AdaPerceiver: Transformers with Adaptive Width, Depth, and Tokens

Supplementary Material

Table of Contents

A. Extended Related Work

We extend the related work presented in Sec. 2. In particular, we add some coverage of recent works in *recursive reasoning* models (or alternatively compute scaling models).

A.1. Adaptive Models

Adaptivity in deep learning has been explored through two main traditions: dynamic (*i.e.* conditional) neural networks (NNs) [2, 21], and elastic models [8, 13] — however there are recent work in recursive reasoning models have emerged [30, 47, 52, 52]. In our view, all three can be seen in a unified light and represent multiple paths prior work attempts to reach a common (often implicitly stated) goal.

Dynamic Neural Networks. Dynamic neural networks adapt computation on a per-input basis, allocating compute or parameters depending on the difficulty or content of the input. Approaches include early-exiting strategies [36, 38, 42, 46, 51] and pruning techniques [7, 16, 35, 48, 49]. These methods generally either use a heuristic or learn a notion of "importance" to decide whether to execute an additional layer (adaptive depth or early-exit), drop tokens (token pruning), or mask features.

Elastic Models. In contrast, the elastic model tradition focuses on training a *single model* that can be executed at multiple capacities under user-defined compute budgets [9, 13, 19, 23, 40]. Early work such as Once-for-All networks [8] demonstrated that convolutional networks can be trained to support a set of sub-networks that trade accuracy for efficiency at inference time. Subsequent work extends this idea to Transformer architectures, enabling flexible inference across 1–2 dimensions: tokens, depth, or width. Width-adaptive models such as MatFormer, HydraViT, and Flextron [9, 13, 19, 40] train shared-weight sub-networks that operate at varying hidden dimensions, while DynaBERT and SortedNet [23, 40] explore joint width–depth adaptivity. Token-adaptivity has been studied in FlexiViT [5], which supports varying patch sizes at inference—and thus token counts—within a single model.

Existing training strategies for these models are either costly (relying on multiple forward-passes per configuration [13]) or noisy (stochastic training approaches that sample configurations [5, 9, 19, 40]).

Recursive Reasoning Models. Compared with elastic models and (some) dynamic neural networks, recursive reasoning models [17, 30, 47, 52] (and the older Universal Transformers [12]) recur over a core architectural block to solve predictive tasks. This is meant as a means of scaling "test-time compute" [18], and often is coupled with some halting condition [52].

We refrained from introducing this work in the main body because although many of these models could be considered adaptive, they are not necessarily comparable to dynamic neural networks and elastic models. Elastic models *budget* the compute of a model, whereas these reasoning models *expand* the base compute. In abstract, if a model expends a single compute unit, then elastic methods attempt to partition that unit to expose models capable of expending various amounts of compute (up to a maximum). Whereas recursive reasoning models can repeatedly expend this unit of compute.

Comparison to Our Work. AdaPerceiver combines elements of both dynamic neural networks and elastic model. Like dynamic neural networks, it supports per-input adaptivity: configurations can be selected at runtime, *e.g.* by a learned policy (see Sec. 4.5). Similar to elastic models and reasoning models, we train a single shared-weight model to support flexible configurations. However, unlike prior elastic models, AdaPerceiver supports simultaneous adaptivity across token, depth, and width axes. For our novel training approach, we structure the network such that multiple configurations can be *jointly optimized within a single forward pass*. Training AdaPerceiver does not require multiple forward evaluations, with less reliance on stochastic configuration sampling.

A.2. Perceiver Architectures

Perceiver architectures follow an *encode-process-decode* paradigm: inputs are *encoded* via attention into a fixed set of latent tokens (the latent stream); this latent stream is *processed* through iterative transformer layers; and finally *decoded* to produce outputs. The original Perceiver introduced a fixed-size latent stream that decoupled input size from internal computation, enabling scalability to large and multi-modal data [26]. PerceiverIO extended this

idea by introducing an output query mechanism, allowing latent representations to be decoded into arbitrarily sized outputs [25]. Subsequent variants further developed this direction. PerceiverAR [22] adapted the architecture for autoregressive modeling, while the Hierarchical Perceiver (HiP) [10] incorporated locality and hierarchical structure to improve efficiency while maintaining generality.

Comparison to Our Work. Prior work on the Perceiver family studies generality and scalability across modalities. Their latent processing streams are fixed once trained. We introduce adaptivity into this latent stream, enabling control over the amount of computation allocated to each input.

B. Why *not* FlexiViT for token adaptivity?

FlexiViT is a plausible means of achieving token adaptivity, and one may naturally ask why a Perceiver-style architecture is required. In principle, one could combine early exiting, Matryoshka learning, and FlexiViT within a standard ViT and obtain adaptivity along all three axes. We provide a summary of FlexiViT in Appendix B.1, and our reasons for not using it Appendix B.2.

B.1. Summary of FlexiViT

FlexiViT achieves token adaptivity by varying the patch size used to encode the input. Smaller patch sizes yield more tokens (and thus more compute), whereas larger patch sizes yield fewer tokens (and lower compute). During training, FlexiViT samples patch sizes uniformly on a per-batch basis, allowing the patch size—and by extension the compute budget—to be adjusted at inference.

Compared to AdaPerceiver, FlexiViT achieves token adaptivity by varying the patch size, whereas AdaPerceiver does so by changing the number of tokens in the latent stream (*cf.* Appendices A.2 and C and Sec. 3.2).

B.2. Reasons not to use FlexiViT

In this work, we do not use FlexiViT for two reasons.

Limited Control Over Token Count. FlexiViT does not support arbitrary token counts. The number of tokens is determined jointly by the input resolution and the patch size. Thus, changing the patch size does *not* guarantee the same number of tokens — nor the same amount of compute — across different input sizes. In contrast, AdaPerceiver can map any input resolution to an arbitrary number of latent tokens. Moreover, it supports interpolation and extrapolation beyond the token granularities used during training (Figs. 10, 12 and 14). We note that this flexibility is a design trade-off: in some settings, scaling compute with input resolution is desirable. AdaPerceiver does not automatically scale compute with increasing input resolution, as the number of latents is controlled independently from input size.

Input and Output Token Counts Are Coupled. A more subtle limitation concerns dense prediction. In FlexiViT, the number of input tokens equal to the number of output tokens: reducing input tokens necessarily reduces output tokens. However, token count is known to strongly influence dense prediction performance [41]. Ideally, one would like to process *fewer* tokens (for efficiency) while still producing *more* output tokens (for predictive performance).

AdaPerceiver supports this decoupling: it can process a lower number of latent tokens while decoding to a higher number of output tokens. Our results suggest that processing fewer tokens while outputting more can yield performance comparable to processing more tokens (*cf.* Tabs. 2 and 3), though further investigation is needed.

If this effect is real, then FlexiViT fundamentally lacks the ability to exploit it, as its output token count is inherently tied to its input token count.

C. Architectural Details

We elaborate upon our architecture details in Sec. 3.2.

