perplexity

Traffic Scene Command Understanding VLM - Complete Project Blueprint

PaliGemma-Inspired Architecture (From Scratch)

Project Overview

Goal: Build a VLM that understands traffic scenes through natural language commands, with complete attention visualization and interpretability.

Architecture Style: PaliGemma (SigLip Vision Encoder + Gemma Decoder + Linear Projection)

Timeline: 7-10 days of focused work

Deliverable: Research-grade implementation with attention analysis

Project Structure & File Organization

```
traffic_vlm/
├── config/
│   ├── model_config.py          # Task 1
│   ├── training_config.py       # Task 2
│   └── dataset_config.py        # Task 3
│
├── data/
│   ├── dataset_builder.py       # Task 4
│   ├── command_generator.py     # Task 5
│   ├── tokenizer.py            # Task 6
│   └── data_loader.py          # Task 7
│
├── model/
│   ├── vision/
│   │   ├── siglip_encoder.py   # Task 8
│   │   ├── vision_embeddings.py # Task 9
│   │   └── vision_attention.py  # Task 10
│   │
│   ├── language/
│   │   ├── gemma_decoder.py    # Task 11
│   │   ├── decoder_layer.py    # Task 12
│   │   └── rope_embeddings.py  # Task 13
│   │
│   ├── fusion/
│   │   ├── projection_layer.py  # Task 14
│   │   ├── cross_attention.py   # Task 15
│   │   └── multimodal_fusion.py # Task 16
│   │
│   └── vlm_model.py            # Task 17
│
├── training/
│   ├── trainer.py              # Task 18
│   ├── loss_functions.py       # Task 19
│   ├── optimizer.py            # Task 20
│   └── scheduler.py            # Task 21
│
├── evaluation/
│   ├── metrics.py              # Task 22
│   ├── attention_metrics.py    # Task 23
│   └── evaluator.py            # Task 24
│
├── visualization/
│   ├── attention_viz.py        # Task 25
│   ├── cross_attention_viz.py  # Task 26
│   └── failure_analysis.py     # Task 27
│
├── experiments/
│   ├── ablation_studies.py     # Task 28
│   └── sensitivity_analysis.py # Task 29
│
├── utils/
│   ├── checkpoint_manager.py   # Task 30
│   ├── logging_utils.py        # Task 31
│   └── tensor_utils.py         # Task 32
│
└── notebooks/
    ├── data_exploration.ipynb  # Task 33
    ├── model_testing.ipynb     # Task 34
    └── results_analysis.ipynb  # Task 35
```

Detailed Task Breakdown

Phase 1: Configuration & Planning (Day 1)

Task 1: model_config.py - Define Model Architecture Parameters

Purpose: Central configuration for all model hyperparameters

What to Define:

- Vision encoder specs (patch size, hidden dim, num layers, num heads)
- Language decoder specs (vocab size, hidden dim, num layers, GQA config)
- Projection layer dimensions
- Maximum sequence lengths (image patches, text tokens)
- Attention configuration (KV-cache size, attention dropout)

- Model size targets (<50M params total)

Key Decisions to Make:

- Image resolution (128×128 vs 224×224)
- Patch size (8×8 vs 16×16) → determines num image tokens
- Hidden dimensions (256 vs 512) → memory tradeoff
- Number of decoder layers (4-8)
- Grouped Query Attention ratio (4:1 or 8:1)

Task 2: training_config.py - Training Hyperparameters

Purpose: All training-related configuration

What to Define:

- Batch size (1-4 for A3000, gradient accumulation steps)
- Learning rate schedules (warmup steps, decay strategy)
- Number of epochs
- Mixed precision settings (FP16/BF16)
- Gradient clipping thresholds
- Checkpoint frequency
- Validation frequency
- Early stopping criteria

Optimization Strategy to Plan:

- AdamW optimizer settings (betas, weight decay)
- Cosine annealing vs linear warmup
- Whether to freeze vision encoder initially
- LoRA/QLoRA configuration (if using parameter-efficient training)

