Phi-3-Vision on Apple Silicon:

MLX Porting Guide

Josef Albers

August 1, 2024

Abstract

This tutorial series presents a comprehensive guide to porting and optimizing Microsoft's Phi-3-Vision, a compact yet powerful vision-language model, to Apple's MLX framework for efficient execution on Apple Silicon. The series covers a range of advanced techniques for model adaptation and performance enhancement, including: 1) Basic implementation of Phi-3-Vision in MLX, 2) Integration of Su-scaled Rotary Position Embeddings (SuRoPE) for handling long contexts, 3) Implementation of efficient batching techniques, 4) Development of caching mechanisms for accelerated text generation, 5) Exploration of advanced decoding strategies for guided outputs, 6) Implementation of Low-Rank Adaptation (LoRA) for efficient fine-tuning, and 7) Creation of an Agent class with a flexible toolchain system for complex AI workflows. Additionally, the series demonstrates the broader applicability of these techniques by extending them to port Google's PaliGemma model. This work contributes to the growing field of optimizing large language models for consumer-grade hardware, potentially broadening access to sophisticated AI capabilities.

Contents

1 Porting Phi-3-Vision to MLX: A Python Hobbyist's Journey

1.1 Introduction:

Welcome to a series on optimizing cutting-edge AI models for Apple Silicon! Over the next few weeks, we'll dive deep into the process of porting Phi-3-Vision, a powerful and compact vision-language model, from Hugging Face to MLX.

This series is designed for AI enthusiasts, developers, and researchers interested in running advanced models efficiently on Mac devices. For those eager to get started, you can find the MLX ports of both Phi-3-Mini-128K and Phi-3-Vision in my GitHub repository: https://github.com/JosefAlbers/Phi-3-Vision-MLX

1.2 Why Phi-3-Vision?

When Microsoft Research released Phi-3, I was immediately intrigued. Despite its relatively small size of 3.8 billion parameters, it was performing on par with or even surpassing models with 7 billion parameters. This efficiency was impressive and hinted at the potential for running sophisticated AI models on consumer-grade hardware.

The subsequent release of Phi-3-Vision further piqued my interest. As an owner of a Mac Studio and a Python hobbyist, I saw an exciting opportunity to bring this capable vision-language model to Apple Silicon. While llama.cpp was a popular option for running large language models on Mac, its C++ codebase was way beyond my skill level, so I looked for a more accessible option. This led me to MLX, Apple's machine learning framework that not only offered a Python-friendly environment but also promised better performance than llama.cpp on Apple Silicon.

What made this journey even more exciting was that it marked my first foray into contributing to open source projects. As I worked on porting Phi-3-Vision, I found myself making my first pull requests to repositories like "mlx-examples" and "mlx-vlm". This experience was an invaluable learning experience that helped me gain a better understanding of the MLX framework and how to apply it to real-world projects. This experience also connected me with the broader AI development community.

1.3 Useful Resources:

Before we dive into the series, I want to highlight some excellent resources that have been invaluable in my journey:

1. **MLX Examples** (https://github.com/ml-explore/mlx-examples): This official repository from the MLX team at Apple is a treasure trove of examples and tutorials that showcase the capabilities of the MLX framework. With a wide range of standalone examples, from basic MNIST to advanced language models and image generation, this repository is an excellent starting point for anyone looking to learn MLX. The quality and depth of the examples are truly impressive, and they demonstrate the team's commitment to making MLX accessible to developers of all levels. I also want to give a special shoutout to awni, the repo owner, who was incredibly kind and patient with me when I made my first-ever pull request to this repository. Despite my lack of experience with Git and GitHub, awni guided me through the process and helped me navigate the precommit hooks and other nuances of the repository. Their patience and willingness to help a newcomer like me was truly appreciated, and I'm grateful for the opportunity to have contributed to this repository. If you're new to MLX or Git, I highly recommend checking out this repository and reaching out to awni - they're a great resource and a pleasure to work with!

2. **MLX-VLM** (https://github.com/Blaizzy/mlx-vlm): A package specifically for running Vision Language Models on Mac using MLX. This repository was particularly helpful in understanding how to handle multimodal inputs, and I found the well-organized and well-written code to be incredibly valuable in learning not only Vision Language Models (VLMs) but also the MLX framework in general. The codebase is a great example of how to structure and implement complex AI models using MLX, making it an excellent resource for anyone looking to learn from experienced developers and improve their own MLX skills. For those interested in other models, Prince Canuma has an excellent tutorial on running Google's Gemma 2 locally on Mac using MLX: https://www.youtube.com/watch?v=CKznaU1HpVQ

3. **Hugging Face** (https://huggingface.co/): A popular platform for natural language processing (NLP) and computer vision tasks, providing a vast range of pre-trained models, datasets, and tools. Hugging Face's Transformers library is particularly useful for working with transformer-based models like Phi-3-Vision. Their documentation and community support are also top-notch, making it an excellent resource for anyone looking to learn more about NLP and computer vision.

These resources provide a great foundation for anyone looking to explore MLX and run advanced AI models on Apple Silicon.

1.4 What to Expect in This Series:

1.4.1 MLX vs. Hugging Face: A Code Comparison

We'll start by comparing the original Hugging Face implementation with our MLX port, highlighting key differences in syntax and how MLX leverages Apple Silicon's unified memory architecture.

1.4.2 Implementing SuRoPE for 128K Context

We'll explore the Surrogate Rotary Position Embedding (SuRoPE) implementation that enables Phi-3-Vision to handle impressive 128K token contexts.

1.4.3 Optimizing Text Generation in MLX: From Batching to Advanced Techniques

Learn how to implement efficient batch text generation, an essential feature for many real-world applications. We'll also cover custom KV-Cache implementation and other text generation optimizations.

1.4.4 LoRA Fine-tuning and Evaluation on MLX

Discover how to perform Low-Rank Adaptation (LoRA) training, enabling efficient fine-tuning of Phi-3-Vision on custom datasets.

1.4.5 Building a Versatile AI Agent

In our finale, we'll create a multi-modal AI agent showcasing Phi-3-Vision's capabilities in handling both text and visual inputs.

1.5 Why This Series Matters:

Phi-3-Vision represents a significant advancement in compact, high-performing vision-language models. By porting it to MLX, we're making it more accessible and efficient for a wide range of applications on Apple

Silicon devices. This project demonstrates the potential of running advanced AI models on consumer-grade hardware, specifically Apple Silicon Macs.

1.5.1 Throughout this series, we'll highlight:

- Performance gains on Apple Silicon
- Challenges in porting and how to overcome them
- The process of contributing to open source AI projects
- Practical applications of the optimized model

1.5.2 Who This Series Is For:

- AI enthusiasts and hobbyists looking to dive deeper into model optimization
- Researchers exploring efficient AI on consumer hardware
- Mac users eager to leverage their devices for AI tasks
- Anyone curious about the intersection of AI and Apple Silicon
- Beginners interested in contributing to open source AI projects

1.6 Stay Tuned!

Our journey into optimizing Phi-3-Vision for MLX promises to be full of insights, challenges, and exciting breakthroughs. Whether you're a fellow hobbyist looking to run advanced AI models on your Mac or simply curious about the future of AI on consumer devices, this series has something for you.

Join me on this adventure in AI optimization, and let's unlock the full potential of Phi-3-Vision on Apple Silicon together!

2 Part 1: Basic Implementation of Phi-3-Vision in MLX

2.1 Introduction

Welcome to Part 1 of the tutorial series on porting Phi-3-Vision from PyTorch to Apple's MLX framework. Our goal is to create a minimal functional implementation of Phi-3-Vision in MLX through:

1. Analyzing the original PyTorch code
2. Translating core components to MLX
3. Building a basic MLX implementation
4. Loading and running the ported model

By the end of this tutorial, we will have a basic working model capable of generating text, setting the stage for further optimizations in subsequent parts of the series.

The full implementation of this tutorial is available at https://github.com/JosefAlbers/Phi-3-Vision-MLX/tree/main/assets/tutorial_1.py

2.2 Analyzing the Source Code

Our first task is to locate the source code for the original Phi-3-Vision:

1. Visit the Hugging Face model hub: https://huggingface.co/models
2. Search for "phi-3-vision"
3. Click on the model to access its repository
4. Look for a file named `modeling_phi3_v.py`

Now let's examine the code:

1. Scroll to the bottom of the file to find the top-level model class (`Phi3VForCausalLM` in our case)
2. Look for the `forward` method in this class
3. Trace the flow of data through the model by following method calls

Through this process, we can identify five key components of the model:

1. `Phi3VModel`: Main model
2. `Phi3DecoderLayer`: Decoder layers
3. `Phi3Attention`: Attention mechanism
4. `Phi3MLP`: Feed-forward network
5. `Phi3ImageEmbedding`: Image embedding

With these components identified, we're ready to begin the translation process to MLX.

2.3 MLX-Specific Considerations

A few differences between PyTorch and MLX to note before we begin the porting:

1. **Array Creation**: MLX doesn't require specifying device location.
2. **Lazy Computation**: Arrays in MLX are only materialized when needed.
3. **Model Definition**: MLX uses `__call__` instead of `forward` for the model's forward pass.

2.4 Understanding the Model Structure

Let's now examine each key component of Phi-3-Vision, translating them to MLX as we go:

2.4.1 Top-Level Model: Phi3VForCausalLM

This class serves as the main entry point of the model. It encapsulates the core `Phi3VModel` and adds a language modeling head.

```python
class Phi3VForCausalLM(nn.Module):
    # ...
    def __call__(self, input_ids, pixel_values=None, image_sizes=None):
        x = self.model(input_ids, pixel_values, image_sizes)
        return self.lm_head(x)
```

This top-level class serves two main functions:

1. **Encapsulating the core model**: It wraps the `Phi3VModel`, which produces contextualized representations of the input.
2. **Applying the language model head**: It uses a linear transformation to convert the contextualized representations into logits over the entire vocabulary, representing the model's predictions for the next token in the sequence.

2.4.2 Core Model: Phi3VModel

The Phi3VModel implements the main transformer architecture.

```python
class Phi3VModel(nn.Module):
    # ...
    def __call__(self, input_ids, pixel_values, image_sizes):
        x = self.embed_tokens(input_ids)
        x = self.vision_embed_tokens(x, pixel_values, image_sizes)
        for l in self.layers:
            x = l(x)
        return self.norm(x)
```

This class processes inputs through four stages:

1. **Text Embedding**: Input tokens are converted to dense vector representations.
2. **Vision Embedding**: If present, image inputs are processed and integrated with the text embeddings.
3. **Transformer Layers**: The combined embeddings are then passed through a series of decoder layers.
4. **Normalization**: The output is normalized before being returned.

