Handout 3: Building Custom GenAI Pipelines (Advanced Level)

Machine Learning for Smarter Innovation

1 Handout 3: Building Custom GenAI Pipelines (Advanced Level)

1.1 Production-Grade Generative AI Architecture

1.1.1 System Architecture

```python
# Complete GenAI Pipeline Implementation
import torch
from transformers import AutoModelForCausalLM, AutoTokenizer
from diffusers import StableDiffusionPipeline
import redis
from celery import Celery
import monitoring

class GenerativeAIPipeline:
    def __init__(self, config):
        self.text_model = self._load_text_model(config['text_model'])
        self.image_model = self._load_image_model(config['image_model'])
        self.cache = redis.Redis(host='localhost', port=6379)
        self.queue = Celery('tasks', broker='redis://localhost:6379')
        self.monitor = monitoring.PrometheusMonitor()

    def _load_text_model(self, model_name):
        """Load and optimize text generation model"""
        model = AutoModelForCausalLM.from_pretrained(model_name)
        model = model.half()  # FP16 for efficiency
        model = torch.compile(model)  # PyTorch 2.0 optimization
        return model

    @queue.task
    async def generate(self, prompt, modality='text'):
        """Async generation with caching and monitoring"""
        cache_key = hashlib.sha256(prompt.encode()).hexdigest()

        # Check cache
        if cached := self.cache.get(cache_key):
            self.monitor.cache_hit()
            return json.loads(cached)

        # Generate
        start_time = time.time()
        result = await self._generate_content(prompt, modality)

        # Monitor
        self.monitor.generation_time(time.time() - start_time)
```

```
        self.monitor.tokens_used(result['token_count'])

        # Cache
        self.cache.setex(cache_key, 3600, json.dumps(result))

        return result
```

1.2 Fine-tuning with LoRA

1.2.1 Efficient Fine-tuning Implementation

```
from peft import LoraConfig, get_peft_model, TaskType
from transformers import TrainingArguments, Trainer
import datasets

def fine_tune_model(base_model, training_data, output_dir):
    """Fine-tune model using LoRA for efficiency"""

    # LoRA Configuration
    lora_config = LoraConfig(
        r=16,  # Rank
        lora_alpha=32,
        target_modules=["q_proj", "v_proj"],  # Target attention layers
        lora_dropout=0.1,
        bias="none",
        task_type=TaskType.CAUSAL_LM,
    )

    # Apply LoRA
    model = get_peft_model(base_model, lora_config)
    model.print_trainable_parameters()  # Only ~0.1% trainable

    # Training Configuration
    training_args = TrainingArguments(
        output_dir=output_dir,
        num_train_epochs=3,
        per_device_train_batch_size=4,
        gradient_accumulation_steps=4,
        gradient_checkpointing=True,  # Memory optimization
        fp16=True,
        optim="adamw_8bit",  # 8-bit optimizer
        learning_rate=2e-4,
        warmup_ratio=0.1,
        logging_steps=10,
        save_strategy="epoch",
        evaluation_strategy="epoch",
    )

    # Train
    trainer = Trainer(
        model=model,
        args=training_args,
        train_dataset=training_data,
        data_collator=DataCollatorForLanguageModeling(tokenizer),
    )

    trainer.train()

    # Merge LoRA weights with base model
    model = model.merge_and_unload()
    return model
```