Task 3: dataset_config.py - Data Configuration

Purpose: Dataset paths, splits, and preprocessing params

What to Define:

- BDD100K subset selection criteria
- Train/val/test split ratios (80/10/10)
- Image preprocessing specs (normalization values, augmentations)
- Command vocabulary constraints

- Maximum command length
- Filtering rules (weather, time of day, scene complexity)
- Label mapping for commands

Dataset Size Planning:

- Target: 5k-10k image-command pairs
- Distribution across command types
- Class balance strategy

Phase 2: Data Pipeline (Day 2)

Task 4: dataset_builder.py - BDD100K Processing

Purpose: Filter and prepare BDD100K for VLM training

What to Implement:

1. **Image Filtering Logic**:
    - Filter by weather (clear only)
    - Filter by time (daytime only)
    - Filter by scene type (urban intersections, highways)
    - Quality checks (resolution, corruption)
2. **Annotation Extraction**:
    - Parse BDD100K JSON annotations
    - Extract relevant objects (traffic lights, pedestrians, vehicles, lane markings)
    - Convert bounding boxes to usable format
    - Create object-scene relationship maps
3. **Image Preprocessing**:
    - Resize to target resolution
    - Store in efficient format (HDF5 or LMDB)
    - Compute dataset statistics for normalization
4. **Metadata Generation**:
    - Scene complexity scores
    - Object counts per image
    - Available commands per scene

Task 5: command_generator.py - Synthetic Command Creation

Purpose: Generate command-answer pairs from annotations

What to Implement:

1. **Command Templates by Category**:
   - Safety commands: "Can the car safely [action]?"
   - Detection commands: "Is there a [object] in the scene?"
   - State commands: "Is the traffic light [color]?"
   - Navigation commands: "Is [direction] clear?"
2. **Rule-Based Answer Generation**:
   - Define logic for each command type
   - Use annotation data to compute answers
   - Handle edge cases (no traffic light → NO for traffic light questions)
3. **Negative Sampling**:
   - Generate false commands for scenes
   - Ensure class balance (50% YES, 50% NO)
4. **Command Complexity Levels**:
   - Simple: Single object detection
   - Medium: State + object ("red traffic light")
   - Complex: Multi-object reasoning ("pedestrian AND no traffic light")
5. **Quality Control**:
   - Verify command-answer consistency
   - Remove ambiguous pairs
   - Log statistics per command type

Task 6: tokenizer.py - Custom Vocabulary Builder

Purpose: Create small, domain-specific tokenizer

What to Implement:

1. **Vocabulary Construction**:
   - Extract all unique words from generated commands
   - Add special tokens: [PAD], [SOS], [EOS], [UNK], [YES], [NO]
   - Keep vocab small (200-500 tokens)
2. **Tokenization Strategy**:
   - Word-level tokenization (simple, interpretable)

   - Handle punctuation
   - Case normalization
3. **Token ID Mapping**:
   - Word to ID dictionary
   - ID to word reverse lookup
   - Handle unknown words
4. **Sequence Processing**:
   - Padding strategy
   - Truncation rules
   - Special token insertion logic

Task 7: data_loader.py - PyTorch Dataset & DataLoader

Purpose: Efficient batch loading for training

What to Implement:

1. **Custom Dataset Class**:
   - Load image + command + answer triplets
   - On-the-fly image transformations (if not pre-processed)
   - Tokenize commands
   - Return properly formatted tensors
2. **Collate Function**:
   - Batch images into [B, C, H, W]
   - Pad command sequences to same length
   - Create attention masks for padding
   - Stack answer labels
3. **Data Augmentation** (for images):
   - Random horizontal flip
   - Color jittering (mild)
   - Random crop/resize
   - Normalization
4. **Efficient Loading**:
   - Multi-worker configuration
   - Prefetching strategy
   - Memory pinning for GPU transfer
5. **Validation/Test Loaders**:

- No augmentation
- Fixed random seed for reproducibility

Phase 3: Vision Encoder (Day 3)

Task 8: siglip_encoder.py - Main Vision Transformer

Purpose: Implement SigLip-style vision encoder

What to Implement:

1. **Encoder Architecture**:
   - Stack of N encoder layers
   - Pre-normalization (norm before attention/FFN)
   - Residual connections
   - Final layer normalization
2. **Forward Pass Logic**:
   - Input: [B, C, H, W] images
   - Output: [B, num_patches, hidden_dim] visual tokens
   - Return intermediate attention maps for visualization
3. **Initialization**:
   - Xavier/Kaiming initialization
   - Optional: Load pre-trained patch embeddings if using transfer learning
4. **Configuration Hooks**:
   - Allow layer freezing
   - Allow intermediate feature extraction
   - Support for different depths

Task 9: vision_embeddings.py - Patch Embedding Layer

Purpose: Convert images to patch tokens (following PaliGemma video)

What to Implement:

1. **Patch Extraction**:
   - Convolution-based patch extraction (kernel=patch_size, stride=patch_size)
   - Flatten spatial dimensions
   - Project to hidden_dim
2. **Positional Embeddings**:

- 2D learned positional embeddings
- OR: 2D sinusoidal embeddings
- Add to patch embeddings

3. **Class Token** (optional):
    - Prepend learnable [CLS] token
    - Decide if using CLS or average pooling later
4. **Output Format**:
    - [B, num_patches, hidden_dim]
    - For 224×224 image with 16×16 patches → 196 patches

Task 10: vision_attention.py - Multi-Head Self-Attention for Vision

Purpose: Self-attention mechanism for vision encoder

What to Implement:

1. **Multi-Head Attention**:
    - Q, K, V projections
    - Split into multiple heads
    - Scaled dot-product attention
    - Concatenate heads
    - Output projection
2. **Attention Computation**:
    - Compute attention scores: softmax(QK^T / sqrt(d_k))
    - Apply attention to values
    - Return attention weights for visualization
3. **Optimizations**:
    - Flash Attention compatible structure
    - Dropout on attention weights
    - Proper masking support (though vision doesn't need causal mask)
4. **Feed-Forward Network**:
    - Two linear layers with GELU activation
    - Dropout between layers
    - Expansion ratio (typically 4x hidden_dim)

Phase 4: Language Decoder (Day 4)

Task 11: gemma_decoder.py - Main Language Decoder

Purpose: Gemma-style decoder stack

What to Implement:

1. **Decoder Architecture**:
    - Stack of N decoder layers
    - Each layer has:
        - Self-attention with causal mask
        - Cross-attention with vision tokens
        - Feed-forward network
    - Pre-normalization with RMS Norm
2. **Forward Pass**:
    - Input: Token IDs [B, seq_len]
    - Vision tokens: [B, num_patches, hidden_dim]
    - Output: [B, seq_len, vocab_size] logits
3. **KV-Cache Implementation**:
    - Store past key-values for efficient inference
    - Update cache incrementally
    - Support prefill vs decode phases
4. **Output Head**:
    - Final linear projection to vocabulary
    - Option: Weight tying with input embeddings

Task 12: decoder_layer.py - Single Decoder Block

Purpose: One layer of the Gemma decoder

What to Implement:

1. **Self-Attention Block**:
    - RMS Norm before attention
    - Grouped Query Attention (GQA)
    - Residual connection
2. **Cross-Attention Block**:
    - RMS Norm before cross-attention

    - Query from text, Key/Value from vision
    - Residual connection
    - **THIS IS WHERE COMMAND-CONDITIONAL ATTENTION HAPPENS**
3. **Feed-Forward Block**:
    - RMS Norm before FFN
    - Two linear layers with activation (SwiGLU or GELU)
    - Residual connection
4. **Attention Weight Extraction**:
    - Store self-attention maps
    - Store cross-attention maps (critical for visualization)
    - Return alongside outputs