C.1. Notation

We denote by $x \in \mathbb{R}^{I \times d_i}$ the sequence of I input tokens with embedding dimension d_i, by $z_l \in \mathbb{R}^{N \times d_z}$ the N latent tokens at layer $l \in [0, L]$ with embedding dimension d_z, and by $o \in \mathbb{R}^{M \times d_o}$ the M output tokens with embedding dimension d_o. The learned latent token is denoted $z \in \mathbb{R}^{1 \times d_z}$, and z_0 is the initial latent array obtained after broadcasting and reading from the input.

We define three sets: $\mathcal{T}$ for token granularities, $\mathcal{W}$ for width configurations, and $\mathcal{D}$ for depths. Each adaptive subnetwork is indexed by a tuple (t, w, l) where $t \in \mathcal{T}$, $w \in \mathcal{W}$, and $l \in \mathcal{D}$.

C.2. AdaPerceiver Architecture

AdaPerceiver follows the *encode–process–decode* paradigm of Perceiver (*cf.* Fig. 1).

Encode. We first broadcast the learned latent token z to N latents, producing $z' \in \mathbb{R}^{N \times d_z}$:

$$z' = \texttt{Broadcast}(z,\, N) = [z, z, \ldots, z] \in \mathbb{R}^{N \times d_z}. \quad (8)$$

We then read the input tokens into this broadcast latents using cross-attention, treating z' as sink (query) tokens and the input tokens x as source (key/value) tokens:

$$z_0 = z' + \texttt{CrossAttention}(z',\, x). \quad (9)$$

We apply RoPE [37] to the sink tokens z' to positionally distinguish the tokens.

The *encode* step assumes we are given input tokens x. In practice, the input tokens are modality specific. For images, they may be patches or features of a network. We elaborate further in Appendix C.5.

Process. Following the encode step, the latents z_0 are refined using a sequence of AdaPerceiver blocks:

$$z_l = \texttt{AdaPerceiverBlock}(z_{l-1}), \qquad l \in [1, L]. \quad (10)$$

Each `AdaPerceiverBlock` has the same structure as ViT blocks:

$$z'_l = z_{l-1} + \texttt{BlockMaskAttention}(\texttt{Norm}(z_{l-1})), \quad (11)$$

$$z_l = z'_l + \texttt{MatFFN}(\texttt{Norm}(z'_l)). \quad (12)$$

We use `Norm` to denote LayerNorm. `BlockMaskAttention` refers to multi-head attention with RoPE applied to the queries and keys and block attention mask. `MatFFN` denotes the feed-forward network with Matryoshka linear layers (`MatLinear`); a minimal `MatLinear` implementation is shown in Algorithm 2.

Decode. Finally, the latent tokens are decoded (read out) to output tokens. This step is nearly identical to the *Encode* step. Given a set of output tokens, o, the latent tokens are read to the output tokens using cross-attention, treating the o as the sink (query) tokens and the latent tokens z_l as the source (key/value):

$$o = o' + \texttt{CrossAttention}(o',\, z_l). \quad (13)$$

Where o' are the initial output tokens. We elaborate on how the output tokens can be initialized for classification and dense prediction tasks in Appendix C.6

C.3. Designing the Latent Token(s)

As noted in Sec. 3.2.2, various choices are available for the latent tokens z. We outline two options below.

Learned Latent Array. PerceiverIO learns N latent tokens equal to the size of their latent stream [25, 26]. This works well for fixed latent arrays (as in PerceiverIO) but does not translate well when we may want the latent stream to be adaptive — in such cases one could up-sample the latent array when requiring $> N$ tokens or down-sample when wanting $< N$. However, this is simply more complicated than learning a single token broadcast to N, then applying RoPE (as we do in Sec. 3.2.2).

Randomly Initialized Latents. A recent work by Geiping *et al*. follows the *encode-process-decode* scheme of Perceiver models, in their case they initialize their latent tokens from a normal distribution $\mathcal{N}(0, \sigma I)$ [17]. Such a scheme can be used in lieu of learning a single token by just sampling a token vector from $\mathcal{N}(0, \sigma I)$, or an array of latents can be sampled. We mention this for sake of completeness but did not study it further.

C.4. Why block masking allows for adaptive tokens?

To understand why block masking enables adaptive token granularities, it is useful to examine the attention layer within the `AdaPerceiverBlock`, as this is the only component that performs sequence-level mixing (*c.f.,* [24]). In standard attention, the layer forms a mixing matrix $A \in \mathbb{R}^{N\times N}$ from the queries and keys $Q, K \in \mathbb{R}^{N\times d}$, and applies it to the values $V \in \mathbb{R}^{N\times d}$:

$$Y = AV,$$

so that each output token y_i is a weighted combination of all value tokens:

$$y_i = \sum_{j=1}^{N} a_{ij} v_j.$$

With block masking, we constrain which tokens may interact during attention. Specifically, a token may only mix with tokens from its own granularity and with those from *smaller* granularities. For example, consider training an AdaPerceiver model with two token granularities $\mathcal{T} = \{2, 4\}$. The block mask enforces:

$$y_i = \begin{cases} \sum_{j=1}^{2} a_{ij} v_j, & i \in \{1, 2\}, \\ \sum_{j=1}^{4} a_{ij} v_j, & i \in \{3, 4\}. \end{cases}$$

Thus, the first two output tokens depend exclusively on the first two input tokens, while the last two depend on all four. Because the first two outputs only result from a mixing of the first two inputs, their computation is *identical* to the computation the model would perform if the sequence length were actually two. In general, block masking ensures that the computation associated with any granularity depends only on the tokens belonging to that granularity and the granularities smaller than it. Consequently, adding additional tokens (granularities) does not alter the computation of earlier ones, and supervising each granularity during training is therefore equivalent to training the model with multiple numbers of latent tokens.

C.5. Input Tokens

We obtain input tokens using the standard patch embedding used in Vision Transformers [15]. Other choices are possible—for example, a smaller pre-trained model or convolutional stem can also be used to produce the input token sequence.

C.6. Output Tokens

We describe how output tokens are instantiated for both classification and dense prediction tasks.

Classification. For classification, we simply learn a single output token.

Dense Prediction. For dense prediction tasks, we consider two cases. If the number of output tokens is known *a priori*, we can directly learn that number of output tokens. However, when the number of output tokens is unknown or

should scale with the input resolution, we initialize the output tokens from the input tokens themselves. In our work, we adopt this latter approach: the output tokens are obtained by applying a learned linear projection to the input tokens.

This construction is beneficial for dense prediction (and feature distillation) because the number of output tokens automatically grows with the number of input tokens (e.g., when increasing image resolution). As a result, the model exhibits shape behaviour similar to a traditional ViT. Furthermore, this design enables variable-resolution training without having to re-learn or interpolate a fixed set of output latents (a common trick in ViT training).

C.7. Model Details

We summarize the key architectural hyperparameters used in our AdaPerceiver model in the table below.