2.4.3 Image Embedding: Phi3ImageEmbedding

This component processes image inputs and integrates them with text embeddings.

```python
class Phi3ImageEmbedding(nn.Module):
    # ...
    def __call__(self, txt_embeds, img_embeds, img_sizes, positions):
        # Process images with CLIP
        img_features = self.img_processor.vision_model(img_embeds)

        # Reshape and concatenate features
        global_features = self._process_global_features(img_features)
        local_features = self._process_local_features(img_features, img_sizes)
        # Apply additional projections
        x = mx.concatenate([local_features, global_features], axis=1)
        for layer in self.img_projection:
            x = layer(x)

        # Integrate with text embeddings
        txt_embeds = self._integrate_features(txt_embeds, x, positions)
        return txt_embeds
```

This class combines a CLIP (Contrastive Language-Image Pre-training) model with custom processing steps:

1. **CLIP Processing**: The model uses a pre-trained CLIP vision model to extract initial features from the input images.
2. **Additional Processing**: After CLIP processing, the model applies additional processing steps:
   - It reshapes and concatenates the features for both global and local (sub-image) representations.
   - It applies additional linear projections and non-linear activations (GELU) to further process these features.
3. **Integration with Text Embeddings**: Finally, the processed image features are integrated with the text embeddings at specific positions in the input sequence.

2.4.4 Decoder Layer: Phi3DecoderLayer

Each decoder layer is a fundamental building block of the transformer architecture.

```
class Phi3DecoderLayer(nn.Module):
    # ...
    def __call__(self, x):
        r = self.self_attn(self.input_layernorm(x))
        h = x + r
        r = self.mlp(self.post_attention_layernorm(h))
        return h + r
```

The decoder layer performs a series of operations to its input:

1. **Self-Attention**: This mechanism allows the model to weigh the importance of different parts of the input when processing each element, enabling it to capture long-range dependencies in the sequence.
2. **Feedforward Neural Network (MLP)**: This subnet processes each position independently, introducing non-linearity and increasing the model's capacity to learn complex functions.
3. **Residual Connections**: After both the self-attention and MLP operations, the input is added to the output. This technique helps in mitigating the vanishing gradient problem and allows for easier training of deep networks.
4. **Layer Normalization**: Applied before the self-attention and MLP operations, this normalizes the inputs to each sub-layer, stabilizing the learning process and allowing for deeper networks.

The combination of these components enables each layer to refine and enrich the representations passed through the model.

2.4.5 Attention Mechanism: Phi3Attention

The attention mechanism allows the model to weigh the importance of different parts of the input when processing each element.

```
class Phi3Attention(nn.Module):
    # ...
    def __call__(self, x):
        B, L, _ = x.shape
        qkv = self.qkv_proj(x)
        q, k, v = mx.split(qkv, self.chop, axis=-1)
        q = q.reshape(B, L, self.n_heads, -1).transpose(0, 2, 1, 3)
        k = k.reshape(B, L, self.n_kv_heads, -1).transpose(0, 2, 1, 3)
        v = v.reshape(B, L, self.n_kv_heads, -1).transpose(0, 2, 1, 3)
        mask = mx.triu(mx.full((x.shape[1], x.shape[1]), -mx.inf), k=1)
        w = (q * self.scale) @ k.transpose(0, 2, 3, 1)
        w += mask
        w = mx.softmax(w, axis=-1)
        o = w @ v
        o = o.transpose(0, 2, 1, 3).reshape(B, L, -1)
        return self.o_proj(o).astype(qkv.dtype)
```

Key aspects of this implementation:

1. **Projection and Splitting**: The input is first projected into query (q), key (k), and value (v) representations using a single linear projection (`qkv_proj`), which is then split.
2. **Multi-head Reshaping**: The q, k, and v tensors are reshaped to separate the heads and prepare for the attention computation.
3. **Attention Mask**: A causal mask is created to ensure that each position can only attend to previous positions.
4. **Scaled Dot-Product Attention**: The core attention computation is performed. Alternatively, you can use a faster, optimized version of this operation available in mlx.core.fast:

   ```
   # This:
   w = (q * self.scale) @ k.transpose(0, 1, 3, 2)
   w += mask
   w = mx.softmax(w, axis=-1)
   o = w @ v

   # Is equivalent to:
   o = mx.fast.scaled_dot_product_attention(q,k,v,scale=self.scale,mask=mask)
   ```

5. **Output Projection**: The attention output is reshaped and projected back to the original dimensionality.

The attention mechanism supports both standard multi-head attention and grouped-query attention by allowing different numbers of heads for queries (`n_heads`) versus keys/values (`n_kv_heads`). In the current configuration, however, these are set to the same value (32), resulting in standard multi-head attention.

2.4.6 MLP Layer: Phi3MLP

The MLP layer applies non-linear transformations to the attention outputs.

```
class Phi3MLP(nn.Module):
    # ...
    def __call__(self, x):
        x = self.gate_up_proj(x)
        gate, x = mx.split(x, 2, axis=-1)
        return self.down_proj(nn.silu(gate) * x)
```

This implements a gated feedforward network:

1. **Gated Architecture**:
   - The input is first projected into two separate spaces: one for the 'gate' and one for the 'values'.
   - This is achieved through a single linear projection followed by a split operation.
2. **Activation Function**:
   - The gate portion uses the SiLU (Sigmoid Linear Unit) activation, also known as the swish function.
   - SiLU is defined as f(x) = x * sigmoid(x), which has been shown to perform well in deep networks.
3. **Gating Mechanism**:
   - The activated gate is element-wise multiplied with the value portion.
   - This allows the network to dynamically control information flow, potentially helping with gradient flow and enabling more complex functions to be learned.
4. **Final Projection**:
   - The gated output is then projected back to the model's hidden size through a final linear layer.

This design combines the benefits of gating mechanisms (often seen in LSTMs and GRUs) with the simplicity and effectiveness of feedforward networks, potentially allowing for more expressive computations within each transformer layer.

2.5 Loading and Using the Model

Now that we've ported our model to MLX, let's load and use it for text generation.

First, we'll download the model configuration and weights from huggingface:

```
model_path = snapshot_download('microsoft/Phi-3-vision-128k-instruct')
```

Next, we'll load the model configuration:

```
with open(f"{model_path}/config.json", "r") as f:
    config = json.load(f)
model_config = SimpleNamespace(**config)
```

Now, let's load and "sanitize" the model weights:

```
model_weight = [(k, v.transpose(0, 2, 3, 1) if "patch_embedding.weight" in k
 ↪  else v)
                for wf in glob.glob(f"{model_path}/*.safetensors")
                for k, v in mx.load(wf).items()]
```

The line `v.transpose(0, 2, 3, 1) if "patch_embedding.weight" in k else v` adapts the patch embedding weights to MLX's format by converting them from PyTorch's NCHW (batch, channel, height, width) to MLX's NHWC (batch, height, width, channel) format. This transposition, often called "weight sanitization", is necessary when porting the model from PyTorch to MLX.

With our configuration and weights ready, we can initialize and load our model:

```
model = Phi3VForCausalLM(model_config)
model.load_weights(model_weight)
mx.eval(model.parameters())
model.eval()
```

Now that our model is loaded, let's use it to generate some text. First, we'll load the pretrained processor:

```
processor =
↪  AutoProcessor.from_pretrained('microsoft/Phi-3-vision-128k-instruct',
↪  trust_remote_code=True)
```

Then, we'll process our input text and generate the first token:

```
inputs = processor('Hello world!', return_tensors='np')
input_ids = mx.array(inputs['input_ids'])
logits = model(input_ids)
token = mx.argmax(logits[:, -1, :], axis=-1)
list_tokens = token.tolist()
```

This code processes the input text "Hello world!" and generates the first token. We use the `AutoProcessor` to tokenize the input, then pass it through the model to get logits. The token with the highest probability is selected as the next token.

To generate more tokens, we can use a simple loop:

```
for i in range(5):
    input_ids = mx.concatenate([input_ids, token[None]], axis=-1)
    logits = model(input_ids)
    token = mx.argmax(logits[:, -1, :], axis=-1)
    list_tokens += token.tolist()
```

This loop generates five additional tokens by repeatedly feeding our model's output back as input.

```
print(processor.tokenizer.decode(list_tokens))
# Output: How are you doing?<|end|>
```

And there we have it!

We've successfully ported Phi-3-Vision to MLX, loaded the model, and generated text. While this implementation is basic, it demonstrates that our port is functional and capable of generating coherent text.

2.6 Limitations

Our minimal implementation works for short sequences, but you'll notice it starts producing gibberish with longer contexts. This is because we haven't yet implemented position encoding, which we'll address in the next part with RoPE (Rotary Position Embedding).

2.7 Conclusion:

We’ve successfully created a barebones implementation of Phi-3-Vision in MLX. While it’s not yet fully functional, it provides a solid foundation for the optimizations we’ll explore in upcoming tutorials.

In Part 2, we’ll implement Su-scaled Rotary Position Embeddings (SuRoPE) to enhance our model’s ability to handle long sequences.

3 Part 2: Implementing Su-scaled Rotary Position Embeddings (RoPE) for Phi-3-Vision

3.1 Introduction

Welcome to Part 2 of our Phi-3-Vision porting series. In Part 1, we've created a basic implementation of the model in MLX. However, we also noted that it struggles with longer sequences. Today, we'll address this limitation by implementing Su-scaled Rotary Position Embeddings (SuRoPE), which will significantly enhance our model's ability to handle long contexts of up to 128K tokens.

