Interactive AI Demos & UI/UX

CS 203: Software Tools and Techniques for AI

Prof. Nipun Batra, IIT Gandhinagar

The "Demo" Gap

The Problem: You trained a great model. It lives in a Jupyter Notebook.
The Goal: Let stakeholders (non-coders) use it.

Options:

1. **Full Stack**: React + FastAPI + Docker + K8s. (Weeks of work).
2. **Low Code**: Streamlit / Gradio. (Hours of work).

Why Demos Matter:

- **Feedback**: Early user testing reveals edge cases.
- **Data Flywheel**: Users interacting = New training data.
- **Portability**: A URL is worth 1,000 GitHub stars.

The ML Model Deployment Spectrum

Approach	Time	Flexibility	Scalability	Use Case
Jupyter Notebook	0h	High	None	Personal use, research
Streamlit/Gradio	2h	Medium	Low-Medium	Demos, MVPs, research
FastAPI + Basic UI	1d	High	Medium	Internal tools
Full Stack App	2w+	Very High	High	Production products

Today's focus: The 2-hour solution that covers 80% of use cases.

Web App Fundamentals (Brief Primer)

Traditional web architecture:

- **Frontend** (Browser): HTML, CSS, JavaScript
- **Backend** (Server): Python, API endpoints, database
- **Communication**: HTTP requests/responses

Streamlit/Gradio simplification:

- **You write**: Pure Python
- **Framework handles**: HTML generation, CSS styling, WebSocket communication
- **Result**: Full web app without touching JavaScript

Trade-off: Less control, faster development.

Frontend vs Backend in AI Apps

Frontend responsibilities:

- User input (text, images, sliders)
- Display results (text, charts, images)
- Handle interactions (buttons, chat)
- Visual design and layout

Backend responsibilities:

- Load ML models
- Run inference
- Process data
- Business logic
- Database operations

Streamlit Architecture: The "Script" Model

Streamlit works differently from standard web apps.

Execution Flow:

1. User loads page -> Entire Python script runs top-to-bottom.
2. User clicks button -> **Entire script runs again from top!**
3. Frontend detects changes (diff) and updates DOM via WebSockets.

Implication:

- Variables are **reset** on every interaction.
- Need explicit **Session State** to remember things.
- Need **Caching** to prevent re-loading models.

Managing State

Without Session State:

```
count = 0
if st.button("Click"):
    count += 1
st.write(count) # Always 0 or 1. Resets every time!
```

With Session State:

```
if 'count' not in st.session_state:
    st.session_state.count = 0

if st.button("Click"):
    st.session_state.count += 1

st.write(st.session_state.count) # Persists!
```

Caching Strategies: Performance is UX

Re-loading a 5GB model on every button click = Unusable App.

1. `@st.cache_resource`: For Global Objects (Models, DB Connections).
   - Loaded once, shared across all users.
   - *Singleton pattern*.
2. `@st.cache_data`: For Computations (DataFrames, API calls).
   - Cached based on function inputs.
   - *Memoization pattern*.

```
@st.cache_resource
def load_llm():
    return AutoModelForCausalLM.from_pretrained("gpt2") # Run once

@st.cache_data
def load_csv(url):
```

UX Patterns for GenAI

1. **Streaming**: Don't make users wait 10s for full text.
   - Show tokens as they arrive.
   - **Streamlit**: `st.write_stream(generator)`.
2. **Latency Masking**:
   - Use spinners (`st.spinner`).
   - Show intermediate steps ("Parsing PDF...", "Embedding...").
3. **Feedback Loops**:
   - Add 👍 / 👎 buttons next to outputs.

Gradio: The Functional Approach

Gradio is optimized for "Input -> Model -> Output" workflows.

```
import gradio as gr

def predict(img):
    # Process image
    return "Cat"

# Auto-generates UI based on types
demo = gr.Interface(fn=predict, inputs="image", outputs="label")
demo.launch()
```

Streamlit vs Gradio:

- **Streamlit**: Full Data Dashboards, Multi-page apps, Custom Logic.
- **Gradio**: Quick Model APIs, Hugging Face Spaces integration.

Gradio Components

Input components (automatic type detection):

- `"text"` : Text box
- `"textbox"` : Multi-line text
- `"image"` : Image upload
- `"audio"` : Audio file/recording
- `"video"` : Video upload
- `"file"` : Any file type
- `"dataframe"` : Pandas DataFrame editor

Output components:

- `"text"` , `"textbox"` : Text display
- `"image"` : Display images

Advanced Gradio: Blocks API

Interface is simple but limited. **Blocks** gives full control.

```
import gradio as gr

with gr.Blocks() as demo:
    gr.Markdown("# Image Classifier")

    with gr.Row():
        img_input = gr.Image(type="pil")
        img_output = gr.Label()

    btn = gr.Button("Classify")
    btn.click(fn=predict, inputs=img_input, outputs=img_output)

demo.launch()
```

Benefits: Custom layouts, multiple inputs/outputs, complex interactions.