Model Configuration	
$\mathcal{W}$	$\{416, 624, 832\}$
$\mathcal{T}$	$\{32, 64, 96, 128, 192, 256\}$
$\mathcal{D}$	$\{1, 2, \ldots, 21\}$
Input Adapter	Patch Embed.
Image Size	224
In Channels	3
Patch Size	14
Embed FFN	✓
Embed. Dim	832
FFN Ratio	2.57
Heads	13
Depth	21
Max Latent Tokens	256
RoPE Theta	10000
QKV Bias	✓
Proj. Bias	✓
Layer Scale Init	1.0×10^{-5}
FFN Activation	GeLU
Norm Layer	LayerNorm
FFN Layer	MLP

D. Training Details

We outline our pre-training and fine-tuning details below.

D.1. Pre-training/Distillation

We summarize our training setting below. We first train solely using logit distillation, and then have a subsequent feature distillation stage for dense prediction tasks. Our teacher model is the ViT-H/14 CLIP model fine-tuned on ImageNet-12k as the teacher, trained in [11] and publicly available in [43]. In both cases we conduct distillation with the ImageNet-12K dataset [44]. We use the same augmentation settings for both logit and feature distillation:

Augmentations	
Image Size	224
Horiz. flip	✓
RandAugment	✓
RandAug Ops	2
RandAug Magnitude	15
Mixup	✓
Mixup α	1.0

Logit Distillation. We base our distillation recipe on [4]. We train in three stages, we first train adaptivity over the *token* dimension, then jointly over *token* and *depth*, and finally over all three dimensions. At the beginning of each stage *we initialize the model weights with the EMA weights from the prior stage*. We use the following optimization hyper-parameters:

	Stage 1	Stage 2	Stage 3
Effective Batch Size	4096		
Optimizer	Shampoo-Soap		
Learning Rate	1×10^{-3}	1×10^{-3}	5×10^{-4}
Weight Decay	0.003		
Betas	(0.9, 0.999)		
Grad Clip	3		
EMA Decay	0.999	0.999	0.9998
Epochs	50	65	20
Schedule	Cosine		–
Warmup Steps	3000		
Warmup LR	1×10^{-6}		
Min LR	1×10^{-5}		–
Precond. Frequency	100		
Max Precond. Dim	8,192		
Start Precond. Step	500		

For each stage we use the following settings for our loss functions:

	Stage 1	Stage 2	Stage 3
Token Loss	✓	✓	✓
Depth Loss	–	✓	✓
Width Loss	–	–	✓
Depth Loss Schedule	–	linear	linear

N.B. We weight the loss from earlier depths lower than that from later depths and linearly increase the weights: the contribution from depth 1 has weight $1/21$, while the contribution from depth 21 has weight 1.0.

Feature Distillation. For feature distillation we configure our model as dense prediction task and attach a MLP

adapter to the output tokens to predict the features our teacher. We base our feature distillation recipe on [34]. Specifically, we use both the cosine similarity loss ($\mathcal{L}_{\text{cos}}$) and smooth L1 magnitude loss ($\mathcal{L}_{\text{norm}}$):

$$\mathcal{L}_{\text{feat}} = w_{\text{cos}}\mathcal{L}_{\text{cos}} + w_{\text{norm}}\mathcal{L}_{\text{norm}}$$

We initialize our model using the *Stage 3* weights. We use the following optimization hyper-parameters:

	Feature Distillation
Effective Batch Size	4096
Optimizer	Shampoo-Soap
Learning Rate	5×10^{-4}
Weight Decay	0.003
Betas	$(0.9, 0.999)$
Grad Clip	3
EMA Decay	0.9995
Epochs	20
Schedule	Cosine
Warmup Steps	3000
Warmup LR	1×10^{-6}
Min LR	1×10^{-5}
Precond. Frequency	30
Max Precond. Dim	8192
Start Precond. Step	500

We use the following parameters for our loss:

	Feature Distillation
Token Loss	✓
Depth Loss	✓
Width Loss	✓
Depth Loss Schedule	linear
w_{cos}	0.9
w_{norm}	0.1

D.2. ImageNet-1K Classification Fine-Tuning

For ImageNet-1K fine-tuning, we fine-tune the write head (cross-attention to output tokens), output tokens, and a final linear projection layer to project the output token to predict 1000 classes. We use the following data augmentations:

We use the same loss functions as Distillation *Stage 3* and use the following optimization hyper-parameters:

D.3. ADE20K Semantic Segmentation Fine-Tuning

We fine-tune the write head (cross-attention to output tokens), output tokens, and a final linear projection layer. We only use the token loss and disable adaptivity training for width and depth. We train using the following optimization hyper-parameters:

Augmentations	
Image Size	224
Horiz. flip	✓
RandAugment	✓
RandAug Ops	2
RandAug Magnitude	20
Mixup	✓
Mixup α	0.8
CutMix	✓
CutMix α	0.5
Random Erasing	✓
Erase Prob.	0.25

	IN-1K Fine-Tuning
Effective Batch Size	1024
Optimizer	Shampoo-Soap
Learning Rate	5×10^{-4}
Weight Decay	0.003
Betas	$(0.9, 0.999)$
Grad Clip	3
EMA Decay	0.9995
Epochs	70
Schedule	Cosine
Warmup Steps	500
Warmup LR	1×10^{-6}
Min LR	1×10^{-5}
Precond. Frequency	30
Max Precond. Dim	8192
Start Precond. Step	250

	ADE20K Fine-Tuning
Effective Batch Size	16
Total Steps	$50,530$
Optimizer	Shampoo-Soap
Learning Rate	5×10^{-4}
Weight Decay	0.1
Betas	$(0.9, 0.999)$
Grad Clip	3
EMA Decay	0.9995
Schedule	Poly.
Poly. Power	1
Warmup Steps	1500
Warmup LR	1×10^{-6}
Min LR	1×10^{-6}
Precond. Frequency	30
Max Precond. Dim	8192
Start Precond. Step	500

D.4. NYUv2 Depth Estimation Fine-Tuning

We fine-tune the write head (cross-attention to output tokens), output tokens, and a final linear projection layer. We only use the token loss and disable adaptivity training for width and depth. We train using the following optimization

hyper-parameters:

	Depth Fine-Tuning
Effective Batch Size	16
Total Steps	$47,584$
Optimizer	NAdamW
Learning Rate	1×10^{-4}
Weight Decay	0.1
Betas	$(0.9, 0.999)$
Grad Clip	3
EMA Decay	0.9995
Schedule	Poly.
Poly. Power	1
Warmup Steps	1500
Warmup LR	1×10^{-6}
Min LR	1×10^{-6}

E. Pseudocode

Here, we introduce PyTorch-esque code for our implementation of Matyroshka linear layers (Algorithm 2) and the AdaPerceiver training regime (Algorithm 3).

Algorithm 2 Matryoshka Linear Layer with per-sample masking.