Task 13: rope_embeddings.py - Rotary Position Embeddings

Purpose: Implement RoPE as shown in PaliGemma video

What to Implement:

1. **Precompute Rotation Matrix**:
    - Compute sin and cos for all positions
    - Store as buffer (not trainable)
    - Handle different dimensions (query/key)
2. **Apply Rotations**:
    - Split Q and K into pairs
    - Rotate using precomputed sin/cos
    - Reconstruct rotated Q and K
3. **Handle Batches and Heads**:
    - Apply to [B, num_heads, seq_len, head_dim]
    - Efficient tensor operations
4. **Position Offset for KV-Cache**:
    - Support incremental position indices during generation

Phase 5: Multimodal Fusion (Day 5)

Task 14: projection_layer.py - Vision-to-Language Projection

Purpose: Map vision tokens to language decoder space (like PaliGemma linear projection)

What to Implement:

1. **Linear Projection**:
    - If vision_dim ≠ language_dim:
        - Single linear layer to project
    - If vision_dim == language_dim:
        - Optional identity or small MLP
2. **Normalization**:
    - Layer norm or RMS norm after projection
    - Ensures stable gradients
3. **Dimensionality Decision**:
    - Vision encoder: 768 dim (standard ViT)
    - Language decoder: 512 dim (smaller)
    - Projection: 768 → 512
4. **Learned vs Fixed**:
    - Make trainable
    - Initialize carefully (Xavier)

Task 15: cross_attention.py - Cross-Modal Attention Module

Purpose: Text queries attend to vision tokens

What to Implement:

1. **Cross-Attention Mechanism**:
    - Query: from text decoder
    - Key/Value: from projected vision tokens
    - Compute attention: softmax(Q @ K^T) @ V
2. **Grouped Query Attention** (for efficiency):
    - Reduce KV heads compared to Q heads
    - 8 query heads, 2 KV heads (4:1 ratio)
3. **Attention Map Extraction**:
    - Store [B, num_heads, text_len, image_patches]
    - This is your GOLD for visualization
    - Shows which image regions each command word attends to

4. **Masking Support**:
    - Padding mask for text
    - No causal mask (text sees all vision tokens)
5. **Multi-Head Output**:
    - Concatenate heads
    - Project to output dimension

Task 16: multimodal_fusion.py - Complete Fusion Strategy

Purpose: Orchestrate how vision and language interact

What to Implement:

1. **Token Concatenation Strategy**:
    - Option A: Prepend vision tokens to text tokens
        - [vision_tokens; text_tokens] → decoder
    - Option B: Cross-attention at every layer
        - Text tokens only, cross-attend to vision
    - **Choose Option B** (more like PaliGemma)
2. **Attention Mask Construction**:
    - Text tokens have causal mask (can't see future)
    - Cross-attention: text can see ALL vision tokens
3. **Position ID Handling**:
    - Vision tokens: position IDs 0 to num_patches-1
    - Text tokens: position IDs 0 to text_len-1
    - Separate position encodings
4. **Forward Pass Integration**:
    - Pass vision tokens to cross-attention in every decoder layer
    - Maintain separate KV-cache for self-attention and cross-attention

Task 17: vlm_model.py - Complete VLM Assembly

Purpose: Integrate all components into single model

What to Implement:

1. **Component Initialization**:
    - Vision encoder (SigLip)
    - Projection layer

    - Text embedding layer
    - Language decoder (Gemma)
    - Output head (classification or generation)
2. **Forward Pass**:
   ```
   image → vision_encoder → visual_tokens
   visual_tokens → projection → projected_vision
   text_ids → text_embeddings
   text_embeddings + projected_vision → decoder → logits
   logits → output (YES/NO or generated text)
   ```
3. **Training Mode**:
    - Return loss directly
    - Combine vision + language outputs
4. **Inference Mode**:
    - Autoregressive generation
    - KV-cache management
    - Beam search or greedy decoding
5. **Attention Collection**:
    - Aggregate attention weights from all layers
    - Return as dictionary for visualization
6. **Model Statistics**:
    - Count total parameters
    - Count trainable parameters
    - Memory footprint estimation

