?

Framework pour développer des applications avec LLMs

- Composition de composants modulaires
- Chaînage d'opérations
- Gestion du contexte et mémoire
- Intégration multi-modèles
- Agents autonomes

LangChain

Build context-aware apps

Pourquoi LangChain?

❌ Sans LangChain

- Gérer manuellement les prompts
- Parser les réponses LLM
- Implémenter la mémoire
- Intégrer les sources de données
- Orchestrer plusieurs appels
- Gérer les erreurs et retries

✅ Avec LangChain

- Templates de prompts
- Parsers intégrés
- Mémoire prête à l'emploi
- Connecteurs de données
- Chains et agents
- Gestion robuste des erreurs

Architecture LangChain

Models

Prompts

Chains

Memory

Tools

Agents

Models dans LangChain

```
from langchain_openai import ChatOpenAI

llm = ChatOpenAI(model="gpt-3.5-turbo")

response = llm.invoke("Explique l'IA")
```

Prompts Templates

```
from langchain.prompts import ChatPromptTemplate

template = ChatPromptTemplate.from_messages([
("system", "Tu es un {role}."),
("user", "{question}")
])

prompt = template.invoke({
"role": "prof", "question": "C'est quoi un LLM?"
})
```

Chains

```
chain = template | llm

result = chain.invoke({
"role": "data scientist",
"question": "Qu'est-ce que le RAG?"
})
```

RAG: Retrieval Augmented Generation

Le cas d'usage star de LangChain

Enrichir les LLMs avec vos documents

Documents

→

Recherche

→

LLM

Workflow RAG

1. Charger et découper les documents
2. Créer embeddings vectoriels
3. Stocker dans vector store
4. Rechercher contexte pertinent
5. Générer réponse avec LLM

RAG: Code

```
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain_community.vectorstores import FAISS
from langchain.chains import RetrievalQA

splitter = RecursiveCharacterTextSplitter(chunk_size=500)
splits = splitter.split_documents(docs)

vectorstore = FAISS.from_documents(splits, embeddings)

qa = RetrievalQA.from_chain_type(llm, retriever=vectorstore.as_retriever())
```

Memory

```
from langchain.memory import
ConversationBufferMemory

memory = ConversationBufferMemory()

memory.save_context(
{"input": "Bonjour"},
{"output": "Salut!"}
)
```

Types

Buffer, Summary, Vector

Composant: Agents

LLMs qui décident quoi faire

Prise de décision autonome avec outils

Raisonnement

Quel outil utiliser?

Outils

Calculatrice, recherche web

Itération

Jusqu'à solution

Points Clés LangChain

1. Framework d'orchestration pour LLMs
2. Composants modulaires et réutilisables
3. RAG: cas d'usage le plus populaire
4. Memory pour applications conversationnelles
5. Agents pour décisions autonomes

Partie 4

LangGraph

Workflows Complexes

Qu'est-ce que LangGraph?

Extension de LangChain pour workflows avec états

- Graphs avec cycles et branches
- Gestion d'états complexes
- Workflows conditionnels
- Multi-agent systems
- Human-in-the-loop

LangGraph

Build stateful graphs

Pourquoi LangGraph?

LangChain

Flux linéaires, pas de cycles

LangGraph

Cycles, branches, état

Concepts Fondamentaux

Graph

Structure du workflow

Nodes

Étapes de traitement

Edges

Transitions

State

Données partagées

Exemple LangGraph

```
from langgraph.graph import StateGraph

class State(TypedDict):
input: str
result: dict

def analyze(state): return {"result": {...}}

workflow = StateGraph(State)
workflow.add_node("analyze", analyze)
```

Edges Conditionnelles

```
def should_continue(state):
if state["validated"]:
return "success"
return "retry"

workflow.add_conditional_edges("validate", should_continue)
```

Cas d'Usage LangGraph

🤖 Multi-Agent Systems
Coordination d'agents

🔄 Workflows itératifs
Révision et amélioration

👤 Human-in-the-loop
Validation humaine

⚖️ Décisions complexes
Logique métier avancée

💡 **Quand l'utiliser:** Workflows avec logique conditionnelle complexe

LangChain vs LangGraph

LangChain

- ✅ Prototypage rapide
- ✅ RAG simple
- ✅ Chains linéaires
- ❌ Pas de cycles
- ❌ État limité

LangGraph

- ✅ Workflows complexes
- ✅ Multi-agents
- ✅ Cycles et boucles
- ✅ État persistant
- ⚠️ Plus complexe