Notes: *Each sample i uses a different embedding dimension w_i. The layer either masks inputs (`mat_input=True`) or outputs (`False`) before or after the linear projection. During test-time we do not need to rely on masking and can just slice the weight matrices as per [13].*

```
class MatLinear(nn.Linear):
  def forward(self, x, mat_dim, mat_input=False):
    # x: (B, T, in_dim); mat_dim: (B,) adaptive width per sample
    B, T, in_dim = x.shape
    out_dim = self.weight.shape[0]
    mat_dim = mat_dim.to(torch.long, device=x.device)

    if mat_input:
      # Mask input features before projection
      col_idx = torch.arange(in_dim, device=x.device)
      mask = (col_idx.unsqueeze(0) < mat_dim.unsqueeze(1)).unsqueeze(1) # (B,1,in)
      x = x * mask.to(x.dtype)
      y = F.linear(x, self.weight, self.bias) # (B,T,out)
    else:
      # Projection, then mask outputs
      y = F.linear(x, self.weight, self.bias) # (B,T,out)
      row_idx = torch.arange(out_dim, device=x.device)
      mask = (row_idx.unsqueeze(0) < mat_dim.unsqueeze(1)) # (B,out)
      y = y * mask.unsqueeze(1).to(y.dtype) # (B,T,out)
    return y
```

F. Image Classification Results

We include additional image classification results. Tab. 5 includes an extended comparison with both adaptive architectures and various ViT models. Fig. 8 illustrates the data in Fig. 2 with alongside the bi-directional attention variant of AdaPerciever. We note that the improvements in throughput with the bi-directional variant are likely due to differences in attention implementation (FlexAttention vs. `scaled_dot_product_attention`). Fig. 9 illustrates how scaling depth and tokens offers different trade-offs; particularly that scaling tokens has minor accuracy degradation for significant latency benefits.

Algorithm 3 AdaPerceiver Training.

```
width_choices = [...]
latent_token_grans = [...] # the last entry corresponds to the max latents used during training
mask = create_block_mask(latent_token_grans) # creates structured mask

class AdaPerceiver(...):
  def forward_training(x, mask, widths):
    latents = ... # [B, N], N corresponds to the maximum latents tokens used during training.
    output_tokens = ... # [B, M], # M corresponds to the output tokens

    # cross attention from input to latents.
    # latents are the sink (Q), x is the source (K, V).
    latents = cross_attention(latents, x)

    # apply adaperceiver blocks to latents
    # we provide the mask and per sample width
    final_latents, intermediate_latents = forward_blocks(latents, mask, widths)

    output_list = []
    intermediate_output_list = []
    for token_gran in latent_token_grans:
      # We select the first token_gran tokens and read them out.
      sliced_latents = latents[:, :token_gran]
      # cross attention from latents to output.
      # output_tokens are the sink (Q), sliced_latents are the source (K, V).
      token_gran_output = cross_attention(output_tokens, sliced_latents)
      output_list.append(token_gran_output)

    for int_latent in intermediate_latents:
      # we sample a token granularity and slice the latents.
      token_gran = sample(latent_token_grans)
      sliced_latents = latents[:, :token_gran]
      # cross attention from latents to output.
      # output_tokens are the sink (Q), sliced_latents are the source (K, V).
      token_gran_output = cross_attention(output_tokens, sliced_latents)
      intermediate_output_list.append(token_gran_output)
    return output_list, intermediate_output_list

model = AdaPerceiver(...)
for x, y in dataloader:
  B = x.shape[0] # get the batch size
  # Sample width for each sample in the batch
  widths = [sample(width_choices) for _ in range(B)]

  # Forward and backward pass
  output_list, int_output_list = model.forward_training(x, mask, widths)

  # token loss
  token_loss = loss_fn(output_list, y)
  # layer_loss
  layer_loss = loss_fn(int_output_list, y)
  loss = token_loss + layer_loss
  loss.backward()
  ...
```

G. Dense Prediction Results

We present additional results from Sec. 4.3. Appendix G.1 contains additional segmentation results and Appendix G.2 contains additional depth estimation results.

G.1. Semantic Segmentation

We include extended experiments on semantic segmentation. Fig. 11 shows the relationship between GFLOPs and mIoU across depth and tokens, increasing depth improve mIoU monotonically with increasing computational costs. Fig. 12 illustrates how extrapolating beyond training-time configurations affects mIoU and GFLOPs; extrapolation degrades mIoU, however interpolation retains performance between training points.

G.2. Depth Estimation

We include extended experiments on depth estimation. Fig. 13 shows the relationship between GFLOPs and RMSE across depth and tokens, increasing depth improve mIoU monotonically with increasing computational costs. Fig. 12 illustrates how extrapolating beyond training-time configurations affects RMSE and GFLOPs; extrapolation degrades RMSE, however interpolation retains performance between training points.

H. Feature Visualizations

We include principal component analyses (PCA) of AdaPerceiver's patch features. Fig. 15 visualizes principal components as embedding dimension is modulated. Fig. 16 visualizes principal components as the number of tokens is changed, even extrapolated beyond training length. Fig. 17 depicts how depth affects principal components, showing that semantic features emerge at different depths, depending on the input image.

I. Policies for Adaptivity

Recall, that AdaPerceiver exposes a large space of valid configurations across tokens, depth, and width but does not prescribe which configuration should be used. We study the effect of different policies in Sec. 4.5, we elaborate on those policies here.

I.1. Baseline Policy

To understand the effect of a using a fixed configuration for all inputs, we study a "Baseline" policy. This policy is *input-independent*. We select a fixed configuration (number of tokens t, width w and depth l) for all inputs.

I.2. Early Exit (EE) Policy

To understand the effect of using a simple adaptivity method (early-exiting) in conjunction with a fixed configuration, we study an early-exit policy. This policy augments the baseline policy; rather than selecting a specific depth, an early-exit threshold is selected. The early-exit threshold, τ, is the threshold which the confidence of a prediction must exceed to exit early [29]. During the forward pass the latent tokens are read out, and if the prediction confidence exceeds τ, we exit. We are able to implement this *without* any further training of our model.

Algorithm 4 Policy Network

```
class PolicyNetwork(nn.Module):
   def __init__(self, dim, seq_len, token_choices):
      super(PolicyNetwork, self).__init__()
      self.dim = dim
      self.token_choices = token_choices

      self.mixer_block = MixerBlock(dim=dim, seq_len=
          seq_len)
      self.mixer_block_2 = MixerBlock(dim=dim, seq_len=
          seq_len)

      # Small fusion MLP after pooling
      self.fuse = nn.Sequential(
         nn.LayerNorm(dim),
         nn.Linear(dim, dim),
         nn.GELU(),
         nn.Linear(dim, dim),
         nn.GELU(),
      )

      self.head_tokens = nn.Linear(dim, len(
          token_choices), bias=False)

   def forward(self, x):
      x = self.mixer_block(x)
      x = self.mixer_block_2(x)
      x = x.mean(dim=1)

      h = self.fuse(x)

      logits_tokens = self.head_tokens(h)
      return logits_tokens
```

I.3. RL Policy

We train a lightweight policy network using REINFORCE [45] to select a token count for an input. Our policy network definition is shown in Algorithm 4, it consists of MLP-Mixer Block [39] and operates on the outputs of the Patch Embedding layer of AdaPerceiver, *i.e.* the input tokens.