Phase 6: Training Infrastructure (Day 6)

Task 18: trainer.py - Training Loop

Purpose: Complete training orchestration

What to Implement:

1. **Training Loop Structure**:
    - Epoch loop
    - Batch iteration
    - Forward pass
    - Loss computation
    - Backward pass

    - Optimizer step
    - Logging
2. **Mixed Precision Training**:
    - AMP (Automatic Mixed Precision) setup
    - GradScaler for FP16
    - Loss scaling to prevent underflow
3. **Gradient Accumulation**:
    - Accumulate gradients over multiple batches
    - Effective batch size = batch_size × accumulation_steps
    - Only update optimizer every N steps
4. **Gradient Clipping**:
    - Clip by norm (typically 1.0)
    - Prevent exploding gradients
5. **Validation During Training**:
    - Run validation every N steps
    - Compute validation loss
    - Save best checkpoint
6. **Logging and Monitoring**:
    - Loss curves (train/val)
    - Learning rate schedule
    - Gradient norms
    - Attention entropy (custom metric)
    - GPU memory usage
7. **Checkpoint Management**:
    - Save every N epochs
    - Keep only best K checkpoints
    - Save optimizer state for resuming

Task 19: loss_functions.py - Loss Design

Purpose: Define task-specific losses

What to Implement:

1. **Classification Loss** (if YES/NO/MAYBE):
    - Cross-entropy loss
    - Handle class imbalance with weights

    - Focal loss (optional, for hard examples)
2. **Generation Loss** (if text generation):
    - Cross-entropy per token
    - Ignore padding tokens in loss
    - Label smoothing (optional)
3. **Contrastive Loss for Vision** (optional):
    - SigLip-style sigmoid loss
    - If pre-training vision encoder separately
    - Match image-text similarity
4. **Regularization Losses**:
    - L2 regularization (weight decay)
    - Attention entropy regularization (encourage confident attention)
5. **Loss Combination**:
    - Weighted sum of multiple losses
    - Tunable loss weights

Task 20: optimizer.py - Optimizer Configuration

Purpose: Set up optimizer with proper settings

What to Implement:

1. **AdamW Setup**:
    - Separate learning rates for vision/language
    - Weight decay (0.01 typical)
    - Betas (0.9, 0.999)
    - Epsilon for numerical stability
2. **Parameter Groups**:
    - Vision encoder: lower LR (or frozen)
    - Projection layer: medium LR
    - Language decoder: higher LR
    - No weight decay on biases and layer norms
3. **Learning Rate Finder** (optional):
    - Implement LR range test
    - Plot loss vs LR
    - Find optimal starting LR

Task 21: scheduler.py - Learning Rate Scheduling

Purpose: Dynamic LR adjustment during training

What to Implement:

1. **Warmup Schedule**:
    - Linear warmup for first N steps
    - Start from small LR (1e-6)
    - Ramp up to peak LR
2. **Decay Schedule**:
    - Cosine annealing after warmup
    - Decay to minimum LR (1e-7)
    - OR: Step decay at specific epochs
3. **Restart Strategy** (optional):
    - Cosine annealing with warm restarts
    - Helps escape local minima
4. **LR Monitoring**:
    - Log current LR every step
    - Visualize LR schedule

Phase 7: Evaluation & Metrics (Day 7)

Task 22: metrics.py - Core Performance Metrics

Purpose: Standard ML evaluation metrics

What to Implement:

1. **Classification Metrics**:
    - Accuracy (overall)
    - Precision (per class)
    - Recall (per class)
    - F1 score (macro and per class)
    - Confusion matrix
2. **Per-Command-Type Metrics**:
    - Accuracy for safety commands
    - Accuracy for detection commands
    - Accuracy for state commands