The full implementation of this tutorial is available at https://github.com/JosefAlbers/Phi-3-Vision-MLX/tree/main/assets/tutorial_2.py

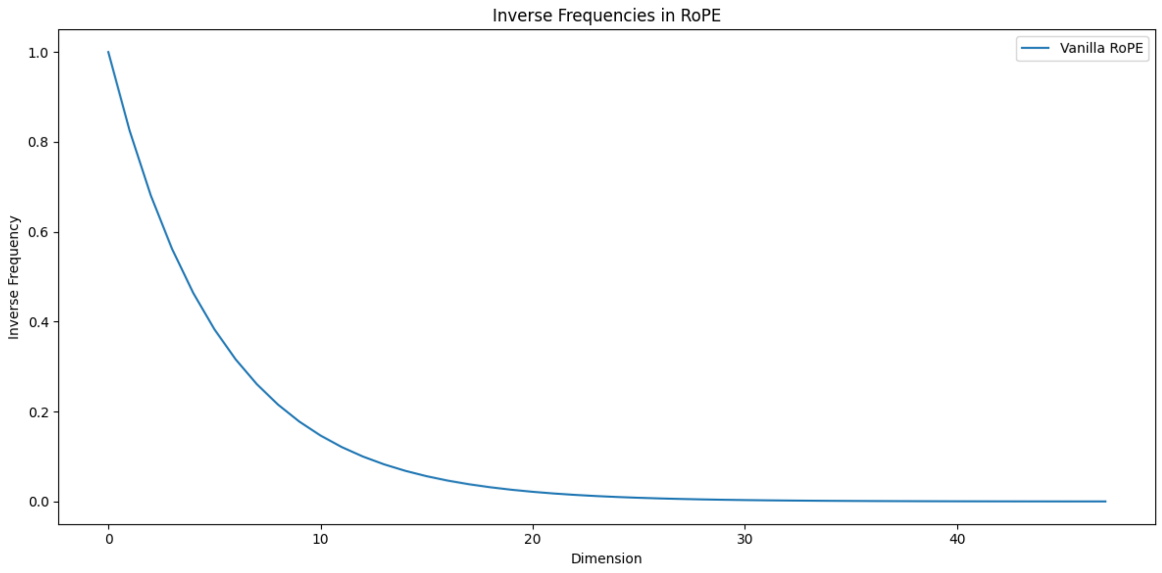
3.2 Understanding Rotary Position Embeddings (RoPE)

Before we delve into SuRoPE, let's first understand the basics of Rotary Position Embeddings (RoPE).

RoPE is a technique that injects positional information into the model's token representations without adding extra tokens or increasing the model's parameter count. The key idea is to apply a rotation to each token's embedding based on its position in the sequence.

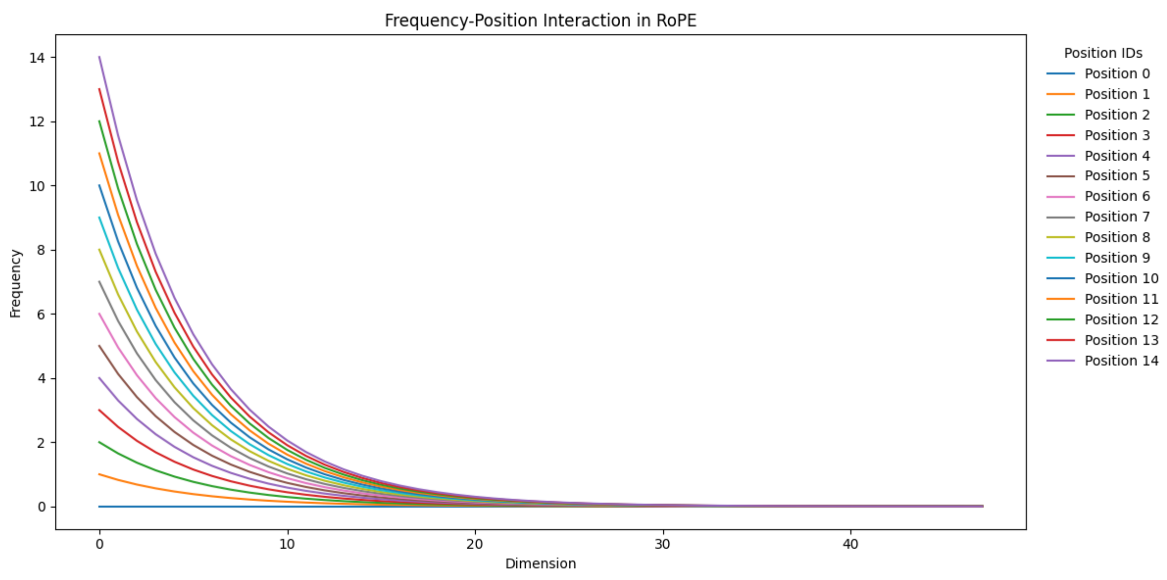
1. **Frequency Calculation**: For each dimension d in the embedding space, RoPE calculates a frequency:

```
inv_freq = 1 / (theta ** (d / dim))
```



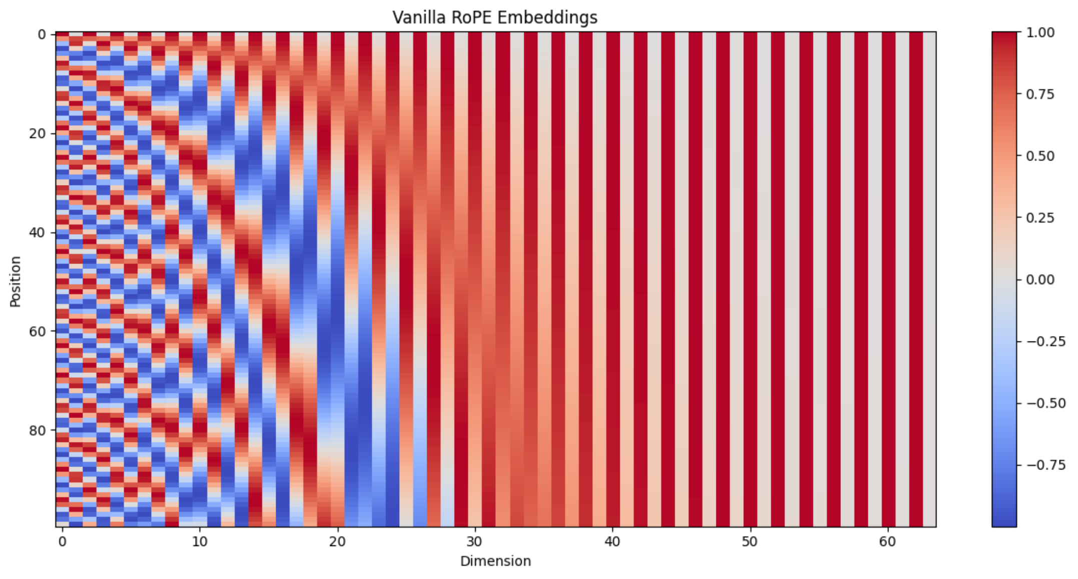
2. **Position-Frequency Interaction**: These frequencies are then multiplied by the token positions to create unique sinusoidal patterns for each position.

```
freqs = inv_freq @ position_ids.T
```



3. **Rotation Application**: The resulting patterns are used to rotate the token embeddings in 2D planes. For a token at position `pos`, RoPE applies the following rotation:

```
x_rotated = [x * cos(pos * freq) - y * sin(pos * freq),
             y * cos(pos * freq) + x * sin(pos * freq)]
```



Now that we understand RoPE, let's explore how Su-scaled RoPE builds upon and enhances this concept.

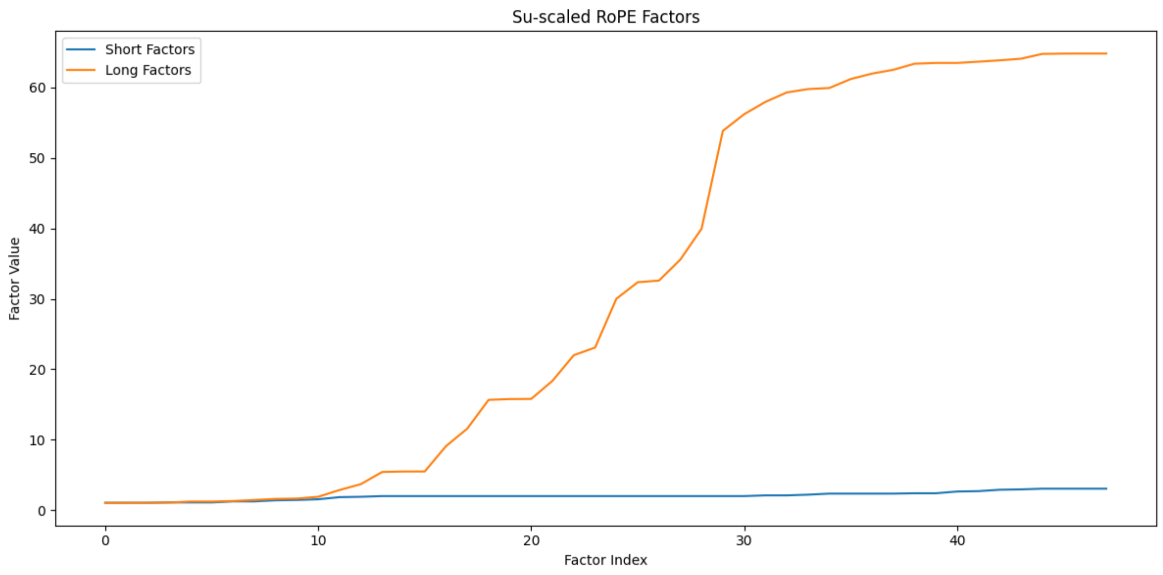
3.3 Understanding SuRoPE

SuRoPE extends RoPE by introducing scaling factors for different sequence length ranges.

