Day 4:
Model Deployment

Session I

Creating a Simple Web API for Model Inference

Motivation

We trained a well-performing model: What comes next?

Goal:

- Make the model accessible to our application(s)

Scope

What we cover today:

Model deployment for inference using a Web API

Other deployments:

- For training and evaluation
- On the edge (e.g., IoT devices, smartphones)

Outline

Session I:

- Model Serialization
- Minimal FastAPI Application

Session II:

- API Improvements: Input validation, Error handling, …
- Create a Docker Image

Session III:

- Monitoring the API

Requirements

- Cloned AI-in-Practice-UOS/course-material repository
- Python environment
- Installed required packages
- Docker installed

Model Serialization

Serialization: Process of converting an object into a format that can be transmitted or stored

For ML models:

- Save, store and distribute trained model weights
- May include architecture, optimizer states (to continue training), or the whole Python model

Serialization Methods: General-purpose

Pickle: General-purpose serialization of Python objects

- Highly flexible and simple
- Unsafe, allows arbitrary code execution
- Depends on the environment and structure
- Variants: joblib (optimizes large arrays), dill (extends to complex objects)

Pickle:

```
# save
with open('model.pkl', 'wb') as file:
    pickle.dump(model, file)

# load
with open('model.pkl', 'rb') as file:
    model = pickle.load(file)
```

Serialization Methods: Tensorflow

model.save_weights: Saves model weights only

- Requires model architecture in code

.keras: New recommended standard

- Stores architecture, weights, optimizer states
- High-level: name-based and allows debugging

HDF5 (legacy): Old format for high-level storage

SavedModel: Low-level computation graph

- Used for TFLite and TFServing

Weights only:

```
# save
model.save_weights('model.weights.h5')

# load
model = create_model()
model.load_weights('model.weights.h5')
```

.keras:

```
# save as .keras
model.save('model.keras')

# load
model = tf.keras.models.load_model('model.keras')
```

Serialization Methods: PyTorch

torch.save state_dict: Saves weights (and optimizer)

- Requires model architecture in code

torch.save model: Uses pickle internally

- Convenient, but not recommended

TorchScript: Convert model to computation graph

- Enables inference in non-Python environments
- Maintenance only: newer alternative torch.export

State dictionary:

```
# save
torch.save(model.state_dict(), "model.pth")

# load
model = create_model()
model.load_state_dict(
    torch.load("model.pth", weights_only=True))
model.eval() # for inference
```

TorchScript:

```
# export and save
model_scripted = torch.jit.script(model)
model_scripted.save("model_scripted.pth")

# load
model = torch.jit.load("model_scripted.pth")
model.eval() # for inference
```

Serialization Methods: transformers

.save_pretrained: Saves weights and configurations

Files:

- Model configuration config.json
- Model weights, uses torch.save or .save_weights
- Additional configurations, e.g., tokenizer

transformers:

```
from transformers import AutoModel, AutoTokenizer

# load from hugging face
model = AutoModel.from_pretrained("hf_model")
tokenizer = AutoTokenizer.from_pretrained("hf_model")

# save
model.save_pretrained("./model")
tokenizer.save_pretrained("./model")

# load
model = AutoModel.from_pretrained("./model")
tokenizer = AutoTokenizer.from_pretrained("./model")
```

Serialization Methods: Framework-agnostic

SafeTensors: Safe format for weights and tensors

- Focuses on security and performance
- Not for complete models, weights only
- Mostly used in Hugging Face (transformers)

ONNX: Cross-framework model portability

- Optimized for hardware accelerators
- Deployment on ONNX Runtime or TensorRT
- Requires careful conversion, does not support framework-specific features

Practice: Serialization

Task:

- Import functions from dummy_model.py to train a small TF model
- Serialize it with weights only and .keras format
- Load the model and verify with print(model.summary())
- Convert to ONNX format and save
- Explore the 3 files on https://netron.app

Hint: Examples are also in the repo

Weights only:

```
# save
model.save_weights('model.weights.h5')

# load
model = create_model()
model.load_weights('model.weights.h5')
```

.keras:

```
# save as .keras
model.save('model.keras')

# load
model = tf.keras.models.load_model('model.keras')
```

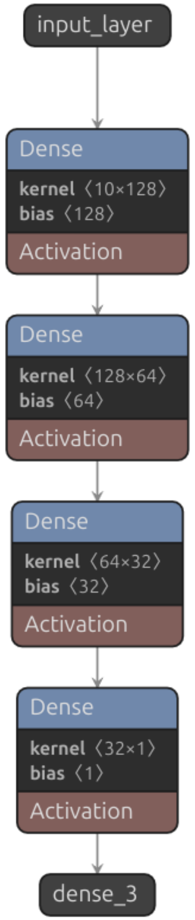
TF to ONNX:

```
inp_size = 10
spec = (tf.TensorSpec((None, inp_size), tf.float32, name="input"),)
onnx_model, _ = tf2onnx.convert.from_keras(model, input_signature=spec)
onnx.save(onnx_model, 'model.onnx')
```