    - Identify which command types are hardest

3. **Confidence Calibration**:
    - Plot predicted probability vs actual accuracy
    - Compute ECE (Expected Calibration Error)
4. **Generation Metrics** (if using text generation):
    - BLEU score
    - ROUGE score
    - Exact match rate

Task 23: attention_metrics.py - Attention Analysis Metrics

Purpose: Novel metrics for attention patterns

What to Implement:

1. **Attention Entropy**:
    - Per head, per layer
    - Formula: $H(A) = -\Sigma A_{ij} \log(A_{ij})$
    - Low entropy = focused attention
    - High entropy = diffuse attention
    - Correlation with prediction confidence
2. **Attention Consistency**:
    - For same command, similar images
    - Compute cosine similarity of attention maps
    - Expected: high similarity
3. **Command Sensitivity**:
    - Same image, different commands
    - Measure attention shift magnitude
    - Quantify: $||A_{cmd1} - A_{cmd2}||_2$
    - Expected: different commands → different attention
4. **Grounding Score**:
    - If you have object bounding boxes
    - Compute IoU between:
        - High-attention regions
        - Ground truth object locations
    - Shows if model attends to relevant objects
5. **Attention Specialization**:

   - Per head: what does each head attend to?
   - Cluster attention patterns
   - Identify "pedestrian head," "traffic light head," etc.
6. **Attention Ablation**:
   - Zero out specific attention heads
   - Measure performance drop
   - Identify critical heads

Task 24: evaluator.py - Complete Evaluation Pipeline

Purpose: Run comprehensive evaluation

What to Implement:

1. **Inference Loop**:
   - Iterate over test set
   - Generate predictions
   - Collect attention weights
   - Store results
2. **Batch Evaluation**:
   - Compute all metrics in Task 22 and 23
   - Generate summary statistics
3. **Failure Case Collection**:
   - Store all incorrect predictions
   - Store attention maps for failures
   - Categorize failure types:
     - False positives
     - False negatives
     - Ambiguous cases
4. **Performance Breakdown**:
   - By scene complexity
   - By command length
   - By object count
   - By time of day (if applicable)
5. **Statistical Significance**:
   - Bootstrap confidence intervals
   - T-tests between model variants

Phase 8: Visualization (Day 8)

Task 25: attention_viz.py - Self-Attention Visualization

Purpose: Visualize self-attention in vision and language

What to Implement:

1. **Vision Self-Attention Maps**:
    - Convert attention weights to 2D grid
    - Overlay on original image
    - Show which patches attend to which
    - Generate heatmaps per layer
2. **Language Self-Attention**:
    - Token-to-token attention matrix
    - Show causal mask effects
    - Identify important word interactions
3. **Multi-Head Visualization**:
    - Show all heads side-by-side
    - Identify head specialization
    - Aggregate attention across heads
4. **Layer-wise Attention Evolution**:
    - Show attention at layer 1, 2, ..., N
    - Observe abstraction hierarchy

Task 26: cross_attention_viz.py - Cross-Modal Attention (CRITICAL)

Purpose: Visualize command-to-image attention

What to Implement:

1. **Attention Heatmap Overlay**:
    - For each word in command
    - Show which image patches it attends to
    - Overlay on original image
    - Color-coded by attention strength
2. **Word-Specific Attention**:
    - Extract attention for key words:
        - "pedestrian" → attends to person regions

        - "traffic light" → attends to signal regions
        - "left" → attends to left side of image
    - Demonstrate grounding
3. **Command Comparison**:
    - Same image, two commands side-by-side
    - Show attention shift
    - Highlight differences in attended regions
4. **Attention Flow Animation**:
    - Show attention changing through decoder layers
    - How attention refines over depth
5. **Failure Case Visualization**:
    - Show attention for wrong predictions
    - Identify attention errors:
        - Attends to wrong region
        - Diffuse attention (uncertain)
        - Correct attention but wrong output