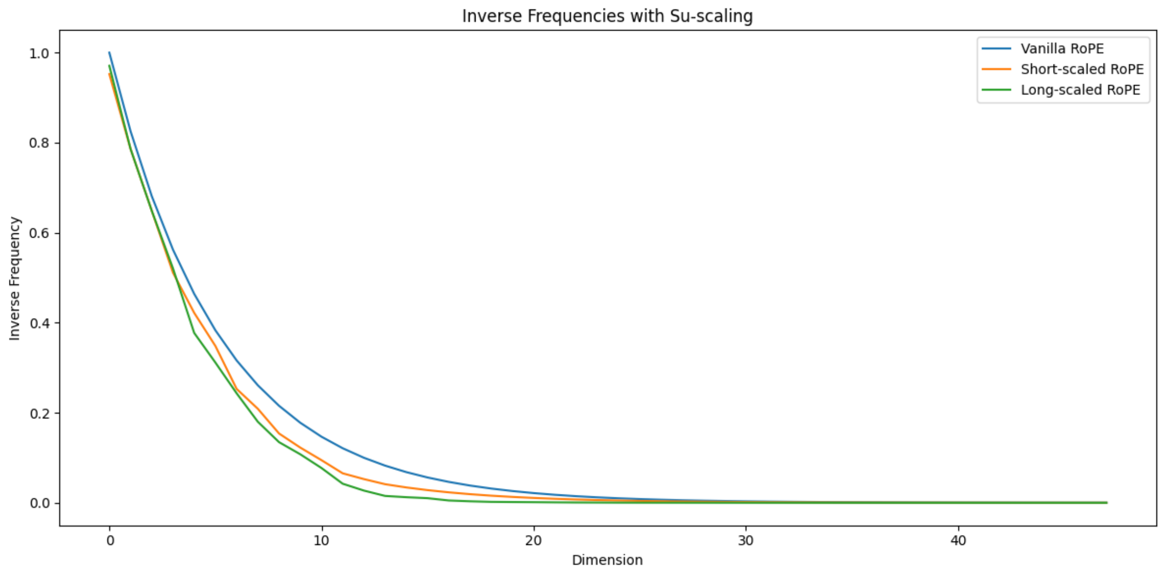
```
freq = 1 / (SU_FACTOR * theta ** (d / dim))
```

This allows the model to better generalize to sequences longer than those seen during training.

1. **Short and Long Factors**: Two sets of scaling factors are used, one for shorter sequences and one for longer sequences.



2. **Adaptive Scaling**: The choice between short and long factors is made based on the sequence length.



3. **Scaling Factor**: An additional scaling factor is applied to adjust for the extended maximum position embeddings.

3.4 Implementing Su-scaled RoPE

Now that we understand the theory behind Su-scaled RoPE, let's implement it in code. We'll create a `SuRoPE` class that encapsulates all the functionality we've discussed:

```python
import mlx.core as mx
import mlx.nn as nn
import math

class SuRoPE:
    def __init__(self, config):
        self.dim = config.hidden_size // config.num_attention_heads
        self.original_max_position_embeddings =
         ↪  config.original_max_position_embeddings
        self.rope_theta = config.rope_theta
        self.scaling_factor = math.sqrt(1 +
         ↪  math.log(config.max_position_embeddings /
         ↪  config.original_max_position_embeddings) /
         ↪  math.log(config.original_max_position_embeddings))
        self.long_factor = config.rope_scaling["long_factor"]
        self.short_factor = config.rope_scaling["short_factor"]

    def __call__(self, q, k, position_ids=None):
        position_ids = mx.arange(q.shape[2], dtype=mx.float32)[None] if
 ↪  position_ids is None else position_ids
        cos, sin = self._get_cos_sin(position_ids)
        q = (q * cos) + (self._rotate_half(q) * sin)
        k = (k * cos) + (self._rotate_half(k) * sin)
        return q, k

    def _get_cos_sin(self, position_ids):
        su_factor = self.long_factor if mx.max(position_ids) >
 ↪  self.original_max_position_embeddings else self.short_factor
        position_ids_expanded = position_ids[:, None, :]
        inv_freq = 1.0 / (mx.array(su_factor, dtype=mx.float32) *
 ↪  self.rope_theta**(mx.arange(0, self.dim, 2, dtype=mx.float32) / self.dim))
        inv_freq_expanded = mx.repeat(inv_freq[None, :, None],
 ↪  position_ids.shape[0], axis=0)
        freqs = (inv_freq_expanded @ position_ids_expanded).transpose(0, 2, 1)
        emb = mx.concatenate([freqs, freqs], axis=-1)
        cos = mx.expand_dims(mx.cos(emb) * self.scaling_factor, axis=1)
        sin = mx.expand_dims(mx.sin(emb) * self.scaling_factor, axis=1)
        return cos, sin

    @staticmethod
    def _rotate_half(x):
        midpoint = x.shape[-1] // 2
        x1, x2 = x[..., :midpoint], x[..., midpoint:]
        return mx.concatenate([-x2, x1], axis=-1)
```

3.5 Integrating Su-scaled RoPE into Phi-3-Vision

Integrating our Su-scaled RoPE implementation into the Phi-3-Vision model is straightforward. We only need to add two lines to our `Phi3Attention` module:

```python
class Phi3Attention(nn.Module):
    def __init__(self, config):
        # ...
        self.rope = SuRoPE(config)

    def __call__(self, x):
        # ...
        q, k = self.rope(q, k)
        # ...
```

These simple modifications allow our model to leverage Su-scaled RoPE, enabling it to handle sequences up to 128K tokens effectively.

3.6 Using the Updated Phi-3-Vision Model

Let's try an example that includes both text and an image:

```
from PIL import Image
import requests
prompt = f"<|user|>\n<|image_1|>\nWhat is shown in this
 ↪  image?<|end|>\n<|assistant|>\n"
images = [Image.open(requests.get("https://assets-
 ↪  c4akfrf5b4d3f4b7.z01.azurefd.net/assets/2024/04/BMDataViz_661fb89f3845e.png"
 ↪  , stream=True).raw)]
inputs = processor(prompt, images, return_tensors="np")
input_ids, pixel_values, image_sizes = [mx.array(inputs[i]) for i in
 ↪  ['input_ids', 'pixel_values', 'image_sizes']]
print(input_ids.shape)
# Output: (1, 1939)
```

Note that the input is translated into 1939 tokens. Let's generate a response:

```
logits = model(input_ids, pixel_values, image_sizes)
token = mx.argmax(logits[:, -1, :], axis=-1)
list_tokens = token.tolist()
for i in range(50):
    input_ids = mx.concatenate([input_ids, token[None]], axis=-1)
    logits = model(input_ids)
    token = mx.argmax(logits[:, -1, :], axis=-1)
    list_tokens += token.tolist()
print(processor.tokenizer.decode(list_tokens))
# Output: The image displays a chart with a series of connected dots forming a
 ↪  line that trends upwards, indicating a positive correlation between two
 ↪  variables. The chart is labeled with 'X' on the horizontal axis and 'Y' on
 ↪  the vertical axis,
```

This example showcases the model's ability to process a long input sequence (1939 tokens from the image plus the text prompt) and generate a coherent response, demonstrating the effectiveness of our Su-scaled RoPE implementation.

3.7 Limitations

While our Su-scaled RoPE implementation enhances the model's capacity for long sequences, two key limitations remain:

1. **Single Input Processing**: The current implementation processes only one input at a time, limiting throughput for multiple queries.
2. **Inefficient Generation**: Our token-by-token generation without caching leads to unnecessary repeated computations, slowing down the process.

These issues will be addressed in upcoming tutorials, where we'll explore efficient batching and caching mechanisms to improve the model's speed and inefficiency.

3.8 Conclusion

In this tutorial, we implemented Su-scaled Rotary Position Embeddings (SuRoPE), enabling our model to handle sequences up to 128K tokens.

In Part 3, we'll explore batching techniques to further optimize our Phi-3-Vision implementation in MLX.

4 Part 3: Implementing Batching for Phi-3-Vision in MLX

4.1 Introduction

In this tutorial, we will explore how to implement batching for the Phi-3-Vision model in MLX. Batching enables the model to process multiple inputs simulatenously, significantly enhancing computational efficiency and accelerating text generation.

The full implementation of this tutorial is available at https://github.com/JosefAlbers/Phi-3-Vision-MLX/tree/main/assets/tutorial_3.py

4.2 Understanding Batching

Batching is a technique that allows the model to process multiple inputs in parallel. This approach is particularly advantageous for smaller large language models (sLLMs) like Phi-3, as it can massively speed up the text generation process.

4.3 Implementing Batching Utilities

To implement batching, we need to create utility functions that can handle padding, updating inputs, and generating attention masks.

4.3.1 Padding Function

The `pad_to_batch` function takes in a dictionary of inputs and returns a padded version of the inputs, along with the corresponding position IDs and attention masks.

```
def pad_to_batch(inputs):
    input_ids = [i.tolist() for i in inputs['input_ids']]
    max_length = max(len(sublist) for sublist in input_ids)
    return {
        'input_ids': mx.array([[0]*(max_length-len(sublist)) + sublist for
         ↪  sublist in input_ids]),
        'position_ids': mx.array([[1]*(max_length-len(sublist)) +
         ↪  list(range(len(sublist))) for sublist in input_ids]),
        'attention_mask': mx.array([[0]*(max_length-len(sublist)) +
         ↪  [1]*len(sublist) for sublist in input_ids]),
    }
```

This function pads the inputs to the same length, adjusts the position IDs, and creates attention masks. Note that we're padding on the left side to preserve the causal structure of the input sequence, as required by autoregressive models.

4.3.2 Input Update Function

The `update_inputs` function updates the inputs with newly generated tokens, maintaining the correct structure for position IDs and attention masks.

```
def update_inputs(inputs, token):
    input_ids, position_ids, attention_mask = inputs['input_ids'],
 ↪  inputs['position_ids'], inputs['attention_mask']
    return {
        'input_ids': mx.concatenate([input_ids, token[:,None]], axis=-1),
        'position_ids': mx.concatenate([position_ids, position_ids[:, -1:] +
         ↪  1], axis=1),
        'attention_mask': mx.concatenate([attention_mask,
         ↪  mx.ones((input_ids.shape[0], 1), dtype=attention_mask.dtype)],
         ↪  axis=1),
    }
```

This function updates our inputs with newly generated tokens, maintaining the correct structure for position IDs and attention masks.

4.4 Modifying the Model for Batched Inputs

To enable batching, we need to update our model to use the `position_ids` and `attention_mask`.