Partie 5

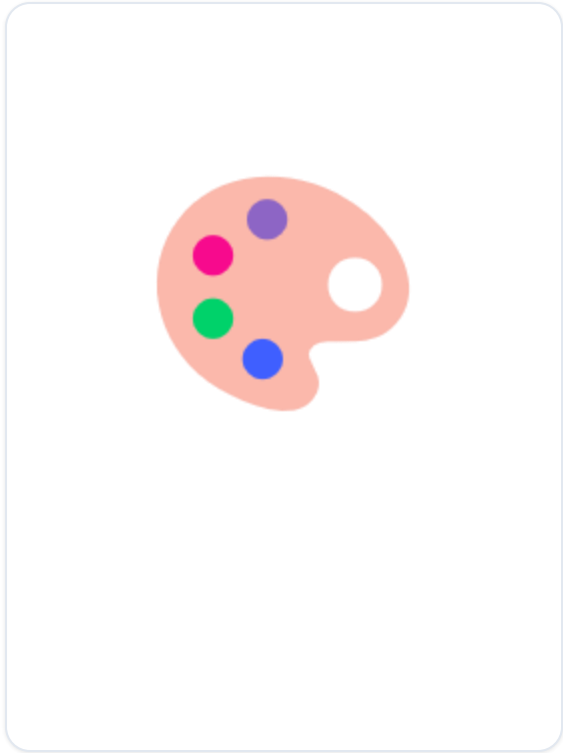
Interfaces Visuelles

Gradio & Streamlit

Gradio

Créer des interfaces en minutes

- UI automatique pour modèles ML
- Prototypage ultra-rapide
- Partage facile (lien public)
- Intégration avec Hugging Face
- Composants interactifs



Gradio: Exemple

```
import gradio as gr
from transformers import pipeline

classifier = pipeline("sentiment-analysis")

def analyze(text):
return classifier(text)[0]['label']

gr.Interface(fn=analyze, inputs="text", outputs="text").launch()
```

Composants Gradio

📝 Inputs

- Textbox, Number, Slider
- Image, Audio, Video
- File, Dataframe
- Checkbox, Radio, Dropdown

📤 Outputs

- Label, Textbox, JSON
- Image, Video, Audio
- Plot, Dataframe

🎯 Features

- Exemples prédéfinis
- Mode batch
- Temps réel
- Thèmes personnalisés

Gradio Blocks

```
with gr.Blocks() as demo:
gr.Markdown("# Analyseur")
with gr.Row():
input_text = gr.Textbox()
output = gr.Label()
btn = gr.Button("Analyser")
btn.click(analyze, input_text, output)
demo.launch()
```

Streamlit

Applications data complètes

- Framework Python pur
- Apps data interactives
- Widgets riches
- Déploiement cloud gratuit
- Communauté active

Streamlit: Exemple

```
import streamlit as st
from transformers import pipeline

st.title("Analyseur")

classifier = pipeline("sentiment-analysis")

text = st.text_area("Texte")
if st.button("Analyser"):
st.metric("Résultat", classifier(text)[0]['label'])
```

Fonctionnalités Streamlit

Visualisation

Charts, graphs, maps

Widgets

Sliders, buttons, selects

Caching

Performance optimisée

Responsive

Mobile-friendly

Layout: Columns, Tabs, Expanders

Media: Images, Audio, Video

Gradio vs Streamlit

Gradio

- ✅ Très rapide à créer
- ✅ Focus modèles ML
- ✅ Partage instantané
- ✅ HuggingFace Spaces
- ⚠️ Moins flexible

Streamlit

- ✅ Apps complètes
- ✅ Visualisation riche
- ✅ Plus de contrôle
- ✅ État de session
- ⚠️ Plus de code

Comparaison

HuggingFace	Modèles, inférence
LangChain	RAG, prototypes
LangGraph	Workflows complexes
Gradio	Démos rapides
Streamlit	Apps complètes

Récapitulatif


HuggingFace


LangChain


LangGraph


Gradio


Streamlit

Merci! Questions?

Modèles Locaux vs APIs Cloud

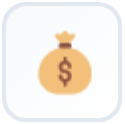
Modèles Locaux

- Contrôle total sur les données
- Pas de coûts récurrents
- Personnalisation complète
- Nécessite infrastructure GPU

APIs Cloud

- Déploiement immédiat
- Mise à l'échelle automatique
- Maintenance gérée
- Coût par utilisation

Critères de Choix



Coût

Local: GPU + électricité vs Cloud: Par token/requête

Performance

Latence réseau vs puissance matérielle

Confidentialité

Données sensibles: privilégier local

Personnalisation

Fine-tuning vs prompting