Streamlit Component Types

Input widgets:

- `st.text_input()` : Single line text
- `st.text_area()` : Multi-line text
- `st.number_input()` : Numbers with validation
- `st.slider()` : Numeric slider
- `st.select_slider()` : Categorical slider
- `st.checkbox()` : Boolean toggle
- `st.radio()` : Single choice from options
- `st.selectbox()` : Dropdown menu
- `st.multiselect()` : Multiple choices
- `st.file_uploader()` : File upload

Streamlit Display Components

Text & data:

- `st.write()` : Universal display (markdown, dataframes, charts)
- `st.markdown()` : Markdown with HTML
- `st.dataframe()` : Interactive table
- `st.table()` : Static table
- `st.json()` : Pretty-printed JSON
- `st.code()` : Syntax-highlighted code

Media:

- `st.image()` : Display images
- `st.audio()` : Play audio
- `st.video()` : Embed video

Multi-Page Apps in Streamlit

Structure:

```
app/
├── Home.py                 # Main page (entry point)
├── pages/
│   ├── 1_Model_Demo.py
│   ├── 2_Data_Explorer.py
│   └── 3_About.py
```

Automatic routing:

- Streamlit detects `pages/` directory
- Creates sidebar navigation
- Each page is independent Python script
- Shared state via `st.session_state`

Use case: Complex dashboards with multiple views.

UI/UX Principles for ML Apps

1. **Progressive Disclosure**:
   - Show simple interface first
   - Reveal advanced options when needed
   - Example: Basic text input → Advanced settings in expander
2. **Immediate Feedback**:
   - Show spinners during processing
   - Display intermediate steps
   - Use progress bars for long operations
3. **Error Handling**:
   - Catch exceptions gracefully
   - Show user-friendly error messages

UX Patterns for AI Applications

Loading states:

```
with st.spinner("Processing image..."):
    result = model.predict(image)
```

Progress tracking:

```
progress_bar = st.progress(0)
for i, item in enumerate(data):
    process(item)
    progress_bar.progress((i + 1) / len(data))
```

Error messages:

```
try:
    result = process_input(user_input)
except ValueError as e:
    st.error(f"Invalid input: {e}")
    st.info("Try entering a number between 1-100")
```

Latency and Performance UX

Perceived performance matters more than actual speed.

Strategies:

1. **Optimistic UI**: Show result immediately, validate later
2. **Skeleton screens**: Show placeholder content while loading
3. **Streaming**: Display partial results as they arrive
4. **Background processing**: Use async for long tasks
5. **Caching**: Avoid recomputing same inputs

Rule of thumb:

- < 100ms: Feels instant
- 100ms - 1s: Acceptable with feedback
- 1s - 10s: Need progress indicator

Streaming Responses (GenAI Apps)

Why stream?

- Large language models generate tokens sequentially
- Users can start reading before completion
- Better perceived performance

Streamlit example:

```
def generate_text(prompt):
    for token in llm.stream(prompt):
        yield token

st.write_stream(generate_text(user_prompt))
```

Gradio example:

```
def stream_response(message, history):
```

Handling Model Errors Gracefully

Common AI model failures:

1. **Input validation**: Wrong format, size, or type
2. **Out of domain**: Input unlike training data
3. **Model errors**: Crashes, OOM errors
4. **Timeout**: Model too slow

Best practices:

```
try:
    # Validate input
    if len(text) > 1000:
        st.warning("Text too long, truncating to 1000 chars")
        text = text[:1000]

    # Predict with timeout
    result = model.predict(text)
```

Collecting User Feedback

Why collect feedback?