4.4.1 Updating the Model Interface

We modify the top-level `Phi3VForCausalLM` class to accept the batched inputs and pass them to its model.

```
class Phi3VForCausalLM(nn.Module):
    # ...
    def __call__(self, input_ids, pixel_values=None, image_sizes=None,
     ↪  position_ids=None, attention_mask=None):
        x = self.model(input_ids, pixel_values, image_sizes, position_ids,
 ↪  attention_mask)
        return self.lm_head(x)
```

4.4.2 Updating the Phi3VModel

Next, modify the `Phi3VModel` to pass `position_ids` and `attention_mask` to each layer:

```
class Phi3VModel(nn.Module):
    # ...
    def __call__(self, input_ids, pixel_values, image_sizes, position_ids,
     ↪  attention_mask):
        x = self.embed_tokens(input_ids)
        x = self.vision_embed_tokens(x, pixel_values, image_sizes)
        for l in self.layers:
            x = l(x, position_ids, attention_mask)
        return self.norm(x)
```

4.4.3 Updating the Attention Mechanism

Finally, update the Phi3Attention module to utilize `position_ids` and `attention_mask`:

```
class Phi3Attention(nn.Module):
    # ...
    def __call__(self, x, position_ids, attention_mask):
        # ...
        q, k = self.rope(q, k, position_ids)
        mask = mx.triu(mx.full((v.shape[2], v.shape[2]), -mx.inf), k=1)
        if attention_mask is not None:
            mask += mx.where(attention_mask[:, :, None]*attention_mask[:, None,
↪   :]==1, 0, -mx.inf)
            mask = mx.expand_dims(mask, 1)
        # ...
```

4.5 Using Batched Inputs

Here's an example of batched text generation:

```
# Prepare batched inputs
inputs = processor(['Hello World!', 'Guten Tag!'], return_tensors='np')
inputs = pad_to_batch(inputs)

# Generate tokens
logits = model(**inputs)
token = mx.argmax(logits[:, -1, :], axis=-1)
list_tokens = [token]
for i in range(5):
    inputs = update_inputs(inputs, token)
    logits = model(**inputs)
    token = mx.argmax(logits[:, -1, :], axis=-1)
    list_tokens.append(token)
list_tokens = mx.stack(list_tokens, axis=1).tolist()
print(processor.tokenizer.batch_decode(list_tokens))
# Output: ['How are you doing today?', 'Was möchten Sie w']
```

4.6 Conclusion

By implementing custom batching for Phi-3-Vision, we've enabled our model to efficiently handle multiple inputs while ensuring correct behavior for autoregressive generation. This approach provides fine-grained control over input processing, position IDs, and attention masks, which is essential for optimal model performance.

In the next part, we'll explore implementing efficient caching mechanisms to further accelerate text generation, especially for longer sequences.

5 Part 4: Implementing Caching for Phi-3-Vision in MLX

5.1 Introduction

In this tutorial, we'll implement caching for our Phi-3-Vision model in MLX. Caching is a key optimization technique that can significantly improve the efficiency of language models, especially during text generation tasks. By storing and reusing intermediate computational results, we can reduce redundant calculations and speed up the overall inference process.

The full implementation of this tutorial is available at https://github.com/JosefAlbers/Phi-3-Vision-MLX/tree/main/assets/tutorial_4.py

5.2 The Need for Caching

Our previous implementation of the Phi-3-Vision model processes the entire input sequence from scratch for each new token. This approach becomes inefficient as the sequence grows:

```
Without Caching:

Iteration 1: [Prompt] -> Model -> Token 1
Iteration 2: [Prompt, Token 1] -> Model -> Token 2
Iteration 3: [Prompt, Token 1, Token 2] -> Model -> Token 3
```

This repetitive processing leads to unnecessary computations.

5.3 How Caching Helps

Caching solves this problem by storing and reusing intermediate computations from previous iterations:

```
With Caching:

Iteration 1: [Prompt] -> Model -> Token 1, Cache
Iteration 2: Cache + [Token 1] -> Model -> Token 2, Cache
Iteration 3: Cache + [Token 2] -> Model -> Token 3, Cache
```

Instead of processing the entire sequence each time, the model processes only the new token and uses the cached information for the rest.

5.4 Implementing Caching

To implement caching, we need to modify the attention mechanism and the model layers to handle the cache.

5.4.1 Modifying the Attention Mechanism

We modify the attention mechanism to handle both cached and non-cached scenarios. We add a cache parameter to the `__call__` method, which is used to store and retrieve the cached values:

```python
class Phi3Attention(nn.Module):
    # ...
    def __call__(self, x, position_ids, attention_mask, cache):
        # ...
        if cache is None:
            position_ids = mx.arange(q.shape[2], dtype=mx.float32)[None] if
↪   position_ids is None else position_ids
            q, k = self.rope(q, k, position_ids)
            mask = mx.triu(mx.full((v.shape[2], v.shape[2]), -mx.inf), k=1)
            if attention_mask is not None:
                mask += mx.where(attention_mask[:, :, None]*attention_mask[:,
↪   None, :]==1, 0, -mx.inf)
                mask = mx.expand_dims(mask, 1)
        else:
            past_k, past_v, past_p, past_m = cache
            position_ids = past_p[:,-1:]+1
            mask = mx.pad(past_m[:,:,-1:,:], ((0,0),(0,0),(0,0),(0,1)))
            q, k = self.rope(q, k, position_ids)
            k = mx.concatenate([past_k, k], axis=2)
            v = mx.concatenate([past_v, v], axis=2)

        cache = (k, v, position_ids, mask)
        # ...
        return self.o_proj(o).astype(qkv.dtype), cache
```

This modification allows the attention mechanism to either compute from scratch or use and update the cache, depending on whether a cache is provided.

5.4.2 Updating the Model Layers

Next, we update the model layers to handle the cache by adding a cache parameter to the `__call__` method and passing it through each layer.

```
class Phi3DecoderLayer(nn.Module):
    # ...
    def __call__(self, x, position_ids, attention_mask, cache):
        r, cache = self.self_attn(self.input_layernorm(x), position_ids,
↪   attention_mask, cache)
        h = x + r
        r = self.mlp(self.post_attention_layernorm(h))
        return h + r, cache

class Phi3VModel(nn.Module):
    # ...
    def __call__(self, input_ids, pixel_values, image_sizes, position_ids,
     ↪  attention_mask, cache):
        x = self.embed_tokens(input_ids)
        x = self.vision_embed_tokens(x, pixel_values, image_sizes)
        cache = [None]*len(self.layers) if cache is None else cache
        for i, l in enumerate(self.layers):
            x, cache[i] = l(x, position_ids, attention_mask, cache[i])
        return self.norm(x), cache

class Phi3VForCausalLM(nn.Module):
    # ...
    def __call__(self, input_ids, pixel_values=None, image_sizes=None,
     ↪  position_ids=None, attention_mask=None, cache=None):
        x, cache = self.model(input_ids, pixel_values, image_sizes,
↪  position_ids, attention_mask, cache)
        return self.lm_head(x), cache
```

5.5 Using Caching

Here's an example use of caching in text generation:

```
# Initial input processing
inputs = processor('Hello world!', return_tensors='np')
input_ids = mx.array(inputs['input_ids'])

# Initial forward pass
logits, cache = model(input_ids)
token = mx.argmax(logits[:, -1, :], axis=-1)
list_tokens = token.tolist()

# Generate additional tokens using cache
for i in range(5):
    logits, cache = model(token[:,None], cache=cache)
    token = mx.argmax(logits[:, -1, :], axis=-1)
    list_tokens += token.tolist()

print(processor.tokenizer.decode(list_tokens))
```

In this example, we first process the initial input and obtain the cache. Then, for each subsequent token generation, we use and update this cache, significantly reducing computation time for longer sequences.

5.6 Conclusion

By implementing caching in our Phi-3-Vision model, we've significantly improved its efficiency for token generation, especially for longer sequences. This optimization is important for practical applications of large language models, enabling faster and more efficient text generation.

In the upcoming tutorials, we'll explore advanced decoding strategies that allow for greater control over the model's output. These techniques will enhance the versatility of Phi-3-Vision, enabling its adaptation to a wide range of specific tasks and requirements.

6 Part 5: Implementing Choice Selection

6.1 Introduction

In this tutorial, we'll be crafting a choice selection function for Phi-3-Vision. This function constrains the model to pick from a small set of predefined options, making it useful for multiple-choice scenarios.

The full implementation of this tutorial is available at https://github.com/JosefAlbers/Phi-3-Vision-MLX/tree/main/assets/tutorial_5.py

6.2 Understanding Choice Selection

Choice selection is a straightforward concept: we give the model a prompt and a set of choices, then ask it to select the most likely one. This is particularly useful when we have predefined answers and want the model to pick the best one.

6.3 Implementing Choice Selection

Here's our choice selection function:

```python
def choose(prompts, choices='ABCDE'):
    # 1. Prompt Processing
    inputs = batch_process(prompts)
    # 2. Option Encoding
    options = [processor.tokenizer.encode(f' {i}')[-1] for i in choices]
    # 3. Model Prediction
    logits, _ = model(**inputs)
    # 4. Option Selection
    indices = mx.argmax(logits[:, -1, options], axis=-1).tolist()
    # 5. Output Formatting
    output = [choices[i] for i in indices]
    return output
```

Let's break it down:

1. **Prompt Processing**: Process the input prompts using the `batch_process` function.
2. **Option Encoding**: Encode each possible choice as a token ID.
3. **Model Prediction**: Run the model to get logits for the next token.
4. **Option Selection**: Use `argmax` to find the index of the highest logit among our choice options.
5. **Output Formatting**: Map these indices back to our choice letters.

The function takes a list of prompts and a string of choice letters. It returns a list of selected choices.

6.4 Using Choice Selection

Here's an example:

```python
prompts = [
    "What is the largest planet in our solar system? A: Earth B: Mars C:
     ↪  Jupiter D: Saturn",
    "Which element has the chemical symbol 'O'? A: Osmium B: Oxygen C: Gold D:
     ↪  Silver"
]
```

```
choose(prompts, choices='ABCD')
# Output: ['C', 'B']
```

In this example, the model correctly selects 'C' (Jupiter) as the largest planet and 'B' (Oxygen) as the element with the chemical symbol 'O'.

6.5 Limitations

The choice selection method is simple and effective for multiple-choice scenarios. It's computationally efficient as it only requires a single forward pass through the model.

However, it's limited to scenarios where we have predefined choices. For more open-ended tasks, we'll need more advanced techniques like constrained beam search, which we'll cover in the next tutorial.

6.6 Conclusion

Choice selection provides a straightforward way to guide our Phi-3-Vision model's output when we have a predefined set of options. This implementation uses the raw logits from the model to make selections, which is computationally efficient and direct.

In our next tutorial, we'll explore constrained beam search, a more advanced technique for guided generation. This will allow us to guide the model's output more flexibly, enabling us to generate new text while following specific constraints.