We define a discrete action space over $\mathcal{T}$, the token granularities, and our goal is to learn a policy $\pi(t \mid x)$, that associates a token granularity with a given input. For our reward we use the negative cross-entropy as our reward a with computational cost term:

$$R(y, \hat{y}, t) = -\text{CrossEntropy}(\hat{y}, y) - \lambda \text{Cost}(t). \quad (14)$$

Where, y is the ground-truth label, $\hat{y}$ is the predicted label, and λ controls the trade-off between accuracy and computational cost. Rather than directly measuring computational, we use a proxy, since computation cost increases monotonically with token count, we increase cost linear with index, *e.g.* if $\mathcal{T} = \{4, 8\}$, then 4 would have cost 0 and 8 would have cost 1. Finally, to reduce the variance of REINFORCE, we use the EMA of previous rewards as a baseline.

I.4. Optimal Policy

To characterize the theoretical upper bound on performance we define an oracle "optimal" policy. Given a trained

model, this policy chooses, for each input, the configuration with the least compute that still yields a correct classification. We perform a grid-search across configurations for each input on the ImageNet-1K validation split, which gives us oracle-like behaviour. During this search, we record the *minimal compute* configuration that will yield a correct classification — when running the policy we look-up the minimal configuration for the given input. This serves as an oracle on ImageNet-1K to help characterize the theoretical upper-bound performance our trained AdaPerceiver model can achieve on this task.

Table 5. **ImageNet-1K Cross-Model Evaluation.** Comparison of Vision Transformer (ViT) variants on ImageNet-1K. Metrics include classification accuracy, inference latency (mean per forward pass), and computational cost in GFLOPs for both forward and backward passes. Latency measured at batch size of 512.

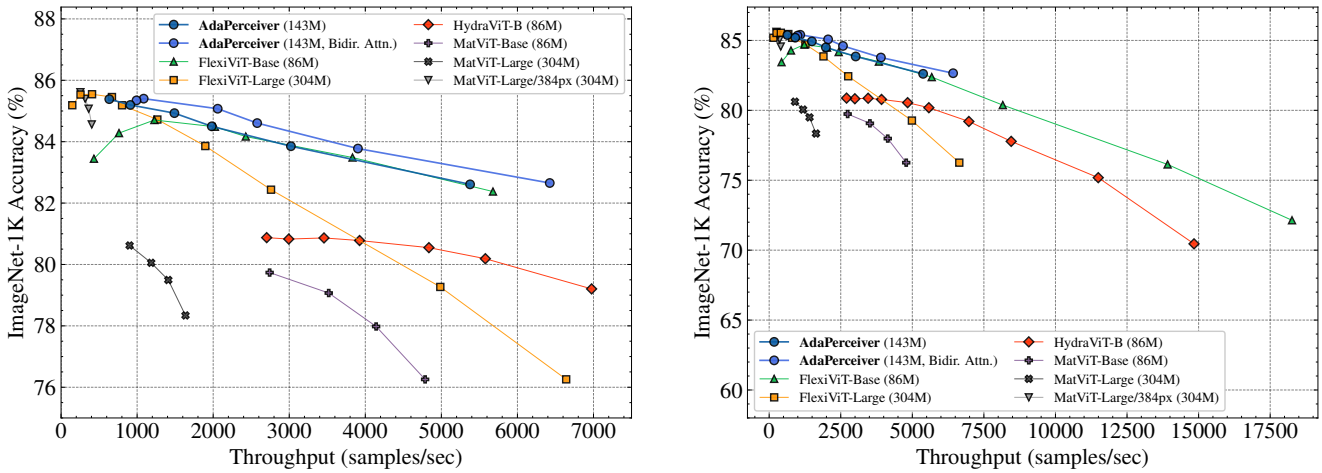
Model	**Params (M)**	**Accuracy (%)**	**Latency (ms)**	**Fwd GFLOPs**	**Bwd GFLOPs**
DeiT-Ti/16	5.7	68.96	29.4	6.4	12.7
DeiT-S/16	22.1	78.21	66.3	25.3	50.4
DeiT-B/16	86.6	80.79	177.1	100.9	201.3
DeiT3-S/16	22.1	80.83	66.2	25.3	50.4
DeiT3-B/16	86.6	83.22	176.2	100.9	201.3
DeiT3-L/16	304.4	84.23	545.5	357.6	714.5
ViT-H/14	632.0	87.11	1504.8	970.9	1941.1
SoViT-150M/16	136.1	87.27	403.2	207.6	414.5
MatViT-B ($w = 96$)	86.6	76.3	107.0	49.4	98.2
MatViT-B ($w = 192$)	86.6	77.9	123.6	59.1	117.7
MatViT-B ($w = 384$)	86.6	79.1	145.5	78.7	156.8
MatViT-B ($w = 768$)	86.6	79.7	186.7	117.7	234.8
MatViT-L ($w = 128$)	304.3	78.3	313.0	174.3	347.8
MatViT-L ($w = 256$)	304.3	79.5	363.1	209.0	417.2
MatViT-L ($w = 512$)	304.3	80.1	432.1	278.4	556.0
MatViT-L ($w = 1024$)	304.3	80.6	567.9	417.2	833.7
MatViT-L/384px ($w = 128$)	304.7	84.6	1268.4	510.6	1018.7
MatViT-L/384px ($w = 256$)	304.7	85.1	1411.2	612.1	1222.0
MatViT-L/384px ($w = 512$)	304.7	85.4	1611.5	815.4	1628.6
MatViT-L/384px ($w = 1024$)	304.7	85.6	2013.4	1222.0	2441.8
HydraViT ($w = 192$)	86.6	70.5	34.7	4.27	8.45
HydraViT ($w = 256$)	86.6	75.2	44.8	7.55	14.99
HydraViT ($w = 320$)	86.6	77.8	60.8	11.76	23.38
HydraViT ($w = 384$)	86.6	79.2	73.9	16.91	33.64
HydraViT ($w = 448$)	86.6	80.2	92.0	22.98	45.75
HydraViT ($w = 512$)	86.6	80.5	106.3	29.98	59.73
HydraViT ($w = 576$)	86.6	80.8	131.0	37.91	75.56
HydraViT ($w = 640$)	86.6	80.9	148.4	46.77	93.25
HydraViT ($w = 704$)	86.6	80.8	171.7	56.56	112.80
HydraViT ($w = 768$)	86.6	80.9	190.3	67.28	134.21