- Identify model failures
- Improve with human feedback (RLHF)
- A/B testing different models
- Track user satisfaction

Simple thumbs up/down:

```
col1, col2 = st.columns([4, 1])
with col1:
    st.write(model_output)
with col2:
    if st.button("
👍
"):
        log_feedback(output, "positive")
    if st.button("
```

Data Collection from Demos

Ethical data collection:

1. **Transparency**: Tell users data is collected
2. **Consent**: Add checkbox for data sharing
3. **Privacy**: Don't store PII without permission
4. **Security**: Encrypt sensitive data

Simple logging:

```
import json
from datetime import datetime

def log_interaction(input_data, output, feedback=None):
    entry = {
        "timestamp": datetime.now().isoformat(),
        "input": input_data,
        "output": output,
        "feedback": feedback
```

Deployment Options Comparison

Platform	Cost	Compute	Best For	Limitations
HF Spaces	Free-$$	CPU/GPU	Public demos	Community visibility
Streamlit Cloud	Free-$$	CPU only	Streamlit apps	Limited resources
Render	Free-$$$	CPU/RAM	Web services	Sleep on inactivity
Railway	Pay-as-go	Flexible	Full apps	Can get expensive
AWS/GCP	Pay-as-go	Infinite	Production	Complex setup

For demos: HF Spaces or Streamlit Cloud are best.

Hugging Face Spaces Deep Dive

What is HF Spaces?

- Free hosting platform for ML demos
- Supports Streamlit, Gradio, Docker, static sites
- Automatic HTTPS, version control
- Optional paid GPU upgrades

Deployment steps:

1. Create Space on huggingface.co
2. Choose SDK (Streamlit/Gradio)
3. Add files: `app.py`, `requirements.txt`, `README.md`
4. Push to Space repository (git)
5. Auto-builds and deploys

Configuration Files

requirements.txt (Python dependencies):

```
streamlit==1.28.0
transformers==4.35.0
torch==2.1.0
pillow==10.0.0
```

README.md (Space metadata):

```
---
title: Image Classifier
emoji:
🖼️

colorFrom: blue
colorTo: purple
sdk: streamlit
python_version: 3.10
---
```

Best Practices for Deployment

1. **Environment management**:
   - Use `requirements.txt` (not conda)
   - Pin exact versions
   - Test locally first
2. **Error handling**:
   - Catch all exceptions
   - Show friendly error messages
   - Log errors for debugging
3. **Resource limits**:
   - Set max file upload size
   - Limit text input length

Streamlit Secrets Management

Local development (`.streamlit/secrets.toml`):

```
OPENAI_API_KEY = "sk-..."
DATABASE_URL = "postgresql://..."
```

Access in code:

```
import streamlit as st

api_key = st.secrets["OPENAI_API_KEY"]
```

In Streamlit Cloud:

- Settings → Secrets
- Paste TOML format
- Secrets are encrypted

Testing Interactive Apps

Local testing:

```
streamlit run app.py
```

Unit tests (for helper functions):

```
def test_process_text():
    result = process_text("hello")
    assert len(result) > 0
```

Integration tests (with Streamlit):

```
from streamlit.testing.v1 import AppTest

def test_app():
    at = AppTest.from_file("app.py")
    at.run()
    assert not at.exception
```

Performance Optimization

Model loading:

```
@st.cache_resource
def load_model():
    # Expensive operation
    return torch.load("model.pt")

model = load_model()  # Only runs once
```

Data processing:

```
@st.cache_data(ttl=3600)  # Cache for 1 hour
def load_data(url):
    return pd.read_csv(url)
```

Lazy loading:

- Don't import heavy libraries at top

Accessibility Considerations

Make your demo accessible:

1. **Alt text for images**: Describe what image shows
2. **Color contrast**: Use high contrast for text
3. **Keyboard navigation**: All features should work without mouse
4. **Screen reader friendly**: Use semantic HTML/markdown
5. **Clear labels**: Describe what each input does

Example:

```
st.image(img, caption="Classification result: Cat (98% confidence)")
st.text_input("Enter movie title", help="Search for any movie in our database")
```

UI/UX Design Principles for ML Apps

Progressive Disclosure: Start simple, reveal complexity gradually.

Example:

```
# Basic mode (default)
mode = st.radio("Mode", ["Simple", "Advanced"])

if mode == "Simple":
    prompt = st.text_input("Enter your question")
else:
    # Advanced options
    prompt = st.text_area("Enter your question")
    temperature = st.slider("Temperature", 0.0, 2.0, 0.7)
    max_tokens = st.number_input("Max tokens", 100, 2000, 500)
    top_p = st.slider("Top-p", 0.0, 1.0, 0.9)
```

Benefit: Beginners aren't overwhelmed, experts can customize.