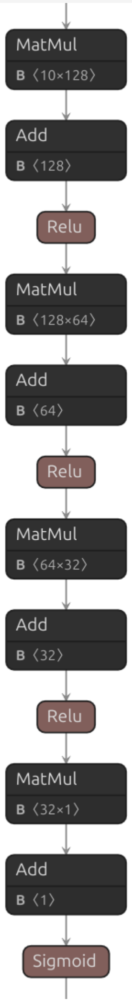
netron.app: Visualize Serialization Types

Weights only:

.keras:



ONNX:



Web APIs

Web API: Interface that allows different systems to communicate over HTTP

Purpose for ML Model Deployment:

- Enable remote model access
- Easy integration with other systems
- Scalability and modularity

Types of Web APIs

REST (Representational State Transfer):

- Simple and widely used
- Clear structure with HTTP methods and endpoints

GraphQL:

- Flexible querying via single endpoint

gRPC:

- High-performance binary communication

Example Endpoint and Request Structure

REST:

GET /books/1

No body

Response:

```json
{
    "id": 1,
    "title": "Book Title",
    "author": "Author Name"
}
```

GraphQL:

POST /graphql

```json
{
    "query": "{ book(id: 1) { title author } }"
}
```

Response:

```json
{
    "data": {
        "book": {
            "title": "Book Title",
            "author": "Author Name"
        }
    }
}
```

Anatomy of Response/Request (REST)

Request

- Endpoint: URL identifying the resource
  e.g., /books, /books/{id}
- Query Parameters: /books?author=john
- HTTP Method: Type of the operation
  GET, **POST**, **PUT/PATCH**, **DELETE**
- Headers: Metadata, e.g., authorization
- Body: JSON payload for **POST** and **PUT/PATCH**

Response

- Status Code: Status of the result
  e.g., **200 OK**, **404 NOT FOUND**
- Headers: Metadata, e.g., content type
- Body: JSON payload of the response

Introducing FastAPI

FastAPI: A modern, fast (high-performance), web framework for building APIs with Python

- Simple and intuitive, requires minimal code
- Type-based validation using Python type hints
- Automatic and built-in API documentations

Minimal FastAPI

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
async def root():
    return {"message": "Hello World"}
```

Task:
Run: fastapi dev minimal_fastapi.py
Go to http://127.0.0.1:8000

FastAPI POST Endpoint

```python
from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()

class EchoRequest(BaseModel):
    message: str

@app.post("/echo")
async def echo(request: EchoRequest):
    return {"echo": request.message}
```

Task:
Run: fastapi dev echo_fastapi.py
In a second terminal:
http POST http://127.0.0.1:8000/echo message="Hello?"

Practice: Minimal FastAPI Chat Endpoint

Task:

- In a new file, create a FastAPI app with the following Endpoint:

POST /chat → **200 OK**

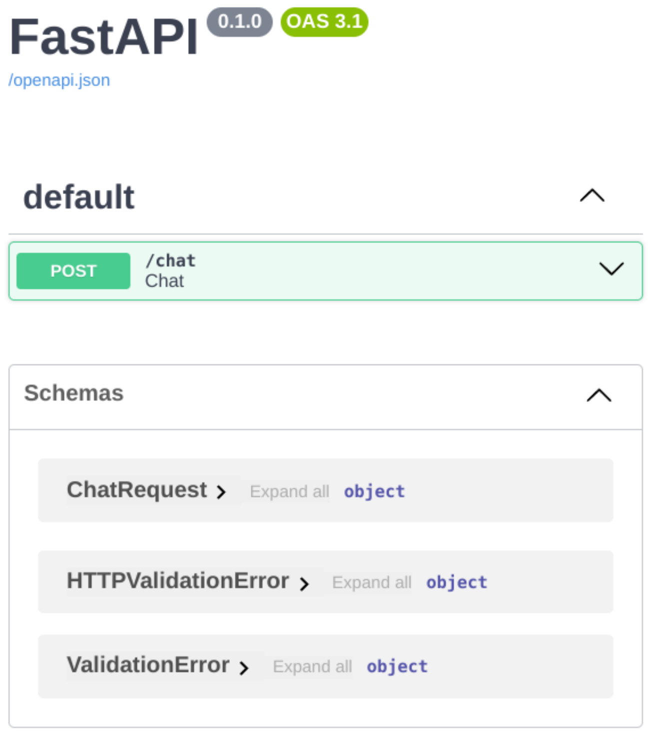
```
{
    "message": "AI is",
    "max_length": 50
}
```

```
{
    "response": "AI is brilliant"
}
```

- Load the model and tokenizer from "distilbert/distilgpt2" with transformers (see examples), use transformers.TFAutoModelForCausalLM instead of AutoModel
- Import and use the inference function from gpt2_inference.py

Hint: Use the examples to have a starting point and revisit the last slides how to run it

Exploring API Documentation



Task: Go to http://127.0.0.1:8000/docs

Other Tools

- Ollama for hosting predefined LLMs (no custom models)
- Self-hosted ML API exposure tools: e.g., TensorFlow Serving, Triton Inference Server, MLFlow Serving
- Cloud-based platforms: e.g., AWS Sagemaker, Google Vertex AI, Azure ML Online