7 Part 6: Implementing Constrained Decoding for Phi-3-Vision

7.1 Introduction

In this tutorial, we'll look at constrained decoding, a powerful technique for guiding the text generation of our Phi-3-Vision model. This method allows us to control the model's output structure and content without altering its underlying knowledge or capabilities. Constrained decoding is particularly useful for tasks requiring specific output formats or structured reasoning.

7.2 Understanding Constrained Decoding

Constrained decoding works by setting "constraints" - specific phrases or structures that the model must include in its output within a certain number of tokens. This approach offers several benefits:

1. **Structured Output**: Ensures generated text follows a predetermined format.
2. **Guided Reasoning**: Encourages step-by-step thought processes.
3. **Consistent Responses**: Helps maintain uniformity across multiple generations.
4. **Error Reduction**: Minimizes off-topic or irrelevant content.

Common applications include:

- Generating code with specific elements
- Creating responses in particular formats (e.g., JSON, XML)
- Producing step-by-step reasoning for problem-solving
- Answering multiple-choice questions with explanations

7.3 Implementing Constrained Decoding

Let's examine a pseudocode implementation of constrained decoding:

```
def constrain(model, processor, prompt, constraints):
    input_ids = process(prompt)
    for each constraint in constraints:
        max_tokens, constraint_text = constraint
        constraint_ids = tokenize(constraint_text)

        best_sequence = input_ids
        best_score = -infinity

        for token_count = 1 to max_tokens:
            candidate_sequences = generate_candidates(best_sequence)
            for each candidate in candidate_sequences:
                full_sequence = concatenate(candidate, constraint_ids)
                score = calculate_sequence_score(full_sequence)

                if score > best_score:
                    best_score = score
                    best_sequence = candidate

            if best_sequence ends with constraint_ids:
                break

        input_ids = concatenate(best_sequence, constraint_ids)
    return decode(input_ids)
```

This pseudocode outlines the core logic of constrained decoding. Here's how it works:

1. We start with the initial prompt.
2. For each constraint:
    - We generate candidate sequences up to the max token limit.
    - For each candidate, we calculate the score of the candidate plus the constraint.
    - We keep track of the best-scoring sequence.
    - If the best sequence naturally ends with the constraint, we stop early.
    - Otherwise, we force-append the constraint after reaching max tokens.
3. We return the final generated text.

This implementation allows for flexibility in how we apply constraints. It tries to generate text that naturally includes the constraints, but if it can't do so within the token limit, it ensures the constraints are still included.

It's worth noting that this is a simplified version of the algorithm. In practice, you might need to adjust this based on your specific model architecture and requirements. For example, you might want to implement beam search or adjust how scores are calculated for better results.

7.4 Using Constrained Decoding

Let's look at two practical applications of constrained decoding:

7.4.1 Structured Code Generation

Constrained decoding can be used to generate structured code snippets. By setting appropriate constraints, we can ensure that the model produces well-formatted, syntactically correct code that includes specific elements we require. Here's an example:

```
from phi_3_vision_mlx import generate, constrain

constrain(
    prompt="Write a Python function to calculate the Fibonacci sequence up to a
 ↪  given number n.",
    constr=[
        (100, "\n```python\n"),
        (100, " return "),
        (200, "\n```")
    ],
    use_beam=True
)
```

In this example, we're using constrained decoding to generate a Python function for calculating the Fibonacci sequence. The constraints serve several purposes:

1. The first constraint `(100, "\n```python\n")` ensures that the output begins with a Python code block, improving readability and making the code easy to copy and use.
2. The second constraint `(100, " return ")` guarantees that the function includes a return statement.

3. The final constraint `(200, "\n```")` closes the code block, ensuring a clean and complete code snippet.

These constraints help structure our generated code for clarity and functionality. This approach is particularly powerful for tasks that require consistent output formats or adherence to specific patterns.

It's especially useful in the context of AI agents and function calling. Constrained decoding ensures AI agents produce properly formatted JSON responses, SQL queries with required clauses, or standardized API calls. It's equally valuable for more complex scenarios, like implementing specific design patterns or creating CRUD operation boilerplate.

In multi-agent systems, constrained decoding maintains consistent interfaces between components, allowing outputs from one model to serve reliably as inputs for another. This consistency is key for building robust, multi-step AI workflows and seamlessly integrating AI-generated code into larger systems.

7.4.2 Guided Reasoning in Complex Decision-Making

Constrained decoding can also guide the model's reasoning process in complex scenarios like medical diagnosis. Let's look at an example:

```
prompts = [
    "A 20-year-old woman presents with menorrhagia for the past several years.
     ↪  She says that her menses "have always been heavy", and she has
     ↪  experienced easy bruising for as long as she can remember. Family
     ↪  history is significant for her mother, who had similar problems with
     ↪  bruising easily. The patient's vital signs include: heart rate 98/min,
     ↪  respiratory rate 14/min, temperature 36.1°C (96.9°F), and blood
     ↪  pressure 110/87 mm Hg. Physical examination is unremarkable.
     ↪  Laboratory tests show the following: platelet count 200,000/mm3, PT 12
     ↪  seconds, and PTT 43 seconds. Which of the following is the most likely
     ↪  cause of this patient's symptoms? A: Factor V Leiden B: Hemophilia A C:
     ↪  Lupus anticoagulant D: Protein C deficiency E: Von Willebrand disease",
    "A 25-year-old primigravida presents to her physician for a routine
     ↪  prenatal visit. She is at 34 weeks gestation, as confirmed by an
     ↪  ultrasound examination. She has no complaints, but notes that the new
     ↪  shoes she bought 2 weeks ago do not fit anymore. The course of her
     ↪  pregnancy has been uneventful and she has been compliant with the
     ↪  recommended prenatal care. Her medical history is unremarkable. She has
     ↪  a 15-pound weight gain since the last visit 3 weeks ago. Her vital
     ↪  signs are as follows: blood pressure, 148/90 mm Hg; heart rate, 88/min;
     ↪  respiratory rate, 16/min; and temperature, 36.6℃ (97.9℉). The blood
     ↪  pressure on repeat assessment 4 hours later is 151/90 mm Hg. The fetal
     ↪  heart rate is 151/min. The physical examination is significant for 2+
     ↪  pitting edema of the lower extremity. Which of the following tests o
     ↪  should confirm the probable condition of this patient? A: Bilirubin
     ↪  assessment B: Coagulation studies C: Hematocrit assessment D:
     ↪  Leukocyte count with differential E: 24-hour urine protein"
]

generate(prompts, blind_model=True)
# Output from the 1st prompt:
# D: Protein C deficiency\n\nProtein C deficiency is a rare genetic disorder
 ↪  that increases the risk of developing abnormal blood clots. It is an
 ↪  autosomal dominant disorder that can lead to venous thromboembolism. The
 ↪  patient's prolonged PTT and history of menorrhagia and easy bruising are
 ↪  consistent with a bleeding disorder. Protein C is a natural anticoagulant
 ↪  that inhibits the coagulation cascade. Deficiency of protein C can lead to
 ↪  a hypercoagulable state, which can result in venous thrombosis. The other
 ↪  options listed are not consistent with the patient's presentation. Factor V
 ↪  Leiden is a genetic disorder that increases the risk of developing abnormal
```

```
# Output from the 2nd prompt:
# E: 24-hour urine protein\n\nThe patient presents with hypertension,
 ↪  proteinuria, and edema, which are the three diagnostic criteria for
 ↪  preeclampsia. The diagnosis of preeclampsia is made when a pregnant woman
 ↪  has hypertension and either proteinuria or end-organ dysfunction after 20
 ↪  weeks of gestation. In this case, the patient has hypertension (blood
 ↪  pressure of 148/90 mm Hg on two separate occasions), edema, and a fetal
 ↪  heart rate of 151/min, which is slightly elevated but not indicative of
 ↪  fetal distress. The most appropriate test to confirm the diagnosis of
 ↪  preeclampsia is a 24-hour urine protein assessment. This test will help
 ↪  determine the degree of proteinuria, which is a key feature of
 ↪  preeclampsia. Other tests, such as bilirubin assessment, coagulation
 ↪  studies, hematocrit assessment, and leukocyte count with differential, are
 ↪  not specific for preeclampsia and would not be the first-line tests to
 ↪  confirm the diagnosis.<|end|>
```

In this unconstrained version, our model kind of jumps the gun, blurting out "D: Protein C deficiency" right off the bat, without really thinking it through. Once it's said that, it's stuck - even though it later correctly explains that Protein C deficiency actually causes blood clots, not bleeding. The model even mentions Von Willebrand disease, which would fit the symptoms better, but it can't backtrack. Instead, it doubles down and starts making up reasons to support its first guess. This is a classic case of an autoregressive model getting tunnel vision because of how it generates text one bit at a time.

This is where constrained decoding can help. By making the model "show its work" before giving an answer, we can help it come to more accurate conclusions. It's like asking a student to explain their reasoning before they circle their final answer. Let's see what happens when we use constrained decoding to guide our model's response:

```
constrain(
    prompts=prompts,
    constrs=[
        (0, '\nThe'),
        (100, ' The correct answer is'),
        (1, 'X.')
    ],
    blind_model=True,
    use_beam=True
)
# Output from the 1st prompt:
# The patient's symptoms of menorrhagia and easy bruising, along with a
 ↪  prolonged partial thromboplastin time (PTT) and normal platelet count and
 ↪  prothrombin time (PT), are suggestive of a coagulation factor deficiency.
 ↪  The correct answer is E.
# Output from the 2nd prompt:
# The patient’s history and physical examination are consistent with
 ↪  preeclampsia. Preeclampsia is a multisystem disorder that is characterized
 ↪  by new-onset hypertension and proteinuria after 20 weeks of gestation. The
 ↪  patient’s blood pressure is elevated (>140/90 mm Hg) and she has 2+ pitting
 ↪  edema of the lower extremity. The correct answer is E.
```

In this constrained version, we've applied a structure that forces the model to provide reasoning before giving an answer, similar to a chain-of-thought approach. This approach offers several benefits:

1. **Forced Reasoning**: The model must explain its thought process before concluding, encouraging a more thorough analysis of the provided information.
2. **Transparency**: The reasoning is clearly laid out, allowing us to understand how the model arrived at its conclusion.
3. **Reduced Bias**: By delaying the answer until after the reasoning, we potentially reduce the risk of the model retrofitting its explanation to match a premature conclusion.
4. **Improved Accuracy**: As seen in the first prompt, the constrained approach led to the correct diagnosis (E: Von Willebrand disease) compared to the incorrect diagnosis in the unconstrained version.