Latency Management: Strategies

Perceived performance matters as much as actual speed

Three key strategies:

1. **Progress indicators**: Show that work is happening
    - Progress bars for known-duration tasks
    - Spinners for unknown-duration tasks
2. **Streaming**: Show partial results as they arrive
    - Essential for LLM text generation
    - Better UX than waiting for complete response
3. **Optimistic UI**: Assume success, update on completion
    - Immediate visual feedback

Latency Management: Implementation

Progress bars:

```
progress = st.progress(0)
for i, item in enumerate(data):
    process(item)
    progress.progress((i + 1) / len(data))
```

Streaming outputs:

```
output_placeholder = st.empty()
result = ""
for chunk in model.stream(prompt):
    result += chunk
    output_placeholder.write(result)
```

Optimistic UI:

```
st.success("Processing...")
result = model.predict(input)  # Async in production
```

Error Handling Patterns for ML Apps

User-friendly error messages:

```
try:
    result = model.predict(image)
except ValueError as e:
    if "shape" in str(e):
        st.error("
❌
 Invalid image size. Please upload an image larger than 224x224 pixels.")
    elif "format" in str(e):
        st.error("
❌
 Unsupported image format. Please upload JPG or PNG.")
    else:
        st.error(f"
❌
 Processing error: {e}")
except Exception as e:
    st.error("
❌
 Unexpected error. Please try again or contact support.")
    # Log for debugging
    logging.error(f"Prediction error: {e}", exc_info=True)
```

A/B Testing ML Demos

Test different UX variants:

```
# Assign users to variants
import hashlib

def get_variant(user_id):
    """Deterministic A/B split based on user ID."""
    hash_val = int(hashlib.md5(user_id.encode()).hexdigest(), 16)
    return "A" if hash_val % 2 == 0 else "B"

# Get or create user ID
if 'user_id' not in st.session_state:
    st.session_state.user_id = str(uuid.uuid4())

variant = get_variant(st.session_state.user_id)

if variant == "A":
    # Variant A: Simple interface
    st.title("AI Assistant")
    prompt = st.text_input("Ask a question")
else:
    # Variant B: Advanced interface
    st.title("
🤖
 AI Assistant - Powered by GPT-4")
    prompt = st.text_area("Ask a question", height=150)
    st.slider("Creativity", 0.0, 1.0, 0.7)

# Track metrics
log_interaction(st.session_state.user_id, variant, prompt)
```

Analytics and Monitoring for Demos

Track usage patterns:

```
import posthog

posthog.init('YOUR_API_KEY', host='https://app.posthog.com')

def log_event(event_name, properties=None):
    """Log events for analytics."""
    posthog.capture(
        distinct_id=st.session_state.user_id,
        event=event_name,
        properties=properties or {}
    )

# Track interactions
if st.button("Generate"):
    log_event("generate_clicked", {
        "prompt_length": len(prompt),
        "model": selected_model
    })

    result = model.generate(prompt)

    log_event("generation_complete", {
        "latency_ms": latency,
        "output_length": len(result)
    })
```

Scaling Demos That Go Viral

Problem: Demo gets popular, traffic spikes.

Solutions:

1. Rate limiting:

```
import time

if 'last_request_time' not in st.session_state:
    st.session_state.last_request_time = 0

def check_rate_limit(requests_per_minute=10):
    """Enforce rate limit."""
    current_time = time.time()
    time_since_last = current_time - st.session_state.last_request_time

    if time_since_last < 60 / requests_per_minute:
        wait_time = (60 / requests_per_minute) - time_since_last
        st.warning(f"
⏳
Please wait {wait_time:.1f}s before next request")
```

Version Control for Interactive Apps

Git strategies for Streamlit apps:

```
# .gitignore for Streamlit
.streamlit/secrets.toml  # Never commit secrets!
*.pyc
__pycache__/
.env

# Separate config from code
```

Config management:

```
# config.yaml
app:
  title: "My ML Demo"
  version: "1.2.0"
  models:
    - name: "GPT-3.5"
      endpoint: "https://api.openai.com/v1"
    - name: "GPT-4"
```

Summary

1. **Frameworks**: Streamlit for dashboards, Gradio for quick model APIs
2. **Architecture**: Understand re-run model and state management
3. **Performance**: Cache models and data aggressively
4. **UX**: Provide feedback, handle errors, stream when possible
5. **Deployment**: HF Spaces for easy, free hosting
6. **Best Practices**: Security, testing, accessibility

Advanced Topics:

- Progressive disclosure for UI design
- Latency management (perceived performance)
- Error handling patterns
- A/B testing variants

Additional Resources

Documentation:

- Streamlit: https://docs.streamlit.io
- Gradio: https://gradio.app/docs
- HF Spaces: https://huggingface.co/docs/hub/spaces

Inspiration:

- HF Spaces gallery: https://huggingface.co/spaces
- Streamlit gallery: https://streamlit.io/gallery

Analytics:

- PostHog: https://posthog.com (open-source analytics)
- Streamlit Analytics: https://docs.streamlit.io/library/api-reference/utilities/st.experimental_get_query_params