Table 5. **(Continued) ImageNet-1K Cross-Model Evaluation.**

Model	Params (M)	Accuracy (%)	Latency (ms)	Fwd GFLOPs	Bwd GFLOPs
FlexViT-B ($ps = 48$)	91.2	72.1	28.0	13.8	27.0
FlexViT-B ($ps = 40$)	89.5	76.1	36.8	19.4	38.3
FlexViT-B ($ps = 30$)	87.9	80.4	62.7	33.7	66.8
FlexViT-B ($ps = 24$)	87.2	82.4	90.2	52.0	103.5
FlexViT-B ($ps = 20$)	86.9	83.5	133.6	74.4	148.3
FlexViT-B ($ps = 16$)	86.6	84.2	210.9	115.7	230.9
FlexViT-B ($ps = 15$)	86.5	84.5	253.2	131.5	262.5
FlexViT-B ($ps = 12$)	86.5	84.7	417.3	204.9	409.2
FlexViT-B ($ps = 10$)	86.5	84.3	672.1	294.6	588.6
FlexViT-B ($ps = 8$)	86.7	83.4	1185.2	459.7	918.9
FlexiViT-L ($ps = 48$)	310.4	76.3	77.1	47.8	94.9
FlexiViT-L ($ps = 40$)	308.3	79.3	103	67.8	134
FlexiViT-L ($ps = 30$)	306.2	82.4	185	118	236
FlexiViT-L ($ps = 24$)	305.2	83.9	270	184	367
FlexiViT-L ($ps = 20$)	304.7	84.7	404	263	526
FlexiViT-L ($ps = 16$)	304.4	85.2	638	410	819
FlexiViT-L ($ps = 15$)	304.4	85.4	765	466	932
FlexiViT-L ($ps = 12$)	304.1	85.5	1257	727	1453
FlexiViT-L ($ps = 10$)	304.2	85.5	2001	1046	2091
FlexiViT-L ($ps = 8$)	304.4	85.2	3469	1633	3266
AdaPerceiver ($t = 32$)	143.8	82.6	95.2	16.2	44.1
AdaPerceiver ($t = 64$)	143.8	83.9	169.4	28.3	76.8
AdaPerceiver ($t = 96$)	143.8	84.5	258.4	40.4	109.6
AdaPerceiver ($t = 128$)	143.8	84.9	343.6	52.5	142.4
AdaPerceiver ($t = 192$)	143.8	85.2	562.3	88.7	240.7
AdaPerceiver ($t = 256$)	143.8	85.4	807.4	100.8	273.5
AdaPerceiver, Bidir. ($t = 32$)	143.8	82.6	79.7	16.2	44.1
AdaPerceiver, Bidir. ($t = 64$)	143.8	83.7	131.2	28.3	76.8
AdaPerceiver, Bidir. ($t = 96$)	143.8	84.6	198.6	40.4	109.6
AdaPerceiver, Bidir. ($t = 128$)	143.8	85.0	248.6	52.5	142.4
AdaPerceiver, Bidir. ($t = 192$)	143.8	85.4	471.3	88.7	240.7
AdaPerceiver, Bidir. ($t = 256$)	143.8	85.3	516.0	100.8	273.5

Table 6. **Adaptivity Policy Evaluation.** Accuracy and computational cost (GFLOPs) for configuration selection policies applied to AdaPerceiver on image classification. Utilizing early-exiting often acts as a "free-lunch" allowing for the reduction in compute costs with little to no degradation in accuracy. *N.B.* The "Optimal" policy is only theoretical and impractical to realize.

Policy	**Accuracy** (%) ↑	**GFLOPs** ↓
Baseline ($t = 32$)	82.7	16.2
Baseline ($t = 64$)	83.8	28.3
Baseline ($t = 96$)	84.5	40.4
Baseline ($t = 128$)	85.0	52.5
Baseline ($t = 192$)	85.3	76.7
Baseline ($t = 256$)	85.4	100.8
EE ($t = 32, \ \tau = 0.90$)	82.4	12.5
EE ($t = 64, \ \tau = 0.90$)	83.6	19.9
EE ($t = 128, \tau = 0.90$)	84.7	35.0
EE ($t = 192, \tau = 0.90$)	85.1	51.2
EE ($t = 256, \tau = 0.90$)	85.3	66.8
EE ($t = 256, \tau = 0.95$)	85.4	76.5
RL (tokens only)	83.9	32.0
RL (tokens, $\tau = 0.9$)	85.0	46.9
Optimal	93.6	32.5



(a) Truncated.

(b) Expanded.

Figure 8. **ImageNet-1K Evaluation.** Comparison of AdaPerceiver and state-of-the-art adaptive architectures, showing ImageNet-1K accuracy versus throughput. Fig. 8a is identical to Fig. 2 but with the addition of bi-directional attention data; Fig. 8b is an expanded version of Fig. 8a. *NB:* Throughput differences between the standard AdaPerceiver its bi-directional form are attributable to changes in the underlying attention implementation. AdaPerceiver uses FlexAttention [14], whereas AdaPerceiver (Bidir.) uses PyTorch's `scaled_dot_product_attention`.

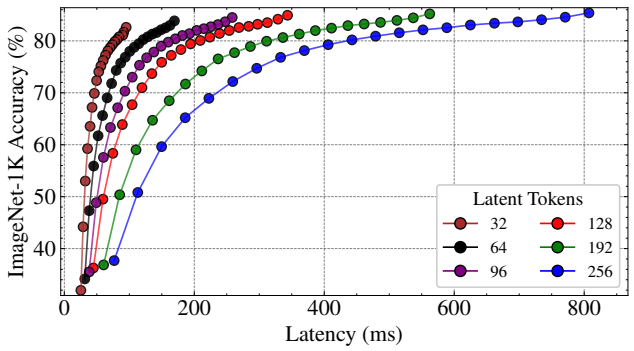


Figure 9. **ImageNet-1K Depth-Token Configuration Trade-offs.** Accuracy vs. Latency (ms) for AdaPerceiver with varying depths and numbers of latent tokens. Importantly, each configuration (point) *does not* require retraining. Depth improves accuracy monotonically while increasing latency monotonically. Reducing the number of latent tokens substantially decreases latency with minimal accuracy loss.

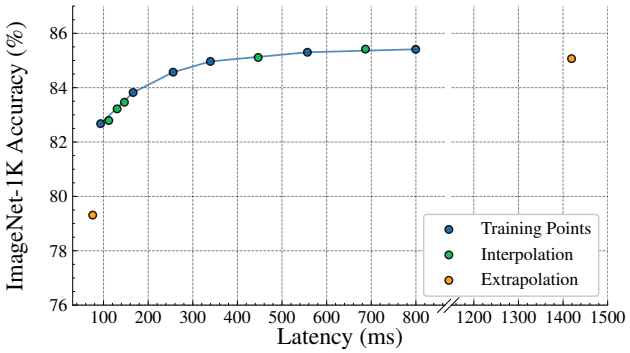


Figure 10. **Effect of Latent Token Interpolation and Extrapolation on ImageNet-1K Accuracy.** AdaPerceiver is trained with token granularities, $\mathcal{T} = \{32, 64, 96, 128, 192, 256\}$, however it able to interpolate (green points) within $\mathcal{T}$ and extrapolate outside $\mathcal{T}$ (yellow points). When interpolating, AdaPerceiver remains on the Pareto frontier, whereas extrapolation leads to some degradation in accuracy, with the largest drop occurring when extrapolating below the smallest trained token granularity. *N.B. The x-axis contains a break to ease visualization.*