This method of constrained decoding is analogous to asking a student to "show their work" in an exam. It not only provides the final answer but also gives insight into the thought process, making the output more informative, transparent, and potentially more accurate.

7.5 Conclusion

By implementing constrained decoding in complex decision-making scenarios, we can create more reliable and interpretable AI systems. This is important in high-stakes domains like medical diagnosis, legal reasoning, or financial analysis, where understanding the reasoning behind a decision is as important as the decision itself.

In the next part of our series, we'll explore techniques for fine-tuning our model on custom datasets, allowing us to adapt Phi-3-Vision for specific tasks or domains.

8 Part 7: Understanding LoRA Training with MLX

8.1 Introduction

In this tutorial, we'll explore the concept of Low-Rank Adaptation (LoRA) and how it can be implemented for training language models using MLX. We'll use a simplified version of LoRA training for the Phi-3 model as an illustrative example.

The full implementation of this tutorial is available at https://github.com/JosefAlbers/Phi-3-Vision-MLX/tree/main/assets/tutorial_7.py

8.2 Understanding LoRA

Low-Rank Adaptation (LoRA) is an efficient fine-tuning technique for large language models. It works by adding small, trainable rank decomposition matrices to existing weights in the model, allowing for task-specific adaptation with minimal additional parameters.

The key idea behind LoRA is to represent the weight update as a low-rank decomposition:

```
W = BA
```

Where:

- B is a matrix of shape (d_model, r)
- A is a matrix of shape (r, d_model)
- r is the rank of the decomposition (typically much smaller than d_model)

This approach offers several advantages:

1. **Efficient Parameter Updates**: LoRA allows for updating only a small subset of parameters, making fine-tuning more computationally efficient.
2. **Flexibility in Layer Selection**: LoRA can be applied to specific layers (often attention layers), allowing for targeted model adaptation.
3. **Rank Control**: The rank of the LoRA decomposition can be adjusted, offering a balance between model adaptability and efficiency.
4. **Low-Rank Update**: By using low-rank matrices for updates, LoRA significantly reduces the number of trainable parameters.
5. **Efficiency in Fine-Tuning**: The approach enables efficient task-specific adaptation of large language models without the need to update all parameters.
6. **Modular Adaptation**: By saving LoRA weights separately, different adapters can be easily swapped for various tasks, enhancing the model's versatility.

8.3 Implementing LoRA in MLX

Let's break down the key components:

8.3.1 LoRA Linear Layer

We create a LoRALinear class that modifies the standard linear layer to incorporate LoRA matrices:

```
class LoRALinear(nn.Module):
    # ...
    def __call__(self, x):
        y = self.linear(x)
        z = (self.dropout(x) @ self.lora_a) @ self.lora_b
        z = y + (self.scale * z)
        return z.astype(x.dtype)
```

This class adds trainable LoRA matrices (`lora_a` and `lora_b`) to an existing linear layer, enabling efficient fine-tuning.

8.3.2 Applying LoRA to the Model

To apply LoRA to specific layers of the model, we use a function that replaces standard linear layers with LoRA-enabled versions:

```
def linear_to_lora_layers(model, lora_targets, lora_layers, lora_rank):
    def to_lora(layer):
        return LoRALinear.from_linear(layer, r=lora_rank, alpha=lora_rank,
         ↪  scale=1.0, dropout=0.0)
    for l in model.layers[-lora_layers:]:
        lora_layers = [(k, to_lora(m)) for k, m in l.named_modules() if k in
 ↪  lora_targets]
        l.update_modules(tree_unflatten(lora_layers))
```

This function allows us to selectively apply LoRA to specific parts of the model, typically attention layers.

8.3.3 Training Loop

The main training function, `train_lora`, orchestrates the LoRA fine-tuning process:

```
def train_lora(model_path='microsoft/Phi-3-vision-128k-instruct',
               adapter_path='adapters',
               lora_targets=["self_attn.qkv_proj"],
               lora_layers=1,
               lora_rank=16,
               num_steps=3,
               learning_rate=1e-4):
```

Key steps in the training process:

```
def prepare_batch(index):
    tokens = mx.array(processor.tokenizer.encode(dataset[index]))[None]
    input_ids = tokens[:, :-1]
    target_ids = tokens[:, 1:]
    return input_ids, target_ids
```

8.3.3.1 Data Preparation This function prepares a single training example. It tokenizes the input text, converts it to an MLX array, and splits it into input and target sequences. The `[None]` adds a batch dimension, and `[:, :-1]` and `[:, 1:]` create overlapping sequences for next-token prediction.

```
def compute_loss(model, input_ids, target_ids):
    logits, _ = model(input_ids)
    return nn.losses.cross_entropy(logits, target_ids, reduction='mean')
```

8.3.3.2 Loss Computation This function computes the loss for a batch. It passes the input through the model to get logits, then calculates the cross-entropy loss between these logits and the target ids. The `reduction='mean'` argument ensures we get the average loss across all tokens.

```
model, processor = load(model_path)
model.freeze()
linear_to_lora_layers(model, lora_targets, lora_layers, lora_rank)
model.train()
```

8.3.3.3 Model Setup Here, we load the pre-trained model and its processor, freeze the base model parameters, apply LoRA to specified layers, and set the model to training mode. Freezing the base model ensures only the LoRA parameters are updated during training.

```
loss_and_grad_fn = nn.value_and_grad(model, compute_loss)
optimizer = optim.AdamW(learning_rate=learning_rate)
```

8.3.3.4 Optimization Setup We create a function that computes both the loss and its gradient with respect to the model parameters. We also initialize the AdamW optimizer with the specified learning rate.

```
for step in range(num_steps):
    input_ids, target_ids = prepare_batch(step)
    loss, gradients = loss_and_grad_fn(model, input_ids, target_ids)
    optimizer.update(model, gradients)
    mx.eval(model_state, loss)
```

8.3.3.5 Training Loop This loop prepares batches, computes loss and gradients, updates model parameters, and evaluates the model state and loss.

```
mx.save_safetensors(f'{adapter_path}/adapters.safetensors',
                    dict(tree_flatten(model.trainable_parameters())))
```

8.3.3.6 Saving LoRA Weights After training, we save only the trainable parameters (which are the LoRA weights) in the safetensors format.

8.4 Conclusion

LoRA training offers an efficient approach to adapting large language models like Phi-3 to specific tasks with minimal additional parameters. This tutorial provides an overview of LoRA and its implementation in MLX, highlighting key components and considerations. While simplified, it serves as a starting point for integrating LoRA into MLX-based model training pipelines.

In upcoming tutorials, we’ll explore practical ways to extend and apply language models like Phi-3. We’ll delve into topics such as implementing agent classes and toolchain systems, which allow for creating flexible AI workflows and chaining together different operations. These extensions will showcase how to build more versatile and powerful applications on top of the core language model capabilities we’ve discussed so far.

9 Part 8: Implementing the Agent Class and Toolchain System

9.1 Introduction

In this tutorial, we'll explore the implementation of the Agent class and its toolchain system in Phi-3-MLX. We'll break down the key components of the class and explain how the toolchain functionality is implemented.

9.2 The Agent Class Structure

Let's start by examining the core structure of the Agent class:

```
class Agent:
    _default_toolchain = """
        prompt = add_code(prompt, codes)
        responses = generate(prompt, images)
        files, codes = execute(responses, step)
        """

    def __init__(self, toolchain=None, enable_api=True, **kwargs):
        self.enable_api = enable_api
        self.kwargs = kwargs if 'preload' in kwargs else
         ↪  kwargs|{'preload':load(**kwargs)}
        self.set_toolchain(toolchain)
        self.reset()

    def __call__(self, prompt:str, images=None):
        # Implementation details
```

The class is designed with a default toolchain and an initializer that sets up the agent's configuration.

9.3 Toolchain Parsing

The `set_toolchain` method plays a key role in parsing and preparing toolchains for execution:

```
def set_toolchain(self, s):
    def _parse_toolchain(s):
        s = s.strip().rstrip(')')
        out_part, fxn_part = s.split('=')
        fxn_name, args_part = fxn_part.split('(')

        return {
            'fxn': eval(fxn_name.strip()),
            'args': [arg.strip() for arg in args_part.split(',')],
            'out': [out.strip() for out in out_part.split(',')]
        }

    def _parse_return(s):
        if 'return ' not in s:
            return ['responses', 'files']
        return [i.strip() for i in s.split('return ')[1].split(',')]

    s = self._default_toolchain if s is None else s
    self.toolchain = [_parse_toolchain(i) for i in s.split('\n') if '=' in i]
    self.list_outs = _parse_return(s)
```

This method does several important things:

1. It parses each line of the toolchain string into a dictionary.
2. It uses `eval` to convert function names into actual function references.
3. It extracts argument names and output variable names.
4. It determines the final outputs of the toolchain.

9.4 Executing the Toolchain

The `__call__` method is where the toolchain is actually executed:

```
def __call__(self, prompt:str, images=None):
    prompt = prompt.replace('"', '<|api_input|>') if self.enable_api else
↪   prompt
    self.ongoing.update({'prompt':prompt})
    if images is not None:
        self.ongoing.update({'images':images})
    for tool in self.toolchain:
        _returned = tool['fxn'](*[self.ongoing.get(i, None) for i in
↪   tool['args']],
                                  **{k:v for k,v in self.kwargs.items()
                                     if k in in-
                                     ↪   spect.signature(tool['fxn']).parameters.keys()})
        if isinstance(_returned, dict):
            self.ongoing.update({k:_returned[k] for k in tool['out']})
        else:
            self.ongoing.update({k:_returned for k in tool['out']})
    self.log_step()
    return {i:self.ongoing.get(i, None) for i in self.list_outs}
```

This method:

1. Prepares the input prompt and images.
2. Iterates through each tool in the toolchain.
3. Executes each function with the appropriate arguments.
4. Updates the ongoing state with the results of each function.
5. Logs the step and returns the final outputs.