Task 27: failure_analysis.py - Systematic Failure Investigation

Purpose: Deep dive into model mistakes

What to Implement:

1. **Error Categorization**:
    - Type 1: Vision error (didn't see object)
    - Type 2: Language error (misunderstood command)
    - Type 3: Reasoning error (saw object, wrong inference)
2. **Qualitative Analysis**:
    - Manual inspection of 50-100 failures
    - Document patterns
    - Common failure modes
3. **Confusion Analysis**:
    - Which command pairs are most confused?
    - Similarity matrix between commands
    - Identify ambiguous commands
4. **Data Quality Issues**:
    - Find annotation errors in dataset

- Find command generation errors

Phase 9: Ablation Studies (Day 9)

Task 28: ablation_studies.py - Systematic Component Removal

Purpose: Prove each component is necessary

What to Implement:

1. **Ablation Experiments**:
    - Remove cross-attention (use only vision features)
    - Freeze vision encoder
    - Remove positional encodings
    - Single attention head instead of multi-head
    - Remove RoPE, use learned positions
    - Smaller model (fewer layers)
2. **Training Multiple Variants**:
    - Train each ablation separately
    - Keep all else constant
    - Fair comparison
3. **Results Table**:
    - Compare accuracy, F1, attention entropy
    - Show performance drop for each ablation
    - Prove your design choices
4. **Statistical Testing**:
    - Is the drop significant?
    - Confidence intervals

Task 29: sensitivity_analysis.py - Attention Robustness

Purpose: Test attention pattern stability

What to Implement:

1. **Adversarial Perturbations**:
    - Add small noise to images
    - Does attention shift drastically?
    - Measure attention robustness

2. **Occlusion Analysis**:
    - Mask image regions
    - Does attention adapt?
    - Shows model's flexibility
3. **Command Paraphrasing**:
    - "Is there a pedestrian?" vs "Do you see a person?"
    - Should have similar attention patterns
    - Measure attention similarity
4. **Out-of-Distribution Testing**:
    - Test on different datasets (if available)
    - Different weather conditions
    - Different cities

Phase 10: Utilities & Infrastructure (Day 10)

Task 30: checkpoint_manager.py - Model Saving/Loading

Purpose: Robust checkpoint handling

What to Implement:

1. **Save Checkpoints**:
    - Model state dict
    - Optimizer state
    - Scheduler state
    - Training step/epoch
    - Best validation metrics
2. **Load Checkpoints**:
    - Resume training from checkpoint
    - Load for inference only
    - Handle missing keys (for architecture changes)
3. **Best Model Tracking**:
    - Save based on validation accuracy
    - Keep top-K checkpoints
    - Auto-delete old checkpoints
4. **Export for Inference**:
    - Save model without optimizer

- Save config alongside weights
- Version tracking

Task 31: logging_utils.py - Experiment Tracking

Purpose: Comprehensive logging

What to Implement:

1. **Logging Framework**:
   - Integrate TensorBoard or W&B (Weights & Biases)
   - Log scalar metrics (loss, accuracy)
   - Log images with attention overlays
   - Log model graphs
2. **Training Logs**:
   - Loss per step
   - Learning rate per step
   - Gradient norms
   - Training throughput (samples/sec)
3. **Evaluation Logs**:
   - Validation metrics every N steps
   - Confusion matrices as images
   - Attention visualization samples
4. **System Metrics**:
   - GPU memory usage
   - GPU utilization
   - Training time estimates

Task 32: tensor_utils.py - Helper Functions

Purpose: Common tensor operations

What to Implement:

1. **Attention Mask Creation**:
   - Causal mask for decoder
   - Padding mask from token IDs
   - Combined masks
2. **Sequence Padding/Truncation**:

    - Pad sequences to same length
    - Truncate if too long
    - Handle batch collation
3. **Attention Weight Extraction**:
    - Collect attention from all layers
    - Average across heads
    - Convert to numpy for visualization
4. **Model Surgery**:
    - Freeze/unfreeze layers
    - Count parameters
    - Print model summary