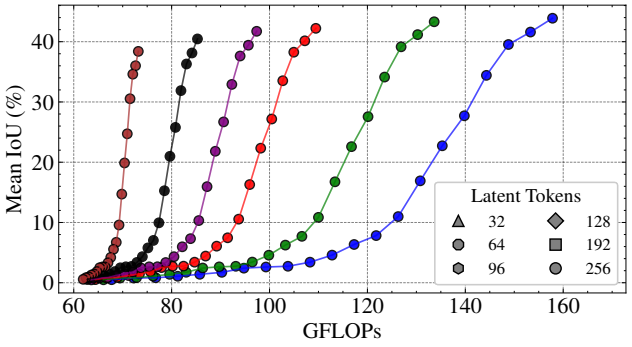


Figure 11. **ADE20K Depth-Token Configuration Tradeoffs.** mIoU vs. GFLOPs for AdaPerceiver with varying depths and numbers of latent tokens.

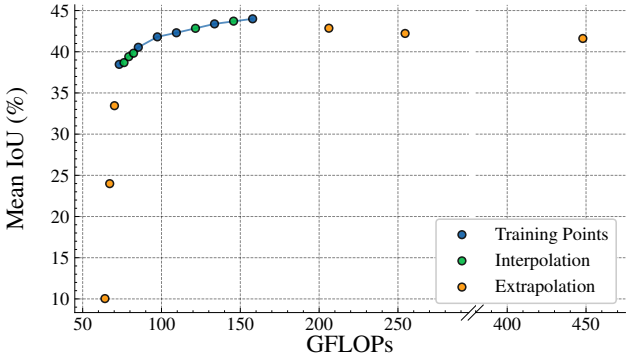


Figure 12. **Effect of Latent Token Interpolation and Extrapolation on ADE20K semantic segmentation.** Similar to Fig. 10, AdaPerceiver is able to interpolate (green points) between its training token granularities and to extrapolate (yellow points) beyond them. Performance degradation appears when extrapolating outside the trained range, with the largest drop occurring when extrapolating below the smallest trained token granularity. *N.B. The x-axis contains a break to ease visualization.*

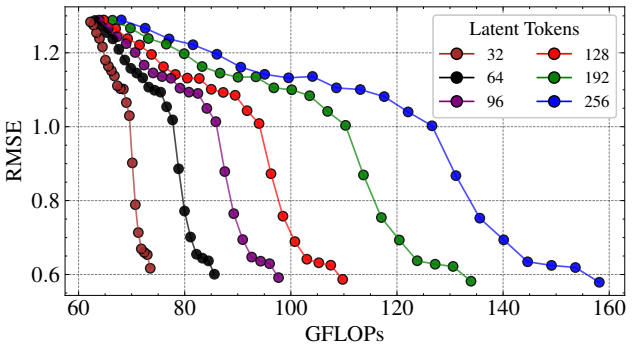


Figure 13. **NYUv2 Depth-Token Configuration Tradeoffs.** RMSE vs. GFLOPs for AdaPerceiver with varying depths and numbers of latent tokens.

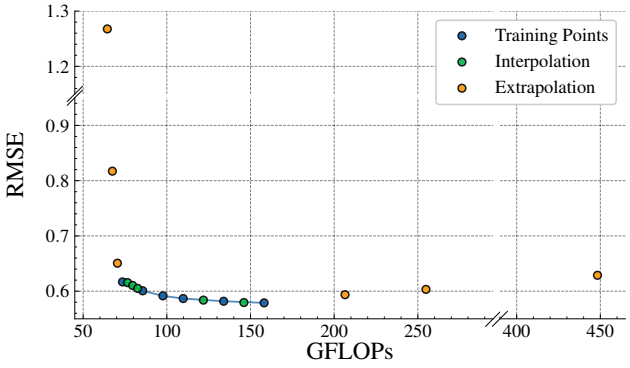


Figure 14. **Effect of Latent Token Interpolation and Extrapolation on NYUv2 depth estimation.** Similar to Fig. 10, AdaPerceiver is able to interpolate however it able to interpolate (green points) between its training token granularities and to extrapolate (yellow points) beyond them. Performance degradation appears when extrapolating outside the trained range, with the largest drop occurring when extrapolating below the smallest trained token granularity. *N.B. Both the x-axis and y-axis contain breaks to ease visualization.*

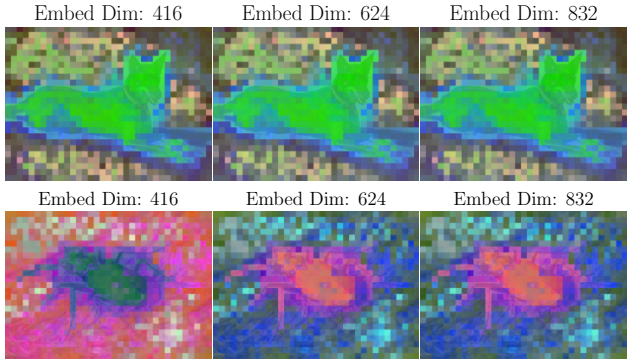


Figure 15. First three principal components of the patch features from AdaPerceiver when varying the number of embedding tokens (the tokens and depth fixed to their respective maximums). In the top sample, the principal components remain consistent across embedding dimension. In the bottom sample, the principal components from 416 → 624 width.