9.5 Logging and State Management

The Agent class includes methods for managing its state and logging:

```
def reset(self):
    self.log = []
    self.ongoing = {'step':0}
    self.user_since = 0

def log_step(self):
    self.log.append({**self.ongoing})
```

```
        with open(f'agent_log.json', "w") as f:
            json.dump(self.log, f, indent=4)
        self.ongoing = {k:None if v==[None] else v for k,v in self.ongoing.items()}
        self.ongoing['step']+=1

    def end(self):
        self.ongoing.update({'END':'END'})
        self.log_step()
        self.reset()
```

These methods handle:

- Resetting the agent's state.
- Logging each step of the toolchain execution.
- Writing logs to a JSON file.
- Properly ending a session and preparing for the next one.

9.6 Key Implementation Insights

1. **Dynamic Function Calling**: The use of `eval` in parsing the toolchain allows for dynamic function calling, making the system highly flexible.
2. **State Management**: The `ongoing` dictionary acts as a state manager, passing data between different steps of the toolchain.
3. **Argument Matching**: The system dynamically matches function arguments with available data, allowing for flexible function definitions in the toolchain.
4. **Error Handling**: While not explicitly shown, proper error handling should be implemented, especially around the `eval` function and dynamic function calling.
5. **Extensibility**: The system is designed to be easily extended with new functions, as long as they follow the expected input/output pattern.

9.7 Conclusion

The Agent class implementation we've explored offers a way to chain together different AI operations. This approach can be useful for creating more complex AI workflows. With this system, you can:

1. Create custom toolchains that combine existing functions in different ways.
2. Add new functions to the system to expand its capabilities.
3. Manage and debug complex AI workflows more effectively.

This implementation provides a framework for experimenting with and building upon language models like Phi-3, allowing for more flexible and tailored AI applications.

10 Addendum: Extending MLX Porting Techniques to PaliGemma

10.1 Introduction

In our previous tutorials, we explored porting Phi-3-Vision to MLX. Now, let's extend these techniques to PaliGemma, a powerful open Vision-Language Model (VLM) developed by Google. PaliGemma combines the SigLIP-So400m vision encoder with the Gemma-2B language model, creating a versatile and knowledgeable base model for various tasks.

The full implementation of this tutorial is available at https://github.com/JosefAlbers/Phi-3-Vision-MLX/blob/main/assets/paligemma_dissected.py

10.2 Key Differences Relevant for Porting PaliGemma

1. Dual-Model Architecture:
   - PaliGemma uses separate vision (SigLIP-So400m) and language (Gemma-2B) models
   - We'll need to implement both components and their integration in MLX
2. Multimodal Projection:
   - A linear adapter connects the vision and language models
   - This needs to be implemented to properly combine visual and textual features
3. Full Block Attention:
   - PaliGemma processes both image tokens and text tokens in a single attention mechanism
   - Our MLX implementation must handle this combined attention approach
4. Input Processing:
   - Visual tokens are prepended to text input
   - We need to implement this specific input preparation in MLX
5. Multiple Resolutions:
   - Support for different input sizes (224x224, 448x448, 896x896)
   - Our implementation should be flexible to handle these variants
6. Task Prefixes:
   - PaliGemma uses task-specific prefixes for conditioning
   - We'll need to incorporate this feature in our input processing and model logic

These key differences will guide our porting process, ensuring we accurately translate PaliGemma's architecture to MLX while optimizing for Apple Silicon.

10.3 Porting PaliGemma to MLX

10.3.1 Core Components

Let's start by implementing the main components of PaliGemma:

```python
class PGemmaModel(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.vision_tower = VisionModel(config.vision_config)
        self.language_model = LanguageModel(config.text_config)
        self.multi_modal_projector = Projector(config)

class VisionModel(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.embeddings = VisionEmbeddings(config)
        self.layers = [EncoderLayer(config) for _ in
         ↪  range(config.num_hidden_layers)]
        self.post_layernorm = nn.LayerNorm(config.hidden_size)

    def __call__(self, x):
        x = self.embeddings(x)
        for l in self.layers:
            x = l(x)
        return self.post_layernorm(x[0])

class LanguageModel(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.scale = config.hidden_size**0.5
        self.embed_tokens = nn.Embedding(config.vocab_size, config.hidden_size)
        self.layers = [TransformerBlock(config=config) for _ in
         ↪  range(config.num_hidden_layers)]
        self.norm = RMSNorm(config)

    def __call__(self, input_ids, inputs_embeds=None, attention_mask_4d=None,
     ↪  cache=None):
        cache = [None] * len(self.layers) if cache is None else cache
        h = self.embed_tokens(input_ids) if inputs_embeds is None else
 ↪  inputs_embeds
        h = h * self.scale
        for e, layer in enumerate(self.layers):
            h, cache[e] = layer(h, attention_mask_4d, cache[e])
        return self.embed_tokens.as_linear(self.norm(h)), cache
```

10.3.2 Attention Mechanism

PaliGemma uses a different attention mechanism. Here's how we can implement it:

```
class Attention(nn.Module):
    def __init__(self, config):
        super().__init__()
        dims = config.hidden_size
        self.n_heads = n_heads = config.num_attention_heads
        head_dim = dims // n_heads
        self.scale = head_dim**-0.5
        self.q_proj = nn.Linear(dims, n_heads * head_dim, bias=config.attn_bias)
        self.k_proj = nn.Linear(dims, n_heads * head_dim, bias=config.attn_bias)
        self.v_proj = nn.Linear(dims, n_heads * head_dim, bias=config.attn_bias)
        self.o_proj = nn.Linear(n_heads * head_dim, dims, bias=config.attn_bias)
        if getattr(config, 'rope_base', False):
            self.rope = nn.RoPE(head_dim, base = config.rope_base)
        else:
            self.rope = lambda x, *args, **kwargs: x

    def __call__(self, x, mask=None, cache = None):
        B, L, _ = x.shape
        queries = self.q_proj(x).reshape(B, L, self.n_heads, -1).transpose(0,
↪   2, 1, 3)
        keys = self.k_proj(x).reshape(B, L, self.n_heads, -1).transpose(0, 2,
↪   1, 3)
        values = self.v_proj(x).reshape(B, L, self.n_heads, -1).transpose(0, 2,
↪   1, 3)

        queries = self.rope(queries)
        keys = self.rope(keys)

        if cache is not None:
            key_cache, value_cache = cache
            keys = mx.concatenate([key_cache, keys], axis=2)
            values = mx.concatenate([value_cache, values], axis=2)

        output = mx.fast.scaled_dot_product_attention(queries, keys, values,
↪   scale=self.scale, mask=mask)
        output = output.transpose(0, 2, 1, 3).reshape(B, L, -1)
        return self.o_proj(output), (keys, values)
```

10.3.3 Image Processing

PaliGemma handles image processing differently. Here's how we can implement it:

```
class VisionEmbeddings(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.patch_embedding = nn.Conv2d(in_channels=config.num_channels,
         ↪  out_channels=config.hidden_size, kernel_size=config.patch_size,
         ↪  stride=config.patch_size)
        self.num_patches = (config.image_size // config.patch_size) ** 2
        self.position_embedding = nn.Embedding(self.num_patches,
         ↪  config.hidden_size)

    def __call__(self, x: mx.array) -> mx.array:
        return mx.flatten(self.patch_embedding(x), start_axis=1, end_axis=2) +
         ↪  self.position_embedding(mx.arange(self.num_patches)[None, :])
```

10.3.4 Assembling Inputs

PaliGemma requires a specific way of assembling inputs:

```
def assemble(input_ids, inputs_embeds, image_features, attention_mask, config):
    inputs_embeds, image_features, attention_mask = [mx.array(i) for i in
↪   (inputs_embeds, image_features, attention_mask)]
    final_embedding = mx.zeros_like(inputs_embeds)
    text_mask = (input_ids != config.image_token_index) & (input_ids !=
↪   config.pad_token_id)
    text_mask_expanded = mx.repeat(mx.expand_dims(text_mask, -1),
↪   final_embedding.shape[-1], axis=-1)
```

```
    final_embedding = mx.where(text_mask_expanded, inputs_embeds,
↪   final_embedding)
    image_mask = input_ids == config.image_token_index
    image_mask_expanded = mx.repeat(mx.expand_dims(image_mask, -1),
↪   final_embedding.shape[-1], axis=-1)
    final_embedding = mx.where(image_mask_expanded, mx.pad(image_features,
↪   ((0,0), (0,input_ids.shape[1] - image_features.shape[1]), (0,0))),
↪   final_embedding)
    attention_mask_expanded = mx.expand_dims(attention_mask, (1, 2))
    final_attention_mask_4d = attention_mask_expanded *
↪   attention_mask_expanded.transpose(0, 1, 3, 2)
    mx.repeat(final_attention_mask_4d, config.text_config.num_key_value_heads,
↪   axis=1)
    return mx.array(final_embedding), mx.array(final_attention_mask_4d)
```

10.4 Using the Ported PaliGemma Model

Here's how we can use our ported PaliGemma model:

```
# Load model components
processor, language_model, vision_model, projector, config = load_parts()

# Process input
image_url = "https://huggingface.co/datasets/huggingface/documentation-
↪   images/resolve/main/transformers/tasks/car.jpg"
image = Image.open(requests.get(image_url, stream=True).raw)
processed = processor('Caption: ', image, return_tensors="np")
input_ids, pixel_values, attention_mask = [mx.array(processed[key]) for key in
↪   ["input_ids", "pixel_values", "attention_mask"]]

# Prepare inputs
inputs_embeds = language_model.embed_tokens(input_ids)
hidden_state = vision_model(pixel_values.transpose(0, 2, 3, 1))
image_features = projector(hidden_state[None]) / (config.hidden_size**0.5)
inputs_embeds, attention_mask_4d = assemble(input_ids, inputs_embeds,
↪   image_features, attention_mask, config)

# Generate output
logits, cache = language_model(input_ids, inputs_embeds, attention_mask_4d,
↪   None)
token = mx.argmax(logits[:, -1, :], axis=-1)
list_tokens = token.tolist()
for _ in range(100):
    logits, cache = language_model(token[None], None, None, cache)
    token = mx.argmax(logits[:, -1, :], axis=-1)
    list_tokens += token.tolist()
    if list_tokens[-1] == processor.tokenizer.eos_token_id:
        break

print(processor.tokenizer.decode(list_tokens))
```

10.5 Conclusion

This addendum demonstrates how the techniques we learned for porting Phi-3-Vision to MLX can be extended to other models like PaliGemma. While the specific implementations differ, the core principles of translating model architecture, attention mechanisms, and input processing remain the same. This flexibility allows us to adapt a wide range of models to run efficiently on Apple Silicon using MLX.