Phase 11: Analysis Notebooks (Final)

Task 33: data_exploration.ipynb

Purpose: Understand dataset before training

What to Analyze:

1. Command distribution
2. Image statistics (mean, std, resolution)
3. Label balance
4. Scene complexity distribution
5. Example visualization

Task 34: model_testing.ipynb

Purpose: Interactive model testing

What to Test:

1. Single forward pass
2. Attention extraction
3. Inference speed
4. Memory profiling
5. Architecture visualization

Task 35: results_analysis.ipynb

Purpose: Final results presentation

What to Present:

1. Training curves
2. Metric tables
3. Attention visualizations
4. Ablation study results
5. Failure case gallery
6. Conclusion and future work

□ Milestones & Checkpoints

Milestone 1: Data Ready (End of Day 2)

- [ ] BDD100K filtered and processed
- [ ] 5k+ command-answer pairs generated
- [ ] Dataloader tested and working
- [ ] Sample batch visualized

Milestone 2: Vision Encoder Working (End of Day 3)

- [ ] Vision encoder forward pass works
- [ ] Attention maps extractable
- [ ] Can encode images to tokens
- [ ] Memory footprint acceptable

Milestone 3: Language Decoder Working (End of Day 4)

- [ ] Decoder forward pass works
- [ ] RoPE embeddings applied correctly
- [ ] KV-cache logic implemented
- [ ] Can generate dummy text

Milestone 4: Full VLM Assembled (End of Day 5)

- [ ] Vision + Language integrated
- [ ] Cross-attention working
- [ ] Can run forward pass on batch
- [ ] Loss computes correctly

Milestone 5: Training Pipeline Complete (End of Day 6)

- [ ] Training loop runs without errors
- [ ] Validation runs correctly
- [ ] Checkpoints save/load
- [ ] Logging to TensorBoard works

Milestone 6: Model Converged (End of Day 7)

- [ ] Validation accuracy > 70%
- [ ] Training curves look healthy
- [ ] Overfitting checked (train vs val gap)
- [ ] Model makes sensible predictions

Milestone 7: Evaluation Complete (End of Day 8)

- [ ] All metrics computed
- [ ] Attention visualizations generated
- [ ] Failure cases identified
- [ ] Results documented

Milestone 8: Ablations Done (End of Day 9)

- [ ] All ablation experiments run
- [ ] Results table complete
- [ ] Proves architecture choices

Milestone 9: Final Report (End of Day 10)

- [ ] All notebooks complete
- [ ] Visualizations polished
- [ ] Documentation written
- [ ] Presentation ready

Success Criteria

Your project will be considered successful if:

1. **Model Works**:
    - Achieves >70% accuracy on test set
    - Trains within 2-3 days on your hardware
    - Converges (loss decreases smoothly)

2. **Attention is Interpretable**:
    - Cross-attention maps show clear grounding
    - Different commands → different attention patterns
    - Attention entropy correlates with confidence
3. **Analysis is Thorough**:
    - Ablation studies show each component matters
    - Failure analysis identifies improvement directions
    - Visualizations are publication-quality
4. **Code is Clean**:
    - Modular design
    - Well-commented
    - Reproducible (config files, random seeds)
5. **Presentation is Clear**:
    - Architecture diagram explains design
    - Results tell a story
    - Attention visualizations are compelling

Next Steps After Project

1. **Write a blog post** documenting your journey
2. **Open-source the code** on GitHub
3. **Submit to a workshop** (e.g., NeurIPS workshop on interpretability)
4. **Extend the project**:
    - Multi-turn dialogue
    - Video understanding (temporal VLM)
    - Real-world deployment on dashcam footage

Recommended Reading Order

Before starting each phase:

1. Re-watch relevant sections of PaliGemma video
2. Read attention mechanism explanations
3. Study similar projects on GitHub

This is your complete execution roadmap. No code, just tasks. Now go build it from scratch and demonstrate mastery of VLMs!