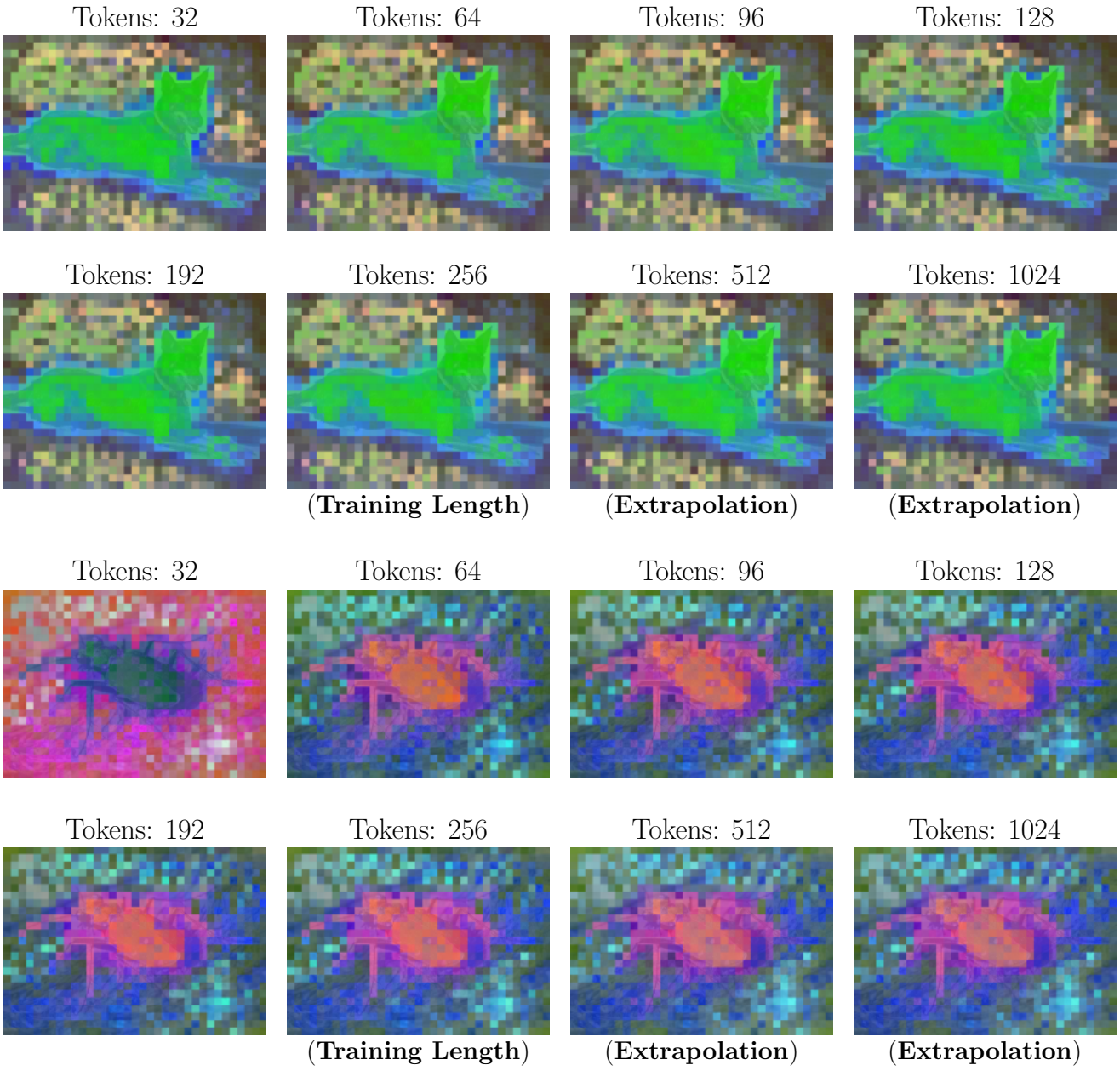


Figure 16. First three principal components of the patch features from AdaPerceiver when varying the number of latent tokens (the embedding dimension and depth fixed to their respective maximums). In the top sample, the principal components remain consistent across token counts (32 $\rightarrow$ 1024), indicating increasing the number of latent tokens does not change feature maps. In the bottom sample, the principal components shift initially from 32 $\rightarrow$ 64 tokens, after which they consistent up to 1024 tokens, suggesting that the model utilizes the additional capacity provided when shifting from 32 to 64 tokens, after which the representations converge. In both cases, the principal components **remain stable when extrapolating past the training length**.

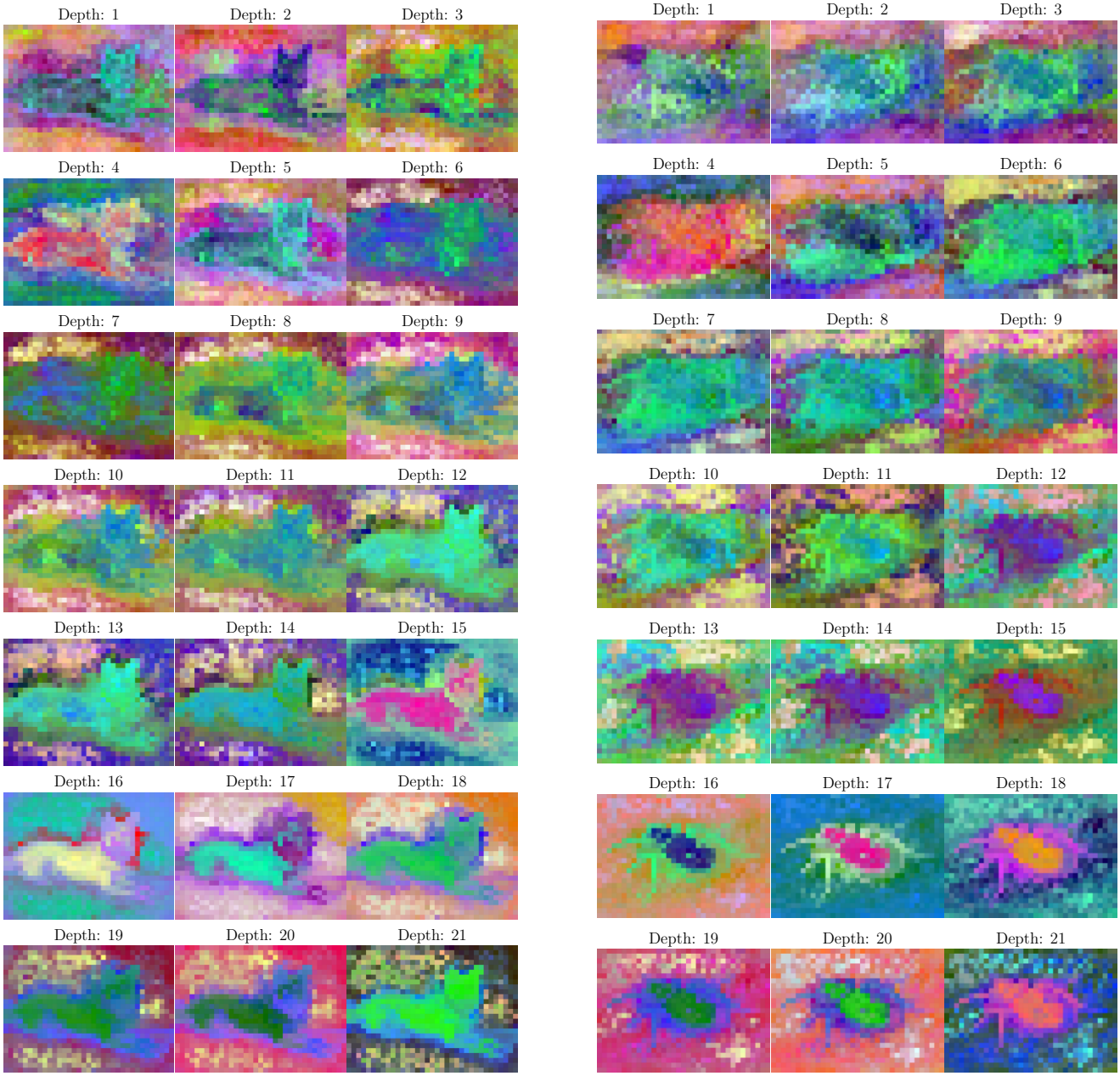


Figure 17. First three principal components of patch features across network depth (1–21) in AdaPerceiver. In both samples, discernible semantic features emerge with greater depth. However, the earliest layer at which discernible features emerge differ with sample.