1.3 RAG Implementation

1.3.1 Production RAG System

```
from langchain.vectorstores import Pinecone
from langchain.embeddings import OpenAIEmbeddings
from langchain.chains import RetrievalQA
import pinecone

class RAGSystem:
    def __init__(self, index_name='knowledge-base'):
        # Initialize vector store
        pinecone.init(api_key=os.getenv('PINECONE_API_KEY'))
        self.embeddings = OpenAIEmbeddings()
        self.vectorstore = Pinecone.from_existing_index(
            index_name, self.embeddings
        )

        # Create retrieval chain
        self.qa_chain = RetrievalQA.from_chain_type(
            llm=ChatOpenAI(model="gpt-4", temperature=0),
            chain_type="stuff",
            retriever=self.vectorstore.as_retriever(
                search_type="similarity",
                search_kwargs={"k": 5}
            ),
            return_source_documents=True
        )

    def index_documents(self, documents):
        """Index new documents with metadata"""
        # Process documents
        text_splitter = RecursiveCharacterTextSplitter(
            chunk_size=1000,
            chunk_overlap=200
        )
        chunks = text_splitter.split_documents(documents)

        # Add metadata
        for i, chunk in enumerate(chunks):
            chunk.metadata['chunk_id'] = f"{chunk.metadata['source']}_{i}"
            chunk.metadata['timestamp'] = datetime.now().isoformat()

        # Index
        self.vectorstore.add_documents(chunks)

    def query(self, question, filters=None):
        """Query with source tracking"""
        if filters:
            self.qa_chain.retriever.search_kwargs['filter'] = filters

        result = self.qa_chain({"query": question})

        return {
            'answer': result['result'],
            'sources': [doc.metadata for doc in result['source_documents']],
            'confidence': self._calculate_confidence(result)
        }
```

1.4 Multi-Modal Generation Pipeline

1.4.1 Combining Text, Image, and Code Generation

```
class MultiModalGenerator:
    def __init__(self):
        self.text_gen = GPT4Generator()
        self.image_gen = StableDiffusionXL()
        self.code_gen = CodeLlamaGenerator()
        self.audio_gen = MusicGenGenerator()

    async def generate_product_concept(self, brief):
        """Generate complete product concept with all modalities"""

        # Parallel generation
        tasks = [
            self.text_gen.generate_description(brief),
            self.image_gen.generate_mockup(brief),
            self.code_gen.generate_prototype(brief),
            self.generate_marketing_materials(brief)
        ]

        results = await asyncio.gather(*tasks)

        return self._combine_outputs(results)

    def _combine_outputs(self, results):
        """Intelligently combine multi-modal outputs"""
        return {
            'description': results[0],
            'visuals': results[1],
            'code': results[2],
            'marketing': results[3],
            'consistency_score': self._check_consistency(results)
        }
```

1.5 Advanced Optimization Techniques

1.5.1 1. Model Quantization

```
from transformers import BitsAndBytesConfig

quantization_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_compute_dtype=torch.float16,
    bnb_4bit_quant_type="nf4",
    bnb_4bit_use_double_quant=True
)

model = AutoModelForCausalLM.from_pretrained(
    "meta-llama/Llama-2-70b",
    quantization_config=quantization_config,
    device_map="auto"
)
```

1.5.2 2. Flash Attention

```python
from flash_attn import flash_attn_func

class FlashAttentionModel(nn.Module):
    def forward(self, q, k, v):
        # 2-3x faster than standard attention
        return flash_attn_func(q, k, v, causal=True)
```

1.5.3 3. Speculative Decoding

```python
def speculative_decode(model_large, model_small, prompt, max_length=100):
    """Use small model to speculate, large model to verify"""
    tokens = tokenizer.encode(prompt)

    while len(tokens) < max_length:
        # Small model generates K tokens
        draft_tokens = model_small.generate(tokens, num_tokens=4)

        # Large model verifies in parallel
        verified = model_large.verify_batch(tokens, draft_tokens)

        # Accept verified tokens
        tokens.extend(verified)

    return tokenizer.decode(tokens)
```

1.6 Deployment Strategies

1.6.1 Kubernetes Deployment

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: genai-service
spec:
  replicas: 3
  template:
    spec:
      containers:
      - name: model-server
        image: genai:latest
        resources:
          requests:
            memory: "16Gi"
            nvidia.com/gpu: 1
          limits:
            memory: "32Gi"
            nvidia.com/gpu: 1
        env:
        - name: MODEL_NAME
          value: "llama-2-70b"
        - name: CACHE_SIZE
          value: "1000"
---
apiVersion: v1
kind: Service
metadata:
  name: genai-lb
spec:
```

```
  type: LoadBalancer
  selector:
    app: genai-service
  ports:
  - port: 80
    targetPort: 8080
```

1.6.2 Edge Deployment with ONNX

```
import onnxruntime as ort

def export_to_edge(model, example_input):
    """Export model for edge deployment"""
    # Convert to ONNX
    torch.onnx.export(
        model,
        example_input,
        "model.onnx",
        opset_version=14,
        dynamic_axes={'input': {0: 'batch_size'}}
    )

    # Optimize for edge
    optimized = optimize_model("model.onnx")

    # Quantize to INT8
    quantized = quantize_dynamic(
        optimized,
        weight_type=QuantType.QInt8
    )

    return quantized
```

1.7 Monitoring and Observability

1.7.1 Comprehensive Monitoring Setup

```
from prometheus_client import Counter, Histogram, Gauge
import logging
import wandb

class ModelMonitor:
    def __init__(self):
        # Metrics
        self.request_count = Counter('genai_requests_total')
        self.request_duration = Histogram('genai_request_duration_seconds')
        self.token_usage = Counter('genai_tokens_total')
        self.error_rate = Counter('genai_errors_total')
        self.cache_hit_rate = Gauge('genai_cache_hit_rate')

        # Logging
        self.logger = self._setup_logging()

        # Experiment tracking
        wandb.init(project="genai-production")

    def track_generation(self, func):
        """Decorator for monitoring generation"""
```

```
        def wrapper(*args, **kwargs):
            start = time.time()

            try:
                result = func(*args, **kwargs)
                self.request_count.inc()
                self.token_usage.inc(result.get('tokens', 0))
                wandb.log({
                    'latency': time.time() - start,
                    'tokens': result.get('tokens', 0)
                })
                return result

            except Exception as e:
                self.error_rate.inc()
                self.logger.error(f"Generation failed: {e}")
                raise

            finally:
                self.request_duration.observe(time.time() - start)

        return wrapper
```

1.8 Cost-Optimized Production Pipeline

1.8.1 Intelligent Request Routing

```
class CostOptimizer:
    def __init__(self):
        self.models = {
            'cheap': {'model': 'gpt-3.5', 'cost': 0.002},
            'balanced': {'model': 'claude-2', 'cost': 0.008},
            'quality': {'model': 'gpt-4', 'cost': 0.03}
        }

    def route_request(self, prompt, quality_required=0.8):
        """Route to appropriate model based on requirements"""
        # Analyze prompt complexity
        complexity = self._analyze_complexity(prompt)

        if complexity < 0.3 and quality_required < 0.7:
            return self.models['cheap']
        elif complexity < 0.7 and quality_required < 0.9:
            return self.models['balanced']
        else:
            return self.models['quality']

    def _analyze_complexity(self, prompt):
        """Estimate prompt complexity"""
        factors = {
            'length': len(prompt) / 1000,
            'technical_terms': self._count_technical_terms(prompt),
            'multi_step': 'step by step' in prompt.lower(),
            'creative': any(word in prompt.lower()
                            for word in ['creative', 'innovative', 'unique'])
        }
        return sum(factors.values()) / len(factors)
```

1.9 Security Considerations

1.9.1 Prompt Injection Protection

```python
def sanitize_prompt(prompt):
    """Protect against prompt injection"""
    # Remove potential injection patterns
    dangerous_patterns = [
        r'ignore previous instructions',
        r'system:',
        r'assistant:',
        r'<script>',
        r'DROP TABLE'
    ]

    for pattern in dangerous_patterns:
        if re.search(pattern, prompt, re.IGNORECASE):
            raise ValueError("Potential injection detected")

    # Limit prompt length
    if len(prompt) > 10000:
        prompt = prompt[:10000]

    return prompt
```

1.10 Your Challenge

Build a complete GenAI application that: 1. Accepts multi-modal input (text + image) 2. Uses RAG for domain knowledge 3. Implements caching and optimization 4. Includes monitoring and cost tracking 5. Deploys to cloud with auto-scaling

Evaluation criteria: - Performance: <2s latency at p99 - Cost: <$0.01 per request average - Quality: Human evaluation score >4/5 - Scale: Handle 1000 QPS

Advanced tip: The difference between a prototype and production is 10% features and 90% infrastructure. Plan accordingly.